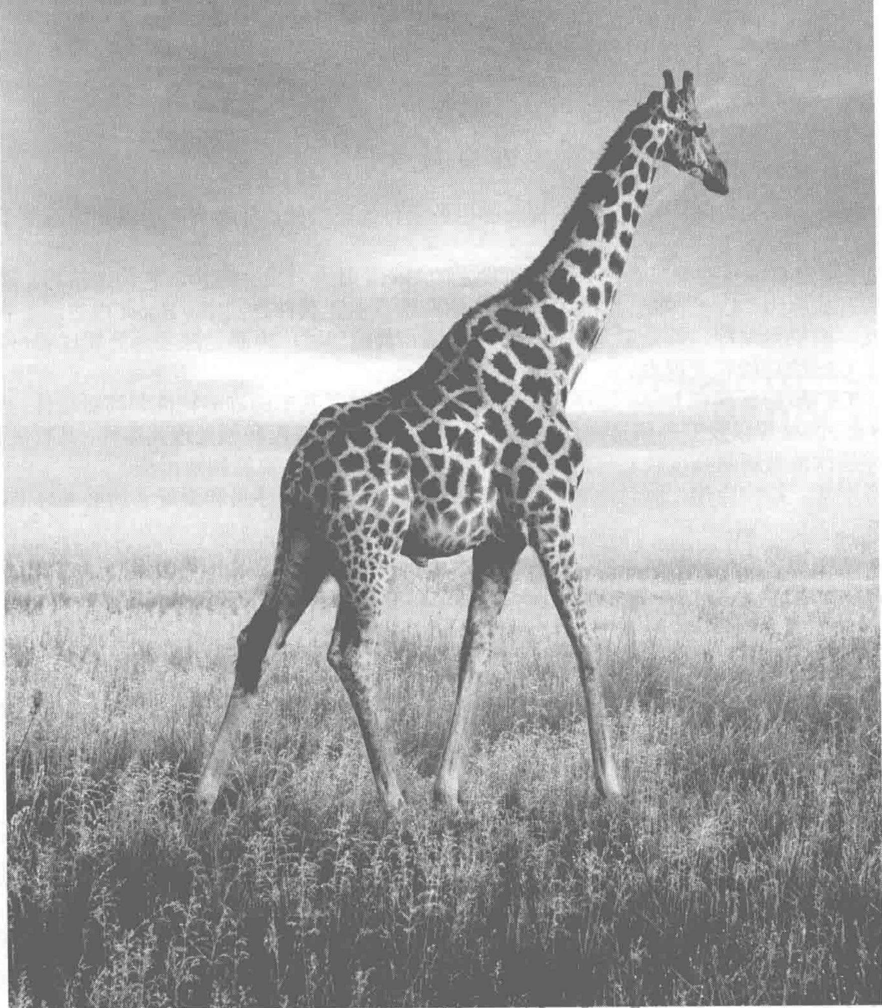
经典畅销书升级

深度解析C++11/14高级编程方法

C++11/14高级编程 Boost程序库探秘（第3版）

罗剑锋◎著

清華大學出版社



C++11/14高级编程
Boost
程序库探秘（第3版）

罗剑锋◎著

清華大学出版社
北京

内 容 简 介

C++的新标准（C++11/14）引入了许多强大易用的新特性新功能，从语言层面深刻地改变了 C++的开发范式。

Boost 程序库由 C++标准委员会部分成员所设立的 Boost 社区开发并维护，它构造精巧、跨平台、开源并且完全免费，被称为“C++‘准’标准库”，已广泛应用在实际软件开发中。Boost 内容涵盖智能指针、文本处理、并发、模板元编程、预处理元编程等许多领域，其范围之广内涵之深甚至要超过 C++11/14 标准，极大地增强了 C++的功能和表现力。

本书基于 C++最新标准和 Boost 程序库 1.60 版，深入探讨了其中的许多特性和高级组件，包括迭代器、函数对象、容器、流处理以及 C++语言中最复杂最具威力的模板元编程和预处理元编程，具有较强的实用性，可帮助读者深层次地理解掌握现代 C++的高级技术和 Boost 的内部实现机制及用法。

全书内容丰富、结构合理、概念清晰、讲解细致，是广大 C++程序员和爱好者的必备好书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

C++11/14 高级编程——Boost 程序库探秘/罗剑锋著. —3 版. —北京：清华大学出版社，2016

ISBN 978-7-302-44175-5

Ⅰ. ①C…　Ⅱ. ①罗…　Ⅲ. ①C 语言－程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2016）第 148590 号

责任编辑： 袁金敏
封面设计： 刘新新
责任校对： 徐俊伟
责任印制： 宋　林

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 清华大学印刷厂
经　　销： 全国新华书店
开　　本： 185mm×235mm　　**印　　张：** 31.75　　**字　　数：** 796 千字
版　　次： 2012 年 3 月第 1 版　　2016 年 9 月第 3 版　　**印　　次：** 2016 年 9 月第 1 次印刷
印　　数： 1～3500
定　　价： 79.00 元

产品编号：069426-01

目录

第 0 章　导读……1

0.1　关于本书……1
0.2　读者对象……2
0.3　C++标准……3
0.4　开发环境……3
0.5　代码风格……3
0.6　本书的结构……4
0.7　如何阅读本书……5
0.8　本书的源码……6

第 1 章　全新的 C++语言……7

1.1　概述……8
1.2　左值与右值……9
1.2.1　定义……9
1.2.2　右值引用……10
1.2.3　转移语义……11
1.2.4　完美转发……12
1.3　自动类型推导……13
1.3.1　auto……13
1.3.2　decltype……15
1.3.3　decltype(auto)……17
1.4　面向过程编程……17
1.4.1　空指针……17
1.4.2　初始化……18
1.4.3　新式 for 循环……19
1.4.4　新式函数声明……20
1.5　面向对象编程……21
1.5.1　default……21
1.5.2　delete……22
1.5.3　override……23
1.5.4　final……24
1.5.5　成员初始化……25
1.5.6　委托构造……26
1.6　泛型编程……27
1.6.1　类型别名……27
1.6.2　编译期常量……28
1.6.3　静态断言……29
1.6.4　可变参数模板……29
1.7　函数式编程……31
1.7.1　lambda 表达式……31
1.7.2　捕获外部变量……32
1.7.3　类型转换……34
1.7.4　泛型的 lambda 表达式……35
1.8　并发编程……35
1.9　面向安全编程……37
1.9.1　无异常保证……37
1.9.2　内联名字空间……37
1.9.3　强类型枚举……38
1.9.4　属性……39
1.10　更多特性……39
1.10.1　语言版本号……39
1.10.2　超长整型……40
1.10.3　原始字符串……40
1.10.4　自定义字面值……41
1.10.5　杂项……43

1.11 总结…………44

第 2 章 模板元编程简介…………45

2.1 概述…………45
2.2 语法元素…………46
2.3 元数据…………46
2.4 元函数…………47
2.5 元函数转发…………49
2.6 易用的工具宏…………50
2.7 应用示例…………51
2.8 总结…………52

第 3 章 类型特征萃取…………55

3.1 概述…………55
3.2 元数据类别…………56
3.2.1 基本类别…………56
3.2.2 复合类别…………58
3.3 元数据属性…………60
3.3.1 基本属性…………60
3.3.2 类相关属性…………61
3.3.3 操作符重载属性…………62
3.4 元数据关系…………62
3.5 元数据运算…………63
3.5.1 基本运算…………63
3.5.2 特殊运算…………65
3.6 解析函数元数据…………67
3.7 实现原理…………68
3.7.1 integral_constant…………68
3.7.2 is_integral…………69
3.8 应用示例…………70
3.8.1 conditional…………70
3.8.2 identity_type…………71
3.8.3 declval…………72
3.9 总结…………73

第 4 章 实用工具…………75

4.1 compressed_pair…………75
4.1.1 空类…………75
4.1.2 类摘要…………77
4.1.3 构造与赋值…………78
4.1.4 用法…………78
4.1.5 实现原理…………79
4.1.6 功能扩展…………80
4.2 checked_delete…………83
4.2.1 函数的用法…………84
4.2.2 函数对象的用法…………85
4.2.3 带检查的删除…………87
4.2.4 实现原理…………89
4.2.5 使用建议…………90
4.3 addressof…………90
4.3.1 用法…………91
4.3.2 实现原理…………92
4.3.3 使用建议…………93
4.4 base_from_member…………93
4.4.1 类摘要…………93
4.4.2 用法…………94
4.4.3 进一步的用法…………96
4.5 conversion…………98
4.5.1 标准转型操作符…………98
4.5.2 多态对象的转型…………99
4.5.3 polymorphic_downcast…………101
4.5.4 polymorphic_cast…………102
4.5.5 对引用转型…………103
4.6 numeric conversion…………104
4.6.1 bounds…………104
4.6.2 numeric_cast…………107
4.7 pointer…………108
4.7.1 get_pointer…………108
4.7.2 pointer_cast…………109
4.7.3 pointee…………110
4.7.4 indirect_reference…………111
4.7.5 pointer_to_other…………111
4.7.6 compare_pointees…………113

4.7.7 pointer_traits …… 114
4.8 总结 …… 115

第5章 迭代器 …… 117

5.1 概述 …… 117
5.1.1 迭代器模式 …… 117
5.1.2 标准迭代器 …… 118
5.1.3 新式迭代器 …… 119
5.1.4 标准迭代器工具 …… 120
5.1.5 迭代器与算法 …… 122
5.2 next_prior …… 122
5.2.1 函数声明 …… 123
5.2.2 用法 …… 124
5.2.3 C++11/14 标准 …… 125
5.3 iterator_traits …… 125
5.3.1 标准迭代器特征类 …… 126
5.3.2 类摘要 …… 127
5.3.3 用法 …… 127
5.4 iterator_facade …… 128
5.4.1 迭代器的核心操作 …… 128
5.4.2 类摘要 …… 129
5.4.3 用法 …… 131
5.5 iterator_adaptor …… 135
5.5.1 类摘要 …… 135
5.5.2 用法 …… 136
5.6 迭代器工具 …… 139
5.6.1 共享容器迭代器 …… 139
5.6.2 发生器迭代器 …… 141
5.6.3 逆向迭代器 …… 143
5.6.4 间接迭代器 …… 144
5.6.5 计数迭代器 …… 145
5.6.6 函数输入迭代器 …… 148
5.6.7 函数输出迭代器 …… 151
5.6.8 过滤迭代器 …… 153
5.6.9 转换迭代器 …… 155
5.6.10 索引迭代器 …… 157
5.6.11 组合迭代器 …… 159
5.7 总结 …… 161

第6章 区间 …… 163

6.1 概述 …… 163
6.2 特征元函数 …… 164
6.3 操作函数 …… 165
6.4 标准算法 …… 166
6.4.1 返回原区间的算法 …… 167
6.4.2 返回定制区间的算法 …… 168
6.5 迭代器区间类 …… 170
6.5.1 类摘要 …… 170
6.5.2 用法 …… 171
6.6 辅助工具 …… 173
6.6.1 sub_range …… 173
6.6.2 counting_range …… 174
6.6.3 istream_range …… 174
6.6.4 irange …… 175
6.6.5 combined_range …… 175
6.6.6 any_range …… 176
6.7 适配器 …… 178
6.7.1 适配器列表 …… 178
6.7.2 用法 …… 179
6.7.3 实现原理 …… 180
6.8 其他议题 …… 181
6.8.1 自定义区间类型 …… 181
6.7.2 连接区间 …… 182
6.9 总结 …… 182

第7章 函数对象 …… 185

7.1 hash …… 185
7.1.1 类摘要 …… 186
7.1.2 用法 …… 186
7.1.3 实现原理 …… 187
7.1.4 扩展 hash …… 188
7.2 mem_fn …… 191

7.2.1 工作原理 …… 191
7.2.2 用法 …… 192
7.2.3 其他议题 …… 193
7.3 factory …… 194
7.3.1 类摘要 …… 194
7.3.2 用法 …… 195
7.3.3 value_factory …… 197
7.4 总结 …… 197

第 8 章 指针容器 …… 199

8.1 概述 …… 199
8.1.1 入门示例 …… 200
8.1.2 指针容器的优缺点 …… 203
8.1.3 可克隆概念 …… 204
8.1.4 克隆分配器 …… 205
8.1.5 指针容器的分类 …… 206
8.2 指针容器的共通功能 …… 208
8.2.1 模板参数 …… 208
8.2.2 构造与赋值 …… 210
8.2.3 访问元素 …… 211
8.2.4 其他功能 …… 213
8.3 序列指针容器适配器 …… 214
8.3.1 配置元函数 …… 214
8.3.2 类摘要 …… 215
8.3.3 接口解说 …… 216
8.3.4 代码示例 …… 216
8.4 ptr_vector …… 217
8.4.1 类摘要 …… 218
8.4.2 用法 …… 219
8.5 空指针处理 …… 220
8.5.1 禁用空指针 …… 220
8.5.2 使用空指针 …… 220
8.5.3 空对象模式 …… 221
8.6 关联指针容器的共通功能 …… 223
8.6.1 类摘要 …… 223
8.6.2 接口解说 …… 224
8.7 集合指针容器适配器 …… 225
8.7.1 配置元函数 …… 225
8.7.2 ptr_set_adapter …… 226
8.8 ptr_set …… 227
8.8.1 类摘要 …… 227
8.8.2 用法 …… 228
8.9 ptr_unordered_set …… 228
8.9.1 类摘要 …… 228
8.9.2 用法 …… 229
8.10 映射指针容器适配器 …… 230
8.10.1 配置元函数 …… 230
8.10.2 ptr_map_adapter …… 231
8.11 ptr_map …… 233
8.11.1 类摘要 …… 233
8.11.2 用法 …… 234
8.12 ptr_unordered_map …… 234
8.12.1 类摘要 …… 235
8.12.2 用法 …… 235
8.13 使用 assign 库 …… 236
8.13.1 向容器添加元素 …… 236
8.13.2 初始化容器元素 …… 237
8.14 使用算法 …… 238
8.14.1 标准算法 …… 238
8.14.2 序列指针容器的算法 …… 242
8.14.3 关联指针容器的算法 …… 244
8.15 其他议题 …… 246
8.15.1 异常 …… 247
8.15.2 间接函数对象 …… 247
8.15.3 插入迭代器 …… 248
8.15.4 使用视图分配器 …… 248
8.15.5 可克隆性的再讨论 …… 249
8.16 总结 …… 250

第 9 章 侵入式容器 …… 251

9.1 概述 …… 251
9.1.1 手工实现链表 …… 252

9.1.2 intrusive 库介绍 253
9.2 入门示例 254
9.2.1 使用基类挂钩 254
9.2.2 使用成员挂钩 255
9.3 基本概念 257
9.3.1 节点 257
9.3.2 节点特征 258
9.3.3 节点算法 258
9.3.4 值特征 260
9.3.5 挂钩 260
9.3.6 选项 262
9.3.7 处置器 263
9.3.8 克隆 264
9.4 链表 264
9.4.1 节点和算法 265
9.4.2 基类挂钩 266
9.4.3 成员挂钩 267
9.4.4 类摘要 267
9.4.5 基本用法 269
9.4.6 特有用法 271
9.5 有序集合 275
9.5.1 节点和算法 275
9.5.2 基类挂钩 276
9.5.3 成员挂钩 277
9.5.4 set 类摘要 277
9.5.5 基本用法 279
9.5.6 特有用法 280
9.6 无序集合 282
9.6.1 节点和算法 282
9.6.2 基类挂钩 283
9.6.3 成员挂钩 284
9.6.4 类摘要 284
9.6.5 基本用法 286
9.6.6 unordered_set 的特有用法 288
9.7 其他议题 290
9.7.1 链接模式 290
9.7.2 同时使用多个挂钩 291
9.7.3 万能挂钩 293
9.8 总结 293

第 10 章 多索引容器 295

10.1 概述 295
10.2 入门示例 296
10.2.1 简单的例子 296
10.2.2 复杂的例子 297
10.2.3 更复杂的例子 299
10.3 基本概念 302
10.3.1 索引 302
10.3.2 索引说明 303
10.3.3 键提取器 304
10.3.4 索引说明列表 304
10.3.5 索引标签 305
10.3.6 多索引容器 305
10.4 键提取器 306
10.4.1 定义 306
10.4.2 identity 307
10.4.3 member 308
10.4.4 const_mem_fun 310
10.4.5 mem_fun 311
10.4.6 global_fun 312
10.4.7 自定义键提取器 313
10.5 序列索引 313
10.5.1 索引说明 313
10.5.2 类摘要 314
10.5.3 用法 315
10.6 随机访问索引 317
10.6.1 索引说明 317
10.6.2 类摘要 317
10.6.3 用法 318
10.7 有序索引 320
10.7.1 索引说明 320
10.7.2 类摘要 320

10.7.3 基本用法 …… 322
10.7.4 高级用法 …… 323
10.8 散列索引 …… 326
10.8.1 索引说明 …… 326
10.8.2 类摘要 …… 326
10.8.3 用法 …… 327
10.9 修改元素 …… 329
10.9.1 替换元素 …… 329
10.9.2 修改元素 …… 330
10.9.3 修改键 …… 332
10.10 多索引容器 …… 333
10.10.1 类摘要 …… 333
10.10.2 用法 …… 334
10.11 组合索引键 …… 337
10.11.1 类摘要 …… 337
10.11.2 用法 …… 338
10.11.3 辅助工具 …… 339
10.12 总结 …… 341

第 11 章 流处理 …… 343

11.1 概述 …… 343
11.1.1 标准库的流处理 …… 343
11.1.2 Boost 的流处理 …… 345
11.2 入门示例 …… 346
11.2.1 示例 1 …… 346
11.2.2 示例 2 …… 347
11.3 设备的特征 …… 349
11.3.1 设备的字符类型 …… 349
11.3.2 设备的模式 …… 349
11.3.3 设备的分类 …… 350
11.4 设备 …… 351
11.4.1 概述 …… 351
11.4.2 数组设备 …… 352
11.4.3 标准容器设备 …… 354
11.4.4 文件设备 …… 355
11.4.5 空设备 …… 357
11.5 过滤器 …… 358
11.5.1 概述 …… 358
11.5.2 设备链和管道 …… 359
11.5.3 计数过滤器 …… 361
11.5.4 换行过滤器 …… 362
11.5.5 正则表达式过滤器（Ⅰ） …… 364
11.5.6 正则表达式过滤器（Ⅱ） …… 366
11.5.7 压缩过滤器 …… 368
11.6 流 …… 369
11.6.1 基本流 …… 370
11.6.2 过滤流 …… 371
11.7 流处理函数 …… 373
11.8 定制设备 …… 374
11.8.1 定制源设备 …… 374
11.8.2 定制接收设备 …… 377
11.9 定制过滤器 …… 377
11.9.1 过滤器的实现原理 …… 378
11.9.2 aggregate_filter …… 379
11.9.3 basic_line_filter …… 380
11.9.4 手工打造过滤器 …… 381
11.10 组合设备 …… 385
11.10.1 combine …… 385
11.10.2 compose …… 386
11.10.3 invert …… 387
11.10.4 restrict …… 389
11.10.5 tee …… 390
11.11 其他议题 …… 391
11.11.1 对象的生存周期 …… 391
11.11.2 与迭代器的比较 …… 391
11.12 总结 …… 392

第 12 章 泛型编程 …… 395

12.1 enable_if …… 395
12.1.1 类摘要 …… 396

12.1.2 应用于模板函数 …… 397
12.1.3 应用于模板类 …… 398
12.1.4 对比 C++11 标准 …… 399
12.2 call_traits …… 399
12.2.1 类摘要 …… 399
12.2.2 用法 …… 400
12.2.3 实现原理 …… 402
12.3 concept_check …… 403
12.3.1 概述 …… 404
12.3.2 基本概念检查 …… 405
12.3.3 函数对象概念检查 …… 405
12.3.4 标准迭代器概念检查 …… 406
12.3.5 新式迭代器概念检查 …… 407
12.3.6 容器概念检查 …… 409
12.3.7 区间概念检查 …… 411
12.3.8 在函数声明中的概念检查 … 411
12.3.9 概念原型类 …… 413
12.4 总结 …… 414

第 13 章 模板元编程 …… 415

13.1 概述 …… 415
13.2 整数类型 …… 416
13.2.1 简介 …… 416
13.2.2 整数类型 …… 418
13.2.3 bool 类型 …… 419
13.2.4 基本运算 …… 419
13.3 流程控制 …… 421
13.3.1 if_和 if_c …… 421
13.3.2 eval_if 和 eval_if_c …… 422
13.4 容器 …… 423
13.4.1 简介 …… 424
13.4.2 vector …… 425
13.4.3 string …… 426
13.4.4 map …… 427
13.4.5 相关元函数 …… 428
13.5 迭代器 …… 429
13.5.1 简介 …… 429
13.5.2 相关元函数 …… 430
13.6 算法 …… 431
13.6.1 插入器 …… 431
13.6.2 查询算法 …… 432
13.6.3 变换算法 …… 433
13.6.4 运行时算法 …… 434
13.7 高级用法 …… 435
13.7.1 高阶元数据 …… 436
13.7.2 占位符 …… 437
13.7.3 bind 表达式 …… 437
13.7.4 lambda 表达式 …… 438
13.7.5 算法的高级应用 …… 439
13.8 断言 …… 441
13.8.1 基本断言 …… 442
13.8.2 否定断言 …… 442
13.8.3 关系断言 …… 443
13.8.4 定制消息的断言 …… 443
13.9 实例研究 …… 444
13.9.1 泛型编程版本 …… 444
13.9.2 元编程第 1 版 …… 446
13.9.3 元编程第 2 版 …… 449
13.10 总结 …… 450

第 14 章 预处理元编程 …… 453

14.1 概述 …… 453
14.1.1 元数据 …… 454
14.1.2 基本语法 …… 454
14.1.3 特殊符号 …… 456
14.1.4 特殊操作符 …… 456
14.2 整数运算 …… 457
14.3 常用元函数 …… 458
14.3.1 ASSERT …… 458
14.3.2 IF …… 459

14.3.3 ENUM ········· 459
14.3.4 REPEAT ········· 460
14.4 高级数据结构 ········· 461
14.5 总结 ········· 462

第 15 章 现代 C++开发浅谈 ········· 463

15.1 基本原则 ········· 463
15.2 内存管理 ········· 467
15.3 容器、迭代器和算法 ········· 468
15.4 其他 ········· 469
15.5 结束语 ········· 471

附录 A 推荐书目 ········· 473

附录 B Boost 程序库组件索引 ········· 475

附录 C Boost 程序库安装简介 ········· 485

第0章

导读

0.1 关于本书

现在是 21 世纪的第二个十年，计算机编程语言领域已不再是早期几家独大的局面，而是风起云涌、各领风骚，新的语言不断出现，同时也有老的语言逐渐衰落。但从一些权威统计机构的数据来看，30 多年前诞生的 C++语言依然有着强大的生命力，稳稳保持着热门语言前三名的位置，即使是后来者 Java、C#、Python 等也未能撼动它的王者地位。

C++能够获得这样的成就绝非运气，而是源于它自身的优异品质。它兼容“中级语言”C，具有良好的结构和绝佳的运行效率，可以开发系统级软件；它又开创了许多新的现代编程范式，支持面向对象、泛型等技术，具有足够的抽象度，灵活方便，可以开发各种大型复杂的应用软件。在众多的编程语言中 C++可称得上是“全能选手”，可上可下，大至企业级应用，小至嵌入式系统，几乎没有什么事情是 C++做不到的。

多年以来 C++保持了良好的稳定性，是很多程序员学习和使用的首选，“对程序员最小限制”的哲学让我们不必受语言的约束，可以在计算机世界里任意驰骋。但 C++也没有故步自封，近几年新发布的 C++11/14 标准不仅全面继承传统，更带来了许多新的特性和强大易用的功能，例如自动类型推导、统一的初始化语法、内建的 lambda 表达式、可变参数模板、线程支持等，为 C++注入了新的活力，很大程度上改变了 C++的开发风格。

而在 C++标准之外，与标准委员会有密切联系的 Boost 程序库更进一步扩展了 C++的功能。

Boost 程序库充分利用了 C++的自扩展性这个最“神奇”的特性，在基本语言完全不变的情况下深入挖掘了语言的潜力，把泛型编程等高级技术发挥到了极致，开发出了上百个功能强大的库，涉及内存管理、文本处理、容器与数据结构、文件系统、并发、模板元编程、预处理元编程等许多领域。由于 Boost 是以社区的形式开发维护的，不受标准委员会的限制，版本迭代更快，比“保守”的官方标准更加“激进”，所以也被称为 C++标准的“试验场”，更有着“C++ ‘准’标准库”的美誉，范围之广内涵之深甚至要超过 C++11/14 标准，极大地扩展了 C++的功能。

C++11/14 和 Boost 代表了现代 C++的最新发展成果，在国外早已经是大行其道，并且在国内也逐渐流行了起来。以作者个人所知，国内一些软件公司都或多或少地应用了 C++11/14 和 Boost 库，也将能否掌握 C++11/14 和 Boost 作为评判个人能力的一个因素。但因为 C++11/14 和 Boost 库的博大精深远非一般的语言和开源库可比，很多程序员也只能使用其中的少量功能特性，不能完全发挥 C++11/14 和 Boost 的真正实力，还有为数不少的人出于偏见仍然把 C++11/14 和 Boost 视作畏途。①

为了能够让更多的朋友结识越来越美妙的 C++，笔者编写了本书，深入探究 C++11/14 和 Boost 的高级编程技术，希望读者借助本书能够汲取更多有用的知识，提升自己的能力，达到“知其然更知其所以然”的境界。

0.2 读者对象

本书定位于中高级读者，假设读者已经对 C++的语言特性有较深层次的理解，并且具备了一定的 C++知识。

在 C++11/14 中，面向对象的编程范式已经不是主要技术，更多的是使用泛型编程，所以本书的读者除了应熟悉基本的面向对象技术外，还应该对模板和泛型、模板的特化/偏特化、静态多态等 C++高级技术有所了解。

STL 是第一个真正把泛型编程技术表现的淋漓尽致的作品，是 C++标准库的核心，现代的很多 C++库都深受其设计思想和结构的影响，所以熟悉 STL 将非常有利于 C++11/14 和 Boost 程序库的学习。作为一个现代的 C++程序员，应该对 STL 的容器、迭代器和算法这三个最重要的部

① C++11/14 和 Boost 库目前的情形与十多年前的 STL 非常相似：当年 STL 甫出，国外的程序员欢欣鼓舞，而国内的程序员却是担心效率、开发风格等诸多问题，畏首畏尾。时至今日，STL 已经成为了 C++程序员的基本素质，曾经的责难都已经烟消云散，相信假以不长的时日 C++11/14 和 Boost 也会获得广大程序员的认同。

分熟稔于心（推荐书目[2]对标准库有详尽的介绍，读者可参考）。

C++标准库和 Boost 程序库里的很多组件作者已经在推荐书目[3]中做了较详细的阐述，本书中将会直接使用这些组件而少做或不做解释。如果读者对 Boost 所知不多，建议先阅读推荐书目[3]然后再学习本书。

0.3 C++标准

C++的最新标准是 2014 年发布的 C++14，但目前很多编译器还未能完美支持，应用得也不够普及，故本书主要采用 C++11 标准。书中所称的“标准”“C++标准”通常指的是 C++11，而不是 C++98 或 C++14，更多的 C++标准知识请参考第 1 章。

涉及 C++标准文档时，本书会引用具体的章节号，形式是“C++11.章.节”，例如“C++11.20.2.3”表示标准的第 20 章第 20.2.3 节。

由于标准库已经成为了 C++的基础设施，故全书大部分代码均省略了标准库头文件和相应的“using namespace std”语句，请读者阅读时留意。不过个别情况下为了特别强调，偶尔会加上名字空间前缀，如 std::copy()。

0.4 开发环境

本书作者使用的开发环境是 Linux，具体如下所示。[①]

- 操作系统 : Ubuntu 14.04（Linux 3.13.0）。
- 编译器 : GCC 4.8.2（对 C++11 的支持较完善，但不支持 C++14）。
- Boost 程序库 : 1.60.0（2015 年 12 月发布）。

0.5 代码风格

遵循 C++标准和 Boost 的惯例，本书中的自定义类和函数均采用小写形式，模板参数列表

① 由于客观原因的限制，作者并未采用 Windows 和 VC 系列编译器，还请谅解。

统一使用更规范的 typename 而不是 class，递增操作使用效率更高的前置式（++i）。

为了改善代码的可读性，本书中的示例代码版式有一点小改进：某些需要强调的代码片段会用**粗体**或者*斜体*标明，读者可藉此迅速领会代码中的要点。

0.6 本书的结构

本书假设读者已经具备了一定的 C++和 Boost 知识，所以不再对 C++编译器和 Boost 库的安装和环境设置做介绍，而是直接切入主题讲解语言和库的使用。[①]

全书共 15 章，大致可以分为以下 8 个部分。

■ 第一部分，第 1 章：C++11/14 标准

主要介绍 C++11/14 的一些新特性，带领读者快速领略现代 C++开发风格和范式，以实用为目的，不求面面俱到和精确描述。

■ 第二部分，第 4～7 章：实用工具

本部分从 C++的深层次概念入手，介绍各种实用工具，涉及类型转换、对象的创建和删除、指针、迭代器、函数对象等诸多内容，学习它们可以对 C++的底层语言细节有更深入的了解，有助于构建更稳固健壮的程序。

■ 第三部分，第 8～10 章：新式容器

本部分较详细地阐述了三类新式容器：指针容器、侵入式容器和多索引容器，它们从不同的方向扩展了标准容器，功能强大内容庞杂，用起来有许多额外的考虑，读者会发现它们能够应用在许多特殊的场景。

■ 第四部分，第 11 章：流式处理

本部分论述了 C++提供的流处理功能。流处理是 C++诞生之初就有的功能，但多年来一直未被重视，boost.iostreams 在标准库的基础上构造了功能完备富有弹性的流式处理框架，可以灵活处理各种数据。

① Boost 程序库中的大部分组件都是以头文件的方式实现的，不需要编译，直接包含头文件即可。少数库需要编译才能使用，Boost 提供类似 make 的 b2 工具来完成库的编译和安装，可以参考附录 C 或者推荐书目[3]。

■ 第五部分，第 2～3 章、第 12～13 章：泛型编程和模板元编程

本部分介绍 C++语言中最高级的技术：泛型编程和模板元编程，包括 type_traits、concept_check 和 mpl 等库，需要读者对 C++的泛型技术具有较深刻的理解。模板元编程虽然已经出现了很多年，但在国内仍然算是一个新的领域，相关的实践经验也不是很多，作者在这里也仅仅是做一个较为粗略的介绍。

■ 第六部分，第 14 章：预处理元编程

本部分介绍预处理元编程，它位于 C++语言体系之外，工作在编译之前的预处理阶段，使用宏的方式任意操作程序代码，不宜乱用，但在有些时候会很有用。

■ 第七部分，第 15 章：开发经验浅谈

本部分基于作者多年使用 C++的经历以条款的形式给出了一些开发经验，不一定完全正确，权做抛砖引玉之举，希望能够给读者起到一点借鉴的作用。

■ 第八部分，附录

书末的附录首先收录了作者认为值得阅读的 C++经典作品，然后是 Boost1.60 版全部组件的索引，最后是 Boost 库的简易安装手册。

0.7 如何阅读本书

对于所有读者来说，作者推荐首先阅读第 1 章和第 2 章，它们详细讲解了 C++11/14 的新特性和模板元编程，是全书的基础。第 1 章简要介绍了 C++新标准，学习难度不高，第 2 章虽然内容较深，但由于模板元编程在本书中的重要地位而被提到了全书的最开始，随后的许多章节都会用到其中的概念，熟悉模板元编程的基本概念对于理解本书是非常有帮助的。

接下来读者可以随个人意愿，或者按照书籍的物理顺序循序渐进逐页阅读，或者查阅目录，然后跳到感兴趣的章节。建议手里再拿一支笔，可以随时圈阅或标注，记录阅读时的领悟、疑问和心得，让本书成为您的学习笔记。

书中包含了大量作为示例的 C++源代码，都附加了详细的注释，希望读者能够耐心仔细阅读。另外本书还编制了交叉索引，阅读时涉及其他章节的内容都会指明具体位置，便于读者快速参考。

由于 C++和 Boost 库里很多组件的原理、用法较复杂，读者可结合附录的推荐书目阅读，最好再打开自己最熟悉的编辑器或集成开发环境，直接查看源代码加深理解。

0.8 本书的源码

为方便读者利用本书学习研究 C++11/14 和 Boost 程序库，作者在 GitHub 网站上发布了本书内所有示例程序的源码，地址是：

```
https://github.com/chronolaw/professional_boost.git
```

第1章

全新的 C++语言

C++诞生至今已经三十余年，是一种功能强大到几乎可被称为“万能”的语言，支持许多种风格迥异的编程范式。

20 世纪 80 年代到 90 年代是 C++的“少年期”，它不但很好地继承了传统的面向过程编程范式，还充分实践了面向对象的编程范式，运行效率与开发效率兼顾，可以说是当时最好的编程语言。

20 世纪 90 年代末 C++98 标准的发布是一个标志性事件，它标志着 C++成为正式的国际标准，进入了“青年期”。这一时期的 C++有了标准的“保驾护航”，不仅更好地专注于传统编程范式领域，而且还发展出了泛型编程、模板元编程、函数式编程等新的范式，给予了程序员更大的发挥空间。

遗憾的是在进入 21 世纪之后，C++标准暂时停滞了前进的脚步。在这段时间里，许多新的编程理念和语言出现了，而 C++因为未能及时更新而显得有些“落伍”，令不少支持者深感失望，这是 C++的“烦恼期”。

21 世纪的第二个十年里，C++11/14 标准终于“千呼万唤始出来”，基于现代编程理念对 C++进行了大幅度的改造，从语言到标准库都面目一新。经过此番磨砺的 C++焕发了“第二春”，变得更成熟稳重，正当壮年。现在，使用 C++来开发程序，限制程序员能力的大概只有自己的想象力了。

C++11/14 标准拥有大量的新特性，C++之父 Bjarne Stroustrup 是这样评价的：“一个全新的语言”（feels like a new language）。对于任何人——无论是初学者还是高级专家，

C++11/14 都值得去静下心来认真学习体会。[①]

1.1 概述

经过十余年的努力工作，C++标准委员会在 2011 年通过了 C++标准的第二个重要版本：ISO/IEC 15582:2011，也就是俗称的“C++11”。它不仅修复了上个标准的很多缺陷，更增加了十余个新关键字和百余个新特性，核心语言特性和标准库都得到了重大提升。[②]

2014 年，C++14 标准发布，用于替代 C++11 标准。但它对 C++11 的改动并不多，近似于 C++11 的“修订版”，所以这两个标准通常也合称为“C++11/14”。

C++11/14 庞大而复杂，标准文档厚达 1300 多页，但为此而工作的标准委员会的总体设计目标却很简单，只有以下两个。

- 更适合系统编程和构建程序库（Make C++ a better language for systems programming and library building）。
- 更容易学习（Make C++ easier to teach and learn）。

设定这两个总体目标依据的是 C++多年来已有的实际应用经验。

C++最初的目的就是成为“更好的 C”，因此新的标准首先要对基本的底层编程进行强化，能够反映当前计算机软硬件系统的最新发展和变化（例如多线程）。另一方面，C++对多编程范式的支持增加了语言的复杂度，通常一个范式就相当于其他的一门编程语言，学习难度高，不仅新手难以入门，即使是曾经的“专家”也很难快速掌握。为了能够让 C++吸引更多的用户，避免“曲高和寡”的局面，降低学习的门槛也是新标准的重中之重。

基于以上的两个基本目标，C++标准委员会又制订了一些更具体的设计目标。

- 保持稳定性和兼容性。
- 尽量使用库而不是扩展语言来增加新特性。
- 对初级用户和高级用户都提供良好的支持。

---

① 本章部分内容参考 Bjarne Stroustrup 的 C++11 FAQ。

② C++11/14 也废弃了一些之前的特性（C++11.Annex D），例如 register、export、auto_ptr、bind1st/bind2nd 等，本章不再对它们多做论述。

- 增强类型的安全性。
- 增强直接操作硬件时的效率和功能。

C++标准委员会的工作是卓有成效的，新的 C++11/14 标准基本实现了这些目标。虽然引入了大量的新特性，但新标准依然较好地保持了与之前版本的兼容性，大部分现存的代码不需要改动就可以直接使用 C++11/14 标准来编译。当然，这样做并不能感受到新标准的好处，我们必须要主动地使用新特性才能够改进我们的代码质量。

1.2 左值与右值

早期的 C++里只有“左值”，C++11/14 标准引入了新的“右值”，它们是现代 C++中非常重要的概念，贯穿整个 C++11/14 标准，本节将简要阐述这两个概念及其应用。

1.2.1 定义

C++11/14 标准描述了左值与右值的含义（C++11.3.10，但比较晦涩和“学术气”），简略摘要如下所述。

- 所有表达式的结果不是左值就是右值。
- 左值（lvalue）是一个函数或者对象实例。
- 失效值（xvalue，eXpiring value）是生命期即将结束的对象。
- 广义左值（glvalue，generalized lvalue）包括左值和失效值。
- 右值（rvalue）包括失效值、临时对象、以及不关联对象的值（如字面值）。
- 纯右值（prvalue）是非失效值的那些右值。

左值和右值的分类如图 1-1 所示。

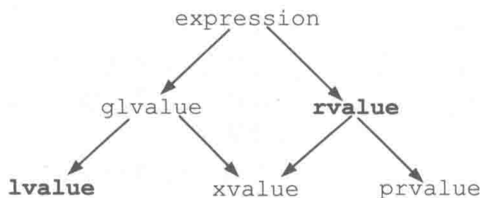


图 1-1 左值和右值的分类

C++标准的定义较难理解，本书在这里做一个简单（但不很精确）的解释：左值是一个可以用来存储数据的变量，有实际的内存地址（即变量名），表达式结束后依然存在，（历史上）它因在赋值操作符左边而得名；而右值（更准确地应该叫“非左值”）是一个“匿名”的“临时”变量，它在表达式结束时生命周期终止，不能存放数据，可以被修改，也可以不被修改（被 const 修饰）。

根据这个解释，鉴别左值和右值最简单的方法是：左值可以用取地址操作符“&”获取地址，而右值则无法使用“&”（会发生编译错误）。例如：

```
int    x = 0;           //对象实例，有名，x 是左值
int*   p = &++x;        //可以取地址，++x 是左值
++x = 10;               //前置++返回的是左值，可以赋值
p = &x++;               //后置++操作返回一个临时对象，不能取地址或赋值，是右值，编译错误
```

因为右值是“临时”的，生命周期即将结束，之后无人会关心它的值，所以我们可以把它的所有内容“偷”（说的好听一些就是转移）到其他对象中，从而完全消除昂贵的拷贝代价。

1.2.2 右值引用

有了右值的概念，右值引用也就应运而生。C++11/14 标准使用“T&&”的形式表示右值引用，而原来的“T&”则表示左值引用，两者可以简称为右引用和左引用，分别引用右值对象和左值对象。[①]

对一个对象使用右值引用，意味着显式地标记这个对象是右值，可以被转移来优化。同时也相当于为它添加了一个“临时的名字”，生命周期得到了延长，不会在表达式结束时消失，而是与右值引用绑定在了一起。

左引用和右引用也可以使用 const 进行修饰。“const T&”是一个“万能引用”，可以引用任何对象（但增加了常量性）。而“const T&&”虽然语法上是正确的，但因为右值对象是“临时”的，即将消失，对它增加常量性会使它无法修改，也就无法转移，没有实际的意义。

下面的代码示范了这四种引用。

```
int&            r1 = ++x;           //左值引用
int&&           r2 = x++;           //右值引用，引用了自增后的临时对象 xvalue
const int&      r3 = x++;           //常量左值引用，也可以引用右值
const int&&     r4 = x++;           //常量右值引用，无实际意义
cout << r2 << endl;                 //右引用延长生命期，右值对象在表达式结束后仍然存在
```

① 在 C++98 标准中只有一种引用，即只左值引用 T&，不能引用右值。

在泛型编程中需要操作未知类型，C++11/14 对此做出了新的规定——引用折叠（reference collapsing，C++11.8.3.2）：对引用类型 TR(可能是左引用或右引用)再进行左引用操作是 T&，右引用操作则不变化（仍然是 TR），因此，函数参数如果使用 T&&的形式将总保持原类型不变。这条规则是 C++11/14 实现转移语义和完美转发的基础。[①]

1.2.3 转移语义

因为右值对象可以被转移进而优化代码，所以 C++11/14 标准在头文件<utility>里专门定义了便捷函数 std::move()来“转移”对象（C++11.20.2.3），它是一个模板函数，声明是：[②]

```
template <class T>
typename remove_reference<T>::type&& move(T&& t) noexcept;
```

move()函数其实并没有做任何“转移”操作，只是把一个对象明确地转换为“匿名”的右值引用，也就是说该对象确认是右值对象，可以被安全地转移，相当于：

```
static_cast<T&&>(t);                    //转型为右值引用
```

从这点来看，move()函数更像是一种语法层面的强制标记。

C++11/14 标准为 class 新增加了参数类型为“T&&”的转移构造函数和转移赋值函数，只要类实现了这两个特殊函数就能够利用右值对象“零成本”构造，这就是转移语义。它是现代 C++ 语言里优化性能的重要手段，只要对象被 move()标记为右值引用，那么就可以毫无损失地转移资源，再也不需要担心“深拷贝”的成本。

下面的代码定义了一个简单的类，实现转移构造函数和转移赋值函数，可以使用转移语义。

```
class moveable                          //实现转移构造函数和转移赋值函数的类
{
private:
    int x;                              //成员变量
public:
    moveable () {}                      //缺省构造函数
    moveable(moveable&& other)          //转移构造函数
    {    std::swap(x, other.x); }       //需要使用某种方式“窃取”右值对象的内容
```

① 在 C++98 中“引用的引用”（T&&）会导致编译错误。
② move()函数使用了模板元编程技术，返回值移除了 T 的引用修饰，可参考第 2 章和第 3 章。

```
    moveable& operator=(moveable&& other)  //转移赋值函数
    {
        std::swap(x, other.x);             //需要使用某种方式“窃取”右值对象的内容
        return *this;
    }
public:
    static moveable create()               //工厂函数创建对象
    {
        moveable obj;                      //函数栈上创建对象
        return obj;                        //返回临时对象，即右值，会引发转移语义
    }
};
```

moveable 类里有一个工厂函数，它直接返回函数内部的局部变量 obj，在函数返回时是一个临时对象，也就是右值（无须再使用 std::move()），而 moveable 类也定义了转移构造函数，所以可以直接使用临时对象的值来创建对象，避免了拷贝的代价。示范代码如下：

```
moveable m1;                               //缺省构造函数创建对象
moveable m2( std::move(m1) );              //调用转移构造函数，m1 被转移
moveable m3 = moveable::create();          //调用转移赋值函数
```

C++标准库里的 string、vector、deque 等组件都实现了转移构造函数和转移赋值函数，可以利用转移语义优化，所以现在在函数里返回一个大容器对象是非常高效的，我们可以放心使用。[①]

这些标准容器（std::array 除外）还特别增加了 emplace()系列函数，可以使用转移语义直接插入元素，进一步提高了运行性能，例如：

```
vector<complex<double>> v;                 //标准序列容器
v.emplace_back(3,4);                       //直接使用右值插入元素，无须构造再拷贝

map<string, string> m;                     //标准映射容器
m.emplace("metroid", "prime");             //直接使用右值插入元素，无须构造再拷贝
```

1.2.4 完美转发

标准头文件<utility>里还有一个函数 std::forward()，用于在泛型编程时实现“完美

---

① 实际上大多数编译器都早已经对函数返回值做了特别的优化（RVO，Return Value Optimization），所以转移语义的效果有可能不会直接体现。

转发”，可以把函数的参数原封不动地转发给其他函数，声明是（有简化）：[①]

```
template <class T> T&& forward(T& t) noexcept;
template <class T> T&& forward(T&& t) noexcept;
```

forward()在使用时必须指定模板参数，它应用了 C++11/14 标准的引用折叠规则，对于任何类型的参数（值类型、引用类型、常量引用类型等）都可以原样转发。

作为示范，下面的代码定义了两个参数不同的重载函数，并在模板函数里转发参数以调用它们。

```
void check(int&)                        //参数是左值引用
{    cout << "lvalue" << endl;}

void check(int&&)                       //参数是右值引用
{    cout << "rvalue" << endl;}

template<typename T>
void print(T&& v)                       //注意，参数是右值引用，会保持类型不变
{    check(std::forward<T>(v));}        //完美转发，依据函数参数类型调用不同的函数
```

使用左值和右值分别调用 print()函数：

```
int x = 10;                             //一个左值对象
print(x);                               //传递左值引用，输出 lvalue
print(std::move(x));                    //传递右值引用，输出 rvalue
```

注意在使用完美转发时函数的参数必须声明为 T&&，只有这样类型才能够保持原状。

1.3 自动类型推导

C++11/14 标准增加了两个新关键字：auto 和 decltype。它们可以推导出表达式的类型信息，个人认为是新 C++标准中最最重要的特性——千万不要小看这两个关键字，它们的类型推导能力不仅可以极大地简化代码，更赋予了程序员原来只属于编译器的无上权力。

1.3.1 auto

C++是一种强类型的静态语言，任何变量、表达式都要有明确的类型，例如：

① Boost 程序库也提供功能类似的库 boost.forward，可用于不支持 C++11/14 标准的编译器。

```
long          x = 0L;                                   //声明x为long类型
const char*   s = "zelda";                              //声明s为字符指针类型
```

对于简单的变量，可以很容易写出它的类型，但由于类、名字空间、模板等技术的广泛应用，变量的类型逐渐变得越来越复杂，有的类型名字很长甚至很难写出正确的类型：

```
map<string, string>::iterator iter = m.begin();  //标准容器的迭代器，类型较复杂
??? f = bind1st(std::less<int>(), 2);            //函数绑定器，类型很难写出
```

事实上，编译器是知道这些表达式的类型的，而且也必须知道表达式的类型（否则就无法判断程序员写的类型是否正确），但在 C++11/14 之前，这些类型信息却都隐藏在编译器内部，有种“猜猜我是谁”“就不告诉你”的感觉。

这种令人尴尬的局面如今终于有了改善，C++11/14 标准重新定义了 auto 关键字的语义（C++11.7.1.6.4），可将其用在赋值表达式里声明变量，并在编译期自动推导出表达式的类型，相当于向编译器“索取”了类型信息。①

使用 auto 能够简化代码，让编译器去做“不必要”的类型声明（而且做得更好），之前的代码可以改写成：

```
auto  x = 0L;                               //推导为long类型
auto  s = "zelda";                          //推导为字符指针类型
auto  iter = m.begin();                     //推导为标准容器的迭代器类型
auto  f = bind1st(std::less<int>(), 2);     //推导为正确的函数对象类型，很难直接写出
```

auto 的用法相当简单，但也有以下几个需要注意的地方。

- auto 只能用于赋值语句里的类型推导，不能直接声明变量（因为无表达式供推导）。
- auto 总是推断出值类型（非引用）。
- auto 允许使用“const/volatile/&/*”等修饰，从而得到新的类型。
- auto&&总是推断出引用类型。

下面的代码示范了 auto 的更多用法：

```
int               x = 0;                    //整数类型int
const long        y = 100;                  //整数类型const long
volatile string   s("one punch");           //字符串类型volatile
```

① 实际上，auto 的推导类型能力是一个“古老的新特性”，早在 C++初期就已经有实验性质的实现，但后来考虑到与 C 语言的兼容问题没有正式发布。

```
auto          a1 = ++x;                //值类型 int
auto&         a2 = x;                  //引用类型 int&
auto          a3 = y*y;                //值类型 long
auto&         a4 = y;                  //引用类型 const long&
auto          a5 = std::move(y);       //值类型 long，右引用被忽略
auto&&        a6 = std::move(y);       //引用类型 const long&&
const auto    a7 = x+x;                //常量值类型 const int
auto*         a8 = &y;                 //const long*，auto 本身推导为值类型
auto&&        a9 = s;                  //引用类型 volatile string&
auto          a10;                     //不是赋值初始化，无法推导，编译错误
```

在 C++14 里 auto 又扩充了能力，还可以用在函数的返回值声明处，自动推导函数的返回值类型（读者还可参考 1.4.4 节进一步了解），例如：

```
auto func(int x)                       //C++14 标准，C++11 不能使用，请注意
{    return x*x;}                      //推导返回值类型为 int
```

在现代 C++编程中应当尽量多使用 auto，它不会有任何的效率损失，而且带来了更好的安全性和可读性。

1.3.2 decltype

auto 关键字能够在赋值语句里推导类型，但这只是 C++语言里一种很少见的应用场景，要想在任意的场合下都能够得到表达式的类型需要使用另一个关键字：decltype（C++11.7.1.6.2）。[①]

decltype 在技术上和用法上与 sizeof（它也是个关键字）非常相似，因为都需要编译器在编译期计算类型，但 sizeof 返回的是整数，而 decltype 返回的是类型。

decltype 的形式很像是函数调用：

```
decltype(expression)                   //获取表达式的类型，编译期计算
```

decltype 可以像 auto 一样用在赋值语句，但可以根据表达式的结果类别和表达式的性质推断出引用或非引用类型，能够更精确地控制类型：

```
int             x = 0;                 //整数类型 int
const long      y = 100;               //整数类型 const long

decltype(x)     d1 = x;                //类型是 int
```

① decltype 这个关键字比较长，本书作者的读法是“declare type”。

```
decltype(x)&       d2 = x;                    //类型是 int&
decltype(&x)       d3 = &x;                   //类型是 int*
decltype(x+y)      d4 = x + y;                //类型是 long
decltype(y)&       d5 = y;                    //类型是 const long&
```

除了赋值语句，decltype 还可以用在变量声明、类型定义、函数参数列表、模板参数列表等任意的地方，因为它实际上就是一个编译期的类型名（只是通过表达式计算得到的），只要我们使用了 decltype，编译器就会无条件地给予我们正确无误的类型。

decltype 的用途非常广泛，更多的用法示范如下：

```
decltype(std::less<int>()) functor;       //声明一个函数对象，注意不是赋值语句

decltype(0.0f) func(decltype(0L) x)       //用于函数返回值和参数声明
{    return x*x;}
typedef decltype(func)* func_ptr;         //简单地定义函数指针类型

vector<int> v;                            //标准容器
decltype(v)::iterator iter;               //计算 v 的类型，再取其迭代器类型

template<typename T> class demo {};       //一个简单的模板类
demo<decltype(v)> obj;                    //在模板参数列表里使用 decltype
```

decltype 的用法和 auto 一样简单，但同样有语法细节需要特别注意。

- decltype(*e*)的形式获得表达式计算结果的值类型。
- decltype((*e*))的形式获得表达式计算结果的引用类型，类似 auto&&的效果。

这两点涉及比较复杂的推导规则细节，本书不做详细的阐述，下面仅用示例代码来简略示范这两者的区别：

```
int                     x = 0;            //整数类型 int
const volatile int      y = 0;            //整数类型 const volatile int

decltype(x)        d1 = x;                //值类型 int
decltype((x))      d2 = x;                //引用类型 int&
decltype(y)        d3 = y;                //值类型 const volatile int
decltype((y))      d4 = y;                //引用类型 const volatile int&
```

接下来的代码更好地示范了 decltype(e)与 decltype((e))的区别：

```
struct demo {int x = 0;};                 //一个简单的类
volatile auto *p = new demo;              //注意是 volatile 指针
```

```
decltype(p->x)    d5 = 42;                  //推导为值类型 int
decltype((p->x))  d6 = p->x;                //推导为引用类型 volatile int&
decltype(p->x)&   d7 = p->x;                //错误，int&类型不匹配
```

这里我们声明了一个 volatile 的指针 p，decltype(p->x)只能得到 p->x 的值类型 int，失去了 volatile 修饰，而 decltype((p->x))却可以完整地得到 p->x 的真正引用类型。

1.3.3 decltype(auto)

auto 和 decltype 这两个关键字都可以推导类型，但用法有所差异。auto 的使用更加简单方便，但用途有限，只能用在赋值语句里；decltype 用途更广，可以推导任意表达式的类型，但使用时必须在括号内写全表达式，用法略微不便。

C++14 标准增加了一种新的语法，允许把这两者结合起来，也就是“decltype(auto)”，使用 decltype 的语义推导类型，但用的却是 auto 语法形式，例如：

```
decltype(auto)    x = 6;              //整数类型 int，仅 C++14
decltype(auto)    y = 7L;             //整数类型 long，仅 C++14
decltype(auto)    z = x+y;            //整数类型 long，仅 C++14
```

1.4 面向过程编程

C++语言继承了 C 语言的传统，支持最基本的面向过程编程。在这个编程范式里 C++11/14 的变化并不多，但增加的新特性却可以很好地改进程序甚至改变我们的编程思维。

1.4.1 空指针

一直以来，在 C/C++语言里空指针都使用宏 NULL 来表示，它的定义通常是“0”，即：[1]

```
#define NULL  0                          //空指针宏 NULL 的定义
```

但 NULL 存在严重的缺陷，它实际上是一个整数，而不是真正的指针，所以在有的时候会造成语义混淆（例如重载函数的参数）。

C++11/14 增加了新的关键字“nullptr”（C++11.2.14.7），彻底解决了这个隐患，同

① 在 C 语言中 NULL 通常定义为“(void*)0”，因为 void*可以隐式转换为任何指针类型，但在 C++里这种转换是不允许的。

时也增强了安全性。[①]

nullptr 明确地表示空指针概念，可以完全替代 NULL，它可以隐式转化为任意类型的指针，也可以与指针进行比较运算，但决不能转化为非指针的其他类型：

```
int*         p1 = nullptr;                  //初始化 int 指针为空指针
vector<int>* p2 = nullptr;                  //初始化 vector 指针为空指针

assert(!p1 && !p2);                         //两个指针都是空指针
assert(p1 == nullptr);                      //执行指针比较运算
assert(10 >= nullptr);                      //编译错误，不能与整数等类型进行运算
```

nullptr 与 NULL 有重要区别，它是强类型的，但类型不是 int 或者 void*，而是一个专用的类型 nullptr_t，其定义利用了关键字 decltype 的能力：

```
typedef decltype(nullptr) nullptr_t;        //nullptr_t 的定义
```

还需要注意，nullptr 并不是指针，而是一个类型为 nullptr_t 的编译期常量对象实例，只是其行为很像指针，所以我们也可以使用 nullptr_t 任意定义与 nullptr 等价的空指针常量，例如：

```
nullptr_t nil;                              //定义一个新的空指针常量 nil

double* p3 = nil;                           //使用 nil 初始化空指针
assert(nil == nullptr);                     //nil 与 nullptr 完全等价
```

在我们编写代码时应当总使用 nullptr 来初始化或比较指针，尽量避免使用 NULL 宏。[②]

1.4.2 初始化

在 C++中初始化是一个基本的操作，但 C++98 标准并没有非常明确的定义，而且初始化的语法也不一致。C++11/14 标准对这个问题给出了完整的解决方案，统一使用花括号“{}”初始化变量，称为“列表初始化”（list initialization，C++11.8.5），例如：

```
int             x{};                        //x 使用缺省初始化，值为 0
double          y{2.718};                   //初始化为 2.718
string          s{"venom snake"};           //标准字符串初始化
complex<double> c{1,1};                     //初始化复数对象
```

① nullptr 通常可以直接读作“null pointer”，如同它的字面含义。
② 为了保持兼容，目前 C++仍然允许使用 NULL 和 0 表示空指针，但不推荐。

这种方式不仅能够初始化简单变量，也能够初始化数组和容器：

```
int              a[] = {1,2,3};          //初始化内建数组
vector<int>      v   = {4,5,6};          //初始化标准容器
```

在函数里也可以直接用“{ ...}”作为值返回，类型会自动推导：

```
set<int> get_set()
{   return {2,4,6};}                     //直接使用花括号返回一个集合容器
```

实际上，花括号形式的语法会生成一个类型为 std::initializer_list 的对象，它定义在头文件<initializer_list>里，具有类似标准容器的接口，只要类型实现对它的构造函数就可以支持列表初始化。

1.4.3 新式 for 循环

遍历并操作 array、vector 等容器里的元素是 C++里常见的操作，通常我们会使用 for 循环语句，利用容器的首尾位置来完成，例如：

```
int a[] = {2,3,5,7};                     //内建数组

for(int i = 0;i < 4; ++i)                //使用 int 索引遍历
{   cout << a[i] << ",";}

vector<int> v = {253, 874};              //vector 容器，直接初始化

for(auto iter = v.begin();               //使用迭代器遍历容器
    iter != v.end(); ++iter)             //迭代器的终点是容器末尾
{   cout << *iter << ",";}               //解引用迭代器访问元素
```

C++11/14 引入了一种更简单便捷的 for 循环形式[1]，无须显式使用迭代器首尾位置，也无须解引用迭代器，就可以直接访问容器序列里的所有元素，例如：

```
for(auto x : a)                          //使用 auto 推导变量类型
{   cout << x << ",";}                   //直接访问元素，无须解引用

for(const auto& x : v)                   //使用 const auto&，推导为常引用类型
{   cout << x << ",";}                   //直接访问元素，无须解引用
```

① Boost 程序库提供功能类似的宏 BOOST_FOREACH，功能更强，不仅支持数组、容器，还支持迭代器的 pair 和逆序遍历，请参考推荐书目[3]。

这种新式 for 循环的正式名称是“基于范围的 for”（range-based for，C++11.6.5.4），使用“:”分隔了两个表达式，第一个是遍历容器时的元素类型，通常我们使用 auto 来自动推导，第二个表达式是遍历的目标容器。

在声明元素类型时使用 auto 将推导出值类型，有拷贝代价，也不能修改元素，所以可以为 auto 添加修饰，使用 const auto&/auto&&避免拷贝，或者用 auto&以修改元素的值。

```
for(auto& x : v)                    //使用auto，推导为引用类型，可修改值
{    cout << ++x << ",";}
```

新式 for 循环支持 C++内建数组和所有的标准容器，对于其他类型，只要它具有 begin() 和 end()成员函数，或者能够使用函数 std::begin()和 std::end()（参考 6.3 节）确定迭代的范围就可以应用于 for。

还需要注意一点：新式 for 循环只是一种“语法糖”，本质上还是使用迭代器来实现，要求遍历的容器必须能够确定迭代的范围，相当于：

```
auto&& _range = v;                       //获得容器的引用
for(auto _begin = std::begin(_range),    //确定范围的迭代起点
         _end   = std::end(_range);      //确定范围的迭代终点
     _begin != _end; ++_begin)           //循环的条件判断
{
    auto x = *_begin;                    //解引用迭代器，获取元素
    ...                                  //操作元素
}
```

可见，迭代范围已经在 for 循环开始前就确定好了，所以在 for 循环里我们不能变动容器，也不能增减容器里的元素，否则会导致遍历的迭代器失效，发生未定义错误。

1.4.4　新式函数声明

C++11/14 增加了一种新的函数语法，允许返回值类型后置（C++11.8.3.5）[①]，它使用了 auto/decltype 的类型推导能力，基本形式是：

```
auto func(...) -> type {...}             //新式函数声明语法
```

新语法的变化有两处。首先，函数返回值处必须使用 auto 来“占位”（与 1.3.1 节不同，这时的 auto 不具有类型推导的功能）；其次，函数名后需要用“-> *type*”的形式来声明真正

① 与 auto 一样，返回值类型后置语法也是 C++早期未被标准化的特性之一。

的返回值类型，这里的“*type*”可以是任意的类型，当然也包括 decltype，例如：

```
auto func(int x) -> decltype(x)          //使用 decltype 推导返回值类型
{    return x*x;}
```

初看上去这种语法十分的“怪异”，但这实际上是不得已而为之的做法，因为在泛型编程的时候，函数返回值的可能类型需要由实际的参数来决定，所以有必要将返回值类型的声明“延后”，请看下面的示范代码：

```
template<typename T, typename U>         //模板参数列表
auto calc(T t, U u) -> decltype(t + u)   //后置式函数声明语法
{    return t+u;}                        //返回两个变量之和
```

函数 calc()是一个简单的模板函数，但它的返回值类型却无法用传统语法声明[①]，因为返回值类型必须由参数 t 和 u 决定，而在函数名之前这两个参数还未定义。

后置式函数声明语法不是很常用，但关键时刻能够解决特定的问题，例如 1.7 节的 lambda 表达式。

1.5 面向对象编程

面向对象是一种重要的编程范式，C++一直支持得非常好，而在 C++11/14 里又增加了数个新特性，可以更好地控制类的封装、继承和多态，增强代码的自表述能力。

1.5.1 default

C++11/14 重用了关键字 default，允许程序员显式地声明类的缺省构造/析构等特殊成员函数，不仅能够明确地表示代码意图，而且可以让编译器更好地优化代码（C++11.8.4.2）。

default 的用法很容易理解，与声明纯虚函数的语法类似，在构造/析构等成员函数后面使用“=default”的方式就可以了，例如：

```
class default_demo
{
public:
    //显式指定构造函数和析构函数使用编译器的缺省实现
```

① 其实也可以用模板元编程来变通实现，使用元函数 common_type 对类型计算。

```
    default_demo()                                  = default;
    ~default_demo()                                 = default;

    //显式指定拷贝构造函数和拷贝赋值函数使用编译器的缺省实现
    default_demo(const default_demo&)               = default;
    default_demo& operator=(const default_demo&)    = default;

    //显式指定转移构造函数和转移赋值函数使用编译器的缺省实现
    default_demo(default_demo&&)                    = default;
    default_demo& operator=(default_demo&&)         = default;
};
```

使用 default 声明缺省构造函数后并不影响其他构造函数的重载与实现，我们仍然可以编写其他形式的构造函数：

```
class default_demo
{
public:
    ...                                  //之前的default构造、析构等函数
    default_demo(int x) {...}            //另一个构造函数
};
```

1.5.2 delete

与 default 类似，在 C++11/14 里关键字 delete 也新增加了一种用法，可以让程序员显式地禁用某些函数——通常是类的构造函数和拷贝构造函数，以阻止对象的拷贝（C++11.8.4.3）[①]，例如：

```
class delete_demo
{
public:
    delete_demo()        = default;          //使用default缺省实现
    ~delete_demo()       = default;

    //显式禁用拷贝构造函数和拷贝赋值函数
    delete_demo(const delete_demo&)               = delete;
    delete_demo& operator=(const delete_demo&)    = delete;
};
```

① 在 C++11/14 之前，禁用拷贝通常使用的是私有化拷贝构造函数的方式，可参考推荐书目[3]。

这样，delete_demo 就禁止了拷贝操作，变成了一个不可拷贝的对象，编译器会阻止任何拷贝构造或拷贝赋值的企图：

```
delete_demo d1;                          //声明一个对象
delete_demo d2 = d1;                     //无法拷贝赋值，发生编译错误
```

显式 delete 不仅可以作用于类成员函数，也可以作用于普通函数，禁用某些形式的重载。

1.5.3 override

C++的类继承体系里有虚函数的概念，它可以在运行时动态绑定，是实现多态的关键。

虚函数的声明需要使用 virtual 关键字，如果一个成员函数是虚函数，那么在后续派生类里的同名函数都会是虚函数，无须再使用 virtual 修饰。

但当继承关系较复杂或者派生类里的成员函数很多时，阅读者很难分辨出哪些函数继承自基类，哪些函数是派生类特有的，增加了代码的维护成本。而且还有一个潜在的隐患，派生类可能无意使用了一个同名但签名不同的函数“覆盖（overwrite）”了基类的虚函数。

下面的代码定义了两个简单的类：

```
struct base                              //基类
{
    virtual ~base() = default;           //虚析构函数，使用 default 实现

    virtual void f() = 0;                //纯虚函数
    virtual void g() const {}            //虚函数，const 修饰
    void h() {}                          //注意，非虚函数，没有 virtual 修饰
};

struct derived : public base             //派生类，缺乏必要的信息，理解困难
{
    virtual ~derived() = default;        //虚析构函数，使用 default 实现

    void f() {}                          //虚函数重载，很难辨别
    void g() {}                          //不是虚函数重载，签名不同，无 const 修饰
    void h() {}                          //不是虚函数重载，是覆盖
};
```

base 类很清晰，它定义了两个虚函数接口 f()和 g()，还有一个非虚函数 h()。但单独看

derived 类却信息很有限，必须结合基类才能理解它的实际含义：f()、g()、h()三个函数中只有 f()是正确的虚函数重载，g()因为“遗漏”了 const 修饰，函数签名与基类不一致，实际上是一个新的成员函数，而 h()则与虚函数没有任何关系，完全是 derived 类自己专有的函数，“覆盖”了 base 类里的原实现。

使用这两个类将导致不符合预期的结果：

```
unique_ptr<base> p(new derived);        //一个派生类对象，使用基类指针

p->f();                                 //正确，调用派生类的 f()
p->g();                                 //错误，调用了基类的 g()
p->h();                                 //调用了基类的 h()，未实现原意图
```

C++11/14 里增加了一个特殊的标识符“override”（C++11.8.2）[①]，它可以显式地标记虚函数的重载，明确代码编写者的意图：派生类里的成员函数名后如果使用了 override 修饰，那么它必须是虚函数，而且签名也必须与基类的声明一致，否则就会导致编译错误。

derived 的成员函数使用 override 标识符可以写成：

```
void f() override {}                    //override 修饰，明确是虚函数重载
void g() const override {}              //有 const 修饰，明确是虚函数重载
```

如果不小心写错了重载函数，override 也能够让编译器给出提示：

```
void g() override {}                    //无 const 修饰，不是虚函数重载，编译错误
void h() override {}                    //非虚函数重载，误用 override，编译错误
```

override 不是关键字，除了在成员函数后有特殊含义之外，在其他地方可以当作变量名或者函数名使用，但最好不要这样做。

1.5.4 final

C++的类体系非常自由灵活，但这种灵活性有时候也会带来麻烦，没有阻止类继承或者阻止重载虚函数的手段。例如，对于标准库里的 vector、list 等众多容器，设计者通常不希望它们派生出子类，但在 C++11/14 之前没有语言层面的强制保证。

与 override 一样，C++11/14 标准增加了特殊的标识符“final”，它不仅可以控制类的

① 在作者的印象中，似乎在 C++早期 override 就是一个关键字，但在标准化的过程中被去掉了。

继承，也可以控制虚函数。[①]

- 在类名后使用 final，显式地禁止类被继承，即不能再有派生类。
- 在虚函数后使用 final，显式地禁止该函数在子类里再被重载。

final 可以与 override 混用，更好地标记类的继承体系和虚函数，示范代码如下：

```
struct interface                              //基类，无 final，可以被继承
{
    virtual void f() = 0;                     //纯虚函数
    virtual void g() = 0;                     //纯虚函数
};

struct abstract : public interface           //抽象类，无 final，可以被继承
{
    void f() override final {}                //虚函数使用 final，f()不能再重载
    void g() override {}                      //仅使用 override，派生类还可以重载
};

struct last final : public abstract          //类声明使用 final，不能再被继承
{
    void f() override {}                      //错误，f()不能重载
    void g() override {}                      //g()仍然可以重载
};

struct error : public last                    //错误，last 类不能被继承
{};
```

在这段代码里类 abstract 继承了 interface，实现了虚函数 f()并使用 final 禁止之后的派生类再实现它。之后的类 last 在类名后使用了 final，使它变成了类体系里的终点，不能再被继承，类 error 尝试继承 last 会发生编译错误。

final 也不是关键字，仅在类声明里有特殊含义，但同样不建议再将它作为其他的标识符。

1.5.5 成员初始化

C++11/14 标准放松了对类成员变量初始化的要求，允许类在声明时使用赋值或者花括号的

① 这种用法类似于 Java 里的 final 或者 C#里的 sealed。

方式直接初始化，无须在构造函数里特别指定。

这是个非常便利的特性，特别是当类有很多成员变量的时候可以减少大量的代码，例如：

```
class demo
{
public:
    int             x = 0;                  //直接初始化整型变量，赋值方式
    string          s = "hello";            //直接初始化标准字符串，赋值方式
    vector<int>     v{1,2,3};               //直接初始化标准容器，花括号方式
};
```

但对于静态成员变量这种方法是不适用的，仍然要在类定义之外初始化（因为需要分配实际且唯一的存储空间）。

另外，这种赋值初始化的形式也不能使用 auto 来推导变量类型。

1.5.6 委托构造

有的时候我们会声明多个不同形式的构造函数，用于在不同情况下创建对象，这些代码大都是初始化成员变量，非常类似，仅有少量的不同，但代码却并不能复用，导致代码冗余。

常用的解决办法是实现一个特殊的初始化函数（名字通常是 init），然后在每个构造函数里调用它，例如：

```
class demo                                  //有多个构造函数的类
{
private:
  int x,y;
  void init(int a, int b)                   //内部的初始化函数
  {    x=a;y=b;}
public:
  demo()                                    //缺省构造函数
  {    init(0, 0);}                         //调用初始化函数

  demo(int a)                               //单参数构造函数
  {    init(a, 0);}                         //调用初始化函数

  demo(int a, int b)                        //双参数构造函数
  {    init(a, b);}                         //调用初始化函数
```

```
};
```

C++11/14 标准引入了委托构造函数（delegating constructor，C++11.12.6.2）的概念，解决方法基本相同，但不必再专门写一个特殊的初始化函数，而是可以直接调用本类的其他构造函数，把对象的构造工作“委托”给其他构造函数来完成。

使用委托构造函数可以很好地简化代码，刚才的 demo 类可以改写成：

```
class demo                                  //有多个构造函数的类
{
public:
  demo() : demo(0, 0) {}                    //缺省构造函数，委托给双参数的构造函数

  demo(int a) : demo(a, 0) {}               //单参数构造函数，委托给双参数的构造函数

  demo(int a, int b)                        //双参数构造函数，被其他构造函数调用
  {     x=a;y=b;}
};
```

委托构造函数可以配合类成员在初始化时使用，在类声明时先初始化成员变量，然后使用委托构造函数进行额外的构造操作，进一步简化代码。

1.6 泛型编程

自从关键字 template 出现后，泛型编程就逐渐成为了 C++的主流编程范式之一，更扩展出了运行在编译期的模板元编程。泛型编程使泛型容器、泛型算法成为可能，不仅深刻地改变了 C++语言，同时也影响了 C++之外的其他编程语言。

在现代 C++语言里泛型编程已经是绝对的主流， C++11/14 增加了不多但很重要的特性，能够更好地支持泛型编程和模板元编程。

1.6.1 类型别名

C++11/14 扩展了 using 关键字的能力，可以完成与 typedef 相同的工作，使用“using *alias = type;*”的形式为类型起别名（有些像 swift 里的 let 语句），例如：

```
using int64 = long;                         //long 类型的别名是 int64
using llong = long long;                    //long long 类型的别名是 llong
```

using 语句里的别名和原类型的顺序与 typedef 正好相反，这种形式很像赋值语句[①]，用的是我们所熟悉的运算符“=”，因而更容易阅读和理解。

但 using 的能力不止于此，它已经超越了 typedef，还可以结合 template 关键字为模板类声明“部分特化”的别名（C++11.14.5.7），功能更加强大，例如：

```
template<typename T>                          //为标准容器 map 起别名
using int_map = std::map<int, T>;             //固定 key 类型为 int

int_map<string> m;                            //使用别名，省略了一个模板参数

template<typename T, typename U>              //自定义类，两个模板参数
class demo final {};

template<typename T>                          //保留一个模板参数
using demo_long = demo<T, long>;              //别名，第二个模板参数固定为 long

demo<char, int> d1;                           //原模板类，给出两个模板参数
demo_long<char> d2;                           //模板别名，只需要一个模板参数
```

对于拥有复杂模板参数列表的类来说，using 的别名用法可以给出一些常用的形式，简化模板类的使用，增加代码的可读性。

1.6.2 编译期常量

在 C++里用关键字 const 可以定义常量，例如：

```
const int k = 1024;                           //整数常量，值为 1024
```

但这样的“常量”实际上是运行时不可修改的“变量”，在泛型编程里有时会需要在编译期可用的“真正的”常量，C++11/14 之前通常只能够使用#define 定义宏来实现，这脱离了 C++编译器的掌控。

C++11/14 增加了新关键字 constexpr（C++11.5.19），它相当于编译期的 const，但功能更强，令所修饰的表达式或函数具有编译期的常量性，可以让编译器更好地优化代码，例如：

```
constexpr int kk = 1024;                      //一个编译期常量
```

① 实际上这就是编译期的赋值语句，可以用于模板元编程，参见 2.3 节。

```
constexpr long giga()                          //编译期常量函数
{    return 1000*1000*1000;}                   //返回一个常量整数
```

constexpr 让编译期的整数计算成为了可能，标准库里的 ratio 就大量应用了 constexpr，实现了编译期的有理数。

在 C++11 里 constexpr 的函数用法还比较严格，仅允许函数里有一个 return 语句，但在 C++14 中做了适当的放宽，这里不做详细阐释。

1.6.3 静态断言

C 语言提供断言 assert，它是一个宏，可以在运行时验证某些条件是否成立，有利于保证程序的正确性。但泛型编程主要工作在编译期，assert 无法起作用。

C++11/14 增加了新关键字 static_assert，它是编译期的断言，可以在编译期加入诊断信息，提前检查可能发生的错误。

static_assert 的用法与 assert 类似，需要一个编译期的 bool 表达式和警告消息字符串，基本形式是：[①]

```
static_assert(condition, message)          //要求编译期的条件必须成立，否则报错
```

如果 bool 表达式在编译期的计算结果为 false，那么编译器会报错，并给出 message 提示。

下面的代码利用 sizeof 可在编译期计算的功能，对整数类型的长度做了明确要求：

```
static_assert(sizeof(int)==4, "int must be 32bit!");
static_assert(sizeof(long)>=8, "need 64bit!");
```

static_assert 通常需要配合 type_traits 库来使用，利用 type_traits 库里的诸多元函数来检查各种编译期条件，读者可进一步参考第 3 章。

1.6.4 可变参数模板

C/C++的函数支持可变参数，允许函数接受任意数量的参数，最典型的就是 printf()：

```
int printf( const char *format, ... );
```

① Boost 程序库提供类似功能的库，使用宏 BOOST_STATIC_ASSERT，请参考推荐书目[3] 。

为了更好地支持编译期的模板元编程，C++11/14 标准引入了与之类似的特性，称为可变参数模板（或简称为“变参模板”，variadic templates，C++11.14.5.3），语法也十分接近，同样使用了省略号“...”来声明不确定数量的参数，形式如下：

```
template<typename ... T> class some_class {};      //模板类
template<typename ... T> void some_func() {};      //模板函数
```

模板参数列表里的“typename ... T”声明了一个具有不确定数量参数的模板列表——模板参数包（template parameter pack），其数量可以是 0，也可以是 1 或者更多，可以使用 sizeof...(T)的方式来得到具体数量。

声明了可变模板参数后，我们还需要解开参数包（unpack）才能使用，这时不需要再使用关键字 typename 或 class，直接在类型名后用“...”即可，例如：[①]

```
template<typename ... T>                         //声明可变参数模板
class variadic_class
{
    using type = x<T...>;                        //使用“...”解包，用于模板类
};

template<typename ... Args>                      //声明可变参数模板
void variadic_func(Args... args)                 //使用“...”解包，用于模板函数
{   cout << sizeof...(Args) << endl;}            //sizeof...计算数量
```

对于模板函数，在解包时还可以对类型附加“const/&/&&”等修饰，更好地限定参数的类型，例如 printf()就可以改写成：[②]

```
template<typename ... Args>                      //声明可变参数模板
int print( const char *format, const Args& ... args);
{
    return printf(format, args...);              //解包参数并转发
}
```

可变参数模板的特性对于泛型编程和模板元编程意义重大，可以摆脱模板参数数量的限制和晦涩难懂的预处理语句，写出更安全易读的高质量代码，标准库和 Boost 库里的许多组件都因此而受益（如 tuple、ref、bind、function 等）。

---

① 可变参数模板的解包过程可以理解为以 typename 起始，用“...”连续展开的链式操作：typename ... T => T ... args => args ... 。

② 这里也可以使用完美转发，代码可以是 std::forward<Args>(args)...。

1.7 函数式编程

函数式编程（functional programming）是与面向过程编程、面向对象编程、泛型编程并列的一种编程范式，它基于 λ 演算理论（lambda calculus），把计算过程视为数学函数的组合运算。

早期 C++语言里就已经有了基本的函数式编程，定义重载了 operator()的类——即函数对象，再搭配标准算法和函数绑定器，就可以嵌套组合这些函数完成各种功能。

C++11/14 标准专门引入了 lambda 表达式（C++11.5.1.2），不仅能够更好地支持函数式编程，而且也能够简化代码。

1.7.1 lambda 表达式

在 C++11/14 标准里，lambda 表达式实际上是对函数对象的一种强化和扩展，可以直接就地定义“匿名”的函数对象（所谓的“语法糖”）。

lambda 表达式是 C++语言里表达式的一种，为了能够在已有语法中引出 lambda 表达式，继尖括号“<>”在泛型编程（模板）中被“废物利用”后，方括号“[]”再度“惨遭毒手”（笑），基本的形式是：

```
[](params){ ...}                              //lambda 表达式的基本形式
```

在这里“[]”称为 lambda 表达式引出操作符（lambda introducer），它之后的代码就是 lambda 表达式，形式如同一个标准的函数，圆括号里是函数的参数，而花括号内则是函数体，可以使用任何 C++语句，实现任何功能。[1]

一个最简单的 lambda 表达式如下：

```
[](){}                                        //一个简单但略“古怪”的 lambda 表达式
```

lambda 表达式的类型称为“闭包”（closure type），无法直接写出，所以通常需要使用 auto 的类型推导功能来存储。下面的代码示范了更多的 lambda 表达式：

```
auto f1 = [](int x){                          //函数参数是整数
    return x*x;                               //整数计算
```

① 目前 lambda 表达式还不支持 function-try 用法，只能在函数体里使用 try-catch 捕获异常。

```
    };                                          //表达式结束，语句末尾需要用分号

auto f2 = [](string s) {                       //函数参数是标准字符串
    cout << "lambda : " << s << endl;          //标准流输出
    };                                          //表达式结束，语句末尾需要用分号

auto f3 = [](int x, int y) {                   //两个函数参数
    return x < y;                               //比较运算
    };                                          //表达式结束，语句末尾需要用分号
```

lambda 表达式的行为类似函数对象，可以用 operator()来调用，也可以用于各种标准算法，完全不必再组合使用那些预定义函数对象和绑定器，而且更具可读性，例如：

```
cout << f1(3) << endl;                         //直接调用 lambda 表达式
f2("heavy rain");
cout << f3(1,5) << endl;

vector<int> v = {1,3,5,7};                     //标准容器
std::for_each(v.begin(), v.end(),              //标准算法
    [](int x ){                                //直接写匿名的 lambda 表达式
        cout << x << ",";
    });

std::for_each(v.begin(), v.end(),              //标准算法
   [](int& x ){                                //lambda 表达式使用引用类型，可以修改值
       if(x > 3) { x *=2;}                     //变动元素，相当于 transform 算法
    });
```

lambda 表达式的返回值会自动推导，但也可以使用新的返回值后置语法，形式是：

```
[](params)-> return_type {...}                 //返回值后置语法的 lambda 表达式
```

例如：

```
auto f4 = [](int x) -> long {...};             //指定 lambda 表达式的返回值类型为 long
```

1.7.2 捕获外部变量

在 1.7.1 节我们看到了 lambda 表达式的基本用法，可以随时随地定义匿名函数对象，非常方便。但 lambda 表达式的功能远不止于此，它还能够“捕获”外部变量，这才是闭包的真正威力。

lambda 表达式的完整声明语法是：

```
[captures](params) mutable ->type {...}  //lambda 表达式的完整形式
```

在 lambda 表达式引出操作符[ ]里的“*captures*”称为“捕获列表”，可以“捕获”表达式外部作用域的变量，在函数体内部直接使用，这是与普通函数或函数对象最大的不同之处。[①]

捕获列表里可以有多个捕获项，以逗号分隔，使用了略微“新奇”的语法，规则如下所述。

- [ ]　　：无捕获，函数体内不能访问任何外部变量。
- [=]　　：以值（拷贝）的方式捕获所有外部变量，函数体内可以访问但不能修改。
- [&]　　：以引用的方式捕获所有外部变量，函数体内可以访问并修改（需当心无效引用）。
- [*var*]　：以值（拷贝）的方式捕获某个外部变量，函数体内可以访问但不能修改。
- [&*var*]　：以引用的方式捕获某个外部变量，函数体内可以访问并修改。
- [this]　：捕获 this 指针，可以访问类的成员变量和成员函数。
- [=,&*var*]：引用捕获变量 var，其他外部变量使用值捕获。
- [&,*var*]　：值捕获变量 var，其他外部变量使用引用捕获。

下面的代码示范了这些捕获列表的用法：

```
int x = 0, y = 0;

auto f1 = [=](){ return x;};            //以值方式捕获使用变量，不能修改
auto f2 = [&](){ return ++x;};          //以引用方式捕获所有变量，可以修改，但要当心引用无效
auto f3 = [x](){ return x;};            //以值方式捕获 x，不能修改
auto f4 = [x,&y](){ y += x;};           //以值方式捕获 x，以引用方式捕获 y，y 可以修改
auto f5 = [&,y](){ x += y;};            //以引用方式捕获 y 之外的所有变量，y 不能修改
auto f6 = [&](){ y += ++x;};            //以引用方式捕获所有变量，可以修改
auto f7 = [](){ return x;};             //无捕获，不能使用外部变量，编译错误
```

值得注意的是变量的捕获发生在 lambda 表达式的声明之时，如果使用值方式捕获，即使之后变量的值发生变化，lambda 表达式也不会感知，仍然使用最初的值。如果想要使用外部变量的最新值就必须使用引用的捕获方式，但也需要当心变量的生命周期，防止引用失效。

---

① C++里的闭包必须显式指定捕获，而 Lua 语言里的则是默认直接捕获所有的外部变量。

刚才的 lambda 表达式运行的结果是：

```
f1();                    //以值方式捕获，x、y 不发生变化
f2();                    //函数内部 x 值为 0，之后变为 1，y 没有被修改，值仍然是 0
f3();                    //函数内部 x 值仍然是 0，即 f3()==0
f4();                    //x、y 均是 0，运算后 y 仍然是 0
f5();                    //y 是 0，引用捕获的 x 是 1，运算后 x 仍然是 1
f6();                    //x、y 均引用捕获，运算后 x、y 均是 2
```

捕获列表增强了 lambda 表达式与外部的交互能力，能够让我们不限于函数参数，使用简明易读的代码实现出复杂的功能，例如：

```
vector<int> v = {1,2,3,4,5};                //标准容器
int sum = 0;                                //外部变量，用于求和

std::for_each(v.begin(), v.end(),           //标准算法
    [&](int x){                             //以引用方式捕获外部变量
        sum += x;                           //直接操作外部变量
    });                                     //相当于 accumulate 算法

int k = 3;                                  //外部变量，用于比较
auto c = std::count_if(                     //标准算法
    v.begin(), v.end(),
    [=](int x){                             //以值方式捕获外部变量
        return x > k;                       //与外部变量进行比较运算
    });                                     //相当于 bind+greater
```

lambda 表达式还可以使用关键字 mutable 修饰，它为值方式捕获添加了一个例外情况，允许变量在函数体内也能够修改，但这只是内部的拷贝，不会影响外部的变量，例如：

```
auto f = [=]() mutable {return ++x;};      //x 仅可以在内部修改，不影响外部变量
```

1.7.3 类型转换

C++11/14 标准允许把无捕获的 lambda 表达式转换为一个签名相同的函数指针，例如：

```
auto f = [](){};                            //注意，无捕获的 lambda 表达式

typedef void (*func)();                     //函数指针类型

func p1 = f;                                //可以隐式转换为函数指针
func p2 = [](){ cout<<endl;};               //也可以直接赋值给函数指针
```

这条特别的规定实现了 lambda 表达式与 C 接口的互操作性，可以让我们在调用外部 C 函数时也能直接使用 lambda 表达式，就地编写小函数传递给 C 接口，相当于变相地为 C 语言增加了闭包的能力。

注意在转换为函数指针时 lambda 表达式必须是无捕获列表的，也就是说必须是以“[]”起始，这样的 lambda 表达式没有内部状态，相当于“纯函数”，否则会编译失败：

```
auto g = [&](){ ++x;};                    //有捕获的 lambda 表达式
func p3 = g;                              //无法转换，编译错误
```

1.7.4 泛型的 lambda 表达式

在 C++11 标准里 lambda 表达式的函数参数必须是具体的类型，而 C++14 则为 lambda 表达式增加了泛型的功能（相当于模板化的函数对象）。

泛型的 lambda 表达式在语法上并没有使用 template 关键字和“<>”，而是用 auto 来声明参数类型，虽然语法不同，但模板参数推导规则是相同的。[①]

除了函数参数声明改用 auto，泛型的 lambda 表达式与普通 lambda 表达式没有任何不同：

```
auto f = [](auto x) {...};             //泛型 lambda 表达式，函数参数使用 auto 推导
```

1.8 并发编程

随着近年来多核 CPU 的普及，并发编程已经成为了提高程序运行性能的一个必备手段。C++11/14 标准充分考虑了这个现实的需求，为并发特别是多线程提供了较好的支持，但主要是以库的形式，包括<atomic>、<mutex>、<thread>等。[②]

在语言本身方面，新增的 lambda 表达式在并发编程里也经常扮演着重要的角色，因为它可以在启动线程或者异步操作时就地定义运行函数，增强代码的可读性和可维护性。

此外，C++11/14 标准还引入了新的关键字 thread_local（C++11.7.1.1），它实现了线程本地存储（TLS，thread local storage），是一个与 extern、static 类似的变量存储指示标记。

① 在理论上，使用 template<...>是完全可以的，但可能是标准委员会也觉得“<>[](){}”这样的“括号全家福”语法太难以接受。

② 读者可参考推荐书目[3]里对 atomic、thread 等库的介绍。

线程本地存储是多线程编程里的概念，是指变量在进程里拥有不止一个实例，每个线程都会拥有一个完全独立的、"线程本地化"的拷贝，多个线程对变量的读写互不干扰，完全避免了竞争、同步的麻烦。例如：

```
extern       int x;                 //外部变量，实体存储在外部，非本编译单元
static       int y = 0;             //静态变量，实体存在在本编译单元
thread_local int z = 0;             //线程局部存储，每个线程拥有独立的实体
```

这段代码声明了三个变量：x 使用 extern 修饰，是一个外部变量；y 使用 static 修饰，是静态变量，只能在本实现文件内访问，在多线程环境下是不安全的；而 z 使用了 thread_local 关键字，是线程安全的，每个线程都有一份只属于自己的实例。

下面的代码验证了线程本地存储的结果：

```
auto f = [&](){                        //lambda 表达式，线程实际执行的函数
    ++y; ++z;                          //分别操作 static 和 thread_local 变量
    cout << y << "," << z << endl;     //在线程里输出值
    };

thread t1(f);                          //启动两个线程
thread t2(f);

t1.join(); t2.join();                  //等待线程执行完毕

cout << y << "," << z << endl;         //在主线程里输出变量值
```

运行的结果是：

```
1,1                                    //第一个线程，y=1,z=1
2,1                                    //第二个线程，y=2,z=1
2,0                                    //主线程，y=2,z=0
```

可见，静态变量 y 在进程里是唯一的，两个线程都改变了 y 的值，而 z 因为是 thread_local 的，两个子线程和主线程分别持有互相独立的三个实例，所以子线程里均独立加 1，而主线程因为没有对 z 做任何操作，所以值是 0。

thread_local 变量的生命周期比较特殊，它在线程启动时构造，在线程结束时析构，也就是说仅在线程的生命周期里是有效的，比 static 变量的生命周期（整个进程）要短，但比普通局部变量的生命周期长。

thread_local 仅适用于线程需要独立存储的情况，当线程间需要共享资源访问时仍然需

要使用互斥量等保护机制。

1.9 面向安全编程

本节叙述的 C++11/14 新特性可以让我们的代码更加安全，更不容易出错。

1.9.1 无异常保证

C++的异常机制比较复杂，允许程序抛出任意类型的对象作为异常。为了规范异常的使用，C++提出了“异常规范”（exception specifications）的概念，可以使用 throw(...)的形式来说明函数可能会抛出哪些异常。虽然这个规定的本意是好的，但实际上却并不成功，很少有人使用这个特性。

在 C++11/14 里“异常规范”被正式废弃，但保留了一个很小的功能：声明函数不会抛出任何异常，并引入一个新关键字 noexcept 来明确表述这个含义（C++11.15.4）[①]，例如：

```
void func() noexcept {}                    //函数绝对不会抛出异常
```

使用 noexcept 来修饰函数可以减少异常处理的额外成本，提高运行效率。

1.9.2 内联名字空间

C++使用名字空间来解决命名冲突的问题，关键字 namespace 可以声明一个专有的作用域，其内的所有变量、函数或者类都不会与外部发生冲突，但使用时也必须加上名字空间限定，或者使用 using 打开名字空间。

namespace 通常都需要用一个名字来标识，但 C++也允许不使用名字而声明一个“匿名”的名字空间，这时它的作用相当于使用 static 静态化了名字空间里的成员，例如：

```
namespace {                                //匿名名字空间，注意没有名字
    int x = 0;                             //具有静态链接属性
}                                          //匿名名字空间结束
assert(x == 0);                            //无须名字空间限定，可直接访问
```

C++11/14 增加了内联名字空间的概念（C++11.7.3.1），允许在一个名字空间声明前用 inline 关键字修饰，使外部同样可以无须限定直接访问这个名字空间内部的成员：

① noexcept 相当于 C++98 里异常规范的 throw()用法。

```
inline namespace temp{                    //内联名字空间，有名字，用inline关键字
    int xx = 0;
}                                         //内联名字空间结束
assert(xx == 0);                          //同样无须名字空间限定，可直接访问
```

内联名字空间的这个特性对于代码的版本化很有用，程序员可以在多个子名字空间里实现不同版本的功能，而发布时用 inline 对外只暴露一个实现，很好地隔离版本的差异，利于维护，示范用法如下：

```
namespace release {                       //对外发布的名字空间
    namespace v001 {                      //旧版本的名字空间
        void func() {}                    //旧版本的实现
    }                                     //旧版本的名字空间结束
    inline namespace v002 {               //当前版本的名字空间，使用inline内联
        void func() {}                    //当前版本的实现
    }                                     //当前版本的名字空间结束
}                                         //对外发布的名字空间结束
```

release 名字空间里有两个子名字空间：v001 和 v002，里面有各自不同的实现，可以独立演化维护，发布时使用 inline 决定真正外界可用的接口，外界无须知道子名字空间的实现细节，直接使用 release 名字空间限定就可以调用正确的版本：

```
release::func();                          //看不到子名字空间，直接使用父名字空间
```

1.9.3 强类型枚举

在早期的 C++里枚举（enum）是弱类型，相当于整数，而且枚举值直接暴露在外部名字空间，缺少限定，命名容易冲突，例如：

```
enum color {                              //一个简单的枚举类型
    red = 1,    green = 2,    blue = 3
};

assert(red == green - 1);                 //枚举值可以直接当作整数来运算
int red = 1;                              //名字冲突，编译错误
```

C++11/14 为 enum 增加了安全性，可以用“enum class/struct”的形式声明强类型的枚举（C++11.7.2），它不能被隐式转换为整数，而且必须要使用类型名限定访问枚举值，就像类的静态成员一样：

```
enum class color {...};                   //使用enum class声明强类型枚举
```

```
auto x = color::red;                  //必须使用类型名限定
auto y = red;                         //错误，没有使用类型名限定
auto z = color::red + 1;              //错误，不能隐式转换为整数运算
```

C++11/14 还允许在枚举声明后用 char、int 等指示枚举使用的整数类型，让程序员更好地控制空间占用。例如，下面的代码让枚举使用 char 类型存储：

```
enum class color : char {...};        //要求枚举使用 char 整数存储
```

1.9.4 属性

在 GCC、MSVC 等编译器里有特殊的语法形式来标识某些编译设置，例如 GCC 使用的是“__attribute__((...))”的方式，但这些语法扩展属于“方言”，不同编译器之间不能通用。

C++11/14 标准为了方便编译器优化代码，规范了类似的用法，提出了属性（attribute）的概念（C++11.7.6），并再次利用了方括号，使用“[[attribute]]”的形式标记编译特征，指示编译器做某种程度的优化。

在 C++11 标准里属性的应用还比较保守，标准委员会只规定了两种属性：noreturn 和 carries_dependency。这两个属性对于通常的编程来说用处并不大。C++14 新增加了一个 deprecated 属性，可以明确地标记被废弃的变量、函数或者类，相当于 GCC 里的 __attribute__((deprecated))。

属性可以作用于很多 C++语法元素，包括变量、函数、类等，位置也比较灵活，下面的示范代码使用了 deprecated 属性：[①]

```
[[deprecated]] int x = 0;             //声明变量 x 已经被废弃
class [[deprecated]] demo {};         //声明类 demo 已经被废弃
```

1.10 更多特性

本节简要叙述 C++11/14 里一些很有用但不易被归类的其他特性。

1.10.1 语言版本号

C++标准预定义了宏“__cplusplus”（C++11.16.8），是一个整型常数，利用它程序能

① GCC 4.8.2 尚不支持 deprecated 属性，只会报出未识别属性警告，GCC 5 可以正确识别。

够辨别当前编译器使用的标准的版本。

- 未定义 ：不是C++编译器，而是C编译器。
- 199711L ：C++98/03。
- 201103L ：C++11。
- 201402L ：C++14。[①]

我们可以直接在代码里输出__cplusplus的值，例如：

```
cout << "C++ : " << __cplusplus << endl;          //在GCC4.8.2里值是201103
```

也可以在预处理指令里使用__cplusplus，用于条件编译的判断逻辑：

```
#if __cplusplus < 201103L                          //要求必须是C++11标准或更高
  #error "we need the latest C++ standard!"        //否则编译报错
#endif                                             //条件编译结束
```

1.10.2 超长整型

C++11/14增加了C99里定义的超长整数类型long long，它至少有64位，即：

```
static_assert(sizeof(long long) >= 8, "long long is 64bit");
```

long long类型可以用unsigned修饰，表示无符号超长整型：

```
long long             m = -1000*1000;          //有符号超长整型
unsigned long long    n = 1000*1000;           //无符号超长整型
```

C++11/14标准也增加了新的字面值后缀，可以用“LL/ll”或者“ULL/uLL/ull”来显式地说明整数的类型是long long，这样可以简单地使用auto来自动推导类型，例如：

```
auto a = 524287LL;                             //梅森素数，2^19-1
auto b = 2147483647uLL;                        //梅森素数，2^31-1
```

1.10.3 原始字符串

C++11/14提供了一种新的字符串书写方式——使用“R"(...)"”形式的原始（raw）字符串，它以大写的“R”开始，之后在括号内的所有字符都会原样保留，作为字符串组成部分，

① 本书使用的GCC 4.8.2不支持C++14，要验证C++14的功能可使用GCC 5，并使用选项-std=c++14。

不会有任何改变，例如：

```
string s = R"(this is a "\string\")";          //使用原始字符串
cout << s << endl;                              //输出“this is a "\string\"”
```

在这段代码里，我们使用了“R"(...)"”来声明原始字符串，虽然字符序列里有特殊符号“"”和“\”，但它们并不需要转义，而是直接保留在了字符串里。

这种特性对于书写正则表达式特别有用，因为正则表达式里经常需要使用“\”来定义规则，使用原始字符串可以获得更清晰的代码，不容易出错：

```
auto reg = R"(^\d+\s\w+)";                      //相当于"^\\d+\\s\\w+"
```

原始字符串还有一种扩展形式，允许在圆括号两边加上最多 16 个字符——称为 delimiter，更好地标记字符串，例如：

```
auto b = R"***(BioShock Infinity)***";          //定界符使用“***”
auto d = R"====(Dark Souls)====";               //定界符使用“====”
```

注意圆括号两边的 delimiter 必须相同，而且不能使用“@”“$”“\”等特殊字符。

1.10.4 自定义字面值

“字面值”(literal）是指在 C++源码里的数字、字符串等常量，并且可以附加一些前缀或者后缀来进一步描述其类型，例如：

```
auto f = 3.14f;                    //后缀 f 指示 float 类型
auto s = L"wide char";             //前缀 L 指示 wchar_t 类型
auto x = 0x100L;                   //前缀 0x 指示十六进制，后缀 L 指示 long 类型
```

C++11/14 增加了用户自定义字面值的功能（C++11.2.14.8），允许我们为字面值增加后缀（但没有开放前缀），定义函数来对字面值进行运算，决定字面值的实际类型，从而简化代码。

自定义字面值需要重载新的操作符“""”（即两个连续的双引号），形式通常如下：

```
return_type operator"" _suffix (unsigned long long);
return_type operator"" _suffix (long double);
return_type operator"" _suffix (const char*, size_t);
```

函数声明里的“_suffix”就是用户自定义的后缀，当编译器发现字面值使用了自定义后缀时就会把字面值转换为函数调用。特别要注意的是函数名字必须以下划线开头，根据标准，没有

下划线的后缀被保留给将来使用，否则会导致编译警告。

自定义字面值函数的参数不是任意的，只能使用 C++11/14 标准里的固定的几种形式（因为源码里字面值的类型只有整数、浮点数、字符串等有限的形式），较常用的参数类型有 unsigned long long、long double、const char* 等。

用户可以自由定义函数的返回值类型，不仅可以返回 int、string 等标准类型，也可以返回任意的自定义类型。

下面的代码示范了自定义字面值的用法：

```
long operator"" _kb(unsigned long long v)              //自定义后缀“_kb”
{    return v * 1024;}                                 //乘以 1024 后返回整数

complex<double>
operator"" _c(const char* s, size_t n)                 //自定义后缀“_c”
{
    using namespace boost::xpressive;                  //使用 Boost 的正则表达式库

    auto reg = cregex::compile(                        //声明正则表达式
            R"--(([0-9\.]+)\+([0-9\.]+)i)--");         //形如“a+bi”的复数
    cmatch what;                                       //捕获子表达式

    auto ok = regex_match(s, what, reg);               //正则匹配
    assert(ok);                                        //要求匹配成功，错误不处理

    return complex<double>(                            //返回复数类型
            stod(what[1]), stod(what[2]));             //两个字符串转换为数字
}
```

这段代码里我们定义了两个字面值后缀。第一个是“_kb”，它转换数字字面值，为数字增加 KB 单位。第二个后缀是“_c”，它使用了 Boost 的 xpressive 正则表达式库处理字符串，解析形如“a+bi”的复数字符串，然后转换为标准复数类 complex。[①]

这些自定义字面值可以这样使用：

```
auto x = 2_kb;                                         //使用后缀“_kb”
assert(x == 2*1024);                                   //实际数值是 2048
```

① 也可以使用 C++标准库里的 regex，读者可自行尝试。

```
auto c = "1.414+1.414i"_c;                      //使用后缀"_c"
cout << c << endl;                              //输出(1.414,1.414)
```

虽然 C++11 标准引入了自定义字面值，但却没有预定义任何可用的新字面值。而 C++14 标准里则增加了"h/min/s/ms"等新的字面值后缀，可以直接在代码里书写时间单位或者标准字符串，例如：

```
auto t1 = 2min;                  //2 分钟
auto t2 = 30s;                   //30 秒钟，注意与下一行的区别
auto s  = "std string type"s;    //字符串后的"s"表示标准字符串，不是"秒"
```

1.10.5 杂项

C++11/14 里还有更多的新特性，这里列出了一些小的语法细节改进。

右尖括号

在 C++98 中，模板参数列表里不能出现两个连续的右尖括号">>"，否则编译器会把它解释为右移运算符，这导致在写某些复杂的模板类时，模板参数列表末尾的多个">"之间必须用空格分隔，虽然这是个很小的问题，但的确有些麻烦。

C++11/14 弥补了这个缺陷，在模板声明里遇到">>"时会优先解释为模板参数列表的结束标记。

函数的默认模板参数

C++98 允许模板类在模板参数列表里使用默认参数，但模板函数却不允许，C++11/14 解除了这个不必要的限制，让模板函数也可以使用默认参数。

增强的联合体

C++11/14 放宽了对 union 成员的限制，union 内的成员不仅可以是 POD 类型，也可以是有构造/析构函数的自定义类型（但不能有虚函数和引用成员），还可以拥有成员函数，使它更接近于 class。[①]

二进制字面值

C++14 标准增加了新的字面值前缀，可以用"0b/0B"来直接书写二进制数字。

---

① 由于成员共用存储的特殊性，union 在使用上还是存在一些局限，Boost 程序库提供类似的库 variant，它是更好的 union，可参考推荐书目[3]。

数字分位符

在书写很长的数字时，C++14 标准允许使用单引号“'”来将数字分组，以增强可读性。

1.11 总结

C++11/14 标准是一个“全新”的语言，它吸收了一些编程语言后辈的优点，既继承传统又积极创新，对语言核心做了很多的改进。本章重点阐述了它的一些语言新特性，这些新特性虽然很多，但并不显得混乱，而是和谐地融为一个整体。

我们首先讨论了左值与右值、自动类型推导这两个在现代 C++里非常重要、非常基本的特性。

早期的 C++并没有明确的右值概念，导致了“深拷贝”等许多问题。C++11/14 里的右值完善了表达式的定义，转移语义也可以零成本构造对象，可以编写出效率更高的代码。

auto 和 decltype 可以推导表达式的类型，对于任何级别的 C++用户都是非常有意义的。初级用户可以使用 auto 来简化类型的声明，减少代码量，写出更容易理解和维护的代码；专家用户则可以使用 decltype 向编译器主动“索取”类型名，用在泛型编程、模板元编程等更多的领域。需要注意的是 auto 和 decltype 的类型推导规则，有的细节必须要小心，否则可能会因类型使用不当而导致编译或运行问题。

在这两个基本特性之后，我们以 C++多编程范式为脉络，分门别类研究了 C++11/14 在面向过程、面向对象、泛型、函数式、并发等范式里的许多（但不是全部）新特性，读者可以依据编程习惯，选择自己喜好的新特性学习并实践应用。在这些新特性中，个人认为最有价值的是 nullptr、range-based for、final、可变参数模板、lambda 表达式和原始字符串，它们可以运用在任何编程范式里，让 C++代码和编程思维更加“现代化”。

此外，本章仅简略介绍了 C++11/14 的语言特性，并没有太多涉及库，实际上库远比语言本身更为庞大、涉及的领域更广，例如单向链表、散列容器、正则表达式、线程等。本书随后将以“准标准库”——Boost 程序库为主，结合 C++11/14 标准介绍更多的 C++高级技术。

第2章 模板元编程简介

从引入 template 关键字开始，C++里就出现了泛型编程，而由泛型编程衍生出的模板元编程（template meta-programming，简称“元编程”）则无疑是众多编程范式中最复杂、最强大和最具威力的一种，可算得上是 C++的“终极”技巧。

所谓“元程序”—— metaprogram，意思就是“a program about a program”，它有着完全不同于普通程序的许多特点，是一种全新的编程体验。

本章简要讨论模板元编程的一些基本概念，它们是许多现代 C++和 Boost 程序库组件的基础，只有熟悉了它们才能够理解模板元编程和元程序，有助于我们对 C++的进一步学习。

当然模板元编程中远不止这些，还有编译期的 lambda 表达式、容器、迭代器、算法、视图等更高级的概念，本书的第 13 章将对其进行叙述。

2.1 概述

元编程（meta-programming）也被译为“超程序”“超编程”或“产生式编程”，这些译法在一定程度上反映了其本质——它是一种位于普通程序之上、超越普通程序的程序，是一种可以操纵、产生程序的程序[①]。

C++中的模板元编程是无意中被“发现”而不是“发明”出来的，是一种对“类型”计算的

① 其实我们对这种“产生程序的程序”并不陌生，相对于汇编语言来说，C/C++程序就可以说是一种产生汇编代码的元程序。但 C++中的元程序的神奇之处就在于元程序和普通程序完全融为一体，没有明显的界限。

程序，关于它的诞生有许多有趣的传奇和故事。

模板元编程本质上是泛型编程的一个子集，从广义上来说，所有使用 template 的泛型代码都可以称作元程序——因为泛型代码并不是真正可编译执行的代码，它们只是定义了代码的产生规则，是用来生成代码的“模板”。

然而模板元编程又不完全等同于泛型编程，它是一种“函数式编程”，并且已经被证明是图灵完备的，也就是说它具有足够的“计算”能力，可以“计算”任何东西。

模板元编程的运行是在编译期，它把编译器变成了元程序的解释器，可以把 C++的类型体系像面团一样捏来捏去，肆意打碎再任意组合起来，拥有近乎不可思议的“魔力”。

2.2 语法元素

虽然模板元编程使用的是标准的 C++语言，遵循同样的语法规则，但它在编程思想、编程范式等很多方面都与普通的运行时 C++程序有很大的不同。

模板元编程产生的元程序是在编译期执行的程序，操作的对象也不是普通的变量，因此不能使用运行时的 C++关键字（如 if、else、for），可用的语法元素相当有限，最常用的包括以下几种。

- enum、static，用来定义编译期的整数常量。
- typedef、using（C++11/14），最重要的元编程关键字，用于定义元数据。
- template，模板元编程的“起点”，主要用于定义元函数。
- “::”，域运算符，用于解析类型作用域获取计算结果（元数据）。

2.3 元数据

元编程可操作的数据就称为“元数据”（meta data），也就是 C++编译器在编译期可操作的数据，它是模板元编程的基础。

元数据都是不可变的，不能够就地修改，最常见的元数据是整数和 C++的类型（type）。

对于整数大家都很熟悉，普通程序在运行时也可以很容易地处理，但在模板元编程领域中的

元数据更多的是以类型（type）的面目出现。

这些元数据不是普通的运行时变量，而是如 int、double、class（非模板类）这样的抽象数据类型——这是模板元编程与普通运行时编程的一个最根本的不同点，也是元编程的威力所在和令初学者感到最困惑的地方（类型的计算）。

如果对元数据再进行细分归类，则元数据又可以分成整数元数据、值型元数据（int、double 等 POD 值类型）、函数元数据（函数类型）、类元数据（class、struct 等用户自定义类型）等。为了更明确地表述元数据的概念，本书后面的“元数据”一词特指非整数类型的元数据。

使用 typedef 关键字可以任意定义（声明）元数据，很像运行时的变量定义语句，例如：

```
typedef int meta_data1;                      //元数据meta_data1，值为int
typedef std::vector<float> meta_data2;  //元数据meta_data2，值为vector<float>
```

C++11/14 标准里的 using 关键字也可以达到同样的效果，但赋值语句的形式更清晰：

```
using meta_data1 = int;                       //元数据meta_data1
using meta_data2 = std::vector<float>;    //元数据meta_data2
```

不过因为 Boost 强调的是对 C++11/14 和 C++98 的兼容性，并没有在实现代码中使用 using，所以本书也仅使用 typedef 的形式，读者可自行在实践中尝试 using 的用法。

2.4 元函数

元函数（meta function）是模板元编程中用于操作处理元数据的“构件”，可以在编译期被“调用”，因为其功能和形式类似运行时的函数而得名，是元编程里的核心概念。

元函数实际上表现为 C++中的一个类或者模板类，它的通常形式是：

```
template<typename arg1, typename arg2, ...>          //元函数参数列表
struct meta_function                                   //元函数名
{
  typedef some-define type;                            //元函数返回的元数据
  //using type = some-define ;                         //C++11/14 的 using 形式
  static int const    value = some-int;                //元函数返回的整数
};                                                     //使用分号结束元函数的定义
```

编写元函数就像是编写一个普通的运行时函数，但形式上却是一个**模板类**。

- 函数参数列表圆括号“()”变成了模板列表声明的尖括号“<>”。
- 函数的形参变成了模板参数（即元数据），并且要使用关键字 typename 修饰。
- 因为不能使用运行时关键字，所以元函数不能像普通函数那样使用 return 返回计算结果，而是需要在元函数（模板类）内部用 typedef/using 定义一个名为 type 的类型（元数据）或者名为 value 的值作为返回[①]。
- 还要注意的一点是元函数最后以分号“;”结束，这是因为元函数实质上是一个类，C++ 的语法要求类定义需要分号。

进一步类比普通函数有助于我们更好地理解元函数。

元函数同样也有形参、实参的概念，元函数的形参就是模板参数列表中的模板参数，实参就是元函数被调用时的真正类型（元数据），元函数的调用就是编译器对模板的实例化计算依赖的类型。

元函数也可以没有返回值（即不定义内部类型 type），也可以有重载（模板特化/偏特化），也可以有缺省参数，也可以分为无参、单参、多参、可变参数等类别。但元函数没有普通函数参数传值、传引用的区别，也没有函数指针的概念。

如果有必要，元函数可以使用 typedef/using 关键字“返回”任意多个返回值，并且这些值没有顺序关系，能够用“::”来任意获取。

为表述方便，本书将只返回::type 的元函数称为标准元函数，而将返回多个元数据的元函数称为非标准元函数。

下面的代码是一个最平淡无奇的值元函数，它计算两个整数的和：[②]

```
template<int N, int M>                    //两个整数元数据
struct meta_func                          //元函数 meta_func
{
  static const int value = N + M;         //编译期计算整数之和，C++11/14 only
};
```

计算结果可以这样得到：

---

① type 和 value 名字仅仅是模板元编程领域的一个非强制性的约定，统一遵循这个约定会方便元函数的使用，用户无须查看源代码就可以确定元函数内部必定有名为 type 或 value 的内部定义。当然我们也可以自己另行建立一套自己的约定，但这通常没有必要。

② 这里的代码使用了 C++11/14 的新特性，可以直接定义类的静态成员变量，如果要兼容 C++98 可以使用<boost/config.hpp>里提供的工具宏 BOOST_STATIC_CONSTANT。

```
cout << meta_func<10, 10>::value << endl;  //使用::value 获取计算结果
```

读者需要注意元函数 meta_func 的执行过程，它的计算在编译的时候就已经完成了（即模板实例化），meta_func::value 实际上是一个编译期的常量，程序运行时不会有任何计算动作而是直接使用结果。如果这是一个大型的元函数，那么在编译期节约的计算工作量就会相当可观，可以显著提高程序运行时的效率。

因为元函数的计算发生在编译期，所以下面的代码不能成立：

```
int i = 10, j = 10;                  //两个运行时变量
meta_func<i, j>::value;              //错误，元函数无法处理运行时的普通数据
```

下面的代码示范了另一个元函数，它返回元函数参数列表中的第一个元数据：

```
template<typename T1, typename T2>   //两个形参，T1 和 T2
struct select1st                     //元函数 select1st
{
  typedef T1 type;                   //返回 T1，等价于 using type = T1;
};
```

请读者应谨记一点，元函数就是一个形式上很像函数的一个模板类，它用于计算（推导）类型。

2.5 元函数转发

元函数转发是模板元编程中一个经常用到的惯用法，相当于运行时的函数转发调用，但在模板元编程中则要使用 public 继承实现，模板参数传递给父类完成元函数的“调用”，这样子类会自动获得父类的::type 定义，同时也就完成了元函数的返回。

例如，下面的代码把元数据调换位置后转发给之前定义的元函数 select1st，相当于 select2nd 的功能：

```
template<typename T1, typename T2>
struct forward :                     //元函数转发，使用 struct 的默认 public 继承
  select1st<T2, T1>                  //注意这里，参数位置变动
{};                                  //元函数体内无实现代码
```

元函数转发等价于如下的写法：

```
template<typename T1, typename T2>
struct forward                                      //元函数，不使用转发
{
  typedef typename select1st<T2, T1>::type type;    //调用元函数计算
```

```
};
```

可见，元函数转发因为使用了类继承所以更加简洁易读。

2.6 易用的工具宏

模板元编程是一种全新的 C++编程范式，但却仍然使用原有的 C++语法，通篇的 typedef/using、template 关键字使元程序看起来尤如“天书”，对于初学者来说通常很难在短时间内适应这种转变。

作者在实际开发工作中使用宏定义了一些模板元编程的“伪关键字”，在一定程度上能够使模板元程序更加清晰易懂，更能够显示其元编程的本意，也确实收到了较好的效果。

这些自定义“伪关键字”均以 mp_开头（如果读者不喜欢这个前缀也可以改用 meta_或者其他更有意义的单词），由于使用了小写的形式让“伪关键字”代码从形式上看更像关键字，代码看起来更加漂亮：

```
#define mp_arglist      template            //元函数参数列表开始
#define mp_arg          typename            //元函数参数声明
#define mp_function     struct              //元函数定义
#define mp_data         typedef             //元数据定义

#define mp_return(T)    mp_data T type      //元函数返回
//#define mp_return(T)  using type=T        //C++11/14 风格的元函数返回
#define mp_exec(Func)   Func::type          //获取元函数返回结果
#define mp_eval(Func)   Func::value         //获取元函数返回值
```

这些宏分别把 template、typename、struct 和 typedef 这四个模板元编程中最常用的关键字进行了重命名。

- mp_arglist 表示元函数的参数列表开始。
- mp_arg 表示元函数的参数。
- mp_function 表示定义一个元函数。
- mp_data 表示定义一个元数据。
- mp_return/mp_exec/mp_eval 定义了元编程中约定的返回值用法，较原写法更清楚。

使用这些“伪关键字”，之前的元数据和元函数可以改写成如下的形式：

```
mp_data int meta_data1;                                   //元数据 meta_data1，值为 int

mp_arglist<mp_arg T1, mp_arg T2>                          //元函数参数是 T1 和 T2
mp_function select1st                                     //元函数 select1st
{
  mp_return(T1);                                          //返回 T1
};
```

很明显，使用了元编程“伪关键字”后元程序看起来更清楚，很容易地就能够把它们与普通的泛型代码区分开来。

不过读者需要注意的是由于宏预处理机制自身的“缺陷”，后三个“伪关键字”的作用有限，它们只能处理简单的参数，如果带有逗号“,”，那么模板类就会失效，但可以使用 Boost 库里的工具 BOOST_IDENTITY_TYPE 来解决。

2.7 应用示例

本节通过两个简单的例子来示范元编程的基本使用，仅作为演示，并不具备太多的实际价值，更多的元编程示例可见本书的后续章节。

编译期比较大小

首先我们来编写一个值元函数，它在编译期比较两个 int 型整数的大小，返回其中的较小者，代码非常简单：

```
mp_arglist<int L, int R>                                  //元函数有两个整数参数
mp_function static_min                                    //元函数 static_min
{
  static const int value = (L < R) ? L : R;               //?:操作符可用于编译期
};
```

元函数 static_min 用起来也很容易，例如：

```
assert((static_min<10, 20>::value == 10));                //编译期比较两个整数
```

虽然 static_min 的代码很简单，但如同标准库的 std::min() 一样，这样小而有用的元函数对于元编程也是非常重要的，Boost 专门在头文件<boost/integer/static_min_max.hpp>中提供了可以处理任意有符号整数和无符号整数的 min/max 元函数，原理相同。

操作类型的元函数

接下来我们编写一个元函数 demo_func，如果输入的元数据 T 是指针类型则返回 const T，否则返回 const T*。

因为元编程中不能使用 if-else 分支语句，所以我们的主要实现手段就是模板特化，不同的条件特化为不同的实现代码：

```
mp_arglist<mp_arg T>                         //单参元函数
mp_function demo_func                        //元函数 demo_func
{
  mp_return(const T*);                       //通常情况返回 const T*
};

mp_arglist<mp_arg T>
mp_function demo_func<T*>                    //对 T*情况进行模板特化
{
  mp_return(const T);                        //返回 const T
};
```

对元函数 demo_func 的验证可以使用第 3 章介绍的 is_same 元函数，它用来比较两个元数据是否相等：①

```
assert((is_same<mp_exec(demo_func<int>) , const int*>::value));
assert((is_same<mp_exec(demo_func<int*>), const int >::value));
```

这里的 assert 语句必须使用两对括号来包围断言，否则会因为宏无法识别模板语法，导致错误的逗号解析而造成编译失败。

2.8 总结

本章简要介绍了模板元编程的基础知识，包括元编程/元程序、元数据、元函数和元函数转发等基本概念。

---

① 示例代码中的 assert 完全可以使用 C++11/14 的 static_assert 或者 BOOST_STATIC_ASSERT 替代，而且会更明显地表明元函数编译期运行的意图，不使用 static_assert 完全是照顾版面布局的原因（单词太长了）。
另外需要注意的是 C++11/14 在名字空间 std 中也有元函数 is_same，用 using namespace std 打开标准库名字空间会造成名字冲突。如果出现这样的编译错误只需要简单地注释掉 using 语句，或者显式加上“std::”“boost::”名字空间限定。

元编程是一种超越普通程序的程序，在 C++中元编程是使用模板技术实现的，所以它又被称为模板元编程。元程序可以由 C++编译器在编译期解释执行，把部分计算量由运行时转移到编译时完成，提高程序的运行效率。但元编程更大的用途是类型推导，操纵 C++类型体系，能够实现普通程序根本无法实现的功能。

元数据是元编程的操作对象，可以是整数（含 bool）或任意的 C++类型。

元函数是元编程的核心，它表现为一个 C++模板类，我们必须使用元函数才能操作元数据。它以内部定义::type 或::value 返回计算的结果，并且可以使用 public 继承的方式实现元函数转发。这种函数式的计算方式初接触会觉得有些古怪，但只要我们理解了它操作类型的本质就能够逐渐掌握用法，如果用起来不顺手，那么还可以使用宏来定义“伪关键字”减轻不适应感。

理解了这些模板元编程基本概念后我们还可以深入考察其他既有的库，它们实际上已经或多或少地实践了模板元编程的理念。

- result_of ：它能够推导出可调用类型的返回值，以 result_of<T>::type 的形式给出，因此它就是一个元函数。
- unwrap_reference ：它能够解开类型的包装，获得真正的类型 T。
- call_traits ：它能够推导出最合适的调用参数类型。

这样的例子还有很多，它们的类型“推导”实际上就是模板元编程中元函数对元数据（类型）的计算。

在接下来的章节中，我们不会立即开始模板元编程，但会使用模板元编程的概念来阐述现代 C++和 Boost 程序库，会看到更多的元编程的应用展示。当然，如果读者感兴趣，那么也可以直接阅读第 13 章，连续完整地学习模板元编程。

第3章

类型特征萃取

在本章中，我们将要学习模板元编程工具 type_traits，它以库的方式实现了人们原本以为必须扩展 C++语言才能实现的类型特征萃取功能，是泛型编程和模板元编程所必需的基础设施。

目前 type_traits 已经成为了 C++11/14 标准的一部分（头文件<type_traits>，C++11.20.9），但 boost.type_traits 并不完全与标准一致，本章主要介绍符合标准的部分。

type_traits 位于名字空间 boost，需要包含头文件<boost/type_traits.hpp>，即：

```
#include <boost/type_traits.hpp>
using namespace boost;
```

3.1 概述

type_traits 库提供一组特征（traits）类——即元函数，可以在编译期确定类型（元数据）是否具有某些特征，例如是否是原生数组、是否是整数、是否重载了 operator<。此外它还提供了判断类型之间关系和操作类型的元函数，可以检查两个类型是否是同一个类型，或者为类型增添或移除 const、volatile 等修饰词。因为这些特征类都是元函数，所以它们都在编译期执行，不会存在任何运行时的效率损失。[①]

根据返回类型 type_traits 库里的元函数可分为以下两大类。

- 检查元数据属性的值元函数 ：以::value 返回一个 bool 值或者一个整数。

① Boost 自带文档中把 type_traits 库操作的对象称为 type（类型），但作者认为名字太过于平淡而不突出，因此依据模板元编程统称为元数据，这样有利于保持概念的一致性，希望读者阅读时注意。

- 操作元数据的标准元函数　　：对元数据进行计算，以::type 返回一个新的元数据。

type_traits 库中以 is_和 has_开头的元函数均属于值元函数，其他则属于标准元函数，但少数元函数也有例外。

根据元函数实现的功能 type_traits 库里的元函数可分为以下 7 类。

- 检查元数据的类别　　：均以 is_开头，都是值元函数。
- 检查元数据的属性　　：大部分以 is_和 has_开头，都是值元函数。
- 检查元函数之间的关系　　：均以 is_开头，都是值元函数。
- 检查操作符重载　　：均以 has_开头，都是值元函数。
- 转换元数据　　：都是标准元函数，返回转换后的类型。
- 解析函数元数据　　：都是非标准元函数。
- 用指定的对齐方式组合类型　：本书暂不做介绍；

在接下来的数节中，我们将参照 C++11/14 标准，以功能分类逐个介绍 type_traits 库里的元函数。

3.2　元数据类别

C++里有许多内建类型，同时还可以自定义任意类型，它们都是模板元编程里可以操作的元数据，使用 type_traits 库里的元函数可以对它们分门别类。

被检查的元数据前可以有 const、volatile 关键字修饰，元函数使用::value 返回 bool 类型的检查结果。

3.2.1　基本类别

基本类别（primary type categories）描述了 C++里最基本的类型体系。

检查简单类别

检查简单 C++类别的元函数有以下三个。

- is_integral<T>　　：检查 T 是否是 bool、char、int、long 等整型。

- is_floating_point<T> ：检查 T 是否是 float、double、long double 等浮点型。
- is_void<T> ：检查类型 T 是否为 void 类型。

示范这三个元函数用法的代码如下，其中的类型应视为元数据常量：

```
assert( mp_eval(is_integral<const char>));            //有 const 修饰
assert( mp_eval(is_integral<unsigned long>));         //有 unsigned 修饰
assert(!mp_eval(is_integral<int*>));                  //指针类型不是整型

assert( mp_eval(is_floating_point <double>));         //double 属于浮点类型
assert( mp_eval(is_floating_point<float>));           //float 属于浮点类型
assert(!mp_eval(is_floating_point<int>));             //整数类型不是浮点类型

assert( mp_eval(is_void<void>));                      //检查 void 类型
assert(!mp_eval(is_void<void*>));                     //void*指针类型不是 void
```

检查其他类别

接下来的八个元函数可检查其他的 C++类别。①

- is_array<T> ：检查 T 是否是一个原生数组（包括一维和多维数组）。
- is_class<T> ：检查 T 是否是一个 class 或者 struct。
- is_enum<T> ：检查 T 是否是一个枚举类型。
- is_union<T> ：检查 T 是否是一个联合类型。
- is_pointer<T> ：检查 T 是否是一个指针或函数指针类型，但不是成员指针。
- is_function<T> ：检查 T 是否是一个函数类型，但不是函数指针或引用。
- is_lvalue_reference<T> ：检查 T 是否是一个左值引用类型。
- is_rvalue_reference<T> ：检查 T 是否是一个右值引用类型。

下面的代码示范了其中部分元函数的用法（请读者注意其中左右引用的判断）：

```
assert(mp_eval(is_array<double[]>));                  //一维数组类型
assert(mp_eval(is_array<int[2][3]>));                 //多维数组类型

assert(mp_eval(is_class<struct dummy>));              //一个空类
```

① boost.type_traits 库暂时没有实现 C++14 标准里的 is_null_pointer 元函数。

```
assert(mp_eval(is_class<std::vector<int> >));          //标准容器类

assert(mp_eval(is_pointer<int*>));                      //整型指针
assert(mp_eval(is_pointer<int(*)(int)>));               //函数指针

assert(mp_eval(is_function<void(int,double)>));         //函数类型

typedef float& float_ref;                               //定义一个引用类型
assert(mp_eval(is_lvalue_reference<float_ref>));        //左引用
assert(mp_eval(is_lvalue_reference<float_ref&>));       //左引用的引用还是左引用
assert(mp_eval(is_lvalue_reference<float_ref&&>));      //右引用类型不变
assert(mp_eval(is_rvalue_reference<float&&>));          //右引用
```

检查成员指针类别

最后是两个专门用于识别类成员指针类型的元函数。

- is_member_object_pointer<T>　：检查 T 是否是指向成员变量的指针。
- is_member_function_pointer<T>　：检查 T 是否是一个成员函数指针。

示范这两个元函数用法的代码如下：

```
struct dummy                                     //一个简单的类
{
    int         x;                               //int 成员变量
    double      y;                               //double 成员变量
    void        func(){}                         //成员函数
};

assert(mp_eval(is_member_object_pointer<int dummy::* >));
assert(mp_eval(is_member_object_pointer<double dummy::* >));
assert(mp_eval(is_member_function_pointer<void(dummy::*)()>));
```

读者需要注意成员变量指针和成员函数指针类型的写法，它们与普通指针的差异较大，需要在“*”前有类域限定。

3.2.2 复合类别

在检查基本类别的元函数之上，type_traits 库又提供了七个检查复合类别（composite type categories）的元函数，它们相当于多个基本类别的组合，使用::value 返回 bool 类型的检查结果。

- is_reference<T> ：检查 T 是否是一个引用类型（左引用或右引用）。
- is_arithmetic<T> ：检查 T 是否是算术类型，相当于 is_integral<T> || is_floating_point<T>。
- is_fundamental<T> ：检查 T 是否是基本类型，相当于 is_arithmetic<T> || is_void<T>。
- is_compound<T> ：检查 T 是否是复合类型，即非基本类型，相当于！is_fundamental<T>。
- is_member_pointer<T> ：检查 T 是否是成员指针，包括指向数据成员和函数成员的指针，相当于 is_member_object_pointer<T> || is_member_function_pointer<T>。
- is_scalar<T> ：检查 T 是否是标量类型，即算术类型、枚举、指针和成员指针。
- is_object<T> ：检查 T 是否是实体对象类型，即引用、void 和函数之外的所有类型。

示范这些检查复合类别元函数的代码如下：

```
assert(mp_eval(is_reference<float&>));                    //左引用是引用
assert(mp_eval(is_reference<float&&>));                   //右引用也是引用
assert(mp_eval(is_reference<std::deque<int> const&>));    //容器常引用

assert(mp_eval(is_arithmetic<char>) );                    //char 是算术类型
assert(mp_eval(is_arithmetic<float volatile>));           //float 是算术类型

assert(!mp_eval(is_arithmetic<void const>));              //void 不是算术类型
assert(mp_eval(is_fundamental<void const>));              //void 是基本类型

assert(mp_eval(is_member_pointer<int(dummy::*)>));        //成员指针

assert(mp_eval(is_compound<std::string>));             //标准字符串是复合类型
assert(mp_eval(is_object<std::string>));               //标准字符串也是对象类型

assert(mp_eval(is_scalar<int>));                       //int 是标量类型
assert(!mp_eval(is_scalar<std::vector<int> >));        //标准容器不是标量类型
```

3.3 元数据属性

除了检查类别，type_traits 库里还有更多的用于获取元数据更细致的属性（type properties）的元函数。

这些元函数都是值元函数，以 is_和 has_开头的元函数使用::value 返回 bool 类型的检查结果，其他元函数使用::value 返回整数。

3.3.1 基本属性

检查基本属性的元函数可以判断元数据使用了哪些修饰词或语法元素。

检查基本的修饰词

- is_const<T> ：检查 T 是否被 const 修饰。
- is_volatile<T> ：检查 T 是否被 volatile 修饰。
- is_signed<T> ：检查 T 是否是有符号整数。
- is_unsigned<T> ：检查 T 是否是无符号整数。

检查数组的属性

- rank<T> ：如果 T 是数组，那么返回数组的维数，否则返回 0。
- extent<T,N> ：如果 T 是数组，那么返回数组第 N 个维度（从 0 计数）的值，否则返回 0。

下面的代码示范了这些元函数的用法：

```
typedef const volatile int cv_int;                  //定义一个元数据

assert(mp_eval(is_const<cv_int>));                  //有 const 修饰
assert(mp_eval(is_volatile<cv_int>));               //有 volatile 修饰
assert(mp_eval(is_signed<cv_int>));                 //是有符号整数

assert(mp_eval(rank<int[2][3]>) == 2);              //获取数组 int[2][3] 的维数
```

```
//因 extent 有多个元参数，mp_eval 失效，只能使用原始形式
assert((extent<int[2][3], 1>::value == 3));        //获取第二个维度
```

3.3.2 类相关属性

类（class/struct）是 C++里最重要的自定义类型，所以 type_traits 库提供了大量的元函数来检查类相关属性，较常用的有以下几个。

- is_pod<T>　　　　：检查 T 是否是一个 POD 类型[①]；
- is_empty<T>　　　：检查 T 是否是一个空类。
- is_abstract<T>　　：检查 T 是否是一个抽象类（有纯虚函数）。
- is_polymorphic<T>：检查 T 是否是一个多态类（有虚函数）。
- is_final<T>　　　：检查 T 是否是一个 final 类（无法被继承）。

此外 type_traits 库里还有其他很多元函数，可专门用于检查 class 的构造、析构等特殊成员函数，但因为数量太多而实际应用的较少，故这里不再一一列举，读者可参考 C++11/14 标准或者 Boost 文档详细了解。

下面的代码简要示范了其中部分元函数的用法：

```
assert(!mp_eval(is_pod<std::string>));                    //标准字符串不是 POD 类型
assert(mp_eval(is_empty<std::plus<int> >));               //函数对象是空类
assert(mp_eval(is_polymorphic<std::iostream>));           //标准流是多态的

struct x final {};                                        //定义一个 final 类
assert(mp_eval(!is_abstract<x>));                         //不是抽象类
assert(mp_eval(is_final<x>));                             //是 final 类，不可继承

//标准字符串的构造函数和拷贝构造函数可能会抛出异常
assert(!mp_eval(has_nothrow_constructor<std::string>));
assert(!mp_eval(has_nothrow_copy<std::string>));

//标准字符串的转移构造函数不会抛出异常
```

① POD 是术语 Plain Old Data 的缩写，但没有明确的定义。通常来说基本类型（is_fundamental<T>::value == true）都是 POD，而复合类型的 POD 则没有构造函数、析构函数、虚函数，内存布局是连续的，与 C 语言定义的类型等价。

```
assert(mp_eval(is_nothrow_move_constructible<std::string>));
assert(!mp_eval(has_trivial_move_constructor<std::string>));
```

3.3.3 操作符重载属性

type_traits 库提供了近 40 个检查元数据是否重载了某些操作符的值元函数，使用::value 返回 bool 类型的检查结果，但这些元函数不属于 C++11/14 标准，故在这里仅列举几个较有代表性的。

- has_greater<T> ：检查 T 是否重载了 operator>。
- has_less<T> ：检查 T 是否重载了 operator<。
- has_equal_to<T> ：检查 T 是否重载了 operator==。
- has_plus<T> ：检查 T 是否重载了 operator+。
- has_minus<T> ：检查 T 是否重载了 operator-。
- has_pre_increment<T> ：检查 T 是否重载了前置 operator++。

示范这些元函数用法的代码如下：

```
//检查 int 的操作符重载
assert(mp_eval(has_greater<int>));                  //支持 operator>
assert(mp_eval(has_less<int>));                     //支持 operator<
assert(mp_eval(has_equal_to<int>));                 //支持 operator==

//检查 string 的操作符重载
assert(mp_eval(has_greater<std::string>));          //支持 operator>
assert(mp_eval(has_equal_to<std::string>));         //支持 operator==
assert(mp_eval(has_plus<std::string>));             //支持 operator+
assert(!mp_eval(has_minus<std::string>));           //不支持 operator-

//检查 string 迭代器的递增操作符
assert(mp_eval(has_pre_increment<std::string::iterator>));
```

3.4 元数据关系

type_traits 库提供了四个计算元数据之间关系的元函数，它们都是两参值元函数，使

用::value 返回 bool 类型的检查结果。

- is_same<T, U> ：检查 T 和 U 是否是相同的类型。
- is_convertible<From, To> ：检查 From 是否可隐式转型为 To 类型。
- is_base_of<B, D> ：检查 B 是否是 D 的基类，或两者相同。
- is_virtual_base_of<B, D> ：检查 B 是否是 D 的虚基类，不属于 C++11/14 标准。

示范这四个元函数用法的代码如下：

```
mp_data int        meta_data1;                         //定义两个元数据
mp_data short      meta_data2;

assert((is_same<int, meta_data1>::value));             //int 与 meta_data1 相等
assert((is_convertible<meta_data2, int>::value));      //short 可转换为 int

assert((is_base_of<string, string>::value));           //string 自身比较，是基类
assert((is_base_of<ios_base, ostream>::value));        //IO 流继承体系

//IO 流继承体系可参见 11.1.1 节
assert((!is_virtual_base_of<ios_base, ostream>::value));
assert((is_virtual_base_of<basic_ios<char>, ostream>::value));
```

这四个元函数中最常用的是 is_same，它被用于在模板元编程中比较两个元数据是否相等。

3.5 元数据运算

之前我们看到的元函数都是值元函数，它们返回 bool 值或者整数，下面元函数将会真正开始对元数据（类型）进行计算，输入一个类型然后输出一个新的类型，随意地处理 C++的类型，从中可以初步体会元编程中**类型计算**的威力和魅力。

元函数处理类型的实际转换规则比较复杂，这里仅给出较一般的情形，更精确的转换规则请读者参考 C++11/14 标准或者 Boost 文档。

3.5.1 基本运算

对元数据的基本“运算”就是为 C++类型添加或者删除各种修饰，因为元数据是不可变的，

所以元函数会以::type 产生一个新的类型。

基本的元数据“加法”

“加法”为元数据 T 添加 const、volatile、指针和引用等修饰，返回变动后的新类型。

- add_const<T> ：返回 T const。
- add_volatile<T> ：返回 T volatile。
- add_cv<T> ：返回 T const volatile。
- add_pointer<T> ：返回 T*。
- add_lvalue_reference<T> ：对于对象或函数类型返回左值引用，通常是 T&，否则返回 T。
- add_rvalue_reference<T> ：对于对象或函数类型返回右值引用，通常是 T&&，否则返回 T。

示范这些元函数用法的代码如下：

```
mp_data mp_exec(add_const<int>) mdata1;                    //添加 const 修饰
assert((is_const<mdata1>::value));                         //新元数据是常数类型
assert((is_same<mdata1, int const>::value));               //比较相等

mp_data mp_exec(add_pointer<double>) mdata2;               //添加指针修饰
assert((is_pointer<mdata2>::value));                       //新元数据是指针类型
assert((is_same<mdata2, double*>::value));                 //比较相等

mp_data mp_exec(add_lvalue_reference<mdata2>) mdata3;//添加左引用修饰
assert((is_lvalue_reference<mdata3>::value));              //新元数据是左引用类型
assert((is_same<mdata3, double*&>::value));                //判断是否是指针的引用类型

mp_data mp_exec(add_lvalue_reference<void>) mdata4;   //为 void 添加左引用
assert((is_void<mdata4>::value));             //因为 void 不是对象类型，所以无变化
```

基本的元数据“减法”

与元数据的“加法”操作相反，“减法”操作可以移除元数据 T 的 const、volatile、指针和引用等修饰，返回变动后的新类型。

- remove_const<T> ：移除 T 的顶层 const 修饰。
- remove_volatile<T> ：移除 T 的顶层 volatile 修饰。
- remove_cv<T> ：移除 T 的顶层 const 和 volatile 修饰。
- remove_pointer<T> ：移除 T 的指针修饰（*）。
- remove_reference<T> ：移除 T 的引用修饰（&或&&），注意不区分左引用或右引用。

示范这些元函数用法的代码如下：

```
mp_data int const **& mdata1;                              //一个指针的指针的引用类型

mp_data mp_exec(remove_pointer<mdata1>) mdata2;            //移除指针修饰
assert((is_same<mdata2, mdata1>::value));                  //因为类型是引用所以不改变

mp_data mp_exec(remove_reference<mdata2>) mdata3;          //移除引用修饰
assert((is_pointer<mdata3>::value));                       //新元数据是指针类型
assert((is_same<mdata3, int const**>::value));             //比较相等

mp_data mp_exec(remove_pointer<                    //连续移除两个指针，元函数嵌套调用
      mp_exec(remove_pointer<mdata3>)>) mdata4;

assert((is_const<mdata4>::value));                         //新元数据是常类型
assert((is_integral<mdata4>::value));                      //新元数据是整型
assert((is_same<mdata4, int const>::value));               //比较相等

mp_data mp_exec(remove_const<mdata4>) mdata5;              //移除 const 修饰
assert((is_same<mdata5, int>::value));                     //新元数据是整型
```

在这段代码中，我们首先要注意第一个元函数的使用，因为元数据 mdata1 是一个引用类型而不是指针，所以元函数 remove_pointer 无效，直接返回元数据本身。接下来在移除顶层（最右边）的引用修饰后又连续调用了两次元函数 remove_pointer，这样就成功消去了 int const 的两个指针修饰，得到了不含引用和指针修饰的实际类型。

3.5.2 特殊运算

本节里的元函数针可对一些特殊类型的元数据进行运算。

处理算术类型

下面的两个元函数可用于处理算术类型元数据（is_arithmetic<T>::value == true），但不能处理 bool 类型。[①]

- make_signed<T> ：返回 T 相应的有符号整数类型，cv 修饰不变。
- make_unsigned<T> ：返回 T 相应的无符号整数类型，cv 修饰不变。

示范这两个元函数用法的代码如下：

```
mp_data short const mdata1;                             //定义元数据

mp_data mp_exec(make_signed<mdata1>) mdata2;            //添加符号
assert((is_const<mdata2>::value));                      //常量性不变
assert((is_signed<mdata2>::value));                     //有符号

//因为 mdata1 原本就是有符号数，所以二者相等
assert((is_same<mdata2, signed short const>::value));

mp_data mp_exec(make_unsigned<mdata2>) mdata3;          //去除符号
assert((is_same<mdata3, unsigned short const>::value));
```

处理数组类型

type_traits 库处理数组类型，主要是操作它的维度。

- remove_extent<T> ：移除数组的最顶层维度（降低一个维度）。
- remove_all_extents<T> ：移除数组的所有维度（变为 0 维的普通类型）。

这些元函数只能操作数组，对于非数组类型则不发生改变，示范代码如下：

```
mp_data int(mdata1)[5][7][9] ;                       //多维数组元数据的定义比较特殊
//using mdata1 = int[5][7][9];                       //使用 using 定义会更清晰一些

mp_data mp_exec(remove_extent<mdata1>) mdata2;       //移除顶层维度
assert((is_same<mdata2, int[7][9]>::value));         //比较相等

mp_data mp_exec(remove_all_extents<mdata1>) mdata4;  //移除所有维度
assert((is_same<mdata4, int>::value));               //得到数组元素类型
```

① type_traits 库里另有三个可以提升整型范围的元函数（promote 等），因为非标准故本书未介绍。

其他运算

最后是两个比较特殊的元函数。

- conditional<b, T, U> ：条件运算，类似“?:”操作，根据 b 的真假决定返回 T 或 U，它等价于 mpl::if_c（参见 13.3 节）。
- common_type<T,...> ：求多个类型的共通类型（类似于数字的最小公倍数）。

下面的代码示范了 common_type 用法：

```
typedef common_type<int, char>::type mdata1;      //计算 int,char 的共通类型
assert((is_same<mdata1, int>::value));            //得到类型 int

typedef common_type<int, double>::type mdata2;    //计算 int，double 的共通类型
assert((is_same<mdata2, double>::value));         //得到类型 double

//int 和 std::string 没有共通类型，发生编译错误（元程序运行失败）
typedef common_type<int, std::string>::type mdata3;
```

3.6 解析函数元数据

解析函数元数据的元函数 function_traits 不属于 C++11/14 标准，是一个非标准元函数，能够返回多个值，包括函数的参数数量、参数类型和返回类型，支持解析最多 10 个参数的函数。

function_traits<T>要求输入的元数据（类型）必须满足 is_function<T>，不能是函数指针或者引用。如果输入的不是函数类型，那么可以用元函数 remove_pointer<T>、remove_reference<T>来转换类型，否则会导致编译错误（即元程序执行失败）。

function_traits 的类摘要如下：

```
template <class T>
struct function_traits
{
   static const std::size_t  arity;           //返回函数的参数数量
   typedef some-define       result_type;     //返回函数的返回值类型
   typedef some-define       argN_type;       //返回函数第 N 个参数的类型
};
```

因为 function_traits 不是标准元函数，所以不能使用宏“伪关键字”来调用，必须使

用域操作符直接写出内部类型定义，示例代码如下：

```
mp_data void(mdata1)(int, std::string);                  //注意函数类型的定义方式
assert((is_function<mdata1>::value));                    //验证函数类型元数据

const size_t n = function_traits<mdata1>::arity;         //获得函数参数数量
assert((n == 2));

mp_data function_traits<mdata1>::result_type rtype;  //函数的返回类型
assert((is_void<rtype>::value));

mp_data function_traits<mdata1>::arg2_type a2type;   //第二个参数的类型
assert((is_same<a2type, std::string>::value));
```

从某种程度上说，function_traits::result_type 的效果与 std::result_of::type 相同，但 function_traits 只能处理函数类型，而 std::result_of 可以处理任意的可调用类型——函数、函数指针和函数对象。

3.7 实现原理

type_traits 库里的元函数虽然功能都比较简单，但其实现却比较复杂，而且还使用了预处理元编程和一些特别的技巧，这里仅以较简单的值元函数 is_integral 为例阐述其实现原理。

3.7.1 integral_constant

type_traits 库里的许多值元函数都使用了元函数转发技术，把元参数转发给元函数 integral_constant 进行计算，也就是说 integral_constant 是大多数值元函数的 public 基类。

integral_constant 的类摘要如下（有简化）：

```
template <class T, T val>                          //计算类型为 T，值为 val 的整数
struct integral_constant                           //元函数 integral_constant
{
   typedef integral_constant<T, val> type;      //定义自身为返回元数据
   typedef T                          value_type; //返回值的类型
   static const T                     value = val; //以::value 返回整数值 val
};
```

顾名思义，integral_constant 是计算整型常数的元函数，它以::type 返回自身作为元函数的计算结果，::value 返回整型常数，相当于把整数 val 做了一层元函数包装。

integral_constant 可以这样使用：

```
//元函数计算 int 型整数 10
assert((integral_constant<int, 10>::value == 10));

//元函数计算 char 型整数 0x30
static_assert(integral_constant<char, 0x30>::value == '0', "");

//元函数计算 short 型整数 100
static_assert(integral_constant<short, 100>::value == 100, "");
```

因为 type_traits 库中的大部分值元函数的计算结果是 bool 值，因此 type_traits 库又特别提供了两个针对 bool 元数据特化的无参元函数 true_type 和 false_type，它们的声明如下：

```
typedef integral_constant<bool, true >  true_type;
typedef integral_constant<bool, false>  false_type;
```

这两个元函数总返回 true 或者 false：

```
//直接使用::value 获得计算结果
static_assert(true_type::value == true, "");

//使用模板元编程"伪关键字" mp_eval 获得计算结果
static_assert(mp_eval(false_type) == false, "");
```

3.7.2 is_integral

is_integral 的实现使用了模板特化技术，对于非整数的类型元函数总是以::value 返回 false，实现代码如下[①]：

```
template< typename T >
struct is_integral : false_type {};                    //元函数转发，返回 false
```

随后 is_integral 对 bool、char、unsigned int、signed int 等数个整数类型进行了模板特化，其手法与 2.7 节的例子完全相同，例如：

```
template<>                                             //对 bool 类型特化
```

① 实际的 is_integral 代码使用了预处理元编程和复杂的条件编译，这里的代码做了适当的简化。

```
struct is_integral<bool> : true_type{};              //元函数转发，返回 true

template<>                                           //对 char 类型特化
struct is_integral<char> : true_type{};              //元函数转发，返回 true

......                                               //更多的整数类型特化
```

通过这样的模板特化，编译器在计算（实例化）is_integral 元函数的时候就可以实现依据参数分别处理：对于整数类型执行特化形式返回 true，对于其他的类型则返回 false。

type_traits 库里的其他值元函数的实现基本与 is_integral 类似，如果想要更深入研究它们的工作原理，那么就需要学习元编程库 mpl（第 13 章）。

3.8 应用示例

本节将使用几个例子来帮助读者进一步熟悉 type_traits 库和模板元编程。

3.8.1 conditional

我们用 type_traits 库改写 2.7 节的元函数 demo_func，不使用模板特化，而是使用元函数转发调用条件元函数 conditional：

```
mp_arglist<mp_arg T>
mp_function demo_func:                          //使用元函数转发
  conditional<                                  //使用条件元函数 conditional
      is_pointer<T>::value,                     //根据是否是指针类型执行分支
      typename add_const<                       //第二步：添加 const 修饰
          typename remove_pointer<T>::type      //第一步：移除指针操作符
          >::type,                              //::type 返回 const T
      typename add_pointer<                     //第二步：添加指针操作符
          typename add_const<T>::type           //第一步：添加 const 修饰
          >::type                               //::type 返回 const T*
  >{};                                          //元函数结束
```

使用 type_traits 元函数的实现明显比直接的模板特化实现要复杂些，它充分展现了模板元编程的函数本质，程序的实现都是通过函数的嵌套调用完成的，程序员需要在自己的头脑中维护一个“函数的堆栈”才能弄清楚它们的调用过程。

实际上 conditional 并没有做什么更多的工作，它使用了模板偏特化来简单实现：

```
template <bool b, class T, class U>
struct conditional { typedef T type; };          //true 情形返回类型 T

template <class T, class U>
struct conditional<false, T, U>                   //对 false 偏特化
{ typedef U type; };                              //返回类型 U
```

boost.mpl 里另有一个元函数 eval_if（参见 13.3 节），也可以达到同样的效果。它会自动计算参数列表里所有元函数的结果，不必再写::type，比 conditional 要方便一些，代码可以得到简化：

```
mp_arglist<mp_arg T>
mp_function demo_func:                            //使用元函数转发
  mpl::eval_if<                                   //使用 mpl 库里的元函数 eval_if
      is_pointer<T>,                              //根据是否是指针类型执行分支
      add_const<                                  //直接写元函数，不需要用 typename
          typename remove_pointer<T>::type        //这里还需要返回元数据
          >,                                      //不需要写::type
      add_pointer<                                //直接写元函数，不需要用 typename
          typename add_const<T>::type             //这里还需要返回元数据
          >                                       //不需要写::type
  >{};                                            //元函数结束
```

3.8.2 identity_type

C/C++的宏预处理器把逗号识别为宏参数分隔符，不能理解 C++模板语法的尖括号，所以在宏里使用带有逗号的模板类会导致参数解析错误，例如：

```
std::map<int,int> m;                              //一个标准关联容器
BOOST_FOREACH(std::pair<int,int> x, m){}          //宏中有两个逗号，编译错误
```

第二行代码会被预处理分解为三个参数，分别是：“std::pair<int”“int> x”和“m”，而 BOOST_FOREACH 要求是两个参数，所以导致编译失败。

identity_type 是 boost.utility 里的一个很小的工具，它提供一个宏 BOOST_IDENTITY_TYPE，把类型用一对圆括号包装起来解决了这个问题：

```
BOOST_FOREACH(BOOST_IDENTITY_TYPE((std::pair<int,int>)) x, m)        //编译正常
```

BOOST_IDENTITY_TYPE 的实现代码很简单：

```
#define BOOST_IDENTITY_TYPE(parenthesized_type) \
    boost::function_traits< void parenthesized_type >::arg1_type
```

它使用了元函数 function_traits，把宏的参数转换为一个函数，再用 arg1_type 返回，所以宏的参数必须要使用圆括号，否则宏展开时不能成为一个函数类型。

之前代码宏展开后的结果是：

```
BOOST_FOREACH(function_traits<void(std::pair<int,int>)>::arg1_type x, m)
```

function_traits::arg1_type 返回类型 std::pair<int,int>，现在宏中不存在多余的逗号，就可以正常编译了。

3.8.3 declval

declval()是 C++11/14 标准（C++11.20.2.4）中定义的一个特别的模板函数，用于简化未评估表达式（unevaluated operands）的定义，Boost 也实现了它：

```
template <typename T>
typename add_rvalue_reference<T>::type declval();
```

declval()没有也不需要有函数实现，它返回一个类型 T 的右引用（T&&），通常被用在一些表达式中配合 C++11 /14 的新关键字 decltype 推导类型，也可以直接调用成员函数而无须特别的对象构造操作（例如类没有缺省构造函数）。

type_traits 库里的元函数 common_type 使用 declval()来实现，计算两个类型的代码如下（做了适当简化）：

```
template < typename T, typename U>
struct common_type<T, U> {
    typedef decltype(declval<bool>() ? declval<T>() : declval<U>()) type;
};
```

common_type 在类内部定义了一个条件表达式（declval<bool>()? declval<T>() : declval<U>()），因为使用了 declval()所以是未评估的，近似于 true?t:u。但在编译期我们不需要关心具体的表达式值，只需要表达式的类型信息，所以编译期就会根据?:表达式的规则推导出这个表达式的类型——T 和 U 的共通类型，最后就可以用关键字 decltype 来获得这个类型。

对于更多类型的共通类型计算，common_type 使用了可变参数模板特性来递归实现：

```
template<typename T, typename U, typename... V>        //C++11/14 可变参数模板
struct common_type<T, U, V...> {
  typedef typename common_type<                        //递归调用 common_type
      typename common_type<T, U>::type, V...>::type type;
};
```

3.9 总结

traits 是模板元编程中的一个非常重要的概念，它可以萃取类型中的许多重要信息，利于我们在编译期提早做出决断。

本章我们重点讨论了 type_traits 库，它可以萃取类型的基本（但不是所有）信息，还顺便完整地探讨了 C++的类型系统。之后我们还会看到其他的 traits 库，如萃取指针信息的 pointer_traits（4.7 节）和萃取迭代器信息的 iterator_traits（5.3 节）。

type_traits 库里有上百个有用的元函数，可以检查元数据的类型和关系，提取各种属性，使用它能够丰富我们的编程词汇，更精确地描述 C++的类型系统，配合静态断言可以极大地保证程序的正确性。

type_traits 库还提供了一些编译期的运算元函数，可以增加 const、volatile 等类型修饰词，也可以执行简单的条件分支处理，这些都是进一步学习模板元编程的基础。

第4章

实用工具

本章将使用前两章的元编程知识研究一些功能比较简单，但实现原理却涉及 C++语言深层次概念细节的 Boost 组件。

首先讨论看似无用的“空类”，然后研究两个“智能操作符”check_delete 和 addressof，再讨论初始化问题、类型转换和指针。这些小工具使用了特别的技巧，有着近乎魔术般的神奇功能，希望读者阅读完本章后能够对 C++有更深入的认识。

4.1 compressed_pair

compressed_pair 库提供一个与 std::pair 非常相似的模板类 compressed_pair，同样能够容纳任意两个元素，但它针对空类成员进行了特别的优化，可以“压缩”pair 的大小。

compressed_pair 位于名字空间 boost，需要包含头文件<boost/compressed_pair.hpp>，即：

```
#include <boost/compressed_pair.hpp>
using namespace boost;
```

4.1.1 空类

compressed_pair 与 std::pair 非常相似，不同之处是它对“空类”成员做了特殊的处理，所以我们首先要了解什么是空类。

所谓的“空类”是一个 class/struct 类型，它没有非静态成员变量（静态成员不会增加类实例大小），也没有虚函数（会导致虚表指针），使用 type_traits 库的元函数表述就是

is_class<T> && is_empty<T>。

下面的代码定义了一个最简单的空类：

```
class empty1{};                                    //最简单的空类
```

乍看起来空类似乎没有什么用。的确，像 empty1 这样仅有类声明而没有任何其他东西的真正的“空类”确实没有用①，但还有另外许多很有用的“空类”：它们被编译器认为是“空类”，但实际上却不是“空类”，可以含有许多有用的功能，但类实例（instance）却不会占用内存空间——这些功能包括 typedef、静态成员变量、成员函数、友元函数、枚举等，起到对功能“打包”的作用。

例如下面的几个“空类”，它们虽然不含普通数据成员变量，但能够用在很多地方：

```
class empty2                                       //空类，含有一个静态成员变量
{
  static const int MAX = 100;
};

class empty3                                       //空类，含有成员函数
{
public:
  void print()
  {   cout << "this is a empty class." << endl;}
                                                   //还可以有其他的非虚成员函数
};
```

C++现实代码中有非常多的有用“空类”的例子，如标准库中的函数对象 greater、less、plus 等，它们都是“空类”，因为它们仅提供一个 operator()。许多 Boost 组件也都使用了大量作为技术手段的“空类”，例如 integer 库和 operators 库，还有模板元编程中使用的所有元函数。

虽然空类不含有任何数据成员，理论上不应占据内存，但实际上单独使用它时仍然需要占用一定的内存空间。因为 C++不允许存在 0 大小的对象，所以大多数 C++编译器会暗地里在类中插入一个 char 以使它具有至少 1 字节的大小②，这样一来下面的断言通常总成立：

---

① 纯粹的空类也不一定完全无用，在模板元编程中它可以作为类别的 tag 标记（如迭代器分类，见 5.1.2 节），或者是特殊的“哨兵”角色（如 tuple）。

② 空类的大小依据编译器的行为而不同，有的编译器可能会令空类增大至 sizeof(int)。

```
assert(sizeof(empty1)         == 1);          //空类的实例占用 1 字节内存
assert(sizeof(greater<int>)   == 1);          //空类的实例占用 1 字节内存
```

因此，如果我们编写的类含有类型为空类的成员变量，通常会导致增加一点点原本不应该存在的空间开销，对于 std::pair 这种简单的结构来说特别明显。

compressed_pair 正是为解决这个问题而来，它使用了模板元编程技术和多重继承来优化空类成员，可以“压缩”pair 的大小以节约空间。

4.1.2 类摘要

compressed_pair 是一个模板类，摘要如下：

```
template <class T1, class T2>
class compressed_pair
{
public:
   typedef T1   first_type;                    //第一个成员类型定义
   typedef T2   second_type;                   //第二个成员类型定义
   ...                                         //其他 typedef

   compressed_pair();                          //四种构造函数
   compressed_pair(first_param_type x, second_param_type y);
   explicit compressed_pair(first_param_type x);
   explicit compressed_pair(second_param_type y);

   first_reference         first();            //返回第一个成员
   first_const_reference   first() const;
   second_reference        second();           //返回第二个成员
   second_const_reference  second() const;

   void swap(compressed_pair& y);              //交换对象
};
```

compressed_pair 同 std::pair 一样，接受任意两个类型数据作为它的模板参数，并使用 typedef 定义了若干内部类型定义方便使用（即 traits），对于两个模板参数是同一类型（is_same<T1, T2>）的情况还进行了模板偏特化。

compressed_pair 不提供任何比较操作符重载，因此不能像 std::pair 一样在两个 compressed_pair 对象间执行比较操作，但这不是什么太大的问题，可以很容易地自行解决。

4.1.3 构造与赋值

compressed_pair 提供了以下四种形式的构造函数。

- 无参数的构造函数将缺省构造两个成员。如果成员是一个 POD 类型，那么不会被自动赋初值，需要特别注意。
- 单参数的构造函数将自动推导类型，把参数赋值给恰当的成员。如果两个成员的类型相同，那么参数会同时赋值给两个成员。
- 两参数的构造函数将使用参数分别初始化两个成员，行为与 std::pair 类似。

compressed_pair 的这四种形式的构造函数可以适应各种情况下的对象创建，比 std::pair 要么不指定初值、要么全指定初值的方式更为灵活。下面的代码示范了这四种构造函数的用法：

```
typedef compressed_pair<int, string> pair_type;  //简化类型定义

pair_type cp1;              //无参构造，第一个成员值不确定，第二个成员缺省构造为空串
pair_type cp2(10);          //单参构造，第一个成员有初值 10，第二个成员缺省构造为空串
pair_type cp3("hello");        //单参构造,第一个成员值不确定,第二个成员有初值"hello"
pair_type cp4(20, "pair");     //双参构造,第一个成员有初值 20,第二个成员有初值"pair"
```

compressed_pair 也支持拷贝构造和赋值操作：

```
pair_type cp5(cp2);                               //拷贝构造
cp1 = cp4;                                        //赋值操作
```

因为使用了 call_traits 库（参见 12.2 节）传递构造函数的参数，所以 compressed_pair 的模板参数可以是引用的：

```
int       i = 313;
string    s;
compressed_pair<int&, string&> cp(i, s);          //正确
```

4.1.4 用法

compressed_pair 访问成员需要使用函数形式的 first() 和 second()，它们返回 compressed_pair 内部成员的引用，这与 std::pair 直接访问内部成员的方式不同。但除了要加上 operator() 之外，compressed_pair 的访问成员用法与 std::pair 的

first/second 几乎相同：

```
compressed_pair<int, string> cp;                    //缺省构造
assert(cp.second().empty());                        //第二个成员是空串

cp.first()    = 10;                                 //可以当作左值被赋值
cp.second()   = "hello";

assert(10 == cp.first());                           //也可以当作右值
assert(!cp.second().empty());                       //调用成员的成员函数
```

compressed_pair 容纳非空类时与 std::pair 的差别不大，不过当 pair 的两个成员中存在空类时就能够起到“压缩存储空间”的优化效果，见下面的代码示例：

```
assert(sizeof(compressed_pair<int, empty1>) == sizeof(int));
assert(sizeof(std::pair<int, empty1>) > sizeof(int));
cout << sizeof(std::pair<int, empty1>)  << endl;           //输出 8(字节)

cout << sizeof(compressed_pair<empty1, empty2>) << endl;   //输出 1(字节)
cout << sizeof(std::pair<empty1, empty2>) << endl;         //输出 2(字节)
```

这段代码的前三行定义了具有一个空类的 compressed_pair 和 std::pair，compressed_pair 只有一个 sizeof(int) 的大小，空类被优化成了不占用空间的真正“空类”，而 std::pair 则因为字节对齐的原因大小变为两个 sizeof(int) 的大小。

代码的后两行定义了成员全为空类的 compressed_pair 和 std::pair，这时 compressed_pair 的大小为 1（字节），有一个空类的存储被优化掉了，而 std::pair 的大小为 2（字节）。

因此，如果用户对内存空间的使用十分敏感，而且程序中可能出现大量的空类时，就可以使用 compressed_pair。

4.1.5 实现原理

compressed_pair 的名字可能会给人以误解，误以为它使用了什么玄妙的压缩算法，而实际上，它并没有做任何的“压缩”动作，而是利用了编译器的空基类优化技术(empty base-class optimisation，简称 EBO)，也许名字叫 optimized_pair 更加恰当。

compressed_pair 库在 boost::details 子名字空间下定义了一个模板类 compressed_pair_imp，它是 compressed_pair 的真正实现，被用于私有继承，其声明

如下：

```
template <class T1, class T2, int Version>
class compressed_pair_imp;
```

库根据模板参数类型是否相同和两个成员是否为空这三个条件共定义了 3×2=6 个偏特化的 compressed_pair_imp 实现，再使用 type_traits 库的 is_same、is_empty、remove_cv 以及 details 子名字空间里的模板元函数 compressed_pair_switch 进行编译期元计算，决定 int 模板参数 Version 的值，从而最终确定使用哪个版本的 compressed_pair_imp。

由于偏特化的 compressed_pair_imp 已经知道了模板参数是否为空类，因此它就从空类 protected 继承，而不是作为成员变量，这样编译器就可以使用空基类优化技术来执行优化。

例如，对于第一个成员（T1）是空类的情况，compressed_pair 将使用偏特化的 compressed_pair_imp<T1,T2,1>，其实现摘要如下：

```
template <class T1, class T2>
class compressed_pair_imp<T1, T2, 1>                    //特化版本 1，T1 是空类
   : protected ::boost::remove_cv<T1>::type              //从 T1 继承，移除 cv 修饰
{
public:
   compressed_pair_imp(first_param_type x, second_param_type y)
      : first_type(x), second_(y) {}

   first_reference     first()    {return *this;}        //注意这里
   second_reference    second()   {return second_;}
private:
   second_type second_;                                  //只有第二个类型的成员变量
};
```

在这种情况下，compressed_pair 将只有一个成员变量，空基类 T1 原本所需的至少为 1 的存储空间将被编译器优化掉。

因为使用了继承来实现 compressed_pair，“空类”不是以成员变量的形式存在，所以 compressed_pair 不能像 std::pair 那样直接访问成员，只能通过函数以引用的方式来间接访问所谓的“成员”。

4.1.6 功能扩展

compressed_pair 与 std::pair 在名字和用法上都十分相似，但 compressed_pair 还缺少类似 std::make_pair() 的辅助函数和比较操作，如果需要，那么我们也可以自己

实现。

make_compressed_pair

仿照 std::make_pair()可以很容易地实现工厂函数 make_compressed_pair()，代码如下：

```
template<typename T1, typename T2> inline
compressed_pair<T1, T2>
make_compressed_pair(T1 t1, T2 t2)
{
  return compressed_pair<T1, T2>(t1, t2);
}
```

make_compressed_pair()很容易使用，配合 C++11/14 的 auto 将使 compressed_pair 的创建更简单，无须指定模板参数，例如：

```
auto cp1 = make_compressed_pair(10, "char*");        //auto 自动推断表达式类型
auto cp2 = make_compressed_pair(3.14, empty1());
```

实现比较功能

为 compressed_pair 添加 operator==函数重载可以实现比较功能，这要求两个成员都是可比较的——确切地说，应该要求非空类是可比较的，空类应该总是相等的：

```
//简单地执行比较操作，存在缺陷，无法比较空类
template<typename T1, typename T2>
bool operator==(const compressed_pair<T1, T2>& l,
                const compressed_pair<T1, T2>& r)
{
  return l.first() == r.first() && l.second() == r.second() ;
}
```

上面的代码可以执行简单的判等操作，但并没有处理空类的功能，真正实现完整的比较操作需要使用 type_traits 库里的元函数，仿造 compressed_pair 的实现用元编程做模板特化处理。

用元编程实现比较功能

在这里，我们将模仿 compressed_pair 的实现方式，使用 compressed_pair_switch

计算版本 Version，然后用模板函数对象特化来进行分支判断：[①]

```
template<typename P, int Version>       //模板参数 P 就是 compressed_pair
struct _compare                         //一个模板函数对象，缺省返回 false
{
    bool operator()(const P& l, const P& r)
    {  return false;}
};

template<typename P>
struct _compare<P,0>                    //版本 0，都不是空类，比较两个成员
{
    bool operator()(const P& l, const P& r)
    {  return l.first() == r.first() && l.second() == r.second() ; }
};

template<typename P>
struct _compare<P,1>                    //版本 1，T1 是空类，比较第二个成员
{
    bool operator()(const P& l, const P& r)
    {  return l.second() == r.second() ; }
};

...                                     //其他版本
```

比较操作符调用 compressed_pair_switch 计算版本，根据版本再决定调用哪个 _compare()特化形式，重新实现如下：

```
template<typename T1, typename T2>
bool operator==(const compressed_pair<T1, T2>& l,
                const compressed_pair<T1, T2>& r)
{
    typedef compressed_pair<T1, T2> pair_type;          //简化类型定义

    typedef                                             //typedef 开始元计算
      boost::details::compressed_pair_switch
        <T1, T2,
        boost::is_same<                                 //T1、T2 是否相同
```

① 不能简单地根据 Version 用 switch-case 来做条件判断，因为模板实例化时空类型是无法执行比较操作的。

```
            typename boost::remove_cv<T1>::type,
            typename boost::remove_cv<T2>::type
            >::value,
        boost::is_empty<T1>::value,                    //T1 是否为空类
        boost::is_empty<T2>::value                     //T2 是否为空类
      > version;                                       //得到元函数计算结果 version

    //完成元计算，调用特化的函数对象
    return _compare<pair_type, version::value>()(l,r);
}

template<typename T1, typename T2>
bool operator!=(const compressed_pair<T1, T2>& l,     //!=操作符重载
                const compressed_pair<T1, T2>& r)
{   return !(l == r); }                               //调用 operator==
```

现在 compressed_pair 就能够完全地支持空类和非空类的比较操作了，可以这样使用：

```
compressed_pair<int, double> cp1(10,0), cp2(10,0);
assert (cp1 == cp2);

compressed_pair<int, empty1> cp3(0), cp4(10);
assert(cp3 != cp4);

compressed_pair<empty1, empty2> cp5, cp6;
assert (cp5 == cp6);
```

4.2 checked_delete

checked_delete 是对 C++关键字 delete 的增强，可以在编译期保证 delete 或 delete[]删除的是一个指向“完整类型”（complete type）的指针，避免在运行时发生未定义行为，是一个更加“智能”的 delete。

checked_delete 位于名字空间 boost，需要包含头文件<boost/checked_delete.hpp>[①]，即：

```
#include <boost/checked_delete.hpp>
using namespace boost;
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

4.2.1 函数的用法

checked_delete 库包含两个函数和两个函数对象，分别如下所示。

- checked_delete　　　：用于删除普通指针。
- checked_array_delete：用于删除数组指针，相当于 delete[]。

首先，我们来了解函数形式的用法，模板函数 checked_delete()和 checked_array_delete()的声明如下：

```
template<class T> void checked_delete(T * p);
template<class T> void checked_array_delete(T * p);
```

正如名字所表达的，它们是“进行检查的 delete 操作”，基本功能等价于 delete 和 delete[]，用法也完全相同，只是我们需要把操作指针变量的 delete 表达式改成函数调用式。例如：

```
auto p1 = new int(10);                      //普通指针
checked_delete(p1);                         //删除普通指针

auto p2 = new int[10];                      //数组指针
checked_array_delete(p2);                   //删除数组指针
```

注意，checked_delete()必须用于删除普通指针，而 checked_array_delete()必须用于删除数组指针，千万不可误用，它们还没有智能到能够识别普通指针和数组指针的程度（这也是为什么使用两个函数的原因），正确地使用它们还是程序员的责任。

除了 int、double 等基本类型，checked_delete()和 checked_array_delete()当然也可以用来删除对象指针：

```
class demo_class                            //一个简单的空类
{
public:
  ~demo_class()                             //析构函数，输出提示信息
  {   cout << "dtor exec." << endl;   }
};

int main()
{
```

```
    auto p1 = new demo_class;                    //对象指针
    checked_delete(p1);                          //删除对象指针

    auto p2 = new demo_class[10];                //对象指针数组
    checked_array_delete(p2);                    //删除对象数组指针
}
```

在这段代码中，我们定义了一个简单的类，它带有“非平凡”的析构函数（non-trivial destructor），使用 checked_delete 会正确地删除它们，程序运行后会在控制台输出 11 行“dtor exec.”。

4.2.2 函数对象的用法

模板类 checked_deleter 和 checked_array_deleter 的声明如下：

```
template<class T>
struct checked_deleter
{
    typedef void        result_type;
    typedef T*          argument_type;
    void                operator()(T* x) const;
};

template<class T>
struct checked_array_deleter
{
    typedef void        result_type;
    typedef T*          argument_type;
    void                operator()(T* x) const;
};
```

checked_deleter 和 checked_array_deleter 重载了 operator()，因而可以像函数一样被调用，它们实际上仅是对同名函数加上了简单的类包装。

因为它们是函数对象，不具备自动推导模板参数的功能，因此在使用时必须用模板参数指明要删除的对象类型，否则会无法通过编译：

```
auto p1 = new demo_class;
checked_deleter<demo_class>()(p1);                //删除对象指针
```

```
auto p2 = new demo_class[10];
checked_array_deleter<demo_class>()(p2);          //删除对象数组指针
```

这段代码的功能与刚才的 checked_delete() 函数调用完全相同，注意函数对象的使用方式：必须先在模板参数中指定要删除的对象类型，然后用一对圆括号调用构造函数生成一个临时函数对象，最后才能使用 operator() 调用删除功能。

表面上看函数对象的用法似乎很麻烦（实际上也确实如此），但因为它们的定义完全符合 C++ 标准规范，故可以传递给那些需要函数对象的泛型代码，例如搭配标准库算法操作容器里的指针：

```
vector<demo_class*> v;                            //一个容纳指针元素的标准容器
v.push_back(new demo_class);                      //添加两个元素
v.push_back(new demo_class);

//调用 for_each 算法删除容器内的指针
for_each(v.begin(),v.end(), checked_deleter<demo_class>());
```

上面这段代码使用标准库容器 vector 保存了若干对象指针，随后使用标准库算法 for_each，传入 checked_deleter 函数对象，逐个安全地删除，这比使用循环遍历容器再用 delete 删除指针要方便得多。①

一点改进

checked_deleter 是一个模板类，使用时必须要指定模板参数，显得有些麻烦。我们可以自定义一个类似的函数对象，它是非模板类，但有模板成员函数，所以能够自动推导模板参数：

```
struct my_checked_deleter
{
    typedef void result_type;                     //返回类型定义

    template<class T>
    void operator()(T* x) const                   //模板成员函数
    {
        boost::checked_delete(x);                 //调用 checked_delete 函数
    }
```

① 在代码中的 for_each 算法中，我们也可以直接传递函数，但同样需要指明它的模板类型，用法如下所示：for_each(v.begin(), v.end(), checked_delete<demo_class>)。

```
};
```

my_checked_deleter 可以如 checked_deleter 一样工作，但省去了写模板参数的麻烦：

```
auto p = new int(10);
my_checked_deleter()(p);                        //无须使用模板参数指明类型

vector<int*> v;
v.push_back(new int(10));
for_each(v.begin(), v.end(),
      my_checked_deleter());                    //无须使用模板参数指明类型
```

4.2.3 带检查的删除

初看上去似乎 checked_delete 与关键字 delete 没有什么不同，它仍然需要使用 delete，而且的确——之前代码中的 checked_delete 函数调用完全可以用关键字 delete 替换，运行结果不会有任何差异。

但 checked_delete 并不完全等同于 delete 关键字，它在 delete 操作之外增加了对不完整类型的检查，能够更好地保证代码的正确性。

所谓不完整类型（incomplete type）是指仅有声明而没有定义的类，通常见于类的前向声明，例如：

```
class demo_class;                      //一个不完整类型，只有声明而没有具体定义
```

对一个不完整类执行 delete 操作会导致析构函数未被执行，从而引发未定义行为，见下面的示例代码：

```
class demo_class;                      //前向声明，不完整类

void do_delete(demo_class* p)          //一个简单的删除函数
{ delete p;}                           //调用 delete 操作符

class demo_class{ ...};                //这里是类的完整定义，同前

int main()
{
  auto p = new demo_class();           //一个对象指针
```

```
    do_delete(p);                          //将导致未定义行为
}
```

这段代码在 GCC 下编译时会引发多个警告：

```
warning: possible problem detected in invocation of delete operator
warning: 'p' has incomplete type
warning: forward declaration of 'class demo_class'
note   : neither the destructor nor the class-specific operator delete will be
       called, even if they are declared when the class is defined
```

这些警告信息清楚地表明了：使用 do_delete()删除指针时不会调用 demo_class 的析构函数，因为 demo_class 是一个前向声明的不完整类，此时编译器还不知道它的析构函数。程序运行后没有任何输出，因为析构函数没有被调用，如果 demo_class 是一个具有复杂内部结构的类，那么就可能会导致资源未释放等未定义行为。

由于这段程序的代码量很少，所以我们能够很容易地查找到警告所在的位置并发现问题，但如果是在一个较大的工程中，头文件的引用和类的前向声明比较复杂，那么这样的警告就很容易被有意或者无意地忽略掉，或者被淹没在其他的错误与警告中无法发现。并且更有可能的是，有的编译器并不对此提出警告。

使用 checked_delete 可以避免这种微妙的问题——如果要删除的指针指向的不是一个完整类型，将引发编译错误：

```
class demo_class;                          //前向声明，不完整类，完整定义在别处

void do_delete(demo_class* p)
{  checked_delete(p);}                     //改用 checked_delete

int main()
{
    auto p = (demo_class*)(1996);          //强制转换一个指针地址
    do_delete(p);                          //编译错误
}
```

上面的代码无法通过编译，通过编译器提供的错误信息，我们可以很容易地发现删除不完整类型的错误，从而消灭未定义行为。

4.2.4 实现原理

checked_delete 的原理相当简单，其全部实现代码如下：

```
template<class T> inline void checked_delete(T * x)
{
    typedef char type_must_be_complete[ sizeof(T)? 1: -1 ];
    (void) sizeof(type_must_be_complete);
    delete x;                                   //调用 delete 操作符删除指针
}
```

它通过 typedef 定义了一个数组类型，其大小由要被删除的类型 T 确定。如果 T 是一个完整类型，那么 sizeof(T)表达式的结果是一个正整数，数组大小为 1；如果 T 是一个不完整类型，那么 sizeof(T)表达式的结果是 0，数组大小为-1。但 C++中数组的定义是不允许为负数的，所以会引发一个编译错误。

checked_array_delete 的实现与之类似，只不过最后的语句是 delete[] x，因为它被用于删除数组指针。

作为一个示范，我们也可以使用 static_assert 和第 3 章介绍的 type_traits 库来实现自己的 checked_delete，代码如下：

```
template<class Pointer> inline                  //模板参数是指针类型
void my_checked_delete(Pointer p)               //一个自定义的 checked_delete
{
  BOOST_STATIC_ASSERT(is_pointer<Pointer>::value);   //要求是指针类型

  typedef typename remove_pointer<Pointer>::type T;  //元函数计算值类型

  BOOST_STATIC_ASSERT(is_object<T>::value);    //要求必须是可删除的对象
  BOOST_STATIC_ASSERT(!is_array<T>::value);    //要求不能是数组类型
  BOOST_STATIC_ASSERT(sizeof(T) > 0 );         //同 checked_delete 的编译期断言

  delete p;                                    //删除指针
}
```

my_checked_delete() 函数的实现手法与 checked_delete 类似，它使用

BOOST_STATIC_ASSERT 宏在编译期断言模板类型 T 必须是一个完整类型，并增加了一些新的检查条件，效果与 checked_delete()相同。

不过读者需要注意的是，检查数组的断言（!is_array<T>）实际作用不是很大，对于指向一维数组的指针来说移除指针修饰后的类型仍然是 T，但如果指针是一个指向多维数组的指针，那么!is_array<T>可以正常工作：

```
auto p = new int[2][2];                          //多维数组指针
my_checked_delete(p);                            //发生编译错误，静态断言生效
```

4.2.5 使用建议

checked_delete 被声明为内联函数，而且函数内部仅使用了编译期的 typedef，没有任何多余的可执行代码，因此它的性能与原始 delete 相比没有任何差异，并且它命名清楚，易于维护，所以可以完全替代 delete 关键字在任何地方使用。

checked_delete 的用法非常简单，但作为库用户的我们最好应当少使用，因为直接操作原始内存很容易发生各种难以察觉的错误。多数情形下我们应该改用智能指针，例如 unique_ptr 和 shared_ptr，它们在内部调用了 checked_delete，可以自动地管理指针的生命周期，而且是异常安全的。

例如，下面的代码使用 shared_ptr 来管理对象指针：

```
shared_ptr<demo_class> sp(new demo_class);
...         //任意操作，无论发生什么，指针总会被正确删除，无须使用 delete
```

如果因为某些原因而不能使用智能指针，必须手工管理内存，那么就请使用 checked_delete，它可以保证在编译期就发现隐藏的错误。

4.3 addressof

addressof 是对 C++取地址操作（&）的增强，因为 C++允许重载 operator&，所以有时候 operator&会被程序员“欺骗”，但 addressof 总能够获取到操作对象的真实地址。它已经被收入 C++11 标准（头文件<memory>，C++11.20.6）。

addressof 位于名字空间 boost，需要包含头文件<boost/utility/addressof.

hpp>[1]，即：

```
#include <boost/utility/addressof.hpp>
using namespace boost;
```

4.3.1 用法

addressof 是一个模板函数，其声明如下：

```
template <typename T> T* addressof(T& v);
```

addressof 与 checked_delete 类似，是一个“智能取地址操作符”，它的用法很简单，可以像通常的&操作符一样使用。例如：

```
int i;                                          //整数
string s;                                       //标准库字符串
assert(&i == addressof(i));                     //断言取地址相等
assert(&s == addressof(s));                     //断言取地址相等
```

addressof 不仅可以操作普通变量，也可以操作数组和函数：

```
int a[10];
assert(&a     == addressof(a));                 //取数组地址
assert(a + 1 == addressof(a[1]));
assert(&printf == addressof(printf));           //取函数地址
```

对于重载了 operator&的类，直接使用&操作符会失效（因为这时实际上调用了重载的 operator&()函数），但 addressof 仍然可以获得变量的真实地址，请看下面的代码：

```
class dont_do_this                      //演示用，最好不要重载 operator&
{
public:
    int x,y;
    size_t operator&() const            //重载 operator&
    {   return (size_t)&y; }            //返回成员变量 y 的地址
};

int main()
{
    dont_do_this d;
    assert(&d != (size_t)addressof(d));    //(1)
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

```
    assert(&d == (size_t)&d.y);                   //(2)
    dont_do_this *p = addressof(d);               //(3)
    p->x = 1;
}
```

类 dont_do_this 重载了 operator&，因此对它的实例调用&操作符将不会返回实例的内存地址，而是改为调用重载的 operator&函数。dont_do_this 通过这种方式“隐藏”了自己的真正地址，而是返回了内部成员变量 y 的地址。

代码行(1)分别使用 operator&和 addressof 来获取变量的地址，因为 dont_do_this 重载了 operator&，因此&d 实际上获得的是 d.y 的地址，代码行(2)更清楚地证明了这一点。

而 addressof 则不受重载 operator&的影响，它总能获取真实的地址，因此代码行(3)是正确的。如果使用&d 来赋值指针变量 p，那么会得到一个编译错误，因为 operator&返回的是一个 size_t 类型，而不是地址[①]。当然我们也可以用强制类型转换把返回值转换为指针地址，但这样做得到的将是一个错误的内存位置，会导致未定义行为。

有的时候某些类甚至把 operator&声明为私有的，通常这种情况下无法使用&来获得变量的地址，但 addressof 同样不受影响。

```
class danger_class                              //不允许调用 operator&的类
{
    void operator&() const;                     //私有的 operator&
};

danger_class d;
// cout << &d;                                  //将导致编译错误，无法调用私有成员函数
cout << addressof(d);                           //正确，获取真实地址
```

4.3.2 实现原理

addressof 并没有应用什么特别的技巧，仅仅是使用了复杂的转型操作，核心实现代码如下：

```
reinterpret_cast<T*>(&const_cast<char&>(
    reinterpret_cast<const volatile char &>(v)));
```

代码中的类型 T 是 addressof 的模板类型参数，v 是 T 的一个引用。addressof 先使用了 reinterpret_cast 转型操作符把 v 先强制解释成 char 类型，然后再将其重新解释成 T*

① 实际上，operator&不一定非要返回一个数值，可以是任何类型。

类型，这样就得到了变量的真正地址。

因为多次使用了运行时转型，所以 addressof 的运行效率没有原始的 operator&效率高，但这点损失通常是微不足道的。

4.3.3 使用建议

为类重载 operator&不是一个非常明智的做法，很少有人会这样做，但这种可能性是存在的。

基于“害人之心不可有，防人之心不可无”的古训，我们在编写代码时应尽量做到不重载 operator&。如果怀疑某个类可能重载了 operator&，那么就使用 addressof 来获取对象的真实地址。

作为一个库作者，必须应对各种可能的情况，因此应当使用 addressof；而作为一个库用户和应用开发者，则应当编写合理的、规范的代码，消除 addressof 的应用可能。

4.4 base_from_member

有时候基类需要由派生类的成员变量来初始化，但通常的写法因 C++的类初始化顺序要求而不能正确实现，因为基类必须在派生类之前完成初始化，而那时派生类的成员是未定义的，解决方法是把派生类的成员移动到另一个辅助基类中。base_from_member 使用多重继承和模板技术提供了这个用成员来初始基类的惯用法。

base_from_member 位于 boost 名字空间，需要包含头文件<boost/utility/base_from_member.hpp>[①]，即：

```
#include <boost/utility/base_from_member.hpp>
using namespace boost;
```

4.4.1 类摘要

base_from_member 是一个很简单的类，摘要如下：

```
template < typename MemberType, int UniqueID = 0 >
class base_from_member
{
```

① 也可以直接包含<boost/utility.hpp>，它内含数个小工具的实现。

```
protected:
    MemberType  member;                          //成员变量

    base_from_member();                          //构造函数
    base_from_member( T1 x1 );                   //单参构造函数
    base_from_member( T1 x1, T2 x2 );            //双参构造函数
    ...                                          //其他构造函数
};
```

base_from_member 有两个模板参数：MemberType 是它的数据成员类型，UniqueID 则是一个起标记作用的整数，用来在使用多个 base_from_member 时进行区分，默认值是 0。

base_from_member 有一个保护数据成员，命名为 member，它的类型就是模板参数中的 MemberType。因为被声明为 protected，所以 member 可以被派生类任意使用，相当于把类的成员用法转化为了继承用法。

默认情况下，base_from_member 有 11 个构造函数，最大支持 10 个参数，这些参数被用来在构造时初始化 member 成员变量。构造函数参数的数量是可以定制的，base_from_member 使用了预处理元编程库 preprocessor，只需要在包含头文件前定义宏 BOOST_BASE_FROM_MEMBER_MAX_ARITY 即可，例如：

```
#define BOOST_BASE_FROM_MEMBER_MAX_ARITY  2      //注意这里
```

将使 base_from_member 最多只有三个构造函数，支持最多两个参数。

4.4.2 用法

先来看一下用派生类的成员来初始基类的通常写法，下面的代码不能正确实现编写者的目的：

```
class base                                    //基类
{
public:
    base(complex<int> c)                      //使用标准库的复数初始化
    {
        cout << "base ctor" << endl;
        cout << c << endl;                    //输出复数的值
    }
};                                            //基类定义结束
```

```
class derived:public base                          //派生类
{
  complex<int> c;                                  //派生类的复数成员
public:
  derived(int a, int b):c(a, b),base(c)           //初始化成员和基类，有错误
  {
      cout << "derived ctor" << endl;
      cout << c << endl;                           //输出复数的值
  }
};                                                 //派生类定义结束

int main()
{
  derived d(10, 20);                               //创建一个实例
}
```

程序的运行结果可能是这样：

```
base ctor
(2041140992,32534)                                 //数值不确定
derived ctor
(10,20)
```

很明显，基类 base 没有被正确初始化，因为在 base 构造时派生类的成员 c 还没有被初始化，其值是未定义的，如果是在真实的代码中这将导致灾难性的后果。

使用 base_from_member 可以很容易地解决这个问题，只需要把派生类的成员变量声明改为使用 base_from_member 的继承方式，把需要使用成员初始化的基类放在继承列表的最后即可：

```
class derived:
  private base_from_member<complex<int> >,          //声明成员变量
  public base                    //从基类派生，应在 base_from_member 之后
{
  //complex<int> c;                                 //不直接声明成员变量
  //typedef 简化关于 base_from_member 代码的编写
  typedef base_from_member<complex<int> > pbase_type;
public:
  derived(int a, int b):pbase_type(a, b),base(member)    //初始化
  { cout << member << endl;}                  //注意成员变量的名字是 member
};
```

派生类 derived 与前一个版本具有少量但关键的变化。

首先，它必须取消原来的成员变量 c 的声明，而改用 base_from_member 来间接声明成员变量。注意在继承时我们使用了 private 而不是 public，这将使 base_from_member 的 member 成员变量在 derived 中成为 private。因为 base_from_member 的名字过长，所以我们最好定义辅助类型来简化代码，通常这个辅助类型命名为 pbase_type。

这样，在 derived 的构造函数的初始化列表中我们就可以先初始化 base_from_member 定义的辅助基类 pbase_type，因为声明的顺序在前，所以它将先于 base 初始化，从而使成员变量 member 拥有正确的初值，随后基类 base 使用 member 也将被正确初始化。

修改后的代码运行结果如下：

```
base ctor
(10,20)
derived ctor
(10,20)
```

4.4.3 进一步的用法

因为 base_from_member 的成员变量被命名为 member，当派生类需要使用多个成员变量来初始化基类时就会存在名字冲突，这时必须要使用基类的名字来限定，使用起来有些麻烦。

假设再增加一个基类 base2，它需要使用两个 string 变量来进行初始化：

```
class base2
{
public:
  base2(string &x, string &y)
  {
      cout << "base2 ctor" << endl;
      cout << x <<  y << endl;
  }
};
```

那么，如果 derived 要从 base 和 base2 同时继承的话，代码应该如下：

```
class derived:
  private base_from_member<complex<int> >,      //第一个成员
  private base_from_member<string, 1>,          //使用第二个模板参数
  private base_from_member<string, 2>,          //用于区分不同的类型
  public base, public base2                     //最后是真正的基类
```

```
{
  typedef base_from_member<complex<int> >    pbase_type;
  typedef base_from_member<string, 1>        pbase_type1;
  typedef base_from_member<string, 2>        pbase_type2;
public:
  derived(int a, int b):
    pbase_type(a, b),
    pbase_type1("str1"),pbase_type2("str2"),
    base(pbase_type::member),                //用基类名字限定来使用成员变量
    base2(pbase_type1::member, pbase_type2::member)
  {
      cout << "derived ctor" << endl;
      cout << pbase_type::member << endl;
  }
};
```

因为使用了大量的多重继承，这段代码明显比之前的代码要难理解一些，写法也显得有点复杂和冗长，特别是使用 member 成员变量的时候，必须要使用基类名来限定。

因此，我们应当尽量避免使用多于一个的 base_from_member 用法，否则将使代码难以维护。如果确实存在一个需要使用成员变量进行复杂初始化的类，那么最好重新设计类的结构，或者根据 base_from_member 的工作原理自己手工编写一个辅助类。

例如，采用手工编写辅助类解决上面的派生类问题的代码如下所示：

```
class pbase_type                                    //手工编写的辅助类
{
protected:
  complex<int> cp;                                  //成员变量都移动到这个辅助类中
  string x,y;
  pbase_type(int a, int b, string c, string d):    //构造初始化
      cp(a,b), x(c), y(d)
  {}
};

class derived: private pbase_type, public base, public base2
{
public:
  derived(int a, int b):
    pbase_type(a, b, "str1", "str2"),               //初始化成员
    base(cp), base2(x, y)                           //初始化基类
```

```
    {
        cout << "derived ctor" << endl;
        cout << cp << endl;
    }
};
```

这个解法比使用 base_from_member 要好看易理解得多。

4.5 conversion

C++是一种静态强类型语言，任何变量都必须有一个类型，不同类型的变量相互操作时必须进行隐式或显式类型转换。隐式类型转换通常由编译器自动完成，而显式类型转换则通常由程序员手工强制完成。

C++提供两种显式类型转换语法：一种是继承自 C 语言的“老”语法，用一对圆括号指明转换的类型；另一种是在 C++标准中定义的新式转型操作符，如 static_cast、dynamic_cast，它们比老式语法更可读，更清晰地表明了转换的意图、不容易出错，而且也很容易使用文本处理工具分析处理。

conversion 库针对 C++标准中的转型操作符的缺陷进行了改进，能够更安全、更清晰地进行多态对象间的转型，库中的另一个组件 lexical_cast 还可以进行字面量的转换（lexical_cast 已经在推荐书目[3] 做过介绍）。

conversion 位于名字空间 boost，需要包含头文件<boost/polymorphic_cast.hpp>，即：①

```
#include <boost/polymorphic_cast.hpp>
using namespace boost;
```

4.5.1 标准转型操作符

C++标准（C++11.5.2）为显式类型转换定义了四个新的转型操作符：const_cast、static_cast、dynamic_cast 和 reinterpret_cast，它们被用于替换老式的显式类型转换语法，能够避免许多任意转型引起的潜在错误。在学习 conversion 库之前，我们有必要先

① 头文件<boost/polymorphic_pointer_cast.hpp>里另外提供 polymorphic_pointer_downcast 和 polymorphic_pointer_cast 两个转型函数，风格与 4.7.2 节的 static_pointer_cast 类似，因功能完全相同故本书不做介绍。

了解这四个 C++标准中的转型操作符。[①]

- const_cast ：用于增加或者去除 const、volatile 修饰，此外不能执行其他任何转换操作。
- static_cast ：可以显式执行所有编译器可执行的隐式类型转换操作，但不能执行多态类的交叉转型。
- dynamic_cast ：用于多态对象（即存在虚函数的对象）间的类型转换，可以向上或者向下转换对象的类型。
- reinterpret_cast ：对目标的内存二进制位进行低层次的重新解释。

这四个转型操作符中最常用的是前三个，而第四个 reinterpre_cast 对有些读者来说可能较为陌生。

reinterpret_cast 的作用很接近老式的显式类型转换，可以变更被转换对象的含义。它最常见的应用场景是把一个已经失去类型信息的指针（例如 void*）"重新解释"，使其重新获得正确的类型。例如，下面的代码调用了 UNIX 的 dlsym() 函数加载动态链接库中名为"SocketBind"的接口，将返回的指针转换为所需的函数指针：

```
typedef int (*FuncBind)();                          //函数指针类型定义
FuncBind fbind = reinterpret_cast<FuncBind>         //类型转换
        (dlsym(fd, "SocketBind"));
```

但应当尽量少用 reinterpre_cast，误用它有时候会得到匪夷所思的结果，例如：

```
int i = 100;
cout << static_cast<double>(i)<< endl;              //正确
cout << *reinterpret_cast<double*>(&i) << endl;     //错误
```

最后一行代码的输出将是一个莫名其妙的浮点数-9.25596e+61。

const_cast 和 static_cast 的用法都很简单，下面将重点讨论 dynamic_cast，这也是 conversion 库关注的焦点。

4.5.2 多态对象的转型

多态对象间的类型转换可以分为两类：从基类到派生类的向下转型（downcast）和从一个

① 提醒读者注意，这些转型操作符不仅可以操作指针，也可以操作引用。

基类到另一个基类的交叉转型（crosscast）。

dynamic_cast 操作符可以同时执行这两种不同含义的转型操作，所以有时候会导致使用混淆。dynamic_cast 的另外一个“缺陷”是转型失败后的行为不一致：引用转型失败时会抛出异常 bad_cast，而指针转型失败时则返回一个空指针（nullptr）。当然，C++标准做出这种设计是有其特定考虑的，因为这可以把转型与测试放在一个表达式中完成，但在很多情况下（特别是泛型编程）会造成一些不大不小的麻烦。

假设有下面的一个多态类继承体系：

```
struct base1                                    //第一个基类
{ virtual ~base1(){};};                         //注意，必须有虚函数

struct base2                                    //第二个基类
{ virtual ~base2(){};};                         //注意，必须有虚函数

struct derived :public base1, public base2     //多重继承
{ virtual ~derived(){};};
```

下面的代码演示了 dynamic_cast 操作符的各种用法：

```
base1 *p = new derived;                         //一个多态对象指针
derived *pd = dynamic_cast<derived*>(p);        //向下转型
base2 *pb2 = dynamic_cast<base2*>(p);           //交叉转型

string *ps = dynamic_cast<string*>(p);          //对指针转型
assert(!ps);                                    //失败，结果是空指针

try
{
  string& s = dynamic_cast<string&>(*p);        //对引用转型
}
catch (bad_cast& e)
{
  cout << e.what() << endl;                     //失败，抛出异常
}
```

在很多情况下，dynamic_cast 对指针和引用的转型行为不一致会引起程序员的困扰，特别是在指针转型时必须编写额外的代码来检查空指针（虽然出现的几率很小），或者使用不太常见的 if 语句临时变量，很容易因漏写误写检查代码而导致安全隐患。

conversion 库针对 dynamic_cast 提供了增强的转型操作，用两个更安全命名更清晰的多态转型操作 polymorphic_downcast 和 polymorphic_cast 区分了多态对象的向下转型和交叉转型。

4.5.3 polymorphic_downcast

polymorphic_downcast 提供对多态对象指针的向下转型功能，它使用 static_cast 提供高效的转型操作，但不具备 dynamic_cast 的错误检测功能。

polymorphic_downcast 是一个很小的模板函数，其实现代码如下：

```
template <class Target, class Source>
inline Target polymorphic_downcast(Source* x )
{
    BOOST_ASSERT( dynamic_cast<Target>(x) == x );      //断言转型成功
    return static_cast<Target>(x);                      //静态转型
}
```

因为 polymorphic_downcast 的转型能力依赖于 static_cast，所以它只能执行向下转型，而不能执行其他的转型操作，否则会导致编译错误。例如：

```
base1 *p = new derived;
derived *pd = polymorphic_downcast<derived*>(p);      //正确

base2 *pb2 = polymorphic_downcast<base2*>(p);         //编译错误
string *ps = polymorphic_downcast<string*>(p);        //编译错误
```

polymorphic_downcast 在调试模式下使用 BOOST_ASSERT 宏来断言向下转型必定成功，因此它仅提供有限的（在调试模式下）运行时安全，在发生转型错误时不会抛出异常，使用它时必须由程序员保证转型必定成功，否则会产生未定义行为。例如下面的代码：

```
struct derived2 :public base1
{ virtual ~derived2(){};};

base1 *p = new derived;
derived2 *p2 = polymorphic_downcast<derived2*>(p);    //转型错误
```

这段代码定义了另一个派生类 derived2，然后使用 polymorphic_downcast 把一个指向 derived 对象的指针转型成为了 derived2 对象指针。无论是 debug 还是 release 模式，

polymorphic_downcast 都不会报出任何错误。

所以 polymorphic_downcast 实际上是一种“优化”的多态转型操作，使用时必须小心谨慎。

4.5.4 polymorphic_cast

polymorphic_cast 是对 dynamic_cast 的一个简单包装，只能对指针执行向下转型或者交叉转型操作，它在内部替程序员做了空指针检查，如果转型结果是空指针（nullptr）则抛出异常。

polymorphic_cast 同样是一个很小的模板函数，实现代码如下：

```
template <class Target, class Source>
inline Target polymorphic_cast(Source* x )
{
    Target tmp = dynamic_cast<Target>(x);                //动态转型
    if ( tmp == 0 ) throw_exception(bad_cast());        //空指针检查
    return tmp;
}
```

polymorphic_cast 统一了转型失败的行为，指针转型如果失败，那么它就会使用函数 throw_exception()抛出 bad_cast 异常，因此我们可以编写 try-catch 块来统一且安全地处理多态对象的转型操作。

polymorphic_cast 的能力基于 dynamic_cast，因此它的用法与 dynamic_cast 完全相同，只需要注意一点：它只能转型指针，不能转型引用。

下面的代码示范了 polymorphic_cast 的用法：

```
base1 *p = new derived;
derived *pd = polymorphic_cast<derived*>(p);           //向下转型
base2 *pb2 = polymorphic_cast<base2*>(p);              //交叉转型

try
{
  derived2 *p2 = polymorphic_cast<derived2*>(p);       //异常
  string *ps = polymorphic_cast<string*>(p);           //异常
}
catch (bad_cast& e)
{
```

```
    cout << e.what() << endl;
}
```

大多数情况下我们都可以使用 polymorphic_cast 替代 dynamic_cast，它比 dynamic_cast 命名更清楚，而且更安全易用。

4.5.5 对引用转型

polymorphic_downcast 和 polymorphic_cast 的实现简单明了，也非常易用，但它们有一个小缺陷：只能转型指针而不能转型引用，因此与 C++标准的转型操作符还有一点差距。不过我们可以为它们添加重载形式来弥补这个缺陷。

我们自定义的 polymorphic_downcast()使用元函数移除了 Target 的引用修饰，然后使用静态断言检查类型，完整的实现代码如下：

```
template <class Target, class Source> inline
Target polymorphic_downcast(Source& x)                    //对引用形式重载
{
    BOOST_STATIC_ASSERT(is_reference<Target>::value);     //断言目标是引用

    typedef typename remove_reference<Target>::type T;    //元函数移除引用
    typedef Source                                   S;   //定义元数据

    BOOST_STATIC_ASSERT( is_polymorphic<T>::value);       //要求 T 是多态类
    BOOST_STATIC_ASSERT( is_polymorphic<S>::value);       //要求 S 是多态类
    BOOST_STATIC_ASSERT((is_base_of<S, T>::value));       //S 和 T 有继承关系

    return static_cast<Target>(x);                        //静态转型
};
```

函数 polymorphic_cast()的实现与之基本相同：

```
template <class Target, class Source> inline
Target polymorphic_cast(Source& x)                        //对引用形式重载
{
    BOOST_STATIC_ASSERT(is_reference<Target>::value);     //断言目标是引用

    typedef typename remove_reference<Target>::type T;    //元函数移除引用
    typedef Source                                   S;   //定义元数据

    BOOST_STATIC_ASSERT( is_polymorphic<T>::value);       //要求 T 是多态类
```

```
    BOOST_STATIC_ASSERT( is_polymorphic<S>::value);        //要求S是多态类

    return dynamic_cast<Target>(x);                         //动态转型
};
```

下面的代码简单测试了这两个自定义转型函数的使用：

```
base1 *p = new derived;
derived *pd = polymorphic_downcast<derived*>(p);
derived &rd = polymorphic_downcast<derived&>(*p);           //正确

pd = polymorphic_cast<derived*>(p);                         //正确
base2 &pb2 = polymorphic_cast<base2&>(rd);                  //交叉转型
```

4.6 numeric conversion

数字类型的转换是个看似简单却非常复杂的问题，C++标准中定义的许多规则非常微妙，如果源类型的值超过了转型目标类型的范围，则有可能发生未定义行为。numeric conversion 库中提供了很多的工具类用于精确处理数字的转换，本书只介绍其中两个工具。

4.6.1 bounds

bounds 是一个模板类，它类似 C++标准中的数值极限类 numeric_limits，以一致的方式给出数值类型的最大极值、最小极值和最小正规值，其实现思想已经被吸纳进 C++11/14 标准。

bounds 位于名字空间 boost::numeric，需要包含头文件<boost/numeric/conversion/bounds.hpp>，即：

```
#include <boost/numeric/conversion/bounds.hpp>
using namespace boost::numeric;
```

numeric_limits

C++标准为数值的极限提供了模板类 numeric_limits，它以数值类型（char、int、double 等）为模板参数，用静态成员变量和静态成员函数给出了数值的一些相关特征，例如最大最小值。但 numeric_limits 完全仿造 C 头文件<climits>和<cfloat>，在处理整数和浮点数的极值时并不一致：整数返回数值区间的上下限，而浮点数则返回最小绝对值（ε）和上限，例如：

```
cout << numeric_limits<short>::min();                    //-32768
cout << numeric_limits<short>::max();                    //32767

cout << numeric_limits<unsigned short>::min();           //0
cout << numeric_limits<unsigned short>::max();           //65535

cout << numeric_limits<float>::min();                    //1.17549e-38
cout << numeric_limits<float>::max();                    //3.40282e+38
```

这种不一致性导致在编写泛型代码时必须手工区分数值类型，而且很容易混淆概念导致出错。

bounds

bounds 弥补了 numeric_limits 的缺点，它使用三个成员函数明确地给出了数值类型的最大极值、最小极值和最小正规值，类摘要如下：

```
template<class N>                                        //数值类型
struct bounds
{
    static N lowest  ();                                 //最小极值,下界
    static N highest ();                                 //最大极值,上界
    static N smallest();                                 //最小正规值
};
```

成员函数 lowest()和 highest ()很好理解，它们分别返回数值区间的下界和上界，也就是可取值区间的两个端点，而 smallest()对于整数总返回 1，而浮点数则返回常量 ε（FLT_MIN 或 DBL_MIN），示范代码如下：

```
using namespace boost::numeric;                          //打开名字空间

cout << bounds<short>::lowest();                         //-32768
cout << bounds<short>::highest()                         //32767;
cout << bounds<short>::smallest();                       //1

cout << bounds<float>::lowest();                         //-3.40282e+38
cout << bounds<float>::highest();                        //3.40282e+38
cout << bounds<float>::smallest();                       //1.17549e-38

//比较 bounds 和 numeric_limits
assert(bounds<short>::lowest()==numeric_limits<short>::min());
assert(bounds<float>::lowest()==-numeric_limits<float>::max()); //注意负号
```

实现原理

bounds 是研究模板元编程的一个很好的范例，它使用元函数转发技术，真正的工作是由 get_impl 完成的：

```
template<class N>
struct bounds : boundsdetail::get_impl<N>::type      //元函数转发
{} ;
```

get_impl 根据类型 N 是整数还是浮点数，分别用 Integral 和 Float 再进行元计算：

```
template<class N>
struct get_impl
{
  typedef mpl::bool_<::std::numeric_limits<N>::is_integer> is_int ;

  typedef Integral<N>     impl_int   ;                  //计算整数的元函数
  typedef Float<N>        impl_float ;                  //计算浮点数的元函数

  typedef typename mpl::if_<is_int,                     //根据是否为整数的条件计算
                             impl_int,impl_float>::type type ;
} ;
```

Integral 和 Float 利用 numeric_limits 完成最后的计算：

```
template<class N>
class Integral                                        //计算整数的元函数
{
    typedef std::numeric_limits<N> limits ;
  public :
    static N lowest  () { return limits::min(); }
    static N highest  () { return limits::max(); }
    static N smallest() { return static_cast<N>(1); }
} ;

template<class N>
class Float                                           //计算浮点数的元函数
{
    typedef std::numeric_limits<N> limits ;
  public :
    static N lowest  () { return static_cast<N>(-limits::max()); }
```

```
    static N highest () { return limits::max(); }
    static N smallest() { return limits::min(); }
};
```

C++11 的 numeric_limits

C++11 修订了 numeric_limits（C++11.18.3），使用 constexpr 关键字令所有函数都成为编译期常量，并且新增了 lowest() 函数，摘要如下：

```
template<class T>
class numeric_limits {
public:
  static constexpr T min() noexcept;              //最小值，同 C++98
  static constexpr T max() noexcept;              //最大极值，上界，同 C++98
  static constexpr T lowest() noexcept;           //最小极值，下界
  ...                                             //省略其他成员
};
```

为了保持兼容，numeric_limits 的 min()、max() 函数的含义没有变化，新增的 lowest() 明确返回数值的下界，它相当于 bounds 的 lowest()，例如：

```
assert(bounds<float>::lowest()==numeric_limits<float>::lowest());
```

4.6.2 numeric_cast

numeric_cast() 函数可以在转型时进行范围检查，如果超出范围就抛出异常 std::bad_cast，比 static_cast 更安全，能够避免很多意想不到的错误。由于 numeric_cast() 的实现很复杂，所以这里只介绍用法，不涉及具体原理。

numeric_cast 位于名字空间 boost，需要包含头文件<boost/numeric/conversion/cast.hpp>，即：

```
#include <boost/numeric/conversion/cast.hpp>
using namespace boost;
```

用法

numeric_cast() 是一个模板函数，声明如下：

```
template<typename Target, typename Source>
Target numeric_cast( Source arg );
```

numeric_cast 的用法与标准转型操作符很像，但它只能执行对数字类型的转型

(is_arithmetic<T>)。它的转型行为非常明确：如果数字转型后可以被正确地容纳到目标类型中，那么它不会有任何问题，否则它就会抛出派生自 std::bad_cast 的异常。

示范 numeric_cast 用法的代码如下：

```
short s = bounds<short>::highest();                //short 的最大值
int   i = numeric_cast<int>(s);                    //可安全地转型为 int 值
assert(i == s);

try
{
  char c = numeric_cast<char>(s);                  //超过 char 的范围，上溢异常
}
catch (std::bad_cast& e)
{
  cout << e.what() << endl;
}
```

因为 short 类型变量 s 的值为 32767，因此在 try-catch 块中把 s 转型为 char 时会抛出 numeric::positive_overflow 异常，表示无法进行数字转换。相比之下，标准的转型操作符的行为是不确定的，例如，如果把转型改用 static_cast，那么 c 将得到一个莫名其妙的 -1 值。

numeric_cast 很容易使用，它可以完全替代 static_cast 进行数字转换，带来更好的安全性。

4.7 pointer

泛化的指针（包括原始指针、智能指针和迭代器）是 C++中一个非常重要的概念，只要一个对象具有 operator*、operator++等类似指针的操作它就可以称为是泛化的指针，能够像指针一样使用。

Boost 库为此提供了数个小工具，可以操作泛化的指针，在编写泛型代码时非常有用。在本小节的最后，我们还将了解 C++11 标准专门提供的指针特征类 pointer_traits。

4.7.1 get_pointer

get_pointer() 是一个很小的辅助函数，它对多个智能指针类型做了重载——包括

unique_ptr、scoped_ptr 和 shared_ptr，总可以获得真正的原始指针。

get_pointer()位于名字空间 boost，需要包含头文件<boost/get_pointer.hpp>。

实现原理

get_pointer()的实现非常简单，就是对各种指针类型进行重载，例如：

```
template<class T> T * get_pointer(T * p)                       //原始指针
{    return p;}

template<class T> T * get_pointer(unique_ptr<T> const& p)      //unique_ptr
{    return p.get();}

template<class T> T * get_pointer(scoped_ptr<T> const & p)     //scoped_ptr
{    return p.get();}
```

用法

get_pointer()主要用于泛型编程的场景，当模板参数是一个泛化的指针时它总可以获得真正的原始指针进行后续的操作，不会引起智能指针的引用计数增加或者所有权转移，7.2 节的 mem_fn 就利用了它的这个特性。

4.7.2 pointer_cast

Boost 在头文件<boost/pointer_cast.hpp>里提供了四个标准转型操作符的包装函数，专门转型指针，声明及实现如下：

```
template<class T, class U>
inline T* static_pointer_cast(U *ptr)
{  return static_cast<T*>(ptr);}

template<class T, class U>
inline T* dynamic_pointer_cast(U *ptr)
{  return dynamic_cast<T*>(ptr);}

template<class T, class U>
inline T* const_pointer_cast(U *ptr)
{  return const_cast<T*>(ptr);}

template<class T, class U>
```

```
inline T* reinterpret_pointer_cast(U *ptr)
{  return reinterpret_cast<T*>(ptr);}
```

这四个函数的实现和用法都很简单，故在此不做过多的介绍。需要提醒大家注意的是 shared_ptr 也重载了这些函数，所以它们可以应用于 shared_ptr[①]，但不能应用于其他智能指针。

4.7.3 pointee

pointee 是一个很小的元函数，可以推导解引用（operator*）的类型，类似 3.5.1 节的 remove_pointer。它位于名字空间 boost，需要包含头文件<boost/pointee.hpp>。

类摘要

pointee 的类摘要如下：

```
template <class P>
struct pointee                  //因为使用了元函数转发，所以会有一个内部的 type 类型定义
  : mpl::eval_if<
      detail::is_incrementable<P>,          //是否可递增操作
      detail::iterator_pointee<P>,          //当作迭代器执行元计算
      detail::smart_ptr_pointee<P>          //取智能指针的 element_type
   >
{};
```

pointee 使用了模板元编程技术，接受一个可解引用的泛化指针类型 P，用::type 返回 P 所指向的值类型。

用法

pointee 不仅能够支持内建指针和标准的迭代器，也支持智能指针——包括标准库里的 unique_ptr 和 Boost 库里的 scoped_ptr、shared_ptr，示范用法的代码如下：

```
//使用元函数 is_same<>比较类型
assert((is_same<pointee<int*>::type, int>::value));
assert((is_same<pointee<unique_ptr<int> >::type, int>::value));
assert((is_same<pointee<string::iterator>::type, char>::value));
```

① C++11 标准里的 std::shared_ptr(C++11.20.7.2.2)没有定义 reinterpret_pointer_cast()函数。

```
// 计算 shared_ptr 的值类型
typedef shared_ptr<int> P;
P p(new int(10));
pointee<P>::type v = *p;
assert(v == 10);
```

请读者注意代码的后面几句，我们使用 pointee::type 来获得值类型从而赋值，这种情况下也可以直接使用 C++11 的 auto/decltype，它会自动推导出赋值表达式的类型。

因为 pointee 是一个元函数，所以它也可以很容易地使用特化来支持其他任意的指针类型。

4.7.4 indirect_reference

indirect_reference 是依赖于 pointee 的另一个元函数，它的功能与 pointee 类似，但::type 返回的是一个引用类型，相当于 pointee::type&。

indirect_reference 位于名字空间 boost，需要包含头文件<boost/indirect_reference.hpp>。

示范 indirect_reference 用法的代码如下：

```
assert((is_same<indirect_reference<int*>::type, int&>::value));
assert((is_same<indirect_reference<unique_ptr<int> >::type,
                int&>::value));
assert((is_same<indirect_reference<string::iterator>::type,
                char&>::value));
```

4.7.5 pointer_to_other

pointer_to_other 是一个工厂元函数，可以基于一个类型生产出另一个同类型的指针或智能指针，它位于名字空间 boost，需要包含头文件<boost/pointer_to_other.hpp>。

类摘要

pointer_to_other 使用了模板特化技术，类摘要如下：

```
template<class T, class U>
struct pointer_to_other< T*, U >            //对原始指针特化
{
    typedef U* type;                        //返回指向 U 类型的指针
};
```

```
template<class T, class U, template<class> class Sp>
struct pointer_to_other< Sp<T>, U >        //对智能指针特化
{
    typedef Sp<U> type;                    //返回指向 U 类型的智能指针
};
```

pointer_to_other 有数个重载形式，这里仅列出了最常用的两种形式。它有两个模板参数 T 和 U，返回与 T*同样形式但却是指向 U 的指针——正如它的名字，把指针转指向另一个类型。

第一种形式，如果 T 是原始指针 T*，那么返回 U*；第二种形式使用了较为罕见的“模板的模板参数”(template template parameters)，如果第一个参数是一个形如 Sp<T>的智能指针，那么返回一个同样形式的智能指针 Sp<U>。

用法

pointer_to_other 的应用场景主要是模板元编程，这里给出验证性质的示范代码如下：

```
//第一种形式，返回原始指针
assert((is_same<int*,
  pointer_to_other<void*, int>::type>::value));
assert((is_same<string*,
  pointer_to_other<void*, string>::type>::value));

//第二种形式，返回智能指针
assert((is_same<scoped_ptr<int>,
  pointer_to_other<scoped_ptr<float>, int>::type>::value));
assert((is_same<shared_ptr<int>,
  pointer_to_other<shared_ptr<string>, int>::type>::value));
```

请读者注意，pointer_to_other 不能处理 C++11/14 标准里的 unique_ptr：

```
assert((!is_same<unique_ptr<int>,                  //两个类型不相同
  pointer_to_other<unique_ptr<char>, int>::type>::value));
```

这是因为 unique_ptr 的类声明比较特殊：

```
template<class T, class D = default_delete<T>> class unique_ptr;
```

unique_ptr 的第二个模板参数 D 与第一个模板参数 T 相关，因此 pointer_to_other<unique_ptr<char>, int>返回的类型实际上是 unique_ptr<int, default_delete<char>>，与真正的 unique_ptr<int, default_delete<int>>不同。如果想让 pointer_to_other 正确工作，那么需要使用如下的形式：

```
assert((is_same<unique_ptr<int>,
               pointer_to_other<unique_ptr<
                       char,default_delete<int>>,int>::type>::value));
```

pointer_to_other 在我们通常的代码中较少使用，但对于编写泛型代码的库作者来说却是必不可缺的工具。

4.7.6 compare_pointees

比较指针所指向的内容是一个经常会使用的功能，但因为空指针的存在，所以指针内容的比较要比直接的值比较麻烦很多，Boost 在头文件<boost/utility/compare_pointees.hpp>中提供了两个便利的“指针”比较函数和函数对象，如下所示。

- equal_pointees(x,y)：比较两个指针是否相等。如果两个指针都是空指针，那么返回 true；如果只有一个是空指针那么返回 false；否则比较两个指针所指的内容，即*x==*y。
- less_pointees(x,y)：比较两个指针是否具有小于关系。如果 y 是空指针，那么返回 false；如果 x 是空指针，那么返回 true；否则比较两个指针所指的内容，即*x<*y。

两个指针比较函数的声明如下：

```
template<class OptionalPointee>
bool equal_pointees( OptionalPointee const& x, OptionalPointee const& y );

template<class OptionalPointee>
bool less_pointees( OptionalPointee const& x, OptionalPointee const& y );
```

因为 equal_pointees()和 less_pointees()是泛型函数，因此只要参数的行为类似指针就都可以执行比较操作（需支持 operator!和 operator*），“指针”可以是原始指针、智能指针（scoped_ptr、shared_ptr）或者 boost::optional。

示范 equal_pointees()和 less_pointees()用法的代码如下：

```
scoped_ptr<int> p1(new int(10));          //两个作用域智能指针
scoped_ptr<int> p2(new int(20));

assert(!equal_pointees(p1, p2));          //不相等
assert(less_pointees(p1, p2));            //小于关系

p2.reset();                               //p2 变为空指针
```

```
assert(!less_pointees(p1, p2));             //不存在小于关系

optional<string> op1, op2;                  //两个 optional 对象，均为空

assert(equal_pointees(op1, op2));           //相等
op2 = "hello";                              //赋值
assert(less_pointees(op1, op2));            //小于关系
```

std::unique_ptr 和迭代器虽然行为也类似指针，但它们不支持 operator!，所以不能应用于这两个函数。

4.7.7 pointer_traits

C++11 标准（C++11.20.6.3）在头文件<memory>里定义了一个指针特征类（元函数）pointer_traits，它可以萃取指针、智能指针的相关类型信息，包含了 pointee 和 pointer_to_other 的功能。

pointer_traits 不支持迭代器，因为那是 iterator_traits 的工作（参见 5.3 节）。

pointer_traits 的基本声明如下：

```
template <class Ptr>                                    //适用于智能指针或原始指针，非迭代器
struct pointer_traits {
  typedef Ptr            pointer;                       //指针类型
  typedef some_define    element_type;                  //元素类型，即解引用类型
  typedef some_define    difference_type;               //距离类型

  template <class U>
using rebind = some_define;                             //绑定到另一个类型 U
  static pointer         pointer_to(r);                 //获得指向 r 的指针
};
```

pointer_traits 还针对原始指针做了特化：

```
template <class T>                                      //对原始指针的特化形式
struct pointer_traits<T*> {
  typedef T*          pointer;                          //原始指针类型
  typedef T           element_type;                     //元素类型，即解引用类型
  typedef ptrdiff_t   difference_type;                  //指针的距离类型

  template <class U> using rebind = U*;                 //指向类型 U
  static pointer      pointer_to(r);                    //获得 r 的指针，使用 addressof
};
```

从上面的代码可以看出：pointer_traits<T>::element_type 相当于 pointee<T>::type，pointer_traits<T*>::rebind<U>相当于 pointer_to_other<T*, U>，示范代码如下：

```
//查看原始指针的特征
assert((is_same<pointer_traits<int*>::pointer, int*>::value));
assert((is_same<pointer_traits<int*>::element_type, int>::value));
assert((is_same<pointer_traits<int*>::rebind<char>,      //使用 rebind 类别名
        pointer_to_other<int*, char>::type >::value));

//查看智能指针的特征
typedef shared_ptr<std::string> sptr;                    //定义智能指针类型

assert((is_same<pointer_traits<sptr>::element_type,
        pointee<sptr>::type>::value));

assert((is_same<pointer_traits<sptr>::difference_type, ptrdiff_t>::value));

assert((is_same<pointer_traits<sptr>::rebind<char>,      //使用 rebind 类别名
        pointer_to_other<sptr, char>::type >::value));
```

遗憾的是 pointer_traits::rebind 同样也无法处理 unique_ptr：

```
typedef unique_ptr<char> char_ptr;                  // unique_ptr 指针类型定义
typedef unique_ptr<int> int_ptr;
typedef std::pointer_traits<char_ptr>::rebind<int> new_ptr; //rebind
assert((!std::is_same<int_ptr, new_ptr>::value));          //类型不同
```

4.8 总结

在本章中，我们讨论了 Boost 程序库中数个有用的小工具，要求读者对 C++中的缺省构造函数、拷贝构造函数、对象的初始化、操作符重载、类型转换等许多基本概念有较深刻的理解，我们还使用模板元编程概念对其中的一些组件做了深入的剖析和扩展。

compressed_pair 是我们看到的第一个小工具，它是对 std::pair 的增强，对于库作者来说它非常有用，因为可以统一地处理空类和非空类，使空间占用得到尽可能的优化。指针容器库 ptr_container（第 8 章）中的 auto_type 类型就使用了它。

checked_delete 是一个“智能操作符 delete”，可以代替 delete，检查删除的对象是

否是一个完整类，能够避免很多在运行时可能发生的错误，它对应的“智能操作符 new”是 factory 函数对象（7.3 节）。address_of 是另一个智能操作符，它比内建的 operator&更好，总能够获取到真实的对象地址，但因为使用了多次强制类型转换，所以在运行效率上有一点损失。通常情况下我们不应该重载 operator&，也应该少使用 address_of。

base_from_member 对基类使用子类成员初始化提供了一个标准的解决方案，在只有一个成员的时候可以简化相当多的工作，因而被 Boost 库的许多其他组件所使用，例如 iostreams 库（第 11 章）的 stream 类。但在多重继承或基类的初始化比较复杂的时候它就不太适用了，这时我们可以基于它的工作原理实现自己的 base_from_member 辅助类。

类型转换是 C 语言遗留的传统，老式的类型转换风格很差，应当尽量少用，因为它使 C++的静态强类型特性部分地失去了效力，真正好的程序应该少使用。在万不得已的情况下类型转换应该使用新式的转换操作符，Boost 也为此提供了专门针对多态转型和数字转型的操作函数 polymorphic_downcast/polymorphic_cast/numeric_cast，使用它们能够让代码更加整洁干净和更少错误。

泛化的指针是泛型编程中经常操作的对象，Boost 库中有多个专门用于处理指针类型的小工具，它们在通常的开发工作中用到的机会可能很少，但学习它们有利于我们理解 Boost 库其他组件的工作原理。

第5章 迭代器

迭代器是现代 C++编程中的重要角色，是连接容器和算法的“粘接剂”，在标准库中占有非常重要的地位，而 Boost 又对迭代器的演化做出了重要的贡献。

本章首先讨论迭代器设计模式和 C++标准中迭代器的基本知识，然后在此基础上讲解 Boost 的新式迭代器定义和分类，之后重点研究 iterators 库里提供的迭代器工具，它简化了程序员编写符合标准的迭代器所需要做的工作。

Boost 库的迭代器功能并没有归在一个统一的头文件中，而是分散成了许多小的头文件，显得有些凌乱，使用时必须根据具体情况包含特定的头文件。

阅读本章需要读者对迭代器模式、适配器模式和标准库的迭代器有一定的了解，必要时请结合推荐书目学习。

5.1 概述

在本小节中，我们将简要讨论迭代器设计模式，回顾 C++中迭代器相关的各种概念和工具，并介绍 Boost 的新式迭代器概念，这些是本章的基础。

5.1.1 迭代器模式

推荐书目[1] 中对迭代器模式的描述如下：

“提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。”

迭代器模式是一个行为模式，它把聚合的表示与其中元素的访问分离开来，这两者都可以独

立地变化，增强了使用的灵活性，因此几乎所有面向对象的系统中都应用了迭代器模式。

迭代器模式有两个基本的参与者：聚合和迭代器。聚合定义了元素的集合方式，它对用户是不透明的，但对迭代器开放了访问接口，允许迭代器访问。迭代器依赖于聚合，它对外提供访问和遍历聚合的接口，这样用户无须关心聚合的内部结构就能够以任意的方式访问聚合里的元素。聚合和迭代器是彼此独立的，这意味着它们都可以是多态的（包括静态多态和动态多态），而且在一个聚合上可以执行多个不同的迭代，一个迭代器也可以同时对多个聚合执行迭代。

除了聚合和迭代器，迭代器模式还可能有其他参与者，例如迭代器产生器，它产生基于聚合的某个访问方式的迭代器。迭代器产生器可以是一个单独的工厂类，也可以是聚合或者迭代器的某个工厂方法。

为了访问和遍历聚合，迭代器必须具备四个操作接口：First、Next、IsDone 和 CurrentItem，分别用于执行迭代器初始化、前进、检测是否完成迭代和访问当前元素。除了基本接口以外，迭代器还可以拥有更多的接口以提供更强大更方便的访问和遍历功能，例如比较迭代位置、后退、前进 N 个位置等。

我们可以用标准库中的容器 vector 和 vector::iterator 来具体理解一下迭代器模式：

vector 定义了一个元素的聚合，其内部实现对用户来说是不透明的（虽然大多数情况下是一个原生数组），vector::iterator 是基于这个聚合的一个迭代器，它可以正向遍历 vector。vector 的成员函数 begin()/end() 是迭代器产生器，可以产生聚合上的迭代器，同时它们还对应迭代器模式里的 First 和 IsDone 操作，确定了迭代器的起点和终点。vector::iterator 重载了 operator++和 operator*，可以在聚合上前进和访问聚合里的元素，对应迭代器模式里的 Next 和 CurrentItem 操作。vector::iterator 还重载了 operator==、operator-- 和 operator+=等操作符，可以在 vector 上访问任意位置上的元素。

5.1.2 标准迭代器

C++标准（C++11.24.2）将迭代器分为五类，它们特征的简要描述如下所述[①]。

- 输入迭代器 ：或称“只读迭代器”（从迭代器输入），只提供 operator++，可以执行相等比较。
- 输出迭代器 ：或称“只写迭代器”（向迭代器输出），只提供 operator++，没有相等比较功能。

① 读者可参考 11.3.4 节“迭代器概念检查”进行对照阅读。

- 前向迭代器　　：可以读写，只提供 operator++，可以执行相等比较和赋值操作。
- 双向迭代器　　：在前向迭代器的基础上增加了 operator--，即可以执行后退操作。
- 随机访问迭代器：在双向迭代器的基础上增加了迭代器的算术运算功能，提供 operator[ ] 和 operator+=。

对于标准容器来说， forward_list 的迭代器是前向迭代器， list、set、map 的迭代器都是双向迭代器，而内建数组、vector、deque、string 的迭代器则是随机访问迭代器。

为了区分不同类型的迭代器，标准库使用一些 tag 类（空类）作为迭代器的标签，如 std::input_iterator_tag、std::output_iterator_tag，这些类别信息可以使用特征类 std::iterator_traits 提取（5.3.1 节）。

标准库对迭代器的分类是一个重要的成果，但它把迭代器的取值和遍历这两个不相关的操作混合在了一起，因而被认为存在缺陷，而且造成许多现实中的迭代器无法被恰当地归类。

5.1.3 新式迭代器

iterators 库定义了一组基于标准库的新的迭代器概念、构造框架和有用的适配器，能够帮助程序员更轻松地应用迭代器模式来创建、使用迭代器类型。这里我们先讨论它的概念定义，库提供的工具将在随后介绍。

iterators 程序库区分了迭代器的值访问概念和遍历概念，重新划分了迭代器的类型，使迭代器的概念描述更加清楚。①

根据值访问方式，迭代器可分为以下四类。

- 可读迭代器　：提供 operator*，可返回可转换为类型 T 的右值。
- 可写迭代器　：提供 operator*，可以执行赋值操作（不一定是左值）。
- 可交换迭代器：两个迭代器所指的值可用标准库的 iter_swap() 函数交换，即同时满足可读迭代器和可写迭代器的要求。
- 左值迭代器　：满足可交换迭代器，并且 operator*可返回左值，即类型 T 的引用。

根据遍历方式，迭代器可分为以下五类（下面的迭代器概念均在前一个的基础上强化定义）。

- 可递增迭代器　　：提供 operator++，可拷贝构造和赋值。

① 这些新式迭代器概念可参考 12.3.5 节阅读，那里提供了对这些迭代器概念的检查功能。

- 单遍迭代器 ：增加 operator==、operator!=比较操作。
- 前向遍历迭代器 ：增加 difference_type 类型定义，可计算距离和缺省构造。
- 双向遍历迭代器 ：增加 operator--，可以逆向遍历。
- 随机访问遍历迭代器 ：增加迭代器的算术运算和比较运算，并提供 operator[ ]。

使用 Boost 的新式迭代器分类，标准库原有的五个迭代器类别就可以更精确地定义，如下所示。

- 输入迭代器 = 可读迭代器 + 单遍迭代器。
- 输出迭代器 = 可写迭代器 + 可递增迭代器。
- 前向迭代器 = 左值迭代器 + 前向遍历迭代器。
- 双向迭代器 = 左值迭代器 + 双向遍历迭代器。
- 随机访问迭代器 = 左值迭代器 + 随机访问遍历迭代器。

而标准库中“不是容器的容器”vector<bool>的迭代器（它返回一个代理而不是 bool 值）也就可以被正确归类为可交换迭代器（可读迭代器+可写迭代器）+随机访问遍历迭代器。

新式迭代器分类与标准迭代器分类的关系可以用下面的表格描述：

	可读	可写	可交换	左值
可递增		标准输出迭代器		
单遍	标准输入迭代器			
前向				标准前向迭代器
双向				标准双向迭代器
随机			vector<bool>的迭代器	标准随机迭代器

表格中的空白部分是标准迭代器分类所没有覆盖的部分，也就是说存在着这样的新式迭代器可以使用。

5.1.4 标准迭代器工具

为了方便地操作迭代器，标准库提供了若干个辅助工具，它们可以让迭代器用起来更加容易。

迭代器基类

标准库提供了一个迭代器基类 iterator，用来简化自定义迭代器（C++11.24.4.2）：

```
template<class Category, class T, class Distance = ptrdiff_t,
         class Pointer = T*, class Reference = T&>
struct iterator {
  typedef T           value_type;
  typedef Distance    difference_type;
  typedef Pointer     pointer;
  typedef Reference   reference;
  typedef Category    iterator_category;
};
```

不过很遗憾，std::iterator 实在是太过简单，在实践开发中几乎起不到什么作用，我们最好使用 5.4 节的 iterator_facade 或 5.5 节的 iterator_adaptor。

迭代器辅助函数

标准库提供了以下三个迭代器辅助函数（C++11.24.4.4）。

- advance(pos, n) ：使迭代器前进或后退 n 个位置。
- distance(p1, p2) ：计算两个迭代器间的距离。
- iter_swap() ：交换两个迭代器所指的元素的值。

这三个迭代器辅助函数能够以统一的方式操作迭代器，而无须关心迭代器的类别，例如，使用 advance()函数，即使是不支持随机访问的迭代器也可以前进或后退任意步，增强了程序的灵活性，5.2 节的 next_prior 就使用了 advance()。

迭代器适配器

标准库另外提供一些迭代器适配器，可以把一种迭代器转换为另一种迭代器（C++11.24.5，C++11.24.6）。

- 逆向迭代器 ：适配后迭代器可以逆序迭代。
- 转移迭代器 ：C++11/14 新增的迭代器适配器，迭代器返回右值引用。
- 插入迭代器 ：把赋值操作转换为执行插入操作的输出迭代器。
- 流迭代器 ：把 IO 流转换为迭代器操作。

这三种迭代器适配器中较常用的是后两者，可以让我们以操作迭代器的方式去操作容器和流，例如：

```
using namespace boost::assign;            //打开 assign 名字空间
vector<int> v1, v2;                       //两个标准容器

push_back(v1),1,2,3,4,5;                  //使用 assign 库添加数据

std::copy(v1.begin(), v1.end(),           //copy 算法操作迭代器
    back_inserter(v2));                   //使用插入迭代器把容器转换为输出迭代器
assert(v2.size() == 5);

std::copy(v2.begin(), v2.end(),           //copy 算法操作迭代器
    ostream_iterator<int>(cout));         //使用流迭代器向流输出数据
```

关于这些迭代器辅助工具更多的知识请参考推荐书目[2]。

5.1.5 迭代器与算法

C++标准库提供了大量的标准算法，这些算法并不直接操作容器，而是以迭代器指定一个容器的区间，然后通过迭代器来操作容器里的元素。从某种程度上来说，算法真正操作的实际上是迭代器，它并不了解容器。

在本章中我们最常用的算法是变动算法 std::copy，它的声明如下：

```
template <class InputIter, class OutputIter>
inline OutputIter copy(InputIter first, InputIter last, OutputIter result);
```

std::copy 顾名思义，它正向遍历[first, last)迭代器区间，通过迭代器把其中的所有元素拷贝到目标区间 result 里，然后返回目标区间的最后位置——这个返回值通常被忽略。

std::copy 还有许多同族的算法，如 copy_backward、reverse_copy、remove_copy 等，读者可自行了解。

5.2 next_prior

next_prior 组件提供了两个非常简单的模板函数 next()和 prior()，使用迭代器辅助工具 advance()为仅提供 operator++和 operator--的迭代器增加了前向或者后向 N 步的通用解法。它们已经被收入 C++11/14 标准（头文件<iterator>，C++11.24.4.4）。

next_prior 位于名字空间 boost，需要包含头文件<boost/next_prior.hpp>，即：[①]

```
#include <boost/next_prior.hpp>
using namespace boost;
```

5.2.1 函数声明

next()和prior()的实现使用了模板元编程，会用 type_traits 库检查类型 T 是否支持 operator+=，简化的实现代码如下：

```
template <class T>
inline T next(T x)                            //单参形式
{ return ++x; }                               //调用 operator++
template <class T, class Distance>
inline T next(T x, Distance n)                //双参形式
{
    std::advance(x, n);                       //调用 advance()辅助函数
    return x;
}

template <class T>
inline T prior(T x)                           //单参形式
{ return --x; }                               //调用 operator--
template <class T, class Distance>
inline T prior(T x, Distance n)               //双参形式
{
    std::advance(x, -n);                      //调用 advance()辅助函数
    return x;
}
```

next()和prior()各有两种重载形式，单参版本只前进或后退一步，双参版本则前进或后退 n 步。它们使用了标准库中的迭代器辅助函数 advance()，会自动根据迭代器的类型采取最有效率的步进措施——对于随机访问迭代器直接调用 operator+=，而对于其他的迭代器则循环调用递增或递减操作。

如果迭代器不提供 operator++或 operator--（例如输入迭代器、输出迭代器和前向迭代器没有 operator--），那么使用这两个函数时会引发编译错误。

① 也可以包含<boost/utility.hpp>，它含有数个小工具的实现。

5.2.2 用法

next()和 prior()的代码虽然简单，但它们很好地统一了迭代器的操作方式，因为只有随机访问迭代器才能够编写如 iter + n 这样的代码，不利于泛型编程。有了这两个函数，我们就可以写出对所有迭代器类型一致的操作代码。

另外需要注意的是虽然它们的名字叫 next（后继）和 prior（前驱），但双参版本的第二个参数是可以传入负数的，也就是说它们实际上可以使迭代器任意前后移动（前提是迭代器支持 operator--）。

下面的代码定义了一个模板函数 get_n()，它可以返回迭代器后 n 个位置的值：

```
template<typename I>
typename iterator_traits<I>::value_type         //使用 iterator_traits
get_n(I& iter, int n)                           //引用方式传递迭代器变量
{
  return *(next(iter, n));           //也可以写成 return *(prior(iter, -n));
}

int main()
{
  list<int> l{ 1,2,3,4};                        //C++11 初始化

  auto p1 = l.begin();                          //获得首末位置的迭代器
  auto p2 = l.end();

  assert(get_n(p1, 1)    == 2);
  assert(get_n(p2, -1)   == 4);
  assert(get_n(p2, -2)   == 3);
}
```

在这段代码中，我们使用的容器是 std::list，它的迭代器是双向迭代器，不支持随机访问，因此，如果要访问任意位置的元素通常需要使用一个循环来迭代或者是使用 std::advance()。而现在有了 next()就方便多了，泛型函数 get_n()可以用一个简单的语句完成这项工作，比直接调用 std::advance()好看得多。

next()的另一个好处是它是以值的方式使用迭代器，因而不会产生 std::advance()改变迭代器位置的副作用，不必为了保持迭代器原位置不变来声明一个变量临时保存迭代器位置。如

果不使用 next()，那么我们必须这么写：

```
template<typename I>
typename iterator_traits<I>::value_type
get_n(I& iter, int n)                    //引用方式传递迭代器变量
{
  I tmp = iter;                          //为不变动 iter 的位置必须使用一个临时变量
  std::advance(tmp, n);                  //tmp 前进 n 个位置，而 iter 不变
  return *tmp;
}
```

本书的第 8 章的部分代码使用了这两个工具函数，读者可进一步参考。

5.2.3 C++11/14 标准

C++11 标准在头文件<iterator>里定义了 std::next()和 std::prev()，实现的功能相同，但声明形式略有不同：

```
template <class ForwardIterator>
ForwardIterator next(ForwardIterator x, difference_type n = 1);

template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator x, difference_type n = 1);
```

可以看到，C++11 标准里的 next()和 prev()使用了缺省参数的形式，迭代器的类型定义也更加明确：next()使用的是前向迭代器，prev()使用的是双向迭代器。

5.3 iterator_traits

iterator_traits 库提供了标准的元函数来访问迭代器的基本属性，较标准库的 std::iterator_traits 而言它更加规范，更容易被用在泛型编程和模板元编程中。

iterator_traits 位于名字空间 boost，需要包含头文件<boost/iterator/iterator_traits.hpp>，即：

```
#include <boost/iterator/iterator_traits.hpp>
using namespace boost;
```

5.3.1 标准迭代器特征类

标准库提供 std::iterator_traits 来获得迭代器（或指针）的属性（C++11.24.4.1），基本实现代码如下：[①]

```
template <class Iterator>                                //针对标准迭代器类型
struct iterator_traits {
  typedef typename Iterator::iterator_category       iterator_category;
  typedef typename Iterator::value_type              value_type;
  typedef typename Iterator::difference_type         difference_type;
  typedef typename Iterator::pointer                 pointer;
  typedef typename Iterator::reference               reference;
};

template <class T>                                       //针对原生指针类型特化
struct iterator_traits<T*> {
  typedef random_access_iterator_tag                 iterator_category;
  typedef T                                          value_type;
  typedef ptrdiff_t                                  difference_type;
  typedef T*                                         pointer;
  typedef T&                                         reference;
};
```

iterator_traits 接受一个迭代器（或指针）类型，可以获得迭代器（或指针）所必备的五个类型信息。

- iterator_category ：迭代器的分类，参见 5.1.2 节。
- value_type ：迭代器所指的值类型。
- reference ：迭代器的值引用类型。
- pointer ：迭代器的指针类型。
- difference_type ：迭代器的距离类型。

std::iterator_traits 可以正常工作，而且它已经被使用了很多年，但从模板元编程的角度来看它不符合元函数的规范定义，是非标准元函数，难以在更广泛的领域中使用。

① 如果迭代器的定义不符合规范，那么 std::iterator_traits 就不能获得这些信息了。

5.3.2 类摘要

iterator_traits 库把 std::iterator_traits 中被“揉成一团”的元数据分解开来，形成了五个标准元函数，而功能则是完全相同的。

- iterator_category<I> ：返回迭代器的分类。
- iterator_value<I> ：返回迭代器所指的值类型。
- iterator_reference<I> ：返回迭代器的值引用类型。
- iterator_pointer<I> ：返回迭代器的指针类型。
- iterator_difference<I> ：返回迭代器的距离类型。

实际上，这五个元函数自己没有做任何工作，只是调用了 std::iterator_traits，把非标准的内部类型定义转化成了标准的::type 返回。

例如，iterator_value 的实现代码如下：

```
template <class Iterator>
struct iterator_value                          //调用 iterator_traits
{
    typedef typename iterator_traits<Iterator>::value_type type;
};
```

5.3.3 用法

因为 iterator_traits 库提供的这五个元函数符合标准定义，所以它们用起来要比 std::iterator_traits 更加灵活方便，可以很容易地与其他元编程工具混合使用，发挥更大的作用。

示范这五个元函数用法的代码如下：

```
typedef int* I;                                //一个指针类型的迭代器

assert((is_same<iterator_value<I>::type, int>::value));
assert((is_same<iterator_reference<I>::type, int&>::value));
assert((is_same<iterator_category<I>::type,
            std::random_access_iterator_tag>::value));
```

```
typedef list<int>::const_iterator II;    //标准容器 list 的 const 迭代器

assert((is_same<iterator_value<II>::type, int>::value));
assert((is_same<iterator_reference<II>::type,
                 const int&>::value));
assert((is_same<iterator_category<II>::type,
                std::bidirectional_iterator_tag>::value));
```

5.4 iterator_facade

iterator_facade 是 iterators 库里的一个重要组件，它使用外观模式提供一个辅助类，能够帮助程序员更容易地创建符合标准的迭代器，比标准库里的 std::iterator 更容易使用，而且更不容易犯错误。

iterator_facade 定义了数个迭代器的核心接口，用户只需要实现这些核心功能就可以编写正确且完备的迭代器。

iterator_facade 位于名字空间 boost，需要包含头文件<boost/iterator/iterator_facade.hpp>，即：

```
#include <boost/iterator/iterator_facade.hpp>
using namespace boost;
```

5.4.1 迭代器的核心操作

一个完备的迭代器拥有相当多的外部接口和内部的类型定义，但它存在一个必需的核心操作集合，这个集合定义了迭代器的必需功能。

iterator_facade 要求用户的迭代器类必须实现下面的五个功能（共六个接口，但依据迭代器的类型某些函数可以不实现）。

- 解引用　：dereference()，实现可读迭代器和可写迭代器必需。
- 相等比较：equal()，实现单遍迭代器必需。
- 递增　　：increment()，实现可递增迭代器和前向遍历迭代器必需。
- 递减　　：decrement()，实现双向遍历迭代器必需。
- 距离计算：advance()和 distance_to()，实现随机访问遍历迭代器必需。

这些核心操作将被 iterator_facade 用来实现迭代器的外部接口，所以这些函数通常应该是 private 的。[1]

为了使 iterator_facade 能够访问这些核心操作函数，库又提供了一个辅助类 boost::iterator_core_access，它定义了可以访问 private 核心操作的若干静态成员函数，用户迭代器需要把它声明为友元。

此外，因为迭代器通常需要拷贝和赋值，故用户自定义迭代器还需要有拷贝构造函数和缺省构造函数。

5.4.2 类摘要

iterator_facade 基于迭代器核心操作实现迭代器功能，类摘要如下：

```
template <
    class Derived,                          //迭代器子类
    class Value,                            //迭代器值类型
    class CategoryOrTraversal,              //迭代器的分类
    class Reference  = Value&,              //迭代器的值引用类型
    class Difference = ptrdiff_t,           //迭代器的距离类型
>
class iterator_facade {
 public:
    //迭代器各种必需的类型定义
    typedef     remove_const<Value>::type    value_type;
    typedef     Reference                    reference;
    typedef     Value*                       pointer;
    typedef     Difference                   difference_type;
    typedef     some_define                  iterator_category;

    //迭代器的各种操作定义
    Reference   operator*() const;
    some_define operator->() const;
    some_define operator[](difference_type n) const;
    Derived&    operator++();
    Derived     operator++(int);
    Derived&    operator--();
```

① 当然，也可以把这些核心操作直接声明成 public（generator_iterator 就是如此），但一般情况下不应该这么做。

```
    Derived     operator--(int);
    Derived&    operator+=(difference_type n);
    Derived&    operator-=(difference_type n);
    Derived     operator- (difference_type n) const;
};
...                                       //许多关系运算符定义
```

iterator_facade 基于用户迭代器的六个核心操作实现了数个迭代器接口，包括 operator++、operator--、operator*以及关系运算符。它有五个模板参数，但后两个可以使用缺省参数，我们通常只需要关心前三个。

- Derived : 子类的名字，也就是用户正在编写的自己的迭代器（即基类链技术[①]）。
- Value : 迭代器的值类型。
- CategoryOrTraversal : 定义了迭代器的分类，它即可以使用标准分类也可以使用新式分类（参见 5.1 节）。

由于 CategoryOrTraversal 的使用比较复杂，下面把它的取值单独列出。

- std::input_iterator_tag : 标准的输入迭代器。
- std::output_iterator_tag : 标准的输出迭代器。
- std::forward_iterator_tag : 标准的前向迭代器。
- std::bidirectional_iterator_tag : 标准的双向迭代器。
- std::random_access_iterator_tag : 标准的随机访问迭代器。
- boost::incrementable_traversal_tag : 可递增遍历迭代器。
- boost::single_pass_traversal_tag : 单遍迭代器。
- boost::forward_traversal_tag : 前向遍历迭代器。
- boost::bidirectional_traversal_tag : 双向遍历迭代器。
- boost::random_access_traversal_tag : 随机访问遍历迭代器。

① 又称 Curiously Recurring Template Pattern (CRTP)，读者可参见推荐书目[3]对 operators 库的论述。

5.4.3 用法

使用 iterator_facade 之前我们必须要规划自己的迭代器的几个基本特征，包括迭代器的名称、在何种集合上迭代、迭代的对象（解引用的类型）和迭代遍历的类型。

首先，迭代器要用 public 的方式继承 iterator_facade，同时指定 iterator_facade 的模板参数，第一个模板参数必须是迭代器自己（用于基类链），然后是值类型和迭代器分类，最后两个模板参数可以使用缺省值。

接下来我们要考虑迭代器的构造函数：迭代器必须能够拷贝构造和赋值（可递增迭代器概念），如果是前向遍历迭代器则还需要缺省构造函数。构造函数通常需要传入被迭代的集合对象。还要有一个成员变量来保存迭代的位置，这样我们才能执行递增或递减操作。

最后我们就可以实现迭代器所必需的核心操作函数了，这些核心函数最好是 private 的，并声明友元类 iterator_core_access 授予 iterator_facade 的访问权。

下面我们使用两个简单的例子示范 iterator_facade 的用法，更多的例子可见 5.5 节和 5.6 节。

示例 1

作为第一个示例，我们定义一个在 std::vector 上的可写单遍迭代器 vs_iterator[①]，所以需要使用 boost::single_pass_traversal_tag 作为迭代器分类标志。因为这个迭代器是可写而不是只读的，不是输入迭代器，所以不能使用 std::input_iterator_tag（当然用了也不算大错）：

```
template<typename T>
class vs_iterator :
     public boost::iterator_facade<                    //基类链技术继承
     vs_iterator<T>, T,                                //子类名和值类型
     boost::single_pass_traversal_tag>                 //单遍迭代器类型
```

然后实现迭代器的内部迭代位置存储、构造函数和赋值函数：

```
{
private:
  std::vector<T>&     v;                               //容器的引用
  size_t              current_pos;                     //迭代器的当前位置
```

① 读者可参考 12.3.5 节的新式迭代器概念检查验证 vs_iterator 的迭代器属性。

```
public:
  typedef boost::iterator_facade<...> super_type;    //定义基类的别名
  typedef vs_iterator                 this_type;     //定义自身的别名

  typedef typename super_type::reference reference; //使用基类的引用类型

  vs_iterator(vector<T> &_v, size_t pos = 0):        //构造函数
   v(_v), current_pos(pos)
  {}
  vs_iterator(this_type const& other):               //拷贝构造函数
     v(other.v), current_pos(other.current_pos)
  {}
  void operator=(this_type const& other)             //赋值函数
  {
     this->v = other.v;
     this->current_pos = other.current_pos;
  }
```

因为 vs_iterator 是单遍迭代器，没有太多的功能，所以不需要实现所有的迭代器核心操作，只要解引用、递增和比较就足够了：

```
private:
  friend class boost::iterator_core_access;          //必需的友元声明

  reference dereference() const                      //解引用操作
  {   return v[current_pos]; }

  void increment()                                   //递增操作
  {   ++current_pos; }

  bool equal(this_type const& other) const           //比较操作
  {   return this->current_pos == other.current_pos;}
};
```

这样，只用了三十多行代码，我们就轻松完成了一个完全符合标准的迭代器类型。vs_iterator 用起来和容器内置的迭代器差不多，示范 vs_iterator 用法的代码如下：

```
vector<int> v{1,2,3,4,5};                            //一个标准容器

vs_iterator<int> vsi(v), vsi_end(v, v.size());       //声明两个迭代器
```

```
*vsi = 9;                                  //使用迭代器的 operator*操作集合里的元素

std::copy(vsi, vsi_end,                                //copy 算法
  ostream_iterator<int>(cout, ","));                   //使用流迭代器输出到标准流
```

程序运行结果如下：

```
9,2,3,4,5,
```

如果把 vs_iterator 模板参数中的值类型改为 const T，那么 vs_iterator 就会变成一个标准的输入迭代器（可读不可写迭代器），下面的语句将无法通过编译：

```
*vsi = 9;                                              //编译错误
```

很显然，如果使用标准的 std::input_iterator_tag 分类标志，那么就无法精确描述可写单遍迭代器的特征。

示例 2

接下来我们定义一个每次跳跃式前进 N 个位置的步进迭代器 step_iterator，它接受一个迭代器 I 和整数 N 作为模板参数，递增时调用 N 次 operator++。[①]

首先迭代器还是要选择恰当的模板参数，继承自 iterator_facade：

```
template<typename I, std::ptrdiff_t N = 2>             //缺省一次前进两步
class step_iterator:
  public boost::iterator_facade<
      step_iterator<I>,                                //子类名
      typename boost::iterator_value<I>::type const, //元函数获得值类型
      boost::single_pass_traversal_tag>                //单遍分类
```

然后是迭代器位置存储、构造函数和赋值函数：

```
{
private:
  I               m_iter;                                //迭代器位置
public:
  typedef boost::iterator_facade<...> super_type;        //定义基类的别名
  typedef step_iterator               this_type;         //定义自身的别名
  using typename super_type::reference;                  //使用 using 关键字
```

① 为了使代码简单，这里没有对迭代器是否越界的检查，请读者注意。

```
  step_iterator(I x) : m_iter(x) {}                    //构造函数

  step_iterator(this_type const& o) = default;         //拷贝构造，使用 default
  this_type& operator=(this_type const& o) = default;//赋值函数
```

同样我们需实现单遍迭代器必需的解引用、递增和比较操作：

```
private:
  friend class boost::iterator_core_access;         //必需的友元声明

  reference dereference() const                     //解引用操作
  {   return *m_iter; }

  void increment()
  {   std::advance(m_iter, N);   }                  //连续前进 N 次,不能用 m_iter+=N

  bool equal(step_iterator const& other) const      //比较操作
  {   return m_iter == other.m_iter;}
};
```

注意在递增操作 increment()中不能使用 m_iter += N 的形式，因为我们不能假设迭代器是随机访问迭代器，只有随机访问迭代器才能使用 operator+=。另外我们也不能使用 next()函数（5.2 节），因为它不改变迭代器的位置，不满足这里的要求。

示范 step_iterator 用法的代码如下：

```
char s[] = "12345678";                             //字符数组

std::copy(s, s + 8,                                //copy 算法，原始指针用作迭代器
  std::ostream_iterator<char>(cout));              //流迭代器输出

step_iterator<char*> first(s), last(s + 8);        //用 char*迭代，使用默认步长 2

std::copy(first, last,                             //copy 算法
  std::ostream_iterator<char>(cout));              //流迭代器输出
```

程序的运行结果是：

```
12345678
1357                                               //跳过了偶数位置的元素
```

step_iterator 的另两个使用例子可见 5.6.8 节和 5.6.11 节。

5.5 iterator_adaptor

迭代器是一种很容易使用的对象，标准库里的很多算法都可以操作迭代器，所以很多时候我们希望把其他对象模拟成一个迭代器的形式来操作，这样能够简化我们的代码。有的时候程序里已经存在了可用的迭代器，例如数组指针、标准容器的迭代器等，但它们不能满足特定的要求，不方便使用。在这两种情况下，iterators 库里的 iterator_adaptor 就可以发挥作用，它基于对象适配器模式，主要功能就是把已经存在的类型适配为一个新的迭代器。

iterator_adaptor 位于名字空间 boost，需要包含头文件<boost/iterator/iterator_adaptor.hpp>，即：

```
#include <boost/iterator/iterator_adaptor.hpp>
using namespace boost;
```

5.5.1 类摘要

iterator_adaptor 派生自 iterator_facade，同样使用了基类链技术，类摘要如下：[1]

```
template <
    class Derived,                                   //迭代器子类名
    class Adaptee,                                   //被适配的迭代器
    class Value                = use_default,        //值类型
    class CategoryOrTraversal = use_default,         //迭代器分类标志
    class Reference            = use_default,        //迭代器值引用类型
    class Difference           = use_default         //迭代器距离类型
>
class iterator_adaptor  : public iterator_facade<Derived,...>
{
    friend class iterator_core_access;               //必需的友元声明
 public:
    iterator_adaptor();
```

① iterator_adaptor 沿用了 C++标准库中的术语“Base”，这个 Base 与继承毫无关系，实际上相当于适配器模式中的 Adaptee，即被适配的对象。为了叙述清晰，下文中将所有的 Base 都改称为 Adaptee，请读者阅读时留意。

```
    explicit iterator_adaptor(Adaptee const& iter);
    typedef Adaptee base_type;                  //被适配的原始迭代器类型定义
    Adaptee const& base() const;
private:
    // 适配Adaptee实现iterator_facade必需的六个核心迭代器接口
    reference             dereference() const;
    bool                  equal(iterator_adaptor const& x) const;
    void                  increment();
    void                  decrement();
    void                  advance(typename difference_type n);
    difference_type       distance_to(iterator_adaptor const& y) const;
protected:
    typedef some_define   iterator_adaptor_;    //自身的类型定义
    Adaptee const&        base_reference() const;
    Adaptee&              base_reference();
private:
      Adaptee             m_iterator;           //迭代器成员变量
};
```

iterator_adaptor 有六个模板参数，但通常我们只需要使用前两个。第一个模板参数 Derived 的含义同 iterator_facade，也是自定义的迭代器类型，即子类；第二个模板参数 Adaptee 是要被适配的、已经存在的类型（可以是已经存在的迭代器，或者是任何其他类型），其他的模板参数可以使用缺省值 boost::use_default 在编译期自动推导。

iterator_adaptor 是对 iterator_facade 的一个具体实现，因此它使用 iterator_core_access 作为友元，并借用 Adaptee 实现了 dereference()、increment()等六个核心操作。

iterator_adaptor 有两个构造函数，有参数的构造函数要求传入被适配的 Adaptee 实例，被保存在成员变量 m_iterator。为了方便子类使用，iterator_adaptor 提供 base_reference()和 base()，它们可以直接获得被适配的原始迭代器对象 m_iterator。

iterator_adaptor 还有一个类型定义 iterator_adaptor_，它就是 iterator_adaptor<Derived,...>自身，这是为了便于子类引用而不必写出一长串的模板参数列表，在写子类代码时可以直接使用。

5.5.2 用法

iterator_adaptor 对适配的对象有一定的要求：Adaptee 必须是可拷贝构造和可赋值的，但不一定要有 operator++，也就是说不一定是一个迭代器。

iterator_adaptor 是 iterator_facade 的子类，故它的用法与 iterator_ facade 也比较相似。不过因为我们要适配已经存在了的类型，所以通常只编写需要行为变化的少数核心操作函数就可以了，其他的操作已经由 iterator_adaptor 替我们实现了。

示例 1

下面的代码把普通数组指针适配为迭代器接口，因为指针本身已经具有了迭代器的操作，所以适配代码相当简单：

```
template<typename P>                                    //适配任意的指针类型
class array_iter:
  public boost::iterator_adaptor<array_iter<P>, P >  //适配类型 P
{
  BOOST_STATIC_ASSERT(is_pointer<P>::value);          //静态断言保证 P 必须是指针
public:
    typedef typename array_iter::iterator_adaptor_ super_type;   //基类定义
    array_iter(P x): super_type(x)                      //必要的构造函数
    {}
};
```

array_iter 的使用也非常简单，可以从一个原生指针很轻松地获得完全的迭代器功能：

```
int a[10] = {1,2,3};                              //一个整数数组
array_iter<int*> start(a), finish(a + 10);        //两个起点和终点迭代器
start += 1;                                       //起点迭代器前进一个位置

std::copy(start, finish,                          //copy 算法
  ostream_iterator<int>(cout, ","));              //流迭代器输出，用逗号分隔
```

改变 iterator_adaptor 的模板参数就可以改变适配后迭代器的功能，充分展现了 iterator_adaptor 强大而灵活的功能，例如下面的迭代器变成了前向迭代器，不能执行迭代器的算术运算：

```
template<typename P>
class array_iter:
  public boost::iterator_adaptor
      <array_iter<P>, P ,
      boost::use_default,                       //值类型使用缺省推导
      boost::forward_traversal_tag>             //变更迭代器的遍历类型
{
public:
```

```
  array_iter(P x):array_iter::iterator_adaptor_(x)        //直接使用基类
  {}
};
```

示例 2

我们还可以用 iterator_adaptor 实现一个 delta_iterator，它可以访问存储增量数值的容器，类似标准算法 partial_sum：

```
template<typename I>
class delta_iterator : public boost::iterator_adaptor<
  delta_iterator<I>, I ,
  typename std::iterator_traits<I>::value_type,           //值类型
  boost::single_pass_traversal_tag,                       //单向遍历标志
  typename std::iterator_traits<I>::value_type const >    //只读迭代器
{
private:
  friend class boost::iterator_core_access;               //必需的友元

  typedef delta_iterator                       this_type;
  typedef typename this_type::iterator_adaptor_ super_type; //基类定义

  typename super_type::value_type m_value;                //存储当前的值
public:
  explicit delta_iterator(const I& iter):          //构造函数,传入被适配的迭代器
     super_type(iter),  m_value(0)                        //当前值从 0 开始
  {}
private:
  using super_type::base;                                 //使用基类的成员函数
  using super_type::base_reference;

  typename super_type::reference dereference() const      //解引用操作
  {
     return m_value + *base();                            //当前值+增量
  }

  void increment()                                        //递增操作
  {
     m_value += *base();                                  //计算当前值
     ++base_reference();                                  //迭代器前进
```

```
  }
};                                                    //迭代器定义结束
```

delta_iterator 在迭代的同时做了一些计算的工作，用于访问存储Δ值的序列，可以这样使用：

```
vector<int> a = {1,2,3};                         //存储Δ值，实际的数值是1,3,6

typedef delta_iterator< decltype(a.cbegin())> delta_iter;
delta_iter start(a.begin()), finish(a.end());    //初始化迭代器

std::copy(start, finish,                         //拷贝到输出流
     ostream_iterator<int>(cout, ","));
```

我们将在接下来的 5.6 节中看到更多的 iterator_adaptor 应用例子。

5.6 迭代器工具

在 iterator_facade 和 iterator_adaptor 这两个辅助类的帮助下，iterators 库提供了很多个非常有用的迭代器，它们既是定义良好的迭代器工具，也是逻辑清晰的代码范例，值得我们通过仔细研究来深入理解迭代器概念。

这些迭代器工具大部分位于<boost/iterator/>目录，少量位于<boost/>目录，并且都提供形如 make_*xxx*_iterator()的工厂函数以方便使用。

5.6.1 共享容器迭代器

共享容器迭代器（shared_container_iterator）把一个被 boost::shared_ptr 管理的容器适配成迭代器的形式来操作，比直接用 shared_ptr 更加简单方便，它位于头文件<boost/shared_container_iterator.hpp>。

类摘要

shared_container_iterator 的类摘要如下：

```
template <typename Container>
class shared_container_iterator :
   public iterator_adaptor<                      //使用适配器 iterator_adaptor
     shared_container_iterator<Container>,       //迭代器名称
```

```
    typename Container::iterator>           //被适配的容器迭代器
{
  typedef typename Container::iterator   iterator_t;
  typedef boost::shared_ptr<Container>   container_ref_t;

  container_ref_t                        container_ref;
public:
  shared_container_iterator() {}

  shared_container_iterator(iterator_t const& x,container_ref_t const& c) :
    super_t(x), container_ref(c) {}
};
```

shared_container_iterator 的代码非常简单，和我们之前实现的 array_iter 非常相似，几乎没有什么自己的功能代码，仅仅是一个简单的适配，它的构造函数要求传入容器的迭代器和指向容器的共享指针（注意，不是 std::shared_ptr）。

用法

shared_container_iterator 非常适合于操作那些使用 shared_ptr 管理的容器，这些容器可以被安全地共享使用，shared_container_iterator“屏蔽”了对 shared_ptr 的操作，使共享容器用起来更像是标准容器。

下面的代码示范了 shared_container_iterator 的用法：

```
auto sv = boost::make_shared<vector<int> >(10);               //共享容器

typedef shared_container_iterator<vector<int>> sci_t;   //typedef 简化定义

sci_t first(sv->begin(), sv);                                  //迭代器起点
sci_t last (sv->end()  , sv);                                  //迭代器终点

std::fill(first, last, 9);                                     //调用 std::fill 算法
```

辅助函数

为了方便使用，shared_container_iterator 还提供一个辅助函数 make_shared_container_iterator()，它可以直接产生 shared_container_iterator：

```
make_shared_container_iterator(iter, container);
```

适当地使用 make_shared_container_iterator()，可以不必写出临时变量简化代码，例如：

```
std::fill(                    //直接创建共享容器迭代器，无须 typedef 和临时的迭代器变量
  make_shared_container_iterator(sv->begin(), sv),
  make_shared_container_iterator(sv->end()  , sv),   9);
```

另外一个辅助函数 make_shared_container_range()以 pair 的形式返回共享容器的两个端点，符合 boost.range 的概念（参见第 6 章），可以传递给使用 range 概念的算法，它的声明如下：

```
std::pair<...>
make_shared_container_range(container) {       //返回一个迭代器的 pair，即 range
  return    std::make_pair(
     make_shared_container_iterator(container->begin(),container),
     make_shared_container_iterator(container->end()  ,container));
}
```

5.6.2 发生器迭代器

发生器迭代器（generator iterator）可以把一个函数或者函数对象适配成**输入迭代器**，每次调用 operator++ 并解引用迭代器都会获得一个值，它位于头文件 <boost/generator_iterator.hpp>。

类摘要

generator_iterator 的类摘要如下：

```
template<class Generator>
class generator_iterator
  : public iterator_facade<                          //使用 iterator_facade
       generator_iterator<Generator>,
       typename Generator::result_type,              //值类型
       single_pass_traversal_tag,                    //单遍迭代器
       typename Generator::result_type const&        //常引用，可读不可写
    >
{
 public:
   generator_iterator() {}                           //构造函数
   generator_iterator(Generator* g) :                //构造函数
```

```
      m_g(g), m_value((*m_g)()) {}

    void increment()                                //递增操作
    {    m_value = (*m_g)(); }

    const typename Generator::result_type&
    dereference() const                             //解引用操作
    {   return m_value; }

    bool equal(generator_iterator const& y) const//相等操作
    {    return this->m_g == y.m_g && this->m_value == y.m_value; }

  private:
    Generator* m_g;                                 //函数对象指针
    typename Generator::result_type m_value;        //产生的值
};
```

generator_iterator 的实现代码相当简单，从它的 iterator_facade 模板参数可以看出它符合输入迭代器的定义（可读不可写的单遍迭代器），也仅实现了所需的最小迭代器核心操作。读者需要注意 generator_iterator 的构造函数，它要求传入一个发生器指针，而不是引用。

用法

generator_iterator 提供了一个辅助函数 make_generator_iterator()，用于直接创建发生器迭代器，它的声明如下：

```
template <class Generator>
inline generator_iterator<Generator>
make_generator_iterator(Generator & gen);
```

它的接口与 generator_iterator 的构造函数不同，不是指针而是引用，因此用起来更加方便，配合 auto 可以更进一步简化创建迭代器的类型声明。

下面的代码使用 generator_iterator 把随机数发生器适配成了迭代器的形式，可以很容易地使用迭代器的操作方式获得随机数：

```
rand48 rng;                                     //rand48 随机数发生器
auto iter = make_generator_iterator(rng);       //创建发生器迭代器
```

```
for (int i = 0;i < 5; ++i)
{     cout << *++iter << ",";}                               //输出 5 个随机数
```

注意在这里我们不能使用 std::copy 算法，因为 copy 算法需要知道迭代器的起点和终点，而这个随机数发生器迭代器的结束位置无法确定，如果要使用 copy 算法，那么可参考 5.6.5 节使用计数迭代器或者 5.6.6 节的函数输入迭代器。

5.6.3 逆向迭代器

reverse_iterator 把一个迭代器适配成可以逆序遍历的逆向迭代器，它位于头文件 <boost/iterator/reverse_iterator.hpp>。

类摘要

reverse_iterator 可以把原迭代器的前进后退操作转换为反向操作，类摘要如下：

```
template <class Iterator>
class reverse_iterator
    :public iterator_adaptor< reverse_iterator<Iterator>, Iterator >
{
public:
     reverse_iterator() {}
     explicit reverse_iterator(Iterator x)
        : super_t(x) {}
private:
     typename super_t::reference dereference() const     //解引用
     { return *boost::prior(this->base()); }              //注意 prior 的使用

     void increment() { --this->base_reference(); }      //递增操作
     void decrement() { ++this->base_reference(); }      //递减操作

     void advance(typename super_t::difference_type n)   //前进操作
     { this->base_reference() += -n;     }
};
```

因为 reverse_iterator 逆序迭代，因此要求适配的迭代器必须满足双向迭代器概念，即提供 operator--。

用法

reverse_iterator 提供函数 make_reverse_iterator()，可以轻松创建逆向迭代器，

下面的代码把原始指针适配成了逆向迭代器，然后逆序打印输出：

```
char s[] = "hello iterator.";                        //字符数组

std::copy(                                           //copy 算法
  make_reverse_iterator(s +                          //逆序迭代器起点
            char_traits<char>::length(s)),           //使用 char_traits 计算长度
  make_reverse_iterator(s),                          //逆序迭代器终点
  ostream_iterator<char>(cout));                     //流迭代器输出
```

程序的运行结果如下：

```
.rotareti olleh
```

5.6.4 间接迭代器

indirect_iterator 把一个迭代器进行适配，在执行 operator* 时再多执行一次解引用操作（即再执行一次 operator*），适合用于查看保存指针、智能指针或者迭代器的容器。它位于头文件<boost/iterator/indirect_iterator.hpp>。

类摘要

indirect_iterator 的类摘要如下：

```
template <
   class Iterator,                                   //被适配的迭代器
   class Value               = use_default,
   class CategoryOrTraversal = use_default,
   class Reference           = use_default,
   class Difference          = use_default >
class indirect_iterator
{
 public:
    indirect_iterator();                             //构造函数
    indirect_iterator(Iterator x);                   //构造函数

    Iterator const&       base() const;
    reference             operator*() const;
    indirect_iterator&    operator++();
```

```
    indirect_iterator&    operator--();
};
```

用法

indirect_iterator的用法很简单，同样提供工厂函数make_indirect_iterator()，需注意它只能用于元素是“指针”的容器：

```
vector<int*> v = {new int(1), new int(2)};              //存储指针，C++11 初始化

//不使用间接迭代器访问元素
for (auto pos = v.begin(); pos != v.end(); ++pos)
{      cout << **pos << ",";   }                          //需用两次解引用

//使用间接迭代器访问元素
auto start  = make_indirect_iterator(v.begin());
auto finish = make_indirect_iterator(v.end());
for(;start != finish;)
{      cout << *start++ << ",";    }                      //只用一次解引用

//需及时删除指针避免内存泄露
for_each(v.begin(), v.end(), checked_deleter<int>());
```

间接迭代器带来的好处很明显，它使我们对指针所指元素的操作透明化，令指针容器操作起来就像是一个普通容器。

读者可对比一下 5.6.1 节的共享容器迭代器：indirect_iterator 的处理对象是存储在容器中的指针或智能指针，而 shared_container_iterator 的处理对象是存储在智能指针中的容器，二者都“屏蔽”了中间的一层指针操作。

5.6.5 计数迭代器

counting_iterator 把一个**可递增**的类型适配成迭代器，它位于头文件<boost/iterator/counting_iterator.hpp>。

类摘要

counting_iterator 使用了元编程来计算迭代器类型等模板参数，简化的类摘要如下：

```
template <
```

```
    class Incrementable,                          //可递增类型
    class CategoryOrTraversal = use_default,
    class Difference = use_default>
class counting_iterator:
    public iterator_adaptor<
          counting_iterator<... >                 //计数迭代器自身
        , Incrementable                           //被适配的可递增类型
        , Incrementable const                     //值类型
        , traversal                               //迭代器分类
        , Incrementable const&                    //值引用类型
        , difference >                            //迭代器距离类型
{
public:
    counting_iterator();
    counting_iterator(counting_iterator const& rhs);
    explicit counting_iterator(Incrementable x);

    reference             operator*() const;
    counting_iterator&    operator++();
    counting_iterator&    operator--();
};
```

counting_iterator 最重要的模板参数是 Incrementable，对它的要求是可拷贝构造和可赋值的。Incrementable 必须能够执行 operator++操作，如果 counting_iterator 是单遍迭代器、双向遍历迭代器或者随机访问遍历迭代器，则 Incrementable 还应具备 operator==、operator--和算术运算的功能。

Incrementable 通常是整数，但也可以是任何符合以上要求的类型（例如迭代器），适配后 counting_iterator 为原类型增加了一个 operator*解引用操作，其他的操作均不变。

用法

counting_iterator 可以为一个类型增加解引用操作，把它变得像是一个迭代器，对于某些需要连续增加的类型来说很有用，通过 counting_iterator 适配后能够搭配标准库算法工作。同样地，它提供便捷的工厂函数 make_counting_iterator()。

示范 counting_iterator 基本用法的代码如下：

```
counting_iterator<int> i(100);                    //把 int 适配成计数迭代器
```

```
assert(*i++ == 100);                              //后++操作
assert(*i == 101);
assert(*++i == 102);                              //前++操作

vector<int> v;
std::copy(                                        //使用 copy 算法
  make_counting_iterator(0),                      //从 0 填充到 9
  make_counting_iterator(10),
  back_inserter(v)                                //插入迭代器适配器
  );
```

下面的代码把随机数发生器包装成了一个可递增的类型 rand_int，它符合 counting_iterator 对 Incrementable 的要求：可递增、可拷贝构造、可赋值、可比较：

```
template<typename R>                              //要求类型是随机数发生器
class rand_int
{
private:
  R &r;                                           //随机数的引用
  int count;                                      //个数统计，用于相等比较
public:
  rand_int(R& _r, int c = 0):                     //构造函数
      r(_r), count(c) {}
  rand_int(rand_int const &other) = default;              //拷贝构造函数
  rand_int& operator=(rand_int const &other) = default; //赋值函数

  void operator++()                               //递增操作，增加计数
  {   ++count;}

  //比较操作，比较计数数量
  friend bool operator==(rand_int const &l, rand_int const &r)
  {   return l.count == r.count;}

  //转型到整数的操作符，返回随机数
  operator typename R::result_type () const
  {   return r(); }
};
```

现在，rand_int 就可以被用于 counting_iterator，适配后可以向任意的容器填充随机数：

```
typedef counting_iterator<rand_int<rand48>,       //定义为单遍迭代器
```

```
                    boost::single_pass_traversal_tag, int> RandIter;

rand48 r;                                         //一个随机数发生器
rand_int<rand48> r1(r, 0), r2(r, 10);             //包装为可递增类型
RandIter first(r1), last(r2);                     //适配为计数迭代器

vector<int> v;
std::copy(first, last, back_inserter(v));         //使用 copy 算法
assert(v.size() == 10);
```

counting_iterator 的另一个使用例子可参见 5.6.8 小节。

5.6.6 函数输入迭代器

函数输入迭代器（function_input_iterator）很类似发生器迭代器（5.6.2 节），同样能够把一个函数或者函数对象适配成输入迭代器，但不同的是可以使用一个状态参数设定迭代器的起点和终点（有界），因而更容易使用，它位于头文件<boost/iterator/function_input_iterator.hpp>。

类摘要

function_input_iterator 使用了模板元编程技术，类摘要如下：

```
template <class Function, class Input>
class function_input_iterator
    : public mpl::if_<                              //注意模板元编程的使用
        function_types::is_function_pointer<Function>,
        impl::function_pointer_input_iterator<Function,Input>,
        typename mpl::if_<
            function_types::is_function_reference<Function>,
            impl::function_reference_input_iterator<Function,Input>,
            impl::function_input_iterator<Function,Input>
        >::type
    >::type
{
    typedef some_define base_type;
public:
    function_input_iterator(Function & f, Input i)  //构造函数
        : base_type(f, i) {}                        //初始化基类
```

```
};
```

function_input_iterator 使用 boost.function_types 库来提取模板参数 Function 的类型信息，再使用 mpl 模板元编程技术进行分支决策，在编译期决定从哪个基类继承，而它本身并没有实际的功能。

function_input_iterator 在名字空间 boost::impl 里定义了三个具体实现类，分别对应函数指针、函数引用和函数对象三种情形，这三个实现类的代码很类似，故下面仅以 boost::impl::function_input_iterator 为例。

boost::impl::function_input_iterator 的实现代码如下：[①]

```
template <class Function, class Input>
class function_input_iterator                   //位于 boost::impl 名字空间
    : public iterator_facade<
    function_input_iterator<Function, Input>,
    typename Function::result_type,             //值类型
    single_pass_traversal_tag,                  //单遍迭代器
    typename Function::result_type const &      //常引用，可读不可写
    >
{
public:
    function_input_iterator() {}                //构造函数
    function_input_iterator(Function & f_, Input state_ = Input())
        : f(&f_), state(state_), value((*f)()) {} //初始化成员变量

    void increment() {                          //递增操作
        value = (*f)();                         //调用函数产生值
        ++state;                                //状态变化
    }

    typename Function::result_type const &      //解引用操作
    dereference() const
    {   return value; }

    bool equal(function_input_iterator const & other) const  //相等比较
  {   return f == other.f && state == other.state; }         //比较状态值
```

① 实际的代码使用了 boost::optional，这里做了适当的简化。

```
private:
    Function * f;                                    //保存函数对象的指针
    Input state;                                     //状态
    typename Function::result_type value;            //解引用值
};
```

function_input_iterator 有两个模板参数：第一个 Function 与发生器迭代器一样，是一个具有 operator() 的可调用物，被用来产生迭代器的解引用值；第二个 Input 要求是一个可递增、可比较、可缺省构造和拷贝构造的类型，也就是说支持 operator++和 operator==，它被用来执行单遍迭代器的相等比较操作。

用法

function_input_iterator 用起来很像发生器迭代器和计数迭代器的混合体，一方面它可以把函数或函数对象适配为一个迭代器，另一方面它又能够使用额外的 state 参数进行计数，在到达终点时自动停止。它的便捷的工厂函数是 make_function_input_iterator()，有两种重载形式，分别针对函数指针和函数引用。

示范 function_input_iterator 基本用法的代码如下：

```
rand48 rng;                                          //rand48 随机数发生器

std::copy(                                           //使用 std::copy 算法
  make_function_input_iterator(rng, 0),              //从 0 开始
  make_function_input_iterator(rng, 5),              //到 5 结束
      ostream_iterator<int>(cout, "\n")              //标准流输出 5 个随机数
);
```

读者可以把这段代码与 5.6.2 节发生器迭代器的示范代码对比一下，这里因为可以自行控制迭代器的起点和终点状态，所以可以使用 std::copy 算法。

还应该注意 function_input_iterator 构造函数中的状态参数的使用，这里用的是最简单最常用的 int 类型，它当然满足可递增、可比较等要求，但也可以使用任意可递增可比较的类型，例如迭代器：

```
vector<int> v(10);                                   //一个大小为 10 的标准容器

std::copy(                                           //使用 std::copy 算法
  make_function_input_iterator(rng, v.begin()),      //使用迭代器定义起点
  make_function_input_iterator(rng, v.end()),        //使用迭代器定义终点
  v.begin()   );                                     //拷贝到标准容器
```

上面的代码使用了 vector 的迭代器作为 function_input_iterator 的状态参数，用随机数填满了容器，从而无须知道容器的确切大小。

function_input_iterator 还提供了一个特别的状态类型 infinite，它永远不会到达终点：

```
struct infinite {
    infinite & operator++()                          { return *this; }
    infinite & operator++(int)                       { return *this; }
    bool operator==(infinite &) const                { return false; };  //注意
    bool operator==(infinite const &) const          { return false; };  //注意
};
```

infinite 通过 operator==返回 false 导致状态比较总是不相等，因而函数输入迭代器可以无限地迭代产生值，相当于发生器迭代器。

5.6.7 函数输出迭代器

function_output_iterator 可以把一个单参函数或函数对象适配成一个标准的**输出迭代器**，它位于头文件<boost/function_output_iterator.hpp>。

类摘要

function_output_iterator 是一个比较特殊的迭代器，并没有使用 iterator_adaptor 或 iterator_facade，类摘要如下：

```
template <class UnaryFunction>
class function_output_iterator
{
public:
  typedef std::output_iterator_tag iterator_category;    //输出迭代器分类
  typedef void                     value_type;
  typedef void                     difference_type;
  typedef void                     pointer;
  typedef void                     reference;

  explicit function_output_iterator();
  explicit function_output_iterator(const UnaryFunction& f);
```

```
    output_proxy operator*();                          //解引用操作
    function_output_iterator& operator++();
  private:
    UnaryFunction m_f;                                 //函数对象
  };
```

function_output_iterator 的模板参数要求是可接受一个参数的单参可调用物，函数或者函数对象都可以，构造函数把函数保存在 private 成员变量 m_f 里，因此要求 UnaryFunction 必须是可拷贝构造和可赋值的。

解引用操作 operator*()是 function_output_iterator 的核心功能所在，它返回一个代理对象 output_proxy，把赋值操作转化为对函数的调用，因此*iter = t 就相当于 m_f(t)。

用法

使用 function_output_iterator 再配合标准算法 std::copy，我们就可以很容易地操作存储在容器中的所有元素，只需要把操作函数适配成迭代器即可。

下面的代码定义了一个转换 ASCII 码到十六进制数的函数对象 to_hex，它逐个地接受字符，再把它们转换成十六进制数存储在一个外部的 vector 中：

```
class to_hex
{
private:
    vector<unsigned char> &v;                        //存储十六进制数的容器
    int count;                                       //字符计数

    char trans(const char c) const               //从 ASCII 码转换到十六进制数
    {
        if      (c >= 'a') {   return c - 'a' + 10;}
        else if (c >= 'A') {   return c - 'A' + 10;}
        else               {   return c - '0';}
    }
public:
    to_hex(vector<unsigned char> &_v):               //构造函数
        v(_v), count(0){}
```

```
    void operator()(const char c)
    {
        static char tmp;
        if ((count++) % 2 == 0)
        {
            tmp = trans(c)* 0x10;
        }
        else
        {
            tmp += trans(c);
            v.push_back(tmp);
        }
    }
};
```

使用 function_output_iterator 适配 to_hex 后，我们就可以很容易地实现字符串到十六进制数（base16）的转换：[①]

```
int main()
{
    char s[] = "1234abcd";                          //base16 编码的字符串

    vector<unsigned char> v;                        //存储十六进制数的容器
    to_hex h(v);                                    //创建函数对象
    function_output_iterator<to_hex> foi(h);        //适配成迭代器

    std::copy(s, s + 8, foi);                       //调用 copy 算法，输出到函数对象
    assert(v.size() == 4);
}
```

使用工厂函数 make_function_output_iterator()，迭代器的创建也可以整合到 copy 算法中，用法更简单，代码如下：

```
std::copy(s, s + 8,                                //copy 算法
    make_function_output_iterator(to_hex(v)));     //工厂函数创建迭代器
```

5.6.8 过滤迭代器

filter_iterator 可以选择性地迭代序列，“筛选”出所需要的元素，如何选择则需要用

① Boost 库已经在<boost/algorithm/hex.hpp>中实现了一个十六进制转换的算法 hex/unhex，原理类似，读者可参考。

一个谓词（返回 bool 值的函数或函数对象）来决定，相当于对原序列提供了一个“子视图”，它位于头文件<boost/iterator/filter_iterator.hpp>。

类摘要

filter_iterator 的类摘要如下：

```
template <class Predicate, class Iterator>
class filter_iterator:
  public iterator_adaptor<...>                        //适配的细节省略
{
 public:
    filter_iterator();
    filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
    filter_iterator(Iterator x, Iterator end = Iterator());
    Predicate              predicate() const;

    Iterator               end() const;
    Iterator const&        base() const;
    reference              operator*() const;
    filter_iterator&       operator++();
private:
    Predicate              m_pred;                    //谓词
    Iterator               m_iter;                    //迭代器
    Iterator               m_end;                     //迭代器终点
};
```

filter_iterator 需要两个模板参数，第一个参数 Predicate 是过滤条件谓词，用于过滤元素，只有满足条件[Predicate (x) ==true] 才会被选择；第二个参数 Iterator 是被适配的迭代器类型，应该满足可读和单遍迭代器概念，filter_iterator 将使用它来迭代。

filter_iterator 的构造函数一般需要传递谓词 Predicate（如果是可缺省构造的那么也可以不必传入）和迭代器，因为在选择元素时可能造成迭代器越界，因此除传入迭代起点外还必须传入迭代终点。

用法

作为示范，我们使用 filter_iterator 来迭代筛选某个整数区间的质数。

首先，我们定义一个谓词函数 is_prime()：

```
bool is_prime(int x)                                //判断整数 x 是否是质数
{
  for (int i = 2; i < x / 2; ++i)
  {
      if (x % i == 0)
      {   return false;}
  }
  return true;
}
```

假设我们要筛选 10～100 区间的整数，那么就可以考虑使用 counting_iterator 来计数生成这些整数。又因为所有的偶数都不是质数，所以又可以用 5.4.3 节定义的步进迭代器 step_iterator 来跳过所有的偶数，形成一个嵌套形式的迭代器，最后再交给 filter_iterator 过滤。实现代码如下：

```
typedef counting_iterator<int> ci_t;      //计数迭代器
ci_t c1(11), c2(101);          //因为使用 step_iterator 迭代，故区间必须是奇数端点

typedef step_iterator<ci_t> si_t;         //步进迭代器，在计数迭代器上步进
si_t si1(c1), si2(c2);

//定义过滤迭代器，注意函数指针类型的写法，使用了 decltype
typedef filter_iterator<decltype(&is_prime), si_t> fi_t;

fi_t first(&is_prime, si1, si2);          //迭代器起点，si2 防止迭代越界
fi_t last (&is_prime, si2, si2);          //迭代器终点，si2 防止迭代越界

std::copy(first, last,                    //调用 copy 算法
  ostream_iterator<int>(cout, " "));      //流迭代器输出
```

同样地，使用工厂函数 make_filter_iterator()可以无须 typedef 类型，简化代码：

```
std::copy(
  make_filter_iterator(&is_prime, si1, si2),
  make_filter_iterator(&is_prime, si2, si2),
  ostream_iterator<int>(cout, " "));
```

5.6.9 转换迭代器

transform_iterator 把一个单参函数或函数对象应用于迭代的序列，解引用时使用函数来操作序列中的元素，效果与标准库的 transform 或 for_each 算法类似。它位于头文件

<boost/iterator/transform_iterator.hpp>。

类摘要

transform_iterator 的类摘要如下：

```
template <class UnaryFunction,                          //单参函数对象
          class Iterator,                               //被适配的迭代器
          class Reference    = use_default,
          class Value        = use_default>
class transform_iterator:
  public iterator_adaptor<...>                          //适配的细节省略
{
public:
  transform_iterator();
  transform_iterator(Iterator const& x, UnaryFunction f);

  UnaryFunction          functor() const;
  Iterator const&        base() const;
  reference              operator*() const;
  transform_iterator&    operator++();
  transform_iterator&    operator--();
private:
  UnaryFunction          m_f;                           //函数对象
};
```

transform_iterator 需要两个基本的模板参数：UnaryFunction 是一个单参的可调用物，同 function_output_iterator 一样要求是可拷贝构造和可赋值的；第二个参数 Iterator 则要求满足可读迭代器概念。

transform_iterator 的核心操作是 operator*()，它变动了 operator*所需的 dereference() 成员函数，解引用原迭代器再调用 UnaryFunction 操作，相当于 m_f(*this->base())，然后返回函数对象的转换结果（返回值）。

用法

transform_iterator 的用法和效果都非常类似于 std::transform 算法，可以在一个序列上迭代操作其中的所有元素，施加某些变动。

下面的代码先使用计数迭代器给容器赋初值，然后用 transform_iterator 把所有元素加上 5 输出到 cout，函数对象使用了 boost::bind 和 std::plus：

```
typedef counting_iterator<int> ci_t;            //使用计数迭代器向容器填充 10 个整数

vector<int> v;
std::copy(ci_t(0), ci_t(10), back_inserter(v));

auto f = bind(plus<int>(), _1, 5);              //使用 bind 创建函数对象，把整数加 5

std::copy(
  make_transform_iterator(v.begin(), f),
  make_transform_iterator(v.end(), f),
  ostream_iterator<int>(cout, " "));
```

这段代码对应的 transform 算法代码如下：

```
std::transform(v.begin(),v.end(),
               ostream_iterator<int>(cout, " "),f);
```

个人认为，transform_iterator 的 copy 算法用法更具有一致性，更容易理解。

5.6.10 索引迭代器

permutation_iterator 虽然直接翻译为“排序迭代器”，但它实际上并不执行真正的排序操作，只是改变了原有序列的索引顺序从而变动了迭代顺序。它位于头文件<boost/iterator/permutation_iterator.hpp>。

类摘要

permutation_iterator 的类摘要如下：

```
template< class ElementIterator, class IndexIterator>
class permutation_iterator :
  public iterator_adaptor< permutation_iterator, IndexIterator,...>
{
public:
  permutation_iterator();
  explicit permutation_iterator(ElementIterator x, IndexIterator y);

  reference                operator*() const;
  permutation_iterator&    operator++();
  ElementIterator const&   base() const;
private:
```

```
  ElementIterator          m_elt_iter;

  reference dereference() const                    //注意这里
    { return *(m_elt_iter + *this->base()); }
};
```

permutation_iterator 的第一个模板参数 ElementIterator 并不用于迭代，它只是作为一个迭代器基准来检索数据；第二个模板参数 IndexIterator 是排序索引所在的迭代器，它才是被 iterator_adaptor 适配的迭代器，定义了 permutation_iterator 迭代的区间和顺序。

permutation_iterator 的核心操作是 dereference()，它先对 IndexIterator 解引用，获得索引值，然后把索引值加上 ElementIterator 再解引用。

因为使用了迭代器的算术运算，permutation_iterator 要求 ElementIterator 必须满足随机访问遍历迭代器概念，而 IndexIterator 解引用返回的值应该可转换为 ElementIterator 的距离类型。

用法

permutation_iterator 使用 IndexIterator 为 ElementIterator 定义了一个区间，并且使用索引值重新排列了元素的顺序。区间的大小不一定与原序列相同，可以是一个子区间，区间中元素也可以重复。

示范 permutation_iterator 用法的代码如下：

```
char s[] = "abcdefg";                       //元素序列，有 7 个元素
int idx[] = { 6, 0, 2, 2, 4};               //索引序列，5 个索引，其中两个重复

std::copy(
  make_permutation_iterator(s, idx),
  make_permutation_iterator(s, idx + 5),
  ostream_iterator<char>(cout, " ")  );
```

这段代码定义了一个字符串序列 s 和一个索引序列 idx，idx 使用索引重新排列了 s 中的部分元素，然后我们使用工厂函数 make_permutation_iterator() 创建了两个索引迭代器。代码的运行结果如下：

```
g a c c e
```

它们分别是字符串 s 中的第 6、第 0、第 2、第 2 和第 4 个元素。

5.6.11 组合迭代器

zip_iterator 使用 tuple[①]对多个迭代器“打包”，可以同时移动所有被打包的迭代器，解引用 zip_iterator 将返回持有多个迭代器解引用结果的 tuple，它位于头文件<boost/iterator/zip_iterator.hpp>。

类摘要

zip_iterator 使用了模板元编程技术，简化的类摘要如下：

```
template<typename IteratorTuple>
class zip_iterator :
  public iterator_facade<...>                          //适配的细节省略
{
public:
  zip_iterator();
  zip_iterator(IteratorTuple iterator_tuple);
  const IteratorTuple& get_iterator_tuple() const;

  reference         operator*() const;
  zip_iterator&     operator++();
  zip_iterator&     operator--();

private:
  IteratorTuple     m_iterator_tuple;
};
```

zip_iterator 只有一个模板参数 IteratorTuple，它是一个迭代器引用类型的 tuple。tuple 里可以包含任意多个迭代器，这些迭代器都应该满足可读迭代器的概念，zip_iterator 的 operator*()将返回 IteratorTuple 中的迭代器各自解引用后的 tuple。

由于 zip_iterator 组合了多个迭代器，构造它比较麻烦，因此，通常需要使用工厂函数 make_zip_iterator()和 make_tuple()来简化创建过程。

① tuple 即元组，是一个可以容纳不同类型元素的数据结构，见推荐书目[3]。

用法

zip_iterator 最常见的应用场景是需要同时对多个序列进行迭代操作，然后把这些迭代结果组合起来。为了操作组合后的迭代结果，通常需要编写一个单参函数对象，它的参数也是 tuple，然后我们就可以使用 for_each、transform 算法或者 transform_iterator 来迭代 zip_iterator 来得到最终的计算结果。

下面我们使用 zip_iterator 来实现 5.6.7 节的由 ASCII 码转换到十六进制数的功能，读者可对比二者实现的异同。

首先我们需要定义一个新的函数对象 to_hex2，它的 operator() 使用 tuple 作为参数：

```
class to_hex2
{
private:
  vector<unsigned char> &v;
  char trans(const char c) const
  { ...}                                        //同 to_hex 的实现
public:
  to_hex2(vector<unsigned char> &_v):           //构造函数
    v(_v){}

  //定义 tuple 类型
  typedef boost::tuple<const char&, const char& > Tuple;
  void operator()(Tuple const& t) const         //用 tuple 接受多个参数
  {
        static char tmp;                        //代码与 to_hex 的类似
        tmp = trans(get<0>(t))* 0x10;
        tmp += trans(get<1>(t));
        v.push_back(tmp);
  }
};
```

接下来我们还需要使用 5.4.3 节定义的步进迭代器 step_iterator 来在字符串上跳跃迭代奇偶位置，并在 for_each 算法中使用 zip_iterator，嵌套工厂函数来创建迭代器，代码如下：

```
char s[] = "1234aBcD";                          //base16 编码字符串
vector<unsigned char> v;                        //存储十六进制数的容器
```

```
typedef step_iterator<const char*> si_t;                    //使用步进迭代器

for_each(                                                    //for_each 算法
  make_zip_iterator(                                         //使用工厂函数简化代码
      boost::make_tuple(si_t(s), si_t(s + 1))),              //两个起点
  make_zip_iterator(
      boost::make_tuple(si_t(s + 8), si_t(s + 9))),          //两个终点
  to_hex2(v)                                      //传递以 tuple 为参数的函数对象
  );
assert(v.size() == 4);
```

for_each 算法也可以用 copy 算法搭配 function_output_iterator 替换，效果相同：

```
std::copy(
  make_zip_iterator(                                         //组合迭代器
      boost::make_tuple(si_t(s), si_t(s + 1))),
  make_zip_iterator(                                         //组合迭代器
      boost::make_tuple(si_t(s + 8), si_t(s + 9))),
  make_function_output_iterator(to_hex2(v))                  //函数输出迭代器
  );
```

5.7 总结

在本章中，我们深入研究了 Boost 对 C++迭代器概念的重要贡献。

首先，我们回顾了迭代器设计模式，它是 C++和其他所有编程语言迭代器实现的理论基础。然后，我们讨论了标准库和 Boost 定义的迭代器分类，不同的迭代器在功能上有很大的区别，了解这些区别有助于我们编写正确的代码。C++标准定义了五类迭代器，但并不完美，存在一些小的缺陷，而 Boost 定义的新式迭代器则弥补了标准的不足，给出了一个更合理的解决方案。

next_prior 和 iterator_traits 是两个很小的工具，它们改进了标准库的迭代器工具，用起来更加容易。next_prior 组件基于 std::advance() 实现了可任意前进和后退的 next() 和 prior() 函数；iterator_traits 基于 std::iterator_traits，把非标准元函数转换为标准元函数。虽然它们都只做了很少的工作，但的确可以简化迭代器的使用。

iterators 库提供两个重要的类：使用外观模式的 iterator_facade 和使用适配器模式

的 iterator_adaptor，可以帮助程序员更容易地构造符合标准的迭代器，从而能够以统一的算法+迭代器的解法操作各种数据，而且这种解法十分优雅。

iterators 库里还有更多的迭代器工具，特别是迭代器适配器，可以把它们任意组合（类似于装饰模式）得到许多神奇的效果。这些迭代器工具可大致分为如下几个类别，有助于我们更好地掌握它们的用途。

- 输入型迭代器 ：发生器迭代器、计数迭代器和函数输入迭代器。
- 输出型迭代器 ：函数输出迭代器。
- 解引用型迭代器：共享容器迭代器和间接迭代器。
- 排序型迭代器 ：逆向迭代器和索引迭代器。
- 其他类型迭代器：过滤迭代器、转换迭代器和组合迭代器。

第6章 区间

算法是C++标准库里的重要角色，可以非常灵活地处理容器或者字符串，但使用的时候常常要指定一对首末位置的迭代器，显得有些麻烦。另外在一些时候，我们只想操作容器里的一部分元素，但标准库并没有提供这样的接口，只能将其拷贝到一个临时容器，增加了不必要的运行成本。

range库在迭代器和容器之上抽象出了“区间”的概念，简化了对算法和容器的操作，并被其他Boost库所使用，其中的部分工具已经被收入C++11标准。

range位于名字空间boost，需要包含头文件<boost/range.hpp>，即：

```
#include <boost/range.hpp>
using namespace boost;
```

6.1 概述

区间（range）是一个比较宽松的概念，它基于迭代器和容器，但要求却比容器低得多，不需要容纳元素，只含有区间的两个首末端点位置，像是一个容器的“视图”，可以简单地想象为一个迭代器的pair（std::pair<I,I>）。

区间都是左闭右开的，可以用成员函数begin()/end()或者自由函数begin()/end()获得其两个端点。range库目前内建支持的有以下几类。

- 标准容器或Boost容器。
- 元素是迭代器的std::pair。
- 原生数组或者字符串。

- range 库自带的 iterator_range 及其子类（6.5 节）。

元函数 has_range_iterator 可以判断一个类型是否是区间：

```
assert(has_range_iterator<vector<int>>::value);        //vector 是区间
assert(has_range_iterator<string>::value);             //string 是区间
assert(!has_range_iterator<stack<int>>::value);        //stack 不是区间

typedef boost::array<char, 5> array_t;                 //boost.array
assert(has_range_iterator<array_t>::value);            //是区间

typedef pair<int*,int*> pair_t;                        //指针的 pair
assert(has_range_iterator<pair_t>::value);             //是区间

char a[] = "range";                                    //原生数组
assert(has_range_iterator<decltype(a)>::value);        //是区间

assert(!has_range_iterator<char*>::value);             //指针类型不是区间
```

range 库完全基于 Boost 新式迭代器概念，区间也可以分为单遍区间、前向区间、双向区间和随机访问区间，所以可以使用概念检查类来更精确地判断类型，请读者参考 12.3.7 节。

6.2 特征元函数

类似于 iterator_traits，range 库提供若干元函数来获取区间的特征。

- range_iterator<R>　　：返回区间的迭代器类型。
- range_value<R>　　：返回区间的值类型。
- range_reference<R>　　：返回区间的引用类型。
- range_pointer<R>　　：返回区间的指针类型。
- range_category<R>　　：返回区间的迭代器分类。
- range_size<R>　　：返回区间的长度类型（无符号整数）。
- range_difference<R>　　：返回区间的距离类型（有符号整数）。
- range_reverse_iterator<R>：返回区间的逆向迭代器（仅对双向区间）。

这些元函数中最基本的是 range_iterator，它根据模板参数的常量性分别转交给元函数 range_const_iterator 和 range_mutable_iterator，后两者再根据标准容器、pair 和数组进行模板特化计算：

```
template< typename C >                           //这里的代码存在简化
struct range_iterator :                          //元函数转发
    mpl::eval_if_c<                              //元计算区间的迭代器类型
        is_const<C>::value,                      //类型是否是常量
        range_const_iterator<C>,
        range_mutable_iterator<C> >::type
{};
```

其他的元函数使用 iterator_traits 等迭代器工具来计算得到其他的特征，例如：

```
template< class T >
struct range_value : iterator_value< typename range_iterator<T>::type >
{ };
```

区间特征元函数主要用于 range 库的内部实现，这里不再举例说明其用法。

6.3 操作函数

range 库抽象了对区间的操作，在 boost 名字空间里给出了若干类似容器的操作。[①]

- begin()　　：返回区间的起点。
- end()　　　：返回区间的终点。
- rbegin()　：返回逆向区间的起点（双向区间）。
- rend()　　：返回逆向区间的终点（双向区间）。
- empty()　　：判断区间是否为空。
- distance()：返回区间两端的距离。
- size()　　：返回区间的大小。

这些操作函数中 begin() 和 end() 是最基本的，其他函数都基于这两个函数返回的迭代器

① begin()、end() 等四个操作另有前缀 const_ 的版本，功能相同。

进行计算，例如：

```
template< class T >
inline bool empty( const T& r )                          //判断区间是否为空
{
    return boost::begin( r ) == boost::end( r );
}
```

由于在区间处理中 begin() 和 end() 函数很重要，所以它们被收入了 C++11 标准（C++11.24.6.5），声明是：

```
//容器的起点和终点，注意新的 auto+decltype 函数返回类型的声明方式
template <class C> auto          begin(C& c) -> decltype(c.begin());
template <class C> auto          end(C& c)   -> decltype(c.end());

//数组的起点和终点
template <class T, size_t N> T*  begin(T (&array)[ N] );
template <class T, size_t N> T*  end(T (&array)[ N] );
```

在其他几个函数中需要注意区分的是 distance() 和 size()，这两个函数功能很相似，都是根据区间的起点和终点计算。不同的是前者的计算方法是 std::distance(begin(r), end(r))，返回类型是有符号类型 range_difference<R>::type，而后者通常会调用 r.size()，返回类型是无符号类型 range_size<R>::type。[①]

因为标准容器、迭代器的 pair 和数组都满足区间概念，所以我们可以用这些区间操作函数写出统一处理它们的代码，不必纠结于类型是否有成员函数 begin()/end()：

```
template<typename R>
void my_sort(R& r)                                       //模板参数是区间类型
{
    std::sort(boost::begin(r), boost::end(r));           //获取区间的端点，调用标准算法
}
vector<int> v{ 3,4,8,1,7};                               //标准容器
my_sort(v);                                              //以区间的方式使用
```

6.4 标准算法

基于以上的区间操作函数和处理思路，range 库在名字空间 boost 里提供了所有标准算法

① 如果 range 没有成员函数 size()，那么 size() 会退化为 distance()。

的区间版本，但它们并不在头文件<boost/range.hpp>里，而是分成了三个头文件.

- <boost/range/algorithm.hpp> ：C++中的所有标准算法。
- <boost/range/algorithm_ext.hpp> ：一些扩展算法，如 erase、itoa。
- <boost/range/numeric.hpp> ：数值标准算法。

这些区间算法的参数除了把两个迭代器改为一个区间外，与标准算法基本相同，例如：

```
template< class SinglePassRange, class Value >
inline range_difference<SinglePassRange>::type        //使用元函数计算返回类型
count(SinglePassRange& rng, const Value& val)         //count 算法
{
    return std::count(boost::begin(rng), boost::end(rng), val);//调用标准算法
}
```

区间算法的使用和标准算法一样，但因为不用写出两个迭代器位置代码会更加简单：

```
vector<int> v{ 8,1,16,3,7,3,42};                  //标准 vector 容器，也是区间

assert(boost::count(v, 3) == 2);                  //统计元素数量
assert(boost::find(v, 7) != v.end());             //查找元素

boost::sort(v);                                   //快速排序
assert(boost::adjacent_find(v) != v.end());       //查找连续相等的元素
```

由于标准算法数量庞大，本书不可能详细列举，读者可参考推荐书目[2] 了解这些算法的具体作用。

在通常情况下，区间算法与标准算法的返回类型相同（很多时候算法的返回值都被忽略），但有的区间算法的重载形式却可以返回一个区间类型，能够搭配其他算法使用。这些算法又可以分为两类，第一类算法返回原区间，第二类算法使用一个模板参数定制返回的区间。

6.4.1 返回原区间的算法

第一类返回原区间的算法都是变动性算法（简单起见未列出_copy 版本），包括以下几种。

- fill/fill_n。
- generate/generate_n。
- inplace_merge。

- random_shuffle。
- replace/replace_if。
- reverse/rotate。
- sort/stable_sort。
- partial_sort/nth_element。
- make_heap/sort_heap/push_heap/pop_heap。

这些算法处理整个区间，然后再返回原区间。由于区间类型里含有两个首末位置的迭代器，所以返回值能够直接被其他区间算法所利用，以类似于函数式编程的方式嵌套使用，例如：

```
vector<int> v(10);                          //一个整数区间
rand48 rnd;                                 //随机数发生器

boost::sort(                                //3.排序
    boost::random_shuffle(                  //2.随机打乱
        boost::generate(v, rnd)));          //1.用随机数填充区间
```

6.4.2 返回定制区间的算法

第二类算法都与某种形式的查找/分割操作有关（同样未列出_copy 版本），包括以下几种。

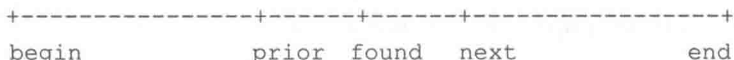
- find/find_if。
- find_first_of/find_end/adjacent_find。
- search/search_n。
- remove/remove_if。
- unique。
- partition/stable_partition。
- lower_bound/upper_bound。
- min_element/max_element。

这些算法的普通版本返回一个处理后的迭代器位置(例如 find 算法返回第一个找到的元素位置)，我们可以将这个迭代器位置称为 found，依据 found 位置可以得到两个常用的子区间。

■ [begin(rng), found)：起点到返回位置的区间。

■ [found, end(rng)) ：返回位置到终点的区间。

除了 found 位置，range 库还定义了 next(found) 和 prior(found) 位置，这五个位置（含起点和终点）和区间分割可以用下面的示意图表示.



range 库在 boost 名字空间里声明了一个枚举类型 range_return_value，作为这些区间算法的模板参数，来为元计算定制返回的区间类型：

```
enum range_return_value
{
    return_found,                   //返回 found 位置，同标准算法
    return_next,                    //返回 next 位置
    return_prior,                   //返回 prior 位置
    return_begin_found,             //返回区间[ begin, found)
    return_begin_next,              //返回区间[ begin, next)
    return_begin_prior,             //返回区间[ begin, prior)
    return_found_end,               //返回区间[ found, end)
    return_next_end,                //返回区间[ next, end)
    return_prior_end,               //返回区间[ prior, end)
    return_begin_end                //返回区间[ begin, end)
};
```

区间算法使用元函数 range_return 计算返回的区间，例如 find 算法：

```
template< range_return_value re, class SinglePassRange, class Value >
range_return<const SinglePassRange,re>::type          //元计算返回类型
find( const SinglePassRange& rng, const Value& val );
```

range 库的这种设计极大地增强了区间算法的灵活性，我们可以随意获取任意的迭代器位置或者区间处理，比如标准库中常见的删除元素操作：[①]

```
vector<int> v{ 8,1,16,3,7,3,42};                 //初始化 vector，有两个 3

boost::erase(v,                                  //boost::erase 算法
  boost::remove<return_found_end>(v, 3));        //返回区间[ found, end)
```

① range 库已经为这个常用的操作提供了专门的算法 remove_erase/remove_erase_if。

```
assert(boost::find(v, 3) == v.end());                //此时 3 已经被彻底删除
boost::copy(v, ostream_iterator<int>(cout, ","));    //boost::copy 算法流输出
```

这段代码对应的标准算法代码是：

```
v.erase( std::remove(v.begin(),v.end(), 3), v.end());
```

对比一下可以看出区间算法的代码含义更清晰明确，不容易出错。

6.5 迭代器区间类

区间概念本质上就是一对迭代器，可以用 std::pair<I,I>来表示，但 std::pair 过于简单，也不能明确地表述区间的含义，所以 range 库提供了一个专门的区间类：iterator_range。

iterator_range 同 std::pair<I,I>类似，封装了两个迭代器，但它的功能更丰富，接口像是一个标准容器，但非常轻量级。

6.5.1 类摘要

iterator_range 的类摘要如下：

```
template<class IteratorT>                                  //模板参数是迭代器类型
class iterator_range
{
public:
   typedef IteratorT   iterator;                             //类型定义
   typedef IteratorT   const_iterator;
public:
   iterator_range();                                         //构造函数
   iterator_range( Iterator Begin, Iterator End );
public:
   iterator      begin() const;                              //区间起点
   iterator      end() const;                                //区间终点
public:
   operator      unspecified_bool_type() const;              //隐式 bool 转型
   bool          operator!() const;

   bool          equal( const iterator_range& ) const;       //区间相等比较
   bool          empty() const;                              //区间是否为空
```

```
    value_type&    front() const;                          //区间首元素
    value_type&    back() const;                           //区间尾元素

    void           drop_front();                           //弹出首元素
    void           drop_back();                            //弹出尾元素

    iterator_range& advance_begin(difference_type n);      //区间位置移动
    iterator_range& advance_end(difference_type n);

    // 下面三个成员函数仅适用于随机访问区间
    reference      operator[]( difference_type at ) const;
    value_type     operator()( difference_type at ) const;
    size_type      size() const;
private:
    IteratorT      m_Begin;                                //区间起点
    IteratorT      m_End;                                  //区间终点
protected:
    typedef iterator_range iterator_range_;            //可用于子类的类型定义
};
```

正如名字所表示的，iterator_range 是“迭代器的区间”，它的模板参数是迭代器类型，用两个成员 m_Begin 和 m_End 保存了区间的起点和终点，相当于 pair 的 first 和 second，但含义更明确。

iterator_range 的成员函数 begin()/end()返回区间的首末位置，也就是 m_Begin 和 m_End，drop_front()、drop_back()、advance_begin()、advance_end()这四个函数分别以不同的方式移动区间的起点和终点，它们并不真正地操作元素。[①]

对于随机访问区间，iterator_range 提供 operator[]和 operator()，两者基本功能相同，都返回 at 位置上的元素，但 operator[]返回的是引用类型，而 operator()返回的是值类型。

此外，iterator_range 还重载了大量的逻辑运算符，支持各种比较操作。

6.5.2 用法

iterator_range 可以直接从两个迭代器或者一个区间/容器来构造：

① 旧接口 pop_front()、pop_back()已被废弃。

```
vector<int> v(10);
typedef iterator_range<vector<int>::iterator> vec_range; //区间类型定义

vec_range r1(v);                                    //从容器构造一个区间
assert(!r1.empty());                                //区间不空
assert(r1.size() == 10);                            //取区间的大小

int a[10];                                          //原生数组
typedef iterator_range<int*> int_range;             //区间类型定义
int_range r2(a, a + 5);                             //从两个迭代器构造区间
assert(r2.size() == 5);
```

直接创建 iterator_range 需要手工指定迭代器类型，很多时候显得很麻烦，所以 range 库提供工厂函数 make_iterator_range()，可以搭配 C++11/14 的关键字 auto 来自动推导类型。它有三种重载形式：

```
make_iterator_range( IteratorT Begin, IteratorT End );  //两个迭代器构造区间

make_iterator_range( ForwardRange& r );                 //区间或容器构造区间

make_iterator_range( Range& r,                          //区间，还需再指定
        range_difference<Range>::type advance_begin,    //起点
        range_difference<Range>::type advance_end);     //和终点
```

make_iterator_range()配合 auto 使用起来非常方便：

```
vector<int> v(10);
auto r1 = make_iterator_range(v);                       //从区间构造
assert(has_range_iterator<decltype(r1)>::value);

int a[10];
auto r2 = make_iterator_range(a, a + 5);                //从迭代器构造

auto r3 = make_iterator_range(a, 1, -1);                //指定迭代器的位置
assert(r3.size() == 8);
```

range 库另外还有一个模板函数 copy_range，它可以把区间里的元素拷贝到一个新的容器里：

```
template< typename SeqT, typename Range >
inline SeqT copy_range( const Range& r );
```

iterator_range 是区间的标准表示，被 Boost 的许多组件作为基础所使用，如

for_each、string_algo、iostreams 等，这里仅以 string_algo 为例简单示范其用法：

```
char a[] = "iterator range";

auto r = boost::find_first(a, " ");                  //使用 find_first 字符串算法
assert(r.front() == ' ');                             //返回查找到的区间

auto r2 = make_iterator_range(a, r.begin());         //取字符串的前半部分
assert(copy_range<string>(r2) == "iterator");        //copy_range 函数
```

6.6 辅助工具

基于 iterator_range，range 库实现了几个很有用的工具类和函数。

6.6.1 sub_range

sub_range 是 iterator_range 的子类，它的主要目的是表示一个子区间，更接近容器。

sub_range 的类摘要如下：

```
template< class ForwardRange >                        //模板参数是一个前向区间
class sub_range : public ...
{
public:
   sub_range();                                       //构造函数
   sub_range( Iter first, Iter last );
public:
   iterator          begin();
   iterator          end();
   difference_type   size() const;

public:
   value_type&       front();
   value_type&       back();

   value_type&       operator[] ( size_type at ); //仅适用于随机访问区间
```

```
};
```

sub_range 与 iterator_range 的主要区别是模板参数，它的模板参数是一个前向区间类型，而 iterator_range 则是迭代器类型，因此它更容易配合标准容器使用。

由于 sub_range 是 iterator_range 的子类，因此 sub_range 具有 iterator_range 的全部功能。

6.6.2 counting_range

counting_range()函数位于头文件<boost/range/counting_range.hpp>，它使用 5.6.5 节的计数迭代器 counting_iterator，返回一个计数迭代器的区间，类似于 C++11/14 的 itoa 算法：

```
template<class Value>
inline iterator_range<counting_iterator<Value> >
counting_range(Value first, Value last);     //从 first 到 last 计数

template<class Range>
inline iterator_range<counting_iterator< range_value<const Range>::type> >
counting_range(const Range& rng);            //以区间的端点值计数
```

counting_range()的用法与 counting_iterator 类似，但需要搭配 boost::copy 算法，并且写法也简便得多：

```
boost::copy(                                  //输出 0～9 共 10 个整数
    counting_range(0,10),  ostream_iterator<int>(cout, ","));
```

6.6.3 istream_range

istream_range()函数位于头文件<boost/range/istream_range.hpp>，封装了输入流迭代器 std::istream_iterator，它可以把输入流的输入过程转化成一个区间。

istream_range()的实现摘要如下：

```
template<class Type, class Elem, class Traits> inline
    iterator_range<std::istream_iterator<Type, Elem, Traits> >
istream_range(std::basic_istream<Elem, Traits>& in)
{
    return iterator_range<std::istream_iterator<Type, Elem, Traits> >(
        std::istream_iterator<Type>(in),
```

```
        std::istream_iterator<Type>());
}
```

下面的代码从输入流读取整数，再写入到输出流，直到遇到非整数输入：

```
boost::copy(istream_range<int>(cin), ostream_iterator<int>(cout, ","));
```

6.6.4 irange

irange()位于头文件<boost/range/irange.hpp>，它使用 iterator_facade 定义了两个可递增的整数迭代器 integer_iterator 和 integer_iterator_with_step，然后产生递增的整数区间。它很类似 counting_range，但多了指定步长的功能：

```
template<typename Integer>
integer_range<Integer>
irange(Integer first, Integer last);                        //递增步长为 1

template<typename Integer, typename StepSize>
strided_integer_range<Integer>
irange(Integer first, Integer last, StepSize step_size);//指定递增步长
```

下面的代码示范了 irange()的用法，第一行功能同 counting_range，第二行指定了步长：

```
boost::copy(irange(0,10), ostream_iterator<int>(cout, ","));

boost::copy(irange(0,20,2), ostream_iterator<int>(cout, ","));
```

6.6.5 combined_range

combined_range 位于头文件<boost/range/combine.hpp>，它利用了 5.6.11 节里的 zip_iterator，可以把多个区间“打包”为一个区间，相当于 zip_iterator 的区间强化版本。

combined_range 的类摘要如下：

```
template<typename IterTuple>
class combined_range
      : public iterator_range<zip_iterator<IterTuple> >
{
   typedef iterator_range<zip_iterator<IterTuple> > base;
public:
```

```
    combined_range(IterTuple first, IterTuple last)
        : base(first, last)
    {}
};
```

工厂函数 combine()可以直接将多个区间组合为一个 combined_range 区间：

```
template<typename... Ranges>
combined_range<...> combine(Ranges&&... rngs);   //C++11/14 可变参数模板
```

使用 combined_range 可以很容易地操作多个区间，在多个区间上同步地处理数据：

```
string str = "abcde";                           //标准字符串
vector<int> v = {5,4,3,2,1};                    //向量容器

auto r = combine(str, v);                       //组合两个区间

for(const auto& x : r)                          //可以使用新式 for 遍历,需要用 const
{
    auto& c = get<0>(x);                        //解开 tuple
    auto& i = get<1>(x);
    cout << "(" << c << "," << i << ") ";
}
```

但需要注意，combined_range 组合的区间必须具有同样的大小，如果区间的长度不一致，那么在迭代器前进时就可能发生逾尾，导致未定义错误，例如：

```
string s2 = "xyz";                              //一个较短的区间
auto r2 = combine(s2, v);                       //危险！与较长的区间组合
```

6.6.6 any_range

any_range 位于头文件<boost/range/any_range.hpp>，它使用了类型擦除技术，可以应用在符合要求的“任意”容器上。

any_range 的类摘要如下：

```
template<
    class Value                                 //值类型
  , class Traversal                             //迭代器遍历类型
  , class Reference = Value&                    //引用类型
  , class Difference = std::ptrdiff_t
```

```
  , class Buffer = use_default
>
class any_range: public iterator_range<...>
{...};
```

any_range 主要使用前两个模板参数，分别确定了区间的值类型和遍历类型。它并不关心区间之下的容器的具体类型，只要容器满足 Value 和 Traversal 的要求即可，例如：

```
typedef any_range<int,                                //区间的值类型是 int
  boost::single_pass_traversal_tag> range_type;  //区间要求可单向遍历

list<int> l = {1,3,5,7,9};                             //标准链表容器

range_type r(l);                                       //使用 any_range 处理 list

for(const auto& x : r)                                 //遍历区间
{    cout << x << ",";}

vector<int> v = {2,4,6,8,0};                           //标准向量容器

r = v;                                                 //使用 any_range 处理 vector

for(const auto& x : r)                                 //遍历区间
{    cout << x << ",";}
```

在这段代码里，我们定义了一个单遍整数的 any_range 区间，它可以任意处理 list、vector 甚至是 set 等容器，因为这些容器都满足基本的单遍迭代器概念。

range 库还提供一个工厂元函数 any_range_type_generator，它可以由一个区间或容器类型产生合适的 any_range：

```
template<
   class WrappedRange                                //区间要包装的类型
  , class Value = use_default                       //值类型
  , class Traversal = use_default                   //迭代器遍历类型
  , class Reference = use_default                   //引用类型
  , class Difference = use_default
  , class Buffer = use_default
>
struct any_range_type_generator
{
```

```
  typedef any_range<...> type;                    //元计算返回 any_range
};
```

any_range_type_generator 的用法比较简单，通常只要提供第一个模板参数区间或容器类型即可，其余的值类型、遍历类型会使用 range_value、range_reference 等元函数自动推导，最后我们使用::type 即可得到 any_range 类型，例如：

```
typedef any_range_type_generator<                 //使用工厂元函数
            decltype(l)>::type range_type;        //区间使用 list 类型
```

但使用 any_range_type_generator 时要注意它“完全”计算了模板参数 WrappedRange 的各种属性并应用于 any_range，有时可能会降低 any_range 的适配性，导致 any_range 只能用在特定 Traversal 属性的容器上：

```
typedef any_range_type_generator<                 //使用工厂元函数
            decltype(v)>::type range_type;        //注意，区间使用 vector 类型
```

上面的代码使用 vector 来产生 any_range，会使 any_range 只能遍历拥有随机访问遍历迭代器的容器，不能处理 list、set 等其他标准容器。

6.7 适配器

类似于迭代器的适配器，range 库也提供区间适配器，可以把一个区间适配成另一个区间。由于区间的能力基于迭代器，因此变动迭代器也就变动了区间，使区间不仅能表达元素的范围，也可以处理元素。

同迭代器类似，区间适配器使用 boost::copy（当然也可以使用其他区间算法）来驱动数据通过区间和迭代器流动，而且由于区间的自表达特性，它还重载了 operator|()，支持以类似 UNIX 管道操作符的形式连接多个区间，每个区间都可以执行一定的工作。[①]

range 库的适配器位于名字空间 boost::adaptors，需要包含头文件<boost/range/adaptors.hpp>。

6.7.1 适配器列表

区间适配器都是一些重载了 operator|()的特殊对象，而且大量使用了 5.5 节和 5.6 节

① 区间适配器也可以使用函数的形式，但显然没有管道操作符方便，故本书不做介绍。

的迭代器工具，在与区间执行 operator|()后返回一个新的区间。

range 库提供的区间适配器包括以下几种。

- adjacent_filtered(P)：使用谓词 P 过滤两个邻接的元素。
- copied(n,m) ：取子区间[n,m)，只能用于随机访问区间。
- filtered(P) ：使用谓词 P 过滤元素。
- indexed(i) ：为迭代器添加一个从 i 开始的索引号。
- indirected ：类似于 indirect_iterator，多一次解引用操作。
- map_keys ：取出 map 容器里的 key。
- map_values ：取出 map 容器里的 value。
- replaced(x,y) ：把区间里的 x 替换为 y。
- replaced_if(P,v) ：把区间里符合谓词 P 的元素替换为 v。
- reversed ：逆序区间。
- sliced(n,m) ：同 copied。
- strided(n) ：在区间上以步长 n 跳跃前进。
- tokenized() ：使用 boost::regex 正则处理，本书不作介绍。
- transformed(F) ：类似于 std::transform 算法，用 F 处理每一个元素。
- uniqued ：过滤掉相邻重复的元素。

6.7.2 用法

区间适配器的用法相当简单，可以把原始区间想象成一个数据源，经过适配器后得到处理的数据，最后再由算法来处理这些数据，形式与第 11 章的流处理非常相似。

在下面的示范代码中，我们把区间算法和区间适配器混合使用：

```
vector<int> v{ 7,8,4,6,53,2,6};                    //一个整数区间

boost::copy(                                         //copy 算法
```

```
    boost::sort(v) |                                   //先排序，区间改变
      adaptors::uniqued,                               //去重，原区间不变
    ostream_iterator<int>(cout, ","));                 //输出到标准流

assert(boost::count(v, 6) == 2);                       //原区间排序但未去重

auto even = [](int x){ return x%2==0;};                //C++11 的 lambda 表达式
assert( boost::distance(                               //计算区间的长度
        v | adaptors::filtered(even)) == 5);           //仅计算偶数

boost::copy(v | adaptors::copied(0,5),                 //输出前 5 个元素
            ostream_iterator<int>(cout, ","));
```

灵活组合区间算法和区间适配器，我们还可以获得标准算法中_if、_copy、_copy_if、_backward、_n 版本的效果，而且写法更清晰，更能组合出_copy_if_backward 之类的新算法：

```
boost::copy(v | adaptors::filtered(even),              //相当于 copy_if
    ostream_iterator<int>(cout, ","));

boost::copy(v | adaptors::reversed,                    //相当于 copy_backward
    ostream_iterator<int>(cout, ","));

boost::copy(v |                                        //相当于 copy_if_backward
    adaptors::filtered(even) | adaptors::reversed,
    ostream_iterator<int>(cout, ","));
```

注意我们不能实现_if_n 之类的算法，即不能有 v|filtered|copied 的组合方式，这是因为 filtered 返回的是双向区间——即使原始区间是随机访问区间，而 copied 和 sliced 要求是随机访问区间，如果强行搭配则会导致编译错误。

6.7.3 实现原理

这里仅以 indirected 适配器来说明区间适配器的实现原理，其他适配器可以触类旁通。

indirected 适配器在 boost::range_detail 子名字空间里定义了一个区间类型 indirected_range，它使用了 5.6.4 节的间接迭代器 indirect_iterator：

```
template< class R >
struct indirected_range :                              //间接区间
    public boost::iterator_range<                      //iterator_range 工具
```

```
            boost::indirect_iterator<                   //间接迭代器
                range_iterator<R>::type                 //迭代器特征
            > >
{
private:
    typedef boost::iterator_range<...>  base;           //基类

public:
    explicit indirected_range( R& r ) : base( r )       //构造函数
    { }
};
```

indirected 适配器因为不需要额外的参数，所以它只是一个简单的空类 indirect_forwarder 的实例，仅仅用作标记：

```
struct indirect_forwarder {};                                    //空类
const indirect_forwarder indirected = indirect_forwarder(); //常量实例
```

有了 indirected_range 和 indirect_forwarder，operator|()的实现就很简单了，它是一个工厂函数，直接创建出 indirected_range：

```
template< class InputRng >
inline indirected_range<InputRng>
operator|( InputRng& r, indirect_forwarder )        //indirect_forwarder仅是个标记
{
    return indirected_range<InputRng>( r );                    //返回间接区间
}
```

这样，每一个 operator|()操作都基于前区间产生出一个新的区间，这个区间或者是迭代器范围变化，或者是迭代器功能变化，通过迭代器的功能串联完成了所有的操作。

6.8 其他议题

本小节讨论一些关于 range 库的其他议题。

6.8.1 自定义区间类型

range 库支持标准容器、Boost 容器、迭代器的 pair 等类型，并且提供了区间工具类 iterator_range 和 sub_range，这在大多数情况下已经能够满足我们的需要，但有的时候我

们想让自己编写的类也能够搭配区间算法使用，这时就要求自定义类型满足区间的概念。

区间的概念已经在 6.1 节做了介绍，可以用以下三种方式来实现一个区间。

- 定义为 iterator_range 的子类。
- 提供成员函数 begin()/end()和内部的类型定义 iterator/const_iterator，这是一个比容器概念低得多的要求，很容易实现。
- 如果不能修改类型，那么可以为类型重载 range_begin()和 range_end()，返回区间的端点，同时还要重载（特化）boost 名字空间里的元函数 range_const_iterator 和 range_mutable_iterator，这种方式比较麻烦。

6.8.2 连接区间

range 库提供一个 join()函数，它可以把两个区间连接为一个区间：

```
template<class SinglePassRange1, class SinglePassRange2>
joined_range<SinglePassRange1, SinglePassRange2>
join(SinglePassRange1& r1, SinglePassRange2& r2)
```

注意 join()函数并不要求被连接的两个区间是相邻的，也就是说它可以把两个彼此独立的区间连接成一个逻辑上连续的区间，例如：

```
vector<int> v;                                  //整数区间，用 irange 生成 10 个整数
boost::copy(boost::irange(0,10), std::back_inserter(v));

auto r1 = make_iterator_range(v.begin(), v.begin() + 3);     //子区间 0～2
auto r2 = make_iterator_range(v.begin() + 5, v.end() );      //子区间 5～9

auto r3 = boost::join(r1, r2);                               //连接区间
//输出 0,1,2,5,6,7,8,9,
boost::copy(r3, ostream_iterator<int>(cout, ","));
```

6.9 总结

本章我们探讨了 range 库，它提出了区间的概念，可以看作是一个轻量级、不容纳元素的“容器视图”。

range 的出发点主要是简化标准算法的使用，通过将两个端点定义为一个“区间”，很多调

用算法的代码都得到了简化。区间还带来了一个有用的“副作用”：算法可以返回区间，从而被其他算法处理，多个算法可以组合起来完成复杂的功能。

为了方便地操作标准容器，range 库还定义了类 iterator_range，它可以精确地描述区间概念，并派生出了 sub_range、counting_range、any_range 等更易使用的小工具。

区间适配器是 range 库的一个非常有用的创新，它重载了 operator|()，通过变动区间使用的迭代器来改变区间的功能，使区间不仅能表达容器的范围，而且可以像流一样处理里面的元素，解法比算法嵌套更加优雅。

第7章 函数对象

本章讨论现代 C++编程中的另一个重要角色：函数对象。

函数对象（function object）又称仿函数（functor），是一个定义了 operator()的类，可以像普通函数一样被调用，因为具有类的所有功能，所以又被称为“智能函数”，比普通函数更加强大。

Boost 库中包含很多功能强大的函数对象，在本章中我们仅研究其中的三个：首先是 hash，它用于计算对象的散列值，被用来实现各种散列容器；然后是 mem_fn，可以绑定成员函数指针；最后是泛化的工厂函数对象 factory，它是一个“智能 new”，完全可以代替 new 关键字。

这些函数对象都非常实用，读者很快就会发现它们的价值。

7.1 hash

hash 是 C++11 标准（C++11.20.8.12）中规定的散列函数的一个具体实现，可以计算任意 C++对象的散列值，主要被用于实现各种无序散列容器，如 unordered_set 和 ptr_unordered_set。

hash 位于名字空间 boost，需要包含头文件<boost/functional/hash.hpp>，即：

```
#include <boost/functional/hash.hpp>
using namespace boost;
```

7.1.1 类摘要

hash 是一个非常简单的函数对象，类摘要如下[①]：

```
template <class T>
struct hash: std::unary_function<T, std::size_t>      //标准单参函数对象
{
  std::size_t operator()(T const& val) const          //计算 val 的散列值
};
```

hash 是一个单参函数对象，成员函数 operator()接受一个类型 T 的常引用值作为输入，计算得到一个类型为 size_t 的散列值返回。

hash 完全符合 C++11/14 标准，对 C++数据类型的支持非常全面（并且有扩展），包括以下几种类型。

- char/wchar_t（但不支持 C++11 的 char16_t/char32_t）。
- short/int/long 等各种整数类型。
- bool 类型。
- float/double 等浮点数类型。
- long long 和 long double（如果编译器支持）。
- 指针和数组。
- std::string/std::wstring。
- 扩展支持 pair、complex 结构和 array、vector、list 等标准容器。

hash 库不支持上述以外的其他类型，包括标准容器适配器（stack、queue、priority_queue）和所有 Boost 容器（bimap、circular_buffer 等）。

7.1.2 用法

hash 是一个很简单的函数对象，因此很容易使用，只需要创建一个实例（左值或右值均可），

① 实际上 hash 函数对象使用模板特化有多种不同的形式，这里的类摘要只是其中较常用的一个，不同形式的区别主要在于 operator()的参数类型。

然后像使用函数一样调用它的 operator()就可以了，例如：

```
cout << hash<int>()(0x2000) << endl;                    //计算整数的散列值
cout << hash<double>()(1.732) << endl;                  //计算浮点数的散列值
cout << hash<const char*>()("string") << endl;          //计算字符数组的散列值

complex<double> c(1.0, 2.0);
cout << hash<decltype(c)>()(c) << endl;                 //计算复数的散列值

cout << hash<string>()("string") << endl;               //计算标准字符串的散列值

vector<int> v(12);
cout << hash<decltype(v)>()(v) << endl;                 //计算标准向量容器的散列值

map<int, string> m;
cout << hash<decltype(m)>()(m);                         //计算标准映射容器的散列值

array<int, 5> ar;                                       //C++11 或 Boost 的数组容器
cout << hash<decltype(ar)>()(ar) << endl;               //现在 hash 支持
```

这段代码计算了各种基本类型和标准库容器的散列值，会输出一系列的十六进制数值。如果企图用 hash 计算不支持的类型，就会导致编译错误，例如：

```
stack<int> s;                                           //栈结构
cout << hash<decltype(s)>()(s) << std::endl;            //编译错误
```

除了直接计算对象的散列值，hash 更常见的用途是被用作无序容器的模板参数，作为容器计算散列值的一个策略对象：

```
boost::unordered_set<int, boost::hash<int> >            us;
boost::unordered_map<int, string, boost::hash<int> >    um;
```

上面的代码使用了 Boost 中实现的散列容器，并统一使用 hash 作为容器计算散列值的模板参数。

7.1.3 实现原理

hash 函数对象实际上只是一个简单的包装类，真正的散列值计算实现是在 boost 名字空间里的 hash_value()函数，而 hash_value()则又针对各种数据类型定义了不同的重载形式，最后 hash 再对不同的类型使用模板特化来调用 hash_value()函数。

hash_value()函数的声明通常采用下面的形式：

```
template <class T>
std::size_t hash_value(T const&);
```

例如，对于基本数据类型 char、int 等，hash 的实现如下：

```
template <typename T>
typename basic_numbers<T>::type hash_value(T v)  //元函数计算简单整数类型
{   return static_cast<std::size_t>(v); }         //计算散列值

template <> struct hash<int>                       //模板特化 hash 类
{
    std::size_t operator()(int v) const
    {   return boost::hash_value(v); }             //调用散列函数，自动模板推导
};
```

因此，只要我们在自己的名字空间或者其他可被 ADL（参数依赖查找规则）查找到的位置里定义了 hash_value()的重载形式，就能够对自定义类型调用 hash 函数对象计算散列值。

7.1.4 扩展 hash

使用 hash 库提供的对基本数据类型计算散列值的功能和一些辅助函数，我们可以实现对自定义类型计算散列值。

简单地计算散列值

下面的代码定义了一个 person 类，在成员函数 hash_value()中计算散列值（也可以是其他的名字），然后再在外部（或者定义为内部友元函数）实现重载的自由函数 hash_value()：

```
class person final
{
private:
  int             id;
  string          name;
  unsigned int    age;
public:
  person(int a, const char* b, unsigned int c):        //构造函数
    id(a),name(b), age(c){}
```

```
    size_t hash_value() const                    //自定义的散列计算函数
    {
        return hash<int>()(id);                  //只调用 hash 函数对象根据 id 计算散列值
    }
};

size_t hash_value(person const & p)              //同名字空间重载 hash_value
{ return p.hash_value();}
```

这样就可以把 hash 函数对象应用于我们自定义的 person 对象了：

```
person p(1, "adam", 20);
cout << hash<person>()(p) << endl;               //可正确执行
```

如果要把 person 对象放入无序容器，除了实现 hash 外，我们还需要定义 operator==：

```
class person final
{
    ...                                          //同前
    friend bool operator==(person const & l, person const & r)
    {   return l.id == r.id;}
};

int main()
{
    unordered_set<person> us;                    //无序集合容器

    us.insert(person(1, "adam", 20));
    us.insert(person(2, "eva", 20));
    assert(us.size() == 2);
}
```

组合散列值

如果要对多个目标计算散列值，那么可以使用 hash 库提供的辅助函数 hash_combine() 和 hash_range() 来组合散列值，它们的声明如下：

```
template<typename T>
void hash_combine(size_t & seed, T const& v);

template<typename It>
```

```
std::size_t hash_range(It first, It last);

template<typename It>
void hash_range(std::size_t& seed, It first, It last);
```

hash_combine()使用一个变量 seed 作为初始输入参数，可以对多个变量连续调用，最终计算得到的散列值再从 seed 输出。散列值的计算与 hash_combine()的运算顺序有关，即使是对同样的元素，如果计算顺序不同，那么最后得到的散列值也会不同。

使用 hash_combine()可以为 person 类定制新的散列计算方法：

```
size_t hash_value() const
{
  size_t seed = 1984;                     //可以是任意的整数
  hash_combine(seed, id);                 //组合 3 个变量
  hash_combine(seed, name);
  hash_combine(seed, age);
  return seed;
}
```

hash_range()是另外一种组合散列值的方式，它对一个迭代器区间内的所有元素调用 hash_combine()计算散列值，如果不给初始 seed 赋值（两参数的形式），那么 seed 默认为 0。hash_range()的用法示例如下：

```
vector<int> v {1,2,5,8,15};                           //C++11 的新式初始化
auto hv = hash_range(v.begin(), v.end());

unordered_set<int> us {1,2,5,8,15};                   //C++11 的新式初始化
hv = hash_range(us.begin(), us.end());
```

灵活使用 hash_combine()和 hash_range()，我们就可以对任意 C++对象使用任意策略计算散列值（实际上 hash 库对标准容器的处理就利用了这两个函数）。

下面的代码定义了一个类 demo_class，它有一个 int 成员 x 和一个 vector<string> 成员 v，散列值的计算方式是把 0 作为 seed，先计算 x，然后再逆序计算 v 里的所有元素：

```
class demo_class
{
private:
  vector<string>  v;
  int             x;
```

```
public:
  size_t hash_value()
  {
      size_t seed = 0;                              //seed 取值为 0
      hash_combine(seed, x);                        //先计算整数的散列值
      hash_range(seed, v.rbegin(), v.rend());       //逆序遍历 vector

      return seed;
  }
};
```

7.2 mem_fn

mem_fn 是对 C++98 标准库中的成员函数适配器的增强，同 bind 一样可以调用对象或对象指针的任意成员函数或成员变量，而且支持多达 8 个参数。它已经被收入 C++11 标准（头文件<functional>，C++11.20.8.10）

mem_fn 位于名字空间 boost，需要包含头文件<boost/mem_fn.hpp>，即：

```
#include <boost/mem_fn.hpp>
using namespace boost;
```

7.2.1 工作原理

同 bind 一样，mem_fn 也不是一个单一的模板函数，而是为了支持各种情况而有许多的重载形式，它基本的声明形式如下：

```
template<class R, class T> mf<R, T> mem_fn(R T::*f);     //mem_fn 函数

template<class R, class T> class mf                       //辅助函数对象
{
public:
    R & operator()(T * p,...) const;                      //调用对象的指针
    R & operator()(T & t,...) const;                      //调用对象的引用
}
```

mem_fn 函数接受一个类型 T 返回类型为 R 的成员函数指针 f，然后返回一个持有成员函数指针的函数对象 mf。mf 的 operator()可以传入类型 T 的指针/引用以及若干参数，再转而调

用其成员函数。

mem_fn 支持最多 8 个参数，这与 bind 的默认参数数量限制是一致的（bind 默认最多支持 9 个参数，但绑定成员函数时需要“牺牲”一个参数传递对象）。与 bind 不同的是 mem_fn 没有_1、_2 这样的参数占位符，因此不能实现对参数的绑定，必须在 operator()中传递所有的参数。

7.2.2 用法

如果读者熟悉 bind，那么 mem_fn 会很容易学习，它就像是一个简化版的 bind，只能绑定类的成员函数或成员变量。

mem_fn 不仅支持普通对象，也支持对象指针和智能指针。对于智能指针，mem_fn 会使用函数 boost::get_pointer()（参见 4.7.1 节）获取真正的指针，不会导致 unique_ptr 的所有权转移或者 shared_ptr 的引用计数增加，因此非常安全。

假设我们有下面的一个类：

```
class demo_class
{
public:
  int x;                                    //一个公开的成员变量
  demo_class(int a = 0):x(a){}              //构造函数

  void print()                              //一个无参的成员函数，非 const
  {   cout << x << endl; }

  void hello(const char* str)               //单参成员函数，非 const
  {   cout << str << endl; }
};
```

那么 mem_fn 可以这样使用：

```
demo_class d;
mem_fn(&demo_class::print)(d);              //绑定普通对象，调用无参成员函数

demo_class *p = &d;
mem_fn(&demo_class::hello)(p, "hello");     //绑定对象指针，调用单参成员函数

unique_ptr<demo_class> up(new demo_class(100));
mem_fn(&demo_class::print)(up);             //绑定 unique_ptr
```

```
assert(up.get() != 0);                          //指针的所有权没有转移

shared_ptr<demo_class> sp(new demo_class);      //绑定共享指针
mem_fn(&demo_class::hello)(sp, "world");
```

mem_fn 更常见的用法是作为标准算法的一个参数，对容器内的所有元素调用无参成员函数，无论容器中存储的是元素本身、指针或是智能指针都可以正常工作：

```
vector<demo_class> v(10);                           //标准向量容器
std::for_each(v.begin(), v.end(),                   //使用 for_each 算法
        mem_fn(&demo_class::print));                //mem_fn 调用成员函数
```

这段代码使用了标准库的 for_each 算法，对 vector 中的每个元素均调用了 print() 函数，无须考虑成员函数的 const 属性和元素是否为指针，mem_fn 自动处理了所有的问题。

mem_fn 不仅可以调用成员函数，也能够选择类的（公开）成员变量，功能类似于非标准函数对象适配器 select1st 和 select2nd，用法与调用成员函数相同，例如：

```
demo_class   d(1);
cout << mem_fn(&demo_class::x)(d) << endl;          //访问成员变量
```

7.2.3 其他议题

本小节讨论关于 mem_fn 的一些其他议题。

配合其他 Boost 组件

mem_fn 可以配合 ref 库和 function 库使用以发挥更大的作用：ref 库可以让 mem_fn 持有一个参数的引用而不是拷贝，function 则可以存储 mem_fn 生成的函数对象，供延后使用。

与 bind 的比较

在很多情况下，bind 都可以代替 mem_fn，只需要增加一个“_1”占位符用于传递对象即可。例如，之前的部分 mem_fn 的 bind 等价表达式是：

```
demo_class d;
bind(&demo_class::print, _1)(d);
bind(&demo_class::hello, _1, "hello")(&d);
```

如果类的成员函数有多个参数，那么 bind 的用法通常会更加灵活，因为它可以使用占位符

绑定参数：

```
vector<demo_class> v(10);
std::for_each(v.begin(), v.end(),
     bind(&demo_class::hello, _1, "world"));          //调用有参成员函数
```

但 mem_fn 也有它自己的优势：写法简单，容易理解和掌握。

7.3 factory

factory 是一个很小却很强大的库，它实现了工厂设计模式，用函数对象封装了对象的创建过程，消除了关键字 new 的随意使用，有助于改善程序的设计结构。

factory 库提供了两个函数对象，factory 和 value_factory，我们将重点讨论 factory。

factory 位于名字空间 boost，需要包含头文件<boost/functional/factory.hpp>，即：

```
#include <boost/functional/factory.hpp>
using namespace boost;
```

7.3.1 类摘要

factory 的类摘要如下：

```
template< typename Pointer,
          class Allocator, factory_alloc_propagation >
class factory
{
 public:
   typedef typename boost::remove_cv<Pointer>::type     result_type;
   typedef typename boost::pointee<result_type>::type   value_type;
   inline result_type                                   operator()() const;
   ...                                              //operator()的其他定义
};
```

factory 是一个模板类，有三个模板参数，但通常我们只需要使用第一个模板参数 Pointer，后两个可以用缺省值。

模板参数 Pointer 是被创建出的“指针”类型。之所以模板参数是 Pointer 而不是 T*，是因为 factory 不仅支持创建原始指针，也能够创建智能指针 unique_ptr 和 shared_ptr。

factory 内部使用元函数定义了两个 typedef，可以被用于泛型编程：result_type 是函数对象调用后返回的类型，即创建的指针类型；value_type 则是指针指向的值类型，它使用了 4.7.3 节的 pointee，由 result_type 进行元计算得到。

operator()是 factory 的核心功能所在，它支持最多 10 个参数，这些参数被传递给 value_type 的构造函数以创建对象。

7.3.2 用法

factory 封装了 new 操作符，基本相当于 new T()，可以直接创建 T*指针。如果说 checked_delete（4.2 节）是“智能 delete”，那么 factory 就是一个“智能 new”。

使用 factory 要求被创建的类型 T 至少要有一个 public 构造函数，否则 factory 会因为无法访问 protected 或者 private 的构造函数而不能完成创建工作。

无参创建指针

factory 可以完全代替 new，只需要在模板参数中指明要创建的指针类型，它就会如 new 一样完成工作。采用无参的形式时 factory 将调用 value_type 的缺省构造函数完成对象的初始化，例如：

```
auto pi = factory<int*>()();                          //注意，两对括号
auto ps = factory<string*>()();                       //使用 auto 自动推导类型
auto pp = factory<pair<int, double>*>()();
```

请读者注意 factory 的调用方式。因为它不是函数而是函数对象，因此我们必须先用一对括号调用它的构造函数，创建出 factory 的一个临时对象，然后再用第二对括号调用它的 operator()来创建所需的对象。

factory 创建出的指针可以用 delete 操作符删除，当然最好使用与它对应的“智能 delete”——checked_delete。

创建智能指针

factory 也可以创建智能指针对象，这样我们就无须关心指针的删除问题，智能指针会自

动处理。创建智能指针时必须在 factory 的模板参数中完全写出智能指针的类型，不必再加*号，因为它们本身就已经是“指针”。

下面的代码创建了 unique_ptr 和 shared_ptr 智能指针：

```
auto up = factory<unique_ptr<int> >()();
auto sp = factory<shared_ptr<string> >()();
```

factory 不能创建 boost::scoped_ptr，这是因为 scoped_ptr 不支持拷贝转移语义，下面企图用 factory 创建 scoped_ptr 的代码将导致编译错误：

```
scoped_ptr<int> p = factory<scoped_ptr<int> >()();              //编译错误
```

带参数创建指针

factory 也支持使用多个（最多 10 个）参数来创建指针对象，这些参数将传递给类的构造函数完成初始化。但带参数的创建功能存在一个小缺陷：要求参数必须是左值类型，如果传递右值则会无法编译。请见下面的示范：

```
int a = 10, b = 20;                                   //声明两个变量用来创建指针
auto pi = factory<int*>()(a);
auto ps = factory<string*>()("char* lvalue");         //字符串也是左值
auto pp = factory<pair<int, int>*>()(a, b);

auto pi2 = factory<int*>()(10);                       //使用右值，无法通过编译
auto pp2 = factory<pair<int, int>*>()(1, 2);          //使用右值，无法通过编译
```

这个技术缺陷大大限制了 factory 的使用——为了创建一个对象必须声明若干个临时变量实在是太麻烦了，而且很不优雅。

不过好在这个缺陷并非不可弥补，使用 bind 包装 factory 函数对象可以解决这个问题。因为 bind 对参数类型没有限制，它内部持有的参数拷贝，可被用作左值。

factory 的 bind 用法要显得麻烦一些，语法上也比较复杂：

```
auto p = bind(factory<int*>(),10)();                  //可以使用右值
```

这行代码首先创建了一个 factory<int*>临时对象，使用 bind 为它绑定了一个值为 10 的参数。bind 函数生成了一个新的函数对象，因此需要再用一对圆括号来调用它的 operator()。然后 bind 把参数 10 转发给 factory 对象最终完成 int 指针的创建工作。

7.3.3 value_factory

value_factory 是 factory 库提供的另外一个函数对象，它同样可以创建对象，用法与 factory 也很类似，但不同的是它创建出的不是指针，而是真正的对象实例，它位于头文件 <boost/functional/value_factory.hpp>。

value_factory 的类摘要如下：

```
template< typename T >
class value_factory
{
public:
  typedef T                  result_type;
  inline result_type         operator()() const;
  ...                                                //其他 operator()的定义
};
```

value_factory 的声明与实现都要比 factory 简单，它仅有一个 typedef (result_type)，同样最多支持传入 10 个参数，也同样要求参数必须是左值，operator() 将返回创建对象的拷贝，因此类型 T 必须支持拷贝构造。

下面的代码示范了 value_factory 的用法：

```
auto i    = value_factory<int>()();
auto str  = value_factory<string>()("hello");
auto p    = value_factory<pair<int, string>>()(i, str);
```

左值的问题同样可以使用 bind 来解决，例如：

```
auto i    = bind(value_factory<int>(), 10)();
```

7.4 总结

本章内容较少，只介绍了三个函数对象。因为函数对象只能用于运行时，所以没有涉及太多的模板元编程知识，相信学习起来会轻松一些。

hash 是一个用来计算散列值的函数对象，它符合 C++11/14 规范并有所增强，可以被用于各种散列容器。hash 的实现原理很简单，使用多个重载或模板特化的 hash_value()函数来计

算不同类型的散列值，并且提供了辅助函数hash_combine()和hash_range()来组合散列值，所以我们可以定制自有类型的散列值计算策略，然后把它容纳到散列容器中。

mem_fn可以适配任意的成员函数，用法更简单。因为它非常有用，所以被收入了C++11/14标准。不过由于有了更好的函数绑定器 bind 和 lambda 表达式，所以更多的时候使用bind/lambda会更灵活方便，mem_fn只能适用于比较简单的场合。

在本章最后，我们研究了factory，可以代替操作符new创建对象，是一个“智能new”。也许有的读者会认为 factory 的用法很累赘麻烦，但这实际上是没有领会设计模式的精髓——封装，factory可以更好地隔离硬式创建对象的代码，具有更好的类型安全性。而且，某种意义上，new 是和 delete 一样的“麻烦存在”，既然强调使用 checked_delete 来消灭 delete 的使用，那也应该相应地使用 factory 来消灭 new 的使用。

第8章

指针容器

容器是C++标准库中最引人注意的部分，泛型的容器可以容纳任意的元素，免去了手工构造数据结构的麻烦，vector、list、set 等标准容器已经成为了广大程序员不可或缺的工具。

Boost 延续了标准库的思想，进一步扩充了容器的范围，为 C++社区提供了更多的容器。推荐书目[3]中已经讨论了 Boost 的大部分新式容器，如 array、dynamic_bitset、any、multi_array 等。从本章开始我们将看到另外三类容器：指针容器、侵入式容器和多索引容器。这三个容器在基本概念上与标准容器非常相似，但它们都在不同的方向上作出了扩展，丰富了我们的容器工具箱。

在本章中，我们先研究指针容器库 ptr_container，它可以安全地容纳指针作为容器元素，并保证没有内存泄漏的风险。

8.1 概述

很多时候我们需要在容器中存储指针而不是元素本身（比如元素不满足标准容器的要求，存储抽象类而不是具体类，避免值语义内存拷贝的代价），但直接存储原始指针手法太初级，很不安全也难于管理，我们可以有如下的替代选择。

(1) 使用 shared_ptr，它可以在容器中很好地管理指针，是最通用的解决方案，但由于容器的拷贝语义，使用存储的 shared_ptr 时需要进行引用计数的增减，操作效率会略微降低，而且使用迭代器操作容器内的 shared_ptr 也显得不太方便（使用 5.6.4 节的间接迭代器可获得一定程度上的改善）。

(2) 使用内存池 boost.pool。这种方法类似于自行创建一个小型的垃圾回收机制，可以任

意分配内存而不必担心回收，创建的对象可以直接放到标准容器中，不必在容器销毁时使用 delete 删除指针。但这种方法也有缺点，因为目前在 C++中暂时还没有一个“泛用”的内存池，boost.pool 必须为每一类对象创建一个单独的内存池，使用起来很麻烦。①

（3）第三种容纳指针的方法就是使用 Boost 提供的指针容器库 ptr_container（下文中如不做特别声明，“指针容器”即是指 boost.ptr_container），它实现了数个标准容器风格的可容纳指针的容器，而且是高效和异常安全的，在某些必须保存指针的情况下特别有用。

ptr_container 库位于名字空间 boost，由数个不同的头文件组成，它们都位于目录 <boost/ptr_container/>下，使用具体的容器时包含所需头文件即可。

8.1.1 入门示例

ptr_container 库在开发时特意模仿了标准容器的风格，很多地方都很像标准容器，因而比较容易学习，但毕竟它容纳的不是普通元素而是指针，所以我们需要留意其特别之处。

在本小节中，我们先通过两个简单的小例子来快速预览一下指针容器的功能、用法和特性。

示例 1

第一个例子使用了类似于 std::vector 的向量指针容器 ptr_vector，它演示了指针容器与标准容器的相同之处：

```
#include <boost/ptr_container/ptr_vector.hpp>     //向量指针容器头文件
using namespace boost;

int main()
{
  typedef ptr_vector<string> ptr_vec;                //一个容纳 string 的指针容器
  ptr_vec vec;                                       //声明一个指针容器实例

  vec.push_back(new string("123"));                  //添加两个元素
  vec.push_back(new string("abc"));                  //注意我们添加的是 new 创建的指针

  assert(!vec.empty());                   //可以使用类似标准容器的 empty()函数
  assert(vec.size() == 2);                //同样可以使用 size()获得容器的大小
```

① 使用内存池的另外一个副作用是程序员失去了对内存的控制，不能利用 RAII 的好处，内存释放的时机只能由内存池来控制，或者使用内存池的方法来强制释放内存调用析构函数，但这种操作实质上与使用 delete 没有本质差别。

```
    assert(vec[0] == "123");          //使用 operator[]访问元素，注意返回的是引用
    vec.back() = "def";               //使用 back()成员函数访问末尾元素，同样返回的是引用
    assert(vec[1] == "def");

    auto iter = vec.begin();          //获取容器的迭代器
    assert(iter->length() == 3);      //迭代器可以直接操作指针指向的内容

    vec.clear();                      //可以使用 clear()清空容器，内存被安全释放
    assert(vec.empty());
}
```

这段代码看起来与标准容器 std::vector 的用法非常接近，但有一点本质的不同：我们向容器添加的元素是用操作符 new 动态创建的对象，容器容纳的是指针而不是元素本身（或者拷贝）。虽然指针容器中存储的是指针，但我们在使用时却感觉不到指针的存在，好像它里面容纳的就是普通的元素——这是因为指针容器在内部替我们多做了一次解引用操作（operator*），所以用起来非常方便，减少了操作原始指针可能出现的错误。

下面对比一下使用标准容器存储指针的用法：

```
vector<string*> vec;                  //使用 vector 存储指针,注意模板参数有*号
vec.push_back(new string("123"));     //添加指针
vec.push_back(new string("123"));

assert(*vec[0] == "123");             //获得元素后需要使用 operator*解引用
*vec.back() = "def";
assert(*vec[1] == "def");

auto iter = vec.begin();
assert((*iter)->length() == 3);       //迭代器同样需要使用 operator*
```

很明显，使用标准容器存储指针时访问元素要麻烦许多，同时，我们还必须自行管理对象的生存周期，使用完毕后要记得删除指针，否则就会造成内存泄漏。

示例 2

接下来我们研究第二个例子，它演示了指针容器的指针所有权转移，这是与标准容器显著不同的地方：

```
typedef ptr_vector<string> ptr_vec;          //一个容纳 string 的指针容器
ptr_vec vec;                                 //声明一个指针容器实例
```

```
auto_ptr<string> ap(new string("123"));       //一个 auto_ptr
vec.push_back(ap);                            //指针容器可以“容纳”自动指针
assert(ap.get() == 0);                        //auto_ptr 失去指针的管理权

vec.push_back(auto_ptr<string>(new string("abc")));
vec.push_back(new string("xyz"));
assert(vec.size() == 3);

ptr_vec::auto_type p =                        //auto_type 是一个智能指针类型
  vec.release(vec.begin());                   //指针容器释放一个迭代器位置指针的管理权
assert(vec.size() == 2);                      //指针容器不再管理已经释放的指针
assert(p && *p == "123");                     //p 可以像指针一样使用

string* sp = p.release();                     //auto_type 也可以释放指针的管理权
assert(!p);                                   //p 不再管理原始指针
delete sp;                                    //原始指针可以手动删除

ptr_vec vec2;                                 //另一个指针容器实例

vec2.transfer(vec2.end(),vec);                //可以在两个容器间转移指针的所有权
assert(vec.empty());                          //转移后原指针容器不再管理指针
assert(vec2.size() == 2);                     //新指针容器唯一管理指针
```

这段代码演示了指针容器的最基本特性：指针的所有权是唯一的，它采取的是专有(exclusive）语义而非共享语义（shared)。这意味着指针容器是指针的唯一持有者，其他人可以使用但不能管理指针。这个特性保证了指针的安全性，但同时也衍生出了许多变化，在一定程度上增加了指针容器的复杂程度。[①]

指针容器可以“容纳”C++98 标准中的智能指针 auto_ptr（在 C++11/14 中被声明为废弃），它也与 auto_ptr 的语义保持一致：容纳的同时也接管了 auto_ptr 里原始指针的管理权，令 auto_ptr 不再对原始指针的生命周期负责。

虽然指针容器管理指针，但不必永远对指针负责，如果需要，那么也可以释放指针的管理权。

① 在指针管理方面 ptr_container 有些类似于标准库的智能指针 unique_ptr，像是它的可容纳多个元素的泛化，我们可以通过类比来理解：
unique_ptr 可以看作是只能容纳一个指针的指针容器，它持有指针的所有权，而且指针的所有权是专有的而不是共享的， unique_ptr 的拷贝、赋值都是转移语义，操作的是指针而不是指针的指向物，一个指针同时只能被一个 unique_ptr 唯一管理。

成员函数 release()可以释放一个指针，它从容器中删除指针（但不删除指针指向的内容），返回一个类型为 auto_type 的智能指针，这个智能指针可以像原始指针一样使用，我们也可以从中获得原始指针。指针容器的成员函数 transfer()可以从另一个指针容器中转移指针的所有权（参见 5.3 节），在效果上很类似元素的拷贝，但转移后原容器就失去了指针的所有权，同时目标容器获得了指针的所有权。

请读者注意：在任何时刻，这些转移操作中指针的所有权都被唯一的管理者持有，不会出现共同管理（shared）的局面。

8.1.2 指针容器的优缺点

与容纳元素的拷贝的标准容器或容纳智能指针的容器相比较，ptr_container 库有许多优点值得我们去尝试。

- 可以直接在容器中存储动态创建的对象，并且无须关心它的生存周期问题，绝对不会发生内存泄漏，用起来更安全。
- 特别支持“容纳”auto_ptr 对象，可以完美地配合 auto_ptr 工作（实际上是使用了 auto_ptr 的转移语义，在进入容器后 auto_ptr 就失去了对指针的管理权）。[①]
- 因为直接存储原始指针，没有了智能指针的引用计数，执行效率更高，使用的内存更少。
- 同样因为只存储原始指针，指针容器对元素的要求很低，可以存储不可拷贝构造、缺省构造和赋值的类型——这些类型是标准容器无法容纳的。
- 不仅可以存储具体类，也可以存储抽象类，也就是说可以是多态的容器。
- 提供异常安全保证。
- 拥有与标准容器近似的风格，许多名字和用法都非常相似，学习成本低。

当然，指针容器并不是完美无瑕的，它并不能替代标准容器，也存在一些缺点。

- 存储的指针是专有的（exclusive），缺少智能指针容器的灵活性。
- 指针的转移、克隆（clone）等深层次概念较难理解。
- 较标准容器的接口有少量但关键的变动，使用时需要小心谨慎。

---

① 非常遗憾的是 ptr_container 库已经较长时间没有更新，未能支持 C++11/14 标准里的 unique_ptr 智能指针。

- 部分标准算法不能用于指针容器（但提供了等价的成员函数，参见 8.14 节）。

了解指针容器的优点和缺点有利于我们在实际工作时针对具体问题选择最佳的解决方案。在通常情况下，如果我们不得不动态创建对象，并且要使用容器来管理这些对象，那么就可以使用指针容器——但需要共享对象的所有权时除外。

8.1.3 可克隆概念

指针容器仅容纳指针，因此对元素的要求远比标准容器的要低，元素类型 T 不必是可缺省构造、可拷贝和可赋值的，几乎任何类型都可以放入指针容器。但如果容纳的对象需要创建副本，那么 T 应该是可克隆的（cloneable）——可以把克隆操作看作是指针容器范畴中的拷贝操作。

“克隆”（clone）是原型设计模式（prototype）的具体应用，ptr_container 库使用克隆分配器（clone allocator）代替标准容器中的内存分配器概念，用来创建等价的对象。一个对象如果是可克隆的，那么它应该支持下面的两个函数。

- new_clone() ：创建一个与原型等价（equivalent）的新对象，相当于 new。
- delete_clone() ：删除之前创建的对象，相当于 delete。

注意，可克隆性对于指针容器来说不是必须的（非强制），只有我们确实需要从指针容器中克隆对象时克隆函数才会被使用，大多数指针容器的操作没有对可克隆性的要求。而且为了便于库的使用，ptr_container 库在头文件 <boost/ptr_container/clone_allocator.hpp>中提供了克隆所需函数的泛型实现，代码摘要如下：

```
template< typename T >
inline T*     new_clone( const T& r )
{
    T* res = new T( r );                         //使用 new 拷贝构造新对象
    return res;
}

template< typename T >
inline void   delete_clone( const T* r )
{   checked_delete( r );}                        //删除对象
```

因此，绝大多数类（通常都有缺省的拷贝构造函数）自动支持可克隆概念，可以使用指针容器的所有功能。

关于可克隆概念的进一步讨论参见 8.15.5 节。

8.1.4 克隆分配器

克隆分配器（clone allocator）相当于标准容器中的内存分配器（allocator）的概念和地位，是对内存模型的一种抽象表述，指针容器使用克隆分配器来克隆（不是创建）和删除指针对象。

ptr_container 库提供了两个克隆分配器：heap_clone_allocator 和 view_clone_allocator。

heap_clone_allocator

heap_clone_allocator 是 ptr_container 库所有指针容器缺省使用的克隆分配器，它使用 new_clone()/delete_clone() 来分配/释放内存，因此要求元素必须满足可克隆的概念。

heap_clone_allocator 的实现代码如下：

```
struct heap_clone_allocator
{
    template< class U >
    static U* allocate_clone( const U& r )
    {   return new_clone( r ); }                //调用 new_clone()

    template< class U >
    static void deallocate_clone( const U* r )
    {   delete_clone( r ); }                    //调用 delete_clone()
};
```

view_clone_allocator

view_clone_allocator 是一个“虚拟”的克隆分配器，它并不真正管理内存，而是提供一个只读的“指针视图”，它的实现代码如下：

```
struct view_clone_allocator
{
    template< class U >
    static U* allocate_clone( const U& r )
    {   return const_cast<U*>(&r); }            //不做内存分配操作

    template< class U >
    static void deallocate_clone( const U* /*r*/ )
    {   /*do nothing*/   }                      //空函数，没有删除动作
```

```
};
```

view_clone_allocator 并不真正地克隆或删除对象，因此如果指针容器使用它作为分配器，那么就不会持有指针的管理权，变成了一个安全的观察者。

view_clone_allocator 可以把指针容器转化为另一个容器的视图来使用，方便我们操作，详见 8.15.4 节。

8.1.5 指针容器的分类

与标准容器一样，指针容器也分为两大类：序列容器和关联容器，基本上标准容器都有对应的加 ptr_ 前缀的同名指针容器（实际上它们都是基于标准容器实现的）。

序列指针容器

ptr_container 库目前提供的序列指针容器包括以下几种。

- ptr_vector : 类似 std::vector 的向量容器。
- ptr_deque : 类似 std::deque 的双端队列容器。
- ptr_list : 类似 std::list 的双向链表容器。
- ptr_array : 类似 std::array 的数组容器。
- ptr_circular_buffer : 类似 boost::circular_buffer 的循环缓冲区容器。

序列指针容器类的关系如图 8-1 所示。

关联指针容器

ptr_container 库目前提供的关联指针容器包括以下几种。

- ptr_set : 类似 std::set 的有序集合容器。
- ptr_multiset : 类似 std::multiset 的有序集合容器。
- ptr_map : 类似 std::map 的有序映射容器。
- ptr_multimap : 类似 std::multimap 的有序映射容器。
- ptr_unordered_set : 类似 std::unordered_set 的无序集合容器。
- ptr_unordered_map : 类似 std::unordered_map 的无序映射容器。

关联指针容器类的关系如图 8-2 所示。

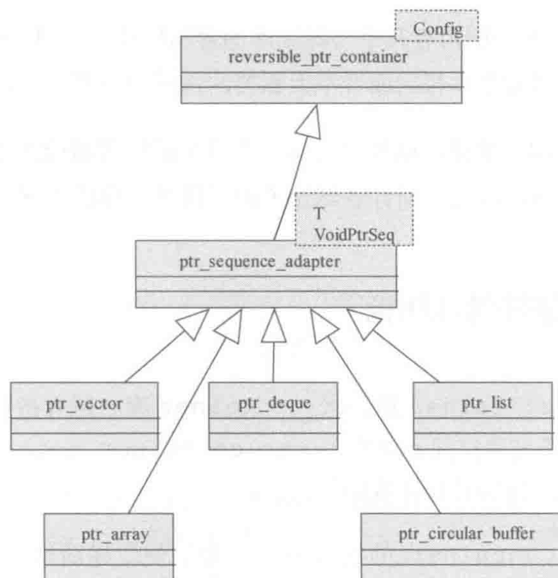


图 8-1 序列指针容器的关系图

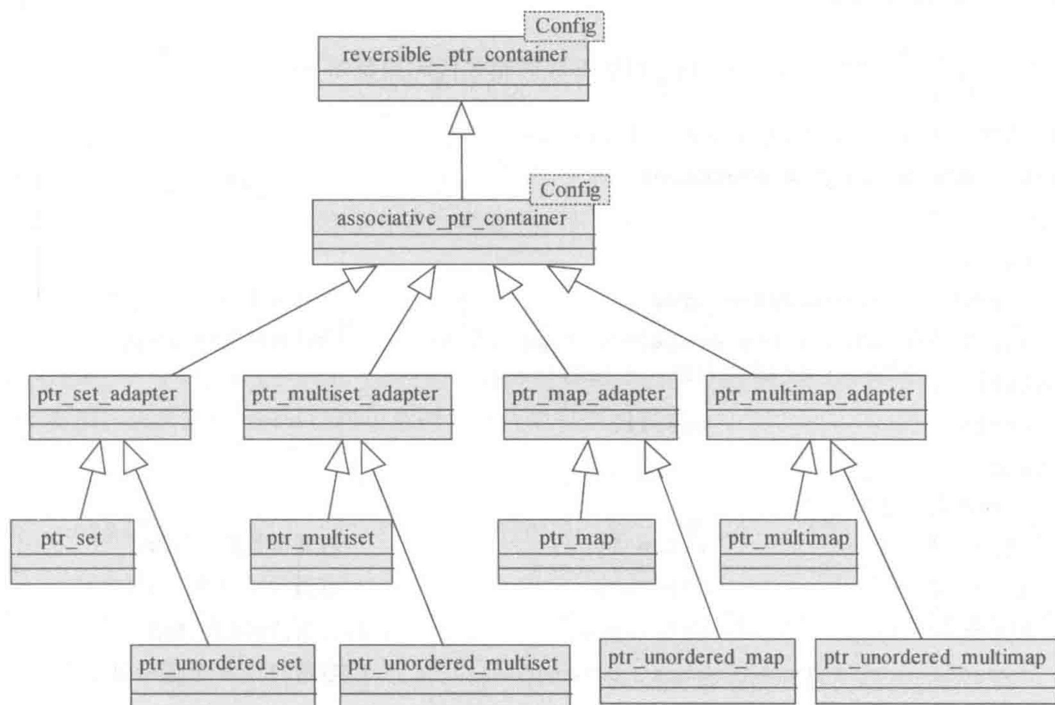


图 8-2 关联指针容器的关系图

由于 ptr_container 库的内容几乎与标准库的容器同样大，所以本书不能完全介绍所有相关的接口和用法，与标准容器基本相同或者不太重要的内容会一带而过，请读者见谅。

本章接下来的部分做如下安排：以类图为脉络先研究指针容器的共通功能，然后顺序研究序列指针容器和关联指针容器，再研究指针容器相关的空指针、算法等深层次概念。

8.2 指针容器的共通功能

reversible_ptr_container 是 ptr_container 库中所有指针容器的基类，包含了指针容器的基本操作，它位于名字空间 boost::ptr_container_detail，通过研究它可以了解指针容器的基本实现原理，加深对指针容器的认识。

reversible_ptr_container 的接口很多，为了使叙述清晰，我们将逐步介绍它的类摘要。

8.2.1 模板参数

reversible_ptr_container 模板参数的相关代码摘要如下：

```
template<class Config, CloneAllocator>
class reversible_ptr_container
{
private:
    typedef Config::value_type              Ty_;      //指针容器的内部值类型
    typedef Config::void_container_type     Cont;     //指针容器的内部容器类型
public:
    Cont&                  base();
public:
    //基本类型定义
    typedef  Ty_*          value_type;                //容器值类型，即指针
    typedef  Ty_*          pointer;                   //容器指针类型，同值类型
    typedef  Ty_&          reference;                 //容器的值引用类型
    typedef  const Ty_&    const_reference;           //容器的 const 值引用类型

    //迭代器类型
```

```
    typedef  Config::iterator                        iterator;
    typedef  boost::reverse_iterator< iterator >  reverse_iterator;

    //指针容器内部的智能指针类型
    typedef  static_move_ptr<Ty_,Deleter>            auto_type;
};
```

reversible_ptr_container 主要的模板参数是 Config，它实际上是一个可返回多个元数据的非标准元函数，可用于元计算有关容器的各种类型，如值、迭代器等。其代码摘要如下：

```
struct Config
{
    typedef some_define   void_container_type;          //容器类型
    typedef some_define   allocator_type;               //内存分配器类型
    typedef some_define   value_type;                   //值类型
    typedef some_define   iterator;                     //迭代器类型
    typedef some_define   const_iterator;               //const 迭代器类型
};
```

reversible_ptr_container 使用 Config 元函数简化了指针容器所需的模板数量，可以一次性获得多个构造指针容器所必需的类型，对于 reversible_ptr_container 来说最重要的就是值类型 Config::value_type 和迭代器类型 Config::iterator/const_iterator，它们定义了指针容器存储的值类型、引用类型和迭代器类型，具体的定义则因具体实现而不同。

ptr_container 内部使用 void*来存储指针，类型 Config::void_container_type 定义了容纳 void*指针的容器，是各种与标准容器对应的指针容器的内部实现，它被重定义为 Cont 类型，成员函数 base()可以取得这个内部容器的引用。

reversible_ptr_container 里另一个重要的类型是 auto_type。它的真实类型是 ptr_container_detail::static_move_ptr，是一个类似于 unique_ptr 的智能指针，使用 boost::compressed_pair（见 4.1 节）来存储指针和对应的删除器，支持隐式 bool 检查，析构时会自动删除指针，release()成员函数同样不会删除指针，仅仅是释放所有权。

下面的代码使用 type_traits 库验证了 ptr_vector 容器的一些内部类型定义：

```
typedef ptr_vector<string> ptr_vec;
```

```
assert((is_same<ptr_vec::value_type, string*>::value));
assert((is_same<ptr_vec::pointer    , string*>::value));
assert((is_same<ptr_vec::reference , string&>::value));
```

8.2.2 构造与赋值

reversible_ptr_container 支持多种形式的构造函数和赋值操作，下面的代码仅列出了其中较常用的一部分：

```
class reversible_ptr_container
{
public:
    //构造函数
    reversible_ptr_container();

    reversible_ptr_container( const reversible_ptr_container& r );
    reversible_ptr_container( const reversible_ptr_container<C,V>& r );
    reversible_ptr_container( InputIterator first, InputIterator last);

    explicit reversible_ptr_container( std::auto_ptr<PtrContainer> clone ) ;

    //析构函数
    ~reversible_ptr_container();

    //赋值操作
    reversible_ptr_container& operator=( const reversible_ptr_container& r );
    reversible_ptr_container& operator=( std::auto_ptr<PtrContainer> clone );
};
```

reversible_ptr_container 有以下三种形式的构造函数。

- 最简单的缺省构造函数可以创建出一个不包含任何指针的空容器。
- 如果传递另一个指针容器或者一个迭代器区间，那么构造函数会使用克隆分配器克隆其中的所有元素，新容器拥有原容器里所有元素的等价副本，但原容器里的指针不受影响。
- 如果把一个指针容器的 auto_ptr 传递给构造函数，那么指针容器将使用转移语义，接

管原容器的所有指针。

reversible_ptr_container 的赋值操作与同参数的构造函数的功能和效果类似，但它只有两种形式，分别支持从一个指针容器或者一个自动指针赋值。

示范这些构造函数和赋值操作的代码如下：

```
typedef ptr_vector<string> ptr_vec;         //一个容纳 string 的指针容器

ptr_vec vec;                                //无参缺省构造函数
assert(vec.empty());                        //指针容器为空

vec.push_back(new string("123"));           //添加两个元素
vec.push_back(new string("abc"));
assert(vec.size() == 2);

ptr_vec vec2(vec);                          //从另一个容器克隆构造
assert(vec2.size() == 2);                   //两个容器各自拥有等价的对象
assert(vec.size() == 2);

auto apv = vec.release();                   //释放所有指针的所有权
assert(vec.empty());

ptr_vec vec3(apv);                          //从一个 auto_ptr 构造
assert(vec3.size() == 2);

ptr_vec vec4, vec5;                         //两个指针容器缺省构造

vec4 = vec2;                                //赋值操作，克隆
assert(vec2.size() == 2);                   //两个容器各自拥有等价的对象
assert(vec4.size() == 2);

vec5 = vec3.release();                      //从一个 auto_ptr 赋值
assert(vec3.empty());
assert(vec5.size() == 2);
```

8.2.3 访问元素

reversible_ptr_container 提供了一些通用的访问元素的接口，代码摘要如下：

```
class reversible_ptr_container
```

```
{
public:
  //插入操作
  iterator    insert( iterator before, Ty_* x );
  iterator    insert( iterator before, std::auto_ptr<U> x );

  //删除操作
  iterator    erase( iterator x ) ;
  iterator    erase( iterator first, iterator last ) ;

  //替换操作
  auto_type   replace( iterator where, Ty_* x ) ;
  auto_type   replace( iterator where, std::auto_ptr<U> x );

  //释放所有权操作
  auto_type                     release( iterator where );
  std::auto_ptr<this_type>      release();

  //克隆操作
  std::auto_ptr<this_type>      clone() const;
}
```

reversible_ptr_container 的插入、删除操作与标准容器相同，只是多了对 auto_ptr 的重载：都是对当前位置进行操作，并返回操作后的新位置。

replace() 是指针容器独有的功能，它可以在容器中替换当前位置的指针，然后以 auto_type 返回当前位置的原指针，有点类似于赋值操作。

成员函数 release() 可以释放一个指针或者整个指针容器的所有权，这可以在函数返回时使用，避免了昂贵的拷贝操作。成员函数 clone() 同样返回一个持有指针容器的自动指针，但它使用了克隆构造函数，返回一个与本容器等价的克隆副本，并不释放指针的所有权。

示范这些操作的代码如下：

```
typedef ptr_vector<string> ptr_vec;        //一个容纳 string 的指针容器
ptr_vec vec;

vec.push_back(new string("123"));          //添加两个元素
vec.push_back(new string("abc"));
assert(vec.size() == 2);
```

```
auto pos =                              //在容器前端执行插入操作，返回新元素的位置
  vec.insert(vec.begin(), new string("000"));
assert(vec.size() == 3);
assert(pos == vec.begin());

++pos;                                  //pos 指向元素"123"
pos = vec.erase(pos);                   //删除操作，返回下一个元素的位置
assert(vec.size() == 2);
assert(*pos == "abc");

auto p =                                //替换操作，返回被替换的元素
  vec.replace(pos, new string("xyz"));
assert(*p == "abc");
assert(vec.size() == 2);

ptr_vec vec2;
vec2 = vec.clone();                     //克隆指针容器并赋值，与 release()不同
assert(vec.size() == 2);                //原容器元素不变
assert(vec2.size() == 2);
```

8.2.4 其他功能

除了以上列举的功能之外，reversible_ptr_container 还具有标准容器所应当具有的一些接口，包括容量、迭代器等操作。因为它们的功能都很简单，而且大都是直接委托给内部的容器处理，所以下面仅列出接口的声明，不再做解释。值得注意的是指针容器的比较操作是基于指针指向的内容，而不是直接比较指针的，这与标准容器一致。

reversible_ptr_container 的其他接口代码摘要如下：

```
class reversible_ptr_container
{
public:
  size_type          size() const;          //容量操作
  size_type          max_size() const;
  bool               empty() const;
  void               swap( reversible_ptr_container& r );
  void               clear():

public:
  iterator           begin();               //迭代器操作
```

```
    iterator                end();
    reverse_iterator        rbegin();
    reverse_iterator        rend();

  public:
    ...                              //各种比较操作符的重载，包括==、!=、<等
};
```

8.3 序列指针容器适配器

ptr_sequence_adapter 是序列指针容器的基类，它使用适配器设计模式把序列普通容器适配成序列指针容器，适配的对象有 vector、list、array 等，如果有必要，那么也可以让它适配其他容器实现定制自定义的指针容器。ptr_sequence_adapter 位于头文件 <boost/ptr_container/ptr_sequence_adapter.hpp>。

8.3.1 配置元函数

ptr_sequence_adapter 使用的配置元函数是 sequence_config，它的类摘要如下：

```
template< class T, class VoidPtrSeq>
struct sequence_config
{
    typedef  remove_nullable<T>::type U;                    //元素值类型
    typedef  VoidPtrSeq              void_container_type;//被适配的容器类型
    typedef  VoidPtrSeq::allocator_type allocator_type;  //内存分配器
    typedef  U                       value_type;          //值类型
    typedef  void_ptr_iterator<VoidPtrSeq::iterator, U >
                                     iterator;            //迭代器类型
    typedef  void_ptr_iterator<VoidPtrSeq::const_iterator, const U >
                                     const_iterator;      //const 迭代器类型
};
```

sequence_config 有两个模板参数，T 是容器的元素类型，VoidPtrSeq 是容纳 void* 指针的序列容器类型，元函数 remove_nullable<T>可用于移除元素类型可为空指针的修饰（参见 8.5.2 节），得到真正的元素类型 U。

sequence_config 中的 void_ptr_iterator 是一个完整的迭代器类，它适配了操作 void*指针元素的迭代器 VoidPtrSeq::iterator，使用 iterator_traits（参见 5.3 节）

元计算与容器迭代器相关的迭代器类型，在 operator*()、operator->()等操作符重载中用 static_cast 把 void*指针转换成 T*，使我们可以方便地用迭代器直接操作元素而非指针。

8.3.2 类摘要

ptr_sequence_adapter 是 reversible_ptr_container 的子类，所以具有 reversible_ptr_container 的全部功能，下面的类摘要仅列出新增的接口：

```
template
< class T, class VoidPtrSeq,
  class CloneAllocator = heap_clone_allocator
>
class ptr_sequence_adapter : public
    reversible_ptr_container<
        sequence_config<T,VoidPtrSeq>, CloneAllocator >
{
public:
    assign( InputIterator first, InputIterator last );   //构造与赋值
public:
    T&        front();                                   //访问元素
    T&        back();
    void      push_back( T* x );
    void      push_back( std::auto_ptr<U> x );
    auto_type pop_back();
    void      resize( size_type size );
    void      resize( size_type size, T* to_clone );
public:
    //指针所有权转移
    void      transfer( iterator before,
                        PtrSequence::iterator object, PtrSequence& from );

    void      transfer( iterator before,
                  PtrSequence::iterator first, PtrSequence::iterator last,
                  PtrSequence& from );

    void      transfer( iterator before, PtrSequence& from );
    void      transfer( iterator before, value_type* from,
```

```
                    size_type size, bool delete_from = true );
};
```

ptr_sequence_adapter 有三个模板参数，前两个（T 和 VoidPtrSeq）被传递给 sequence_config 形成配置元函数，再和克隆分配器参数 CloneAllocator 一起传递给 reversible_ptr_container。

ptr_sequence_adapter 还提供了一些内置的算法，如 sort()、erase_if()、merge()和 unique()等，这些将在 8.14 节进行讨论。

8.3.3 接口解说

ptr_sequence_adapter 具有序列容器的通用接口，如 assign()、front()、push_back()，这些接口的功能与标准容器的同名接口完全相同，用法也是一样的。在这里我们要稍微留意一下 resize()函数：在扩大容器的容量时它需要使用 T 的缺省构造函数来创建对象填充序列，或者是使用一个指定的克隆。

transfer()系列函数用于指针的所有权管理，是指针容器的核心操作，它们可以在两个指针容器之间移动指针，三种形式具体效果如下所述。

- (before,object,from)：把 object 插入到当前容器的 before 位置之前，并从 from 中移除所有权。
- (before,first,last,from)：把[first,last)之间的元素插入到当前容器的 before 位置之前，并从 from 中移除这些元素的所有权。
- (before,from)：把 from 里的所有元素插入到当前容器的 before 位置之前，之后 from 不再持有任何元素。

8.3.4 代码示例

我们仍以最简单的 ptr_vector 来示范这些序列指针容器的共通功能。首先是基本的元素访问功能：

```
typedef ptr_vector<string> ptr_vec;        //容纳 string 的指针容器
ptr_vec vec;

vec.push_back(new string("123"));          //添加两个元素
vec.push_back(new string("abc"));
```

```
assert(vec.front() == "123");                //使用 front()获取序列前端元素
assert(vec.back() == "abc");                 //使用 back()获取序列末端元素

ptr_vec vec2;
vec2.assign(vec.begin(), vec.end());         //从另一个指针容器赋值
assert(vec2.size() == 2);

assert(*vec2.pop_back() == "abc");           //弹出序列末端元素
assert(vec2.size() == 1);

vec.resize(5);                               //修改容器的大小，多出的元素使用缺省构造
assert(vec.back().empty());

vec.resize(10, new string("xyz"));           //修改容器的大小，多出的元素使用克隆
assert(vec.back() == "xyz");
```

接下来我们使用 transfer()系列函数来转移指针的所有权：

```
//转移一个元素
vec.transfer(vec.end(), vec2.begin(), vec2);
assert(vec.size() == 11);
assert(vec2.size() == 0);

//转移一个迭代器区间
vec2.transfer(vec2.end(), vec.begin(),vec.begin() + 5, vec);
assert(vec.size() == 6);
assert(vec2.size() == 5);

//转移整个容器
vec.transfer(vec.begin(), vec2);
assert(vec.size() == 11);
assert(vec2.size() == 0);
```

8.4 ptr_vector

指针容器 ptr_vector 是最简单常用的指针容器，它基于标准容器 std::vector 容纳 void* 指针，使用 ptr_sequence_adapter 适配实现，位于头文件<boost/ptr_container/ptr_vector.hpp>。

ptr_deque、ptr_list、ptr_array 等的接口与 ptr_vector 基本相同，读者可参考 GitHub 资源了解用法。

8.4.1 类摘要

ptr_vector 是 ptr_sequence_adapter 的子类，下面的类摘要仅列出它的新增接口：

```
template
<  class T,
   class CloneAllocator    = heap_clone_allocator,
   class Allocator         = std::allocator<void*> >
class ptr_vector : public ptr_sequence_adapter
                    <  T,
                       std::vector<void*,Allocator>,
                       CloneAllocator>
{
public:
    explicit      ptr_vector( size_type n );          //构造函数

public:
    size_type     capacity() const;                   //容量
    void          reserve( size_type n );
public:
    T&            operator[] ( size_type n );         //访问元素
    T&            at( size_type n );
public:
    auto_type     replace( size_type idx, T* x );     //替换操作
    auto_type     replace( size_type idx, std::auto_ptr<U> x );
public:
    void          transfer( iterator before, T** from, //C 风格数组的支持
                            size_type size, bool delete_from = true );
    T**           c_array();
};
```

ptr_vector 有三个模板参数，但通常只给定一个元素类型 T 即可，它会自动把 T 和 std::vector<void*>提供给父类 ptr_sequence_adapter 从而完成模板参数的配置。

ptr_vector 基于标准容器 vector，接口与 vector 基本相同，大部分操作都使用 base() 转发给内部的容器实现，因此很容易理解。但构造函数 ptr_vector(n) 的行为不同于

std::vector，它不会创建 n 个元素，而是保留 n 个元素的空间，相当于调用 reserve(n)。

因为 ptr_vector 支持随机访问，所以它有 operator[]，可以用整数索引直接访问元素，也可以用索引来指示位置来替换元素。

为了像 vector 一样提供对数组的良好兼容性，ptr_vector 增加了一个 c_array() 函数，它可以返回一个类型为 T**的指针，即 T*[]，可以传递给 C 风格的 API。成员函数 transfer() 也有对应的数组形式的重载，可操作动态创建的指针数组，把大小为 size 的 from 数组中的所有元素插入到当前容器的 before 位置之前，如果 delete_from == true，那么函数执行后 from 将被删除（即 delete[] from）。

8.4.2 用法

ptr_vector 继承了 std::vector 的优良传统，无疑是所有指针容器中最容易使用的一个，之前已经多次使用它演示了指针容器的用法，现在只需要很少的代码来展示它其余的特性：

```
typedef ptr_vector<string> ptr_vec;                    //向量指针容器
ptr_vec vec(10);                                       //保留 10 个元素的空间待用

assert(vec.empty());                                   //此时容器内无元素
assert(vec.capacity() == 10);                          //可容纳 10 个元素

vec.push_back(new string("star"));                     //尾部添加一个元素
assert(vec[0] == "star");

vec.replace(0, new string("fox"));                     //使用整数索引
assert(vec[0] == "fox");                               //使用 operator[]

auto arr = new string*[2];                             //一个动态指针数组
arr[0] = new string("123");
arr[1] = new string("abc");

vec.transfer(vec.begin(), arr, 2);                     //转移指针的所有权
assert(vec.size() == 3);                               //此时原动态指针数组已经失效

auto p = vec.c_array();                                //获得指针数组
assert(*p[0] == "123" && *p[2] == "fox");
```

8.5 空指针处理

提到指针，有一个特别的概念不得不涉及——那就是空指针（nullptr），对于专门容纳指针的 ptr_container 库来说，这更是一个重要的议题。

8.5.1 禁用空指针

如果不做什么特别的声明，那么 ptr_container 库的所有指针容器均不允许处理空指针，也就是说不可能向容器插入或者从容器中取出空指针（ptr_array 是一个例外）。如果向容器中插入一个空指针，那么会得到一个 boost::bad_pointer 异常，比如：

```
ptr_vector<int> vec;
vec.push_back(nullptr);                          //抛出 bad_pointer 异常

ptr_list<string> lt;
lt.push_front(nullptr);                          //抛出 bad_pointer 异常
```

指针容器的这种处理方式很好地防止了无意中使用空指针所可能带来的灾难性后果，可以让代码更加安全。

8.5.2 使用空指针

然而有的时候用户又确实需要向容器中插入空指针，ptr_container 库为此引入了一个特别的包装类：nullable<T>。

在指针容器声明时使用 nullable<T>作为元素类型（而不是直接使用 T）即表明容器可容纳空指针，例如：

```
ptr_vector<nullable<int> > vec;                  //使用 nullable<T>包装元素类型
vec.push_back(nullptr);                          //可插入空指针

ptr_list<nullable<string> > lt;                  //使用 nullable<T>包装元素类型
lt.push_front(nullptr);                          //可插入空指针
```

nullable<T>对元素类型的包装并不影响指针容器的接口，因为指针容器内部已经使用元函数去除了 nullable<T>的包装，它的作用仅仅是在编译期给指针容器一个提示而已，我们千万不可写出下面的代码：

```
ptr_vector<nullable<int> > vec;
vec.push_back(new nullable<int>(10));          //编译错误
```

引入空指针后指针容器的处理就变得复杂了一些，因为对空指针的操作是无效的，所以在访问元素时必须时刻检查指针是否为空。ptr_container 库提供了一个自由函数 is_null()，用来检查指针容器的迭代器是否指向了一个“真正的”空指针：

```
template< class Iterator, class T >
inline bool is_null( void_ptr_iterator<Iterator,T> i );
```

对于 ptr_vector 这样的支持整数索引的指针容器还有一个成员函数 is_null()，它也可以检查当前位置上是否是空指针：

```
bool is_null( size_t idx ) const;
```

空指针检查函数可以这样使用：

```
typedef ptr_vector<nullable<int>> ptr_null_vec;  //可容纳空指针的容器
ptr_null_vec vec;

vec.push_back(nullptr);                          //添加一个空指针
vec.push_back(new int(100));                     //添加一个正常元素

assert(vec.is_null(0));                          //使用成员函数检查空指针
assert(!vec.is_null(1));

for (auto i = vec.begin();                       //使用迭代器迭代元素
  i != vec.end(); ++i)                           //因为空指针的原因不能使用 for
{
  if (!boost::is_null(i))                        //使用自由函数检查是否为空指针
  {   cout << *i << " "; }
}
```

8.5.3 空对象模式

虽然 ptr_container 库使用 nullable<T> 允许我们在容器中存储空指针，但这并不是问题的最佳解决方案。空指针会迫使我们在每次访问元素时都进行检查以防止出错，破坏了代码的优雅和整洁，同时空指针的隐患并没有消除，一旦疏忽大意，对空指针的操作就会使整个程序崩溃。

这个时候可以采用“智能空指针”——空对象模式（null object）来解决这个问题。空对象是一个模仿了空指针的对象，它给空指针赋予了一个合理的、可接受的行为（通常是空操作），使得代码可以一致地处理实对象和空对象，无须再使用专门的条件判断语句来检查空指针。

下面首先实现一个简单的多态类继承体系，注意其中使用了 boost::noncopyable，类关系图如图 8-3 所示。

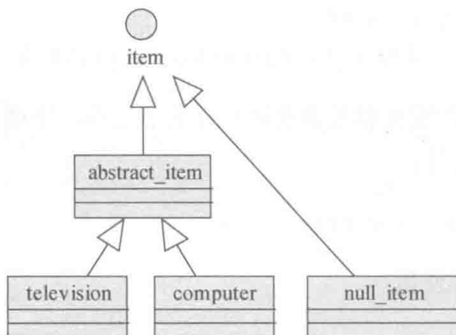


图 8-3 简单的多态类继承体系

```
class item : boost::noncopyable                    //接口类
{
public:
  virtual ~item(){}                                //析构函数必须是虚的
  virtual void print() = 0;                        //纯虚函数
};

class abstract_item : public item                  //抽象类
{
  string name;
public:
  abstract_item(const string& str):name(str){}
  virtual ~abstract_item(){}                       //析构函数必须是虚函数

  virtual void print() final override              //注意 final 的使用
  {   cout << name << endl; }
};

class television : public abstract_item {...};
```

```
class computer : public abstract_item {...};

class null_item final: public item                         //空对象，final
{ virtual void print(){} };                                //什么也不做的空操作
```

因为基类使用了 boost::noncopyable，所有这些类都是不可拷贝的，无法被标准容器所容纳，但指针容器没有这些限制，所以可以管理它们。null_item 实现了 item 要求的所有接口，但没有任何有意义的操作，所以是一个空对象，可以当作一个空指针来使用。

```
typedef ptr_vector<item> ptr_vec;          //指针容器，不使用 nullable<T>
ptr_vec vec;

vec.push_back(new television);             //添加对象
vec.push_back(new computer);
vec.push_back(new null_item);              //添加空对象，相当于空指针

for(auto& i : vec)                         //可以使用新式 for 遍历容器
{      i.print();}                         //无须空指针判断

vec.replace(2, new computer);              //空对象可以在适当的时候被替换成实对象
vec[2].print();
```

对比 8.5.2 节的代码，很明显，空对象模式的使用简化了我们的操作代码。

这个例子中的类体系是 noncopyable 的（不能拷贝构造），并且没有定义 new_clone() 和 delete_clone()函数，因此它们不支持可克隆概念，但仍然可以使用指针容器容纳。关于它们的进一步讨论可见 8.15.5 节。

8.6 关联指针容器的共通功能

associative_ptr_container 是关联指针容器的基类，定义了一些关联指针容器的共有特征，它位于名字空间 boost::ptr_container_detail。

8.6.1 类摘要

associative_ptr_container 是 reversible_ptr_container 的子类，由于集合与

映射这两种关联容器的差距较大，它只有很少的公开接口，大部分是保护的成员，类摘要如下(reversible_ptr_container 已有的未列出)：

```
template<class Config, class CloneAllocator>
class associative_ptr_container :
    public reversible_ptr_container<Config,CloneAllocator>
{
public:
  //类型定义
  typedef  Config::key_type        key_type;
  typedef  Config::key_compare     key_compare;
  typedef  Config::value_compare   value_compare;
public:
  //函数对象访问
  key_compare                      key_comp() const;
  value_compare                    value_comp() const;
public:
  //删除操作
  iterator                         erase( iterator before );
  size_type                        erase( const key_type& x );
  iterator                         erase( iterator first, iterator last );
};
```

8.6.2 接口解说

associative_ptr_container 有大量的内部保护成员供子类复用，仅提供了少量的公开接口，但这些接口都很重要。它使用配置元函数增加了几个新的内部类型定义，是关联容器所必需的类型定义。

- key_type ：对于集合是值类型 value_type(即元素 T)，对于映射是键的类型。
- key_compare ：键值比较函数对象的类型。
- value_compare ：值的比较函数对象的类型，基本相当于 key_compare。

成员函数 key_comp() 和 value_comp() 分别用于返回比较函数对象的拷贝。

8.7 集合指针容器适配器

集合指针容器可以分为有序和无序，也可以分为不允许重复（单键）和允许重复（多键），所以它的适配器比序列指针容器适配器要复杂一些。[①]

8.7.1 配置元函数

集合指针容器使用的配置元函数是 set_config，类摘要如下：

```
template
<   class Key,                                          //集合元素类型
    class VoidPtrSet,                                   //被适配的集合容器类型
    bool  Ordered>                                      //是否有序
struct set_config
{
   typedef VoidPtrSet                    void_container_type;//内部集合容器容器
   typedef VoidPtrSet::allocator_type allocator_type;    //内存分配器

   typedef Key                           value_type;         //值类型
   typedef value_type                    key_type;           //键类型同值类型

   typedef mpl::eval_if_c<...>::type value_compare;      //有序专用
   typedef value_compare                 key_compare;        //有序专用

   typedef mpl::eval_if_c<...>::type hasher;             //无序专用
   typedef mpl::eval_if_c<...>::type key_equal;          //无序专用

   typedef mpl::if_c<...>::type        container_type;

   typedef void_ptr_iterator<...>        iterator;
   typedef void_ptr_iterator<...>        const_iterator;
};
```

set_config 有三个模板参数，Key 和 VoidPtrSet 与序列指针容器适配器的配置元函数 sequence_config（见 8.3.1 节）含义相同，分别是容纳的元素类型和容纳 void*指针的集

① 由于 ptr_multiset、ptr_multimap 等多键容器的接口与单键容器基本相同，为节省篇幅本书不做介绍，读者可参考 GitHub 源码或者 Boost 文档学习。

合容器类型，bool 参数 Ordered 标志了集合是有序还是无序的。

set_config 接下来的元计算实现了集合指针容器所需的各种类型，这些元函数都非常简单，读者可参照 sequence_config。

还需要注意的是集合指针容器没有使用 nullable<T>系列元函数，因为对于集合这样的关联容器来说容纳空指针没有意义。我们不能使用 nullable<T>作为模板参数，也不能向集合指针容器中插入空指针，这与序列指针容器相比是一个很大的不同。

8.7.2 ptr_set_adapter

ptr_set_adapter 用于适配不允许重复（单键）的集合容器，它的类摘要如下：[①]

```
template
<   class Key,
    class VoidPtrSet,
    class CloneAllocator = heap_clone_allocator,
    bool  Ordered        = true >
class ptr_set_adapter: public
      associative_ptr_container< set_config<Key,VoidPtrSet,Ordered>,
                                 CloneAllocator >
{
public:
    //集合特有的插入操作
    std::pair<iterator,bool>  insert( key_type* x );
    std::pair<iterator,bool>  insert( std::auto_ptr<U> x );

public:
    //指针所有权的转移
    bool                transfer( iterator object,
                                  ptr_set_adapter& from );

    size_type           transfer( iterator first,
                                  iterator last,
                                  ptr_set_adapter& from );
```

① 实际上 ptr_set_adapter 的真正父类是 ptr_set_adapter_base，这里做了适当的简化。

```
    size_type           transfer( PtrSetAdapter& from );
};
```

ptr_set_adapter 是 reversible_ptr_container 的子类,具有 reversible_ptr_container 的全部功能，还额外增加了集合容器特有的插入操作。

insert()在插入元素时会返回一个 std::pair，除了表明新位置的迭代器外还有一个 bool 值，用来表示插入操作是否成功，这是因为单键集合不允许重复的值，如果值已经存在则插入操作会失败。

ptr_set_adapter 同样提供一系列用于指针的所有权管理的 transfer()函数，它们的行为与序列指针容器完全相同，可参见 8.3.3 节。

8.8 ptr_set

ptr_set 是有序集合指针容器，它使用标准容器 std::set 容纳 void*指针，位于头文件 <boost/ptr_container/ptr_set.hpp>。

8.8.1 类摘要

ptr_set 的代码实现非常简单，因为大部分工作都已经在适配类中完成了，类摘要如下：

```
template
<  class Key,
   class Compare          = std::less<Key>,
   class CloneAllocator   = heap_clone_allocator,
   class Allocator        = std::allocator<void*> >
class ptr_set :
   public ptr_set_adapter< Key,
         std::set<void*,void_ptr_indirect_fun<Compare,Key>,Allocator>,
                       CloneAllocator, true >
{...};
```

代码中的 void_ptr_indirect_fun 是一个用于比较 void*指针的函数对象，它在执行比较操作时使用 static_cast 把 void*转换为 Key 类型，然后再使用 Compare 进行比较，可参见 8.15.2 节。

8.8.2 用法

ptr_set 用起来与标准容器 std::set 没有太大的差别，结合之前序列指针容器的使用经验可以很快掌握。再提醒一下，这些集合容器不能容纳空指针。

示范 ptr_set 用法的代码如下：

```
typedef ptr_set<string> ptr_set_t;                    //有序集合指针容器
ptr_set_t s;                                          //缺省构造
assert(s.empty());                                    //容器为空

assert(s.insert(new string("fire")).second);          //插入元素
assert(s.insert(new string("emblem")).second);
assert(s.size() == 2);

assert(!s.insert(new string("fire")).second);         //不允许重复元素
assert(s.size() == 2);

auto ap = s.release();                                //释放指针的所有权
assert(s.empty());
```

8.9 ptr_unordered_set

ptr_unordered_set 是无序集合指针容器，它使用 boost::unordered_set 容纳 void*指针，位于头文件<boost/ptr_container/ptr_unordered_set.hpp>。

8.9.1 类摘要

ptr_unordered_set 是散列容器，不保持元素的有序，因此它们的模板参数不同于标准容器，需要使用散列函数对象和相等函数对象，缺省是 boost::hash（见 7.1 节）和 std::equal_to，类摘要如下：

```
template
<   class Key,
    class Hash              = boost::hash<Key>,
    class Pred              = std::equal_to<Key>,
    class CloneAllocator    = heap_clone_allocator,
```

```
    class Allocator        = std::allocator<void*> >
class ptr_unordered_set :
    public ptr_set_adapter< Key,
      boost::unordered_set<void*,void_ptr_indirect_fun<Hash,Key>,
                         void_ptr_indirect_fun<Pred,Key>,Allocator>,
                         CloneAllocator, false >
{
public:
    //散列容器函数对象类型定义
    typedef  Config::hasher        hasher;
    typedef  Config::key_equal     key_equal;
private:
    //注意，有序容器相关的接口均禁用
    using base_type::rbegin;
    using base_type::rend;
    using base_type::crbegin;
    using base_type::crend;
    using base_type::key_comp;
    using base_type::value_comp;
    using base_type::front;
    using base_type::back;
public:
    using base_type::begin;
    using base_type::end;
    ...         //其他集合操作函数
};
```

ptr_unordered_set 通过声明父类接口为 private 的方式禁用了一些只有有序容器才能使用的接口，如 rbegin()、front()、back()等。

8.9.2 用法

虽然内部实现机制不同，但 ptr_unordered_set 仍然具有集合容器的大部分标准接口，只是要注意集合是无序的这个特点，不能在无序容器上应用有序区间的算法。

示范 ptr_unordered_set 用法的代码如下，它们与 8.8.2 节的很类似，只是改用了 transfer()成员函数来转移指针的所有权：

```
typedef ptr_unordered_set<string> ptr_set_t;          //无序集合指针容器
ptr_set_t s;
assert(s.empty());

assert(s.insert(new string("king")).second);          //插入元素
assert(s.insert(new string("kong")).second);
assert(s.size() == 2);

assert(!s.insert(new string("king")).second);         //不允许重复元素
assert(s.size() == 2);

ptr_set_t s2;
s2.transfer(s);                                       //转移s里的所有指针
assert(s.empty() && s2.size() == 2);
```

8.10 映射指针容器适配器

ptr_container库的映射指针容器同样可以分为有序和无序、不允许重复（单键）和允许重复（多键），位于头文件<boost/ptr_container/ptr_map_adapter.hpp>。

8.10.1 配置元函数

映射指针容器使用的配置元函数是map_config，类摘要如下：

```
template
<  class T,
   class VoidPtrMap,
   bool  Ordered >
struct map_config
{
  typedef remove_nullable<T>::type    U;                  //容器的值类型
  typedef VoidPtrMap              void_container_type;    //被适配的容器类型

  typedef  VoidPtrMap::allocator_type allocator_type;

  typedef mpl::eval_if_c<...>::type  value_compare;       //有序专用
```

```
  typedef  mpl::eval_if_c<...>::type key_compare;        //有序专用

  typedef mpl::eval_if_c<...>::type  hasher;             //无序专用
  typedef mpl::eval_if_c<...>::type  key_equal;          //无序专用

  typedef mpl::if_c<...>::type       container_type;
  typedef  VoidPtrMap::key_type      key_type;           //键类型
  typedef U                          value_type;         //值类型

  typedef ptr_map_iterator<...>      iterator;
  typedef ptr_map_iterator<...>      const_iterator;
};
```

由于都是关联容器，所以 map_config 与 set_config 相同，都有三个模板参数。T 和 VoidPtrMap 分别映射容器的 value 类型（key 类型不在这里指定）和被适配的映射容器类型，bool 参数 Ordered 标志了映射是有序还是无序的。

map_config 同样元计算了映射指针容器所需的各种类型，只是迭代器类型不是之前一直使用的 void_ptr_iterator，这是因为映射容器容纳的元素是一个 std::pair。ptr_map_iterator 使用了 boost::iterator_adaptor（参见 5.5 节）适配了标准映射容器的迭代器，返回一个模仿 std::pair 的 ref_pair 类型。

我们还需要注意类型 value_type，它使用了元函数 remove_nullable<T>，这意味着我们可以在映射容器中使用值的空指针。

8.10.2 ptr_map_adapter

ptr_map_adapter 用于适配不允许重复（单键）的映射容器，它的类摘要如下：[①]

```
template
<   class T,
    class VoidPtrMap,
    class CloneAllocator   = heap_clone_allocator,
    bool  Ordered          = true >
class ptr_map_adapter :
 public associative_ptr_container< map_config<T,VoidPtrMap,Ordered>,
```

① 实际上 ptr_map_adapter 的真正父类是 ptr_map_adapter_base，这里做了适当的简化。

```
                                          CloneAllocator >
{
public:
    //类型定义
    typedef  base_type::iterator          iterator;
    typedef  base_type::const_iterator    const_iterator;
    typedef  base_type::size_type         size_type;
    typedef  base_type::key_type          key_type;
    typedef  base_type::mapped_type       mapped_type;
    typedef  base_type::const_reference const_reference;
    typedef  base_type::auto_type         auto_type;
    typedef  VoidPtrMap::allocator_type allocator_type;
public:
    //插入操作
    std::pair<iterator,bool>  insert( key_type& key, mapped_type x );

    std::pair<iterator,bool>  insert( const key_type& key,
                                       std::auto_ptr<U> x );

    iterator                  insert( iterator before,
                                       key_type& key, mapped_type x );

    iterator                  insert( iterator before, const key_type& key,
                                       std::auto_ptr<U> x );
public:
    //访问元素
    mapped_reference          at( const key_type& key );
    mapped_reference          operator[]( const key_type& key );

public:
    //指针所有权的转移
    bool                      transfer( PtrMapAdapter::iterator object,
                                         PtrMapAdapter& from );

    size_type                 transfer( PtrMapAdapter::iterator first,
                                         PtrMapAdapter::iterator last,
                                         PtrMapAdapter& from );
```

```
    size_type                       transfer( PtrMapAdapter& from );
};
```

因为映射容器存储的是pair,所以ptr_map_adapter的类型定义较之前的容器有所不同。它的value_type是一个ref_pair类型,映射的左右类型分别定义为key_type和mapped_type（即T*），这与标准容器std::map是一致的。

ptr_map_adapter不允许重复的键值，所以我们可以使用at()和operator[]来直接访问元素。注意：如果要使用operator[]来访问元素，那么要求容纳的元素类型T必须有一个缺省构造函数，而且不能是空指针，否则会发生编译错误。[①]

映射容器的成员函数 insert()直接使用 mapped_type 的形式有点特别，它要求key_type 必须是一个左值，这是出于对异常安全的考虑，const 引用的形式必须使用std::auto_ptr来包装new的结果（示例代码参见8.11.2节）。

指针转移操作与其他的指针容器都是类似的，不再赘述。

8.11 ptr_map

ptr_map是有序映射指针容器，它使用了标准容器std::map容纳void*指针，位于头文件<boost/ptr_container/ptr_map.hpp>。

8.11.1 类摘要

ptr_map的代码实现非常简单，因为大部分工作都已经在适配类中完成了，类摘要如下：

```
template
<   class Key,
    class T,
    class Compare           = std::less<Key>,
    class CloneAllocator    = heap_clone_allocator,
    class Allocator         = std::allocator< std::pair<const Key,void*> >>
class ptr_map :
    public ptr_map_adapter<T,std::map<Key,void*,
                    Compare,Allocator>,CloneAllocator>
{...};
```

① 这一点与使用标准映射容器容纳指针时有明显的不同，但与标准映射容器使用operator[]时的要求一致，读者需要特别留意。

8.11.2 用法

ptr_map 用起来与标准映射容器 std::map 非常类似，在大多数情况下，我们只需要指定模板参数 Key 和 T 就可以正常工作了。

示范 ptr_map 用法的代码如下：

```
typedef ptr_map<int, string> ptr_map_t;                  //有序映射指针容器
ptr_map_t m;

int a = 1;                                               //一个左值
m.insert(a, new string("one"));                          //必须使用左值才能通过编译
m.insert(10, auto_ptr<string>(new string("ten")));       //auto_ptr 可使用右值

assert(m.size() == 2);
assert(m[10] == "ten");                                  //使用 operator[]，返回引用

m.replace(m.begin(), new string("neo"));                 //替换操作
m[3] = "three";                                          //operator[]直接赋值,不用 new
assert(m.at(1) == "neo" && m.at(3) == "three");

for(const auto& i : m)                                   //for 遍历循环
{      cout << *i->second << ",";}                        //second 是一个 T*指针
```

如果允许存储空指针，那么需要注意 operator[] 是无法使用的，可以改用 at()，例如：

```
typedef ptr_map<int, nullable<string> > ptr_map_t;       //允许空指针
ptr_map_t m;

m.insert(10, auto_ptr<string>(new string("ten")));
m.at(10) = "tenten";                                     //编译正常
m[3] = "three";                                          //编译错误
```

8.12 ptr_unordered_map

ptr_unordered_map 是无序映射指针容器,它使用了 Boost 容器 boost::unordered_set_t 容纳 void*指针，位于头文件<boost/ptr_container/ptr_unordered_map.hpp>。

8.12.1 类摘要

ptr_unordered_map 是散列容器，需要使用散列函数对象和相等函数对象，与散列指针集合容器很相似，类摘要如下：

```
template
<   class Key,
    class T,
    class Hash              = boost::hash<Key>,
    class Pred              = std::equal_to<Key>,
    class CloneAllocator    = heap_clone_allocator,
    class Allocator         = std::allocator< std::pair<const Key,void*> >>
class ptr_unordered_map :
    public
ptr_map_adapter<T,boost::unordered_map<Key,void*,Hash,Pred,Allocator>,
                        CloneAllocator,false>
{
public:
    //散列容器函数对象类型定义
    typedef  Config::hasher        hasher;
    typedef  Config::key_equal     key_equal;

private:
    using base_type::rbegin;                    //有序容器相关的接口均禁用
    ...                                         //同 ptr_unordered_set
public:
    using base_type::begin;
    using base_type::end;
    ...                                         //其他集合操作函数
};
```

ptr_unordered_map 同样通过声明父类接口为 private 的方式禁用了一些只有有序容器才能使用的接口。

8.12.2 用法

如果读者熟悉了 ptr_map 和 ptr_unordered_set 的用法，那么 ptr_unordered_map 将会很容易掌握，因为它既有映射容器的特性又有无序容器的特性。

示范 ptr_unordered_map 用法的代码如下：

```
typedef ptr_unordered_map<int, string> ptr_map_t;     //无序映射指针容器
ptr_map_t m;

int a = 1;
m.insert(m.begin(),a, new string("one"));            //使用迭代器位置插入，左值
m.insert(m.end(), 10,                      //使用迭代器位置和 auto_ptr 插入，可用右值
         auto_ptr<string>(new string("ten")));
m[3] = "three";                                      //使用 operator[]

assert(m.at(3) == "three");
assert(m[10] == "ten");                              //使用 operator[]

for(const auto& i : m)                               //也可以使用 foreach
{     cout << i->second << ",";}
```

8.13 使用 assign 库

boost.assign 是一个可以快速方便地赋值或初始化容器的库，它使向容器赋初值的操作变得异常简单，例如：

```
vector<int> v1, v2;                        //两个向量容器

using namespace boost::assign;             //打开 assign 名字空间
v1 += 1, 2, 3;                             //使用 operator+=和 operator,
push_back(v2)()(10), 20, 30;               //使用 operator()和 operator,
```

ptr_container 库出现后，assign 库也提供了对指针容器赋值和初始化的支持，使我们可以不必再使用烦琐的 push_back()函数和 new 去填充元素。

8.13.1 向容器添加元素

assign 库在头文件<boost/assign/ptr_list_inserter.hpp>和<boost/assign/ptr_map_inserter.hpp>中提供了以下四个向指针容器添加元素的辅助函数。

- ptr_push_back<T>() ：使用 push_back()在末端添加元素。
- ptr_push_front<T>() ：使用 push_front()在前端添加元素。

- ptr_insert<T>()　　：使用 insert()添加元素。
- ptr_map_insert<T>()：使用 insert()添加元素。

这些辅助函数接受一个指针容器的引用，返回一个 inserter 函数对象。inserter 重载了 operator()，最多支持五个参数（map 是六个，含键值），内部采用操作符 new 和参数创建对象再调用指针容器的方法添加元素，如果没有参数则使用 T 的缺省构造函数来创建对象。所以我们无须再使用 new，同时还避免了映射容器对键值的左值要求。

在通常情况下，辅助函数创建的对象类型是容器的元素类型（即 T），是使用元编程技术自动推导出来的，但如果有必要，那么辅助函数可以使用模板参数来明确指定创建的对象类型，这在指针容器容纳多态对象时是非常有用的，因为抽象类型无法实例化。

示范这些辅助函数用法的代码如下：

```
using namespace boost::assign;                          //打开 assign 名字空间

ptr_vector<int> v;
ptr_push_back(v)()(1)(2)(100);                          //使用 ptr_push_back()
assert(v.size() == 4);

ptr_list<complex<double> > lt;
ptr_push_front(lt)(1, 2)(0.618, 1.732);                 //使用 ptr_push_front()
ptr_push_back<complex<double>>(lt)(2.718, 3.14);        //指明模板参数

ptr_multimap<int, string> m;
ptr_map_insert(m)(1, "one")(1, "neo");                  //使用 ptr_map_insert()
```

8.13.2 初始化容器元素

函数 ptr_list_of<T>()可以创建一个匿名的指针容器列表 generic_ptr_list<T>，在指针容器构造时直接初始化，较 ptr_push_back()的赋值方式效率更高，它位于头文件 <boost/assign/ptr_list_of.hpp>。

ptr_list_of<T>()在使用时必须指定模板参数 T，也是最多支持使用五个参数来创建对象，如果没有参数则使用 T 的缺省构造函数来创建对象，例如：

```
using namespace boost::assign;                          //打开 assign 名字空间

ptr_vector<int> v = ptr_list_of<int>()(1)(2);           //初始化向量指针容器
```

```
ptr_deque<complex<double> > dq =                    //初始化双端队列指针容器
  ptr_list_of<complex<double>>(1, 2)(0.618, 1.732);

ptr_set<string> s ;
s = ptr_list_of<string>()("abc")("xyz").to_container(s);//注意 to_container
```

generic_ptr_list<T>内部使用 ptr_vector 来存储元素，因此它可以很容易地搭配序列指针容器使用，对于集合指针容器则需要使用成员函数 to_container()进行转换（它也可以解决部分编译器的兼容问题），但对于映射指针容器就无能为力了。

令人遗憾的是 assign 库没有提供 ptr_map_list_of<T>()这样的函数，我们不能对一个映射指针容器直接初始化。

8.14 使用算法

ptr_container 库提供的指针容器都符合标准容器的要求，因此我们也可以在这些指针容器上应用算法。但因为指针容器存储元素的特殊性，不是所有的标准算法都适用，接下来我们将进行详细的讨论。

8.14.1 标准算法

C++标准库提供了上百种算法，分为不变算法、修改算法、变序算法（含排序算法）、移除算法等类别，因为指针容器的迭代器屏蔽了内部的 void*指针，提供了间接访问接口，所以大部分算法都可以搭配指针容器工作。

对于一些比较重要的算法，指针容器提供了成员函数版本，它们通常能够比标准算法提供更高的运行效率，这些内部算法将在 8.14.2 节和 8.14.3 节进行介绍。

不变算法

不变算法基本上都可以安全地应用于指针容器，下面的代码简单示范了 count、find、equal 等算法的使用，与标准容器无任何差异：

```
ptr_deque<int> dq;                          //双端队列指针容器
ptr_push_back(dq)(1)(2)(10)(10)(9);         //顺序插入元素

//计算元素的个数，输出 2
```

```
cout << std::count(dq.begin(), dq.end(), 10);

//计算满足条件的元素个数,使用了 lambda 表达式,输出 3
cout << std::count_if(dq.begin(), dq.end(),
                [](int x){ return x > 8;});

//获取最小和最大的元素，输出 1 和 10
cout << *std::min_element(dq.begin(), dq.end());
cout << *std::max_element(dq.begin(), dq.end());

//查找元素，输出 2
cout << *std::find(dq.begin(), dq.end(), 2);

//对容器内的元素求和，输出 32
cout << std::accumulate(dq.begin(), dq.end(), 0);

//克隆构造另一个指针容器
ptr_list<int> lt(dq.begin(), dq.end());

//比较两个容器是否相等
assert(std::equal(dq.begin(), dq.end(), lt.begin()));
```

for_each 算法也可以用于指针容器：

```
ptr_vector<item> vec;                         //使用 8.5.3 节定义的类
...                                           //添加元素
std::for_each(vec.begin(), vec.end(),
  mem_fn(&item::print));                      //使用 mem_fn 调用成员函数，参见 7.2 节
```

修改算法

部分修改算法也可以应用于指针容器，但需要小心它们的变动语义，访问元素时要求元素必须是存在的而且可赋值，例如：

```
ptr_deque<int> dq;                                    //双端队列指针容器
ptr_push_back(dq)(1)(2)(10)(10)(9);                   //顺序插入元素

std::transform(dq.begin(), dq.end(), dq.begin(),      //转换算法
            [](int x){ return x+3;});                 //把所有元素加 3
assert(dq.front() == 4);

std::replace(dq.begin(), dq.end(), 13, 20);           //替换算法
```

```
assert(dq[2] == 20);

std::fill(dq.begin(), dq.end(), 99);                //填充算法
assert(dq.back() == 99);

ptr_vector<int> vec;                                //向量指针容器
ptr_push_back(vec)(1)(2)(3);                        //添加3个元素

std::copy(dq.begin(), boost::next(dq.begin(), 3),//拷贝算法
          vec.begin());
assert(vec[1] == 99);
```

下面的代码会产生运行时异常：

```
ptr_vector<int> vec(10);                            //保留10个元素的空间
std::copy(dq.begin(), dq.end(), vec.begin());       //抛出异常
```

这是因为 ptr_vector 的构造函数不同于 std::vector 的构造函数，它仅仅是保留了空间，内部并没有真正分配保存元素，而 copy 算法使用的是覆盖语义，向未分配空间写入会产生错误。解决办法可以如前面代码那样预先添加元素，或者使用插入迭代器（参见 8.15.3 节）：

```
std::copy(dq.begin(),dq.end(),
  ptr_container::ptr_back_inserter(vec));           //可以正常工作
```

变序算法和排序算法

变序算法和排序算法使用的是赋值和交换，因此要求元素必须是可赋值的，但因为指针容器中存储的是指针，因而效率没有内置的直接操作指针同名算法那么高效：

```
ptr_deque<int> dq;
ptr_push_back(dq)(20)(1)(2)(10)(9)(10)(100);       //顺序插入元素

//逆序算法，100,10,9,10,2,1,20
std::reverse(dq.begin(), dq.end());

//稳定排序，1,2,9,10,10,20,100
std::stable_sort(dq.begin(), dq.end());

//删除重复元素，搭配 erase，1,2,9,10,20,100
dq.erase(std::unique(dq.begin(), dq.end()), dq.end());
```

```
//删除元素，搭配 erase，1,2,9,10,100
dq.erase(std::remove(dq.begin(), dq.end(), 20), dq.end());

//随机打乱数据
std::random_shuffle(dq.begin(), dq.end());

//对前 3 个位置进行部分排序，1,2,9,100,10（后两个数字随机）
std::partial_sort(dq.begin(), boost::next(dq.begin(), 3), dq.end());
```

已序区间算法

只要不涉及元素的变动，已序区间算法就是不变算法，因而完全可以应用于指针容器：

```
ptr_deque<int> dq;
ptr_push_back(dq)(1)(2)(9)(10)(20);                     //添加已序元素

assert(std::binary_search(dq.begin(), dq.end(), 9)); //二分查找
cout << *std::lower_bound(dq.begin(), dq.end(), 3);  //下界
cout << *std::upper_bound(dq.begin(), dq.end(), 10); //上界

ptr_vector<int> vec;
ptr_push_back(vec) (2)(9)(10);                           //添加已序元素
assert(std::includes(dq.begin(), dq.end(),               //子集判定
  vec.begin(), vec.end()));
```

已序区间算法中的 merge()、set_union()等算法因为涉及元素的变动，用于指针容器时不够方便，因此，需要搭配插入迭代器（参见 8.15.3 节），通常使用成员函数会更好：

```
ptr_deque<int> dq;                                       //同前
ptr_vector<int> vec;                                     //同前
ptr_list<int> lt;                                        //一个空的双向链表指针容器

std::set_union(dq.begin(), dq.end(),                     //计算并集
  vec.begin(), vec.end(),
  ptr_container::ptr_back_inserter(lt));

lt.clear();

std::set_intersection(dq.begin(), dq.end(),              //计算交集
  vec.begin(), vec.end(),
```

```
ptr_container::ptr_back_inserter(lt));
```

8.14.2 序列指针容器的算法

ptr_sequence_adapter 为所有序列指针容器提供了下列四个内部算法。

- sort()　　: 快速排序（使用 std::sort）。
- unique()　: 删除相邻的重复元素。
- erase_if() : 删除满足条件的元素（使用 std::remove_if）。
- merge()　 : 合并两个已序区间的元素（使用 std::inplace_merge）。

这些算法都有类似的形式，可以操作整个容器，也可以操作一个指定的区间，比较准则也可以自定义。

sort

sort()算法对应于标准算法 std::sort()，用来对序列指针容器排序，它有以下四种形式：

```
void sort();
void sort( iterator first, iterator last );
void sort( Compare comp );
void sort( iterator begin, iterator end, Compare comp );
```

sort()算法与标准排序算法 std::sort()很像，它可以对整个容器排序，也可以指定一个区间（但 ptr_list 不能指定区间）。示范 sort()算法的代码如下：

```
ptr_deque<int> dq;
ptr_push_back(dq)(100)(1)(2)(10)(9);           //添加元素

//对前 3 个元素排序，1,2,100,10,9
dq.sort(dq.begin(), boost::next(dq.begin(), 3));

ptr_list<int> lt(dq.begin(), dq.end());        //克隆构造

//使用大于比较准则排序，100,10,9,2,1
lt.sort(std::greater<int>());
```

unique

unique()算法对应于标准算法 std::unique()，可以移除连续的重复元素，它也有以下四种形式：

```
void unique();
void unique( iterator first, iterator last );
void unique( Compare comp );
void unique( iterator begin, iterator end, Compare comp );
```

unique()算法操作的不一定是已序区间，当然，如果排序后再执行 unique()则必定会删除所有重复的元素。

示范 unique()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back(vec)(100)(1)(2)(2)(2)(10)(9)(7)(9);          //添加元素

//移除重复的元素，100,1,2,10,9,7,9
vec.unique();
```

erase_if

erase_if()算法相当于标准算法 std::remove_if()搭配 erase()函数，可以删除某些特定的元素，它必须指定比较谓词。

erase_if()算法只有以下两种形式：

```
void erase_if( Pred pred );
void erase_if( iterator begin, iterator end, Pred pred )。
```

示范 erase_if()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back<int>(vec)(100)(1)(2)(2)(2)(10)(9)(7)(9);   //添加元素

//删除所有等于 2 的元素，使用 lambda 表达式定义谓词，结果是 100,1,10,9,7,9
vec.erase_if([](int x){ return x==2;});

//删除前 4 个元素中大于 1 的元素，结果是 1,7,9
vec.erase_if(vec.begin(), boost::next(vec.begin(), 4),
```

```
        [](int x){ return x > 1;});                    //lambda 表达式
```

merge

merge()是一个已序区间算法，要求参与合并的两个容器都已经排序，最后得到一个已序的容器，容器 from 中的元素被转移到目标容器内。

merge()算法的声明有如下四种形式：

```
void merge( ptr_sequence_adapter& r );
void merge( ptr_sequence_adapter& r, BinPred comp );
void merge( iterator first, iterator last, ptr_sequence_adapter& from );
void merge( iterator first, iterator last, ptr_sequence_adapter& from,
            BinPred comp );
```

示范 merge()算法的代码如下：

```
ptr_vector<int> vec;
ptr_push_back<int>(vec)(1)(2)(7)(9)(10);

ptr_vector<int> v2;
ptr_push_back(v2)(3)(5)(100);

v2.merge(vec);                              //合并两个容器，1,2,3,5,7,9,10,100
assert(vec.empty());                        //vec 里的指针被转移
```

8.14.3 关联指针容器的算法

关联指针容器分为有序和无序两种，因而支持的算法也不相同。

所有关联指针容器都提供如下三个算法。

- find(k)　　　　　：查找键值 k，返回迭代器（查找到的位置）。
- count(k)　　　　 ：计算键值 k 的数量。
- equal_range(k) ：返回一个迭代器区间，里面的所有元素键值都是 k。

有序关联指针容器（set）额外提供下面两个算法。

- lower_bound(k) ：大于等于 k 的元素“下界”，返回第一个满足>=k 的迭代器。
- upper_bound(k) ：大于 k 的元素“上界”，返回第一个满足>k 的迭代器。

对于有序关联指针容器来说，equal_range(k) 相当于 (lower_bound(k), upper_bound(k))。

find 和 count

find()和 count()算法相当简单，它们相当于标准映射容器提供的同名函数，声明如下：

```
iterator          find( const key_type& x );
size_type         count( const key_type& x ) const;
```

find()和 count()算法用起来也很容易，例如：

```
ptr_unordered_set<int> us;                       //无序单键集合容器
ptr_insert(us)(3)(5)(7)(13);

assert(us.find(5) != us.end());                  //查找元素
assert(us.find(10) == us.end());
assert(us.count(10) == 0);                       //计算数量

ptr_multimap<int, int> mm;                       //有序多键映射容器
ptr_map_insert(mm)(1, 1)(1, 5)(2, 2)(3, 4);

assert(mm.count(1) == 2);                        //计算数量
assert(*mm.find(3)->second == 4);                //查找元素
```

equal_range

equal_range()返回一个迭代器区间，主要用于允许重复的关联指针容器，可以一次性获取所有键值等于 k 的元素，相当于 find()算法的泛化。

equal_range()算法的声明如下：

```
iterator_range<iterator>        equal_range( const key_type& x );
```

equal_range()返回一个 iterator_range 对象，它是一个具有类似容器接口的迭代器区间，比简单的 std::pair<iterator>功能强很多（参见 6.5 节）。

示范 equal_range()算法的代码如下：

```
ptr_multiset<int> us;                              //有序多键集合容器
ptr_insert(us)(3)(5)(7)(13)(3);

assert(us.count(3) ==2);

auto r1 = us.equal_range(3);                       //获取迭代器区间
assert(!r1.empty() );                              //可以判断区间是否为空
assert(r1.front() == 3);                           //具有类似容器的接口

ptr_unordered_multimap<int, int> mm;               //无序多键映射容器
ptr_map_insert(mm)(1, 1)(1, 5)(2, 2)(3, 4);

auto r2 = mm.equal_range(1);                       //获得迭代器区间
for (auto p = r2.begin(); p != r2.end(); ++p)
{
  cout << *p->second << ",";                       //输出1，5
}
```

lower_bound 和 upper_bound

lower_bound()和upper_bound()算法只能用于有序映射容器，它们的声明如下：

```
iterator          lower_bound( const key_type& x );
iterator          upper_bound( const key_type& x ) ;
```

它们的用法比较简单，下面的代码示范了它们的用法：

```
ptr_set<int> us;
ptr_insert(us)(3)(5)(13)(7)(2);                    //集合容器会自动排序

assert(*us.lower_bound(4) == 5);
assert(*us.upper_bound(4) == 5);
assert(*us.upper_bound(5) == 7);
```

8.15 其他议题

指针容器是一个比较庞大的库，在本节中，我们将讨论关于指针容器的一些其他议题，但并未囊括所有相关的内容。

8.15.1 异常

ptr_container 库是强异常安全的，如果出现访问空指针或者不存在的键值等情况会以抛出异常的方式通知用户。

ptr_container 库提供了三个异常类，它们都是 std::exception 的子类。

- bad_ptr_container_operation ：最通用的指针容器异常，表示发生的操作错误。
- bad_index ：序列指针容器使用整数索引时出错。
- bad_pointer ：不允许使用空指针时使用空指针出错。

在 ptr_container 库缺省情况下直接使用 throw 关键字将会抛出异常，但如果我们定义了宏 BOOST_NO_EXCEPTIONS 或者 BOOST_PTR_CONTAINER_NO_EXCEPTIONS，那么它将转化为断言 BOOST_ASSERT 从而完全禁用异常。

禁用异常适用于不允许使用异常的情形，同时也可能带来少量的性能提升，但除非有必要不建议禁用异常，因为异常已经成为了 C++的一部分，禁用它不会带来太多的好处。

8.15.2 间接函数对象

为了更方便地操作指针或智能指针所指的对象，ptr_container 库在头文件<boost/ptr_container/indirect_fun.hpp>中提供了两个函数对象适配器 indirect_fun 和 void_ptr_indirect_fun。它们可以把一个单参或者双参函数对象适配成可以直接操作指针的函数对象，工厂函数 make_indirect_fun()和 make_void_ptr_indirect_fun()也可以直接生产出包装后的函数对象。

使用这两个函数对象，我们可以很容易地操作指针所指的对象，例如：

```
typedef shared_ptr<string> ptr_string;              //智能指针

ptr_string s1(new string("chrono"));
ptr_string s2(new string("trigger"));

assert(indirect_fun<not_equal_to<string> >()(s1, s2));
cout << indirect_fun<std::plus<string> >()(s1, s2) << endl;
```

```
void* p1 = s1.get();                                    //void*指针
void* p2 = s2.get();

assert((void_ptr_indirect_fun<less<string>, string >()(p1, p2)));
```

8.15.3　插入迭代器

C++标准库为标准容器提供了三种插入迭代器，可以把容器适配成迭代器之后，再将其应用于算法，例如：

```
vector<int> v1, v2;
...                                                     //某些赋值操作
std::copy(v1.begin(), v1.end(),                         //copy 算法
  std::back_inserter(v2));                              //使用末端插入迭代器
```

ptr_container 库在头文件<boost/ptr_container/ptr_inserter.hpp>里提供了等价的三种指针插入迭代器和辅助函数，它们位于名字空间 boost::ptr_container。

- ptr_back_inserter(cont)　　：使用 push_back()克隆指针然后插入。
- ptr_front_inserter(cont)　　：使用 push_front()克隆指针然后插入。
- ptr_inserter(cont, before)：使用 insert()克隆指针然后插入。

这三种指针插入迭代器的用法与标准库的插入迭代器用法完全相同。

8.15.4　使用视图分配器

之前我们介绍的所有指针容器都使用的是 heap_clone_allocator（参见 5.1.4 节），它是 ptr_container 库缺省使用的容器。在本节中，我们将简单了解一下 view_clone_allocator 的用法。

view_clone_allocator 并不真正管理内存，而是提供一个只读的“指针视图”，因此我们可以使用它创建一个“视图容器”，它可以安全地用另一个容器的视角去观察原容器，不会造成任何影响。

示范 view_clone_allocator 用法的代码如下：

```
ptr_vector<int> v = ptr_list_of<int>(100)(1)(10) (2);    //向量指针容器

typedef ptr_set<int,std::less<int>,
```

```
    boost::view_clone_allocator> set_view_t;         //定义有序集合指针容器视图
set_view_t    view(v.begin(), v.end());                  //创建视图

for(auto& i : view)                                      //for 遍历
{     cout << i << ",";}
```

这段代码先创建了一个向量指针容器，然后用 view_clone_allocator 在其上建立了一个有序集合指针容器视图，这样就可以用有序集合的方式来只读地访问向量中的元素。

代码的运行结果如下：

```
1,2,10,100,                                   //有序集合重新整理了元素的顺序
```

8.15.5 可克隆性的再讨论

可克隆性（cloneable）是 ptr_container 库里的一个很重要的概念，相当于标准容器对元素可拷贝的要求，但它不是指针容器所必须的，因为指针容器是泛型的，如果对元素的操作不涉及克隆（克隆构造、克隆赋值等）就不会使用克隆分配器，也就不涉及可克隆概念。

8.5.3 节已经示范了不支持克隆概念类用于指针容器的情形，下面我们再来看一些代码：

```
typedef ptr_vector<item> ptr_vec;              //使用之前的 item 类定义
ptr_vec vec;

using namespace boost::assign;
ptr_push_back<television>(vec)();
ptr_push_back<computer>(vec)();

ptr_vec v2;
v2.transfer(v2.begin(), vec.begin(), vec);      //可以使用转移所有权操作
```

但下面的代码由于使用了克隆操作所以无法通过编译：

```
ptr_vec v2(vec);                                //克隆构造
v2 = vec.clone();                               //调用克隆方法
```

另外，在有些情况下，代码里虽然没有明显的容器克隆，但却存在隐含的克隆操作，这时也会导致编译失败：

```
typedef std::map<int, ptr_vec> wrong_map;       //标准容器要求拷贝语义
wrong_map m;
m[1];                                           //operator[]发生克隆，编译错误
```

总的来说，由于指针容器仅存储指针，所以大大降低了对元素的要求，虽然有的操作需要使用克隆，但因为存在 new_clone()和 delete_clone()函数的缺省实现，几乎无须任何多余的工作就可以使用 ptr_container 库的全部功能。不过对于不可拷贝的类型来说，必须像在标准容器里使用它们时一样谨慎。

8.16 总结

本章讨论了 Boost 库中的指针容器，它专门用于容纳指针元素，提供了与标准容器等价的 ptr_vector、ptr_set 等各种容器，而且是异常安全的，保证没有内存泄漏。

ptr_container 库里的指针容器基本与标准容器对应，接口和用法也非常相近，但它的根本特点是容器里容纳的是指针而不是元素的拷贝，因此存在许多与标准容器不同的地方。

指针容器对元素的要求非常低，不需要元素是可拷贝或可赋值的，因此几乎可以容纳任意类型的元素。指针容器也可以存储多态对象的指针，成为一个多态容器，这在很多时候都是非常有用的。

谈到指针就离不开空指针，虽然指针容器允许使用包装类 nullable<T>来容纳空指针，但这种做法很不安全，也很容易出错，应该尽量采用空对象模式来代替空指针的使用。

克隆是指针容器的一个特有概念，它类似于标准容器中的内存分配器概念，可以在需要的时候创建指针所指对象的拷贝。ptr_container 库提供缺省的克隆分配器实现，但这不是必须的，很多指针容器的操作无须克隆概念也可以工作。

ptr_container 库提供了数量众多的指针容器，因此我们必须对指针容器的优缺点做到心中有数，充分发挥它们的优点同时避免它们的缺点。在使用时首先要决定是选择标准容器还是指针容器，标准容器容纳智能指针的用法通用灵活，而指针容器有指针不可共享的限制，但用起来更方便。第二个决定是选择何种指针容器，由于指针容器与标准容器基本等价，因此可以参照标准容器的使用策略，最常用的是向量指针容器 ptr_vector。

ptr_container 库内容十分丰富，可以说与标准库不相上下，读者可在阅读完本章后进一步参考 Boost 文档和源码进行深入研究。

第9章

侵入式容器

在本章中，我们将看到另一类新式容器：侵入式容器。

“侵入式容器”名字中的“侵入”一词容易令人产生不愉快的联想，但它并没有任何恶意。侵入式容器也是一类用于容纳元素的容器，但元素必须做出一些代码上的适度修改才能被容纳，因此，侵入式容器看起来像是“侵犯/闯入”了元素的内部实现。而实际上，我们对侵入式容器并不陌生，在数据结构课程中实现的链表、二叉树等数据结构都属于侵入式容器的范畴。

与侵入式容器相对应的是非侵入式容器。标准容器和第 8 章介绍的指针容器都属于非侵入式容器，这类容器不要求对容纳的元素做任何修改即可容纳，较侵入式容器的实现手法要温和很多。因为非侵入式容器用起来简单方便，自 C++标准库出现后非侵入式容器逐渐大行其道，侵入式容器却日薄西山。

但侵入式容器也有自己的优点，它没有非侵入式容器的拷贝、克隆等要求，也可以保存抽象类，对内存管理的要求很低，而且允许定制数据结构，所以通常可以提供更好的性能。Boost 使用库 intrusive 重新引入了侵入式容器，而且具有近似标准容器的接口，大大降低了应用的难度，值得我们去学习使用。

9.1 概述

在这一节中我们将展示 intrusive 库的大致轮廓，了解它的一些基本概念。为了让读者对侵入式容器有一个比较明晰的印象，我们先来回顾一下数据结构的知识。

9.1.1 手工实现链表

读者应该对数据结构不会陌生，线性表、链表、二叉树、堆等数据结构是计算机科学的基础，构建这些数据结构是一个程序员应该具备的基本素质。遗憾的是自从完整精致的标准库横空出世，这一基本功就渐渐被大多数人荒废了。

下面我们来实现一个单向链表结构，它是一种最简单的侵入式容器。

首先，我们定义一个简单的节点类 point：

```
class point: boost::noncopyable              //简单的节点类，不可拷贝和赋值
{
public:
  int x,y;                                   //节点的“有效载荷”

  //下面的代码是构造链表所需的“附件”
  typedef point*  node_ptr;                  //指针类型定义
  node_ptr        next;                      //后继指针

  point(int a = 0, int b = 0):               //构造函数
      x(a), y(b), next(nullptr){}            //后继初始化为空指针，即无后继

  node_ptr get_next()                        //获得后继节点
  {   return next;}

  void set_next(node_ptr p)                  //设置后继节点
  {   next = p;}
};
```

point 类的真正作用是保存坐标值，但为了实现链表必须增加一个额外的存储空间：后继指针变量 next。为了规范指针的使用，我们定义了两个 get/set 成员函数，这种间接层比直接操作指针要好得多。[1]

下面的代码示范了节点类是如何连接成链表并使用的：

```
point p1, p2(2,2), p3(3,3);                  //三个节点对象，未连接

p1.set_next(&p2);                            //p1 连接 p2，链表形如 p1->p2
p2.set_next(&p3);                            //p2 连接 p3，链表形如 p1->p2->p3
```

① 后面我们会看到 intrusive 库同样使用了这样的实现手法。

```
for (point::node_ptr p = &p1;                   //指向头节点
    p != nullptr;                               //循环终止条件
    p = p->get_next())                          //指针前进到下一个节点
{
    cout << p->x << "-" << p->y << " ";
}

p1.set_next(&p3);                       //从链表中移除 p2，链表形如 p1->p3
```

这个简单的例子展示了侵入式容器的一些重要特性。

（1）元素除了它自身的必备功能外还必须要增加一些额外的能力（这里是指向后继节点的指针）才能被纳入侵入式容器（链表）。

（2）侵入式容器不负责内存分配，元素的创建是容器之外的事情，与它无关。

（3）侵入式容器并不真正的“容纳”对象，元素仍然散落在内存中的各个位置，仅仅是使用指针以某种算法（这里是单向顺序算法）把它们连接起来便于访问而已，在某种程度上把它称为“链接视图”或许会更恰当。

（4）侵入式容器的插入删除等操作也只是操纵链接的指针，调整元素的链接顺序，不涉及内存的分配管理。

9.1.2 intrusive 库介绍

intrusive 库位于名字空间 boost::intrusive，由数个不同的头文件组成，实现了许多非常有用的侵入式容器，它们都位于目录<boost/intrusive/>下，使用具体的容器时包含所需头文件即可。

因为侵入式容器都是使用指针链接实现的，所以也可以称为“链式容器”——不存在与 std::vector 等价的基于连续内存空间实现的容器。

intrusive 库提供的侵入式容器都具有与标准容器类似的接口，主要有以下几种。

- slist ：单向链表。
- list ：类似 std::list 的双向链表，是最常用的侵入式容器。
- set/rbtree ：基于红黑树，类似 std::set 的关联容器。
- avl_set/avltree ：基于 AVL 树，类似 std::set 的关联容器。

- splay_set/splaytree ：基于 splay 树，类似 std::set 的关联容器。
- sg_set/sgtree ：基于 scapegoat 树，类似 std::set 的关联容器。
- treap_set/treap ：基于堆二叉树，类似 std::set 的关联容器。
- unordered_set ：无序关联容器。

根据侵入的程度这些容器又可分为纯侵入式容器（intrusive container）和半侵入式容器（semi-intrusive container）。

intrusive 库提供的大多数容器都属于纯侵入式容器（intrusive container），仅仅调整节点中的链接指针，并没有“容纳”任何东西。无序关联容器 unordered_set 属于半侵入式容器，这是因为它需要一个额外的内存空间来维护散列容器所需的负载因子（load factor），不是完全的侵入。

侵入式容器的一个重要特点是它不负责管理元素的生命周期，仅仅是调整指针的链接，因此没有对象拷贝的运行开销，元素的创建工作被外部化，这与标准容器（使用内存分配器）和指针容器（使用克隆分配器）是明显不同的。带来的好处是最小化了内存使用，提高了运行效率。

但这同时也是侵入式容器的缺点，因为内存管理的工作终归是要有人做的，现在就交给了用户——元素的创建与删除对于侵入式容器来说是一个重要的工作，有些时候可以使用内存池或者第 8 章的指针容器来简化。

更进一步的推论是，侵入式容器与容纳的元素两者的生命周期是不一致的，有可能因为元素被销毁而导致访问失败，所以元素的生命周期应该比侵入式容器要长。

9.2 入门示例

使用侵入式容器必须要修改自有类的定义，为此 intrusive 库提供了挂钩（hook）这一方便的工具类，它包含了一些必要的数据，可以把它近似地理解为链接指针的聚合。挂钩可以分为基类挂钩（base hook）和成员挂钩（member hook）两类，可以以基类或者成员的方式嵌入到自有类中使用。

下面我们使用单链表来示范侵入式容器的基本用法，其中涉及的概念参见 9.3 节。

9.2.1 使用基类挂钩

单链表侵入式容器使用的基类挂钩是 slist_base_hook，并不需要我们做太多的工作，只

要从它继承就可以了。point 类可改写成如下的形式：

```
#include <boost/intrusive/slist.hpp>           //单链表侵入式容器
using namespace boost::intrusive;              //侵入式容器的名字空间

class point: public slist_base_hook<>          //使用基类挂钩，缺省配置
{
public:
  int x,y;                                     //无须自定义链接指针，已由挂钩实现
  point(int a = 0, int b = 0):
      x(a), y(b){}
};
```

单链表侵入式容器 slist 具有类似于标准容器的接口，可以这样使用：

```
point p1, p2(2,2), p3(3,3);             //三个节点对象，未连接

slist<point> sl;                        //声明一个单链表侵入式容器，可容纳 point

sl.push_front(p1);                      //缺省情况下不能使用 push_back()
sl.push_front(p2);
sl.push_front(p3);                      //链表形如 p3->p2->p1

assert(sl.size() == 3);
sl.reverse();                           //提供内置的逆序算法，链表形如 p1->p2->p3

for(auto& p : sl)                       //可以使用新式 for 循环
{
  cout << p.x << "-" << p.y << " ";
}

sl.erase(boost::next(sl.begin()));      //删除链表中的第二个节点
```

slist 是 intrusive 库提供的一个单链表容器，缺省情况下只能使用 push_front()添加元素，如果在 slist 的模板参数中增加一个配置选项 cache_last<true>，那么它也可以使用 push_back()：

```
slist<point, cache_last<true> > sl;     //使用元数据选项配置
sl.push_back(p1);                       //可以使用 push_back()
```

9.2.2 使用成员挂钩

基类挂钩是使用侵入式容器最简单的方式，但有的时候我们可能不想使用基类挂钩，因为这种继承关系可能不是我们想要的。这时可以使用成员挂钩，把挂钩作为自有类的一个 public 成员：

```
class point                                          //不使用基类挂钩
{
public:
  ...                                                //同前
  slist_member_hook<> m_hook;                        //成员挂钩，缺省配置
};
```

侵入式容器在容纳使用成员挂钩的类时要多做一些工作，需要用一个元函数 member_hook 作为配置选项，告诉容器成员挂钩的类型和挂钩变量名：[①]

```
  slist<point,                                       //元素类型
      member_hook<point,                             //成员挂钩选项
          slist_member_hook<>,&point::m_hook>,       //成员挂钩变量
      > sl;
```

使用成员挂钩的完整代码如下，这里我们还使用了指针容器来动态创建对象：

```
ptr_vector<point> vec =                              //指针容器
  ptr_list_of<point>()(2,2)(3,3);                    //使用 assign 库初始化

typedef member_hook<point, slist_member_hook<>,      //使用成员挂钩 typedef
  &point::m_hook>     member_option;                 //简化类型定义

slist<point, member_option> sl;                      //单链表侵入式容器

BOOST_REVERSE_FOREACH(auto& p, vec)                  //逆序遍历指针容器
{
  sl.push_front(p);                                  //使用 push_front()
}
assert(sl.size() == 3);

for(point& p : sl)                                   //遍历侵入式容器
{
  cout << p.x << "-" << p.y << " ";
}
```

在这段代码中，我们需要注意侵入式容器与元素的生命周期问题，指针容器及里面的元素必须先于侵入式容器的创建，如果把两者的声明顺序调换一下，像这样：

```
  slist<...> sl;                        //先声明侵入式容器
```

① 实际上这相当于我们手工处理了成员变量指针类型，分解出类的类型和成员变量的类型，因为编译器不具备从成员变量指针中推导出这些类型的功能，算是个“无奈之举”吧。

```
ptr_vector<point> vec = ...;                //后声明指针容器并插入元素
```

那么在函数结束时指针容器 vec 和里面的元素会先被销毁，当侵入式容器 slist 销毁时会因为访问不存在的元素而抛出异常。

如果要避免这个错误，那么可以在程序结束前调用侵入式容器的成员函数 clear()，提前“清空”容器。实际上它并不删除元素，仅是把元素间的链接关系断开而已：

```
sl.clear();                                 //程序结束前调用，此时元素还是有效的
```

或者我们再改变一下挂钩和侵入式容器的配置选项，允许元素在析构时自动从容器中断开连接：

```
class point
{
public:
  slist_member_hook<link_mode<auto_unlink>> m_hook; //使用自动断开连接的选项
};

typedef member_hook<point,
      slist_member_hook<link_mode<auto_unlink> >,    //挂钩类型
      &point::m_hook> member_option;

//自动断开连接功能需禁用常量时间的size()功能
slist<point, member_option, constant_time_size<false> > sl;
```

9.3 基本概念

intrusive 库抽象了侵入式容器的实现手法，使用了大量的元编程技术，本节将简要介绍 intrusive 库中使用的这些基本概念。

9.3.1 节点

节点（node）概念抽象了手工实现链表时的链接指针和其他信息，把它们封装为一个类：单链表提供一个指针（后继），双向链表提供两个指针（前驱和后继），树结构则提供三个指针（父指针和左右指针）以及颜色、平衡等其他信息。

有了节点概念，用户就可以直接以继承或者成员的方式复用它而无须自行编写侵入代码。例

如，单链表 slist 的节点代码如下：[1]

```
template<class VoidPointer>
struct slist_node
{
   typedef typename pointer_rebind<                     //元函数计算
      VoidPointer, slist_node>::type node_ptr;          //单链表节点指针类型
   node_ptr                           next_;            //单链表节点指针
};
```

因为 pointer_rebind 模仿 C++11/14 的元函数 pointer_traits（参见 4.7.7 节）来定义节点指针类型，所以节点不仅可以使用原始指针，也能够使用智能指针。[2]

9.3.2 节点特征

节点特征（node traits）是一个 traits 类，它封装了操作节点的基本方法，对外提供了一致的抽象接口。不同的节点特征有不同的接口，但通常都提供节点类型、节点指针类型、取节点的前驱后继（静态成员函数）等操作。

例如，单链表 slist 的节点特征代码摘要如下：

```
template<class VoidPointer>
struct slist_node_traits
{
   typedef some_define    node;                            //节点类型
   typedef some_define    node_ptr;                        //节点指针类型
   typedef some_define    const_node_ptr;                  //节点常指针类型

   static node_ptr get_next(const_node_ptr& n);            //取后继节点
   static void     set_next(node_ptr n, node_ptr next);    //设置后继节点
};
```

读者可以对比一下 9.1.1 节的代码，比较两者之间的异同。

9.3.3 节点算法

节点算法（node algorithms）抽象了链式数据结构的算法，使用节点特征类来操作节点，以静态成员函数的形式提供链表的一些基本操作，如初始化、连接节点、断开连接节点等，来实现某个特定数据结构——从简单的单链表到复杂的红黑树、AVL 树等。

---

① 侵入式容器的早期版本使用的元函数是 boost::pointer_to_other。

② 不是所有智能指针都能被用于侵入式容器，特别是共享语义的智能指针是不可以的（如 shared_ptr）。

节点算法与配套的节点特征类必须是接口兼容的（静态多态），否则就会在编译时发生错误。

例如，单链表 slist 的循环节点算法的部分代码如下：

```
template<class NodeTraits>                                          //使用节点特征类
class circular_slist_algorithms
{
public:
   typedef NodeTraits::node                  node;                  //节点类型
   typedef NodeTraits::node_ptr              node_ptr;              //节点指针类型
   typedef NodeTraits::const_node_ptr        const_node_ptr;        //节点常指针类型
   typedef NodeTraits                        node_traits;           //节点特征类

   static void                               init();                //初始化操作
   static bool                               inited();              //是否已经初始化
   static void                               unlink_after();        //断开连接
   static void                               link_after();          //连接
   static void                               init_header();         //初始化头节点
   static std::size_t                        count();               //计算节点的数量
   ...                                                              //其他操作
};
```

使用节点算法我们就可以避免直接操作节点，在更高的抽象层次上实现数据结构。circular_slist_algorithms 可这样使用：

```
typedef slist_node<void*>          node_t;     //节点类型，使用 void*来定义指针
typedef slist_node_traits<void*> node_traits_t;          //节点特征类
typedef circular_slist_algorithms<node_traits_t> algo;   //节点算法

node_t n1, n2;                                           //两个节点

algo::init_header(&n1);                                  //初始化头节点
assert(algo::count(&n1) == 1);                           //链表中有一个元素

algo::link_after(&n1, &n2);                              //连接 n1 和 n2
assert(algo::count(&n1) == 2);                           //链表中有两个元素

algo::unlink(&n1);                                       //断开 n1 的连接
assert(algo::count(&n2) == 1);                           //链表中有一个元素
```

9.3.4 值特征

值特征（value traits）是一个类似非侵入式容器中值类型 T 的概念，它把用户自有的值类型与节点、节点特征封装在一起，可以提供给节点算法使用。

值特征类通常具有下面的形式：

```
struct value_traits
{
    typedef some_define    node_traits;                     //节点特征类
    typedef some_define    value_type;                      //值类型
    typedef some_define    node_ptr;                        //节点指针类型
    typedef some_define    pointer;                         //指向值的指针类型

    static const link_mode_type link_mode = some_link_mode;   //链接策略

    static node_ptr        to_node_ptr(value_type &value);    //指针转换
    static pointer         to_value_ptr(node_ptr n);          //指针转换
};
```

值特征类中的一个重要常量是链接模式 link_mode，它是一个枚举值，用于确定侵入式容器的链接处理策略，有以下三个取值（即三种策略）。

- safe_link ：节点在插入删除时会检查指针，可以安全地插入，是最常用的一种策略。
- auto_unlink ：可以在节点对象析构时自动从容器中移除，虽然方便但安全性有所降低，只能用于非常量时间的容器。
- normal_link ：节点在进行插入删除操作时不做任何检查。

这三种链接模式的更详细解说见下一小节。

9.3.5 挂钩

挂钩（hook）是实现侵入式容器的核心概念，它包含节点、节点算法和其他一些信息，我们必须以继承或者成员的方式把挂钩加入自有类（侵入式修改），这样侵入式容器才能通过挂钩用算法操纵节点从而容纳元素。

类摘要

头文件<boost/intrusive/detail/generic_hook.hpp>定义了所有侵入式容器通用的挂钩 generic_hook，类摘要如下：

```
template
   < class NodeAlgorithms                        //节点算法
   , class Tag                                   //标记挂钩的标签
   , link_mode_type LinkMode                     //链接策略
   , base_hook_type BaseHookType                 //挂钩的类型
   >
class generic_hook
   : public detail::if_c<...>::type              //元计算节点类型
{
   typedef some_define             node_algorithms;
   typedef some_define             node;
   typedef some_define             node_ptr;

public:
   struct hooktags    //定义一些挂钩的基本属性，实际是由 hooktags_impl 元计算得到
   {
      static const link_mode_type link_mode = LinkMode;
      typedef Tag                 tag;
      typedef some_define         node_traits;
      static const bool           is_base_hook = some_define;
      static const bool           safemode_or_autounlink = some_define;
      static const int            type = some_define;
   };

   bool   is_linked() const;              //挂钩是否已经连接
   void   unlink();                       //断开挂钩的连接，要求链接模式为 auto_unlink
};
```

generic_hook 使用了模板元编程来计算类型，元函数 detail::if_c 用参数 Tag 决定挂钩的节点类型。如果是成员挂钩，那么直接使用节点算法里的节点类型 NodeAlgorithms::node；如果是基类挂钩，则计算得到一个包装类型 node_holder<...>。

无论作为成员还是基类，generic_hook 都是 NodeAlgorithms::node 的子类，含有节点和相应的算法定义，可以被侵入式容器使用。

generic_hook 有一个重要的成员函数 is_linked()，它可以被节点随时调用，用于检查节点是否已经被链入侵入式容器。

generic_hook 模板参数 LinkMode 的取值与值特征类中的定义相同，它决定了 generic_hook 在构造和析构时的行为。

- safe_link ：构造时初始化节点为未连接状态，析构时检查节点的连接状态，如果已连接（保存在某个侵入式容器中）则使用 BOOST_ASSERT 断言抛出异常。
- auto_unlink ：构造时初始化节点为未连接状态，析构时如果已连接则断开连接。
- normal_link ：构造、析构时无任何操作。

9.3.6 选项

选项(option)是 intrusive 库中用于配置、调整侵入式容器行为的一大族高阶元数据(参见 13.7 小节)，如 link_mode、optimize_size、cache_last 等。

选项元数据的定义在<boost/intrusive/options.hpp>里，其通常形式是：

```
template<class OptionName>
struct option_name                          //选项名
{
    template<class Base>
    struct pack : Base                      //继承
    {
        typedef OptionName option_name;     //类型定义
    };
};
```

元数据 option_name 内部的元函数是 pack，它会把参数 OptionName 定义为一个与元数据同名的类型（即 option_name）。使用 pack 而不是元编程中更常用的 apply 的原因很简单，因为这些高阶元数据要配合另一个元函数 pack_options 来执行元数据打包的操作。[①]

pack_options 支持多个选项元数据，由于使用了元编程，所以不必用固定的顺序指定参数，它的简化形式如下：

```
template<class Prev, class Next>
struct do_pack                    //子元函数，调用高阶元数据的 pack 元函数
```

① 当然，使用 mpl 中的惯例 apply 也是可以的，大概库作者认为使用 pack 的意义会更明确吧。

```
{
   typedef typename Next::template pack<Prev> type;
};

template< class DefaultOptions, class O1, class O2 >
struct pack_options
{
  typedef
  typename do_pack
    <  typename do_pack
       < DefaultOptions , O1 >::type              //对 O1 打包
    , O2                                          //对 O2 打包
    >::type type;
};
```

pack_options 连续调用元函数 do_pack 对所有选项逐个命名，中间又有元数据的继承，所以最后得到的类型就是一个含有 N 个具有小写同名类型定义的大类型，把选项打包到了一起。

intrusive 库定义了一个默认的选项集合 hook_defaults，包含了大部分常用的选项，出于效率的考虑它没有使用元计算：

```
struct hook_defaults
{
   typedef void* void_pointer;                           //使用 void*定义指针
   static const link_mode_type link_mode = safe_link; //安全链接模式
   typedef dft_tag tag;                                  //默认标签，基类挂钩
   static const bool optimize_size = false;              //不优化空间，优化时间
   static const bool store_hash = false;                 //无序容器专用，散列值存储在外部
   static const bool linear = false;                     //非线性化，使用循环链接
   static const bool optimize_multikey = false;          //无序容器不优化重复键值的元素
};
```

9.3.7 处置器

因为侵入式容器不做内存管理，不能销毁元素，所以 intrusive 库提出了处置器（Disposer）的概念来帮助侵入式容器“销毁”元素。

处置器是一个函数对象，它可以用某种策略处理元素指针，形如：

```
struct Disposer                         //处置器函数对象
```

```
{
   void operator()(T *to_dispose) //operator()接受指针
   {  ...}                          // 处置指针，通常是 delete to_dispose 来销毁对象
};
```

侵入式容器的 pop_back()、clear()、erase()等涉及容器内元素移除的操作都有两个版本。第一个版本是与标准容器相同的接口，只是简单地断开元素在容器中的链接，元素没有被真正销毁，用户必须在适当的时候管理这些移出容器的元素，这通常是一件很困难的事情（如果使用内存池或者指针容器则不会有这样的困扰）。第二个版本是带“_and_dispose”后缀的函数，它除了标准参数外还多出一个处置器参数，侵入式容器使用它来处置对象。

9.3.8 克隆

侵入式容器没有内存管理功能，因此是不可赋值和不可拷贝的，但与指针容器类似，它提供了克隆的概念，允许一个容器从另一个容器克隆元素。

所有的侵入式容器都提供一个 clone_from()成员函数，它实现克隆操作，基本形式是：

```
template <class Cloner, class Disposer>
void clone_from(const Cont &src, Cloner c, Disposer d);
```

两个模板参数分别是克隆器和处置器函数对象，容器先使用处置器 d 清除内部的所有元素，然后使用克隆器 c 从 src 逐个克隆元素到本容器，如果克隆过程中发生异常则调用处置器 d 清除已经克隆的元素。

克隆器是一个函数对象，它的形式是：

```
struct Cloner
{
   template<typename ValueType>
   ValueType* operator()(const ValueType& r);
};
```

克隆器的 operator()接受一个元素类型的常引用，然后以指针的形式返回它的克隆对象。

9.4 链表

intrusive 库提供两种链表式侵入式容器：slist 和 list。

我们已经在 9.2 节见到了 slist 的部分用法，它是单链表，只有一个指针的空间开销，但时间复杂度较高，因此这里介绍更常用的双向链表 list，它使用两个指针，类似于 std::list，其用法更灵活。

list 位于名字空间 boost::intrusive，需要包含头文件<boost/intrusive/list.hpp>。

9.4.1 节点和算法

list 使用的节点是 list_node，它提供了前驱和后继两个指针：

```
template<class VoidPointer>
struct list_node
{
   typedef typename pointer_rebind<
      VoidPointer, list_node>::type  node_ptr;           //指针类型定义
   node_ptr                          next_;              //前驱指针
   node_ptr                          prev_;              //后继指针
};
```

list_node 对应的节点特征类是 list_node_traits，封装对 list_node 的所有操作：

```
template<class VoidPointer>
struct list_node_traits
{
   typedef list_node<VoidPointer> node;
   typedef some_define            node_ptr;
   typedef some_define            const_node_ptr;

   static node_ptr                get_previous(const_node_ptr n);
   static void                    set_previous(node_ptr n, node_ptr prev);
   static node_ptr                get_next(const_node_ptr n);
   static void                    set_next(node_ptr n, node_ptr next);
};
```

list 使用的节点算法是 circular_list_algorithms，接口与 9.3.3 节的 circular_slist_algorithms 类似。

9.4.2 基类挂钩

list 使用的基类挂钩是 list_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3>
class list_base_hook
   : public make_list_base_hook<O1, O2, O3>::type    //工厂元函数
{
  void swap_nodes(list_base_hook &);
  bool is_linked() const;                            //挂钩是否已经连接
  void unlink();                       //断开挂钩的连接，要求链接模式为 auto_unlink
}
```

list_base_hook 派生自 generic_hook，实际上是工厂元函数 make_list_base_hook 的计算结果：

```
template<class O1 = none, class O2 = none, class O3 = none>
struct make_list_base_hook
{
   typedef typename pack_options             //选项元数据打包
     < hook_defaults,                        //使用缺省挂钩参数
     O1, O2, O3 >::type packed_options;

   typedef generic_hook                      //使用挂钩基类
   < circular_list_algorithms<...>           //使用 circular_list_algorithms
   , typename packed_options::tag            //标签
   , packed_options::link_mode               //链接模式
   , ListBaseHookId                          //链表基类挂钩枚举值
   > implementation_defined;

   typedef implementation_defined type;   //返回元函数的计算结果
};
```

list_base_hook 可以使用如下三个选项，因为使用了元数据打包，所以对顺序无要求。

- tag　　　　：一个语法层面的标签，用于区分不同类别的挂钩。不同的类可以使用相同的标签，但同一个类如果使用多个基类挂钩则标签不能相同，默认值是 default_tag。
- void_pointer：挂钩使用的指针类型，默认值是 void*。
- link_mode　：挂钩的链接模式，默认值是 safe_link。

一个完整定义的链表基类挂钩可以是这样：

```
list_base_hook<
        tag<struct some_tag>,                //使用一个"不完整类型"定义标签
        void_pointer<void*>,                 //指针通常都使用void*定义
        link_mode<safe_link> >               //链接模式使用safe_link
```

9.4.3 成员挂钩

list 使用的成员挂钩是 list_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3>
class list_member_hook
   : public make_list_member_hook                  //工厂元函数
     <O1, O2, O3>::type
{...};                                             //接口同list_base_hook
```

list_member_hook 是由另一个工厂元函数 make_list_member_hook 产生的，与 make_list_base_hook 仅有微小的不同。

```
template<class O1 = none, class O2 = none, class O3 = none>
struct make_list_member_hook
{
   ...                                             //省略相同的代码

   typedef generic_hook                            //使用挂钩基类
   < circular_list_algorithms <...>                //使用circular_list_algorithms
   , member_tag                                    //注意这里，成员标签
   , packed_options::link_mode                     //链接模式
   , NoBaseHookId                                  //成员（非基类）挂钩类型
   > implementation_defined;
};
```

list_member_hook 仅仅是在挂钩类别上与 list_base_hook 不同，模板参数和成员函数均相同。

9.4.4 类摘要

双向链表容器 list 的定义同样也使用了复杂的元编程计算，它的真正实现是模板类 list_impl。元函数 make_list 是根据模板参数计算值特征等配置选项的，但由于其元计算过程较复杂，因此本书只给出了 list 的接口定义：

```
template<class T, class O1, class O2, class O3 >
class list {
public:
  // 容器必需的若干类型定义
  typedef some_define          value_traits;       //各种内部类型定义
  typedef some_define          value_type;
  ...

  list();                                          //构造函数
  list(Iterator, Iterator);

  void        push_back(reference);                //双向链表的通用操作
  void        push_front(reference);
  void        pop_back();
  void        pop_front();
  ...         //其他迭代器、容量、删除、连接、排序、逆序、合并等操作，同标准链表容器

  void        shift_backwards(size_type = 1) ;     //左右移动算法
  void        shift_forward(size_type = 1) ;

  void        pop_back_and_dispose(Disposer) ;     //下面是使用处置器的操作
  void        pop_front_and_dispose(Disposer) ;    //简单起见忽略了模板参数列表

  iterator    erase_and_dispose(iterator, Disposer) ;
  iterator    erase_and_dispose(iterator, iterator, Disposer) ;

  void        clear_and_dispose(Disposer) ;

  void        dispose_and_assign(Disposer, Iterator, Iterator) ;

  void        remove_and_dispose(const_reference, Disposer) ;
  void        remove_and_dispose_if(Pred, Disposer) ;

  void        unique_and_dispose(Disposer) ;
  void        unique_and_dispose(BinaryPredicate, Disposer) ;

  void        clone_from(const list &, Cloner, Disposer) ;   //克隆

  iterator    iterator_to(reference) ;                       //从值获得迭代器
  static      iterator s_iterator_to(reference) ;

  //迭代器到容器的静态成员函数
  static list& container_from_end_iterator(iterator) ;
};
```

```
...                                          //各种比较操作符定义，如==、!=、<
```

list 有四个模板参数，第一个参数 T 是容器的元素类型，其他三个是配置选项，其中两个通常使用缺省值，无须变动。

- constant_time_size ：是否使用一个额外的变量保存容器的大小，这样可以在常量时间里获得容器的容量信息，默认是 true。如果挂钩的链接模式使用 auto_unlink，那么它必须被置为 false。
- size_type ：容器大小的类型，默认是 std::size_t。

下面的三个选项标明了容器的值特征，只能选择一个使用。

- base_hook ：使用基类挂钩，值特征被自动推导，是默认配置。
- member_hook ：使用成员挂钩，要在模板参数中明确元素类型等参数，值特征被自动推导。
- value_traits ：用户手工指定值特征。

值特征选项里的 member_hook 比较复杂，它的形式是：

```
member_hook<typename Parent,                        //元素类型
            typename MemberHook,                    //成员挂钩类型
            MemberHook Parent::* PtrToMember>       //成员变量访问指针
```

工厂元函数 make_list 可以使用同样的模板参数得到链表容器 list，它的编译速度更快，而且能够保证使用不同的选项顺序都能够创建出相同的链表类型。

9.4.5 基本用法

使用 list 容纳元素必须先使用基类挂钩 list_base_hook 或者成员挂钩 list_member_hook“侵入式”修改元素的定义，通常我们无须修改挂钩的选项。

在定义 list 时我们最好使用工厂元函数 make_list，必需的参数是容纳的元素类型。基类挂钩 base_hook 是被默认使用的，但如果挂钩没有使用缺省标签而是自定义标签，那么必须明确写出，如果元素使用了成员挂钩，那么就需要用 member_hook 指定挂钩的类型和访问方式。

list 完全模仿了 std::list 的接口，符合标准容器的定义，基本上 std::list 的所有操作都适用于 list，但需要注意 list 不支持拷贝构造和赋值。

为了示范 list 的用法，我们给 point 类增加比较操作符定义：

```
#include <boost/intrusive/list.hpp>
```

```
using namespace boost::intrusive;

class point: public list_base_hook<>                         //使用基类挂钩
{
public:
  ...//数据和构造函数定义同前
  friend bool operator==(const point& l, const point& r)     //相等定义
  {   return l.x == r.x && l.y == r.y;}
  friend bool operator<(const point& l, const point& r)      //小于定义
  {   return l.x < r.x;  }
};
```

示范 list 基本操作的代码如下：

```
using namespace boost::assign;                    //assign 名字空间
ptr_vector<point> vec =                           //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);              //使用 assign 库初始化

typedef make_list<point>::type list_t;            //使用工厂元函数，基类挂钩
list_t lt;                                        //声明链表侵入式容器
assert(lt.empty());                               //此时容器为空

lt.push_back(vec[2]);                             //向末端添加元素，链表是[2]
lt.push_front(vec[3]);                            //向前端添加元素，链表是[3，2]

assert(!lt.empty() && lt.size() == 2);            //获得容器的大小
assert(lt.front().x == 3);                        //访问前端元素
assert(lt.back().x == 2);                         //访问末端元素

lt.insert(boost::next(lt.begin()),                //在第二个位置执行插入操作
  vec.begin(), vec.begin() + 2);                  //插入一个区间内的所有元素
for(auto& p : lt)                                 //使用新式 for 遍历
{
  cout << p.x << ",";                             //输出3，0，1，2
}

lt.reverse();                                     //逆序链表，链表是[2，1，0，3]
lt.pop_front();                                   //弹出前端元素，链表是[1，0，3]
assert(lt.size() == 3);

//在末端前插入一个元素，链表是[1，0，4，3]
```

```
lt.insert(boost::prior(lt.end()), vec[4]);

lt.sort();                                      //排序，链表是[0，1，3，4]

//删除元素，链表是[0，1，4]
lt.erase(boost::prior(lt.end(), 2));            //注意prior()前进了两步
```

基类挂钩使用自定义标签时代码需要做少量的变动，因为这时无法自动推导出基类挂钩类型，必须使用 base_hook 选项明确地写出挂钩类型：

```
class point: public list_base_hook<tag<struct a_tag> >{};    //自定义标签
typedef make_list<point,
      base_hook<list_base_hook<tag<a_tag>>> >::type list_t;
```

成员挂钩的用法与基类挂钩没有太大的差别，只是必须在 list 的模板参数列表中配置 member_hook 选项，略显麻烦，例如：

```
class point                                           //注意，无继承
{
public:
  ...                                                 //同前
  list_member_hook m_hook;                            //成员挂钩，缺省配置
};
typedef make_list<point,
                member_hook<point,                    //成员挂钩选项
                list_member_hook<>, &point::m_hook>>::type list_t;
```

9.4.6 特有用法

list 拥有一些不同于 std::list 的特有用法，本小节将逐个详解这些功能。

左右移动

成员函数 shift_backwards()和 shift_forward()相当于 std::rotate 算法，令链表中的元素循环后移或前移，默认移动一个元素。例如：

```
using namespace boost::assign;                  //assign名字空间
ptr_vector<point> vec =                         //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);            //使用assign库初始化

typedef make_list<point>::type list_t;          //使用基类挂钩
```

```
list_t lt(vec.begin(), vec.end());                //区间元素构造
assert(lt.size() == 5);                           //5个元素，链表是[ 0, 1, 2, 3, 4]

lt.shift_backwards(2);                  //循环后移两个元素，链表是[ 3, 4, 0, 1, 2]
lt.shift_forward();                     //循环前移一个元素，链表是[ 4, 0, 1, 2, 3]
```

处置器相关操作

pop_back()、pop_front()、clear()、erase()、remove()、assign()和unique()等涉及容器内元素移除的操作，有对应的使用处置器的版本，缀词“dispose”的位置表示了处置器的使用时机，大部分函数都是操作完毕后使用处置器的operator()处理移除的元素，只有dispose_and_assign()是先使用处置器然后再赋值。

作为示例我们先定义一个简单的处置器函数对象，它输出处置的值然后删除它：

```
struct disposer                                  //一个简单的处置器
{
  void operator()(point* p)                      //操作指针
  {
      cout << "dispose:" << p->x << endl;        //输出值
      checked_delete(p);                         //使用checked_delete，更加安全
  }
};
```

处置器接口的示范代码如下：

```
std::vector<point*> vec;                           //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                         //添加5个指针
{
  vec.push_back(new point(i, i));
}

typedef make_list<point>::type list_t;             //侵入式容器
list_t lt(make_indirect_iterator(vec.begin()),     //迭代器区间构造
  make_indirect_iterator(vec.end()));              //使用间接迭代器，见5.6.4节

disposer d;                                        //一个处置器对象

lt.pop_front_and_dispose(d);                           //弹出前端元素

lt.erase_and_dispose(boost::next(lt.begin()), d);     //删除第二个元素
```

```
lt.remove_and_dispose(point(4, 4), d);              //移除值为(4, 4)的元素
lt.push_back(*(new point(3, 3)));                   //末端增加一个元素

lt.unique_and_dispose(d);                           //移除重复元素
```

程序的运行结果如下：

```
//初始链表元素是[ 0, 1, 2, 3, 4]
dispose:0                                           //链表是[ 1, 2, 3, 4]
dispose:2                                           //链表是[ 1, 3, 4]
dispose:4                                           //链表是[ 1, 3]
 //插入一个元素，链表是[ 1, 3, 3]
dispose:3                                           //链表是[ 1, 3]
```

处置器在大多数情况下执行删除指针操作，但它也可以执行其他任意的操作，比如把指针移动到一个指针容器中。例如，把处置器修改如下：

```
struct disposer
{
    template<typename Cont>                         //模板参数是一个序列指针容器
    void operator()(point* p, Cont* c)              //成员模板函数，双参数
    {
        c->push_back(p);                            //把指针移动到指针容器中
    }
};
```

这个处置器可以使用 bind 绑定使用的指针容器，将其适配为侵入式容器可以使用的单参数版本：

```
disposer d;                                         //处置器对象
ptr_vector<point> pvec;                             //一个指针容器
lt.pop_front_and_dispose(
        boost::bind<void>(d, _1, &pvec));           //bind 绑定指针容器
assert(pvec.size() == 1);                           //处置后指针被移至指针容器
```

代码中的 bind 需用模板参数显式指明返回类型（void），这是因为处置器函数对象 disposer 没有内部类型定义 result_type。

克隆

成员函数 clone_from()使用侵入式容器的克隆概念（9.3.8 节）从另一个容器中克隆元素到本容器内，要求元素必须有克隆器和处置器两个函数对象。

对于支持拷贝构造的类型来说，泛用的克隆器函数对象可以采用如下代码实现：

```
struct cloner                                    //泛用的克隆器函数对象
{
  template<typename T>
  T* operator()(const T& r)
  {    return factory<T*>()(r);}                //使用 factory 函数对象代替 new
};
```

示范 clone_from()用法的代码如下：

```
ptr_vector<point> vec =                         //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);            //使用 assign 库初始化

typedef make_list<point>::type list_t;
list_t lt(vec.begin(), vec.end());              //迭代器区间构造

list_t lt2;                                     //另一个侵入式链表容器
assert(lt2.empty());                            //此时容器无元素

lt2.clone_from(lt, cloner(), disposer());       //从 lt 克隆元素
assert(lt2 == lt);                              //比较两个容器内的元素，必定相等
```

迭代器特殊操作

因为侵入式容器修改了元素的代码，所有的元素都含有链接信息，所以侵入式容器可以直接从一个值获得对应的迭代器。

这个功能要求元素必须已经被容器容纳，即挂钩的 is_linked()返回 true，否则会产生一个断言异常。

所有的侵入式容器都提供成员函数 iterator_to()，它可以返回元素对应的迭代器位置。大多数侵入式容器（无序容器除外）还提供一个同等功能的静态成员函数 s_iterator_to()。

更进一步，静态成员函数 container_from_end_iterator()还可以从逾尾迭代器获得侵入式容器的引用。

这些成员函数用法的示范代码如下：

```
ptr_vector<point> vec =                             //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                //使用 assign 库初始化

typedef make_list<point>::type list_t;              //侵入式链表容器
list_t lt;
```

```
lt.push_back(vec[1]);                              //添加两个元素
lt.push_back(vec[3]);

assert(vec[1].is_linked());                        //元素已经被侵入式容器容纳
assert(lt.iterator_to(vec[1]) == lt.begin());      //获得迭代器

assert(vec[3].is_linked());                        //元素已经被侵入式容器容纳
assert(list_t::s_iterator_to(vec[3]) ==            //获得迭代器
  boost::prior(lt.end()));                         //是逾尾迭代器之前的位置

//从逾尾迭代器得到容器的引用
list_t& rlt = list_t::container_from_end_iterator(lt.end());
assert(addressof(rlt) == addressof(lt));           //两者是同一个对象
```

9.5 有序集合

intrusive 库基于红黑树、AVL 树、splay 树、scapegoat 树、二叉树和堆实现了五种侵入式有序集合容器，这些集合容器除了内部使用的算法不同外，接口基本相同，都类似于标准集合容器，故本书仅介绍基于红黑树的 set，其他容器读者可举一反三自行研究。

set 位于名字空间 boost::intrusive，需要包含头文件<boost/intrusive/set.hpp>。

9.5.1 节点和算法

set 基于红黑树实现，使用的节点类名字是 rbtree_node。它有两个实现类，使用 optimize_size 选项决定优化策略，紧凑版本可以节省一个整数的空间：[①]

```
template<class VoidPointer>
struct compact_rbtree_node                     //紧凑版本
{
   typedef pointer_rebind<VoidPointer, node>::type  node_ptr;
   enum color { red_t, black_t };              //颜色枚举
   node_ptr parent_, left_, right_;            //父指针和左右指针
};
```

① 其实现方式与 Nginx 类似，利用指针的最低位来存储颜色标志。

```
struct rbtree_node                             //普通版本
{
   ...                                         //同 compact_rbtree_node
   color color_;                               //多出的颜色变量
};
```

节点特征类 rbtree_node_traits 使用一个 bool 类型模板参数 OptimizeSize 定制，它再使用元函数 rbtree_node_traits_dispatch 特化到具体的实现类 default_rbtree_node_traits_impl 或者 compact_rbtree_node_traits_impl：

```
template<class VoidPointer, bool OptimizeSize = false>
struct rbtree_node_traits
   : public rbtree_node_traits_dispatch<VoidPointer, ...>
{};
```

set 使用的算法是 rbtree_algorithms，接口较链表容器的算法要复杂很多，这里从略。

9.5.2 基类挂钩

set 使用的基类挂钩是 set_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class set_base_hook
   : public make_set_base_hook<O1, O2, O3, O4>::type
{...};                                          //成员同 list_base_hook
```

set_base_hook 同样使用了工厂元函数，核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_set_base_hook
{
   typedef generic_hook                        //使用挂钩基类
   < rbtree_algorithms<...>                    //使用 rbtree_algorithms
   , typename packed_options::tag              //标签
   , packed_options::link_mode                 //链接模式
   , RbTreeBaseHookId                          //红黑树基类挂钩枚举值
   > implementation_defined;
};
```

set_base_hook 可以使四个选项：tag、void_pointer、link_mode 和 optimize_

size，其中前三个的含义同 list_base_hook（9.4.2 节），而第四个 optimize_size 可以通过取 true 或 false，来决定 set 是否优化空间。

9.5.3 成员挂钩

有序集合容器使用的成员挂钩是 set_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class set_member_hook
  : public make_set_member_hook<O1, O2, O3, O4>::type
{...};
```

set_member_hook 使用的工厂元函数 make_set_member_hook 的核心代码如下：

```
template<class O1, class O2, class O3, class O4e>
struct make_set_member_hook
{
   typedef detail::generic_hook>              //使用挂钩基类
   < rbtree_algorithms<...>                   //使用 rbtree_algorithms
   , member_tag                               //成员标签
   , packed_options::link_mode                //链接模式
   , NoBaseHookId                             //成员挂钩枚举值
   > implementation_defined;
};
```

set_member_hook 仅仅是在挂钩类别上与 set_base_hook 不同，模板参数、成员函数均相同。

9.5.4 set 类摘要

set 与 std::set 类似，容纳不允许重复的元素，它的类摘要如下：

```
template<class T, class O1, class O2, class O3, class O4>
class set
  : public make_set<T, O1, O2, O3, O4>::type    //工厂元函数
{
  public:
  ...                                          //类型定义和与 std::set 相同的操作

  set(const value_compare & );                 //构造函数
  set(Iterator, Iterator, const value_compare &);
```

```
  void         clone_from(const set &, Cloner, Disposer) ; //克隆

  //使用键操作
  size_type    erase(const KeyType &, KeyValueCompare) ;
  size_type    erase_and_dispose(const KeyType &, KeyValueCompare, Disposer) ;
  size_type    count(const KeyType &, KeyValueCompare) const;
  iterator     lower_bound(const KeyType &, KeyValueCompare) ;
  iterator     upper_bound(const KeyType &, KeyValueCompare) ;
  iterator     find(const KeyType &, KeyValueCompare) ;
  std::pair    equal_range(const KeyType &, KeyValueCompare) ;

  //带检查的插入操作
  std::pair< iterator, bool >
               insert_check(const KeyType &, KeyValueCompare,
                            insert_commit_data &);
  std::pair< iterator, bool >
               insert_check(const_iterator, const KeyType &, KeyValueCompare,
                            insert_commit_data &) ;
  iterator     insert_commit(reference, const insert_commit_data &) ;

  //使用处置器的操作
  iterator     erase_and_dispose(iterator, Disposer) ;
  iterator     erase_and_dispose(iterator, iterator, Disposer) ;
  size_type    erase_and_dispose(const_reference, Disposer) ;
  size_type    erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
  void         clear_and_dispose(Disposer) ;

  //从值获得迭代器
  iterator        iterator_to(reference) ;
  static iterator s_iterator_to(reference) ;

  //迭代器到容器的静态成员函数
  static set &    container_from_end_iterator(iterator) ;
  static set &    container_from_iterator(iterator) ;
};
```

set 使用四个定制选项，其中的 constant_time_size 、size_type 和 base_hook/member_hook/value_traits 与 list 的含义相同，compare 选项相当于 std::set 的比较谓词参数，用于定制排序准则，缺省是 std::less<T>。

工厂元函数 make_set 同样用于生产 set 容器，我们应该总使用它。

9.5.5 基本用法

set 具有与 std::set 相同的接口，因而很容易使用，我们只需要设置好元素类的挂钩，然后就可以使用 set 了，需要注意的是比较谓词应该使用 compare 来配置，这一点与 std::set 不同。

为了使用 set 容器，需要修改 point 的代码，使用 set_base_hook 或者 set_member_hook，例如，基类挂钩的实现如下：

```
class point: public set_base_hook<>,          //set 基类挂钩
boost::less_than_comparable<point>            //使用 operators 库，增加其他操作符
{...};                                        //同前
```

示范 set 基本用法的代码如下：

```
ptr_vector<point> vec =                                    //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                       //使用 assign 库初始化

typedef make_set<point,                                    //使用工厂元函数
  compare<std::greater<point>> >::type set_t;              //设置大于比较谓词
set_t s;

assert(s.empty());                                         //初始容器为空

assert(s.insert(vec[0]).second);                           //插入一个元素
assert(s.insert(vec[2]).second);
assert(!s.insert(vec[2]).second);                          //不允许重复插入

assert(s.size() == 2);                                     //获取容器大小
assert(s.count(point()) == 1);                             //计算元素的个数

s.insert(vec.begin(), vec.end());                  //插入一个区间内的元素，重复的不插入
assert(s.size() == 5);

s.erase(s.lower_bound(2), s.upper_bound(2));               //删除元素
```

set 的用法比较简单，代码中需要注意的是我们使用了 compare 选项，用 std::greater 而不是 std::less 作为比较谓词，所以这个有序集合是降序的。

9.5.6 特有用法

set 具有与 list 相同的一些侵入式容器特有的接口，包括克隆、处置器、迭代器的特殊操作，比较特别的是它无须使用逾尾迭代器就可以从迭代器获得容器的引用。

示范这些特殊操作用法的代码如下：

```
std::vector<point*> vec;                          //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                        //添加五个指针
{
  vec.push_back(new point(i, i));
}

typedef make_set<point>::type set_t;              //有序集合侵入式容器，升序
set_t s(make_indirect_iterator(vec.begin()),      //迭代器区间构造
        make_indirect_iterator(vec.end()));       //使用间接迭代器

//删除容器中的前两个元素
s.erase_and_dispose(s.begin(), boost::next(s.begin(), 2), disposer());

set_t s2;
s2.clone_from(s, cloner(), disposer());           //克隆容器
assert(s2.begin()->x == 2);
assert(s2 == s);                                  //比较两个容器，应相等

//从一个任意迭代器获取容器的引用
assert(addressof(s2) ==
  addressof(s2.container_from_iterator(boost::next(s2.begin()))));
```

使用键操作值

有的时候容器内存储的值类型（value_type）的构造成本很高，为了避免使用临时对象带来的开销，set 允许使用一个函数对象 KeyValueCompare 比较容器中元素是否与一个低成本的键 KeyType 相等，从而提高了运行效率。

函数对象 KeyValueCompare 是一个不等关系比较，它应该与集合容器 compare 选项配置的排序准则谓词一致（简单地说就是同为小于关系或者同为大于关系）。

我们可以为 point 定义一个小于关系的键比较函数对象，它与 std::less<point>一致：

```
struct key_compare
```

```
{
  typedef const int& key_type;                    //键类型
  typedef const point& value_type;                //值类型

  //因为要比较不同类型，所以必须有两个 operator()函数
  bool operator()(key_type k, value_type p) const
  {    return k < p.x;}
  bool operator()(value_type p, key_type k) const
  {    return p.x < k;}
};
```

set 的 count()、find()、erase()等成员函数都使用键的重载形式，使用 key_compare 函数对象的代码如下：

```
ptr_vector<point> vec =                           //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);              //使用 assign 库初始化

typedef make_set<point>::type set_t;              //使用缺省的 std::less<>
set_t s(vec.begin(), vec.end());                  //区间构造

key_compare kc;                                   //一个键比较函数对象

assert(s.count(1, kc) == 1);                      //使用键计算元素数量
assert(s.find(2, kc)->x == 2);                    //使用键查找元素
assert(s.find(9, kc) == s.end());

assert(s.erase(3, key_compare()) == 1);           //使用键删除元素
assert(s.find(point(3)) == s.end());              //构造值对象查找，开销大
```

“键—值”的功能除了用于处理大对象，也可以用来模拟实现映射容器，读者可自行试验。

检查插入

set 使用键操作为插入提供了一套类似于数据库的 check-commit 机制，它可以用在值类型的构造成本很高的场合。

insert_check()函数使用 KeyValueCompare 比较容器中的元素是否与 KeyType 相等，避免了构造大对象的开销，如果允许插入（返回的 pair 的 second 成员为 true），那么就可以紧接着调用 insert_commit()来完成插入操作。

insert_check()和 insert_commit()特别适合于侵入式容器容纳大对象的场合，在这

里我们仅用 point 类举例：

```
ptr_vector<point> vec =                                  //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                     //使用 assign 库初始化

typedef make_set<point>::type set_t;                     //使用缺省的 std::less
set_t s(vec.begin(),boost::next(vec.begin(), 3));        //插入三个元素

set_t::insert_commit_data idata;                         //一个用于提交的数据类型

//比较键 0，已经存在，无法插入
assert(!s.insert_check(0, key_compare(), idata).second);

//比较键 4，不存在，可以插入
assert(s.insert_check(4, key_compare(), idata).second);
s.insert_commit(vec[ 4] , idata);

assert(s.find(4, key_compare()) != s.end());             //4 已经插入容器
```

9.6 无序集合

unordered_set 和 unordered_multiset 是无序集合容器，它们不是使用指针链接而是使用散列表来实现元素的查找与存储，因而其实现方式比较特殊。这两个容器的功能基本相同，本节只介绍 unordered_set。

unordered_set 位于名字空间 boost::intrusive，需要包含头文件<boost/intrusive/unordered_set.hpp>。

9.6.1 节点和算法

无序集合容器使用的节点类是 unordered_node，它基于单链表节点 slist_node 实现，并使用 bool 类型模板参数进行了特化，摘要如下：

```
template<class VoidPointer, bool StoreHash, bool OptimizeMultiKey>
struct unordered_node    : public slist_node<VoidPointer>
{
```

```
    typedef some_define      node_ptr;
    node_ptr                 prev_in_group_; //根据 OptimizeMultiKey 选项可被优化
    std::size_t hash_;                       //根据 StoreHash 选项可被优化
};
```

无序集合容器的节点特征类是 unordered_node_traits，它基于单链表特征类，增加了操作散列值的成员函数：

```
template<class VoidPointer, bool StoreHash, bool OptimizeMultiKey>
struct unordered_node_traits  : public slist_node_traits<VoidPointer>
{
    static std::size_t     get_hash(const_node_ptr n);
    static void            set_hash(node_ptr n, std::size_t h);
};
```

无序集合容器使用的算法是 unordered_algorithms，它基于 circular_slist_algorithms。

9.6.2 基类挂钩

无序集合容器使用的基类挂钩是 unordered_set_base_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class unordered_set_base_hook
    : public make_unordered_set_base_hook< O1, O2, O3, O4>::type
{...};          //成员同 list_base_hook
```

工厂元函数 make_unordered_set_base_hook 的核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_unordered_set_base_hook
{
    typedef generic_hook                        //使用挂钩基类
    < get_uset_node_algo<...>                   //使用 unordered_algorithms
    , typename packed_options::tag              //标签
    , packed_options::link_mode                 //链接模式
    , HashBaseHookId                            //有序集合基类挂钩枚举值
    > implementation_defined;
};
```

unordered_set_base_hook 可以使用五个选项：tag、void_pointer、link_mode、

store_hash 和 optimize_multikey，其中前三个的含义同 list_base_hook（9.4.2 节），后两个选项的含义如下所述。

- store_hash　　　：挂钩（节点）中保存散列值（hash_），重散列时无须重新计算，可以提高性能，缺省值为 false。
- optimize_multikey：unordered_multiset 专用的选项，挂钩（节点）中存储指针（prev_in_group_），可以把相等的元素聚集在一起提高性能，缺省值为 false。

9.6.3　成员挂钩

无序集合容器使用的成员挂钩是 unordered_set_member_hook，它的类摘要如下：

```
template<class O1, class O2, class O3, class O4>
class unordered_set_member_hook
  : public make_unordered_set_member_hook< O1, O2, O3, O4>::type
{...};             //成员同 list_base_hook
```

工厂元函数 make_unordered_set_member_hook 的核心代码如下：

```
template<class O1, class O2, class O3, class O4>
struct make_unordered_set_member_hook
{
   typedef generic_hook                          //使用挂钩基类
   < get_uset_node_algo<...>                     //使用 unordered_algorithms
   , member_tag                                  //成员挂钩标签
   , packed_options::link_mode                   //链接模式
   , NoBaseHookId                                //成员挂钩枚举值
   > implementation_defined;
};
```

unordered_set_member_hook 仅仅是在挂钩类别上与 unordered_set_base_hook 不同，模板参数、成员函数均相同。

9.6.4　类摘要

unordered_set 类似于无序容器 std::unordered_set，类摘要如下：[①]

① unordered_set 有一个特殊的局部迭代器概念（local_iterator），本书未涉及，读者可自行研究。

```
template<class T, class O1, ..., class O10>
class unordered_set
   : public make_unordered_set<T,...>::type          //工厂元函数
{
  public:
  typedef some_define     bucket_type;                //桶类型
  typedef some_define     bucket_traits;              //桶特征
  ...                     //其他类型的定义以及与 std::unordered_set 相同的操作

  //构造函数
  unordered_set(const bucket_traits & );
  unordered_set(Iterator, Iterator, const bucket_traits &);

  //散列容器专用接口
  hasher       hash_function() const;
  key_equal    key_eq() const;
  void         rehash(const bucket_traits & new_bucket_traits) ;

  //克隆
  void         clone_from(const set &, Cloner, Disposer) ;

  //使用处置器的操作
  iterator     erase_and_dispose(iterator, Disposer) ;
  iterator     erase_and_dispose(iterator, iterator, Disposer) ;
  size_type    erase_and_dispose(const_reference, Disposer) ;
  void         clear_and_dispose(Disposer) ;

  //使用键操作
  size_type    count(const KeyType &, KeyHasher, KeyValueEqual);
  size_type    erase(const KeyType &, KeyHasher, KeyValueEqual) ;
  size_type    erase_and_dispose(const KeyType &, KeyHasher,
                                 KeyValueEqual, Disposer) ;
  iterator     find(const KeyType &, KeyHasher, KeyValueEqual) ;
  std::pair< iterator, iterator >
               equal_range(const KeyType &, KeyHasher, KeyValueEqual) ;

  //带检查的插入操作
  std::pair< iterator, bool >
               insert_check(const KeyType &, KeyHasher ,
                            KeyValueEqual, insert_commit_data &);
  iterator     insert_commit(reference, const insert_commit_data &) ;
```

```
  //由值获取迭代器
  iterator    iterator_to(reference);
};
```

unordered_set 可以使用多达十个元数据选项，常用的有 base_hook/member_hook/value_traits、constant_time_size、size_type、hash 和 equal，前几个已经做过介绍，后两个的含义如下所述。

- hash ：散列容器使用的散列函数对象，缺省是 boost::hash<T>。
- equal ：散列容器使用的相等比较函数对象，缺省是 std::equal_to<T>。

工厂元函数 make_unordered_set 用于生产 unordered_set 容器。

9.6.5 基本用法

unordered_set 属于半侵入式容器，故它的用法与纯侵入式容器有所不同——需要在外部额外分配一个辅助内存空间来维护散列容器所需的负载因子。

辅助内存空间是一个 unordered_set::bucket_type 类型的桶数组（静态或动态均可），每个无序容器必须使用专用的桶数组，其生命周期必须长于 unordered_set，也就是说必须在 unordered_set 销毁以后才能销毁，否则会发生未定义错误。

unordered_set 的构造函数需使用 unordered_set::bucket_traits 包装 bucket_type 数组，以使其作为参数初始化，例如：

```
typedef make_unordered_set<point>::type set_t;            //无序侵入式容器
set_t::bucket_type buckets[20];                            //辅助桶数组
set_t s(set_t::bucket_traits(buckets, 20));                //构造一个空容器
```

桶数组也可以使用动态数组：

```
std::vector<set_t::bucket_type> buckets(20);
set_t s(set_t::bucket_traits(&buckets[0], 20));
```

桶数组一旦被指定就不能变动，可以使用 rehash() 传入一个新的桶数组（原桶数组也可）重散列。

unordered_set 的用法基本类似于 set，但因为它是无序容器，故不要求元素有顺序关系（operator<），只要可以被散列和判等即可。

为了让 unordered_set 容纳 point 类，需要做如下的修改，增加相等比较和散列值计算：

```
class point: public unordered_set_base_hook<>            //基类挂钩
{
public:
  ...                                                    //数据成员和构造函数
  friend bool operator==(const point& l, const point& r) //相等操作
  {   return l.x == r.x && l.y == r.y;   }

  friend std::size_t hash_value(const point& p)          //散列函数，参见 7.1 节
  {
      size_t seed = 2016;
      hash_combine(seed, p.x);
      hash_combine(seed, p.y);
      return seed;
  }
};
```

示范 unordered_set 基本用法的代码如下：

```
ptr_vector<point> vec =                              //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                 //使用 assign 库初始化

typedef make_unordered_set<point>::type set_t;       //无序侵入式集合容器

std::vector<set_t::bucket_type> buckets(2);          //定义一个很小的辅助空间
set_t s(vec.begin(), vec.end(),                      //迭代器区间构造
  set_t::bucket_traits(&buckets[0], 2));             //指定桶

assert(s.size() == 5);                               //获取容器大小
assert(s.count(point(1)) == 1);                      //计算元素数量

for(auto& p : s)                           //无序容器只能正向遍历，无逆序遍历
{
  cout << p.x << ",";                                //输出结果是无序的
}

s.erase(s.find(point(1)));                           //查找并删除
assert(s.size() == 4);

set_t::bucket_type buckets2[10];                     //定义一个新的辅助空间
```

```
s.rehash(set_t::bucket_traits(buckets2, 10));          //重新散列

s.clear();                                             //清空容器
```

9.6.6 unordered_set 的特有用法

unordered_set 具有与 set 类似的一些侵入式容器特有的接口，包括克隆、处置器、迭代器特殊操作等，但稍有不同，区别如下所述。

- 没有 container_from_end_iterator()和 container_from_iterator()。
- 没有静态成员函数 s_iterator_to()。
- 不能执行容器比较操作。

示范这些特有接口的代码如下：

```
std::vector<point*> vec;                            //一个容纳原始指针的标准容器
for (int i = 0;i < 5; ++i)                          //添加五个指针
{
  vec.push_back(new point(i, i));
}

typedef make_unordered_set<point>::type set_t;

std::vector<set_t::bucket_type> buckets(5);         //辅助空间
set_t s(make_indirect_iterator(vec.begin()),        //迭代器区间构造
  make_indirect_iterator(vec.end()),                //使用间接迭代器
  set_t::bucket_traits(&buckets[0], 5));

//删除容器中的前两个元素
s.erase_and_dispose(s.begin(), boost::next(s.begin(), 2), disposer());

set_t::bucket_type buckets2[10];                    //定义一个新的辅助空间
set_t s2(set_t::bucket_traits(buckets2, 10));       //另一个无序容器

s2.clone_from(s, cloner(), disposer());             //克隆容器
assert(*s2.begin() == *s.begin());
```

使用键操作值

unordered_set 也可以使用键来操作值，但因为它是无序的，所以不使用比较函数对象

KeyValueCompare，而是使用 KeyHasher 和 KeyValueEqual，这两个函数对象分别对应于 equal 和 hash，差别仅在于是对键操作，结果应与对值的操作一致。

point 的两个键操作函数对象可采用如下代码实现：

```
typedef std::pair<int, int> ukey_type;                  //使用一个 pair 作为键
struct key_hasher                                       //散列函数对象
{
  std::size_t operator()(const ukey_type& k)           //算法应与 point 一致
  {
      size_t seed = 2016;
      hash_combine(seed, k.first);
      hash_combine(seed, k.second);
      return seed;
  }
};
struct key_equal                                                    //相等函数对象
{
  bool operator()(const ukey_type& k, const point& p)
  {
      return k.first == p.x && k.second == p.y;
  }
  bool operator()(const point& p, const ukey_type& k )
  {
      return operator()(k, p);
  }
};
```

示范 unordered_set 键操作的代码如下：

```
ptr_vector<point> vec =                                            //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4);                               //使用assign库初始化

point tmp(5, 5);                                                   //一个新元素，注意位置

typedef make_unordered_set<point>::type set_t;

std::vector<set_t::bucket_type> buckets(5);                        //辅助空间
set_t s(vec.begin(), vec.end(),                                    //区间构造
  set_t::bucket_traits(&buckets[ 0] , 2));

key_hasher kh;                                                     //散列函数对象
key_equal  keq;                                                    //相等函数对象
```

```
assert(s.count(make_pair(1,0), kh, keq) == 1);             //计算元素数量
assert(s.find(make_pair(2, 0), kh, keq) != s.end());       //查找元素

s.erase(make_pair(4, 0), kh, keq);                         //删除元素
assert(s.find(make_pair(4, 0), kh, keq) == s.end());

set_t::insert_commit_data idata;                           //用于提交的数据类型

//带检查的插入操作
assert(s.insert_check(make_pair(5, 5), kh, keq, idata).second);
s.insert_commit(tmp, idata);
assert(s.find(make_pair(5, 5), kh, keq) != s.end());
```

在这段代码中，我们需要注意被插入的元素 tmp 的生命周期，它必须在侵入式容器之前声明，否则就会在侵入式容器析构的时候产生运行时错误。这是因为 unordered_set 缺省采用的是 safe_link，具体原因可参见 9.7.2 节。

9.7 其他议题

在本节中，我们将讨论关于侵入式容器的一些其他议题。

9.7.1 链接模式

挂钩的链接模式（link mode）是一个非常重要的属性，因此有必要在这里再讨论一下。

链接模式有三种策略：safe_link、auto_unlink 和 normal_link。

safe_link 是最常用的一种链接模式，正如它的名字，使挂钩可以安全地处理链接。挂钩（也就是含有挂钩的节点类）构造时是未连接状态，挂钩析构（即节点类被销毁）时检查挂钩的连接状态，如果已连接则产生断言异常，从而保证了侵入式容器不会发生访问无效指针的错误。在插入容器时 safe_link 的挂钩也会检查连接状态，不允许重复插入同一个容器，移出容器时会自动把节点置为未连接状态，因此使用 safe_link 挂钩的容器总是安全的。

auto_unlink 挂钩的大部分行为与 safe_link 挂钩相同，但在析构时会自动调用算法的 unlink() 函数从容器中自我移除。auto_unlink 挂钩同时还为节点提供了可用的 unlink() 成员函数，用户可以在容器之外任意地把节点移出侵入式容器。auto_unlink 的这些特点令它在某些时候很方便，但它的安全性要差一些，因为容器内容有可能在未经过容器操作的情况下就

发生了改变，而且多线程时没有安全保证。还有一点，auto_unlink 不能使用常量时间的获取大小操作，也就是说必须在容器的模板参数中明确配置选项 constant_time_size<false>。

最后，normal_link 是一个完全无任何操作的“空模式”，在节点的构造、析构、插入、移除时都不做任何检查，适用于我们需要完全手工管理节点类的情形。

在本书中，我们主要使用的是 safe_link 模式，另外两种模式则较少使用，感兴趣的读者可以自行实践它们的用法。

9.7.2 同时使用多个挂钩

经过前几节的学习，我们已经了解了侵入式容器的用法，应该注意到侵入式容器的一个特点：它并不拷贝对象，也不分配内存，仅仅是把已有的对象链接在一起，因此，如果对象含有多个挂钩，那么就可以同时被多个（挂钩对应的）侵入式容器容纳，相当于给这些对象建立了多个不同索引方式的视图。

使用多个挂钩与使用单个挂钩没有什么区别，基类挂钩和成员挂钩可以任意混用，只是多个基类挂钩需要使用 tag 选项区分。

下面我们修改 point 的定义，为它增加多个挂钩：

```
class point:
  public list_base_hook<>,                              //链表基类挂钩，缺省标签
  public unordered_set_base_hook<tag<struct us_tag>> //无序集合基类挂钩
{
public:
  ...                                                   //同前

  set_member_hook<> m_shook;                          //用于有序单键集合的成员挂钩
  set_member_hook<> m_mshook;                         //用于有序多键集合的成员挂钩

  friend bool operator==(const point& l, const point& r)
  {...}
  friend bool operator<(const point& l, const point& r)
  {...}
  friend std::size_t hash_value(const point& p)
  {...}
};
```

这里我们为 point 增加了四个挂钩，这意味着一个对象可以同时被放入四种不同的侵入式容器，示范代码如下：

```
ptr_vector<point> vec =                                   //指针容器
  ptr_list_of<point>(0)(1)(2)(3)(4)(3)(2);                //使用 assign 库初始化

typedef make_list<point>::type list_t;                    //链表容器

typedef make_unordered_set<point,                         //无序集合容器
  base_hook<unordered_set_base_hook<tag<us_tag>>> >::type uset_t;

typedef make_set<point,                                   //有序单键集合容器
  member_hook<point, set_member_hook<>, &point::m_shook>>::type set_t;

typedef make_multiset<point,                              //有序多键集合容器
  member_hook<point, set_member_hook<>, &point::m_mshook>>::type mset_t;

list_t lt(vec.rbegin(), vec.rend());                      //逆序插入链表
for(auto& p : lt)
{
  cout << p.x << ",";
}

set_t s(lt.begin(), lt.end());                            //有序单键集合容器
mset_t ms(lt.begin(), lt.end());                          //有序多键集合容器
assert(s.size() == 5 && ms.size() == 7);

uset_t::bucket_type buckets[20];
uset_t us(uset_t::bucket_traits(buckets, 20));            //无序集合容器

us.insert(vec.begin(), vec.end());                        //插入元素
assert(us.size() == 5);

lt.pop_front();                                           //链表弹出一个元素
assert(lt.size() == 6 &&                                  //链表容量减少
  s.size() == 5 && ms.size() == 7);                       //其他容器不受影响

set_t::iterator iter =
  set_t::s_iterator_to(*us.begin());                      //从其他容器的迭代器获取迭代器
cout << iter->x << endl;
```

9.7.3 万能挂钩

同时使用多个挂钩是一个很有用的功能，但为被侵入类编写不同容器对应的挂钩代码显得有些麻烦。为此，intrusive 库在头文件<boost/intrusive/any_hook.hpp>中特别提供了可用于任意侵入式容器的“万能”式挂钩—— any_base_hook 和 any_member_hook。

any_base_hook 和 any_member_hook 可以替代任意的基类挂钩和成员挂钩，选项参数和用法完全相同，但注意不能使用 auto_unlink 连接模式，例如：

```
class point:
  public any_base_hook<>
{
public:
  ...                                           //数据成员，同前
  any_member_hook<>  m_shook;
  any_member_hook<>  m_mshook;
};
```

any_base_hook 和 any_member_hook 在用于配置容器时需要使用辅助元数据 any_to_*xxx*_hook，它用来把 any_hook 转换成特定容器所需的挂钩选项，代码如下所示：

```
typedef any_to_list_hook<base_hook<          //转换为链表挂钩
    any_base_hook<>>>      list_hook_opt;
typedef make_list<point, list_hook_opt>::type list_t;

typedef any_to_set_hook<member_hook<point,     //转换为有序单键集合挂钩
    any_member_hook<>, &point::m_shook>> set_hook_opt;
typedef make_set<point, set_hook_opt>::type set_t;

typedef any_to_set_hook<member_hook<point,     //转换为有序多键集合挂钩
    any_member_hook<>, &point::m_mshook>> mset_hook_opt;
typedef make_multiset<point, mset_hook_opt>::type mset_t;
```

9.8 总结

在本章中，我们研究了侵入式容器库 intrusive，它同指针容器一样，也是一个比较庞大的库，内容很丰富。

侵入式容器历史悠久，但大多数都需要由程序员自己手工编写，自 C++标准库出现以后，使

用者寥寥无几。boost.intrusive 则构建了全新的侵入式容器框架，提供了可与标准容器媲美的大量有用的侵入式容器，而且具有标准容器所缺乏的一些特性。

侵入式容器在本质上属于链式容器，使用指针连接各个节点，本身不具有内存管理功能，所以元素的生命周期管理是一个关键点：在使用侵入式容器时必须保证所有元素的生命周期都要长于侵入式容器，否则就可能会产生未知错误。如果要动态创建侵入式容器的元素，那么我们最好使用内存池或者指针容器来简化内存管理工作。

侵入式容器定义了节点、算法、挂钩、选项、处置器和克隆等一系列的概念，用到了大量的模板元编程技术，这些概念中最重要的是挂钩和选项。挂钩可分为基类挂钩和成员挂钩，两者的用法不同但效果相同，分别适用于需求不同的场合。选项（options）是元编程的一个很好的具体应用例子，它可以无视顺序对元数据打包，在编译期配置侵入式容器的各个特性，提高侵入式容器的运行时效率。由于元编程使用的类型声明比较复杂，通常需要适时使用 typedef 来使代码更加清晰易读。

intrusive 库提供了序列、有序集合和无序集合三大类侵入式容器，它们基本与标准容器等价，接口也非常类似，因而学习成本较低，易于掌握。但侵入式容器也有一些不同于标准容器的特殊操作，如处置器、克隆器、键—值操作等，这些操作的工作原理略为复杂，需要对侵入式容器的实现机制有较深刻的理解才能较好地掌握。

关于侵入式容器的时间复杂度、空间复杂度的定性定量分析，以及定制值特征等更高级的内容，限于作者水平暂不做讨论，请读者见谅。

第10章

多索引容器

在前两章里我们已经看到了Boost提供的指针容器和侵入式容器，本章研究本书中的最后一种特殊容器——multi_index_container，它有些类似于标准容器，但可以同时提供多种对元素的访问方式——很像是数据库领域中的多索引机制。

读者可以把多索引容器与前两章的指针容器和侵入式容器对比学习，三者之间有许多相似之处，通过交叉参考能够更好地理解这三大容器库。

10.1 概述

C++标准库提供的容器（如list、set）能够容纳可拷贝可赋值的元素，它们对外部呈现的访问接口反映了内部的存储结构，只能以一种已经确定的顺序访问元素（例如，list就只能顺序访问元素）。鉴于标准库的权威性，这个设计准则也被许多其他容器所接受。

但有的时候，这种固定顺序的访问方式会带来不便，我们可能会想对同一组元素执行不同的访问顺序准则，比如即要以顺序方式遍历list里的元素，又想以大小排序的方式遍历元素，或者像倒排表那样针对元素的某个属性查找，这时使用标准容器显然不能满足要求。

指针容器（第 8 章）和侵入式容器（第 9 章）在一定程度上为这个问题提供了解决方案。指针容器可以使用视图分配器（view_clone_allocator）对另一个指针容器建立视图，从而在不改变原容器的情况下提供新的访问方式；侵入式容器直接修改元素的结构，可以添加任意多个挂钩，每一个挂钩都可以对应一种访问接口。但这两种解决方案也有局限性：指针容器要求元素必须是专有（exclusive）的，有时还要满足克隆概念；而侵入式容器则必须修改元素的定义，而很多时候这是不被允许的。

boost.multi_index库是对这一问题的完美解决方案。从名字上就可以知道，它提供了“多索引”的容器，能够以不同的索引方式访问同一组存储在容器中的元素，并且没有指针容器或者侵入式容器的特殊要求。

multi_index位于名字空间boost::multi_index，需要包含头文件<boost/multi_index_container.hpp>和其他的一些索引信息头文件，即：

```
#include <boost/multi_index_container.hpp>    //容器头文件
#include <boost/multi_index/xxx.hpp>          //索引头文件如 ordered_index.hpp
using namespace boost::multi_index;           //打开名字空间
```

本书有时候为了特别强调也会使用名字空间的别名mi：

```
namespace mi = boost::multi_index;
```

10.2 入门示例

同前两章的指针容器和侵入式容器一样，对于multi_index这种新式容器，我们也用一些简单的代码来演示多索引容器的基本用法和功能，帮助读者快速了解multi_index库。

10.2.1 简单的例子

第一个例子非常简单，它虽然使用了多索引容器，但仅有一个索引，用法与标准容器set无异：

```
#include <boost/multi_index_container.hpp>          //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>      //有序索引
using namespace boost::multi_index;                 //打开名字空间

int main()
{
  multi_index_container<int> mic;                 //一个多索引容器，缺省使用有序索引
  assert(mic.empty());                            //缺省容器为空

  mic.insert(1);                                  //插入一个元素
  assert(mic.size() == 1);                        //使用 size()成员函数查看元素数量

  using namespace boost::assign;                  //打开 assign 库名字空间
  insert(mic)(2), 7, 6, 8;                        //使用 assign 库的插入函数添加元素
```

```
  assert(mic.size()   == 5);
  assert(mic.count(2) == 1);                         //使用 count()函数计算元素的数量
  assert(mic.find(10) == mic.end());                 //可以查找元素

  for(int i : mic)                                   //可以使用新式 for
  {
      cout << i << ',';                              //元素顺序输出: 1, 2, 6, 7, 8
  }
}
```

这段代码非常短小，除了容器的声明是multi_index_container<int>，其他部分与set<int>没有任何差异，从中我们可以看到多索引容器的几个基本特征。

- 多索引容器可以提供一个或多个索引接口，也就是说不一定是“多索引”。
- 多索引容器缺省使用有序索引，类似于 std::set，是一个有序单键集合。
- 多索引容器提供与标准容器用法完全一致的接口，如 insert()、find()等成员函数，因而很容易学习和使用。
- 多索引容器也能够使用标准容器的辅助工具，如新式 for、Boost 的 foreach 算法、C++11 的初始化列表和 assign 库，大大增强了它的实用价值。

当然，只有一个索引的多索引容器的意义是不大的，这实际上是多索引容器的退化形式，接下来我们看第二个例子，它是真正的“多索引容器”。

10.2.2 复杂的例子

第二段示例代码略微有些复杂，其重点在多索引容器的类型定义部分，它实质上是模板元编程领域中的元函数概念：

```
#include <boost/multi_index_container.hpp>              //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>          //有序索引
#include <boost/multi_index/hashed_index.hpp>           //散列(无序)索引
#include <boost/multi_index/key_extractors.hpp>         //键提取器
using namespace boost::multi_index;

int main()
{
  typedef multi_index_container<int,                  //多索引容器，元素类型为 int
      indexed_by<                                     //使用 index_by 元函数定义多个索引
```

```
        ordered_unique<identity<int>>,       //第一个是有序单键索引
        hashed_unique <identity<int>>        //第二个是散列（无序）单键索引
        >                                    //索引定义结束
    > mic_t;                                 //多索引容器定义结束，有两个索引
  mic_t mic = {2, 1, 7, 6, 8};               //多索引容器使用{...}初始化

  assert(mic.size() == 5);                   //默认第一个索引的接口同std::set
  assert(mic.count(2) == 1);
  assert(mic.find(10) == mic.end());

  //使用模板成员函数get()获取第二个索引，编号从0算起
  auto& hash_index = mic.get<1>();

  assert(hash_index.size() == 5);            //第二个索引的用法同unordered容器
  assert(hash_index.count(2) == 1);
  assert(hash_index.find(10) == hash_index.end());

  BOOST_FOREACH(int i, hash_index)           //同样可以使用foreach算法
  {
     cout << i << ",";                       //输出是无序的
  }
}
```

在这段代码中，我们需要注意的有下面几点：包含的头文件、多索引容器的定义和第一个索引以外的其他索引的获取与使用方法。

要使用multi_index库提供的其他索引方式必须在<boost/multi_index_container.hpp>之外手工添加所需的索引头文件，例如有序索引需使用<boost/multi_index/ordered_index.hpp>，散列（无序）容器需使用<boost/multi_index/hashed_index.hpp>，为了定义键的索引方式还需要包含键提取器头文件<boost/multi_index/key_extractors.hpp>。

接下来的重点是多索引容器的定义。多索引容器的类型是multi_index_container，它有两个模板参数。第一个是容纳的元素类型，对元素的要求与标准容器略低，必须是可拷贝的（不必是可赋值）。第二个模板参数用来声明容器上的索引类型，是一个indexed_by结构，它可以接受数个不同的索引，在本例中使用了一个有序单键索引（ordered_unique）和一个无序单键索引（hashed_unique）。索引又是一个元数据，它需要使用一种被称为键提取器（key extractor）的函数对象来获取用于产生索引的键类型。在这里，由于我们直接使用元素本身作为键，所以使用identity。这样我们就成功地定义了一个具有两个索引的多索引容器。

由于多索引容器的定义比较复杂，通常我们需要使用typedef来简化类型定义。

如果不显式指定索引，那么容器将使用第一个索引，效果和用法同之前的例子。想要获得第一个以外的索引就要使用模板成员函数get<*N*>()，它返回一个特定的索引类型的引用："mic_t::nth_index<*N*>::type&"。通常我们无须关心具体的类型信息，可以直接使用auto/decltype来简化变量的声明（注意必须是引用）：

```
auto&   hash_index  = mic.get<1>();          //正确，使用&声明引用变量
auto    hash_index  = mic.get<1>();          //错误，不能声明索引实例变量
```

索引就是多索引容器的访问接口，每个索引的用法都如同标准容器一样，只是访问准则是在最开始声明多索引容器时就已经确定好的。在这段代码中第一个索引是有序单键索引，相当于std::set，第二个索引是散列（无序）单键索引，相当于std::unordered。除了类型不同之外，这些索引的用法并无太多特殊之处。

10.2.3 更复杂的例子

在最后这个例子中，我们将一个具有较复杂结构的自定义类：person，作为我们后续讨论的元素类型，它实现了<、==等比较操作和散列运算，定义如下：

```
class person:
  boost::less_than_comparable<person>          //使用 operators 库实现全序比较
{
public:
  int     m_id;                                //身份标识号
  string  m_fname, m_lname;                    //姓名

  person(int id, const string& f, const string& l):      //构造函数
      m_id(id), m_fname(f), m_lname(l){}

  const string& first_name() const             //const 成员函数，取 m_fname
  {   return m_fname;}

  string& last_name()                          //非 const 成员函数，取 m_lname
  {   return m_lname;}

  friend bool operator<(const person&l, const person& r)     //比较操作
  {   return l.m_id < r.m_id;}

  friend bool operator==(const person& l, const person& r)   //相等操作
```

```
  {   return l.m_id == r.m_id ;}

  friend std::size_t hash_value(const person& p)      //散列函数，参见4.1节
  {
      size_t seed = 2016;
      hash_combine(seed, p.m_fname);
      hash_combine(seed, p.m_lname);
      return seed;
  }
};
```

下面的代码定义了一个有四个索引的容器：

```
#include <boost/multi_index_container.hpp>              //多索引容器头文件
#include <boost/multi_index/ordered_index.hpp>          //有序索引
#include <boost/multi_index/hashed_index.hpp>           //散列(无序)索引
#include <boost/multi_index/sequenced_index.hpp>        //序列索引
#include <boost/multi_index/key_extractors.hpp>         //键提取器
using namespace boost::multi_index;

int main()
{
  typedef multi_index_container<                     //多索引容器
      person,                                        //容器的元素类型为person
      indexed_by<                                    //使用index_by元函数定义多个索引
          sequenced<>,                               //第一个是序列索引
          ordered_unique<identity<person>>,          //第二个是有序单键索引
          ordered_non_unique<                        //第三个是有序多键索引
              member<person, string, &person::m_fname>>, //使用成员变量
          hashed_unique<identity<person>>            //第四个是散列（无序）单键索引
      >                                              //索引定义结束
  > mic_t;                                           //多索引容器定义结束

  mic_t mic;                                         //声明多索引容器变量

  using namespace boost::assign;                     //使用assign库
  push_front(mic)                        //第一个索引是序列索引，所以可以使用push_front()
      (person(2, "agent", "smith"))                  //插入四个元素
      (person(20, "lee", "someone"))                 //顺序无关紧要
      (person(1, "anderson", "neo"))                 //id不可重复
```

```
    (person(10, "lee", "bruce"));           //m_fname 可重复

auto& index0 = mic.get<0>();                //使用 auto 获取四个索引
auto& index1 = mic.get<1>();
auto& index2 = mic.get<2>();
auto& index3 = mic.get<3>();

//使用索引 0，顺序输出元素，没有排序
for(const person& p : index0)
{
    cout << p.m_id << p.m_fname << ",";
}

//索引 1，获取 id 最小的元素
assert(index1.begin()->m_id == 1);

//索引 2，允许重复键的元素
assert(index2.count("lee") == 2);

//使用 assign::insert 在索引 2 上插入重复键的元素
insert(index2)(person(30, "lee", "test"));
assert(index3.size() == 5);

//插入 id 重复的元素因索引 1 的限制而不能成功
assert(!index2.insert(person(2, "lee", "test2")).second);
}
```

上面这段代码中的多索引容器明显比前两个例子要复杂很多，它一共定义了四个索引，我们可以用序列方式、有序单键方式、有序多键（基于m_fname）和无序单键方式来访问同一组元素。因为多索引容器的定义很复杂，所以良好的缩进格式和注释是很有必要的。

容器的第一个索引是sequenced，它并不对元素排序，所以通常不需要模板参数；第二个索引是ordered_unique，它是有序单键索引，要求元素具有operator<；第四个索引是hashed_unique，它是无序单键索引，要求元素满足boost.hash可以计算散列值；第三个索引是ordered_non_unique，它使用了一个不同于identity的新的键提取器member，形式上类似侵入式容器的member_hook，可以使用元素的成员变量作为键（倒排索引）。

这些索引都可以使用模板成员函数get<N>()获取，彼此的读操作是独立的，但插入删除等变动元素的操作可能会受到其他索引的制约。比如代码的最后三行，第三个索引可以向容器里插入任意m_fname重复的元素，但不能插入m_id重复的元素，因为第二个索引被声明为

ordered_unique，它的operator<()使用了m_id成员，所以必须是唯一的。

10.3 基本概念

通过前面的三个例子我们已经对多索引容器有了初步的了解，本节将介绍multi_index库中构建多索引容器的基本概念，这些概念的实现大量使用了模板元编程，本书节选代码时从略。

10.3.1 索引

索引（index）是multi_index库中最重要的概念，它是多索引容器里访问元素的接口，决定了外部用户以何种准则对元素执行读/写/查找等操作。不同的索引有不同的使用接口，索引通常的形式如下：

```
template<typename KeyFromValue,...>
class some_index
{
public:
   typedef some_define     key_type;
   typedef some_define     value_type;
    ...                    //其他类型定义

   iterator                begin();
   iterator                end();
   bool                    empty();
   size_type               size();
   ...                     //其他成员函数
};
```

索引的接口定义完全模仿了标准容器，具有大多数常用的成员函数，所以在使用索引时完全可以把它当作是标准容器，但索引也有不同于标准容器的地方：索引通常不允许直接修改元素以保护索引的结构不会被意外破坏，而且因为一个多索引容器可能持有多个索引，故操作元素时可能存在相互的制约关系，某些操作可能会因此无法执行。

虽然索引是确定的类型，但它被multi_index库定义为一个内部使用的类型（在boost::multi_index::detail子名字空间里），不能单独使用，也就是说我们无法自行声明一个索引类型的变量。

索引的使用必须依附于多索引容器，在multi_index_container的模板参数中用索引说明（index specifier）定义，然后用模板成员函数get<N>()来获得引用才能操作。

10.3.2 索引说明

索引的定义需要在multi_index_container的模板参数中使用索引说明（index specifier）。

索引说明是一个高阶元数据，它使用键提取器和其他参数来定义索引，内部有两个名为node_class和index_class的元函数，返回可供容器使用的节点类型和索引类型。

索引说明的基本形式如下：

```
template<typename Arg1,typename Arg2,...>
struct some_index_specifier
{
  template<typename Super>
  struct node_class
  {
    typedef some_define type;
  };

  template<typename SuperMeta>
  struct index_class
  {
    typedef some_define type;
  };
};
```

目前multi_index库提供以下四类索引说明（为叙述方便，本章之后有时会将索引说明简称为索引，请读者注意）。

- 序列索引　　　　　：这类索引只有一个 sequenced，是类似 std::list 的双向链表序列访问接口，见 10.5 节。
- 随机访问索引　　　：这类索引只有一个 random_access，是类似 std::vector 的序列访问接口，提供 operator[ ] 方式的随机访问能力，见 10.6 节。
- 有序索引　　　　　：包括 ordered_unique 和 ordered_non_unique，是类似 std::set 的有序集合访问接口，见 10.7 节。
- 散列（无序）索引　：包括 hashed_unique 和 hashed_non_unique，是类似 std::unordered_set 的无序集合访问接口，见 10.8 节。

10.3.3 键提取器

键提取器（key extractor）是一个单参函数对象，它可以从元素或元素的boost.ref包装中获取用作索引（排序）的键，通常被用作索引说明的第一个模板参数，其简要形式如下：

```
template<typename Arg1,typename Arg2,...>
struct some_key_extractor
{
  typedef some_define     result_type;
  some_type&              operator()(...)const;
};
```

multi_index库提供了以下六种键提取器，可以适用于各种情况。

- identity ：使用元素本身作为键。
- member ：使用元素的某个 public 成员变量作为键。
- const_mem_fun ：使用元素的某个 const 成员函数的返回值作为键。
- mem_fun ：使用元素的某个非 const 成员函数的返回值作为键。
- global_fun ：使用操作元素的某个全局函数或静态成员函数的返回值作为键。
- composite_key ：可以把以上的键提取器组合为一个新的键。

键提取器获取的键类型必须能够满足索引的要求，例如有序索引要求键定义了比较操作符，散列索引要求键可以计算散列值和执行相等比较。这些键提取器的具体讨论参见 10.4 节。

10.3.4 索引说明列表

索引说明列表（index specifier list）是多索引容器multi_index_container的第二个模板参数，实现为一个indexed_by结构，用来定义多索引容器使用的索引，类摘要如下：

```
template<typename T0, typename T1, ...>
struct indexed_by: mpl::vector<T0, T1, ...>
{};
```

indexed_by基于模板元编程库mpl里的元数据序列mpl::vector（见 13.4.2 节），是一个类型的容器，可以容纳多个索引说明。它使用了预处理元编程，当前的实现限定最多可容纳 20 个元数据，也就是说最多支持在一个容器上同时使用 20 个索引（相信已经足够用了）。

10.3.5 索引标签

多索引容器通常会持多个索引，这些索引可以使用容器的模板成员函数get<N>()获取，其中的N是索引在索引说明列表中的序号（从0算起），但单纯使用序号来获取索引很不直观，不方便记忆，所以multi_index库提供索引标签（tag）的概念，允许使用语法标签来访问索引。

索引标签的定义使用模板类tag，它的类摘要如下：

```
template<typename T0, typename T1, ...>
struct tag
{
  typedef some_define type;
};
```

tag也是一个mpl类型容器，最多支持20个类型同时作为索引标签。标签类型是任意的，不仅可以使用自定义类型，甚至还可以使用C++内建类型如int、std::string。

使用索引标签需要把tag作为索引说明的第一个模板参数，键提取器需在标签之后，例如：

```
ordered_unique<tag<struct id>,...>         //使用自定义类型 struct id 作为标签
hashed_unique<tag<int,short>,...>          //使用内建整数类型 int、short 作为标签
```

但在同一个多索引容器中不允许使用重复的标签，否则会发生编译错误，例如：

```
ordered_unique<tag<struct id, id>,...>    //自定义标签重复
hashed_unique<tag<int,id>,...>            //与上一个索引的标签重复
```

模板成员函数get<N>()有针对索引标签的重载形式，可以使用标签来获得索引，例如：

```
auto& ordered_index = mic.get<id>();
auto& hashed_index  = mic.get<int>();
```

10.3.6 多索引容器

在multi_index库中多索引容器的类型是multi_index_container，它的声明是：

```
template<
  typename Value,                                              //元素类型
  typename IndexSpecifierList=                                 //索引说明列表
           indexed_by<ordered_unique<identity<Value> > >,      //缺省值
  typename Allocator=std::allocator<Value> >                   //内存分配器
```

```
class multi_index_container;
```

multi_index_container的类声明可以与标准库的容器进行对比学习：模板参数分别是元素类型，排序准则和分配器类型，只不过排序准则非常复杂，是一个indexed_by类型的索引声明列表。索引说明列表缺省提供一个ordered_unique，意味着如果不指定索引说明，多索引容器的默认行为同std::set，是一个不允许重复的有序集合。

multi_index_container很像是一个索引的管理器，自身并没有太多的元素操作功能，大多数时候我们都通过get<N>()以索引序号或者索引标签来获取索引再使用。

多个索引的好处不只是提供多个不同形式的访问接口那么简单，我们还可以充分利用不同存储结构的时间优势，例如使用序列索引顺序存储元素，再用散列索引实现对元素的快速查找，用有序索引实现对元素的排序，多索引容器为我们提供了强大的数据操作能力。

10.4 键提取器

multi_index库强化了标准库中的键（key）概念，把原始的键类型（key type）扩展为一个功能更强的函数对象——键提取器，可以从元素中提取关联的用于排序的键，是multi_index库建立索引的基础。

键提取器分为可写和只读两类，可写键提取器总可以返回一个非常量的键引用，不仅可以读也可以写，而只读键提取器总返回常量的键引用，只能读。大多数索引只要求只读键提取器，仅在有序索引和散列索引使用modify_key()时才要求可写键提取器（参见10.9.3节）。

10.4.1 定义

键提取器是一个函数对象，它的基本形式如下：

```
struct some_key_extractor
{
  typedef T result_type;                                //返回类型定义

  T& operator()(T& x)const;                             //操作原始类型
  T& operator()(const reference_wrapper<T>& x)const;  //操作引用包装类型

  template<typename ChainedPtr>
  Type& operator()(const ChainedPtr& x)const;          //操作链式指针类型
```

```
};
```

键提取器的operator()不仅可以操作类型本身（T&），也可以操作被boost.ref库包装的对象（reference_wrapper<T>&），而且它还支持“链式指针”（chained pointer）对象。

所谓“链式指针”是指一系列类似于指针对象的组合——包括原始指针、智能指针、迭代器等一切具有operator*可以执行解引用操作的类型，T*、T**、shared_ptr<T*>都属于链式指针类型，它们可以被连续递归解引用得到一个非指针类型T&或者reference_wrapper<T>&。键提取器的这个特性可以让我们轻松地操作指针，让多索引容器可以轻松地容纳指针类型的元素。

multi_index库中所有预定义的键提取器均位于头文件<boost/multi_index/key_extractors.hpp>，如果想要减少编译的时间（元计算的时间），那么也可以视需要只包含必要的头文件。

接下来我们将逐个介绍这些键提取器，但由于composite_key用法比较复杂，将留在10.11节学习。

10.4.2 identity

identity是一个最简单的键提取器，它不做任何“提取”动作中，直接使用元素本身作为键，相当于标准容器的键类型（key type）。只要元素类型不是const，那么它就是可写的键提取器。

identity位于头文件<boost/multi_index/identity.hpp>，其类摘要如下：

```
template<class Type>
struct identity:
  mpl::if_c<
    is_const<Type>::value,
    detail::const_identity_base<Type>,
    detail::non_const_identity_base<Type>
  >::type
{};
```

identity使用了元函数if_c，根据类型Type是否被const修饰分别转交给const_identity_base和non_const_identity_base处理，这两个实现类的具体代码差异很小，下面列出const_identity_base的主要代码：

```
template<typename Type>
struct const_identity_base
```

```
{
  typedef Type result_type;                               //返回类型定义

  //操作元素类型本身
  Type& operator()(Type& x)const
  {    return x; }

  //操作元素类型的 reference_wrapper 包装
  Type& operator()(const reference_wrapper<Type>& x)const
  {    return x.get(); }

  //操作链式指针
  template<typename ChainedPtr>
  typename disable_if<
          is_convertible<const ChainedPtr&,Type&>,Type&>::type
  operator()(const ChainedPtr& x)const
  {    return operator()(*x); }
};
```

const_identity_base的前两个operator()都很好理解，它们直接返回变量自身。最后一个重载形式用于处理链式指针，它使用了 12.1 节介绍的元函数disable_if，当模板参数ChainedPtr是指针类型时编译器递归生成解引用的operator()，这样在运行时就可以连续调用直至获取最终的Type&类型。

离开多索引容器的范围，identity就是一个普通的函数对象，像是对类型做了一层薄薄的包装，例如：

```
assert((is_same<string, identity<string>::result_type>::value));
assert(identity<int>()(10) == 10);
assert(identity<string>()("abc") == "abc");

int* p      = new int(100);                         //指针
int** pp    = &p;                                   //指针的指针(链式指针)
assert(identity<int>()(pp) == 100);                 //从链式指针中获取键
```

在多索引容器里，identity可以用于简单的类型（如int、string）或者本身已经定义了比较、散列操作的类型，如之前的person。

10.4.3 member

键提取器member有些类似于非标准函数对象select1st的功能，可以提取类型里的某个public成员变量作为键。它的接口、实现与identity基本相同，支持从类型本身、ref包装和链式指针里提取键。只要元素类型不是const，那么它就是可写的键提取器。

member位于头文件<boost/multi_index/member.hpp>，类摘要如下：

```
template<class Class,typename Type,Type Class::*PtrToMember>
struct member:
  mpl::if_c<
    is_const<Type>::value,
    detail::const_member_base<Class,Type,PtrToMember>,
    detail::non_const_member_base<Class,Type,PtrToMember>
  >::type
{};
```

member有三个模板参数，在形式上很像侵入式容器的member_hook选项，指定要提取的类型Class、键类型Type（即类型的成员变量类型）以及成员变量指针PtrToMember，例如：

```
typedef pair<int, string> pair_t;
pair_t p(1, "one");

assert((member<pair_t, int, &pair_t::first>()(p)         == 1));
assert((member<pair_t, string, &pair_t::second>()(p)     == "one"));

person per(1, "anderson", "neo");
assert((member<person, int, &person::m_id>()(per)         == 1));
```

某些对C++标准支持较差的编译器可能无法使用member，所以multi_index提供了一个等价的替代品：member_offset，它使用偏移量来代替成员指针，本书对它不做讨论，读者有需要可阅读Boost文档（用法很简单）。

为了获得最大的兼容性并且方便使用，multi_index库定义了一个宏BOOST_MULTI_INDEX_MEMBER，它无须我们手工写出成员变量指针的声明，而且会根据编译器的功能自动选用member或者member_offset：

```
#define BOOST_MULTI_INDEX_MEMBER(Class,Type,MemberName)
```

BOOST_MULTI_INDEX_MEMBER有三个参数，前两个Class和Type与member的模板参数定义相同，后一个是成员变量名，不需要写出取地址操作符和类名限定。对于大多数支持成员指针模板参数的编译器，宏展开如下：

```
::boost::multi_index::member< Class,Type,&Class::MemberName >
```

使用宏BOOST_MULTI_INDEX_MEMBER可以改写上面的代码，如下所示：

```
assert((BOOST_MULTI_INDEX_MEMBER(pair_t, int, first)()(p)        == 1));
assert((BOOST_MULTI_INDEX_MEMBER(pair_t, string, second)()(p)    == "one"));
```

```
assert((BOOST_MULTI_INDEX_MEMBER(person, int, m_id)()(per)        == 1));
```

显然，因为无须写出成员变量指针，所以宏的写法更简单清晰，只是宏的名字略长，算是个小缺点。

10.4.4 const_mem_fun

const_mem_fun使用类型中的某个const成员函数的返回值作为键，有些类似于函数对象mem_fn（参见7.2节），但它在使用上有两个限制：只能调用const成员函数，而且这个成员函数必须是无参调用。

const_mem_fun是一个只读键提取器，位于头文件<boost/multi_index/mem_fun.hpp>，它的类摘要如下：

```
template<class Class,typename Type,
         Type (Class::*PtrToMemberFunction)()const>
struct const_mem_fun
{
  typedef typename remove_reference<Type>::type result_type;

  Type operator()(const Class& x)const
  {
    return (x.*PtrToMemberFunction)();          //调用无参 const 成员函数
  }
  ...                                           //其他 operator()定义
};
```

const_mem_fun的模板参数与member类似，但最后一个参数是成员函数指针，同时Class和Type参数必须与成员函数指针精确匹配，示范代码如下：

```
string str("abc");
typedef const_mem_fun<string, size_t, &string::size > cmf_t;
assert(cmf_t()(str) == 3);

person per(1, "anderson", "neo");
typedef const_mem_fun<person, const string&, &person::first_name> cmf_t2;
assert((cmf_t2()(per) == "anderson"));
```

下面的两行代码会因为类型错误而无法编译。

```
typedef const_mem_fun<const person,
                  const string&, &person::first_name>       cmf_t2;
```

```
typedef const_mem_fun<person, string&, &person::first_name> cmf_t2;
```

const_mem_fun也有一个用于屏蔽编译器差异的宏BOOST_MULTI_INDEX_CONST_MEM_FUN，定义如下：

```
#define BOOST_MULTI_INDEX_CONST_MEM_FUN(Class,Type,MemberFunName) \
::boost::multi_index::const_mem_fun<
                                Class,Type,&Class::MemberFunName >
```

宏BOOST_MULTI_INDEX_CONST_MEM_FUN可以这样使用：

```
typedef BOOST_MULTI_INDEX_CONST_MEM_FUN(string, size_t, size)   cmf_t;
typedef BOOST_MULTI_INDEX_CONST_MEM_FUN(person, \
                                const string&, first_name)        cmf_t2;
```

10.4.5 mem_fun

mem_fun与const_mem_fun类似，使用元素的某个成员函数的返回值作为键，但它只适用于成员函数是非const的情形。

mem_fun是一个只读键提取器，位于头文件<boost/multi_index/mem_fun.hpp>，它的类摘要如下：

```
template<class Class,typename Type,
        Type (Class::*PtrToMemberFunction)() >
struct mem_fun
{
  typedef typename remove_reference<Type>::type result_type;
  ...                           // operator()定义，同 const_mem_fun
};
```

注意：mem_fun与标准库里的函数对象std::mem_fun重名，因此，在使用时可能需要加名字空间boost::multi_index限定，例如：

```
person per(1, "anderson", "neo");
typedef mi::mem_fun<person, string&, &person::last_name> mf_t;
assert((mf_t()(per) == "neo"));
```

我们也可以使用宏BOOST_MULTI_INDEX_MEM_FUN，它的声明如下：

```
#define BOOST_MULTI_INDEX_MEM_FUN(Class,Type,MemberFunName) \
      ::boost::multi_index::mem_fun< Class,Type,&Class::MemberFunName >
```

宏BOOST_MULTI_INDEX_MEM_FUN不必考虑名字空间的问题（展开时已经有了名字空间限定），可以这样使用：

```
typedef BOOST_MULTI_INDEX_MEM_FUN(person, string&, last_name) mf_t;
```

我们在使用时必须注意mem_fun的特点：它操作的是非const成员函数，而索引的大多数函数的接口都是常量性的const T&，所以如果直接存储元素类型T会因为无法获取键而导致编译失败。不过如果容器存储的是指针T*，那么mem_fun就可以不受限制地使用。

10.4.6 global_fun

global_fun使用一个全局函数或静态成员函数来操作元素，将函数的返回值作为键，它的实现类似于identity，支持const或非const的函数。

global_fun是一个只读键提取器，位于头文件<boost/multi_index/global_fun.hpp>，它的类摘要如下：

```
template<class Value,typename Type,Type (*PtrToFunction)(Value)>
struct global_fun:
  mpl::if_c<...>::type
{};
```

global_fun的用法类似于const_mem_fun和mem_fun，只是最后一个模板参数必须是一个参数为Value类型的函数指针。

我们为person类定义一个全局函数nameof()，它返回person的全名：

```
string nameof(const person& p)
{
  return p.m_fname + " "  +p.m_lname;
}
```

之后我们就可以使用global_fun：

```
person per(1, "anderson", "neo");
typedef global_fun<const person&, string, &nameof> gf_t;
assert(gf_t()(per) == "anderson neo");
```

使用global_fun同样要注意类型参数要与函数指针精确匹配，否则将无法通过编译，例如：

```
typedef global_fun<person&, string, &nameof>          gf_t;   //错误
typedef global_fun<const person&, string&, &nameof>  gf_t;   //错误
```

10.4.7 自定义键提取器

键提取器实质上是一个单参函数对象，所以我们也可以不使用multi_index库预定义的键提取器，完全自行编写——只需要满足键提取器的定义即可（静态多态）。

自定义键提取器首先要满足标准函数对象的要求，定义包含内部类型result_type，它同时也是键类型。然后要实现操作元素类型的operator()，必须是const成员函数，根据需要可实现数个针对T&、reference_wrapper<T>&和ChainedPtr&的重载，但不必都实现。

例如，我们可以编写一个键提取器person_name，它实现与global_fun<const person&, string, &nameof>等价的功能：

```
struct person_name
{
    typedef string result_type;                          //返回值类型定义，必需
    result_type operator()(const person& p) const        //必须为 const
    {
        return p.m_fname + " " +p.m_lname;
    }
    result_type operator()(person *const p) const        //支持容纳原始指针
    {
        return p->m_fname + " " +p->m_lname;
    }
};
```

自定义键提取器的用法参见 10.10.2 节。

10.5 序列索引

序列索引是multi_index库提供的最简单的一种索引，它实际上没有对元素做任何索引操作，仅仅是顺序存储元素。

序列索引位于头文件<boost/multi_index/sequenced_index.hpp>。

10.5.1 索引说明

序列索引的索引说明是sequenced，它的类摘要如下：

```
template <typename TagList = tag<> >
```

```
struct sequenced
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::sequenced_index<...> type;
  };
};
```

因为序列索引不基于键排序，所以sequenced不使用任何键提取器，只有一个TagList模板参数，用来给索引贴语法标签。

10.5.2 类摘要

序列索引使用的类是detail::sequenced_index，它提供类似于std::list的双向链表操作，但有些接口是常量性的，不能随意修改元素。如果想要修改容器里的元素则需要使用特别的成员函数，参见10.9节。

sequenced_index的类摘要如下：

```
class sequenced_index
{
  public:
  typedef some_define          value_type;
  typedef some_define          iterator;
  typedef iterator             const_iterator;
  ...                          //其他类型定义

  //赋值操作
  sequenced_index&             operator=(const sequenced_index& x);
  void                     assign(InputIterator first,InputIterator last);
  void                     assign(size_type n,const value_type& value);

  //迭代器操作
  iterator                     begin();
  iterator                     end();
  iterator                     iterator_to(const value_type& x);

  //元素访问
  const_reference              front()const;
  const_reference              back() const;
```

```
    std::pair<iterator,bool>    push_front(const value_type& x);
    void                        pop_front();
    std::pair<iterator,bool>    push_back(const value_type& x);
    void                        pop_back();

    std::pair<iterator,bool>    insert(iterator position,const value_type& x);
    void                 insert(iterator position,size_type n,const value_type& x);
    iterator                    erase(iterator position);
    iterator                    erase(iterator first,iterator last);
    ...                         //其他 remove()、unique()、splice()、sort()等操作

    ...                         //各种比较操作符定义
};
```

sequenced_index的接口与list基本相同，因而很容易使用，需要注意的有下几点。

- 索引不提供 public 的构造函数和析构函数（由多索引容器处理），但可以使用 operator=和 assign()赋值，用法同标准容器。
- 解引用迭代器（begin()、rbegin()等）返回的都是 const 类型，所以不能使用迭代器修改元素。
- 访问元素的 front()和 back()函数返回的是 const 引用，不能修改元素。
- 因为可能存在其他索引的约束，push_front()、push_back()和 insert()可能会执行失败，所以它们的返回值是同 set::insert()一样的一个 pair，second 成员表示操作是否成功。
- 与侵入式容器类似，索引提供 iterator_to()函数，可以从容器内一个元素的引用（不是拷贝）获取相应的迭代器。

10.5.3 用法

因为序列索引不使用键提取器，也不涉及元素的排序，因而用法非常简单，完全可以把它当作std::list的一个等价物（但不允许直接修改元素的值）。

示范序列索引用法的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
using namespace boost::multi_index;                    //打开名字空间
```

```
int main()
{
  typedef multi_index_container<int,                    //存储 int 类型的元素
      indexed_by<sequenced<                             //使用序列索引
                 tag<int, struct int_seq> > >           //添加两个标签
      > mic_t;

  mic_t mic = {2,3,5,7,11};                             //C++11 列表初始化

  assert(!mic.empty() && mic.size() == 5);              //容器容量操作

  assert(mic.front() == 2);                       //容器缺省使用第一个索引，即序列索引
  assert(mic.back() == 11);

  assert(mic.push_front(2).second);                     //目前没有其他索引约束
  assert(mic.push_back(19).second);                     //所以总可以增加元素

  auto& seq_index = mic.get<int_seq>();                 //使用标签获得索引

  seq_index.insert(seq_index.begin(), 5, 100);          //插入五个元素
  assert(std::count(seq_index.begin(), mic.end(), 100) == 5);

  seq_index.unique();                                   //删除重复的元素
  assert(std::count(seq_index.begin(), mic.end(), 100) == 1);
}
```

序列索引也支持容器间的比较操作，与std::list一样：

```
mic_t mic1 = (list_of(2),3,5,7,11);                    //初始化

mic_t mic2 = mic1;                                     //赋值操作
assert(mic1 == mic2);                                  //两个容器内的元素一致，比较相等

mic2.push_back(3);                                     //添加一个元素
assert(mic1 < mic2);                                   //比较不等
```

虽然序列索引很像std::list，但它与std::list的一个重要区别是不能随意修改元素的值，使用时要注意。例如下面的代码会产生编译错误：

```
*seq_index.begin() = 9;                                //编译错误，报 const 错
```

另外，成员函数iterator_to()的使用也值得注意，只有当参数确实是容器内元素的引用时才能返回正确的迭代器：

```
auto& x = mic.front();                          //获取元素的引用
assert(mic.begin() == mic.iterator_to(x));      //获取对应的迭代器
```

使用与元素等价的拷贝虽然也可以调用iterator_to()，但返回的迭代器与索引没有任何关系，使用这个迭代器去执行其他操作会导致运行时错误：

```
assert(mic.begin() != mic.iterator_to(2));      //返回的迭代器不能使用
```

10.6 随机访问索引

随机访问索引是multi_index库提供的另一种序列索引，它同样是顺序存储元素，但提供了比序列索引更多的访问接口。

随机访问索引位于头文件<boost/multi_index/random_access_index.hpp>。

10.6.1 索引说明

随机访问索引的索引说明是random_access，它的类摘要如下：

```
template <typename TagList = tag<> >
struct random_access
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::random_access_index<...>  type;
  };
};
```

random_access与sequenced非常相似，同样不使用键提取器，只有一个标签模板参数。

10.6.2 类摘要

随机访问索引使用的类是detail::random_access_index，它是sequenced_index的超集，拥有sequenced_index的全部功能，类摘要如下：

```
class random_access_index
{
```

```
  public:
    typedef some_define      value_type;
    typedef some_define      iterator;
    typedef iterator         const_iterator;
    ...                      //其他类型定义

    //赋值操作
    random_access_index&     operator=(const random_access_index& x);
    void                     assign(InputIterator first,InputIterator last);
    void                     assign(size_type n,const value_type& value);

    //迭代器操作
    iterator                 begin();
    iterator                 end();
    iterator                 iterator_to(const value_type& x);

    //元素访问
    const_reference          operator[](size_type n);           //随机访问接口
    const_reference          at(size_type n)const;              //随机访问接口
    const_reference          front()const;
    const_reference          back()const;

    ...                      //其他同 sequenced_index 的操作
    ...                      //各种比较操作符定义
};
```

random_access_index比sequenced_index多出了两个可以随机访问任意位置元素的operator[]和at()，其他的接口与sequenced_index相同，但操作的时间复杂度不同。

10.6.3 用法

random_access_index的用法几乎与sequenced_index一样，因为提供了随机访问的功能所以颇接近std::vector。不过它本质上还是链式容器，不像std::vector那样提供连续的元素存储，能够使用push_front()在前段添加元素，将其看作是附加了部分std::vector功能的std::list比较合适。

示范随机访问索引用法的代码如下：

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/random_access_index.hpp>
using namespace boost::multi_index;                   //打开名字空间

int main()
```

```
{
    typedef multi_index_container<int,
        indexed_by<random_access<> >                    //随机访问索引，没有标签
        > mic_t;

    using namespace boost::assign;
    mic_t mic1 = (list_of(2),3,5,7,11);                 //初始化容器

    assert(mic1[0] == 2);                               //使用 operator[]
    assert(mic1.at(2) == 5);                            //使用 at()

    mic1.erase(boost::next(mic1.begin(), 2));           //删除元素
    assert(mic1[2] == 7);

    mic_t mic2;
    mic2.splice(mic2.end(), mic1);                      //使用链表的接合操作
    assert(mic1.empty() && mic2.size() == 4);

    push_front(mic1)(8), 10, 20, 16;                    //调用 push_front()

    mic1.sort();                                        //内部的排序算法
    mic2.sort();

    mic1.merge(mic2);                                   //合并两个链表

    for (auto iter = mic1.rbegin();                     //逆序输出元素
        iter != mic1.rend(); ++iter)
    {
        cout << *iter << ",";
    }
}
```

代码的运行结果如下：

```
20,16,11,10,8,7,3,2
```

虽然random_access_index提供了随机访问元素的功能，但因为它的接口是常量性的，所以很多需要使用迭代器赋值操作的标准算法（如排序算法、替换算法）都无法使用：

```
std::sort(mic1.begin(), mic1.end());                    //编译错误
std::random_shuffle(mic1.begin(), mic1.end());          //编译错误
std::replace(mic1.begin(), mic1.end(), 2, 222);         //编译错误
```

10.7 有序索引

有序索引基于键提取器对元素进行排序，使用红黑树结构提供类似于set的有序集合访问接口。

有序索引位于头文件<boost/multi_index/ordered_index.hpp>。

10.7.1 索引说明

有序索引的索引说明包括ordered_unique和ordered_non_unique，前者不允许重复键（单键）而后者允许重复（多键），两者的声明和接口均相同，故下面仅以ordered_unique为例。

ordered_unique的类摘要如下：

```
template<typename Arg1,typename Arg2=mpl::na,typename Arg3=mpl::na>
struct ordered_unique
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::ordered_index<...> type;
  };
};
```

ordered_unique有三个模板参数，但因为使用了模板元编程技术，所以最少提供一个模板参数就可以工作。

- 第一个参数可以是标签或者键提取器，必须提供。
- 如果第一个参数是标签，那么第二个参数必须是键提取器。
- 最后一个参数是比较谓词对象，缺省是 std::less<typename KeyFromValue::result_type>，即对键提取器的小于比较。

10.7.2 类摘要

有序索引使用的类是detail::ordered_index，接口类似于std::set，类摘要如下：

```
template<typename KeyFromValue,typename Compare,...>
class ordered_index
{
```

```
public:
  typedef some_define          key_type;
  typedef some_define          value_type;
  typedef some_define          iterator;
  typedef iterator             const_iterator;
  ...                          //其他类型定义

  //赋值操作
  ordered_index&               operator=(const ordered_index& x);

  //迭代器操作
  iterator                     begin();
  iterator                     end();
  iterator                     iterator_to(const value_type& x);

  //元素访问
  std::pair<iterator,bool>     insert(const value_type& x);
  std::pair<iterator,bool>     insert(iterator position,const value_type& x);
  iterator                     erase(iterator position);

  //有序相关操作
  iterator                     find(const CompatibleKey& x)const;
  iterator                     find(const CompatibleKey& x,
                                   const CompatibleCompare& comp)const;

  size_type                    count(const CompatibleKey& x)const;
  size_type                    count(const CompatibleKey& x,
                                    const CompatibleCompare& comp)const;

  iterator                     lower_bound(const CompatibleKey& x)const;
  iterator                     lower_bound(const CompatibleKey& x,
                                     const CompatibleCompare& comp)const;

  iterator                     upper_bound(const CompatibleKey& x)const;
  iterator                     upper_bound(const CompatibleKey& x,
                                     const CompatibleCompare& comp)const;

  std::pair<iterator,iterator> equal_range(const CompatibleKey& x)const;
  std::pair<iterator,iterator> equal_range(const CompatibleKey& x,
                                   const CompatibleCompare& comp)const;
```

```
  std::pair<iterator,iterator> range(LowerBounder lower,
                                     UpperBounder upper)const;

  ...                                       //各种比较操作符的定义
};
```

ordered_index的接口与std::set基本相同，但find()、count()、lower_bound()等涉及键比较的函数都有两种重载形式，其原理与侵入式容器的使用键操作值（参见 9.5.6 节）的方式类似，可以定义一个比较谓词函数对象CompatibleCompare使用兼容键比较，避免了构造大对象的成本。

ordered_index的另一个特殊函数是range()，它使用两个函数对象简化了取上下界的操作（lower_bound和upper_bound)。

10.7.3 基本用法

有序索引的基本用法很简单，与std::set没有太多区别，当然还要注意不能使用迭代器来修改元素，因为多索引容器的元素是不可变的。

下面的代码示范了单键有序索引的简单用法：

```
typedef multi_index_container<int,                  //元素类型为 int
  indexed_by<                                       //索引说明列表
      ordered_unique<identity<int>>                 //有序单键索引，使用 identity
  >
> mic_t1;

mic_t1 mic1;
insert(mic1)(2), 3, 5, 7, 11;                       //插入不允许重复的元素

assert(mic1.size() == 5);
assert(!mic1.insert(3).second);                     //重复元素无法插入
assert(mic1.find(7) != mic1.end());
```

接下来的代码示范了多键有序索引的用法，读者需注意键提取器的使用：

```
typedef multi_index_container<string,          //元素类型为 string
  indexed_by<                                  //索引说明列表
      ordered_non_unique<                      //有序多键索引，使用字符串长度作为键
          BOOST_MULTI_INDEX_CONST_MEM_FUN(string, size_t, size)>
  >
```

```
> mic_t2;

mic_t2 mic2;
insert(mic2)("111")("22")("333")("4444");     //注意，有重复元素
assert(mic2.count(3) == 2);                    //两个重复元素

//使用 equal_range()输出重复的元素，不能用 for
BOOST_FOREACH(auto& str, mic2.equal_range(3))
{ cout << str << ",";}
```

这里的有序索引定义略微复杂一些。虽然元素类型是字符串string，但我们并没有使用字符串本身作为键（identity<string>），而是使用字符串长度作为键，键提取器是const_mem_fun，调用了string的const成员函数size()。

因为键的特殊性，虽然我们向容器中插入了四个字面值不同的字符串，但实际上"111"和"333"这两个元素是重复的——它们的长度都是 3，所以调用count()和equal_range()能够看到两个重复的元素。

10.7.4 高级用法

本小节使用person类作为容器的元素，讨论有序索引的两种高级用法：兼容键比较和获得范围区间。

兼容键比较

所谓兼容键（compatible key），是指不同于索引说明中键本身的一个类型，但它的比较效果与键相同。例如，对于person类来说，identity<person>定义的键类型为person，但它的比较操作符operator<使用的是类型为int的m_id成员变量，所以int就是它的兼容键。很显然，构造一个int类型的成本要比构造一个person类型的成本低很多，在效率上自然会有很大提升。

为了使用兼容键比较函数，我们需要为person类自定义一个与int类型比较的谓词，比较关系应与容器的比较谓词一致（在这里是小于关系）：

```
struct compare_by_id                          //比较 person 对象和 id
{
  typedef bool result_type;
  bool operator()(const person& p, int id) const
  {   return p.m_id < id;}
  bool operator()(int id, const person& p) const
  {   return id < p.m_id ;}
};
```

这样我们就可以仅使用兼容键类型而不必使用整个元素（键类型）来执行比较操作了，提高了效率，示范代码如下：

```
typedef multi_index_container<person,
    indexed_by<
        ordered_unique<identity<person>>          //有序单键索引
    >
> mic_t;

mic_t mic;
insert(mic)
    (person(2, "agent", "smith"))                 //插入四个元素
    (person(20, "lee", "someone"))
    (person(1, "anderson", "neo"))
    (person(10, "lee", "bruce"));

//构造一个大对象执行比较操作，成本很高
assert(mic.count(person(1, "abc","xby")) == 1);

//使用自定义的兼容键比较谓词
assert(mic.count(1, compare_by_id()) == 1);
assert(mic.find(10, compare_by_id()) != mic.end());
```

获取范围区间

成员函数range()是一个模板函数，它的参数（LowerBounder和UpperBounder）是两个以键为参数的谓词函数或函数对象，用于确定区间的下界和上界，相当于a < x && x < b，比直接使用lower_bound()和upper_bound()要方便直观得多。

例如，我们为person的id定义一个 2 <= p.m_id < 20 的左闭右开区间，使用lower_bound()和upper_bound()的方法如下：

```
...                                              //多索引容器的声明和数据插入同前
//获得 id>=2 的下界
mic_t::iterator l = mic.lower_bound(2 , compare_by_id());
//获得 id>=20 的下界，即<20 的上界
mic_t::iterator u = mic.lower_bound(20, compare_by_id());

//foreach 循环，使用 make_pair 构造一个迭代器区间输出元素
BOOST_FOREACH(const person& p, std::make_pair(l, u))
{
```

```
  cout << p.m_id << ":"                     //输出两个元素
      << nameof(p) << endl;                 //2:agent smith 和 10:lee bruce
}
```

这段代码中的两个lower_bound()很令人迷惑，即使是有经验的C++程序员也很难把握区间的端点条件，很容易出错。

使用有序索引的range()函数就简单多了，所需的上下界函数对象可简单清晰地用以下代码实现：

```
struct lower_bounder                        //下界函数对象，定义 p >= 2
{
  typedef bool result_type;
  bool operator()(const person& p)
  {   return p.m_id >= 2;}
};
struct upper_bounder                        //上界函数对象，定义 p< 20
{
  typedef bool result_type;
  bool operator()(const person& p)
  {   return p.m_id < 20;}
};
```

然后调用range()函数就可以简单且精确地获取这个区间：

```
BOOST_FOREACH(const person& p,                       //使用 boost.foreach 循环
    mic.range(lower_bounder(), upper_bounder()))
```

如果我们经常执行获取区间的操作，那么编写大量类似的上下界谓词会很烦琐，这时我们可以使用C++11/14 提供的语言级别的lambda表达式，就地生成匿名函数对象：

```
BOOST_FOREACH(const person& p,                       //使用 boost.foreach 循环
  mic.range(
    [](const person& p){ return p.m_id >= 2;},       //C++11/14 的 lambda
    [](const person& p){ return p.m_id <  20;}
  ))
```

为了支持无限制上界和无限制下界，有序索引还定义了一个特别的谓词函数unbounded，它可以用在区间的任意一个端点，表示该端点无限制，例如：

```
mic.range(lower_bounder(), unbounded)                //p.m_id >= 2
mic.range(unbounded, upper_bounder(),)               //p.m_id <  20
mic.range(unbounded, unbounded)                      //所有元素
```

10.8 散列索引

散列索引基于键提取器对（元素的）键进行散列，提供类似于std::unordered_set的无序集合接口。

散列索引位于头文件<boost/multi_index/hashed_index.hpp>。

10.8.1 索引说明

散列索引的索引说明包括hashed_unique和hashed_non_unique，前者不允许重复键而后者允许重复键，两者的声明和接口均相同，故下面仅以hashed_unique为例。

hashed_unique的类摘要如下：

```
template<typename Arg1,typename Arg2,typename Arg3,typename Arg4>
struct hashed_unique
{
  template<typename SuperMeta>
  struct index_class
  {
    typedef detail::hashed_index<...> type;
  };
};
```

hashed_unique有四个模板参数，使用了与ordered_unique相同的模板元编程技术，最少提供一个模板参数就可以工作。

- 第一个参数可以是标签或者键提取器，必须提供。
- 如果第一个参数是标签，那么第二个参数必须是键提取器。
- 键提取器后的参数是散列函数对象，缺省是 boost::hash<typename KeyFromValue::result_type>。
- 最后一个参数是相等比较谓词对象，缺省是 std::equal_to<typename KeyFromValue::result_type>。

10.8.2 类摘要

散列索引使用的类是hashed_index，接口类似于std::unordered_set，类摘要如下：

```
template<typename KeyFromValue,typename Hash,typename Pred,...>
class hashed_index
{
public:
  typedef some_define          key_type;
  typedef some_define          value_type;
  typedef some_define          iterator;
  typedef iterator             const_iterator;
  ...                          //其他类型的定义

  //赋值操作
  hashed_index&                operator=(const hashed_index& x);

  //迭代器操作
  iterator                     begin();
  iterator                     end();
  iterator                     iterator_to(const value_type& x);

  //元素访问
  iterator                     find(const CompatibleKey& x)const;
  iterator                     find(const CompatibleKey& x,
                                    const CompatibleHash& hash,
                                    const CompatiblePred& eq)const;

  size_type                    count(const CompatibleKey& x)const;
  size_type                    count(const CompatibleKey& x,
                                     const CompatibleHash& hash,
                                     const CompatiblePred& eq)const;

  std::pair<iterator,iterator> equal_range(const CompatibleKey& x)const;
  std::pair<iterator,iterator> equal_range(const CompatibleKey& x,
                                           const CompatibleHash& hash,
                                           const CompatiblePred& eq)const;

  ...                          //各种比较操作符的定义
};
```

hashed_index与unordered_set的接口类似，因为是无序的，所以不提供成员函数lower_bound()、upper_bound()和range()。此外hashed_index还有一些与散列容器相关的特殊成员函数，如桶数量、负载因子等，本书从略。

10.8.3 用法

散列索引用起来就像是std::unordered_set，完全遵循C++标准。示范散列索引用法的

代码如下：

```
typedef multi_index_container<person,
  indexed_by<
      hashed_unique<mi::identity<person>>         //散列单键索引
      >
  > mic_t;

mic_t mic;
insert(mic)
  (person(2, "agent", "smith"))                   //插入元素
  (person(1, "anderson", "neo"))
  (person(10, "lee", "bruce"));

assert(mic.size() == 3);
assert(mic.find(person(1, "anderson", "neo"))!= mic.end());
```

散列索引同样可以使用兼容键来查找元素，但它需要使用两个函数对象来进行散列和相等比较，比有序索引要多做一些工作。

由于person类散列使用了m_fname和m_lname，相等比较使用了m_id，涉及的因素较多，所以我们使用一个boost::tuple来定义兼容键：

```
//用 tuple 组合三个类型定义兼容键
typedef boost::tuple<int, string, string> hash_key_t;

//定义散列函数对象，算法与 person 的一致
struct hash_func
{
  typedef size_t result_type;
  size_t operator()(const hash_key_t& k) const        //必须是 const
  {
      size_t seed = 2016;
      hash_combine(seed, k.get<1>());
      hash_combine(seed, k.get<2>());
      return seed;
  }
};

//定义相等比较函数对象
```

```
struct equal_func
{
  typedef bool result_type;
  bool operator()(const hash_key_t& k, const person& p) const
  {   return k.get<0>() == p.m_id;}
  bool operator()(const person& p, const hash_key_t& k) const
  {   return k.get<0>() == p.m_id;}
};
```

这样我们就可以使用兼容键在散列索引中查找元素了：

```
assert(mic.count(make_tuple(1, "anderson", "neo"),
      hash_func(), equal_func()) == 1);
assert(mic.find(make_tuple(10, "lee", "bruce"),
      hash_func(), equal_func())!= mic.end());
```

10.9 修改元素

multi_index库中所有索引的迭代器接口都是常量性的，不允许用户直接修改，这是因为一个索引的元素变动操作可能会导致其他索引不一致，破坏整个多索引容器的结构。

但并不是说多索引容器里的元素就是不可修改的，multi_index为此使用了另外的机制：所有的索引都提供两个专门用于修改元素的成员函数，replace()和modify()，有序索引和散列索引还有一个特别的修改键的成员函数modify_key()，使用这些操作可以保持多索引容器的状态不被破坏。

10.9.1 替换元素

成员函数replace()可以替换一个有效迭代器位置上的元素的值，其他索引保持同步更新。multi_index库为replace()操作提供了强异常安全保证，即使发生异常，多索引容器也会保持不变，所有索引及相关的迭代器和引用也保持不变。

replace()的声明如下：

```
bool replace(iterator position,const value_type& x);
```

迭代器通常可以使用find()算法或者find()成员函数获取，如果使用iterator_to()则需要小心，使用元素的等价拷贝获得的是一个无效迭代器，用于replace()会产生运行时异常。

示范replace()用法的代码如下：

```
typedef multi_index_container<person,                    //定义一个三索引的容器
  indexed_by<
      ordered_unique<identity<person>>,                  //单键有序索引
      ordered_non_unique<                                //有序多键索引
          member<person, string, &person::m_fname>>,     //使用成员变量
      hashed_unique<identity<person>>                    //散列单键索引
  >
> mic_t;

mic_t mic;
insert(mic)                                              //插入四个元素
  (person(2, "agent", "smith"))
  (person(20, "lee", "someone"))
  (person(1, "anderson", "neo"))
  (person(10, "lee", "bruce"));

auto pos = mic.find(20, compare_by_id());                //查找 id 为 20 的元素
assert(pos != mic.end());

mic.replace(pos, person(20, "lee", "long"));             //替换这个元素
assert(pos->m_lname == "long");
```

执行替换操作时可以任意修改元素的值，元素的键也可以被修改，例如：

```
mic.replace(pos, person(15, "lee", "long"));             //修改元素的键
```

但如果修改后的元素与索引约束发生冲突则会导致替换失败，函数返回false：

```
assert(!mic.replace(pos, person(2, "lee", "someone")));  //索引 0 冲突
assert(!mic.replace(pos, person(10, "lee", "bruce")));   //索引 2 冲突
```

10.9.2 修改元素

replace()成员函数提供了强异常安全保证，但操作的成本较高，因为在替换时我们必须构造一个完整的临时元素对象，但很多时候这是不必要的。

modify()是另外一种修改元素的方法，它使用一个被称为修改器的函数或函数对象，接受一个元素的引用作为参数，可以只变动元素的某个成分，是轻量级的修改元素的方法。

直接修改

modify()的声明如下：

```
template<typename Modifier>
```

```
bool modify(iterator position, Modifier mod);
```

modify()有两个参数，第一个是要修改的迭代器位置，含义与replace()相同，第二个是修改器对象，它执行元素的修改操作。

例如，我们想要修改person的各个成员变量，可以使用C++11/14 的lambda表达式在modify()里直接编写修改器：

```
mic.modify(pos,                                          //修改m_id
        [](person& p){ p.m_id = 15; });                  //使用 lambda 表达式
assert(pos->m_id == 15);

mic.modify(pos,                                          //修改m_fname
        [](person& p){ p.m_fname = "mike";});            //使用 lambda 表达式
assert(pos->m_fname == "mike");

mic.modify(pos,                                          //修改m_lname
        [](person& p){ p.m_lname = "david";});           //使用 lambda 表达式
assert(pos->m_lname == "david");
```

回滚修改

modify()避免了构造临时对象的成本，执行效率高，但也有不如replace()的地方：它不能保证操作的安全性，如果元素修改后与索引的约束发生冲突，那么修改将失败，而元素会被删除！

请看下面的代码：

```
//找到 id 为 20 的元素
auto pos = mic.find(20, compare_by_id());

//将 id 修改为 1，与索引 0 的约束(ordered_unique)发生冲突，修改失败，元素被删除
assert(!mic.modify(pos, [](person& p){p.m_id = 1;}));

//此时元素已被删除，无法找到
assert(mic.size() == 3);
assert(mic.find(20, compare_by_id()) == mic.end());
```

为了避免这样的“灾难”发生，modify()提供了一种类似于数据库“回滚”机制(rollback)的重载形式，允许用户使用一个“回滚”函数或函数对象在修改失败时恢复原有的值，这种形式的modify()函数的原型如下：

```
template<typename Modifier,typename Rollback>
```

```
bool modify(iterator position, Modifier mod, Rollback back);
```

回滚机制的modify()的用法如下：

```
assert(!mic.modify(pos,
  [](person& p){p.m_id = 1;},                    //将id修改为1，发生冲突
  [](person& p){p.m_id = 9999;}));              //回滚操作，把id改为9999，不是很好
assert(mic.size() == 4);
assert(mic.find(9999, compare_by_id()) != mic.end());
```

上面的代码不是解决问题的最佳答案，因为固定的id还有可能造成冲突，但这时索引却一无所知，可能会导致索引混乱（读者可以试着把 9999 改为已有的id试验一下）。正确的做法是先使用迭代器获得要修改的原值，然后把原值作为回滚的参数：

```
auto tmp = pos->m_id;                           //修改前先保存原值
assert(!mic.modify(pos,
  [](person& p){p.m_id = 1;},                   //修改器修改
  [&](person& p){p.m_id = tmp;}));              //如果修改失败那么回滚恢复原值
```

10.9.3 修改键

有序索引和散列索引拥有一个特别的修改函数modify_key()，它可以直接修改索引使用的键而不是元素本身，是modify()的特化版本。使用modify_key()要求索引的键提取器必须是可写的。

modify_key()的声明如下：

```
template<typename Modifier>
bool modify_key(iterator position,Modifier mod);

template<typename Modifier,typename Rollback>
bool modify_key(iterator position,Modifier mod,Rollback back);
```

modify_key()的修改器操作的不是元素本身的类型，而是键类型，所以编写lambda表达式时参数要使用键类型：

```
auto& index = mic.get<1>();                           //获取索引
auto pos = index.find("agent");                       //查找元素

index.modify_key(pos,                                 //修改键
      [](string& str){  str = "virus";});             //注意 lambda 表达式的参数
```

```
assert(pos->m_fname == "virus");
```

同样地，如果modify_key()修改键后导致索引冲突，那么元素也会被立即删除，为了安全起见，应该使用回滚操作：

```
auto tmp = pos->m_fname;                              //修改前先保存原值
index.modify_key(pos,
  [](string& str){   str = "virus";},                 //修改键
  [&](string& str){  str = tmp;});                    //失败则恢复原值
```

10.10 多索引容器

经过了前面数节对键提取器和索引的研究，我们终于来到了真正的多索引容器multi_index_container面前，下面就来看看multi_index_container的真面目。

10.10.1 类摘要

10.3.6节已经列出了multi_index_container的前置声明，下面是它的类摘要：

```
template<
  typename Value,                                     //元素类型
  typename IndexSpecifierList=indexed_by<...>,        //索引说明列表
  typename Allocator=std::allocator<Value> >          //内存分配器
class multi_index_container:
    public detail::multi_index_base_type<...>::type
{
public:

  //构造函数、赋值操作
  multi_index_container(InputIterator first,InputIterator last);
  multi_index_container(const multi_index_container& x);
  multi_index_container& operator=(const multi_index_container& x);

  //索引类型定义
  template<int N> struct          nth_index;
  template<typename Tag>  struct  index;

  //获取索引操作
  template<int N>
  typename nth_index<N>::type&    get();
```

```
    template<typename Tag>
    typename index<Tag>::type&      get()

    //投射操作
    template<int N,typename IteratorType>
    typename nth_index<N>::type::iterator  project(IteratorType it);
    template<typename Tag,typename IteratorType>
    typename index<Tag>::type::iterator    project(IteratorType it);
};
```

multi_index_container本质上是一个索引的容器，它本身只负责管理索引，各个索引负责元素的管理。

为了方便使用容器，multi_index_container使用模板元编程技术实现了从第一个索引继承（public detail::multi_index_base_type<...>::type），获得了它的所有接口和能力，因此我们可以直接以容器的方式使用第一个索引。

get<*N*>()是multi_index_container最重要的成员函数，它有两种重载形式，分别可以使用整数序号或者类型标签来获取索引，因此索引也有两个类型，分别是nth_index<*N*>和index<tag>。这两个类型实际上是元函数，需要使用::type的形式才能获得真正的索引类型，不过通常我们可以使用auto/decltype来简单地避免这个类型声明的问题。

project<*N*>()用来在多个不同的索引之间转换迭代器，可以把一个索引的迭代器投射到另一个索引的迭代器中，而这两个迭代器指向的是同一个元素，这可以方便我们以一个索引查找元素再改用另一个索引操作元素。

multi_index_container是可序列化的，如果不需要使用序列化能力，那么可以定义宏BOOST_MULTI_INDEX_DISABLE_SERIALIZATION，这样将禁用序列化代码，加快编译速度。

10.10.2 用法

multi_index_container真正属于自己的操作不是很多，因为它主要负责管理索引，使用get<*N*>()获得索引后由索引来操作元素。

作为示范，这里我们综合使用之前介绍的所有键提取器和索引，定义一个持有八个索引的容器。对于这样的一个复杂多索引容器，如果直接写出其索引说明列表是非常庞大的，所以我们必须使用typedef进行简化。

首先是序列索引和随机访问索引的定义，需要使用索引标签：

```
typedef sequenced<tag<int, struct seq_idx> >          idx_sf0;
```

```
typedef random_access<tag<struct rnd_idx, string> > idx_sf1;
```

然后是三个有序索引，分别使用了identity、member和const_mem_fun键提取器，最后一个ordered_non_unique索引还使用std::greater谓词来改变排序准则：

```
typedef ordered_unique<mi::identity<person>> idx_sf2;     //有序单键索引
typedef ordered_non_unique<                                //有序多键索引
  BOOST_MULTI_INDEX_MEMBER(person, string, m_fname) > idx_sf3;

typedef ordered_unique<                                    //有序单键索引
  BOOST_MULTI_INDEX_CONST_MEM_FUN(
      person, const string&, first_name),
  std::greater<const string> >   idx_sf4;                  //使用大于比较排序
```

最后是三个散列索引，使用了member、global_fun和自定义键提取器：

```
typedef hashed_unique<                                     //散列单键索引
  BOOST_MULTI_INDEX_MEMBER(person, string, m_lname) > idx_sf5;

typedef hashed_non_unique<                                 //散列多键索引
  global_fun<const person&, string, &nameof>>          idx_sf6;

typedef hashed_unique< person_name> idx_sf7;   //散列多键索引，使用自定义键提取器
```

读者需注意：在以上八个索引中，我们没有使用mem_fun键提取器，这是因为容器存储的是元素本身而不是指针，如果使用mem_fun则会因为从常量中无法提取键而导致编译失败。

有了这些索引的类型定义，多索引容器的定义就简单多了：

```
typedef multi_index_container<person,                 //定义多索引的容器
  indexed_by<
      idx_sf0, idx_sf1,                               //序列索引和随机访问索引
      idx_sf2, idx_sf3, idx_sf4,                      //有序索引
      idx_sf5, idx_sf6, idx_sf7>                      //散列索引
> mic_t;
```

对于这个多索引容器，我们可以使用以下八个完全不同的接口来访问它：

- 像 std::list 一样的双向链表。
- 像 std::vector 一样的随机访问序列。
- 基于 m_id 从小到大排序的不允许重复的有序集合。

- 基于 m_fname 从小到大排序的允许重复的有序集合。
- 基于 m_fname 从大到小排序的不允许重复的有序集合。
- 基于 m_lname 的不允许重复的无序集合。
- 基于 m_fname+m_lname 的允许重复的无序集合。
- 基于 m_fname+m_lname 的不允许重复的无序集合。

由于这个容器使用的索引比较多，所以相互之间的制约也就错综复杂，单纯的读操作不会有什么影响，如果执行插入、修改等涉及元素变动的操作就必须小心，不能违反任何一个索引的约束。例如，idx_sf3 允许m_fname重复，但idx_sf4 却不允许m_fname重复，这样实际上是把idx_sf3 的non_unique效果“抵消”了。代码如下所示：

```
mic_t mic ;

using namespace boost::assign;
push_back(mic)                                      //插入四个元素
  (person(2, "agent", "smith"))                     //插入成功
  (person(20, "lee", "someone"))                    //插入成功
  (person(1, "anderson", "neo"))                    //插入成功
  (person(10, "lee", "bruce"));                     //因为m_fname 重复所以插入失败

assert(mic.size() == 3);                            //最终只插入了三个元素
```

示范成员函数project()用法的代码如下：

```
auto& idx1 = mic.get<rnd_idx>();                    //获取随机访问索引
auto& idx2 = mic.get<2>();                          //获取有序索引

auto pos = idx2.find(1, compare_by_id());           //在有序索引中查找元素
auto pos2 = mic.project<string>(pos);               //投射到随机访问索引

assert(pos2 == idx1.iterator_to(idx1[ 2]));         //使用 iterator_to()
```

在这段代码中，我们使用有序索引查找m_id为 1 的元素，然后把迭代器投射到随机访问索引上，因为我们已经知道这个元素的位置，所以可直接使用operator[ ] 获取元素的引用，再用iterator_to()转换得到迭代器，验证了投射的正确性。

mem_fun键提取器可以用在容纳指针或智能指针的多索引容器里，例如：

```
typedef shared_ptr<person> person_ptr;              //存储智能指针
```

```
typedef hashed_unique<                          //定义一个新的散列单键索引
  BOOST_MULTI_INDEX_MEM_FUN(                    //使用mem_fun
  person, string&, last_name)
  > idx_sf8;
typedef multi_index_container<person_ptr,       //定义多索引容器
  indexed_by<idx_sf8> > mic_t;

mic_t mic ;
mic.insert(make_shared<person>(2, "agent", "smith"));
```

10.11 组合索引键

类似于数据库里的联合主键概念，有的时候我们仅使用一个键对元素排序可能还不够，需要同时基于多个键来查找元素，这时我们就要用到multi_index库提供的组合索引键composite_key的功能。

composite_key可以把多个键提取器组合为一个新的键供索引使用，能够提供更多的灵活性，它位于头文件<boost/multi_index/composite_key.hpp>。

10.11.1 类摘要

composite_key是一个只读键提取器，类摘要如下：

```
template<typename Value,typename KeyFromValue0,...>
struct composite_key : private tuple<...>
{
  typedef Value                                   value_type;
  typedef composite_key_result<composite_key> result_type;

  result_type operator()(const value_type& x)const;
  ...                    //其他operator()定义
};
```

composite_key基于boost.tuple实现对多个键提取器的组合，它的第一个模板参数是元素类型Value，可以跟着一个或多个组合的键提取器。这些键提取器可以是identity、member、const_mem_fun等任意的键提取器，但必须也以Value作为元素类型。当前multi_index库最多支持组合十个键提取器。

composite_key的operator()返回类型是composite_key_result，它是一个非常简单的struct，重载了标准比较谓词equal_to、less、greater和boost的hash操作，并且支持与tuple的比较操作，可以像基本类型一样使用。

composite_key_result的比较规则与tuple相同，依据字典序，即依据键的顺序逐个比较。

10.11.2 用法

composite_key可以用于有序索引和散列索引，由于它自身定义已经很复杂了，如果直接在multi_index_container中定义则会进一步增加多索引容器的复杂性，所以最好使用typedef，同时注意缩进格式。

我们先使用composite_key定义一个基于m_id和m_fname的组合索引键，注意元素类型使用了指针，这意味着我们要在容器中存储指针类型：

```
typedef composite_key<person*,                    //元素类型为指针
  BOOST_MULTI_INDEX_MEMBER(person, int, m_id),
  BOOST_MULTI_INDEX_MEMBER(person, string, m_fname)
> comp_key;
```

使用组合键的多索引容器的定义如下，元素类型与组合键相同，也是指针：

```
typedef multi_index_container<
  person*,                                        //元素类型为指针
  indexed_by<
      ordered_unique<comp_key>                    //使用组合键
  >
> mic_t;
```

为了避免手工删除指针的麻烦，我们可以使用指针容器来存储元素，把多索引容器当作指针容器的一个视图来使用：

```
ptr_vector<person> vec;
using namespace boost::assign;
ptr_push_back(vec)                                //使用 assign 库插入动态创建的元素
  (2, "agent", "smith")
  (1, "anderson", "neo")
  (1, "the one", "neo");                          //注意，id 有重复

mic_t mic;
for(auto& p : vec)                                //插入指针元素到多索引容器
{ mic.insert(&p);}
```

```
for(auto  p : mic)                          //顺序输出多索引容器内的元素
{  cout << p->m_id << ":" << p->m_fname << ",";}
```

请读者注意组合键的排序准则，这里是基于m_id和m_fname，与单纯的identity<person>等不同，只有m_id和m_fname都相同时才能判断为重复。

使用find()、count()等涉及组合键的查找操作时我们需要使用tuple来构造查找值，因为composite_key_result不能够直接创建：

```
assert(mic.count(make_tuple(1, "anderson")) == 1);
assert(mic.find(make_tuple(2, "agent")) != mic.end());
```

在有序索引中我们也可以仅顺序指定部分键值，这样索引将执行“模糊查找”，仅对指定的查找值执行比较：

```
assert(mic.count(make_tuple(1)) == 2);          //仅指定一个键
```

如果仅使用第一个键，那么multi_index库将允许我们不使用tuple，直接使用值：

```
assert(mic.count(1) == 2);                      //仅指定一个键，不必用 tuple
```

对于散列索引我们不能指定部分键值，因为散列必须基于所有的键。

10.11.3 辅助工具

单单把键组合起来可能还不够，我们有时还想对组合键做更多的定制工作，比如对一个键执行升序排序而对另一个键执行降序排序，multi_index库为此提供了多个操作composite_key_result的比较谓词和散列函数对象。

三个基本的函数对象可以通过组合基于单个键的比较谓词或散列函数对象来任意定制排序准则。

```
template<typename Pred0,...,typename Predn>
struct composite_key_equal_to;

template<typename Compare0,...,typename Comparen>
struct composite_key_compare;

template<typename Hash0,...,typename Hashn>
struct composite_key_hash;
```

例如，我们可以对person类的m_id升序而对m_fname降序，定制排序准则如下：

```
typedef composite_key_compare<
```

```
    std::less<int>,                                   //m_id 升序
    std::greater<string>                              //m_fname 降序
  >         comp_key_compare;
```

多索引容器的定义需要增加组合比较谓词的定义：

```
typedef multi_index_container<
  person*,
  indexed_by<
      ordered_unique<
          comp_key,                                   //组合键
          comp_key_compare>                           //组合比较谓词
  >
> mic_t;
```

为了方便组合比较谓词的使用，multi_index库又定义了四个简化的函数对象，使用标准库的equal_to、less、greater和boost.hash操作所有键：

```
template<typename CompositeKeyResult>
struct composite_key_result_equal_to;

template<typename CompositeKeyResult>
struct composite_key_result_less;

template<typename CompositeKeyResult>
struct composite_key_result_greater;

template<typename CompositeKeyResult>
struct composite_key_result_hash;
```

例如，对m_id和m_fname降序排列的定义如下：

```
typedef multi_index_container<
  person*,
  indexed_by<
      ordered_unique<
          comp_key,
          composite_key_result_greater<comp_key::result_type>
      >
  >
> mic_t;
```

注意composite_key_result_greater的用法，它需要使用一个composite_key_result作为模板参数，我们必须使用comp_key::result_type来获取这个类型，它实际上是composite_key_result<comp_key>。

10.12 总结

在本章中，我们讨论了Boost中的多索引容器库multi_index，它提供了一个可以同时持有多个访问接口的容器multi_index_container，很适合需要以多种检索方式操作大量数据的情形，可以显著地提高性能，通常要比单纯使用标准容器再搭配算法要好很多。

multi_index_container的用法比较复杂，它的核心是索引说明列表，可以用index_by结构组合六种索引说明和五种键提取器，从而定义任意的索引方式。虽然multi_index_container提供了极大的灵活性，但付出的成本是代码复杂难懂，对于不熟悉多索引容器的人来说维护较困难，使用它需要慎重考虑这些程序之外的成本。

因为多个索引之间的制约关系，保持各个索引的正确性是一项必要的工作。multi_index_container要求元素不能被随意修改，迭代器接口都是常量性的，这样特意的设计可以在很大程度上保持多索引容器的稳定。另一方面，multi_index_container又提供了replace()、modify()和modify_key()三个成员函数，可以使用函数、函数对象或者bind/lambda表达式安全地修改全部或部分元素的值。

multi_index_container可以很好地替代标准容器vector、list、set和multiset，但替代映射容器map和multimap则比较麻烦，必须手工定义一个pair类作为容器的元素。这时我们可以考虑使用boost.bimap，它是一个基于multi_index库实现的双向映射容器，具有类似于map的接口，同时又有multi_index_container的一些特性（在推荐书目[3]中有介绍）。

本章并没有完全覆盖multi_index的所有内容，也没有对各种索引的时间复杂度进行分析，读者可以在今后的实际工作中深入研究multi_index的更多用法。

第11章

流处理

流处理库（iostream）是 C++“出生”时即搭配的“标准库”，历史悠久，用于为 C++提供基于流式的输入输出功能（并不限于控制台）。很多人对 C++流处理的认识通常仅限于 cin、cout，把它当作 C 语言 scanf()、printf()的 C++等价物，作用也通常限于打印日志或调试语句等简单应用。在图形界面和应用程序框架“泛滥成灾”的今天，iostream 更是几乎被打入了“冷宫”，鲜有程序员把关注的目光投在它身上。

boost.iostreams 库重新挖掘了这颗“沉睡的珠宝”，令流处理再度回到大众的视野。它基于标准库的 I/O 流框架提供了富有弹性且易于使用的流式处理机制，使 C++处理数据更加简单、方便和高效。

通过本章的讨论，希望读者能够重新审视 iostream，给予它应有的位置，而不是仅仅把它当作控制台的输入输出。

11.1 概述

本节将简要介绍标准库的流处理和 boost.iostreams 库的基本情况，不以精确描述或评判为目的，仅使读者对流处理有个大致的了解。

11.1.1 标准库的流处理

C++将输入输出视为“流”（stream），即数据在其中“流动”的序列，数据处理即在流动中完成操作。

流处理的数据通常是字符类型（char 或 wchar_t），但也可使用模板参数指定处理任意的

类型（就像迭代器），因此，流是一套完整的数据处理框架。

根据处理数据的方向，流可分为输入流和输出流两大类：输入是指从流中输入（类似于可读迭代器），输出是指向流输出（类似于可写迭代器）。

C++中 stream 的核心类体系结构如图 11-1 所示。

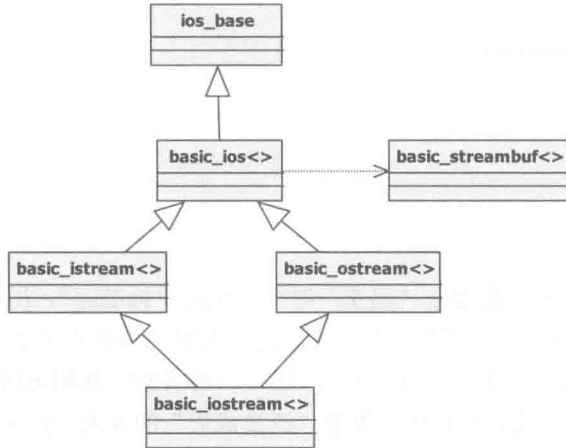


图 11-1 stream 核心类体系结构图

其中，ios_base 是流的基类，它的子类 basic_ios 使用模板参数规定了流处理的数据类型和特性（traits），并依赖 basic_streambuf 完成实际的读写操作。basic_istream 和 basic_ostream 分别从 basic_ios 虚继承，定义了输入输出流（读写流），最后的 basic_iostream 实现了既可读又可写的双向流。

我们常用的 cin、cout 的类型分别是 istream 和 ostream，而这两个类型则分别是 basic_istream 和 basic_ostream 的模板特化形式：

```
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
```

stream 重载了右移操作符（>>）和左移操作符（<<），将其重新定义为输入输出操作符（或读取/写入、提取/插入操作符），可以用很简单的方式从流中读取数据或者向流中写入数据。此外，stream 还提供了大量的成员函数和操控器，能够更细致地操作数据，例如 get()、put()、getline()、read()、write()等。

在后续的讨论中，我们将把精力集中于流的数据处理部分，而忽略数据的格式表述功能。流的数据类型使用最常见的 char，但个别例子会使用特殊的数据类型。

11.1.2 Boost 的流处理

boost.iostreams 库建立在标准库的 IO 流框架基础之上，它定义了 device、source、sink、filter 等新的流处理概念，构造了一套全新的、更易于使用和定制的流处理框架，几乎可以用流来处理任何数据，例如字符串处理、base64 编解码、压缩解压缩、加密解密等。

总的来说，iostreams 库包含如下的组成部分。

- 设备（device） ：流的起点 source（源设备）和流的终点 sink（接收设备）。
- 过滤器（filter）：读取时处理数据的输入过滤器和写入时处理数据的输出过滤器。
- 流（stream） ：用来连接设备和过滤器，让数据得以流动，实现类似标准流的功能。
- 过滤流 ：增强的流，由多个设备组成的链，数据流过链内设备完成过滤处理。
- 其他概念 ：设备链（chain）、管道（pipe）、视图（view）等高级概念。
- 预定义工具类 ：基于设备、过滤器等概念实现的工具，方便库用户的使用。
- 流操作函数 ：功能与标准库相近，但更加强大，其中最重要的是 copy()。

iostreams 库基本上是一个由头文件组成的库[①]，位于目录<boost/iostreams/>下，大部分功能不需要编译就可以使用，头文件组织结构如下所述。

- <boost/iostreams/> ：库的核心功能，如流、各概念的实现、操作函数等。
- <boost/iostreams/device/> ：预定义的设备类。
- <boost/iostreams/filter/> ：预定义的过滤器类。
- <boost/iostreams/detail/> ：库的实现细节，库用户通常无须关心。

iostreams 库需要编译后才能使用的功能包括 zlib/gzip/bzip2 格式的压缩解压缩等，需要安装 zlib、bzip2 库，在 Linux 里的安装命令可以是：

```
apt-get install libz-dev libbz2-dev      #Ubuntu 安装 zlib、bzip2
yum install zlib-devel bzip2-devel       #CentOS 安装 zlib、bzip2
```

① 由于 iostreams 库较长时间没有更新，在编译时会报出一些警告（使用了声明为废弃的 type_traits 头文件），但并不影响使用。

本章接下来的内容可分为两部分，11.2 节～11.7 节将先研究 iostreams 库的基本使用方法，11.7 节之后再研究更进一步的定制功能。

为方便书写代码，本章假设有如下的名字空间定义：

```
namespace io = boost::iostreams;          //iostreams 名字空间的别名
using namespace io;                       //所有 iostreams 功能均位于此名字空间内
```

11.2 入门示例

本节将概览 iostreams 库的既有类和函数，初步了解流处理的各种基本概念和使用方法。

我们从两个示例程序开始，它们演示了 iostreams 一些组件的基本用法，可以让我们对流处理有初步的了解。

11.2.1 示例 1

第一个示例程序比较简单，使用了类似标准流的功能：

```
#include <boost/iostreams/stream.hpp>           //基本流处理所需的头文件
#include <boost/iostreams/device/array.hpp>     //预定义的数组设备头文件

namespace io = boost::iostreams;                //iostreams 名字空间的别名
using namespace io;                             //打开名字空间

int main()
{
  char str[] = "123";                           //字符数组
  array_source asrc(str, str + 3);              //一个数组源设备，注意构造函数
  stream<array_source> in(asrc);                //搭配源设备定义输入流

  char c1, c2, c3;
  in >> c1 >> c2;                               //使用读取操作符从流中读取数据
  assert(c1 == '1' && c2 == '2');

  in.get(c3);                                   //可以调用标准流的成员函数
  assert(c3 == '3' && in.good());               //查看流的状态，正常
  assert(in.get(c3).eof());                     //流已经结束
```

```
    char str2[10];                                 //另一个字符数组
    array_sink asnk(str2);                         //定义接收设备，支持直接传入数组
    stream<array_sink> out(asnk);                  //搭配接收设备定义输出流

    out << 'a' << 'b' << 'c';                      //使用写入操作符向流写入数据
    assert(str2[0] == 'a' && str2[2] == 'c');
}
```

这段代码用到了三个 iostreams 库的组件：array_source、array_sink 和 stream，用法看起来非常像标准流 cin/cout。

array_source 是 iostreams 库提供的一个设备（参见 11.4.2 节），它可以把一个字符缓冲区（字符数组）适配成一个源设备。有了源设备，stream 就可以用模板参数+构造函数的方式连接到这个设备，形成一个与标准流完全兼容的新的输入流。注意，因为流连接的设备是源设备，而源设备是可读不可写的，所以新流必然是一个输入（只读）流。接下来我们就可以像标准输入流 cin 一样使用 operator>>或者 get() 函数从流中也就是字符缓冲区中读取数据，当流被耗尽无数据可读时可以用成员函数 eof() 来检测。

同样地，array_sink 是 iostreams 库提供的另一个设备，它可以把一个字符缓冲区适配成一个接收设备，stream 连接 array_sink 后就成为了一个输出流，可以写入数据，对流的写入操作最终均会被字符缓冲区接收。

11.2.2 示例 2

下面我们再来看一个稍微复杂一些的例子，它演示了过滤流、管道和 iostreams 库中最重要的 io::copy() 算法的用法：

```
#include <boost/iostreams/stream.hpp>                //基本流处理所需的头文件
#include <boost/iostreams/device/array.hpp>          //预定义的数组设备头文件
#include <boost/iostreams/filtering_stream.hpp>      //过滤流头文件
#include <boost/iostreams/device/back_inserter.hpp>  //接收设备适配头文件
#include <boost/iostreams/copy.hpp>                  //io::copy 算法头文件
#include <boost/iostreams/filter/counter.hpp>        //计数过滤器头文件

namespace io = boost::iostreams;                     //iostreams 名字空间的别名
using namespace io;                                  //打开名字空间

int main()
{
```

```
    char arr[] = "12345678";                        //字符数组
    stream<array_source> in(arr, arr + 8);          //直接创建输入流

    string str;                                     //标准字符串类，也可以算是标准容器
    filtering_ostream out(                          //输出过滤流，需连接 sink 设备作为链结束
        counter() |                                 //预定义的计数过滤器
        io::back_inserter(str));                    //适配标准容器作为接收设备

    io::copy(in, out);                              //调用 io::copy() 算法
    assert(str.size() == 8);
    assert(str == arr);
    assert(out.component<counter>(0)->characters() == 8);  //获取计数结果
}
```

这段代码没有用源设备，而是直接把字符缓冲区传递给了 stream 的构造函数，创建了一个长度为 8 的输入流，写法更加简单（详细的原因可参见 11.6.1 节）。

接下来的过滤流是代码的核心功能所在：filtering_ostream 声明了一个用于输出（可写）的过滤流 out，与基本流不同的地方是它不仅可以连接接收设备，也可以连接过滤器，更可以把这些设备连成一个处理链。过滤流 out 的构造函数中传入了两个设备对象，中间用重载的 operator|连接。“|”被称为管道操作符，这与 UNIX 中的管道操作符的用法非常相似，数据可以从一个设备通过管道流向另一个设备（更详细的信息可参见 11.5.2 节）。

过滤流设备链中的第一个设备是 counter（参见 11.5.3 节），它是 iostreams 库预定义的一个过滤器，可以计算通过过滤器的字符数和行数。第二个设备使用了 io::back_inserter()函数（参见 11.4.3 节），它是一个接收设备生成器，可以把一个标准容器适配成一个接收设备。

最后我们调用 io::copy()算法，它与标准库的 copy()算法非常相似（参见 11.7 节）。标准库的 copy()操纵迭代器，把数据从一个可读迭代器拷贝到另一个可写迭代器，而 io::copy()则用来操作流，把数据从一个输入流拷贝到输出流。这样，数据就从输入流开始，先流过 counter 过滤器，然后再流入被适配的 string 容器，完成了流处理过程。[①]

代码的最后一行调用了过滤流的成员函数 component()，它可以返回流的设备链中的第 n 个设备，在这里就是第一个过滤器 counter。然后再调用 counter 的成员函数 characters() 获取字符数的统计结果。

① 流处理与迭代器处理存在很多相似的地方，进一步的讨论参见 11.11 节。

11.3 设备的特征

通过上一节的两个代码示例，我们初步了解了 iostreams 的工作机制和 iostreams 中的源设备、接收设备、过滤器和流的用法，然而，要让这些设备一起协同工作，它们必须要满足一定的要求，也就是我们在模板元编程领域中经常提到的 traits。

iostreams 库在头文件<boost/iostreams/traits.hpp>里定义了所有与 traits 相关的工具。

11.3.1 设备的字符类型

设备最基本的特征是它们能够处理的“字符类型”（char type），流中的所有设备协同工作的最低要求是它们处理的必须是同一种“字符类型”。

设备的“字符类型”可以使用元函数 char_type_of<T>::type 获取，它类似于标准库的 std::char_traits<Ch>::char_type。

注意：我们所说的“字符类型”，不一定必须是 char 或者 wchar_t——虽然在大多数情况下流都是处理真正的字符，但有的时候我们也可以处理其他的数据类型，例如 unsigned char 或者 int 等，这个时候整个流中的设备的“字符类型”必须是一致的，否则会导致编译失败。

例如，如果有一个处理 char 的源设备，那么随后的过滤器和接收设备也必须是 char 设备，如果使用 wchar_t 的设备（如 wsink）就会无法通过编译。

11.3.2 设备的模式

设备的另一个重要特征是它的模式（mode）。在 iostreams 库中，模式指的是设备的输入输出（读写）访问特征，与 C 语言的文件模式或新式迭代器的遍历概念比较接近。

最常用的模式有以下四种。

- 输入模式（input）：可以在一个字符序列上执行读操作，如 std::cin。
- 输出模式（output）：可以在一个字符序列上执行写操作，如 std::cout。
- 双向模式（bidirectional）：可以在两个不相关的字符序列上分别执行读写操作，如 std::iostream。

■ 可定位模式（seekable） ：可以在一个字符序列上执行读写操作，但可重定位操作的位置，如 std::fstream。

这四种模式的关系图如图 11-2 所示（摘自 Boost 文档）。

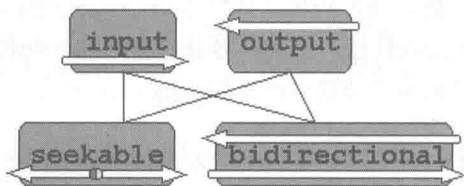


图 11-2 四种模式的关系图

另外还有四种模式，它们也是 iostreams 库的组成部分，但用途没有上面的四种那么广泛。

■ 可定位输入模式：在一个字符序列上执行读操作，可重定位操作位置，如 ifstream。

■ 可定位输出模式：在一个字符序列上执行写操作，可重定位操作位置，如 ofstream。

■ 双可定位模式 ：在一个字符序列上执行读写操作，读写操作可独立地重定位位置。

■ 双向可定位模式：在两个不相关的字符序列上执行读写操作，可重定位操作位置。

全部八种模式的关系图如图 11-3 所示（摘自 Boost 文档）。

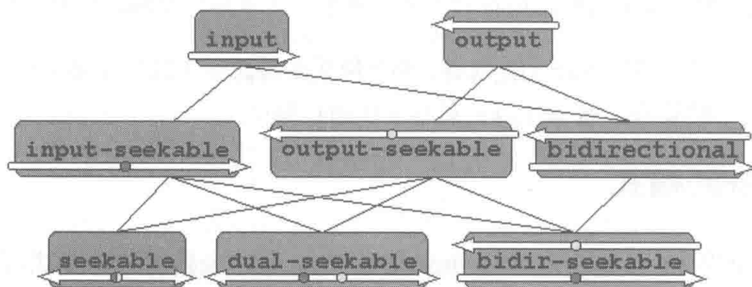


图 11-3 八种模式的关系图

使用元函数 mode_of<T>可获取设备的模式。

除了这八种模式，为了方便起见 iostreams 库还定义了一个“伪模式”dual_use，表示设备即可以用于输入也可以用于输出，但不能同时用于输入输出，所以也可以把它称为单向模式。

11.3.3 设备的分类

字符类型和访问模式是设备的两个最重要的特征，此外 iostreams 还定义了许多其他的特

征来更精确地描述设备，例如设备的类别（源设备、接收设备还是过滤器）、是否可关闭、是否可阻塞等，这些特征 tag 类都位于头文件<boost/iostreams/categories.hpp>。①

设备的分类（category）是除字符类型外所有设备特征的总和，它表现为设备的一个内部类型 category，使用多重继承的方式派生自各种 traits 类：

```
struct category                                    //一个可关闭的输入设备分类定义
    : input, device_tag, closable_tag              //包含了多个特征
    { };
```

因此 category 可以转型到其他特征 tag 类。

元函数 category_of<T>可以获取设备的分类信息，另有一个自由函数 get_category(const T&)包装了 category_of<T>，可以直接返回分类对象。

iostreams 库还提供了其他一些获取设备属性的元函数，我们将在后面逐步介绍。

11.4 设备

设备是 iostreams 库中的一个重要概念，在本节中，我们将首先研究基本概念，然后学习四个常用的预定义设备。

判断一个类是否是设备可以使用元函数 is_device<T>。

11.4.1 概述

设备（device）是 iostreams 库中最基础的概念，它位于头文件<boost/iostreams/concepts.hpp>，是一个可以对序列执行读或写操作的对象，其行为取决于它的模式。

device 是一个模板类，类摘要如下：

```
template<typename Mode, typename Ch = char>
struct device {
    typedef Ch      char_type;                         //字符类型
    struct          category                           //设备的分类，有缺省值
        : Mode, device_tag, closable_tag, localizable_tag{ };

    void            close();
```

① categories.hpp 里定义了大量用作标志的 tag 类，本书不能全部介绍，读者可通过阅读源代码进行研究。

```
    void            close(openmode);
};
```

device 是一个只被用于继承的概念类，它有两个模板参数，参数 Mode 决定了设备的模式，参数 Ch 是设备能够处理的字符类型，默认是窄字符 char。

device 有两个非常重要的内部类型定义，分别是 char_type 和 category，它们标记了设备的字符类型和分类信息，可以用相应的元函数 char_type_of 和 category_of 获取。

为了方便用户使用，device 还定义了若干特化，它们分别表示一些设备的子概念，例如：[1]

```
typedef device<input>    source;                    //char 源设备
typedef device<output>   sink;                      //char 接收设备
```

iostreams 提供了数个预定义的设备，下面我们将着重介绍数组设备、标准容器设备（适配器）、文件设备和空设备，其他设备读者可自行研究。

11.4.2 数组设备

数组设备位于头文件<boost/iostreams/device/array.hpp>，它提供了对内存字符序列的访问，可以把字符缓冲区适配成 seekable 设备。数组设备不负责缓冲区的内存管理，因此它通常搭配静态数组或者 std::vector 来使用。

iostreams 库提供三个数组设备：array_source、array_sink 和 array，分别是源设备、接收设备和可定位设备，我们之前已经初步学习了它们的使用方法。

类摘要

这三个数组设备实际上是 basic_array_source、basic_array_sink 和 basic_array 的 char 类型特化，它们仅在模式上存在不同，最后的实现都是同一个类 array_adapter。

array_adapter 的类摘要如下：

```
template<typename Mode, typename Ch>
class array_adapter {
public:
    typedef Ch                                   char_type;      //字符类型
    typedef std::pair<char_type*, char_type*>    pair_type;
    struct category                                              //设备的分类
```

① 这些子概念均有对应窄字符和宽字符的版本，为了叙述简单，本书忽略了宽字符版本，如 wsource、wsink 等，请读者见谅。

```
        : public Mode, public device_tag, public direct_tag
        { };

    array_adapter(char_type* begin, char_type* end);            //构造函数
    array_adapter(char_type* begin, std::size_t length);
    array_adapter(char_type (&ar)[ N] )
        : begin_(ar), end_(ar + N) { }

    pair_type           input_sequence();
    pair_type           output_sequence();
private:
    char_type*          begin_;                                 //数组的位置指针
    char_type*          end_;
};
```

解说

array_adapter 符合设备的概念，因此它的模板参数、字符类型和分类的含义均与 device 相同。

array_adapter 的构造函数是它的主要功能，能够用以下三种形式把数组适配成设备。

- 两个首尾指针（可为 const）分别标明数组的起点和终点。
- 数组起点（可为 const）和数组的长度。
- 直接使用数组的引用形式，数组的长度可使用模板推导。

另外两个成员函数 input_sequence() 和 output_sequence() 返回一个 pair 对象，标记了数组设备所能控制的序列范围，即成员变量 begin_ 和 end_。

用法

数组设备的用法我们之前已经看过了，下面再简单地举几个例子，重点是演示其构造函数：

```
char buf[] = "123";                                 //字符数组
array_source as1(buf, buf + 3);                     //数组设备 1
array_source as2(buf);                              //数组设备 2

assert(as1.input_sequence().first == buf);
assert(as1.input_sequence().second == buf + 3);
assert(as2.input_sequence().second == buf + 4);
```

请读者注意代码中两个数组设备的声明形式。as1 使用的是首末指针的方式，因此流的读取长度是 3，而 as2 使用的是数组引用方式，因此流的读取长度是数组的真正长度 4（含字符串的最后一个 NULL 结束标记）。

我们也可以改变字符类型定制操作其他数据类型的设备，这时就不能直接使用 array_source、array_sink 这些处理 char 的设备，而是要使用 basic_array_source 等模板类自行特化，例如：

```
int buf1[10] = {1, 2, 3}, buf2[10];                    //两个整型数组
basic_array_source<int> ai(buf1);                      //使用 int 类型的源设备
basic_array<int> ar(buf2);                             //使用 int 类型的可定位设备

stream<basic_array_source<int> > in(ai);               //输入流
stream<basic_array<int> > out(ar);                     //输出流

io::copy(in, out);                                     //调用 copy 算法处理流
```

11.4.3 标准容器设备

如果仅能使用原始数组，而不能使用标准容器来输入输出那可实在是太不方便了，iostreams 库考虑到了这方面的需求，对标准容器提供了较好的支持。

源设备

iostreams 库不能把一个标准容器直接适配成源设备，但借助 boost.range 库提供的函数 boost::make_iterator_range()（参见 6.5 节），可以把容器直接传递给过滤流从而创建出可用于输入的流。示例代码如下：

```
string str("123");                                     //标准字符串容器
filtering_istream in(make_iterator_range(str));        //创建输入流

vector<char> v(str.begin(), str.end());                //标准向量容器
filtering_istream in2(make_iterator_range(v));         //创建输入流
```

接收设备

iostreams 库为标准容器提供了一个接收设备的适配类 back_insert_device，它位于头文件<boost/iostreams/device/back_inserter.hpp>，类摘要如下：

```
template<typename Container>
```

```
class back_insert_device {
public:
    typedef typename Container::value_type  char_type;        //字符类型
    typedef sink_tag                        category;         //设备的分类
    back_insert_device(Container& cnt);                       //构造函数
protected:
    Container* container;
};
```

back_insert_device 调用了标准容器的 insert()操作，向容器的末尾追加数据，与 std::back_insert_iterator 很像。

为了方便使用，iostreams 库提供工厂函数 back_inserter(Container& cnt)，它可以直接返回一个被适配的接收设备，在使用过滤流或者 io::copy 算法时非常有用：

```
string str("123");                                    //字符串标准容器
vector<char> v;                                       //标准向量容器

io::copy( make_iterator_range(str),                   //调用 io::copy 算法
          io::back_inserter(v));
```

容器的设备适配器

使用上述两种方法可以把标准容器用于流处理，但无论如何，我们还是没有符合设备概念的容器设备，有的时候可能真的非常需要（虽然可能性很小）。

iostreams 库在示例代码<libs/iostreams/example/container_device.hpp>中提供了三个可用的模板类：container_source、container_sink 和 container_device，它们可以把符合随机访问遍历概念的标准容器（包括 vector、string、deque）适配成设备。

如果确有需要，那么读者可以使用这三个适配器类，注意它们位于名字空间 boost::iostreams::example。当然，也希望有那么一天 iostreams 能把它们“扶正”成为库的正式组件。

11.4.4 文件设备

文件设备位于头文件<boost/iostreams/device/file.hpp>，它们基于标准库的文件流提供了对文件的访问，其用法与标准库的 fstream 类似。

iostreams 库提供了三个文件设备：file_source、file_sink 和 file，分别是源设备、接收设备和可定位设备。

类摘要

这三个文件设备实际上是 basic_file_source、basic_file_sink 和 basic_file 的 char 类型特化，核心类是 basic_file。

basic_file 的类摘要如下：

```
template<typename Ch>
class basic_file {
public:
    typedef Ch               char_type;                       //字符类型
    struct                   category                         //设备的分类
        : public seekable_device_tag,
          public closable_tag,
          public localizable_tag
        { };
    basic_file( const std::string& path,  openmode mode);

    std::streamsize          read(char_type* s, std::streamsize n);
    std::streamsize          write(const char_type* s, std::streamsize n);
    bool                     putback(char_type c);
    std::streampos           seek( stream_offset off, seekdir way,
                                   openmode which = in | out);
    void                     open( const std::string& path, openmode mode);
    bool                     is_open() const;
    void                     close();
private:
    struct impl {                                             //内部实现类
        impl(const std::string& path, openmode mode)
            { file_.open(path.c_str(), mode); }
        ~impl() { if (file_.is_open()) file_.close(); }
        std::basic_filebuf file_;
    };
    shared_ptr<impl>         pimpl_;
};
```

解说

basic_file 使用 shared_ptr 和 pimp 惯用法包装了 std::basic_filebuf，因此用户无须关心文件的打开关闭问题，shared_ptr 可以自动管理生命周期。

basic_file 的用法与 std::fstream 非常相似，可以使用文件名构造直接打开文件，使用 read() 和 write() 读写数据，seek() 移动文件指针的位置。

用法

示范文件设备流用法的代码如下：

```
string str("file device");                  //标准字符串
file_sink fsink("test.txt");                //文件接收设备

io::copy(                                   //io::copy 算法把字符串写入文件
  make_iterator_range(str),                 //可以直接使用 make_iterator_range()
  fsink);                                   //文件接收设备接收

file_source fsrc("test.txt");               //文件源设备
io::copy(fsrc, cout);                       //从文件流中读取字符，拷贝到标准输出流
```

11.4.5 空设备

空设备位于头文件<boost/iostreams/device/null.hpp>，它们是空对象模式的具体应用，是不执行任何动作的设备。

iostreams 库提供了两个空设备：null_source 和 null_sink，分别是源设备和接收设备。

类摘要

null_source 和 null_sink 实际上是 basic_null_source 和 basic_null_sink 的 char 类型特化，它们仅是输入输出模式不同，最后实现的都是同一个类 basic_null_device。

basic_null_device 的类摘要如下：

```
template<typename Ch, typename Mode>
class basic_null_device {
public:
    typedef Ch              char_type;              //字符类型
    struct                  category                //设备的分类
```

```
        : public Mode,
          public device_tag,
          public closable_tag
        { };
    std::streamsize      read(Ch*, std::streamsize) { return 0; }
    std::streamsize      write(const Ch*, std::streamsize n) { return n; }
    std::streampos       seek(...)
                         { return -1; }
    void                 close() { }
    void                 close(BOOST_IOS::openmode) { }
};
```

解说

空设备的所有成员函数都不执行任何操作，不处理数据：用户无法从 null_source 中获取任何数据，因为 read()函数总返回 0 字节的数据；用户可以向 null_sink 写入任意数据，但数据会被完全忽略，就像是写进了一个“黑洞”。

空设备可以用在某些特定的场合，比如完全不关心数据的去向或者禁止读取数据处理。

11.5 过滤器

设备的概念很重要，它定义了数据的起点和终点。我们可以从源设备读取数据，向接收设备写入数据，但仅有设备还不够，因为单纯的数据读写是没有意义的，更重要的是数据在流转过程中的处理，这正是过滤器做的工作。

判断一个类是否是过滤器可以使用元函数 is_filter<T>。

11.5.1 概述

过滤器（filter）是一种特殊的设备，它不具有源设备或接收设备的读写功能，只能允许数据流过，但它能够对流过的数据执行任意操作，完成某些特殊的功能。

filter 是过滤器的概念类，它位于头文件<boost/iostreams/concepts.hpp>，类摘要如下：

```
template<typename Mode, typename Ch = char>
struct filter {
```

```
    typedef Ch                          char_type;           //字符类型
    struct                              category             //设备的分类
        : Mode, filter_tag, closable_tag, localizable_tag{ };

    template<typename Device> void   close(Device&);
    template<typename Device> void   close(Device&, openmode);
};
```

同 device 一样，filter 也有模式和字符类型两个模板参数，它们定义了 filter 的特征，也可以使用元函数 char_type_of<T>和 mode_of<T>等获取。

为了方便用户使用，filter 定义了若干特化用于表示子概念，例如：

```
typedef filter<input>        input_filter;                //输入过滤器
typedef filter<output>       output_filter;               //输出过滤器
typedef filter<seekable>     seekable_filter;             //可定位过滤器
typedef filter<dual_use>     dual_use_filter;             //两用过滤器
```

iostreams 库提供了大量的预定义的过滤器和辅助用户定制的过滤器，下面先介绍设备链和管道概念，然后着重介绍几个常用的过滤器。

11.5.2 设备链和管道

iostreams 库借鉴了 UNIX 中 “管道”的概念和形式，数据通过管道可以从一个设备流向另一个设备。因为重载了 operator|，所以我们可以使用“|”把多个过滤器连接成链，最后通常以一个设备结束，这个设备就是数据的起点或终点。

设备链通常的形式如下：

```
filter1 | filter2 | ... | filterN | filter-or-device
```

对于输出链，数据从左到右依次流过每一个过滤器，每一次流过过滤器时都被该过滤器处理，最后到达链的终点时写入接收设备。

对于输入链，数据的流向则刚好相反，数据先从最右端的源设备流入，然后从右到左依次流过每一个过滤器。

如果链的终点不是一个设备，那么链就称为“不完整的链”（incomplete chain），不能用于 IO 操作，反之则称为“完整的链”（complete chain）。

类摘要

chain 位于头文件<boost/iostreams/chain.hpp>，实际上是 chain_base 的子类，因为真实接口较复杂，故简化摘要如下：

```
template< typename Mode, typename Ch = char >
class chain {
public:
    typedef Ch          char_type;              //字符类型
    typedef Mode        mode;                   //设备的模式
    typedef Tr          traits_type;            //其他特征

    chain();
    chain(const chain&);

    std::streamsize    read(char_type* s, std::streamsize n);
    std::streamsize    write(const char_type* s, std::streamsize n);
    stream_offset      seek(stream_offset off, std::ios_base::seekdir way);

    const std::type_info&          component_type(int n) const;
    template<typename T>    T*     component(int n) const;

    void               push( const T& t);       //添加设备
    void               pop();                   //移除设备
    bool               empty() const;
    size_type          size() const;
    void               reset();
    bool               is_complete() const;     //是否完整
};
```

解说

chain 的主要功能就是管理设备链，因此它内部使用 std::list 来保存所有设备的拷贝，成员函数 component_type() 可返回第 n 个设备的类型信息，而 component() 则以指针的形式返回该设备。注意，component() 的模板参数不能自动推导，必须手工指定，我们已经在 11.2.2 节见过这样的用法。

chain 可以一开始构造为一条空链，也可以传入一个设备或者用管道连接的多个设备。如果

链是空的或者不完整的，那么还可以随时使用成员函数 push() 向链的末尾追加设备或流，pop() 函数的功能则刚好相反——它从链的末尾删去一个设备。使用 push() 的时候要注意，chain 存储设备的拷贝，因此有时候可能需要 boost.ref 库的帮助，它可以包装引用。

chain 还提供一些简单的成员函数来查看链的属性，如判断是否为空、是否完整，获取链的长度等类似于标准容器的操作，都比较简单，读者可自行参考文档了解。

管道

iostreams 库的管道功能位于头文件<boost/iostreams/pipeline.hpp>，提供了一个辅助类 pipeline 和重载的“operator|”操作符，让我们可以用“语法糖”简单地把设备 push 到链中。

但一个不幸的消息是，被 boost.ref 库包装后的引用不能够使用管道功能，这是因为经过 ref 库包装后返回的是 reference_wrapper 类型，没有为它定义的“operator|”重载。

11.5.3 计数过滤器

计数过滤器位于头文件<boost/iostreams/filter/counter.hpp>，是一个两用（dual_use）过滤器，也就是说既可以用作输入也可以用作输出。它是一个“透明”的过滤器，不对流经的数据做任何更改，仅仅统计字符数和行数，类似于 UNIX 工具 wc。

类摘要

basic_counter 是计数过滤器的基本类，counter 只是 basic_counter 的 char 特化。

basic_counter 的类摘要如下：

```
template<typename Ch>
class basic_counter {
public:
    typedef Ch          char_type;                              //字符类型
    struct              category                                //设备的分类
      : dual_use, filter_tag,                                   //两用过滤器
        multichar_tag, optimally_buffered_tag
      { };
    explicit            basic_counter(int first_line = 0, int first_char = 0);
    int                 characters() const { return chars_; }
    int                 lines() const { return lines_; }
```

```
private:
    int                  lines_;                              //行数
    int                  chars_;                              //字符数
};
```

用法

counter 是一个很简单的过滤器，它的用法也很简单，只需要把它加入设备链，用 io::copy 算法让数据流经它，就可以使用成员函数 characters() 和 lines() 获取统计数据了。

我们已经在 11.2.2 节中学习了 counter 的使用范例，下面再用代码简单地示范一下：

```
string str("counter\nfilter\n");                         //标准字符串

//过滤流，使用计数过滤器和空接收设备构成链
filtering_ostream out(counter() | null_sink() );

io::copy(                                                //io::copy 算法流处理
  boost::make_iterator_range(str), out );

auto pc = out.component<counter>(0);                     //获得计数过滤器指针
assert(pc->characters() == 15);
assert(pc->lines() == 2);
```

因为只统计字符数，所以这段代码中使用了 null_sink 这个空设备，它只是简单地消耗输入，不做任何输出动作。使用 io::copy 算法后调用函数 component() 获取计数过滤器的指针，然后得到统计数据。

11.5.4 换行过滤器

由于历史原因，各种操作系统的文本文件使用的行结束符各不相同：UNIX 使用的是'\n'，Apple II 和旧型号 Mac（System1 直至 OS9）使用的是'\r'，而 DOS/Windows 使用的则是'\r\n'。iostreams 库提供换行过滤器 newline_filter，它能够在这三种文件格式之间转换行结束符。

类摘要

newline_filter 位于头文件<boost/iostreams/filter/newline.hpp>，是一个两用（dual_use）过滤器，类摘要如下：

```
template<typename Ch>
```

```
class newline_filter {
public:
    typedef Ch          char_type;                      //字符类型
    struct              category                        //设备的分类
        : dual_use, filter_tag, closable_tag            //两用过滤器
        { };
    explicit            newline_filter(int target);     //构造函数，设置目标格式
};
```

解说

newline_filter 类本身很简单，它的构造函数要求指定转换的目标格式常量，在名字空间 boost::iostreams::newline 里定义了如下几个换行符常量：

```
namespace newline {
  const char CR              = 0x0D;
  const char LF              = 0x0A;

  const int posix            = 1;
  const int mac              = 2;
  const int dos              = 4;
  const int mixed            = 8;
}
```

用法

newline_filter 需要在构造函数中设置要转换的目标格式，然后把它加入设备链，再使用 io::copy，在接收设备里就可以获得转换好的文本，示范代码如下：

```
string str("abcdef\n" "12345\n" "aochijk\n");   //待处理的字符串，UNIX 换行格式
string result;                                   //结果字符串

filtering_ostream out(                           //过滤流
    newline_filter(newline::mac) |               //转换到旧式 mac 格式
    io::back_inserter(result));                  //标准字符串作为接收设备

io::copy(boost::make_iterator_range(str), out);  //io::copy 算法流处理
```

换行符检查器

<boost/iostreams/filter/newline.hpp> 里还提供一个检查换行符的过滤器

newline_checker，它不修改文本，在过滤完数据后可以用成员函数获取文本的换行符相关信息，其声明如下：

```
class newline_checker {
public:
    explicit newline_checker();

    bool is_posix() const;
    bool is_dos() const;
    bool is_mac() const;
    bool is_mixed_posix() const;
    bool is_mixed_dos() const;
    bool is_mixed_mac() const;
    bool is_mixed() const;
    bool has_final_newline() const;
};
```

newline_checker 的用法与 newline_filter 的用法基本相同，例如：

```
filtering_ostream out(                              //过滤流，使用空设备结束
     newline_checker() | io::null_sink());

io::copy(boost::make_iterator_range(str), out);
assert(out.component<newline_checker>(0)->is_posix());
```

11.5.5 正则表达式过滤器（I）

iostreams 库提供了两个正则表达式过滤器，第一个过滤器的名字是 regex_filter，它在流里搜索匹配的正则表达式，匹配成功则替换文本，位于头文件<boost/iostreams/filter/regex.hpp>，类似于 UNIX 工具 sed。

使用 regex_filter 必须要链接 boost.regex 库。

类摘要

basic_regex_filter 是正则表达式过滤器的核心类，摘要如下：

```
template< typename Ch>
class basic_regex_filter
    : public aggregate_filter<Ch, ...>
{
public:
```

```
    typedef function1<string_type, const match_type&> formatter;

    basic_regex_filter( const regex_type & pattern, const string_type & fmt);
    basic_regex_filter( const regex_type & pattern, const Ch* fmt);
    basic_regex_filter( const regex_type & pattern, const formatter& replace);
};
```

为简单起见，类摘要中忽略了正则表达式的匹配标志参数。

解说

basic_regex_filter 使用 regex 库执行正则表达式匹配，因此在构造时必须传入一个 regex 对象。构造函数的第二个参数是替换的目标文本，可以直接使用 C 字符串或者标准字符串形式的简单文本。

需要留意 basic_regex_filter 的第三种形式的构造函数，它使用了 boost.function 库，可以接受任意一个用于格式化的单参函数或函数对象。这个函数接受正则表达式匹配结果的 match_results，然后返回处理后的字符串。因为有了这个格式化函数的间接层，所以就可以在里面利用正则表达式的全部功能做任何事情。

简单用法

regex_filter 的用法也很简单，与 regex 库的 regex_replace() 的功能基本相同，但手法却是流处理的形式。

第一个例子使用了简单的文本替换，把表达式“a.c”替换为“test”：

```
regex reg("a.c");                              //构造一个正则表达式
string str("abcdef aochijk");                  //要被处理的字符串
string result;                                 //结果字符串

filtering_ostream out(                         //过滤流
  regex_filter(reg, "test") |                  //正则表达式过滤器
  io::back_inserter(result));                  //标准容器接收设备

io::copy(                                      //io::copy 算法流处理
  boost::make_iterator_range(str),             //适配字符串为输入设备
  out);                                        //输出到过滤流
```

程序的运行结果是：

```
testdef testhijk                                //“a.c”已替换为“test”
```

格式化函数的用法

为了使用格式化函数，正则表达式必须做出一点改动，增加子表达式的定义，并编写一个参数为 regex_filter::match_type 的函数，来处理匹配结果，代码如下：

```
regex reg("a(.)c");                             //使用圆括号定义一个子表达式
string str("abcdef aochijk");
string result;

//lambda 表达式定义格式化函数，使用第 1 个子表达式
auto formatter = [](const regex_filter::match_type& match)
            {
                return string("test-") + match[1] + "-";
            };

filtering_ostream out(                          //过滤流
    regex_filter(reg, formatter) |              //使用格式化函数
    io::back_inserter(result));                 //标准容器接收设备

io::copy(                                       //io::copy 算法流处理
  boost::make_iterator_range(str), out);        //适配字符串为输入设备
```

程序的运行结果是：

```
test-b-def test-o-hijk
```

11.5.6 正则表达式过滤器（II）

grep_filter 是 iostreams 库里的另一个正则表达式过滤器，提供类似于 UNIX 工具 grep 的功能，可以抓取流中含有匹配的行，它位于头文件<boost/iostreams/filter/grep.hpp>。

类摘要

basic_grep_filter 是 grep_filter 的核心类，摘要如下：

```
namespace grep {                                        //grep 选项定义
      const int invert       = 1;                       //保留不匹配的行
      const int whole_line   = invert << 1;             //保留匹配的行
}
```

```
template< typename Ch>
class basic_grep_filter : public basic_line_filter<Ch> {
public:
   basic_grep_filter( const regex_type& re,                //正则表达式
                      match_flag_type match_flags ,        //正则表达式匹配标志
                      int options = 0 );                   //grep 选项
   int    count() const;
};
```

用法

grep_filter 的主要功能集中在它的构造函数，它使用正则表达式匹配流中的字符串，并根据 options 来决定匹配的方式。options 的取值定义在子名字空间 boost::iostreams::grep 里。如果是 whole_line，那么保留匹配的行；如果是 invert，那么保留不匹配的行。最后提取的行数可以使用成员函数 count() 获取。

下面的代码与 regex_filter 很接近，可用来提取含有正则表达式的行：

```
regex reg("a.c");                                    //构造一个正则表达式
string str("abcdef\n" "12345\n" "aochijk\n");        //要被处理的多行字符串
string result;                                       //结果字符串

filtering_ostream out(                               //在算法中直接创建过滤流
     grep_filter(reg) |                              //正则表达式过滤器，缺省选项
     io::back_inserter(result));                     //标准容器接收设备

io::copy(                                            //io::copy 算法流处理
  boost::make_iterator_range(str), out);             //适配字符串为输入设备
```

程序的运行结果是：

```
abcdef
aochijk
```

如果使用 invert 方式来提取行，那么代码就是：

```
filtering_ostream out(
       grep_filter(reg, regex_constants::match_default, grep::invert) |
       io::back_inserter(result));
```

运行后将输出第二行“12345”。

11.5.7 压缩过滤器

iostreams 库支持三种压缩算法，分别是 zlib、gzip 和 bzip2，必须配合相应的 zlib、bzip2 库才能使用。

因为这些压缩过滤器用法类似，所以下面我们主要介绍 zlib 压缩算法，它位于头文件 <boost/iostreams/filter/zlib.hpp>。

类摘要

zlib_compressor/zlib_decompressor 实现了 zlib 压缩算法，实际上是 basic_zlib_compressor/ basic_zlib_decompressor 的特化形式，前者用于压缩，而后者用于解压缩，这两个过滤器都是两用（dual_use）过滤器，因此既可以用于输入也可以用于输出。

basic_zlib_compressor 和 basic_zlib_decompressor 的类摘要如下：

```
template<typename Alloc = std::allocator<char> >
struct basic_zlib_compressor: symmetric_filter<>
{
public:
   basic_zlib_compressor( const zlib_params&, int buffer_size );
   zlib::ulong    crc();
   int            total_in();
};

template<typename Alloc = std::allocator<char> >
struct basic_zlib_decompressor: symmetric_filter<>
{
public:
   basic_zlib_decompressor( int window_bits, int buffer_size);
   basic_zlib_decompressor( const zlib_params& p, int buffer_size);
   zlib::ulong    crc();
   int            total_out();
};
```

basic_zlib_compressor 和 basic_zlib_decompressor 调用 zlib 库实现压缩解压缩功能，构造函数都有缺省值，不需要特别的配置就可以工作得很好。

子名字空间 boost::iostreams::zlib 里定义了 zlib 算法所需要的全部参数信息，可以设置压缩级别、压缩方法、压缩策略等参数，可以使用 zlib_params 结构传递给压缩过滤器。它的参数较多，本书不做过多的介绍，有需要的读者请参考 Boost 文档或者源代码。

用法

压缩过滤器很容易使用（只要不去定制复杂的压缩策略），下面直接给出示范代码：

```
string str("12345678 12345678");                //待压缩的数据
string zip_str, unzip_str;                      //保存压缩和解压缩的数据

filtering_ostream zout(                         //使用压缩过滤器输出
  zlib_compressor() | io::back_inserter(zip_str) );

io::copy(                                       //io::copy 算法流处理
  boost::make_iterator_range(str),              //适配字符串为输入设备
  zout);

filtering_ostream uzout(                        //使用解压缩过滤器输出
  zlib_decompressor() | io::back_inserter(unzip_str) );

io::copy(                                       //io::copy 算法流处理
  boost::make_iterator_range(zip_str),          //适配字符串为输入设备
  uzout);

assert(unzip_str == str);                       //解压缩后数据还原
```

如果在压缩处理过程中发生错误，那么压缩过滤器会抛出异常 zlib_error，它是 std::exception 的子类，可以用成员函数 error()获取与 zlib 兼容的错误代码。

11.6 流

设备和过滤器仅是一些孤立的对象，而流则是连接它们的纽带，只有通过流数据才能从一个设备向下一个设备转移，在流转的过程中实现数据的处理。

在前几节中，我们已经多次使用了基本流 stream 和过滤流 filtering_ostream，本节将对它们做详细的讨论。

11.6.1　基本流

在 iostreams 库中基本流有 stream 和 stream_buffer 两个模板类，两者的功能和行为很相似，为简单起见，我们只讨论 stream，它位于头文件<boost/iostreams/stream.hpp>。

类摘要

stream 使用了模板元编程技术，可以根据模板参数的特征信息推导出它的父类，可能是 basic_istream、basic_ostream 或者 basic_iostream，简化的摘要如下：

```
template< typename T >
class stream {
public:
    stream();
    stream( const T& t);
    stream(const U1& u1, const U2& u2, ..., const UN& uN);

    void        open( const T& t);
    void        open(const U1& u1, const U2& u2, ..., const UN& uN);

    bool        is_open() const;
    void        close();

    T&          operator*();
    T*          operator->();
};
```

解说

stream 是标准库流的子类，因此具有标准 IO 流的所有功能，例如 get()、read()等操作。它实际上有四个模板参数，但只有第一个流要连接的设备类型 T 是必须的，其他的都可以使用缺省值。

stream 并不像标准库的流那样用名字来区分输入流和输出流，它的方向完全取决于它关联的设备 T——使用源设备就是输入流，使用接收设备就是输出流。

stream 可以在构造的时候直接打开设备，也可以稍后调用 open()函数打开设备。它们的函数参数很有特点：单参形式接受设备的实例，多参版本则接受构造设备所需的 N 个参数，由

stream 在内部创建设备的实例。正因为如此，我们在之前的代码中才可以通过直接传递数组首末地址来直接创建流。

如果流已经关联到实际的设备（构造时打开或者调用 open()），那么成员函数 is_open() 会返回 true。这时可以使用重载的 operator*和 operator->来获取流内部的设备引用，就像是一个智能指针。

基本流的用法之前已经演示了很多，故这里不再举例。

11.6.2 过滤流

过滤流可以说是基本流的强化版，它不仅能够连接设备，更可以连接过滤器和链，因此它的用途更加广泛，功能更加强大。

在 iostreams 库中过滤流有 filtering_stream 和 filtering_streambuf 两个模板类，两者的功能和行为很相似，同样我们只讨论 filtering_stream，它位于头文件<boost/iostreams/filtering_stream.hpp>。

类摘要

与 stream 一样，filtering_stream 使用了模板元编程技术，根据模板参数信息推导出它的父类，之前使用的 filtering_ostream 和 filtering_istream 实际上是它的模式特化类，模式分别为 output 和 input。

filtering_stream 简化的摘要如下：

```
template< typename Mode>
class filtering_stream {
public:
   typedef Ch                    char_type;                  //字符类型
   typedef Mode                  mode;                       //过滤器的模式

   filtering_stream();
   filtering_stream( const T& t);
   filtering_stream( StreamOrStreambuf& t);

   const std::type_info&     component_type(int n) const;

   template<typename T> T*   component(int n) const;
```

```
    void                        push( const T& t);
    void                        pop();
    bool                        empty() const;
    size_type                   size() const;
    void                        reset();
    bool                        is_complete() const;
private:
    Chain                       chain_;              //设备链
};
```

解说

虽然过滤流是基本流的强化版，但比较代码可以发现两者之间的差异还是较大的，filtering_stream 的接口与 chain 更接近一些。

filtering_stream 的主要模板参数是 Mode，指定了过滤流的模式。构造函数与 stream 差别很大，它不仅能够接受设备，还能够接受设备链和流，这一点务必请注意。

filtering_stream 内含了 chain 的实现，因此它具有与 chain 相似的接口，可以向链中添加、删除或者访问设备，以及检查链是否完整。

用法

与 stream 不同，filtering_stream 不能直接通过连接设备来决定它的模式，而是需要使用模板参数指明，但我们通常会直接使用定义好的 filtering_istream 和 filtering_ostream。

过滤流的构造函数可以传入多个设备对象，中间用管道符（“|”）连接，形成一个设备链。如果终点是一个接收设备那么它就相当于一个增强了的输出流，可以被写入，写入数据时执行过滤操作，反之则是一个增强的输入流。但需要注意，管道操作只能针对设备，它不能连接流，如果要把流对象加入过滤流则必须使用成员函数 push()。

示范过滤流用法的代码如下：

```
char ar[10] = "abcd";
stream<array_source> sa(ar);            //定义一个字符数组输入流

counter ct;                             //一个计数过滤器
filtering_istream in;                   //定义一个空的输入过滤流
assert(in.empty());
```

```
in.push(ref(ct));                      //加入计数过滤器，使用 ref 包装，链不完整
assert(!in.is_complete());
in.push(sa);                           //加入输入流，链完整
assert(in.is_complete());

filtering_ostream out(cout)            //输出过滤流直接连接标准输出，没有添加过滤器
io::copy(in, out);                     //io::copy 算法从输入过滤流拷贝

assert(ct.characters() == 10);
```

这段代码示范了 filtering_stream 的多个成员函数的使用，需要注意的是输入输出流的末端都是流而不是设备。

11.7 流处理函数

设备、过滤器和流是 iostreams 库的三大要件，但仅有这三个概念还是不够的，流虽然把设备和过滤器连接在一起，但它们仍然是静止的，要想让数据真正地流动起来还需要一个外部的驱动——流处理函数。

iostreams 库提供了数个流处理函数，其中最重要的是 **io::copy()**，它驱动了整个数据流——从源设备或流中读出字符，再写入到接收设备或流中，最后关闭两个设备并返回处理的字符数。

io::copy()位于头文件<boost/iostreams/copy.hpp>，声明形式如下：

```
template<typename Source, typename Sink>
std::streamsize copy( Source& src, Sink& sink);
```

io::copy()有四种重载形式，用来应对 src 和 sink 是设备或流的情形。我们之前已经多次使用了 io::copy()，但都是操作流对象，实际上它也可以直接操作设备，有的时候会使代码更加简单，例如：

```
char ar[] = "abcd123";                 //字符数组
stream<array_source> in(ar);           //源设备
io::copy(in, cout);                    //从源设备到输出流

string result;                         //标准字符串
in.open(ar);                           //再次打开源设备
```

```
io::copy(in, io::back_inserter(result));          //从源设备到接收设备
```

除了 io::copy()之外，iostreams 库中还有其他流处理函数，它们大都位于头文件<boost/iostreams/operations.hpp>中，常用函数包括以下几个。

- get(d) ：从源设备或流中获取一个字符。
- read(d,s,n) ：从源设备或流中获取多个字符。
- put(d,c) ：向接收设备或流中写入一个字符。
- write(d,s,n) ：向接收设备或流中写入多个字符。
- seek(d,off,way)：随机访问设备或流。
- flush(d) ：刷新设备或整个流，清空缓冲区。
- close(d) ：关闭设备或流。

应用流处理时我们通常较少使用这些函数，它们主要在编写自定义设备或过滤器时发挥作用，使得我们能够以一致的方式方便地编写泛型代码来操作流。

11.8 定制设备

通过前几节的学习，我们已经基本熟悉了 iostreams 库的各个组件和用法，能够操作预定义的设备、过滤器和流执行流处理，但 iostreams 库不仅仅是一个工具库，它更是一个框架，允许——更进一步地说是鼓励我们在这个框架之内编写自己的代码，去扩充、增强流处理的功能。

我们必须对 iostreams 库的设备、过滤器、流等概念有一定程度的了解，才能编写可用于流处理框架的类。好在我们不必白手起家，iostreams 库预定义的若干设备和过滤器都是良好的范例，在熟悉它们功能的基础上阅读其实现源代码是一个很好的提高自身水平的方法。

对 iostreams 的扩展可从编写设备和编写过滤器这两个方面入手，本节将研究编写设备，下一节将研究编写过滤器。

11.8.1 定制源设备

编写设备首先要满足设备的概念（参见 11.4.1 节），具有 char_type 和 category 内部类型定义，对于源设备来说其模式必须为 input。

实现原理

头文件<boost/iostreams/concepts.hpp>中定义了若干特化来表示一些设备的子概念，其中就包括为了方便用户扩展而定义的 source，这里再声明如下：

```
typedef device<input>    source;                    //char 源设备
```

只要我们自定义的设备 public 继承 source，那么设备就会自动满足源设备的概念。当然，不一定非要使用 source 作为基类（例如 iostreams 库自己的数组设备和文件设备），但对于用户来说通常这样会更加方便。

源设备被用于输入，因此它需要实现一个如下形式的 read()成员函数：

```
std::streamsize read(char_type* s, std::streamsize n);
```

read()函数最多可读取 n 个字符到缓冲区 s 中，然后返回读取的字符数表示读取成功，返回 EOF（-1）则表示已经读取完毕。

示例 1：随机数源设备

这里我们使用 boost.random 库的 rand48 随机数发生器编写一个随机数源设备 rand_source，它可以产生从“0”到“z”的随机字符，其实现代码如下：

```
class rand_source: public io::source             //定义随机数源设备
{
private:
  std::streamsize  count;                         //需要产生的随机数数量
public:
  rand_source(int c):                             //构造传入数量
      count(c){   }

  //核心函数，实现源设备的流读取功能
  std::streamsize read(char_type* s, std::streamsize n)
  {
      // 最多读取 n 个字符到缓冲区 s 中，返回读取的字符数
      auto read_count = (min)(n, count);

      if (read_count)                             //应读取的数量
      {   return EOF; }                           //返回 EOF 表示已经读取完毕
```

```
    for (streamsize i = 0;i < read_count; ++i)
    {    *s++ = rand();}                      //产生随机数，拷贝到缓冲区

    count -= read_count;                      //当前剩余的读取数量
    return read_count;                        //返回读取的字符数
  }                                           //read()成员函数结束
private:
  //使用变量发生器组合rand48和uniform_smallint
  typedef variate_generator<rand48, uniform_smallint<>> rand_t;

  static rand_t& rand()                       //静态成员函数，返回随机数对象
  {
     static rand_t r(rand48(time(0)), uniform_smallint<>('0', 'z'));
     return r;
  }
};
```

有了这个源设备，我们就可以把它作为 stream 的模板参数，创建一个流用于流处理：

```
string out;                                   //输出用标准字符串
io::copy(rand_source(20),                     //直接使用源设备
  io::back_inserter(out));                    //输出到字符串
```

示例 2：改进的随机数源设备

在第二个例子中，我们对 rand_source 做出了一点小小的改动，把流处理的字符类型由 char 变为 unsigned char，因为 source 是 char 设备，因此我们不能从它继承，需要从 device 直接特化，其他的逻辑不变：

```
class rand_source: public io::device<input, unsigned char>
{...};                                        //实现代码同前
```

因为流处理设备链上的设备必须使用一致的字符类型，所以我们要再定义一个配套的标准容器 block，它是 basic_string 的特化：

```
typedef std::basic_string<unsigned char> block;
```

新的 rand_source 的用法与原来的相同，但它处理的是 unsigned char 字符类型：

```
block out;                          //注意，不能使用 std::string，因为字符类型不同
io::copy(rand_source(10),           //直接从源设备拷贝
    io::back_inserter(out));
```

11.8.2 定制接收设备

如果理解了源设备的编写原理，那么我们也可以很快地学会如何编写接收设备。自定义接收设备同样要满足 device 的定义，具有 char_type 和 category 内部类型定义，只不过对于接收设备来说其模式必须为 output，它的便捷特化形式是 sink：

```
typedef device<output>  sink;              //char 接收设备
```

对于接收设备，我们需要编写如下形式的 write() 成员函数：

```
std::streamsize write(const char_type* s, std::streamsize n)
```

write() 从缓冲区 s 中最多获取 n 个字符，然后写入某个地方（文件、内存等），最后返回写入的字符数，处理逻辑比源设备的 read() 要简单一些。

下面的代码实现了一个非常简单的接收设备 cout_sink，它仅仅把字符输出到标准流上：

```
class cout_sink: public io::sink
{
public:
  streamsize write(const char_type* s, streamsize n)
  {
      cout << string(s,n) << endl;
      return n;
  }
};
```

11.9 定制过滤器

过滤器是 iostreams 真正展现威力的地方，流处理的大多数有用的功能都是利用过滤器实现的，因此，编写自己的过滤器是使用 iostreams 库的必备技能。

iostreams 库中预定义的过滤器都是很好的范例，读者可以参照研究。

11.9.1 过滤器的实现原理

编写过滤器必须符合 filter 的概念，它位于头文件<boost/iostreams/concepts.hpp>。

根据模式的不同，过滤器可以分为输入过滤器、输出过滤器、两用过滤器、可定位过滤器等，最常用的是前两者，分别用于输入流和输出流。根据访问字符的方式过滤器又可分为普通（单字符）过滤器和多字符过滤器，普通过滤器一次只处理一个字符，虽然效率低但较容易处理，而多字符过滤器则正好相反，接下来我们将主要编写多字符过滤器。

过滤器辅助类

iostreams 库提供了数个编写过滤器的辅助类，让用户可以更容易地编写自己的过滤器，这些过滤器包括以下几种。

- aggregate_filter ：两用过滤器，一次性接收所有的数据。
- basic_line_filter ：两用过滤器，每次提取一行数据。
- basic_stdio_filter ：两用过滤器，用于适配标准输入输出流。
- symmetric_filter ：两用过滤器，带有内部缓冲区。

以上四个过滤器中的前三个用起来比较简单，我们只需要遵循模板方法模式实现纯虚函数 do_filter()，在里面编写适当的处理逻辑就可以了。而 symmetric_filter 的应用较复杂，本书暂不做介绍。

管道符

让自定义过滤器支持 iostreams 的管道符操作通常是个好主意，它可以让过滤器更好地搭配其他设备和流工作。iostreams 库特别提供了一个宏 BOOST_IOSTREAMS_PIPABLE，它可以只用一行代码就给过滤器增加管道功能：

```
#define BOOST_IOSTREAMS_PIPABLE(filter, arity)
```

BOOST_IOSTREAMS_PIPABLE 通常用在过滤器定义之后，它的第一个参数 filter 是过滤器的名字，第二个参数 arity 是过滤器模板参数的数量，如果过滤器不是模板类，那么值应该是 0。

BOOST_IOSTREAMS_PIPABLE 使用了预处理元编程技术，宏展开后会形成一个针对 filter 重载的 operator| 函数，因此宏后面无须添加分号（当然，加上分号也不算错误）。

11.9.2 aggregate_filter

aggregate_filter 可以一次性读取全部字符序列，它位于头文件<boost/iostreams/filter/aggregate.hpp>，类摘要如下：

```
template< typename Ch>
class aggregate_filter {
public:
    typedef std::vector<Ch> vector_type;
private:
    virtual void do_filter(const vector_type& src, vector_type& dest) = 0;
    virtual void do_close() { }
};
```

aggregate_filter 是一个两用过滤器，这意味着使用它既可以实现输入过滤器也可以实现输出过滤器。

aggregate_filter 使用模板方法模式完成了过滤器所需的大部分读写工作，只需要实现过滤的核心功能 do_filter()纯虚函数。它有两个参数，src 是过滤器接收到的所有字符，我们可以对这些字符进行任意的处理，然后把处理结果添加到 dest，这两个参数都存储在 std::vector<Ch>中。

另一个虚函数 do_close()在过滤器被关闭时调用，通常可以用缺省的空实现。但有的时候也可以重载它做一些特别的关闭操作。

11.5.5 节介绍的 regex_filter 就是利用 aggregate_filter 实现的。

示例：SHA1 过滤器

作为示范，我们使用 aggregate_filter 来编写一个十六进制编码的输出过滤器 hex_filter，它利用了 boost.algorithm 库的 hex 算法，代码如下：

```
class hex_filter final:
      public io::aggregate_filter<char>        //使用 aggregate_filter，窄字符
{
private:
  typedef io::aggregate_filter<char>  super_type;     //简化类型定义
  typedef super_type::vector_type     vector_type;
```

```
  virtual void do_filter(                              //实现虚函数，过滤器的核心功能
          const vector_type& src, vector_type& dest)
  {
      boost::algorithm::hex(src, std::back_inserter(dest));
  }
};

BOOST_IOSTREAMS_PIPABLE(hex_filter, 0)                         //增加管道功能实现
```

下面的代码示范了这个新过滤器的使用：

```
char str[] = "313";                                            //字符数组
string result;                                                 //标准字符串

filtering_ostream out(                                         //输出流
  hex_filter () |                                              //hex 过滤器
  counter() |                                                  //统计流过的字符数
  io::back_inserter(result));                                  //用 string 接收摘要

io::copy(array_source(str, str + 3), out);                     //流处理，输出“333133”

assert(result.size() == 6);                                    //hex 编码后长度变为两倍
assert(out.component<counter>(1)->characters() == 6);
```

11.9.3 basic_line_filter

basic_line_filter 是一个类似于 aggregate_filter 的过滤器辅助类，它位于头文件<boost/iostreams/filter/line.hpp>，类摘要如下：

```
template<typename Ch>
class basic_line_filter {
private:
   virtual string_type do_filter(const string_type& line) = 0;
};
```

basic_line_filter 是一个两用过滤器，用法与 aggregate_filter 很像，但它一次过滤一行字符，由纯虚函数 do_filter()处理这行字符，再返回处理后的新行。

为了方便使用，basic_line_filter 提供了 line_filter 和 wline_filter 两个特化：

```
typedef basic_line_filter<char>        line_filter;
```

```
typedef basic_line_filter<wchar_t>    wline_filter;
```

11.5.6 节的 grep_filter 就是利用 basic_line_filter 实现的。

示例：简单的行过滤器

下面的代码实现了一个示范性质的行过滤器 bracket_line_filter，它的功能非常简单，即为每一行首尾两端增加一对尖括号：

```
class bracket_line_filter final:
  public io::basic_line_filter<char>              //使用 basic_line_filter
{
private:
  typedef io::basic_line_filter<char>         super_type;
  typedef typename super_type::string_type    string_type;

  virtual string_type do_filter(const string_type& line)
  {
      return "<" + line + ">";                      //简单处理一下行然后返回
  }
};

BOOST_IOSTREAMS_PIPABLE(bracket_line_filter, 0); //增加管道功能实现
```

bracket_line_filter 可以这样使用：

```
string str("The\n" "Phantom\n" "Pain");

filtering_istream in(                               //使用输入过滤流
      bracket_line_filter() |                       //行过滤器
      boost::make_iterator_range(str));             //从标准字符串输入
io::copy(in, cout);                                 //输出到标准输出流
```

在这段代码中，需要注意的是我们把 bracket_line_filter 加入到了输入流中，因为 bracket_line_filter 是一个两用流，所以它允许这样使用。

11.9.4 手工打造过滤器

以上我们都是使用 iostreams 库提供的过滤器辅助类来编写过滤器，但有的时候这些辅助类并不能满足我们的要求，我们还需要完全手工编写过滤器类。

手工编写过滤器其实并不复杂，只是把原来辅助类替我们实现的 char_type、category、read()、write()、close()等过滤器的必备要素自己编写出来就可以了。因为这些都由我们

自己控制，所以可以获得更多的灵活性。

对于多字符过滤器来说，我们需要实现如下形式的读写函数：

```
template<typename Source>
std::streamsize read(Source& src, char_type* s, std::streamsize n)
{
    //从 src 读取字符处理，处理后最多写入 n 个字符到缓冲区 s，返回读取的字符数或者 EOF
}

template<typename Sink>
std::streamsize write(Sink& dest, const char_type* s, std::streamsize n)
{
    //从缓冲区 s 最多读取 n 个字符处理，将处理后的字符写到 des，返回已处理的字符数
}
```

示例：SHA1 过滤器

我们使用 boost.uuid 库的 SHA1 工具类来实现一个计算 SHA1 摘要的输出过滤器：

```
class sha1_filter final                     //不是模板类，也不使用任何的概念类
{
public:
  typedef char char_type;                   //字符类型定义
  struct category:                          //过滤器分类定义
      output,                               //输出模式
      filter_tag,                           //过滤器设备
      multichar_tag,                        //处理多字符
      closable_tag                          //可关闭
      { };
private:
  boost::uuids::detail::sha1 sha;           //SHA1 对象
public:
  template<typename Sink>                   //输出模式需要实现 write()函数
  std::streamsize write(Sink&, const char_type* s, std::streamsize n)
  {
      sha.process_bytes(s, n);              //处理所有输入字符序列
      return n;                             //返回处理的字符数，但暂不输出摘要结果
  }

  template<typename Sink>
```

```
    void close(Sink& snk)                          //过滤器关闭时给出摘要结果
    {
        unsigned int digest[ 5];                   //获得摘要值
        sha.get_digest(digest);

        char_type* p = reinterpret_cast<char_type*>(digest);
        io::write(snk, p, sizeof(digest)/sizeof(char_type));   //写入接收设备
    }
};

BOOST_IOSTREAMS_PIPABLE(sha1_filter, 0);                  //支持管道符操作
```

请读者注意 sha1_filter 与之前两个自定义 filter 的区别,它没有继承任何过滤器概念,因此需要手工定义它的 char_type 和 category。为了方便我们不使用模板参数,而是直接把 char 定义为使用的字符类型,过滤器的分类则是可处理多字符的、可关闭的输出过滤器。

sha1_filter 的模式决定了我们要实现的成员函数。write() 中执行摘要操作,但它过滤后并不向接收设备输出,而是在 close() 关闭过滤器时才输出,因此我们可以向 sha1_filter 中多次传入待摘要的数据,最后关闭过滤器获取摘要结果。

示范 sha1_filter 用法的代码如下:

```
char str[] = "The Evil Within";                //字符数组
string result;                                 //标准字符串

filtering_ostream out(                         //用于输出流
  sha1_filter() |                              //SHA1 过滤器
  io::back_inserter(result));                  //用 string 接收摘要

io::write(out, str, 3);                        //使用 write() 函数向流写入数据
assert(result.empty());                        //此时没有输出结果

io::write(out, str + 3, 3);                    //继续使用 write() 函数向流写入数据
assert(result.empty());                        //此时仍然没有输出结果

io::close(out);                                //关闭流
assert(result.size() == 20);                   //得到摘要结果
```

为了示范 sha1_filter 多段摘要的功能,在代码中我们没有使用 io::copy() 函数,这是因为 io::copy() 不仅拷贝了流数据还会自动关闭流,故 copy 后就无法再使用流了。而 write()

函数不会关闭流，所以我们可以向流多次写入数据，最后手动关闭流来获取结果。

sha1_filter 也可以使用另一个概念类 multichar_output_filter 来简化代码，它仅定义了 char_type 和 category，代码如下所示：

```
class sha1_filter_ex: public io::multichar_output_filter
{                                         //无须再定义 char_type 和 category
  ...                                     //其他同前
};
```

实现不关闭设备的 copy 算法

对于 sha1_filter 这样的可多段输入的过滤器来说不能使用 io::copy()算法，而 write()用起来显然没有 io::copy()方便，可惜 iostreams 库并没有提供不自动关闭设备的 copy 版本。下面我们就仿照 io::copy()编写一个 copy_no_close()函数，限于篇幅仅实现了一个对流操作的版本，其他操作设备的版本读者可自行实现。

io::copy()算法的核心是函数对象 copy_operation，并使用模板元编程技术来处理 source 和 sink 的各种可能情况，我们的版本则简化了很多：

```
//使用辅助函数来自动推导模板参数
template<typename Source, typename Sink>
std::streamsize
copy_no_close_impl(Source src, Sink snk,
                   std::streamsize buffer_size )
{
  return io::detail::copy_operation<Source, Sink>(
      src, snk, buffer_size )
      ();                                 //调用 copy 操作，不关闭设备
}

//调用辅助函数完成拷贝操作
template<typename Source, typename Sink>
std::streamsize
copy_no_close( const Source& src, Sink& snk,
               std::streamsize buffer_size =
                  io::default_device_buffer_size
               BOOST_IOSTREAMS_ENABLE_IF_STREAM(Source)
               BOOST_IOSTREAMS_ENABLE_IF_STREAM(Sink) )
{
  return copy_no_close_impl(
```

```
        io::detail::wrap(src),
        io::detail::wrap(snk),
        buffer_size );
}
```

copy_no_close 的用法与 io::copy()基本相同，记得最后要使用 io::close():

```
copy_no_close(in, out);                          //拷贝数据,不关闭流
assert(in.is_open());                            //输入流未关闭
io::close(in);                                   //关闭输入流
io::close(out);                                  //关闭输出流
```

11.10 组合设备

现在我们已经初步具备了编写自定义设备和过滤器的能力，但是在很多情况下复杂的设备和过滤器的实现难度很大，因此 iostreams 库提供了一些模板类和函数，可以把简单易实现的设备或过滤器组合起来，形成可用的新设备。

11.10.1 combine

模板类 combination 可以“并联”组合一对源/接收设备或者输入/输出过滤器，形成一个新的设备或者过滤器。新的设备是一个可在两个独立字符序列上工作的双向设备，使用前者输入的同时使用后者输出。iostreams 库同时提供工厂函数 combine()来简化类的构造，它们位于头文件<boost/iostreams/combine.hpp>。

模板类 combination 和函数 combine()的声明如下:

```
template<typename In, typename Out>
class combination;

template<typename In, typename Out>
combination<In, Out> combine(const In& in, const Out& out);
```

使用 combine 工具我们可以很容易地创建双向设备，只需要分别实现两个单向的输入输出设备，再使用 combine()把它们组合起来就可以了。

下面的代码使用 combine()组合了数组设备和标准容器设备:

```
char src[] = "12345678";                         //输入字符序列
string result;                                   //输出字符序列
```

```
typedef io::combination<array_source,           //组合设备类型定义
      back_insert_device<string> > bi_dev;

assert(is_device<bi_dev>::value);               //元函数检查是否是设备
assert((is_same<                                //元函数检查是否是双向模式
      mode_of<bi_dev>::type, bidirectional>::value));

bi_dev dev = io::combine(                       //使用函数生成组合设备
  array_source(src),
  io::back_inserter(result));

io::write(dev, src, 2);                         //向输出序列写入数据
assert(result.size() == 2);
```

11.10.2 compose

compose 是另一种组合设备或过滤器的工具，它把两个设备“串联”起来，数据顺序（输出模式）或逆序（输入模式）流过新的设备，有些类似于管道符的作用，它位于头文件<boost/iostreams/compose.hpp>。

compose 提供一个模板类 composite，它的第一个模板参数是过滤器，第二个模板参数是过滤器、设备或者流，类似于 std::pair 可以用 first()和 second()获取它组合的对象，工厂函数 compose()用来自动推导参数生成新设备对象，它们的声明如下：

```
template<typename Filter, typename FilterOrDevice>
class composite {
public:
   typedef typename char_type_of<Filter>::type  char_type;
   composite(const Filter& first, const FilterOrDevice& second);

   Filter& first();
   Device& second();
};

composite<Filter, FilterOrDevice>
compose(const Filter& first, const FilterOrDevice& second);
```

下面的代码简单示范了 compose 的用法，它把正则表达式过滤器和标准输出流组合在了一

起，效果与 11.5.4 节完全相同：

```
string str("abcdef aochijk");                    //输入字符串

io::copy(                                        //io::copy 算法流处理
  boost::make_iterator_range(str),
  compose(regex_filter(regex("a.c"), "test"),    //正则表达式过滤器
          cout)                                  //标准输出流
  );
```

11.10.3 invert

有的时候我们在实现过滤器时编写 read()很容易，但编写 write()却比较麻烦，或者情况刚好相反，这时我们可以尝试使用 iostreams 库的 invert 工具。invert 利用 compose 的功能，适配一个过滤器并反转它的模式——也就是说把输入过滤器变成输出过滤器或者相反，它位于头文件<boost/iostreams/invert.hpp>。

invert 使用模板类 inverse 来实现模式的反转，元计算被适配过滤器的模式，然后转变为相反的模式，内部定义一个 source 或 sink 设备，用 compose 把过滤器和设备组合起来，再完成读写操作。工厂函数 invert()用来自动推导参数生成反转后的新设备对象，声明如下：

```
template<typename Filter>
class inverse {
   typedef typename category_of<Filter>::type   base_category;
public:
   typedef typename char_type_of<Filter>::type  char_type;
   typedef typename
         mpl::if_<
            is_convertible< base_category, input>,      //元计算过滤器的模式
            output, input                               //反转模式
         >::type                                 mode;

   inverse(const Filter& filter);
};

template<typename Filter>
inverse<Filter> invert(const Filter& filter);
```

虽然 invert 只需要一个模板参数，看起来用法比较简单，但它不是万能的，实际使用时还

是会受到一些限制。它不能用于两用（dual-use）过滤器，因为两用过滤器本身就支持输入输出，强行使用invert会导致编译错误(无法反转模式)。另外，如11.9.4节实现的sha1_filter那样在关闭时有读写操作的过滤器也不能使用 invert，这是因为模板类 inverse 内部compose 的设备只是一个临时对象，仅在读写操作时有效，而 sha1_filter 需要在关闭时写入，此时没有可操作的设备，会导致抛出异常。

在此我们编写一个简单的输出过滤器 null_filter：

```
class null_filter final                        //一个简单的空过滤器，类似于 counter
{
public:
    typedef char char_type;                    //字符类型定义
    struct category:                           //过滤器分类定义
        output,                                //输出模式
        filter_tag,                            //过滤器设备
        multichar_tag,                         //处理多字符
        closable_tag                           //可关闭
    { };
public:
    template<typename Sink>                    //输出模式需要实现 write()函数
    std::streamsize write(Sink& snk, const char_type* s, std::streamsize n)
    {
        io::write(snk, s, n);                  //原样写入
        return n;
    }

    template<typename Sink>
    void close(Sink& snk)                      //过滤器关闭空操作
    {}
};
BOOST_IOSTREAMS_PIPABLE(null_filter, 0);       //支持管道符操作
BOOST_IOSTREAMS_PIPABLE(inverse, 1);           //为 inverse 添加管道符操作
```

invert 的用法如下：

```
char str[] = "Ground Zero";                    //字符数组
string result;                                 //标准字符串

filtering_istream in(                          //输入流
```

```
    invert(null_filter()) |                    //反转过滤器为输入模式，可使用管道符
    array_source(str));

io::copy(in, io::back_inserter(result));       //流处理
```

11.10.4 restrict

restrict 是另外一种适配器，为被适配的设备或过滤器添加一个约束，限定只能访问一个子区间，它位于头文件<boost/iostreams/restrict.hpp>。

因为 restrict 在 C99 标准中是一个关键字（但在 C++中不是），为了保持兼容它提供了另外一个名字 slice，位于头文件<boost/iostreams/slice.hpp>。

模板类 restriction 实现访问限制，关键在于它的构造函数，用参数 off 指定偏移位置，len 指定偏移长度，限定只能访问[start + off, start + off + len)的区间，有点类似于 C 的文件操作函数 fseek。工厂函数 restrict()或 slice()用来自动推导参数生成约束后的新设备对象，它们的声明如下：

```
template<typename Component>
class restriction {
public:
    typedef typename char_type_of<Component>::type   char_type;
    typedef some-defined                             mode;

    restriction( Component& component,
                 stream_offset off, stream_offset len = -1 );
};

restriction<Component>
restrict( Component& component, stream_offset off, stream_offset len = -1 );
```

下面的代码示范了 restrict 的用法，它限制 counter 过滤器只访问源设备的[1,3)区间里的两个字符：

```
BOOST_IOSTREAMS_PIPABLE(restriction, 1);              //添加管道符支持

char str[] = "1234";                                  //字符数组
string result;                                        //标准字符串
```

```
filtering_istream in(                          //输入流
    restrict(counter(),1,2) |                  //添加约束
    array_source(str));

io::copy(in, io::back_inserter(result));       //流处理
assert(result == "23");
```

restrict 最常见的应用场景是文件流的访问，指定读写文件的某个特定位置，比调用标准库的 seekg()/seekp()要方便得多。

11.10.5 tee

tee 工具提供了分离流输出的功能，它可以建立流的分支，把上游的数据同时发给另外一个接收设备，配合 compose 就可以实现对一个序列同时执行两种或两种以上不同的流处理，它位于头文件<boost/iostreams/tee.hpp>。

tee 可以把两个接收设备组合成一个接收设备，加到设备链的末尾，其声明如下：

```
template<typename Sink1, typename Sink2>
class tee_device {
public:
    typedef typename char_type_of<Sink1>::type  char_type;
    tee_device(const] const& sink1, const Sink2& sink2);
};

tee_device<Sink1, Sink2> tee(const Sink1& sink1, const Sink2& sink2);
```

下面的代码把 11.10.2 节的例子做了一点变动，使用 compose 创建了两个接收设备，并用 tee 分流，这样数据将同时输出到标准流和标准容器：

```
string str("abcdef aochijk");                  //输入字符串
string result;                                 //标准容器接收设备

filtering_ostream out(                         //输出流
      tee(                                     //使用 tee
          compose(                             //compose 到标准流
              regex_filter(regex("a.c"), "test"),cout),
```

```
        compose(                                    //compose 到标准容器接收设备
            hex_filter(), io::back_inserter(result))
    ));

io::copy(                                           //io::copy 算法流处理
  boost::make_iterator_range(str), out);            //源设备
```

11.11 其他议题

本小节将简要讨论关于 iostreams 库的一些其他议题。

11.11.1 对象的生存周期

如果读者曾经使用过 Crypto++或者 Botan 这两个密码学库，那么可能会知道它们也使用了流处理的思想，也具有 iostreams 库中类似的 source、sink、filter、pipe、chain 等概念，不过它们与 iostreams 库对对象的生命周期的管理方式存在着很大的区别。在 Crypto++或者 Botan 的设备链中可以直接传递 new 生成的设备对象指针，由链来管理设备对象的生命周期，用起来很方便。而 iostreams 库则要求设备必须是可拷贝构造的，否则就要使用 boost.ref 来包装引用，用起来有那么一点不便。

实际上，iostreams 库曾经考虑过动态创建对象并传递所有权的方式，并且有过基于这一方式的实现，但最后考虑到异常安全的问题而最终废弃了这种方式，现在的拷贝构造方式对设备虽然有更多的要求但也更加安全。

11.11.2 与迭代器的比较

把 iostreams 库与第 5 章和第 6 章的迭代器和区间放在一起比较会很有意思，下面就对它们做个简单的对比，可以让我们更好地理解两者。

首先看相似之处，流处理与迭代器对数据序列的处理方式非常相似。

- 都使用 copy 算法，遍历一个源序列，然后把处理过的数据写入另一个序列。
- 流的模式与迭代器的分类很接近，也可以分为可读的输入模式和可写的输出模式。
- 迭代器可以嵌套组合提供复杂的功能，而流则可以用设备链的形式过滤处理数据。
- 迭代器适配为区间的概念后，也可以像流一样使用过滤器来处理数据。

流处理与迭代器之间的差异也是很明显的，简单列举如下。

- 迭代器基于迭代器设计模式，不使用虚函数，通过改写迭代器的核心操作来完成对数据的处理；iostreams 则基于已确定的标准流框架，使用流处理思想，并使用设备、过滤器等概念来处理数据。
- 迭代器是泛型的，可以处理任何类型的数据；iostreams 虽然也是泛型的，但它更侧重于处理字符类型。
- 迭代器通常用于处理内存中的数据，而 iostreams 则还可以处理文件、socket 等更广范围的数据。
- 迭代器虽然也可以组合使用，但它只能在编译期确定，显然没有流的设备链可以任意添加拆卸的方式灵活。

11.12 总结

在本章中，我们研究了 boost.iostreams 库，它扩展了标准库的流处理功能，提供了一个更加强大易用的流处理框架。

iostreams 库基于标准库的流实现，而标准流使用了多重继承和虚函数，因此它存在少量的虚函数调用开销，这是不可避免的，但 iostreams 库通过大量的泛型设计和模板元编程技术降低了虚函数的代价，在一般的应用中是可以接受的。

iostreams 库已经被广大的 Boost 用户证明结构清晰并且很容易学习，但因为包含的内容太过庞杂丰富，一些组件的用法作者也没有完全掌握，故书中展现给读者的也只能是其中的一小部分，它的更多的功能还需要读者在实践开发中进一步探索。

本章大致可分为以下三部分。

- 11.1 节至 11.2 节 ：阐述流处理的工作原理和 iostreams 库的组织结构。
- 11.3 节至 11.7 节 ：详细介绍 iostreams 库中的基本概念和组件的使用方法。
- 11.8 节至 11.10 节：介绍基于 iostreams 库提供的设备和过滤器来扩展流处理功能。

在第一部分中，我们讨论了标准库的流处理框架和 iostreams 库对它的扩展，随后用两个

示例程序初步示范了 iostreams 库中的设备、过滤器、流等的用法。

在第二部分中，我们讨论了流处理的核心内容，介绍了设备、过滤器、流等重要概念，并研究了若干 iostreams 库预定义的设备和过滤器。使用 iostreams 库提供的这些设施我们可以很容易地编写流处理代码，以流的方式执行很多有用的功能。

在第三部分中，我们讨论了定制流处理，可以基于 iostreams 库的框架编写自己的设备和过滤器，能够随心所欲地操作各种数据流，几乎一切数据都能以流的方式处理。iostreams 库不仅提供了实现设备和过滤器的基本类，还提供了一些把简单的设备组合成复杂设备的辅助工具，利用好它们会使代码更加简洁优雅。

第12章

泛型编程

泛型编程是进入新世纪以来 C++的主流编程范式，它带来了更好更快的代码，但同时离早期的编程概念也越来越远，我们更多地是和类型打交道，代码编写工作更像是数学中的“代数”——真实的类型用占位符 T、U 等来代替，然后让编译器实例化模板去求解这些难题——这最终导致了模板元编程的诞生。

本章讨论 Boost 库中的三个泛型编程用的工具。

- enable_if　　　：在编译期启用或禁用特定的泛型代码。
- call_traits　 ：非标准元函数，计算类型 T 可能的多种类型，经常被用于函数的入口参数或者返回值类型的计算。
- concept_check：以库的方式实现了泛型编程中急需的概念检查功能，在标准提供语言级别的概念检查支持之前是我们唯一可用的工具。

12.1 enable_if

enable_if 主要用来解决模板函数或模板类的重载解析问题，允许模板函数或模板类仅针对某些特定类型有效，即依据条件启用或禁用某些特化形式，其中的一部分已经被收入 C++11（头文件<type_traits>，C++12.20.9.7.6）。

enable_if 库位于名字空间 boost，需要包含头文件<boost/core/enable_if. hpp>，即：

```
#include <boost/core/enable_if.hpp>
```

```
using namespace boost;
```

12.1.1 类摘要

enable_if 库提供了两类元函数，分别是 enable_if 的“启用”系列和 disable_if 的“禁用”系列。

enable_if 的类摘要如下：

```
template <bool B, class T = void>              //T 缺省值是 void 类型
struct enable_if_c {
  typedef T type;                              //默认返回类型 T
};
template <class T>
struct enable_if_c<false, T> {};               //对 false 特化，无::type 返回

template <class Cond, class T = void>
struct enable_if :
     public enable_if_c<Cond::value, T> {};    //计算元函数 Cond
```

enable_if 使用元函数转发技术，计算条件元函数 Cond 的值，再交给 enable_if_c。如果条件为 true，那么 enable_if/enable_if_c 将返回类型 T，否则 enable_if/enable_if_c 将是一个无返回的元函数。

disable_if 与 enable_if 相似，但在语义上是反义词，即条件 Cond 成立时无返回：

```
template <bool B, class T = void>              //T 缺省值是 void 类型
struct disable_if_c {
  typedef T type;                              //默认返回类型 T
};
template <class T>
struct disable_if_c<true, T> {};               //对 true 特化，无::type 返回

template <class Cond, class T = void>
struct disable_if :
     public disable_if_c<Cond::value, T> {};   //计算元函数 Cond
```

enable_if/disable_if 的工作原理涉及 C++中模板实例化的重载解析：

处理重载函数时编译器要构造所有同名函数的集合，再从中选择一个最恰当的函数。当存在模板函数时，如果模板函数可以被模板实参推演实例化，那么它就是一个候选函数；反之，如果

某个参数或返回值类型无效而导致推演失败无法实例化，那么这个模板函数就不是候选函数，编译器也不会认为这是一个编译错误。这就是著名的 SFINAE 原则，即“替代失败不是错误”（substitution failure is not an error）。

12.1.2 应用于模板函数

enable_if 通常需要配合 type_traits 或者 mpl 使用，作为模板推演时的控制条件，检查类型 T 是否满足某些条件。

enable_if 可放在函数参数列表的最末尾用作缺省参数，或者是用作返回值，两种形式的效果是相同的，但有的时候只能使用一种形式，比如用于构造函数和析构函数时没有返回值，用于操作符重载时不能变动参数的数量。

下面的代码示范了 enable_if 的用法，这个 print() 函数仅在类型是整数时才生效：

```
template<typename T>
T print(T x ,
        typename enable_if<is_integral<T> >::type* =0)    //整数时启用
{
  cout << "int:" << x << endl;
  return x;
}
```

代码中 enable_if 作为函数 print() 的缺省参数出现，声明了一个无名指针参数，默认值是空指针。这样，当编译器进行模板实例化时，如果 T 不是整数，那么 enable_if 将不会返回任何类型，导致实例化失败，从而使这个 print() 被禁用。

enable_if 的返回值用法如下，效果与缺省参数的形式相同：

```
template<typename T>
typename enable_if<is_integral<T>, T >::type                    //整数时启用
print(T x )
{ ...}
```

注意，在这里我们向 enable_if 传递了第二个元参数 T，因为如果不这么做，enable_if 的返回值将是 void，不符合函数的签名，这与 enable_if 的缺省参数用法略微有些不同[①]。

① 当然 enable_if 的缺省参数用法也可以使用 enable_if<is_integral<T>, T >的形式，但因为指针参数并不被实际使用，因此默认的 void 类型就可以正常工作了。

第二个元参数也不一定必须是 T，我们也可以在这里再进行元计算，比如使用 promote<T>提升 T 的范围。

使用 disable_if 可以禁止函数的实例化，例如不允许 print()操作类类型：

```
template<typename T>
typename disable_if<is_class<T>, T >::type          //T 是 class 时禁用
print(T x )
{ ...}
```

下面的这个例子摘自 ptr_container 库的 ptr_sequence_adapter 类（8.3 节），它使用了 disable_if，当类型是指针或者整数时禁用该函数：

```
template< class Range >
disable_if< is_pointer_or_integral<Range> >::type
insert( iterator before, const Range& r )
{
    insert( before, boost::begin(r), boost::end(r) );//使用 range 操作
}
```

多索引容器的键提取器（10.4 节）也使用了 disable_if，它被用来递归地生成解引用指针类型的成员函数。

12.1.3 应用于模板类

enable_if 的启用或禁用模板类偏特化的用法与模板函数用法类似，它需要为类的模板参数列表增加一个额外缺省参数，缺省值是 void，然后再使用 enable_if 来偏特化。

示范 enable_if 的模板类用法的代码如下：

```
template<typename T, typename Enable = void>      //增加一个模板参数
class demo_class
{ ...};
template<typename T>                              //然后使用 enable_if 偏特化
class demo_class<T, typename enable_if<is_arithmetic<T> >::type>
{ ...};
```

在这里 demo_class 使用 enable_if 对 int、double 等算术类型进行了偏特化。很显然，enable_if 使得模板偏特化的应用范围更大了，可以针对某一些而不是某一个特定的类型偏特化，简单的偏特化相当于使用 is_same：

```
//对 string 类型特化
template<typename T>
class demo_class<T, typename enable_if<is_same<T, string> >::type>
{ ...};
```

12.1.4 对比 C++11 标准

enable_if 很有用，所以被收入了 C++11/14 标准，但 C++11/14 标准里的 enable_if 只是 boost.enable_if 的一个很小的子集（并且没有 disable_if），其声明如下：

```
template <bool, class T = void> struct enable_if;
```

std::enable_if 的功能很有限，不能像 boost::enable_if 那样计算元函数，因为第一个模板参数是一个 bool 类型，它实际上相当于 boost::enable_if_c。

12.2 call_traits

call_traits 是一个很小的泛型工具，它封装了 C++中编写函数时可能是“最好的”传递参数给函数的方式，会自动推导出最高效的传递参数的类型，某种程度上可以说是一个“智能参数类型”。

call_traits 位于名字空间 boost，需要包含头文件<boost/call_traits.hpp>，即：

```
#include <boost/call_traits.hpp>
using namespace boost;
```

12.2.1 类摘要

call_traits 是一个返回多个值的非标准元函数，其类摘要如下：

```
template <typename T>
struct call_traits
{
public:
   typedef T             value_type;             //T 的值类型
   typedef T&            reference;              //T 的引用类型
   typedef const T&      const_reference;        //T 的 const 引用类型
   typedef some_define   param_type;             //T 的被调用参数类型
};
```

这段代码只是 call_traits 的基本形式，它还同时提供针对 T&、T*、T[N] 等的其他特化形式，代码与之类似。

12.2.2 用法

在实际的编程工作中经常需要编写函数相关参数，有很多规则告诉我们如何书写会使代码更加高效，比如 POD 类型通常传值比传引用或者指针更高效，而类类型通常应该传引用，为了更安全，有时候还需要加上 const 修饰，函数返回时有值和引用的区别等。虽然规则并不多也不难理解，但实际使用的时候总是难免有所偏差，毕竟每个人对规则的理解程度都不一样，而且还存在偶尔笔误的可能。

使用 call_traits 我们就可以不必过多地去考虑类型的各种形式，直接交给它来处理。call_traits 对类型 T 执行元计算，依据 C++社区已经达成的共识返回以下四个最高效的类型（元数据）任我们选用：

- value_type ：T 的“值类型”，通常是 T，但对于数组 T[N] 则退化为 const T*，可用于保存值或者以值返回。
- reference ：T 的“引用类型”，通常是 T&，可用于返回值引用。
- const_reference：T 的“常引用类型”，通常是 const T&，可用于返回值引用。
- param_type ：T 的“参数类型”，通常是 const T&，但对于 POD 类型或者指针类型则是 const T，可用作“最好的”函数参数传递类型。

这四个类型中最方便也是最常使用的是 param_type，它可以用作函数的参数类型。

假设我们现在要编写一个连接两个字符串的函数 scat()：

```
string scat(string& s1, string& s2)
{ return s1 + s2;}
```

初看上去函数并无多大缺陷，参数的传递使用了引用，避免了大对象的拷贝成本，返回也使用了值返回，不会出现悬空引用。但这个函数不是最佳的，接口存在一点小缺陷，下面的简单代码无法通过编译：

```
cout << scat("1", "2");                              //编译错误
```

这是因为编译器无法把一个字符串类型转换为 string&类型，我们必须使用临时变量的形式来写，显得麻烦许多：

```
cout << scat(string("1"), string("2"));
```

使用 call_traits 我们无须费力就可以避免这样的“低级错误”，使我们的函数接口更加合理且高效：

```
typedef call_traits<string> str_traits;             //简化类型定义

str_traits::value_type                              //返回值类型
scat(str_traits::param_type s1,                     //参数类型
     str_traits::param_type s2)                     //参数类型
{
  //断言参数类型是引用，并且是const string&
  assert(is_reference<str_traits::param_type>::value);
  assert((is_same<str_traits::param_type, const string&>::value));

  return s1 + s2;
}
```

对于模板类来说 call_traits 也非常有用，它可以很好地推断最合适的模板参数类型，节约我们定义类型的时间和精力。例如，下面的代码定义了一个模板类 demo_class，得益于 call_traits 的使用，它可以容纳任意的类型，包括数组和引用：

```
template<typename T>
class demo_class
{
public:
  typedef typename call_traits<T>::value_type v_type;
  typedef typename call_traits<T>::param_type p_type;
  typedef typename call_traits<T>::reference  r_type;
  typedef typename call_traits<T>::const_reference cr_type;
private:
  v_type v;
public:
  demo_class(p_type p): v(p){}

  v_type value()
  {   return v;   }

  r_type get()
  {   return v;   }
```

```
};
```

示范 demo_class 的用法的代码如下：

```
int a[3] = {1,2,3};
demo_class<int[3]> di(a);                   //容纳数组类型
assert(di.value()[0] == 1);

char c = 'A';
demo_class<char&> dc(c);                    //容纳引用类型
assert(dc.get() == c);
```

读者可以自行尝试一下，如果不使用 call_traits，那么编写这样一个可以容纳任意类型的模板类将是一项多么复杂的工作。

12.2.3 实现原理

call_traits 的实现原理并不复杂，它使用模板特化技术，针对 T&、T*、T[N] 等类型做了特殊处理，解决了“引用的引用”和数组的类型问题。例如，对于 T&，call_traits 的特化代码如下：

```
template <typename T>
struct call_traits<T&>                      //针对 T&特化
{
    typedef T& value_type;
    typedef T& reference;                   //注意这里
    typedef const T& const_reference;
    typedef T& param_type;
};
```

通过元函数 call_traits 这个间接层将 T&的引用类型仍然定义为 T&，因而成功地回避了旧标准里“引用的引用”错误。

对于一般的情况，call_traits 使用位于名字空间 boost::detail 里的两个元函数 ct_imp 和 ct_imp2 来实现对 param_type 的类型计算。ct_imp 的定义如下：

```
template <typename T, bool isp, bool b1, bool b2>
struct ct_imp
{
    typedef const T& param_type;
};
```

ct_imp 有三个元参数，T 是要计算的类型，isp 和 b1、b2 是 bool 值，分别表示 T 是否为指针、是否为算术类型、是否是枚举类型，对应 type_traits 库的元函数则是 is_pointer、is_arithmetic 和 is_enum。如果 T 既不是指针也不是算术、枚举类型（元参数均为 false），那么 param_type 就被定义为常引用类型 const T&，否则 ct_imp 将进行模板特化：

```
template <typename T, bool b1>
struct ct_imp<T, true, b1 , b2>                        //指针类型
{
  typedef const T param_type;
};
template <typename T, bool isp>
struct ct_imp<T, isp, true , b2>                       //算术类型
{
  typedef typename ct_imp2<T,                          //调用元函数 ct_imp2
      sizeof(T) <= sizeof(void*)>::param_type param_type;
};
```

元函数 ct_imp2 再根据类型 T 是否是一个“小”类型（小于一个指针的大小）来决定 param_type 是 const T 或是 const T&：

```
template <typename T, bool small_>
struct ct_imp2
{
  typedef const T& param_type;
};

template <typename T>
struct ct_imp2<T, true>
{
  typedef const T param_type;
};
```

12.3 concept_check

泛型编程中使用的是“静态多态”，它在语义上经常要求类型具有某种“特征”或者满足某种“条件”，例如有内嵌的类型定义或者固定名字的成员函数、支持迭代操作，这些要求通常被称为“概念”(concept)。

但 C++最初并不是为泛型编程所设计的，它对运行期检查做得很好，但对泛型编程缺乏足够的支持，没有有效的对模板类型参数的验证机制和手段，这给编写泛型程序带来了很大的不方便。C++11 曾经有提案要求在语言级别增加对“概念检查”的支持，但最后被否决了，不过 Boost 的 concept_check 以库的方式达到了同样的效果，并且能够在编译出错时给出更可读的信息。

concept_check 位于名字空间 boost，需要包含头文件<boost/concept_check.hpp>，即：

```
#include <boost/concept_check.hpp>
using namespace boost;
```

12.3.1 概述

概念检查的基本工具是宏 BOOST_CONCEPT_ASSERT，它的用法很像静态断言 BOOST_STATIC_ASSERT，可以用在任何域（scope）：函数域、类域、名字空间域，如果概念检查不通过则导致编译错误。但 BOOST_CONCEPT_ASSERT 与静态断言也是有区别的，不能在宏中使用逻辑运算符（!、&&等）。还需要注意的是宏的参数必须要用括号括起来，也就是说使用双重括号，像这样：

```
BOOST_CONCEPT_ASSERT((some_check_class));              //双重括号
```

concept_check 库提供了大量的概念检查类来检查类型是否符合某个概念，这与 type_traits 库（第 3 章）有些类似，它们同样是检测类型的属性，只是 type_traits 的检查更偏重于 C++的类型系统，而 concept_check 库更偏重于类型的功能属性。另外，type_traits 库提供的是标准的元函数，而 concept_check 库的概念检查类虽然也可以算作是元函数，但它们通常不用于元计算，在多数情况下用来配合检查宏工作。

concept_check 库里用于概念检查元函数有很多，可分为如下六个类别。

- 基本的概念检查 ：检查整数类型、拷贝构造、缺省构造、赋值函数等的基本概念。
- 函数对象概念检查 ：检查函数对象的相关概念。
- 标准迭代器概念检查：检查标准库的五种迭代器的分类概念。
- 新式迭代器概念检查：检查 Boost 定义的九种迭代器的分类概念（参见 5.1.3 节）。
- 容器概念检查 ：检查容器的相关概念。
- 区间概念检查 ：检查区间的相关概念（参见第 6 章）。

12.3.2 基本概念检查

基本概念检查的功能与 type_traits 提供的功能很相似，包括如下的检查类。

- Integer<T> ：检查 T 是否是内建的整数类型。
- SignedInteger<T> ：检查 T 是否是内建的有符号整数类型。
- UnsignedInteger<T> ：检查 T 是否是内建的无符号整数类型。
- Convertible<X, Y> ：检查 X 是否可转换为 Y。
- Assignable<T> ：检查 T 是否是可赋值的。
- DefaultConstructible<T> ：检查 T 是否有缺省构造函数。
- CopyConstructible<T> ：检查 T 是否有拷贝构造函数。
- EqualityComparable<T> ：检查 T 是否可以进行相等比较。
- LessThanComparable<T> ：检查 T 是否可以进行小于比较。
- Comparable<T> ：检查 T 是否是可以进行所有关系运算。

我们可以使用这些概念检查类来实现标准库的 min() 函数：

```
template<typename T>
T my_min(const T& l,const T& r)
{
  BOOST_CONCEPT_ASSERT((LessThanComparable<T>));      //要求可小于比较

  return (l < r) ? l : r;
}
```

这样，当一个没有定义 operator<的类被传入 my_min() 时会产生编译错误：

```
complex<double> cp1, cp2;
my_min(cp1, cp2);                                         //编译错误
```

12.3.3 函数对象概念检查

函数对象概念检查主要基于标准库的函数对象定义，它们的模板参数较复杂，除了输入要检查的函数对象类型外，还依情况需要输入返回值类型和参数类型。下面列出了几个较常用的检查

类，其中 F 是函数对象类型，R 是返回值类型，A 和 B 分别是两个参数类型。

- Generator<F，R> ：检查 F 是否是无参函数对象。
- UnaryFunction<F,R,A> ：检查 F 是否是单参函数对象。
- BinaryFunction<F,R,A,B> ：检查 F 是否是双参函数对象。
- UnaryPredicate<F,A> ：检查 F 是否是单参谓词。
- BinaryPredicate<F,A,B> ：检查 F 是否是双参谓词。
- Const_BinaryPredicate<F,A,B> ：检查 F 是否是 const 双参谓词。

示范这些概念检查类的代码如下：

```
BOOST_CONCEPT_ASSERT((UnaryFunction< negate<int>, int, int>));
BOOST_CONCEPT_ASSERT((BinaryFunction< plus<int>,int, int, int>));
```

12.3.4 标准迭代器概念检查

concept_check 的迭代器概念检查完全依据 C++标准的定义（参见 5.1.2 节），概念检查类还定义了 value_type、reference、pointer 等内部类型，等价于 std::iterator_traits。

迭代器概念检查类列表如下。

- InputIterator<I> ：检查 I 是否是输入迭代器。
- OutputIterator<I,T> ：检查 I 是否是输出类型 T 的输出迭代器。
- ForwardIterator<I> ：检查 I 是否是前向迭代器。
- Mutable_ForwardIterator<I> ：检查 I 是否是可变前向迭代器（即可修改，支持*i++ = *i 操作）。
- BidirectionalIterator<I> ：检查 I 是否是双向迭代器。
- Mutable_BidirectionalIterator<I> ：检查 I 是否是可变双向迭代器。
- RandomAccessIterator<I> ：检查 I 是否是随机访问迭代器。
- Mutable_RandomAccessIterator<I> ：检查 I 是否是可变随机访问迭代器。

示范这些概念检查类用法的代码如下：

```
//原生指针满足迭代器的概念
BOOST_CONCEPT_ASSERT((InputIterator<int*>));
BOOST_CONCEPT_ASSERT((OutputIterator<int*, int>));
BOOST_CONCEPT_ASSERT((RandomAccessIterator<int*>));

//可以从概念检查类获取迭代器的类型定义
assert((is_same<InputIterator<int*>::pointer,
  iterator_traits<int*>::pointer>::value));

//forward_list 是 C++11 提供的单向链表容器，支持前向迭代
BOOST_CONCEPT_ASSERT((ForwardIterator<forward_list<int>::iterator>));
BOOST_CONCEPT_ASSERT((Mutable_ForwardIterator<
                                    forward_list<int>::iterator>));

//vector 支持双向迭代器和随机访问
typedef vector<int>::iterator I;
BOOST_CONCEPT_ASSERT((BidirectionalIterator<I>));
BOOST_CONCEPT_ASSERT((RandomAccessIterator<I>));
```

同样地，我们可以把它应用于自己的代码，例如下面的 _sort() 函数包装了标准库的 stable_sort()，增加了概念检查，要求必须是可随机访问的迭代器：

```
template <typename I>
void _sort(I first, I last)
{
  BOOST_CONCEPT_ASSERT((RandomAccessIterator<I>));
  std::stable_sort(first, last);
}
```

12.3.5 新式迭代器概念检查

iterators 库定义了一组新式的迭代器概念（参见 5.1.3 节），同时也提供了相应的概念检查类，同样可以用作 traits 类来使用。

这些概念检查类位于名字空间 boost_concepts（注意，不是 boost），需要包含头文件 <boost/iterator/iterator_concepts.hpp>，即：

```
#include <boost/iterator/iterator_concepts.hpp>
```

新式迭代器概念检查类如下。

- ReadableIteratorConcept<I> ：检查 I 是否是可读迭代器。
- WritableIteratorConcept<I,T> ：检查 I 是否是可写迭代器。
- SwappableIteratorConcept<I> ：检查 I 是否是可交换迭代器。
- LvalueIteratorConcept<I> ：检查 I 是否是左值迭代器。
- IncrementableIteratorConcept<I> ：检查 I 是否是可递增迭代器。
- SinglePassIteratorConcept<I> ：检查 I 是否是单遍迭代器。
- ForwardTraversalConcept<I> ：检查 I 是否是前向迭代器。
- BidirectionalTraversalConcept<I> ：检查 I 是否是双向迭代器。
- RandomAccessTraversalConcept<I> ：检查 I 是否是随机访问遍历迭代器。

下面的代码使用这些新式迭代器概念检查类检查 vector<bool>::iterator 和 5.4.3 节实现的 vs_iterator：

```
using namespace boost_concepts;                          //打开名字空间
typedef vector<bool>::iterator I;

// vector<bool>::iterator 是可交换的随机访问迭代器
BOOST_CONCEPT_ASSERT((ReadableIterator<I>));
BOOST_CONCEPT_ASSERT((WritableIterator<I>));
BOOST_CONCEPT_ASSERT((SwappableIteratorConcept<I>));
BOOST_CONCEPT_ASSERT((RandomAccessTraversalConcept<I>));

//检查 vs_iterator，可读可写可交换的单遍迭代器
BOOST_CONCEPT_ASSERT((ReadableIterator<vs_iterator<int> >));
BOOST_CONCEPT_ASSERT((WritableIterator<vs_iterator<int> >));
BOOST_CONCEPT_ASSERT((SwappableIterator<vs_iterator<int>>));
BOOST_CONCEPT_ASSERT((SinglePassIterator<vs_iterator<int> >));

//不是左值迭代器，编译错误
BOOST_CONCEPT_ASSERT((LvalueIteratorConcept<I>));
```

因为新式迭代器概念兼容标准库迭代器概念，因此标准迭代器概念检查类相当于是新式迭代

器概念检查类的组合，读者可参考 12.3.9 节。

12.3.6 容器概念检查

容器概念检查类检查是否符合标准库的容器定义，也就是说是否具有 begin()、end()、empty()、size()等成员函数，是否有若干容器必备的内嵌类型定义。

这些容器概念检查类都有内部的 value_type、reference 等概念满足的类型定义，因此也可以把它们当作容器的 traits 元函数。

基本的容器概念检查类如下。

- Container<C>　　：检查 C 是否满足标准容器定义。
- Mutable_Container<C>　　：检查 C 是否满足可变容器定义（即可以修改元素的值）。
- ForwardContainer<C>　　：检查 C 是否可以前向迭代。
- Mutable_ForwardContainer<C>　　：检查 C 是否满足可变前向迭代容器定义。
- ReversibleContainer<C>　　：检查 C 是否可以逆向迭代。
- Mutable_ReversibleContainer<C>　　：检查 C 是否满足可变逆向迭代容器定义。
- RandomAccessContainer<C>　　：检查 C 是否满足随机访问容器定义。
- Mutable_RandomAccessContainer<C>：检查 C 是否满足可变随机访问容器定义。

下面的代码检查了一些标准容器和 Boost 容器：

```
BOOST_CONCEPT_ASSERT((Container<vector<int>>));
BOOST_CONCEPT_ASSERT((RandomAccessContainer<vector<int>>));

//容器检查类有相应的类型定义
assert((is_same<vector<int>::value_type,
               Container<vector<int>>::value_type>::value));

//forward_list 没有 size()成员函数，编译错误
BOOST_CONCEPT_ASSERT((Container<forward_list<int>>));
BOOST_CONCEPT_ASSERT((ForwardContainer<forward_list<int>>));

// forward_list 是单向链表，不能逆向迭代，编译错误
```

```
BOOST_CONCEPT_ASSERT((ReversibleContainer<forward_list<int>>));

//循环缓冲容器circular_buffer符合标准容器定义
BOOST_CONCEPT_ASSERT((Container<boost::circular_buffer<int> >));

//array虽然很像容器，但它不符合标准容器定义，编译错误
BOOST_CONCEPT_ASSERT((Container<boost::array<int> >));
```

接下来的概念检查类检查容器的序列类型。

- Sequence<C>：检查C是否是线性序列容器，如vector。
- FrontInsertionSequence<C>：检查C是否支持序列头插入操作。
- BackInsertionSequence<C>：检查C是否支持序列尾插入操作。
- AssociativeContainer<C>：检查C是否是关联容器。
- UniqueAssociativeContainer<C>：检查C是否不允许重复键。
- MultipleAssociativeContainer<C>：检查C是否允许重复键。
- SimpleAssociativeContainer<C>：检查C是否键即值，即集合类型。
- PairAssociativeContainer<C>：检查C是否是键—值关联类型，即映射。
- SortedAssociativeContainer<C>：检查C是否是有序的。

这些概念检查类的使用方法如下：

```
BOOST_CONCEPT_ASSERT((Sequence<vector<int>>));
BOOST_CONCEPT_ASSERT((Sequence<deque<int>>));
BOOST_CONCEPT_ASSERT((Sequence<list<int>>));

BOOST_CONCEPT_ASSERT((FrontInsertionSequence<deque<int>>));
BOOST_CONCEPT_ASSERT((BackInsertionSequence<list<int>>));

BOOST_CONCEPT_ASSERT((AssociativeContainer<set<int>>));
BOOST_CONCEPT_ASSERT((AssociativeContainer<map<int,int>>));
BOOST_CONCEPT_ASSERT((MultipleAssociativeContainer<multimap<int,int>>));

BOOST_CONCEPT_ASSERT((SimpleAssociativeContainer<set<int>>));
BOOST_CONCEPT_ASSERT((SortedAssociativeContainer<set<int>>));
```

12.3.7 区间概念检查

区间概念检查类检查是否符合区间（第 6 章）的概念——类似容器，但要求比容器要低一些。

区间概念检查类位于头文件<boost/range/concepts.hpp>，包括以下几种。

- SinglePassRangeConcept<R> ：检查 R 是否是单遍区间。
- ForwardRangeConcept<R> ：检查 R 是否是前向区间。
- WriteableForwardRangeConcept<R> ：检查 R 是否是可写前向区间。
- BidirectionalRangeConcept<R> ：检查 R 是否是双向区间。
- WriteableBidirectionalRangeConcept<R>：检查 R 是否是可写双向区间。
- RandomAccessRangeConcept<R> ：检查 R 是否是随机访问区间。
- WriteableRandomAccessRangeConcept<R>：检查 R 是否是可写随机访问区间。

这些概念检查类的使用方法如下：

```
//检查 std::forward_list 符合的区间概念
BOOST_CONCEPT_ASSERT((SinglePassRangeConcept<std::forward_list<int>>));
BOOST_CONCEPT_ASSERT((ForwardRangeConcept<std::forward_list<int>>));

//检查 list 和 vector 符合的区间概念
BOOST_CONCEPT_ASSERT((BidirectionalRangeConcept<std::list<int>>));
BOOST_CONCEPT_ASSERT((RandomAccessRangeConcept<std::vector<int>>));

//检查内建数组符合的区间概念
char a[] = "range";
BOOST_CONCEPT_ASSERT((RandomAccessRangeConcept<decltype(a)>));
```

12.3.8 在函数声明中的概念检查

BOOST_CONCEPT_ASSERT 在基本的概念检查中工作得足够好，但仍然有不足，有时候我们希望能够在函数的声明中“显式”给出概念检查，让用户在使用接口时明确函数所需要的概念，这时 BOOST_CONCEPT_ASSERT 就无能为力了。

因此，concept_check 库另外提供一个宏 BOOST_CONCEPT_REQUIRES，它可以在模板

函数的声明里做概念检查，把检查的时机更向前提一步。

要使用 BOOST_CONCEPT_REQUIRES 必须另外包含如下的头文件：

```
#include <boost/concept/requires.hpp>
```

宏 BOOST_CONCEPT_REQUIRES 的基本用法如下：

```
template <...>
BOOST_CONCEPT_REQUIRES(
  ((some_check_class 1))                   //概念检查类列表开始
  ((some_check_class 2))
  ...
  ((some_check_class N)),                  //概念检查类列表结束
  (return type)      )                     //返回值类型
function_name(...)                         //函数名
{ ...}
```

BOOST_CONCEPT_REQUIRES 只能用在函数声明里，它有两个参数，第一个是概念检查类列表，是多个双重括号的序列，第二个是函数的返回类型，也必须用括号括起来。

使用 BOOST_CONCEPT_REQUIRES 可以把 12.3.2 节的 my_min()改写为如下形式：

```
template<typename T>
BOOST_CONCEPT_REQUIRES(
((LessThanComparable<T>)),                    //检查列表没有逗号分隔，结束使用逗号
 (T)                   )                      //返回类型是第二个宏参数
my_min(const T& l,const T& r)
{     return (l < r) ? l : r;}
```

12.3.4 节对 stable_sort() 的包装类_sort() 也可以改用 BOOST_CONCEPT_REQUIRES，代码如下：

```
template <typename I>
BOOST_CONCEPT_REQUIRES(
((Mutable_RandomAccessIterator<I>))           //检查列表没有逗号分隔，结束使用逗号
((LessThanComparable<typename RandomAccessIterator<I>::value_type>)),
(void)                          )
_sort(I first, I last)
{     std::stable_sort(first, last);}
```

12.3.9 概念原型类

为了方便测试验证泛型代码，concept_check 库提供了一些精确匹配标准库概念的最小类型——称为原型类（archetype class），因为它们足够小，因而能够比普通类（通常具备更多的特性）更严格地测试泛型代码。

使用概念原型需要包含额外的头文件：①

```
#include <boost/concept_archetype.hpp>
```

concept_check 库目前有基本概念原型、函数对象概念原型和迭代器概念原型，暂时还没有容器概念原型。原型类与概念检查类基本是一一对应的，故在此不做列举。

使用概念原型相当于用这些原型类自行构造出一套类型系统，然后把这个原型类型系统送入到泛型代码中进行测试。例如，下面的代码测试了 my_min() 和 _sort() 函数：

```
//定义可拷贝、赋值和比较的基本原型
typedef null_archetype<>                          T;
typedef assignable_archetype<T>                   at;
typedef copy_constructible_archetype<at>          cat;
typedef less_than_comparable_archetype<cat>       vt;

//使用 boost::detail::dummy_constructor 初始化
boost::detail::dummy_constructor dummy_cons;
vt v1(dummy_cons), v2(dummy_cons);

my_min(v1, v2);

//定义可变随机访问迭代器原型
typedef mutable_random_access_iterator_archetype<vt> rt;
rt begin, end;
_sort(begin, end);
```

这段代码中如果把 mutable_random_access_iterator_archetype 改为 random_access_iterator_archetype，那么将导致编译错误，因为原型类精确地描述了所需的概念。

① 新式迭代器概念原型需要包含头文件<boost/iterator/iterator_archetypes.hpp>，其用法与标准迭代器原型略有不同，需要制定值类型、访问类型和遍历类型三个模板参数。

下面的代码示范了标准库迭代器原型和 Boost 迭代器原型的测试（需要包含头文件 <boost/iterator/iterator_archetypes.hpp>）：

```
typedef copy_constructible_archetype<            //可拷贝赋值的原型类
            assignable_archetype<>>      T;
typedef input_iterator_archetype<T>      I;      //标准的输入迭代器

BOOST_CONCEPT_ASSERT((ReadableIterator<I>));
BOOST_CONCEPT_ASSERT((SinglePassIterator<I>));
BOOST_CONCEPT_ASSERT((InputIterator<I>));

//使用 Boost 的迭代器原型定义输入迭代器
typedef boost::iterator_archetype<T,
  boost::iterator_archetypes::readable_iterator_t,
  boost::single_pass_traversal_tag > II;

BOOST_CONCEPT_ASSERT((ReadableIterator<II>));
BOOST_CONCEPT_ASSERT((SinglePassIterator<II>));
BOOST_CONCEPT_ASSERT((InputIterator<II>));
```

12.4 总结

在本章中，我们讨论了 Boost 库中的三个泛型编程组件：enable_if、call_traits 和 concept_check，使用它们可以很好地提高泛型代码的质量。

enable_if 用于编写模板函数或模板类，在编写泛型代码时可以主动地控制编译器的行为，把不希望出现的形式从重载决议中去掉。它比简单地使用静态断言效果更好，因为静态断言仅仅是验证了某些编译期条件，并不能阻止函数或模板类的重载形式。Boost 库里的许多组件都使用了 enable_if，虽然我们在实际开发中不一定会用到它，但理解它可帮助我们深入理解其他 Boost 组件的工作原理。

call_traits 是一个比较简单的泛型编程工具，它可以计算类型 T 相关的各种类型，所以很有用，特别是在大型工程中——可以很好地保证函数接口的清晰性和正确性，并且始终高效。

concept_check 以库的方式提供了丰富的概念检查功能，用途非常广泛，只要我们编写泛型代码就可以使用概念检查库来约束模板参数，要求它必须满足某些要求。灵活使用概念检查，再结合 static_assert、type_traits 等其他工具可以有效地保证泛型代码的正确性。

第13章 模板元编程

在第 2 章中我们初步学习了模板元编程的基本知识，本章我们将更加深入地研究 C++中这一最为强大的编程武器，即 boost.mpl（meta programming library）。

mpl 是模板元编程的主要工具，它是整个 C++模板元编程的核心，也是理解 Boost 中许多其他组件工作原理的基础。熟悉并掌握 mpl 可以让我们把工作更多地放在编译期，更好地发挥 C++静态类型体系的优越性，开发出效率更高的运行时程序。

因为 mpl 十分庞大，故本书在这里只能择要介绍其中的部分内容，希望读者可以触类旁通。

13.1 概述

mpl 是专门为模板元编程创建的工具库，它以基本的元编程概念为基础，辅以预处理元编程和其他几个基本库（如 type_traits），逐步发展出编译期的整数类型、类似 STL 的容器和算法，甚至还有编译期的 lambda 表达式，在 C++已有的语法体系中独立构造出了一个崭新的、完整的元编程体系。

mpl 提供了大量高质量高效率的元编程工具，它们大大降低了元编程的难度，使用这些高级元编程工具会使元编程工作更加轻松，这些工具包括：

- 基本的数据类型　　：整数、pair、void 等运行时类型的对应物。
- 基本的数据运算　　：以元函数形式提供的算术运算、逻辑运算等运算功能。
- 程序流程控制　　：以元函数形式提供的分支流程处理功能。

- 容器 ：模仿 STL 风格的存储元数据（类型）的编译期容器。
- 视图 ：容器的适配器，可以简化容器的操作。
- 迭代器 ：模仿 STL 风格应用于容器的编译期迭代器。
- 算法 ：模仿 STL 风格应用于容器和迭代器的编译期算法。
- 高阶元数据 ：类似于函数对象的元编程构件。
- bind/lambda 表达式：编译期的 lambda 表达式，功能强大灵活。
- 调试支持 ：提供了编译期的断言功能。
- 辅助宏 ：各种配置宏和 traits 宏。

mpl 是一个仅由头文件组成的库，无须编译，所有的组件均位于名字空间 boost::mpl，源代码则在目录<boost/mpl/>下，文件名与组件的名字通常是一致的。

由于不存在一个统一的头文件，所以使用元编程组件时必须手工添加包含的头文件，这样虽然比较麻烦，但也减少了不需要包含的头文件，一并减少了元计算（即编译）的时间：

```
#include <boost/mpl/xxx.hpp>                //所需的 mpl 元编程头文件
using namespace boost::mpl;                 //打开名字空间
```

13.2 整数类型

整数是任何编程方式都需要处理的数据，模板元编程当然也不例外，由于整数本身就是元数据，在编译期可以计算，所以元程序操作起来毫无困难。但直接操作整数对于元编程并不太方便，因为元编程更大的作用是类型计算，而整数并不是类型，所以直接使用整数计算不能够体现元编程类型计算的好处。

mpl 库为此提供了数值包装器概念，可以把整数（包括 int、long、bool 等类型）包装为值元函数，这样元程序就能够把整数用于类型计算，并基于这些高级整数类型建立起整个元编程计算体系。

13.2.1 简介

mpl 库里的数值包装器元函数共六个，分别包装了如下不同的整数类型（注意没有 short）。

- char_<N> ：包装 char 类型，值为 N。
- int_<N> ：包装 int 类型，值为 N。
- long_<N> ：包装 long 类型，值为 N。
- size_t<N> ：包装 size_t 类型，值为 N。
- integral_c<T,N> ：包装类型为 T 的整数类型，值为 N。
- bool_<N> ：包装 bool 类型，值为 N。

这些元函数都具有基本相同的形式，类摘要如下：[①]

```
struct integral_wrapper                                     //整数包装器
{
    BOOST_STATIC_CONSTANT(T, value = N);                    //包装整数值 N
    typedef integral_c_tag      tag;                        //类型标记
    typedef T                   type;                       //返回自身
    typedef T                   value_type;                 //整数类型
    operator T() const { return this->value; }              //转型操作

    //注意，bool_没有下列两个元数据
    typedef some_define         next;                       //++N
    typedef some_define         prior;                      //--N
};
```

数值包装器的功能比较简单，可以用::type 返回自身，用::value 返回被包装的整数值。对于非 bool 类型还提供了 next 和 prior 两个内部类型定义，可以获得类似于 operator++ 和 operator--的效果，在元计算时递增或递减整数，另外我们也可以使用辅助元函数 next 和 prior，效果相同但更加通用（参见 13.2.4 节）。

数值包装器同时重载了转型操作，所以它们的实例可以在运行时隐式转换为被包装的整数，像真正的整数一样被使用。为了称呼方便，以下有时会将这些包装器元函数简称为“整数”或者“整数类型”，读者需自行辨析它在具体语境中的含义。

① 实际上除了 bool_之外，其他整数类型都是使用头文件<boost/mpl/aux_/integral_wrapper.hpp>进行宏预处理实现的。

13.2.2 整数类型

整数类型的用法基本相同，所以在这里我们仅介绍 int_ 和 integral_c，这两个元函数的类摘要如下：

```
template<int N >
struct int_
{...};                                    //见上一小节 integral_wrapper 的代码

template<typename T, T N >
struct integral_c
{...};                                    //见上一小节 integral_wrapper 的代码
```

int_ 包装了标准的 int 类型，它使用::value 返回模板参数 N，而 integral_c 因为支持任意整数类型所以有两个模板参数，使用::value 返回类型为 T 的模板参数 N。

示范整数类型包装器用法的代码如下：

```
typedef int_<2> i2;                                   //包装 int 型整数 2
typedef integral_c<short, 2> s2;                      //包装 short 型整数 2

assert(i2::value == 2);                               //获取整数值
assert(i2::value == s2::value);                       //两个类型的值相等

assert((is_same<i2::type, i2>::value));               //返回包装器自身
assert((is_same<s2::value_type, short>::value));      //返回整数类型

assert(i2::next::value == 3);                         //内部成员获取递增值
assert(prior<s2>::type::value == 1);                  //使用元函数获取递减值

i2 two1;                                              //声明两个包装器实例
s2 two2;                                              //值均为常量 2

int i = two1 + two2;                      //隐式类型转换为 int 参与运行时计算
assert(i == int_<4>());                   //与 int_元函数比较，使用()创建临时对象
```

在这段示例代码中，我们混合了编译期的元编程和运行时的普通编程。需要注意的是编译期包装器不能直接与整数进行比较，必须使用::value 获取整数值才能与整数比较。元函数 next/prior 返回的是包装器，所以要先用::type 获取元函数返回值才能调用::value。运行时由于包装器有隐式类型转换，所以实例可以直接参与整数运算。

13.2.3 bool 类型

bool 包装器 bool_是一个比较特殊的元函数，因为它的取值只有 true/false 两个值，而且也不能递增或递减，它的类摘要如下：

```
template< bool C_ >
struct bool_
{
    BOOST_STATIC_CONSTANT(bool, value = C_);            //包装bool值C_
    typedef integral_c_tag    tag;                      //类型标记
    typedef bool_             type;                     //返回自身
    typedef bool              value_type;               //bool类型
    operator bool() const { return this->value; }       //转型操作
};
```

bool_的用法与 int_和 integral_c 差不多甚至更加简单，只是需要注意没有 next/prior 成员，也不能使用 next/prior 元函数。为了方便使用，mpl 还提供了两个快捷 typedef：

```
typedef bool_<true>          true_;
typedef bool_<false>         false_;
```

示范 bool_用法的代码如下：

```
assert(true_::value == true);
assert(false_::value == false);

assert((is_same<true_::type, bool_<true> >::value));
assert((is_same<false_::value_type, bool>::value));

next<true_>::type;                                    //不能执行递增操作，编译错误
```

13.2.4 基本运算

同 C++内建的整数运算支持一样，mpl 库也对这些元编程整数类型提供了等价的编译期运算支持——当然，对这些整数类型的运算只能使用元函数。

mpl 库对整数类型运算的支持是非常全面的，不仅有我们之前看到的递增递减运算，还包括算术运算、比较运算等，详细信息如下所述。

■ 递增递减运算 ：位于头文件<boost/mpl/next_prior.hpp>，包括 next 和 prior

两个元函数，不支持 bool_。

- 算术运算 ：位于头文件<boost/mpl/arithmetic.hpp>，包括加减乘除、取模、取负值，支持使用多个参数，元函数有 plus、minus、negate 等。
- 比较运算 ：位于头文件<boost/mpl/comparison.hpp>，包括小于、大于、等于和不等于等六种，元函数有 less、greater、equal_to 等。
- 位运算 ：位于头文件<boost/mpl/bitwise.hpp>，包括与、或、异或、移位等，元函数有 bitand_、bitor_等。
- 逻辑运算 ：位于头文件<boost/mpl/logical.hpp>，包括与、或、非三种逻辑运算，元函数有 and_、or_和 not_。
- 其他运算 ：包括最大最小值 min/max、整数的大小 sizeof_等。

可见 mpl 为模板元编程提供了完整的整数运算功能，能够在编译期执行任何整数计算工作。

由于整数运算元函数很多，不可能一一介绍，故下面仅列出它们的基本形式：

```
template<T1, ...>                            //元函数参数列表
struct some_operation                        //元函数名称
{
   typedef some_define type;                 //返回运算后的整数类型
};
```

示例代码

示范这些整数运算元函数的代码如下：

```
typedef int_<2> i2;                                              //定义三个整数类型
typedef int_<5> i5;
typedef int_<7> i7;

assert((plus<i2, i5, i7>::type::value == 14));                   //加法
assert((equal_to<minus<i7, i5>::type, i2>::type::value));        //减法

assert((less<i2, i7>::type::value));                             //小于比较
assert((is_same<greater<i5, i2>::type, true_>::value));          //大于比较

assert((not_<and_<true_, false_>::type>::type::value));          //逻辑与
assert((or_<true_, false_>::type()));                    //逻辑或，使用了隐式类型转换
```

这段代码比较简单，读者需要注意减法运算中我们使用的是相等比较 equal_to 元函数，而不是之前我们常用的 is_same。这是因为整数的算术运算结果不一定是与参数同类型的整数，因为有可能混用不同的整数类型，所以结果通常都是一个 integral_c 类型，只能使用 equal_to 来比较数值的等价，例如：

```
//混用 int_和 long_类型的加法，定义运算结果元数据
typedef plus<int_<1>, char_<2>, long_<3>>::type  result;

assert(!(is_same<result,long_<6>>::type::value));    //结果不是 long_类型

assert((is_same<result,
  integral_c<long, 6>>::type::value));               //结果是 integral_c 类型
```

13.3 流程控制

元程序本质上也是程序，它也具有顺序、分支和循环三类程序结构。但它不同于普通的运行时程序，属于函数式编程，没有循环和跳转语句，最常用的循环手段是递归然后模板特化终止。

对于分支结构，mpl 提供了四个元函数，它们类似于运行时的 if-else 语句。

13.3.1 if_和 if_c

if_和 if_c 位于头文件<boost/mpl/if.hpp>，是两个最简单的分支元函数，可以近似地看作?:操作符，能够像 if-else 语句一样根据条件执行不同的分支，类摘要如下：

```
template<bool C, typename T1, typename T2>
struct if_c                                   //if_c 元函数的标准形式
{
   typedef T1 type;                           //C 为 true 时返回第一个元数据
};
template<typename T1, typename T2>
struct if_c<false,T1,T2>                      //if_c 对 false 模板偏特化
{
   typedef T2 type;                           //C 为 false 时返回第二个元数据
};

template<typename C, typename T1, typename T2>
struct if_                                    //if_元函数
```

```
{
    typedef if_c<C::value, T1, T2> almost_type_; //调用 if_c 元函数
    typedef typename almost_type_::type type;
};
```

if_c 和 if_非常相似，不同的是 if_c 的条件是一个 bool 类型，而 if_的条件是一个有::value 返回的值元函数(不一定是 bool_)。两者的应用范围各有不同，对于 type_traits、mpl 等库中的大量元函数来说，if_因为可以少写一个::value 会更方便一些。[①]

简单示范 if_c 和 if_用法的代码如下：

```
typedef if_c<true, int, long>::type mdata1;              //使用 if_c 计算
assert((is_same<mdata1, int>::value));                    //得到元数据 int

typedef if_<false_, float, double>::type mdata2;         //使用 if_计算
assert((is_same<mdata2, double>::value));                 //得到元数据 double

typedef if_<is_integral<mdata2>,                        //使用 is_integral 作为条件
    integral_promotion<mdata2>::type,                   //提升整数类型
    floating_point_promotion<mdata2>::type              //提升浮点数类型
>::type mdata3;

assert((is_same<mdata3, double>::value));               //得到元数据 double
```

if_c 和 if_并非全无缺点，因为它们在元计算时必须计算出所有的元数据，而不能根据条件有选择地“忽略”不需要计算的数据（即缓式评估），增加了元计算的时间。

13.3.2 eval_if 和 eval_if_c

eval_if 和 eval_if_c 位于头文件<boost/mpl/eval_if.hpp>，弥补了 if_c 和 if_不能缓式评估的缺陷，它们专门用于计算元函数，可以根据条件只计算需要的部分，类摘要如下：

```
template<typename C, typename F1, typename F2>
struct eval_if                                  // eval_if 元函数
    : if_<C,F1,F2>::type                        //元函数转发给 if_，并得到返回结果
{};
```

① if_c 元函数等价于 type_traits 库中的 conditional 元函数，参见 3.5.2 节。

```
template<bool C, typename F1, typename F2>
struct eval_if_c                                // eval_if_c 元函数
    : if_c<C,F1,F2>::type                       //元函数转发给 if_c，并得到返回结果
{};
```

eval_if 和 eval_if_c 的实现代码非常简单，注意它们不仅仅是元函数转发，同时还使用::type 得到了 if_c 和 if_的计算结果（即 F1 或 F2），所以使用 eval_if::type 或 eval_if_c::type 就相当于 F1::type 或 F2::type，方便了很多。

eval_if 和 eval_if_c 的后两个模板参数必须是元函数，在使用的时候需要特别注意，它们名字中的“eval”清楚地表明了这一点。

eval_if 和 eval_if_c 的示范代码如下，与上一小节的很相似：

```
typedef eval_if_c<true,                         //使用 eval_if_c 计算
  identity<int>, identity<long>>::type mdata1;
assert((is_same<mdata1, int>::value));

typedef eval_if<false_,                         //使用 eval_if 计算
  identity<float>, identity<double>>::type mdata2;
assert((is_same<mdata2, double>::value));

typedef eval_if<is_integral<mdata2>,            //使用 is_integral 作为条件
  integral_promotion<mdata2>,
  floating_point_promotion<mdata2>
>::type mdata3;
assert((is_same<mdata3, double>::value));
```

在这段代码中，我们使用了另外一个元函数 identity，它是一个很小的辅助元函数，直接返回参数自身。

13.4 容器

作为一个完整的模板元编程框架，mpl 不仅拥有整数类型和流程控制，C++标准库中的容器也有对应的元编程版本，这就是 mpl 容器。[1]

① 在 mpl 库中，容器称为“序列”(sequence)，并且有自己的概念、分类和定义，与标准库相似但也有不同，本书为了方便学习沿用了标准库的描述方式，不涉及更复杂的概念。

13.4.1 简介

mpl 中的容器与标准库中的容器很相似，所以可以把它们对比研究，这样更容易学习。

mpl 容器可分为序列容器和关联容器两类，当然，容器里容纳的元素都是元数据——也就是类型，所以没有标准容器对元素的可拷贝可赋值的要求，并且元素可以是任意类型（有点类似于 tuple）。mpl 容器可以添加删除元素，也有迭代器的概念，也可以在这些容器上使用算法，这些都与标准容器很相像。但 mpl 容器与标准容器的一个重要区别是它们虽然也是类型（元数据），但没有成员函数（这是运行时的概念），自身没有操纵容器内元素的功能，只能通过外部元函数才能处理元数据。

mpl 提供几乎与标准容器完全等价的序列容器和关联容器，它们同时也是元函数，能够以::type 返回自身。

mpl 序列容器包括以下内容。

- list ：不同于 std::list，它是一个单向链表，只能在序列的前端操作元素，可以容纳无限个元素。
- vector：它非常像 std::vector，是一个可以随机访问的序列，但它又具有一些 std::list 的特性，可以在序列的两端操作元素。另外不同于 std::vector 的一点是它的容量是有限的，不能无限增加，缺省最多能够容纳 20 个元数据。
- deque ：它类似 std::deque，可以在序列两端操作元素，与 vector 几乎是相同的。

mpl 关联容器缺省最多能够容纳 20 个元素，包括以下两种。

- set ：类似 std::multiset，是一个允许重复的元数据集合。
- map ：类似 std::multimap，是一个允许重复的元数据映射关系集合。

除了以上五个基本容器外，mpl 库还针对整数类型（数值包装器）提供了一些特别的整数容器。

- range_c<T,N,M> ：不可修改的包含[N,M]区间内整数的 vector 容器。
- list_c<T,...> ：包含类型为 T 的若干个整数的 list 容器。
- vector_c<T,...> ：包含类型为 T 的若干个整数的 vector 容器。

- set_c<T,...> ：包含类型为 T 的若干个整数的 set 容器。
- string ：专门存储 char 字符的容器，类似 std::string，可以在编译期处理的字符串。

判断一个类型是否是 mpl 容器可以使用值元函数 is_sequence，例如：

```
assert(!is_sequence<int>::value);
assert((is_sequence<mpl::vector<>>::value));
assert((is_sequence<mpl::set<int,char>>::value));
```

由于 mpl 容器较多，而且实现较复杂，我们只讲解较为常用的 vector、string 和 map，对于容器之上的视图（view）概念本书不做介绍。

13.4.2 vector

vector 容器很像是 boost::tuple 的编译期版本，可以容纳异种类型（元数据），是最容易使用的 mpl 容器。

由于 vector 使用了预处理元编程，它的实现代码很难直接给出，其示意形式如下：

```
template< t1, t2,...>                                  //容纳若干个元数据
struct vector{ ...};                                   //具体实现代码无法给出
```

我们可以在编译期使用相关的元函数处理 vector 里的元数据，如判断容器的大小、获取前端和末端的元素、添加或者删除元素等。示范 vector 用法的代码如下，其中涉及的元函数参见 13.4.5 节：

```
typedef mpl::vector<char, int , long,                  //容纳五个元数据的 vector
  int_<8>, std::string> vec;                           //可以不必使用::type

assert((empty<vec>::value == false));                  //判断容器不空
assert((size<vec>::value == 5));                       //检查容器的大小

assert((is_same<front<vec>::type, char>::value));          //访问前端元素
assert((is_same<back<vec>::type, std::string>::value));    //访问末端元素
assert((is_same<at_c<vec, 1>::type, int>::value));         //随机访问元素

typedef push_front<vec, float>::type vec2;                 //前端添加元素 float
assert((size<vec2>::value == 6));                          //元素数量加 1
```

```
assert((is_same<front<vec2>::type, float>::value));       //访问前端元素

typedef clear<vec2>::type vec3;                           //清空容器
assert((empty<vec3>::value));                             //判断容器已经被清空
```

从这段代码中，我们可以看到 vector 的用法与 std::vector 的用法非常相似，只是它容纳的是 C++的类型元数据，必须在编译期使用元函数以自由函数而不是成员函数的方式操作。

另外还需要注意，因为容器是元数据，而元数据是不可变的，所以对容器的任何变动操作都必须返回一个新的容器，原容器不会变化。

13.4.3 string

string 是一类特殊的 mpl 序列容器，它专门容纳 char 类型的字符，相当于字符串常量的元编程版本，它的示意形式如下：

```
template<c1, c2, ...>                               //容纳若干个字符数据
struct string{ ...};                                //具体实现代码无法给出
```

string 的模板参数较为特别，它们可以是任意多组使用**单引号**（注意！）的 char 型字面量，由于受到预处理的限制每组字符最多只能有 4 个字符，因此如果要容纳一个较长的字符串就必须适当分组，例如：

```
typedef mpl::string<'a','b','c'> abc;               //字符串是"abc"
typedef mpl::string<'Hell','o !!'> hello;           //字符串是"Hello !!"
typedef mpl::string<'Hello !!'> hello2;             //编译错误，字符过多
typedef mpl::string<"Hello !!"> hello3;             //编译错误，使用了双引号
```

string 容纳的字符串长度也是有限制的，mpl 缺省配置最大能够容纳 32 个字符，如果需要可以在<boost/mpl/string.hpp>前使用宏 BOOST_MPL_LIMIT_STRING_SIZE 更改。

string 具有同 vector 一样的功能，也可以判断字符串的大小、获取前端和末端的元素、添加或者删除元素，这里的元素都是 char_类型。除了这些操作之外，string 还有一个特别的值元函数 c_str，它可以用::value 返回一个以 NULL 结尾的标准字符串，可以在运行时使用。

示范 string 用法的代码如下：

```
typedef mpl::string<'Hell','o !!'>   hello;             //一个字符串容器

assert((empty<hello>::value == false));                 //判断非空
assert((size<hello>::value == 8));                      //字符串长度为 8
```

```
assert((front<hello>::type::value == 'H'));          //取第一个字符
assert((back<hello>::type::value == '!'));           //取最后一个字符

typedef push_back<hello, char_<'?'>>::type hello2;   //追加一个字符
assert((c_str<hello2>::value ==                 //使用c_str获取编译器字符串常量
  std::string("Hello !!?") ));                  //与运行时std::string比较
```

13.4.4 map

map 本质上与 vector、string 差不多，都是容纳元数据的容器，但它的元素类型却比较特别，是一个 mpl::pair 类型，里面包含了键—值的映射关系。

mpl::pair 很像 std::pair，不同的是它把元数据作为成员，其定义如下：

```
template<typename T1, typename T2>
struct pair
{
    typedef pair  type;                       //返回自身
    typedef T1    first;                      //第一个成员
    typedef T2    second;                     //第二个成员
};
```

因为 pair 是一个 struct，所以我们可以使用::first 或::second 直接获取其中的两个成员，也可以使用元函数 first 和 second 间接获得。

map 使用 mpl::pair 作为元素，其示意代码如下：

```
template<p1, p2, ...>                         //容纳若干个键—值对元数据
struct map{ ...};                             //具体实现代码无法给出
```

map 具有大多数容器的共通操作，如判断容器的大小、添加或者删除元素等，但它只能获取前端元素，没有操纵末端元素的 back 元函数，此外还有一些专门用于键—值操作的专用元函数，如 has_key、count 等。

示范 map 用法的代码如下：

```
typedef mpl::map<                                   //定义map容器
  mpl::pair<int, std::vector<int>>,                 //以下是三个pair
  mpl::pair<char,std::string>,                      //元数据可以任意映射
  mpl::pair<int_<3>, float[3]>                //一个数值包装器映射到一个浮点数组
> m;                                                //map容器的名字是m
assert((empty<m>::value == false));                 //判断非空
```

```
assert((size<m>::value == 3));                          //容器大小为 3

typedef front<m>::type p1;                              //获取第一个元素
assert((is_same<first<p1>::type, int>::value));         //使用 first 元函数
assert((is_same<p1::second,std::vector<int>>::value));  //直接取成员

assert((has_key<m, char>::value));                      //检查键是否存在
assert((count<m, int_<0>>::value == 0));             //count 算法计算键的个数

typedef at<m, char>::type mdata;                        //根据键获取值
assert((is_same<mdata, std::string>::value));
```

13.4.5 相关元函数

mpl 为容器提供了数个元函数，它们的功能基本与容器的同名成员函数等价，常用的元函数可以按如下分类。

1）容器容量操作。

- empty<C> ：值元函数，检查容器是否为空。
- size<C> ：值元函数，获取容器的大小。

2）元素访问操作。

- front<C> ：返回容器里的第一个元素。
- back<C> ：返回容器里的最后一个元素，对于 list 和关联容器不适用。
- at<C,k> ：对于序列容器，返回第 k 个位置上的元素（k 是个数值包装器），对于关联容器，返回键为 k 的元素；
- at_c<C,n> ：仅适用于序列容器，返回第 n 个位置上的元素（n 是 long 型整数）。

3）迭代器操作（参见 13.5 节）。

- begin<C> ：返回一个容器对应的迭代器，指向第一个元素的位置。
- end<C> ：返回一个容器对应的迭代器，指向最后一个元素之后的位置（逾尾）。

4）元素变动操作。

- push_front<C,t> ：在容器前端插入一个元素，返回新容器。
- push_back<C,t> ：在容器末端插入一个元素，返回新容器。
- pop_front<C> ：删除容器的第一个元素，返回新容器。
- pop_back<C> ：删除容器的最后一个元素，返回新容器。
- clear<C> ：清除容器里的所有元素，返回新容器。
- erase<C,pos>/erase<C,f,l> ：删除容器中某个位置或者某个区间里的元素，其中的 pos、f 和 l 都是容器的迭代器。
- insert<C,pos,t>/insert<C,t> ：向序列容器的 pos 位置或关联容器里插入一个元素，其中的 pos 是容器的迭代器。

5）关联容器特有操作。

- has_key<C,k> ：检查容器中是否有键为 k 的元素。
- erase_key<C,k> ：删除容器中键为 k 的元素。

这些元函数与标准库的容器操作很类似，用法也比较简单，示范代码可见前几节对容器的操作示例。

13.5 迭代器

在 mpl 中迭代器同样占有非常重要的地位，它是 mpl 容器和 mpl 算法之间的粘接剂，本节我们简要研究 mpl 中的元编程迭代器，读者可与第 5 章对比参考。

13.5.1 简介

mpl 中的迭代器也遵循迭代器设计模式，与标准迭代器和 Boost 新式迭代器不同的地方仅在于它是编译期的迭代器，只能使用元函数操纵。

mpl 迭代器具有基本的操作接口，包括解引用、前进、后退、计算距离等，也能够按照取值和遍历操作进行分类，但因为元数据都是不可变的，所以所有的 mpl 都是只读迭代器，我们仅需以遍历方式进行分类。

mpl 迭代器可分为以下三类，与 Boost 新式迭代器颇类似但要更简单。

- 前向迭代器　　：可以使用元函数 next 前进。
- 双向迭代器　　：在前向迭代器的基础上增加使用元函数 prior 后退，即可逆向遍历。
- 随机访问迭代器：在双向迭代器的基础上增加迭代器的距离运算。

有了迭代器的概念，mpl 容器也可以按照提供的迭代器进行分类，具体分类如下所述。[①]

- 前向序列　　：所有的容器都符合前向序列的概念，其迭代器可以执行递增操作。
- 双向序列　　：string、vector 和 deque 符合双向序列的概念。
- 随机访问序列：vector 和 deque 符合随机访问序列的概念。

13.5.2 相关元函数

与运行时的迭代器一样，mpl 迭代器也可以执行解引用、前进、后退等操作，只不过这些操作返回的都是编译期的元数据。

mpl 提供的迭代器元函数包括以下几个。

- deref<I>　　：解引用迭代器，返回元数据。
- next<I>　　：前进迭代器。
- prior<I>　　：后退迭代器。
- advance<I,n>　　：迭代器移动 n 个位置，对于双向迭代器 n 可以是负值。
- distance<I1,I2>　　：返回两个迭代器之间的距离。
- iterator_category<I>：获得迭代器的分类标志。

关于元函数 next/prior 要做一点特别的说明，它们实际上与 13.2 节操作整数的 next/prior 是完全相同的。这是由于泛型编程的静态多态特性，只要元数据有内部的::next/::prior 定义，对于整数和迭代器就可以分别执行不同的操作。

mpl 的迭代器通常可以用容器的 begin 和 end 元函数获取，示范用法的代码如下：

```
typedef mpl::vector<char, int , long,                    //容纳五个元数据的 vector
    int_<8>, std::string> vec;
```

① mpl 中容器的分类还有关联性和可扩展性两个概念，本书从略。

```
typedef begin<vec>::type i1;                        //获得随机访问迭代器
assert((is_same<deref<i1>::type, char>::value));    //解引用迭代器

typedef next<i1>::type i2;                          //前进迭代器
assert((is_same<deref<i2>::type, int>::value));     //解引用迭代器
assert((is_same<i1, prior<i2>::type>::value));      //后退迭代器并比较

typedef advance<i2, int_<4>>::type i3;              //前进迭代器至末尾
assert((distance<i1, i3>::type::value == 5));       //计算迭代器间距离
assert((distance<i3, i1>::type::value == -5));
assert((is_same<i3, end<vec>::type>::value));       //与逾尾迭代器比较

typedef insert<vec, i2, float>::type vec2;          //在i2的位置插入一个元数据
```

13.6 算法

mpl 基于编译期的容器和迭代器提供了大量的算法，使得我们可以对存储在容器中的元数据执行复杂的计算，许多算法与标准库是对应的。

由于 mpl 中的算法很多，本节仅择要介绍一些较常用的算法，另一些算法参见 13.7.5 节。

13.6.1 插入器

mpl 中的插入器类似标准库中的插入迭代器，它可以把一个容器适配成迭代器供算法操作。

mpl 插入器共有以下三个。

- back_inserter<C>　：在容器后端插入元素，要求容器支持 push_back。
- front_inserter<C>：在容器前端插入元素，要求容器支持 push_front。
- inserter<S,Op>　　：通用的插入器，以初始状态 S 开始执行 Op 操作完成插入动作。

这三个插入器中最常用的是前两个，front_inserter 和 back_inserter，而第三个 inserter 通常需要配合元编程 lambda 表达式使用。

示范插入器用法的代码如下：

```
typedef mpl::vector<char, int , long> vec;         //容纳三个元素
```

```
typedef copy<vec,
  mpl:back_inserter<vec>>::type vec2;              //插入自己，相当于双倍
assert((size<vec2>::value == 6));

typedef mpl::string<> str1;                        //编译期空字符串
typedef copy<
  mpl::string<'time'>,                             //使用一个“临时”元数据
  mpl::front_inserter<str1>>::type str2;           //前端插入

cout << c_str<str2>::value;                        //输出字符串 emit
```

13.6.2 查询算法

查询算法也可以称为只读算法，因为它们只是以只读的方式访问容器里的元素，不会变动容器。这些算法与标准算法非常相似，常用的有以下几个。

- find<C,t> ：在容器中查找元素 t，返回迭代器。
- contains<C,t> ：值元函数，检查容器中是否存在元素 t。
- count<C,t> ：值元函数，返回容器中元素 t 的个数。
- equal<C1,C2> ：值元函数，比较两个容器是否等价，即元数据相同。

我们需要注意算法 equal，不同于 type_traits 库的元函数 is_same，它比较的是两个容器的等价性，两个等价的 mpl 容器完全有可能是两个不同的类型，所以比较容器时通常应该使用 equal 算法而不是 is_same 元函数。

示范这些算法的代码如下：

```
typedef range_c<int, 0, 10> rc;                    //定义一个整数区间
assert((size<rc>::value == 10));

//使用宏简化容器内元素的写法
#define INTV(n)   integral_c<rc::value_type, n>

assert((deref<                                     //解引用迭代器或者元素
  find<rc, INTV(0)>::type                          //查找容器内的元素
  >::type::value == 0));

assert((contains<rc, INTV(9)>::value));            //检查元素是否存在
```

```
assert((!contains<rc, INTV(10)>::value));              //检查元素是否存在

assert((count<rc, INTV(5)>::value == 1));              //计算元素的数量
assert((count<rc, INTV(-5)>::value == 0));             //计算元素的数量

typedef vector_c<int, 0,1,2,3,4,5,6,7,8,9> vec;        //定义一个整数容器
assert((equal<rc, vec>::value));                       //等价比较
assert((!is_same<rc, vec>::value));                    //两者是不同的类型
```

在这段代码中，我们定义了一个宏 INTV 来简化代码，这是因为 range_c 等整数容器内部存储的元素都是 integral_c，而不是 int_、long_等类型，所以在查询元素时必须使用相同类型的元素，如果直接使用 int_等类型则会产生编译错误。

13.6.3 变换算法

变换算法处理容器中的全部或部分元素，然后返回一个新的容器，带前缀“reverse_”的形式可以返回操作后的逆序容器。

mpl 中的部分变换算法如下所述。

- copy<From,To>　　　　　　：拷贝一个容器里的所有元素。
- replace<From,Old,New,To>：把容器中的 Old 元素全部替换为 New 元素。
- remove<From,t,To>　　　　：移除容器中所有值为 t 的元素。
- reverse<From,To>　　　　　：逆序拷贝容器里的所有元素，相当于 reverse_copy。

这四个算法都比较简单，需要注意的是算法中的最后一个参数 To，它可以省略不用。如果不使用 To 参数，那么算法直接用::type 返回变换后的新容器；如果使用 To 参数，那么通常需要搭配插入器工作。

示范算法的代码如下：

```
typedef mpl::vector<char, short, int, long> vec;          //容纳四个元素
typedef mpl::replace<vec,                                 //替换 vec 中的元素
      int, char>::type vec2;                              //int 替换为 char

assert((mpl::count<vec2, char>::value == 2));             //容器里有两个 char
assert((is_same<deref<                      //使用迭代器取第 3 个位置的元素比较验证
        mpl::advance<begin<vec2>::type, int_<2>>::type>::type,
```

```
        char>::value));

typedef mpl::remove<vec2, short>::type vec3;          //删除 short 元素
assert((!contains<vec3, short>::value));              //容器中不存在 short 元素

assert((mpl::equal<                                   //比较等价性
  mpl::reverse<vec3>::type,                           //逆序拷贝
  mpl::vector<long, char, char>>::value));            //一个“临时”容器
```

13.6.4 运行时算法

mpl 库中有一个特别的 for_each 算法，它工作在**运行时**，可以遍历类型容器，调用一个函数对象操作类型容器里类型对应的实例对象。

for_each 算法是一个模板函数，其声明如下：

```
template<typename Sequence, typename F>
void for_each(F f);
```

for_each 算法有两个模板参数：第一个参数 Sequence 是 mpl 容器，必须显式指定；第二个参数 F 是一个函数对象，它应该具有模板成员函数 operator()，能够处理 Sequence 中的所有类型，否则会发生编译错误。

为了示范 for_each 算法的用法，我们先定义两个函数对象：

```
struct mpl_func1
{
  template<typename T>
  void operator()(T t)                                //输出类型名
  {   cout << typeid(t).name() << endl;  }
};

struct mpl_func2
{
  template<typename T>
  void operator()(T t)                                //输出整数类型的值
  {
      if (is_same<typename tag<T>::type, integral_c_tag>::value)
      {   cout << t << ',';}
  }
```

```
};
```

第一个函数对象 mpl_func1 很简单，仅仅是使用 typeid 获取类型的名称，第二个函数对象 mpl_func2 略复杂一些，它使用元函数 tag 获取了类型的标志，仅当类型是一个 mpl 整数时才输出。

示范 for_each 算法用法的代码如下：

```
typedef range_c<int, 0, 5> rc;                          //定义一个整数范围

typedef mpl::vector<> vec;                              //一个空类型容器

typedef mpl::copy<rc,                                   //copy 算法拷贝整数到空容器
  mpl::back_inserter<vec>>::type vec2;

typedef push_front<vec2, float>::type vec3;             //容器前端添加一个元素

mpl::for_each<vec3>(mpl_func1());                       //输出容器中的类型信息
mpl::for_each<vec3>(mpl_func2());                       //输出容器中的整数
```

代码的运行结果如下：

```
float
struct boost::mpl::integral_c<int,0>
struct boost::mpl::integral_c<int,1>
struct boost::mpl::integral_c<int,2>
struct boost::mpl::integral_c<int,3>
struct boost::mpl::integral_c<int,4>
0,1,2,3,4,
```

13.7 高级用法

在本节中，我们简要介绍 mpl 库中的一些高级特性，主要是用于参数绑定的 bind 表达式和 lambda 表达式。

bind 和 lambda 表达式都是函数式编程中的重要角色，它们强化了函数对象的作用，Boost 和 C++11/14 标准都提供了 bind 和 lambda 表达式功能，而在 mpl 中则实现了编译期的 bind 和 lambda 表达式，配合算法使用会令模板元编程更加强大（和复杂）。

13.7.1 高阶元数据

在研究编译期 bind/lambda 表达式之前，我们先来了解一个元编程新概念：高阶元数据。[①]

高阶元数据是一种特殊的元数据，是一个 struct 或 class，并且内嵌了一个名为 apply 的元函数，形如：[②]

```
struct high_order_meta_data                        //高阶元数据
{
  template<typename T1, typename T2, ...>
  struct apply                                     //内嵌名为 apply 的元函数
  {
     typedef some_define type;
  };
};
```

高阶元数据主要的功能是包装元函数，把它变成元数据，从而可以把元函数传递给其他的元函数进行调用。从功能来看，它的作用颇类似函数对象，元函数 apply 可以看作是运行时的 operator()。

与高阶元数据对应，操作或者返回高阶元数据的元函数就被称为高阶元函数，是一种更为复杂和强大的元函数。

头文件<boost/mpl/apply_wrap.hpp>中定义了一系列的 apply_wrap*N* 高阶元函数，它可以调用高阶元数据里的 apply 元函数，其声明如下：

```
template<typename F, typename arg1, typename arg2,... >
struct apply_wrapN
{
   typedef some_define type;
};
```

apply_wrap 相当于在高级元函数调用时多了一个间接层，apply_wrap*N* <F,a1,a2,...>与 F::apply<a1,a2,...>等价。

---

① 高阶元数据这个名词是作者在实践中自行“发明”的，目前现有的元编程资料并没有使用这个词来称呼这种元编程构件，在 Boost 文档中使用的名称是“元函数类”。但作者个人认为这个名词欠妥，因为在模板元编程中已经不存在 C++的 class 概念了，所有的类型（type）都作为元数据出现，使用“类”这个称呼容易造成概念上的混乱。

② 高阶元数据内部的 apply 元函数命名只是 mpl 的约定，我们也可以自行选择其他的名字，例如 9.2.6 节就使用了 pack。

我们也可以使用另一个高阶元函数 mpl::apply 来达到同样的效果，写法更加简单，它没有 apply_wrap 的数字后缀，缺省最多支持五个参数，但可以通过配置宏 BOOST_MPL_LIMIT_METAFUNCTION_ARITY 改变。

13.7.2 占位符

mpl 库在头文件<boost/mpl/placeholders.hpp>中定义了若干编译期占位符，它们都是高阶元数据，被用于构造 lambda 表达式，其定义与 bind 中的占位符很相似：

```
typedef some_define  _;                               //匿名占位符
typedef arg<1>       _1;                              //占位符 1
typedef arg<2>       _2;                              //占位符 2
...
typedef arg<n>       _n;                              //占位符 n
```

这些编译期占位符实际上是参数元函数 arg<N>的别名，与运行时占位符的作用相似，可以选择参数列表中的第 n 个参数（默认最大为 5）。比较特殊的是匿名占位符“_”，它可以根据在表达式中的位置自动变为带数字的占位符，例如 bind<_,_>相当于 bind<_1,_2>。

示范占位符用法的代码如下：

```
typedef apply_wrap2<_1, int, char>::type t1;          //获得第一个参数
typedef apply<_3, int, char, float>::type t2;         //获得第三个参数

assert((is_same<t1, int>::value));                    //验证占位符的效果
assert((is_same<t2, float>::value));
```

13.7.3 bind 表达式

bind 表达式是一个高阶元函数，类似 boost.bind，配合占位符实现参数的传递，可以在编译期绑定高阶元数据生成一个新的高阶元数据，就像 boost.bind 绑定函数对象那样。

bind 位于头文件<boost/mpl/bind.hpp>，其声明如下：

```
template< typename F, typename a1, typename a2,... >
struct bind
{...};
```

示范 bind 用法的代码如下：

```
struct func1                                //一个简单的高阶元数据，类似函数对象
```

```
{
  template<typename T1>
  struct apply                                  //apply 元函数
  {    typedef T1 type;};                       //直接返回参数
};

typedef bind<func1, _1> f1;        //用占位符绑定 func1，也可写作 bind<func1, _>

typedef apply<f1, int>::type data1;                 //使用 apply 调用
assert((is_same<data1, int>::value));

typedef apply<bind<func1, float>>::type data2;    //直接绑定参数调用
assert((is_same<data2, float>::value));
```

13.7.4 lambda 表达式

在 mpl 里，所谓“lambda 表达式”泛指含有占位符的表达式，例如 plus<_1,int_<10>>、add_const<_>。

注意：这些元函数在使用了占位符后就变成了普通的无参元函数（参数是占位符），无法完成元计算。它们也不是高阶元数据，也不能使用 apply_wrap 元函数调用。例如，下面的代码无法通过编译：

```
typedef apply_wrap2<plus<_,_>,                      //调用 apply_wrap
     int_<1>, int_<2>>::type data0;                 //编译失败
```

mpl 库在头文件<boost/mpl/lambda.hpp>中提供了一个专门的元函数 lambda，它可以把一个占位符表达式包装成一个匿名高阶元数据，之后就可以被高阶元函数调用：

```
typedef apply_wrap2<lambda<plus<_,_>>::type,       //使用 lambda 包装
                    int_<1>, int_<2>>::type data0;
assert((data0::value == 3));
```

高阶元函数 apply 比 apply_wrap 更强大，它可以直接调用占位符表达式，这是因为它内部已经使用了 lambda 来包装占位符表达式：

```
typedef apply<plus<_,_>,                            //注意，不需要 lambda 包装
  int_<1>, int_<2>>::type data1;
assert((data1::value == 3));
```

lambda 表达式也可以应用于 bind，例如：

```
typedef bind<lambda<plus<_,_>>::type, _1,_2> f1;
typedef apply<f1, int_<1>, int_<2>>::type data2;
assert((data2::value == 3));
```

但这通常没有必要，因为 apply 可以直接使用占位符表达式，比 bind 更加灵活方便。

13.7.5 算法的高级应用

有了高阶元数据、bind 和 lambda 表达式，算法的功能就更加强大了，它可以像运行时标准算法一样，传入一个高阶元数据执行复杂的操作。

本节简要介绍这些使用 lambda 表达式算法的用法，不做深入的分析。

插入器

通用的插入器 inserter<S,Op>中的参数 Op 是一个 lambda 表达式，以一个初始状态 S 开始连续执行 Op 操作完成插入动作，故 back_inserter<C>相当于 inserter<C, push_back<_1, _2>>，front_inserter<C>相当于 inserter<s, push_front<_1, _2> >。

使用 inserter 我们也可以做出与“插入”完全无关的操作，例如下面的代码在编译期执行了累加计算：

```
typedef range_c<int, 1, 11> rc;                //从 1 到 10 的整数区间
typedef mpl::copy<rc,                          //拷贝整数到插入器
  mpl::inserter<int_<0>, plus<_,_> >           //初值为 0，使用加法元函数
  >::type sum;
cout << sum::value;                            //输出元计算结果 55
```

查询算法

使用 lambda 表达式的查询算法如下，它们与标准算法很类似。

- find_if<C,P> ：查找容器中满足谓词 P 的元素。
- count_if<C,P> ：计算容器中满足谓词 P 的元素的个数。
- min_element<C,P> ：返回容器中的第一个最小元素的位置。
- max_element<C,P> ：返回容器中的第一个最大元素的位置。
- lower_bound<C,t,P> ：返回已经排序的容器中第一个可插入 t 的位置。

■ upper_bound<C,t,P> ：返回已经排序的容器中最后一个可插入 t 的位置。

在使用这些查询算法时我们必须编写谓词高阶元数据定义元素的关系，例如，我们可以以类型的 sizeof 大小判定它们的顺序，其示范代码如下（注意同样是运行在编译期）：

```
typedef mpl::vector<float, double,
  char, int, long> vec;                          //容纳各种数值类型的容器

assert((mpl::count_if<vec,                       //count_if 算法
      is_float<_> >::value==2));                 //使用 is_float 计算其中的浮点数
assert((is_same<
      deref<mpl::find_if<vec,                    //find_if 查找 char 类型
      is_same<_, char>>::type>::type,
      char>::value));

typedef lambda<                        //使用 sizeof_定义 lambda 表达式谓词
         less<sizeof_<_1>, sizeof_<_2>> > Comp;

typedef deref<min_element<vec, Comp>::type>::type mint; //最小元素
assert((is_same<mint, char>::value));

typedef deref<max_element<vec, Comp>::type>::type maxt; //最大元素
assert((is_same<maxt, double>::value));
```

变换算法

使用 lambda 表达式的部分变换算法如下，它们与标准算法同样很类似。

■ copy_if<C,P> ：复制容器中满足谓词 P 的元素。

■ replace_if<C,P,New> ：替换容器中满足谓词 P 的元素。

■ remove_if<C,P> ：删除容器中满足谓词 P 的元素。

■ unique<C,P> ：删除容器中连续的重复元素，是否连续由 P 确定。

■ sort<C,P> ：对容器以谓词 P 作为排序准则排序。

■ transform<C,Op> ：对容器内所有元素调用 Op 操作。

示范这些变换算法的代码如下：

```
#include <boost/mpl/vector_c.hpp>
```

```
typedef vector_c<int, 5,3,7,2,6,4,2> vec;          //一个整数容器

typedef copy_if<vec,                                //copy_if<>算法
  equal_to<modulus<_1, int_<2> > ,                  //只复制偶数
        int_<0>> >::type vec2;                      //注意元函数的组合使用

typedef replace_if<vec2,                            //替换整数 6
  equal_to<_1, int_<6> >, int_<10>>::type vec3;

typedef remove_if<vec3,                             //删除大于 5 的整数
  greater<_1, int_<5>>>::type vec4;

typedef sort<vec4, less<_,_>>::type vec5;           //降序排序

typedef unique<vec5, equal_to<_,_>>::type vec6;     //删除重复元素
```

这段代码演示了大部分算法的用法。

13.8 断言

模板元编程的调试是一项艰巨的工作，因为现有的大多数编译器对运行时调试均支持得很好，但并没有对元编程提供特别的支持，我们无法像运行普通程序时一样设置断点、逐步跟踪并查看元数据的值，通常只能使用 typeid(T).name()来输出元计算的中间检查结果[①]，但这不总是可用的。

保证元程序正确性最基本的工具就是断言，C++11/14 和 boost.static_assert 库提供了静态断言的功能，可以用在函数域、类域或名字空间域等程序的任何地方，相当于运行时的 assert 宏，例如：

```
typedef mpl::vector<int, char> vec;                 //一个整数类型的容器
BOOST_STATIC_ASSERT((is_same<int,                   //静态断言
  front<vec>::type>::value));                       //断言前端元素是 int
```

静态断言 static_assert 比运行时断言 assert 好的地方是它在编译期执行，可以更早地检查出元程序的错误，而不必等到编译完成再运行。

① 使用 boost.type_index 库可以输出更可读的类型名。

但静态断言不是专门为元编程设计的，它虽然可以很好地保证元程序的正确性，但不能够给出有利于元编程调试的更多可用信息，而且写法也较麻烦。为此 mpl 库在头文件<boost/mpl/assert.hpp>中特意定义了几个元编程断言，它们能够在元程序发生错误时提供有用的信息，虽然仍不够完备，但已经可以部分减少我们的调试工作量。

13.8.1 基本断言

宏 BOOST_MPL_ASSERT 是 mpl 中最常用的一个静态断言，它使用一个返回 bool_ 的值元函数 pred 作为参数，断定 pred::value 为真，其调用形式是：

```
BOOST_MPL_ASSERT(( pred ));
```

注意宏的调用方式，必须使用两对圆括号，即使 pred 的模板参数列表中没有逗号。

BOOST_MPL_ASSERT 的用法与 BOOST_STATIC_ASSERT 类似，只是它不接受普通的条件表达式，只能返回 bool_ 值元函数：

```
typedef mpl::vector<int, char> vec;                //一个整数类型的容器

BOOST_MPL_ASSERT((is_same<int,                     //静态断言，两对圆括号
  front<vec>::type> ));                            //注意不需要使用::value

BOOST_MPL_ASSERT((equal_to<                        //验证容器的大小
  size<vec>::type, int_<3>>));                     //发生一个断言错误
```

第二个断言会在编译时报出一个形如“************pred::************”的错误，同时指出出错的行号，例如：

```
... ************boost::mpl::equal_to<N1,N2>::* ************ ...
```

13.8.2 否定断言

因为 BOOST_MPL_ASSERT 的测试条件是元函数，不能使用逻辑运算符，所以如果要验证否定条件就需要使用 not_ 元函数进行包装。为了简化否定条件的测试，mpl 定义了断言 BOOST_MPL_ASSERT_NOT。

BOOST_MPL_ASSERT_NOT 的用法与 BOOST_MPL_ASSERT 相同，也要求必须使用两对圆括号，只是判断的条件相反：

```
template<typename T>
struct my_operation                                //定义一个简单的元函数
```

```
{
  BOOST_MPL_ASSERT_NOT((is_integral <T>));     //要求不是整数
  BOOST_MPL_ASSERT((                           //要求有内部类型 value_type
      is_integral<typename T::value_type>)); //并且是整型

  typedef typename next<T>::type type;         //递增 mpl 整数类型
  };

typedef my_operation<int_<3>>::type t;         //正确
typedef my_operation<int>::type error;         //编译错误
```

13.8.3 关系断言

仅有判断真或假的断言还是不够的，我们还需要判断逻辑关系的断言。同样地，为了解决使用元函数带来的麻烦，mpl 提供了简化关系判断的宏，其声明如下：

```
BOOST_MPL_ASSERT_RELATION(x, rel, y)
```

宏 BOOST_MPL_ASSERT_RELATION 的形式比较“怪异”，它有三个参数，x/y 是两个编译期整数（非包装器），rel 是一个合法的 C++关系操作符，如==、<。

BOOST_MPL_ASSERT_RELATION 不需要使用两对圆括号（使用了反而是错误的），其示范代码如下：

```
BOOST_MPL_ASSERT_RELATION(int_<5>::value, >, 0);              //正确
BOOST_MPL_ASSERT_RELATION(sizeof(int), >, sizeof(long));      //编译错误
```

编译的错误信息如下：

```
... ************assert_relation<...>::************ ...
```

13.8.4 定制消息的断言

运行时断言宏 assert 可以在断言的同时用&&定制错误消息，例如：

```
assert(1>2 && "error message");
```

这种定制消息的功能对于程序的调试显然是很有用的，所以 mpl 也提供了一个功能类似的宏，其声明如下：

```
BOOST_MPL_ASSERT_MSG( c, msg, types_ )
```

宏的三个参数的含义如下所述。

- c : 条件表达式，非元函数。
- msg : 自定义的消息，但它不是一个 c 字符串，而是一个符合 C++语法的标志符，被宏用来生成一个仅用于编译报错的不完整类（struct）。
- types_ : 类型列表，用于定制输出断言失败时的类型信息，它可以是一个被圆括号包围的若干类型（可以为空），也可以是一个 types<...>结构。

示范 BOOST_MPL_ASSERT_MSG 用法的代码如下：

```
template<typename T>
struct my_operation                                 //对之前的元函数略做修改
{
  BOOST_MPL_ASSERT_MSG(!is_pod<T>::value,           //要求是非 pod 类型
      IS_POD_ERROR, (T));                           //定制错误消息和类型信息
};

int main()
{
  BOOST_MPL_ASSERT_MSG(1>2, DEMO_MESSAGE, ());      //简单的定制错误消息
  my_operation<int>::type;                          //编译错误
}
```

编译后输出的错误消息类型的形式如下：

```
************ DEMO_MESSAGE::************
************ my_operation<T>::IS_POD_ERROR::************
```

13.9 实例研究

在本节中，我们将实际使用模板元编程技术和 mpl 实现一个动态加载动态链接库函数的功能。

大多数读者应该都对动态加载函数比较熟悉，基本的工作原理很简单，使用函数 dlsym() 获取函数指针然后再强制转型就可以了，我们来看在这里 mpl 能够发挥什么作用。

13.9.1 泛型编程版本

首先，我们来看最简单的泛型版本，它很简单地封装了 UNIX 的底层 API：

```
#include <dlfcn.h>                                  //UNIX 动态链接头文件
```

```
class DlManager                                    //定义一个加载 so 函数的包装类
{
public:
  DlManager(const char* name)                      //构造函数
  {
      m_h = dlopen(name, RTLD_NOW);                //使用文件名加载 so
  }
  ~DlManager()                                     //析构函数
  {
      dlclose(m_h);                                //释放 so
  }

  template<typename FuncType>                      //模板参数是函数指针类型
  FuncType load(const char* func_name)             //模板函数加载 so 函数
  {
      FuncType pf = reinterpret_cast<FuncType>     //强制类型转换
          (dlsym(m_h, func_name));                 //获得函数指针

      if (!pf)                                     //检查指针的正确性
      {   throw std::runtime_error(dlerror());}    //失败抛出异常

      return pf;
  }
private:
  typedef void* handle_t;                          //句柄类型定义
  handle_t m_h;                                    //句柄
};
```

类 DlManager 在构造时使用传入的文件名加载 so，在析构时释放 so。它的核心功能是模板函数 load()，模板参数是函数指针类型，调用 dlsym() 获取 so 中导出函数的地址，然后使用转型操作符 reinterpret_cast 将其转换为正确的函数指针类型。由于可能出现加载函数失败的情况，因此通常需要检查函数指针是否为空指针（为了代码清晰，接下来的代码都将忽略检查，请读者注意）。

假设我们在 libtest.so 中有如下两个导出函数：

```
extern "C"                                    //导出为 extern "C"
{
  int so_func1(int x)                         //测试用导出函数 1
```

```
  {    return x * x;}
  int so_func2(int x, int y)                    //测试用导出函数 2
  {    return x + y;}
}
```

那么 DlManager 可以这样使用:

```
typedef int (*Func1)(int);                      //首先定义函数指针
typedef int (*Func2)(int, int);

int main()
{
  DlManager dm("libtest.so");                   //加载 so

  Func1 f1 = dm.load<Func1>("so_func1");        //定义函数指针并加载
  Func2 f2 = dm.load<Func2>("so_func2");        //定义函数指针并加载

  cout << f1(10) << endl;                       //调用加载的函数指针
  cout << f2(10, 20) << endl;
}
```

调用导出函数的代码也可以不使用函数指针变量暂存而直接调用:

```
cout << dm.load<Func1>("so_func1")(10) << endl;
cout << dm.load<Func2>("so_func2")(10, 20) << endl;
```

DlManager 使用了泛型编程，在一定程度上封装了系统底层接口，用起来也较原始方法要方便一些，但它仍然是运行时的，而且函数指针定义和函数名称定义之间缺乏紧固的联系，容易产生编写错误导致误用。

13.9.2 元编程第 1 版

本小节我们将使用 mpl 来改进 DlManager。

仔细分析加载动态库这个例子，我们可以把它分解为两部分：一部分是编译期的数据，包括文件名、函数名以及函数指针类型，另一部分是运行时的代码，包括加载 so 文件和获取 so 函数。泛型版本没有很好地把两者解耦，而是将两者简单地在运行时混合在了一起，所以不够清晰。

因为我们现在拥有 mpl 这个强大的元编程工具，所以能够把编译期的数据分离出来。为了明确区分编译和运行时的数据，我们把程序实现分为前端和后端两个部分：前端使用 mpl 定义编译

期数据，后端使用前端数据实现运行时的功能。

前端

前端的数据有 so 文件名、接口函数名和函数指针类型，前两者可以使用 mpl::string 表述，后者本身就是元数据，为了实现函数名与函数指针类型的对应关系我们还应该使用 mpl::map，最后要用一个 struct 把这些数据封装成一个前端类。

前端类 dl_front 的实现代码如下：

```
#include <boost/mpl/vector.hpp>                              //元编程各种工具
#include <boost/mpl/string.hpp>
#include <boost/mpl/map.hpp>
#include <boost/mpl/at.hpp>
namespace mpl = boost::mpl;

struct dl_front                                              //前端类定义
{
  typedef mpl::string<'lib','test','.so'> so_name;   //so 文件名

  typedef int (*Func1)(int);                                 //函数指针类型定义
  typedef int (*Func2)(int, int);

  typedef mpl::string<'so_','func','1'> fun1_name;   //函数名定义
  typedef mpl::string<'so_','func','2'> fun2_name;

  typedef mpl::map<                                          //函数名到函数指针类型的映射
      mpl::pair<fun1_name, Func1>,                           //使用 mpl::pair
      mpl::pair<fun2_name, Func2>
      >   map_fun;
};                                                           //前端类定义结束
```

dl_front 也可以看作一个无参的非标准元函数，它可以返回与 so 相关的多个数据，其中最重要的是 map_fun，它把函数名和函数指针类型紧密地联系在了一起。

后端

有了前端定义，后端 dl_back 的实现就容易多了，基本代码与 13.9.1 节的 DlManager 很相似，只是操作的数据都变成了 dl_front 返回的元数据。

对于 dl_back，我们将其实现为一个模板类，这样它就可以使用不同的前端类支持不同的 so 加载功能（静态多态），而本身的代码保持稳定，构造函数和析构函数的实现如下：

```
template<typename Front>                                    //模板参数是前端类
class dl_back
{
private:
  void* m_h = nullptr;                                      //so 句柄
public:
  dl_back()                                                 //构造函数
  {
      m_h = dlopen(                       //使用 c_str 元函数获取 so 文件名
          mpl::c_str<typename Front::so_name>::value, RTLD_NOW);
  }
  ~dl_back()                                                //析构函数
  {
      dlclose(m_h);
  }
};
```

dl_back 的核心功能是成员模板函数 func_ptr()，它使用函数名作为索引，在前端的 map 中使用 at 元函数查找对应的函数指针类型：

```
template<typename FuncName>
typename mpl::at<typename Front::map_fun, FuncName>::type    //返回类型
func_ptr()
{
  typedef typename                                  //typedef 以简化代码
      mpl::at<typename Front::map_fun, FuncName>::type result_type;

  result_type pf = reinterpret_cast<result_type>          //函数指针转型
      (dlsym(m_h, mpl::c_str<FuncName>::value));

  return pf;
}
```

验证

前端和后端都已经实现，现在我们只需要传递一个在前端定义好的函数名元数据，后端就会

使用模板元编程技术自动推导出对应的函数类型完成函数的加载，代码非常简洁：

```
dl_back<dl_front> dl;                                    //使用前后端定义 so 加载功能

cout << dl.func_ptr<dl_front::fun1_name>()(10) << endl; //调用 so 函数
cout << dl.func_ptr<dl_front::fun2_name>()(10,20) << endl;
```

注意，在调用成员函数 func_ptr()时我们除了要显式写出前端定义的函数名外，还必须调用两次 operator()，因为第一次调用 operator()只是获取了函数指针，第二次调用 operator()才是真正的 so 接口函数调用。

13.9.3 元编程第 2 版

元编程的第 1 个版本应该说是比较成功的，它使用元编程技术分离了编译和运行两个时期的数据和代码，把 so 相关的编译期数据都封装在了前端，而后端则使用了静态多态技术，任何符合前端定义的类都可以应用于后端，大大地增强了后端的稳定性和灵活性。

但第 1 版还有改进的余地。一方面我们可以在后端使用静态断言和元编程断言，约束前端的定义，从而避免前端的代码错误，另一方面我们可以实现一个增强的成员函数 func()，直接传递参数减少一次 operator()的调用。断言的工作比较简单，读者可自行尝试完成，本小节实现第二个改进。

要减少一次 operator()的调用，这就要求函数 func()返回的是 so 函数指针的返回类型而不是函数指针类型，同时传递相应数量的参数。前者可以使用 result_of 结合 mpl 来自动推导函数的返回值类型[①]，后者可以使用可变模板参数列表来解决。

改进后的 func()代码如下：

```
template<typename FuncName, typename ... Args>          //可变参数模板
typename boost::result_of<                        //使用 result_of 推导函数的返回值
  typename mpl::at<typename Front::map_fun, FuncName>::type(Args...)
  >::type
func (Args const& ... args)                             //可变参数模板
{
  typedef typename mpl::at<typename Front::map_fun, FuncName>::type
      func_type;                                  //由 mpl::map 得到函数指针类型
```

① 如果编译器支持 C++14，那么也可以直接使用 auto，让编译器自动推导。

```
  func_type pf = reinterpret_cast<func_type>          //获取函数指针
      (dlsym(m_h, mpl::c_str<FuncName>::value));

  return pf(args...);                                  //直接调用函数指针
}
```

我们也可以直接使用第 1 版已经写好的 func_ptr()函数直接返回函数指针来调用，这样可以在很大程度上简化函数体内部的代码：

```
typename boost::function_traits<              //改用 function_traits 推导类型
    typename boost::remove_pointer<           //需要先移除类型里的指针
        typename mpl::at<typename Front::map_fun, FuncName>::type>::type
    >::result_type                            //获取函数的返回类型
func(Args const& ... args)
{
  return func_ptr<FuncName>()(args...);  //直接调用获取函数指针的成员函数
}
```

改进后的 func()可以这样调用，写法更加简单：

```
cout << dl.func<dl_front::fun1_name>(10)<< endl;          //只有一对括号
cout << dl.func<dl_front::fun2_name>(10,20) << endl;
```

这样，通过模板元编程，我们在编译期对函数的返回类型执行元计算，让编译器自动生成多个同名不同调用形式的函数，实现了普通编程无法完成的功能。

13.10 总结

在本章中，我们讨论了模板元编程库 mpl，它是进行模板元编程的主要工具。因为 mpl 内容庞杂，本章也只能阐述其中的部分内容。

mpl 基于现有的 C++标准创建了一套完整的元编程体系框架，使我们无须从最基本的元编程概念开始，可直接使用已经定义好的若干高级工具，极大地简化了元编程的开发工作。mpl 的体系结构完全是仿造 C++标准库的，许多概念都非常相似，所以只要理解了它在编译期运行的原理就能够较容易地掌握 mpl 的用法。

同 STL 一样，mpl 也由三大组成部分，分别是容器、迭代器和算法，此外还有类似函数对象的高阶元数据以及 bind 和 lambda 表达式。这些高级工具是 mpl 的核心，它们的实现很复杂，但用法却比较简单。

虽然 mpl 的高级元编程工具简化了开发工作，但现有的 C++编译器仍然缺少调试元程序的功能，这给开发元程序带来了不小的麻烦。mpl 在 static_assert 静态断言外又提供了专用的元编程断言，这在一定程度上增强了元编程的正确性，提高了元编程的效率。

模板元编程目前仍然算是一个较新的编程范式，但已经得到了广泛的应用，尤其是在 Boost 库中，如 proto（领域嵌入式语言框架）、xpressive（静态正则表达式）、msm（元编程状态机）、fusion（混合了编译和运行时的代码）等都使用了大量的元编程技术和 mpl 组件，了解 mpl 对研究这些库会非常有帮助。而且，相信随着 C++的进一步发展，模板元编程最终将进入普通程序员的视野，本章的最后展示了一个实际使用 mpl 的小例子，有助于读者理解元编程的实际应用。

第14章

预处理元编程

预处理（preprocessing）是从C/C++源码生成最终二进制代码过程中的一个重要步骤，但它并不识别C/C++语法，只是依据简单的规则做文本替换，把文本形式的源码转换为另外一种更适合编译器处理的形式。因为它发生在编译器处理源码之前，所以被称为“预处理”。

虽然预处理是与C/C++同时诞生的，但多年来几乎没有随着语言的进化而改变，基本保持着三十多年前的原貌，而且由于替换规则过于简单随意而招致了许多“恶名”，“慎用少用宏”差不多是每一个C/C++程序员所恪守的编程准则。

在这种简陋且恶劣的环境下，boost.preprocessor库却成功创立了一个比较完整的预处理元编程体系，可以在预处理阶段计算整数，执行函数，甚至还有数组、链表等高级数据结构，从而能够完成一些复杂的任务。

在本章之后的叙述中，我们使用“预处理”表示C/C++标准定义的预处理功能，而使用“preprocessor库”表示boost.preprocessor库提供的预处理功能。

preprocessor库基于预处理程序，在预处理阶段工作，独立于C/C++编译器，不与C/C++语言或者Boost程序库发生任何关系，所以完全可以独立使用。

preprocessor库完全由头文件组成，可以直接包含头文件，即：[①]

```
#include <boost/preprocessor.hpp>
```

14.1 概述

对于预处理，我们最熟悉的可能就是#include和#define了，但预处理并不仅限于这两

① 它转而包含<boost/preprocessor/library.hpp>，是预处理库的真正实现。

个指令，本小节我们将简要论述预处理里的一些基本概念和规则。

使用选项“-P -E”可以让 GCC 编译器只执行预处理动作，也就是运行预处理元程序，例如：

```
gcc -P -E -o a.out basic.cpp;        #预处理 basic.cpp，输出为 a.out
```

C++11/14 标准在第 16 章完整定义了预处理，读者可以进一步参考。

14.1.1 元数据

与模板元编程一样，在预处理元编程领域里也有元数据的概念，同样是不可变的，但它的元数据要比模板元编程要简单很多，因为这里并没有 C/C++里的“类型”。

预处理元编程可以识别两种基本数据。

- 整数 ：可以有正负号和 0x/L/LL 等修饰。
- 字符串：使用引号（''和""）包围的字符序列，注意与 C/C++不同，单引号在预处理程序里定义的也是字符串。

此外，另一大类元数据就是“标记”（token，也可以称之为记号、符号），它相当于 C/C++程序或者 Shell 脚本里的变量，预处理元编程主要就是对各种标记进行运算，最终将标记展开为某种形式的文本，作为 C/C++源码的组成部分。

虽然预处理程序只能识别整数和字符串，但对标记的内容却没有限制，可以是任意的字符序列文本。

指令#define 可以定义标记，例如：

```
#define t1 3.14                    //定义标记 t1，值为 3.14，非整数非字符串
#define t2 ~!@#$%^&*()-+           //定义标记 t2，值为一个字符序列
#define t3 'snake'                 //定义标记 t3，值为字符串
#define t4 "quiet"                 //定义标记 t4，值为字符串
```

14.1.2 基本语法

预处理指令都以符号“#”开始，在预处理元编程中较常用的有以下几个。

- \#　　　　　　：空预处理指令，相当于空行。
- #include　　　：引入一个文件（不一定是头文件，任何文本文件都允许）。
- #define　　　 ：定义一个标记，用来声明元数据和元函数。

- #undef ：删除一个标记。
- #if/#else/#endif ：分支语句，只能识别整数表达式和 defined 表达式。[①]
- #ifdef/#ifndef ：检查标记是否已经被定义，相当于#if defined()。
- #error ：产生一条错误信息，停止预处理。

#define 是预处理元编程里最重要的指令，它通过定义标记的方式来声明元数据和元函数（也就是宏），常用的形式是：

```
#define  token            expression          //定义元数据
#define  function(...)    function_body       //定义元函数
```

因为预处理不使用分号表示语句的结束，理论上一行就是一条语句，所以当代码较多时需要在行末及时使用续行符“\”，最好再对齐格式，这一点请读者务必牢记。

C++11/14 标准为宏增加了可变参数的特性，类似于可变参数模板，也使用“...”来声明可变参数，之后可以用宏“__VA_ARGS__”来展开参数列表。

#undef 是另一个重要的指令，因为在预处理元编程领域里没有作用域的概念，所有标记都是全局可见的，为了避免名字冲突，在用完一个标记后需要及时将其 undef。

下面是使用#define 声明元数据和元函数的一些例子：

```
#define data1   vector                         //定义元数据 data1=vector
#define data2   boost::factory                 //定义元数据 data2=boost::factory

#define func(x,y)    int x = y                 //定义元函数
#define vfunc(x,...) x{ __VA_ARGS__}           //定义可变参数元函数

//调用元函数，展开为 int vector = boost::factory
func(data1, data2)

//调用元函数，展开为 vector v{ 1,2,3}
vfunc(vector v, 1,2,3)

#undef data1                                   //删除元数据 data1
#undef data2                                   //删除元数据 data2
```

preprocessor 库里提供了大量的预处理元编程工具，大多数以“BOOST_PP_”作为前缀，

① C++17 草案新增了__has_include 表达式，可以检查一个文件是否存在。

我们之后会看到很多这样命名的元数据和元函数。

14.1.3 特殊符号

虽然在预处理元编程里标记可以使用任意字符，但有三个字符对于元函数有特殊含义，它们就是逗号“,”和左右圆括号“(”“)”，逗号被用于分隔函数参数，而圆括号则标记了函数参数列表。[①]

为了能够在预处理元编程里使用这三个特殊字符，preprocessor 库定义了三个专用的元函数来表示它们：

```
# define BOOST_PP_COMMA()  ,                //间接表示逗号
# define BOOST_PP_LPAREN() (                //间接表示左括号
# define BOOST_PP_RPAREN() )                //间接表示右括号
```

通过这种元函数的包装，我们就可以把这三个符号作为参数传递给其他元函数进行处理。

preprocessor 库还定义了一个元函数 BOOST_PP_EMPTY，表示无定义的空标记：

```
# define BOOST_PP_EMPTY()                   //空元函数，无标记
```

14.1.4 特殊操作符

在元函数里，符号“#”还有别的含义，可以操作标记。

- #x　：字符串化，把标记转换为 C/C++里的字符串类型，即"x"。
- x##y：标记粘贴（拼接），把两个标记“粘贴”为一个新标记，即 xy。

例如：

```
#define op1(x)        #x                //元函数，字符串化参数 x
#define op2(x,y,z)    x##y##z           //元函数，拼接三个参数

op1(data1)                              //展开为"data1"
op2(data1, data2, z)                    //展开为 data1data2z
```

“#”和“##”操作符只提供基本的字符串化和标记粘贴功能，不能处理元数据，所以 preprocessor 库提供了两个元函数：BOOST_PP_STRINGIZE(text)和 BOOST_PP_CAT

① C/C++里的“~”“{}”“[]”“<>”等符号在预处理元编程里被视为普通的标记，没有特别的对待。

(x,y)，它们封装了“#”和“##”，可以正确处理元数据。

把刚才的代码改写如下：

```
#define op1(x)  BOOST_PP_STRINGIZE(x)       //字符串化参数x，可处理任意元数据
#define op2(x,y,z)  BOOST_PP_CAT(x, \       //拼接三个参数，注意使用了续行符
                     BOOST_PP_CAT(y,z))     //需嵌套调用
```

再调用元函数会得到不同的结果：

```
op1(data1)                                  //展开为"vector"
op2(data1, data2, z)                        //展开为vectorboost::factoryz
```

在实际的预处理元编程时我们应当尽量使用这两个元函数，避免直接使用“#”和“##”操作符。

14.2 整数运算

虽然预处理程序能够识别整数，但仅能够在#if/#endif里对整数进行运算，不能用于元函数，在元函数里的表达式会被认为是一个普通的标记，例如：

```
#define calc(x, y) int x = y                //定义元函数
calc(x, 1+2)                                //调用元函数，展开为int x = 1+2
```

preprocessor库为此实现了一整套整数运算的元函数，包括以下几类。

- ADD/SUB/MUL/DIV/MOD ：加减乘除和取余。
- INC/DEC ：加减1运算。
- MIN/MAX ：取较小值或较大值。
- AND/OR/NOT/XOR ：逻辑运算，与/或/非/异或。
- BITAND/BITOR/BITXOR ：位运算。
- BOOL ：bool运算，转换为bool值0或1。
- EQUAL/LESS/GREATER ：比较运算。

不过这些元函数不能处理负数，而且正整数也不能够超过宏 BOOST_PP_LIMIT_MAG 的上

限，也就是 256：①

```
# define BOOST_PP_LIMIT_MAG 256              //当前预处理元编程能够处理的最大整数
```

下面的代码简单示范了部分整数运算元函数的用法：

```
calc(x, BOOST_PP_ADD(1,2))                   //展开为 int x = 3

#define x 1                                  //两个元数据
#define y 2

#define v BOOST_PP_ADD(x, y)                 //加法，v=3
#define u BOOST_PP_SUB(v, x)                 //减法，u=2
#define w BOOST_PP_INC(BOOST_PP_INC(u))      //连续加 1，w=4

#if BOOST_PP_BOOL(w)                         //bool 值检查
#define a BOOST_PP_MOD(10, 4)                //取余运算，a=2
#define b BOOST_PP_MUL(a, 300)               //乘法，超过整数上限，出错
#endif
```

14.3 常用元函数

preprocessor 库里有很多元函数，本小节简要介绍几个较为常用的。

14.3.1 ASSERT

元函数 BOOST_PP_ASSERT()提供基本的断言功能，类似于 C/C++里的 assert，形式是：

```
BOOST_PP_ASSERT(cond)                        //断言条件成立
```

其中的 cond 是判断条件，如果预处理计算结果是 0，那么断言就会失败，例如：

```
#if __cplusplus < 201103L                    //检查 C++标准
   BOOST_PP_ASSERT(0)                        //如果不是 C++11/14 则断言失败
#endif

#define x BOOST_PP_MUL(6, 6)                 //乘法元函数
```

① preprocessor 库在头文件<boost/preprocessor/config/limits.hpp>里还定义了其他的上限值，读者可阅读源码参考。

```
BOOST_PP_ASSERT(BOOST_PP_EQUAL(x, 36))          //检查运算结果
```

注意在这段代码里我们在检查__cplusplus 宏时并没有使用 BOOST_PP_ASSERT()或者其他元函数，这是因为__cplusplus 的值是 201103L，已经远大于 preprocessor 库的整数上限值 256。

14.3.2 IF

元函数 BOOST_PP_IF()实现了条件控制语句，类似于 C/C++里的 if，形式是：

```
BOOST_PP_IF(cond, t, f)                          //条件控制元函数
```

其中参数 cond 是判断条件，如果 cond 的计算结果为 0 或者未定义，那么结果为 f，否则为 t。

BOOST_PP_IF()是 preprocessor 库里的一个很重要的函数，很多其他元函数都要利用它来进行分支处理，例如决定是否产生逗号的 BOOST_PP_COMMA_IF：

```
#define BOOST_PP_COMMA_IF(cond)            \    //根据条件产生逗号
      BOOST_PP_IF(cond, BOOST_PP_COMMA,    \    //cond 为真调用 BOOST_PP_COMMA
                        BOOST_PP_EMPTY)()       //cond 为假调用 BOOST_PP_EMPTY
```

14.3.3 ENUM

元函数 BOOST_PP_ENUM()内部用到了 BOOST_PP_COMMA_IF，可以生成一个以逗号分隔的标记序列，形式是：

```
BOOST_PP_ENUM(count, helper, data)              //连续调用多次 helper
```

它有三个参数，含义如下。

- count ：调用 helper 的次数，必须是整数，但不能超过 256。
- helper：形如 helper(z,n,data)的元函数，被 BOOST_PP_ENUM 调用。
- data ：任意元数据，会作为参数传递给 helper 元函数。

元函数 BOOST_PP_ENUM()在执行后会展开为：

```
helper(z,0,data), helper(z,1,data),..., helper(z,count-1,data)
```

参数 helper 实际上是执行真正功能的函数，它的第一个参数 z 没有实际意义，被 BOOST_PP_ENUM()用于迭代，我们主要利用 n 和 data 这两个参数进行运算。

使用 BOOST_PP_ENUM()可以很容易地生成大量“机械”的代码，例如：

```
//定义辅助函数，拼接 d 和 n，即 d##n
#define helper(z, n, d)  BOOST_PP_CAT(d,n)

//定义元函数，通过 BOOST_PP_ENUM 多次调用 helper
#define DECL_VARS(n, var)     BOOST_PP_ENUM(n, helper, var)

//调用元函数，展开为“int x0 , x1 , x2 , x3 , x4 , x5 , x6 , x7 , x8 , x9;”
int DECL_VARS(10, x);
```

我们也可以忽略参数 n，仅使用参数 data，生成多个重复的标记：[①]

```
#define NGX_NULL_HELPER(z, n, d) d          //辅助函数

#define NGX_MODULE_NULL(n)      \           //元函数，调用 BOOST_PP_ENUM
    BOOST_PP_ENUM(n, NGX_NULL_HELPER, nullptr)

//调用元函数，展开为“{ nullptr , nullptr , nullptr} ”
{ NGX_MODULE_NULL(3)}
```

14.3.4 REPEAT

元函数 BOOST_PP_REPEAT()类似 BOOST_PP_ENUM()，同样可以生成重复的标记序列，但它的结果没有用逗号分隔，形式是：

```
BOOST_PP_REPEAT(count, helper, data)          //连续调用多次 helper
```

BOOST_PP_REPEAT()展开后与 BOOST_PP_ENUM()非常类似，唯一的区别是没有逗号：

```
helper(z,0,data) helper(z,1,data) ... helper(z,count-1,data)
```

因为这个特点，BOOST_PP_REPEAT()的用法更灵活，可以自由控制分隔符，用在需要生成多个 C/C++语句的场合，例如：

```
#define helper(z, n, d)  d##n{ n} ;              //辅助函数，产生变量声明语句
#define DECL_VARS(n, decl)    \                  //定义元函数
      BOOST_PP_REPEAT(n, helper, decl)           //调用 BOOST_PP_REPEAT
```

① 此例子来自作者的 ngx_cpp_dev 项目，参见 https://github.com/chronolaw/ngx_cpp_dev 或推荐书目[9] 。

```
//调用元函数，展开为“int x0{ 0} ; int x1{ 1} ; int x2{ 2} ;”
DECL_VARS(3, int x)
```

14.4 高级数据结构

除了元数据和元函数外，preprocessor 库还支持几种高级数据结构，包括 sequence、tuple、array 和 list，本节只介绍较常用的 sequence，其他的请读者参考 Boost 文档。

sequence 是一串以圆括号包围的标记，例如：

```
#define seq1 (a)(b)(c)(d)(e)                    //包含 5 个元素
#define seq2 (~123)([ ])(--{ }--)               //包含 3 个元素
```

sequence 类似于 C++里的 vector，是预处理元编程中最容易使用的一种数据结构，可以对它施加很多操作，例如取长度、拼接、遍历等，下面列出了几个常用的操作。

- SIZE(s) ：获取序列的长度，即元素数量。
- HEAD(s) ：获取序列的第一个元素。
- TAIL(s) ：获取序列的最后一个元素。
- ELEM(n,s) ：获取序列里的第 n 个元素。
- CAT(s) ：拼接整个序列。
- FOR_EACH(f,x,s)：遍历序列，对每个元素执行函数 f(r,x,e)，类似于 REPEAT。

示范这些元函数用法的代码如下：

```
BOOST_PP_SEQ_SIZE(seq1)                         //获取序列长度，结果为 5
BOOST_PP_SEQ_CAT(seq1)                          //拼接序列，结果是“abcde”

int BOOST_PP_SEQ_HEAD(seq1);                    //取首元素，展开为 int a;

#define helper(r, d, e) \                       //定义辅助函数，字符串化
            BOOST_PP_STRINGIZE(e)
//遍历序列，其中的~参数不使用，宏最后展开为"~123" "[ ]" "--{ }--"
BOOST_PP_SEQ_FOR_EACH(helper, ~, seq2)
```

C++17 草案为 namespace 增加了新语法，使用 namespace a::b::c 的方式声明嵌套的名字空间，我们也可以使用预处理元编程来模拟实现这个特性：

```
#define bns_helper(r, d, e) namespace e {          //定义名字空间起始的辅助函数
#define begin_namespace(s)    \                     //使用序列产生名字空间代码
    BOOST_PP_SEQ_FOR_EACH(bns_helper, ~, s)         //遍历序列，调用辅助函数

#define ens_helper(r, d, e) }                       //定义名字空间结束的辅助函数
#define end_namespace(s)    \                       //使用序列产生名字空间代码
    BOOST_PP_SEQ_FOR_EACH(ens_helper, ~, s)         //遍历序列，调用辅助函数

#define seq (boost) (detail) (test)                 //定义名字空间序列的元数据

//展开为namespace boost { namespace detail { namespace test { } } }
begin_namespace(seq)
end_namespace(seq)
```

14.5 总结

在本章中，我们讨论了预处理元编程，这已经是 C/C++语言所能达到的最上层元编程了，再向上就超出了 C/C++的范围。

预处理发生在编译之前，可以对源码做任意的变换处理，不受 C/C++语法的控制。预处理元编程的主要工具是宏（也就是#define），虽然 C++标准已经做了很多的努力来减少宏的使用（例如 nullptr、可变参数模板），但在未来可预见的一段时间里，预处理器还不会马上消失。

比起其他的脚本语言，预处理的文本操作功能是非常弱的，但因为它内建在 C/C++工具链里，随手可得，所以在简化代码或者生成重复代码的时候很有用，无须依赖外部的工具。

预处理元编程可以说是专门处理标记（token）的编程工具，preprocessor 库提供了大量实用的元函数，构建了较为完整的体系。预处理元编程的结构与模板元编程类似，也属于函数式编程，但毕竟它的语法规则比较简单，核心还是文本替换，所以没有模板元编程那么复杂，学习的难度不太高。

本书对预处理元编程中的实现细节并未深究，有兴趣的读者可以自行从简单的 INC/DEC/NOT 等源码看起。

第15章

现代 C++开发浅谈

本章是全书的最后一章，在这里我们将不再探究 C++和 Boost 程序库的细节和用法，而是讨论与它相关而又在它之外的东西——如何使用现代 C++编写清晰、优雅、高效、灵活和易维护的代码。

C++是一门伟大的语言，几乎能够胜任任何工作，但却对普通程序员不太友好，存在着太多的“未定义行为”的陷阱，如同某些小说中的奇门异宝，威力虽大但使用不当也容易反噬自身。高效地使用 C++的优点，同时避免 C++的缺点，是 C++程序员永恒的功课。

已经有许多专家学者论述了如何正确高效地使用 C++，其中的许多经典条款也被大众所熟知，如多态基类应使用虚析构函数、避免过长的函数、避免 new 动态分配内存等。相信读者也已经阅读过这些经典著作，笔者也不必就此多费口舌，仅把注意力集中在标准库和 Boost 的使用方面，限于个人开发经验不一定完全正确（而且有时候原则是可以违反的），论述也难免简陋，愿与读者共同探讨。

15.1 基本原则

熟悉并使用 STL/Boost 编程范式

自 C++发明至今已经三十多年了，从最早的简单面向对象逐渐发展成为包含泛型、函数式、模板元等许多范式的复杂混合体，其中的每一个编程范式都可以自成体系，在开发过程中打出一片天地。

二十年前，面向过程、基于对象是 C++编程的主流范式；十年前，主流范式变成了面向对象+设计模式；而现在，C++编程的主流范式则有“返璞归真”的趋势，过度使用虚函数的庞大类继承体系逐渐被摈弃，而使用泛型、函数式等新范式开发精致的小类并进行良好的组合成为了大方

向。[①]

STL 和 Boost 充分实践了现代 C++编程方法，不使用复杂的继承体系（少数例外，如 iostreams），特别是 Boost，它使用泛型编程、模板元编程和编译期的静态多态构建了功能完善的组件，代码简明精炼，是我们学习的极好范例。

STL/Boost 编程范式通常会要求我们编写带有模板参数的泛型代码，使用 typedef/using 进行类型计算，同时因为编译器模板实例化的原因，功能实现代码通常都写在 hpp 头文件里，最后由少量 cpp 完成功能的组装。

尽量理解 Boost 的工作原理

Boost 程序库庞大复杂，内部又十分精致细腻，由于提供了丰富的自说明文档，只要多花心思和时间，了解功能组件的接口和用法应该不是什么难事。

但在掌握了 Boost 组件的基本用法之后，我们应该再多用些时间去查看它们的实现代码：一是学习库作者的设计思想，学习这些顶级大师的经验；二是了解内部的实现机制和运行原理，从而能够洞悉其优缺点——很多细节在文档中并没有相应的描述。这样在使用时自然就能够避免一些性能不佳的用法，提高 Boost 组件的运行效率，达到“知其然更知其所以然”的境界。

boost.optional 是一个学习 Boost 库及模板元编程的好例子，它足够小却又并不简单，为了支持容纳 T 和 T&两种模板类型使用了一些 type_traits 和 mpl 里的元函数，并用一个 detail::reference_content 类型来保存值，这些巧妙的手法都值得我们认真揣摩。

学会使用对象包装 C++原始语言概念

基于操作符重载、面向对象和泛型的强大威力，在 STL 和 Boost 中许多“原始”的 C++操作概念都有了对象版本，是更“智能”（smart）的操作。这些“智能操作”的形式和用法都与原始操作非常类似，很容易学习和使用，并且它们还具有许多原始操作所不具备的“智能”特性。

使用这些智能操作可以更好地编写出健壮稳固的代码，也更利于代码的长期维护，所以我们应当尽量避免使用原始的 C++语言要素，而是使用它们对应的“智能”对象版本。

- 智能创建 ：factory，可取代操作符 new。
- 智能删除 ：checked_delete，可取代操作符 delete。
- 智能引用 ：reference_wrapper，可取代原始引用。

① 这并不是说我们要完全不使用面向过程和面向对象，而是说应该逐渐少使用它们，逐渐转向泛型等新范式。

- 智能指针 ：unique_ptr/shared_ptr，可取代原始指针。
- 智能函数指针：function，可取代原始函数指针。
- 智能函数 ：各种函数对象以及 bind 和 lambda 表达式。
- 智能结构体 ：tuple，可取代简单组合数据的 struct。[①]
- 智能参数 ：call_traits，可取代简单的函数接口参数声明。
- 智能数组 ：array/vector，可代替原生静态数组和动态数组。

尽量使用 exception 代替错误返回码

异常已经成为了 C++标准里的一个不可或缺的组成部分，不论我们编写的代码是否使用异常，C++都会使用异常处理机制，因此我们不必担心异常处理会带来额外的运行成本。相反，我们应该充分利用异常处理机制，简化对错误（即异常）的处理，摈弃 C 风格的错误返回码的方式——代码的主处理流程会只有正常的处理逻辑，不再需要麻烦的 if 错误检查语句，错误处理都会集中在某个特定位置的 catch 块中。

C++标准中提供了标准异常类 std::exception，但功能还比较有限，要把它用在自己的项目中还需要多做许多工作。boost.exception 库在标准异常的基础上进行了大幅度的强化，可以向异常对象中添加任意的信息，轻松构建出任意复杂的异常体系，增强了异常的表达能力，让异常处理变得更加简单。

使用异常时还需要避免过度使用 try-catch 块，避免多层嵌套，尽量多使用更简洁的 function-try 形式。

恰当地使用新式转型操作符

转型操作是 C 语言的“遗产”，通常我们应该尽量少用强制转型，因为这通常表明我们设计的接口有问题，真正好的代码应该是无须转型就可以使用的，不会发生编译警告。

但有的时候——通常是我们在与 C API 打交道时——我们又必须用强制转型，在这里建议读者透彻地理解标准库的四个新式转型操作符和 Boost 提供的几个转型工具，彻底消灭编译时的转型警告。虽然这样可能会让代码显得有些冗长繁杂，但它会增强代码的可靠性，避免了代码的含糊语义。

① 在 C 中 struct 关键字仅起到对数据打包组合的作用，但在 C++中 struct 基本上是 class 的同义词，在这里的意思是 tuple 可取代 C 用法的 struct。

新式转型操作符还有一个额外的好处，可以很容易地使用诸如 grep/sed/awk 等文本工具处理。

使用 test 库进行测试驱动开发

测试是软件开发过程中一项非常重要的工作，极限编程/敏捷开发都强调测试驱动开发，要求测试代码先于功能代码开发，功能代码开发的目的就是保证测试代码通过。

虽然不是所有工程都适合使用极限编程，但对测试的重视无论如何都是非常必要的。在 boost.test 库出现之前，为 C++代码编写单元测试是一项颇为麻烦的工作，需要做大量与测试本身无关的外围工作。boost.test 库构建了一套完整的测试框架，我们只需要使用几个简单的宏就可以创建整套的测试夹具、测试用例和测试套件，大大简化了编写测试代码的工作，使之成为了一种“享受”，令人简直无法抗拒写单元测试的诱惑。

多使用 Boost 小工具来增强代码的健壮性

Boost 程序库包罗万象，其中的组件大的分类就达到上百个，而每个组件里面还有更多更小的组成部分，让人眼花缭乱。很多程序员常常会把注意力集中在 Boost 库中的那些重量级组件上，而不自觉地忽略了那些功能小而简单的组件，认为这些小工具无关大局，用不用无所谓——个人认为这种想法是相当错误的，正所谓“勿以善小而不为”。

所有的成功都是由无数的细节搭建起来的，编写程序也是一样，Boost 库中有许多非常实用的小工具，适当地使用它们可以很好地增强代码的可读性和健壮性。

- foreach 可以很容易地遍历容器内所有元素。
- noncopyable 可以明确地定义一个不允许复制的类。
- integer 定义了标准整数。
- factory 封装了 new 操作符。
- array 封装了原始数组概念。
- tribool 提供三态布尔逻辑。
- utility 库包含数个有用的小工具。

学习并熟练使用这些小工具将会丰富我们的编程词汇，使我们的每一行代码都能够达到 Boost 源码那样的优雅程度。

15.2 内存管理

学会使用 Boost 管理内存

C++在可预见的一段时期内还不会有垃圾回收机制，手工管理内存还是程序员们必需的工作，这虽然给予了我们最大的灵活性，但也带来了许多的麻烦。

基本的内存管理方法有很多，可以使用 C 语言的 malloc/free 分配释放内存，也可以使用 new/delete 关键字分配释放内存，我们也可以重载 operator new/delete 来定制内存分配策略，但这些方式都比较“原始”。

Boost 使用 pool 库提供了一个内存池功能，它基于简单分隔存储的思想，不一定是最好最高效的内存池实现，但足够快速，并且简单易用。我们可以使用它预先向系统申请大块的内存，然后在里面自行取用，避免了反复用 new/delete 向系统申请释放内存，可以极大地提高内存的使用效率。

智能指针和指针容器也是管理内存的有效手段，它们不仅能够管理 new 分配的内存，也可以管理内存池分配的内存（使用删除器或克隆分配器）。

使用智能指针代替原始指针

指针是 C/C++中一个重要的概念，但它用起来缺乏足够的安全性，会导致很多潜在的问题。智能指针是一个“伟大的发明”，它使用 RAII 很好地解决了内存泄漏的问题，同时又没有带来过多的性能损失，对于广大程序员来说是不可或缺的工具。

boost.smart_ptr 库提供了数个优秀的智能指针，特别是其中采用了引用计数的 shared_ptr，几乎完全可以代替原始裸指针，其应用范围非常广泛，我们应该尽量在自己的代码中使用。

尽量避免直接使用 new/delete

智能指针和 checked_delete 可以很好地消除 delete 关键字的使用，但 new 关键字的调用有时候还是不可避免。好在 Boost 也给我们提供了替代的方式，例如 make_shared()和 factory，前者可以创建 shared_ptr，后者更可以创建任意的指针类型。这些 Boost 工具通过引入间接层封装了原始概念的使用，因而可以更好地对内存进行管理。

15.3 容器、迭代器和算法

了解现有的容器

STL 和 Boost 都提供了大量的容器，种类繁多，要从中选择一个适合于自己应用的容器并不是一件容易的事情，把这些容器分门别类进行整理划分有助于我们快速定位所需的容器。

- 按实现方式可分为侵入式容器和非侵入式容器，侵入式容器目前仅有 boost.intrusive，其他的容器都属于非侵入式容器。
- 按容纳元素的类型可分为值容器和指针容器，值容器容纳的是元素的拷贝，指针容器容纳的是指针。
- 按容器的数据结构可分为线性容器、树结构容器和散列容器，三者各有优缺点。
- 按元素的访问方式可分为序列容器和关联容器，而多索引容器 boost.multi_index 则结合了两者的特点。
- 此外还有编译期可容纳类型的 mpl 容器，用于模板元编程。

有了这些容器，我们就完全可以不使用 C/C++在底层语言级别提供的数组，可以更高效地利用资源。最常用的容器是 vector、map、array 和 unordered，其他的容器则应该根据具体的应用场景取舍。

还需要注意的是，这些容器基本上都不保证线程安全，在并发环境中使用时如果必要需加锁。

恰当地使用指针容器

指针容器是 Boost 库对容器做出的一个重要扩展，它可以安全地容纳指针，消除了元素拷贝的代价，同时它具有与标准容器基本相同的接口，因而学习成本低上手容易。

与标准容器相比指针容器有很多优点，特别适用于那些需要管理大对象的场合，这时管理指针比管理对象的拷贝要高效许多。但使用它也不是没有代价的，由于指针的特殊性，它的基本概念要比标准容器复杂（如对空指针的处理），涉及一些底层细节时如果不留心则很可能会出错。

因此，恰当并且审慎地对待指针容器才是正确的做法。

编写新容器或迭代器应满足概念要求

虽然 STL 和 Boost 提供了足够多的容器，但有的时候我们还是需要编写新的容器或者迭代

器来操纵特定的数据结构，这时不应仅仅以实现容器或迭代器的可用接口为目标，而是应该提高要求——新的容器或迭代器应该满足标准概念，这样它才能更好地与 STL 或 Boost 库的组件配合工作。

很多 Boost 组件都要求容器或迭代器满足标准概念，例如，如果容器的迭代器没有定义 iterator_category，那么就无法使用 boost.foreach。

对于容器来说，它应该具有一些必备的接口，如 size()、max_size()、begin()、end()等，同时还必须有大量的内部 traits 类型定义；对于迭代器来说则应该具有 operator*、operator->等接口，还要有 value_type、reference 等 traits 定义。对于容器或迭代器是否满足标准概念的要求则可以使用概念检查库 concept_check 来验证。

多使用 boost.foreach 算法

循环是计算机程序的基本结构，遍历一个容器中的所有元素是我们经常要做的工作。标准库的 for_each 算法就是为此准备的，它需要指定容器的两个端点，然后对其中的元素执行某个操作。但它也有缺点，为了使用 for_each 我们必须额外编写一个函数或函数对象，本来应该减少的代码编写工作反而增加了，这也令很多人不愿意使用 for_each 算法，而宁愿继续手写 for 循环。

boost.foreach 使用模板元编程技术彻底地解决了遍历容器的问题，只需要一个简单的宏就可以正序或逆序遍历容器里的所有元素，用法与 for 循环完全一样，这样的“语法糖”带来的好处是显而易见的，绝不只是少打几个字符那么简单。

虽然 C++11/14 标准扩展了 for 关键字，支持区间遍历，但它只能正向遍历区间，而且不支持迭代器的 pair，所以 boost.foreach 仍然很有用。

15.4 其他

熟悉 Boost 库中已有的设计模式实现

设计模式是领域专家的经验结晶，给出了针对特定问题的最佳解决方案，自 1995 年推荐书目[1] 出版以来，各式各样的模式大量涌现，许多常见的问题都可以找到对应的模式，极大地促进了软件业的发展。

需要强调的是，设计模式不单纯指面向对象的解决方案，不是单纯的类的嵌套与组合，它更是一种解决问题的思想和思路。虽然经典的 23 个模式都是基于面向对象的，而 STL 和 Boost 主要使用的是泛型，但这并不影响设计模式的使用。

Boost 库中的许多组件的设计和实现都深受设计模式的影响（例如代理模式就有 smart_ptr、optional、array 等，推荐书目[3]和[9]对此有较好的总结，读者可参考），熟悉这些组件使用的设计模式不但可以加深对 Boost 的了解，而且也可以反过来加深对设计模式的了解，可谓一举两得。

熟悉多线程领域的基本概念和模式

boost.thread 库提供了用于多线程编程的大量工具，方便易用，但仅有这些工具还是不够的，如果不明白如何正确地使用这些工具那么它们也无法发挥出应有的作用。

多线程开发有许多不同于单线程开发的新问题，要避免或者解决这些问题就需要透彻理解线程安全、可重入等概念，在编写代码时时刻考虑多线程并发的情况，避免使用全局变量、成员变量或静态变量，根据具体情况使用 boost.thread 提供的互斥量、条件变量和线程局部存储等功能。

多线程开发还有许多成熟的范式，如主动对象、future 等，熟悉这些范式有助于我们构建更稳固的多线程程序。

留意 Boost 与 C++11/14 的区别

Boost 被称为“C++‘准’标准库”，是 C++标准的试验场和后备军，许多新组件都需要经过 Boost 的锤炼然后才能进入标准。C++11 中来自 Boost 库的就有 array、bind、function、ref、shared_ptr、system、type_traits，今后则可能有 optional、filesystem。

对于那些不是来自 Boost 的标准组件，Boost 也提供了实现（如 chrono、unordered），使我们可以毫无困难地在 Boost 实现与标准实现之间自由切换。此外，Boost 还用模板元编程技术模拟实现了 auto、range-based for 等 C++11/14 中才有的新特性。

但在使用 Boost 时需要注意，有的组件并不是完全符合 C++11 标准的，虽然大部分功能一致，但也有极少的部分存在差别，如果使用到这些存在差异的细节，那么就可能会遇到不可移植的问题。

谨慎使用 bind、正则表达式、模板元编程等功能强大且灵活的库

Boost 中有许多功能强大的组件，它们专注于某一特定领域，用法灵活复杂，能够解决许多实际的问题。例如 bind 就被应用于函数式编程，使用占位符创建出新的函数对象，是 function、signals2、thread、asio 等库的必备搭档。但这些组件功能强大的另一面是用法的复杂，学习并掌握它们需要花费相当多的时间，并且过于灵活的用法也会产生副作用——使代码有如“天书”，增加了后续维护的成本。

“过犹不及”，谨慎地使用这些高级技术会让程序更易读——我们编写的代码决不能成为个人编程水平的技术炫耀，代码写出来应该是给“人”看而不是给“计算机”看的。

让你周围的同事熟悉 C++11/14 和 Boost

作为最后一条，这也许是最重要的。

软件开发绝不是一个人短期能够完成的事情，而是许多人协同的长期工作，包含了需求、设计、测试、维护等多个阶段。C++11/14 和 Boost 的确是强大的武器，但正所谓“独乐乐，众乐乐，孰乐？”，如果仅仅只有自己一个人掌握了 C++的高级用法，那么他在工作过程中就避免不了“自说自话”的窘境，写出的代码没有人能看懂，也就没有人能够为他做 code review，发现其中可能存在的 bug，他将不得不“孤军奋战”，一个人自己去研究解决使用中遇到的各种问题，可谓“高处不胜寒”。[①]

知识的传递是一件快乐的事，好的东西更应该与大家一起分享，把 C++、Boost 的最前沿开发技术在自己的周围传授可以达到“众人拾柴火焰高”的境界。在这个知识分享的过程中我们不会有任何的损失，相反，还有可能因为不同人的思维角度的不同而产生“头脑风暴”的效果，迸发出更多的思维火花，进一步加深我们对 C++和 Boost 的理解，提高我们的编程水平——这也正是笔者编写本书的原始动力。

15.5 结束语

本章从笔者自身经历出发，简要阐述了一些在实际开发工作中使用现代 C++的经验，限于文笔只能表述出一鳞半爪，不能说是金科玉律，但至少是肺腑之言。

开发出高质量高性能的软件是每一个程序员的梦想，每一个人都有他自己的 Effective 开发习惯，但有一点肯定是共通的：一个 Effective 程序员必定是一个快乐的程序员，希望读者在阅读完本章乃至本书后都能够 Effective and Happy，高质量且快乐地度过生活中的每一天。

---

① 当然也不排除少数人就喜欢这种“独孤求败”“唯我独尊”的感觉，但这样不合群的“技术专家”还是越少越好，希望读者没有遇到过。

附录 A 推荐书目

[1] Gamma E, Helm R, Johnson R. 李英军，马晓星，蔡敏等，译.《设计模式：可复用面向对象软件的基础》

这本书永远位于作者推荐榜的最前列，软件开发历史上里程碑式的著作，设计模式的开山作品，内容权威经典，是每一个精益求精的程序员都必须拥有的宝典和圣经，值得经常阅读以获取设计灵感。

[2] Nicolai M J. 侯捷，译.《C++标准库》（第 2 版）

这本书同样是作者强力推荐的 C++书籍，全面分析讲解 C++11 标准库，无论是初学者还是高手都能从中获益，C++程序员必备，厚达千余页的大部头组织得井井有条，查阅起来毫不费力。

[3] 罗剑锋.《Boost 程序库完全开发指南——深入 C++“准”标准库》（第 3 版）

本书是国内的第一本详细介绍 Boost 程序库的技术书籍，第 3 版根据 C++11/14 做了全面更新，面向 Boost 初学者，内容详细丰富，可以放在手头随时查阅，是一本很好的 Boost 工具书。

[4] Stroustrup B. 裘宗燕，译.《C++语言的设计和演化》

C++语言之父所著，介绍了 C++语言的发展历史、设计理念，同时也详细描述了 C++的各种语言特性，从中可以更深刻地理解 C++语言的内涵，虽然出版时间较早但仍不失为一本好的参考书。

[5] Stanley B L, Lajoie, Barbara E M. 王刚，杨巨峰，译.《C++ Primer》（第 5 版）

C++入门的经典教材，适合编程新手循序渐进地学习 C++语言。

[6] Meyers S. 侯捷，译。《Effective C++ 中文版》(第 3 版)
享誉世界的“Effective”系列，提供了 55 个如何编写高效 C++代码的忠告。

[7] Fowler M. 熊节，译.《重构：改善既有代码的设计》
本书讲述了若干“代码坏味”和重构准则，有的甚至相当简单琐碎，但的确是很有价值的实用工程准则，了解这些准则有助于我们更好地提高代码质量。不过全书使用 Java 编写范例代码，对于我们这些 C++程序员来说阅读起来略微有些不便。

[8] Sutter H, Alexandrescu A. 等著，刘基诚，译.《C++编程规范：101 条规则、准则和最佳实践》
本书由两位 C++大师联手撰写，类似《Effective C++》，把专家们数十年的经验和智慧凝结成了 101 条简练的规范，熟悉并运用这些准则可以大大提高我们的 C++编码水平。

[9] 罗剑锋.《Nginx 模块开发指南：使用 C++11 和 Boost 程序库》
本书深入解析了当前最流行的高性能 Web 服务器 Nginx，详细讲解如何使用 C++11 和 Boost 程序库来开发定制 Nginx，实践了模板元编程、lambda 表达式等众多现代 C++编程范式。

附录 B

Boost 程序库组件索引

本部分按字母顺序列出了 Boost 1.60 版（2015 年 12 月发布）中包含的所有组件，有*标记的组件表示在推荐书目[3] 中有详细阐述，黑体字表示该组件对应 C++11/14 标准。

A

accumulators：是一个用于增量统计的库，也是一个用于增量计算的可扩展的累加器框架，可以看作是 std::accumulate 算法的扩展。

algorithm*：包含了一些有用的小算法和 C++11/14 算法的实现，如 Boyer-Moore Search、KMP、all_of、any_of、none_of 等。

align：处理内存对齐相关的各种工具函数和类。

any*：可以容纳任意类型数据的容器，有了它，C++的强类型特性似乎失去了效力。

array*：包装了 C++语言内建的数组，为其提供标准容器接口，速度、性能上与原始数组相差无几。

asio*：基于操作系统提供的异步机制，采用前摄器设计模式（proactor）实现了可移植的异步（或者同步）IO 操作。它主要关注于网络通信方面，封装 Socket API 提供了一个现代 C++风格的网络编程接口。但 asio 的异步操作并不局限于网络编程，还支持 UNIX 信号、串口读写、定时器、SSL 等功能。

assert*：增强的断言宏。

assign*：重载了赋值操作符、逗号操作符和括号操作符，可以用简洁的语法非常方便地对容器赋值或者初始化，在需要填入大量初值的地方很有用。

atomic*：实现了 C++11 标准定义的原子操作库。

B

bimap*：扩展了标准库的映射型容器，提供双向映射的能力，功能强大，其接口被特意设计为符合标准容器规范，以减少学习的成本。

bind*：对 C++98 标准库中函数适配器 bind1st/bind2nd 的泛化和增强，可以适配任意的可调用对象，包括函数指针、函数引用、成员函数指针和函数对象，完全符合 C++11 标准。

C

call_traits：封装了可能是最好的给函数传递参数的方式，会自动推导出最高效的传递参数的类型。

chrono*:实现了 C++标准里的时间处理库，侧重于对计算机系统时间而不是日期的处理。

circular_buffer*：实现了循环缓冲区的数据结构，支持标准的容器操作，但大小是固定的，当到达容器末尾时将自动循环利用容器另一端的空间。

compatibility:主要用于 Boost 库作者,对于某些不符合标准的实现提供变通解决办法。

compressed_pair：提供一个与 std::pair 非常相似的模板类 compressed_pair，用法也基本相同，不同之处在于 compressed_pair 使用了空基类优化技术。

concept check：实现了 C++11 标准中被否决的概念检查功能，可以在编译期检查模板类型参数是否符合某个概念。

config*：主要用于 Boost 库作者，解决 Boost 库组件对各种编译器、平台的兼容性问题。

container:完全实现了 C++11/14 中的所有容器,并增加了 flat_map、stable_vector 等新容器。秉承 Boost 的优良品质，是一份很好的 STL 实现产品。

context：提供在单线程下多任务协作的功能，由于使用了汇编代码，所以需要特定的编译器和 CPU 支持。

conversion：增强了 C++标准中的 static_cast、dynamic_cast 等转型操作符。

convert：可扩展的类型转换框架，相当于泛化的 lexical_cast。

core*：Boost 程序库的核心工具集，包含数个小工具，例如 addressof、enable_if、noncopyable 等。

coroutine：它依赖于 context 库，提供并发的例程操作，也就是所谓的“协程”或“纤

程”。

coroutine2：功能同 coroutine，但仅支持 C++14。

crc*：实现了计算循环冗余码（CRC）的功能。

D

date_time*：非常全面且灵活的日期时间库，基于日常使用的公历提供时间相关的各种功能。

dynamic_bitset*：类似于标准库的 bitset 或者 vector<bool>，提供丰富的位运算，同时长度又是动态可变的。

E

enable_if：允许模板函数或者模板类仅针对某些特定类型有效，即启用或禁用某些特化形式，依赖于 SFINAE。

endian：处理计算机里字节序“大小端”的问题。

exception*：针对标准库中异常类的缺陷进行了强化，提供 operator<<重载，可以向异常传入任意数据，有助于增加异常的表达力。

F

filesystem*：可移植的文件系统操作库，使用 POSIX 标准表示文件系统的路径，可以写出跨平台通用的程序，即将进入 C++17 标准。

flyweight：实现了享元设计模式，可以管理大量的小对象以节约内存的使用。

foreach*：使用宏提供 C++11 风格的序列遍历，简便好用，不需要使用麻烦的迭代器，也不需要定义新的函数对象。

format*：实现了类似于 printf()的格式化对象，可以把参数格式化到一个字符串，而且是完全类型安全的。

function*：函数对象的“容器”，概念上像是函数指针类型的泛化，是一种“智能函数指针”，能够容纳任意符合函数签名的可调用对象，经常搭配 bind/lambda 表达式使用。

function_types：提供了对函数、函数指针、函数引用和成员函数指针等类型进行分类、分解和合成的功能。

functional：增强了 STL 中的函数对象适配器。

functional/factory：工厂模式的实践，封装了new操作符。

functional/forward：函数对象的适配器，解决了使用右值的问题。

functional/hash：实现了C++11中定义的散列函数，可以对C++的内置类型和标准容器计算散列值，也可以扩展它使其支持自定义类型。

functional/overloaded_function：用一个函数对象包装各种重载函数（在模板参数中给出函数签名），对外提供统一的operator()。

fusion：基于tuple的编译期容器和算法，是模板元编程的强大工具，可以与mpl很好地协同工作。

G

geometry：几何图形处理库，功能包括求面积、周长、凸壳、距离等。

gil：通用图像库，为像素、色彩、通道等图像处理概念提供了泛型的容器和算法，可以对图像做灰度化、梯度、均值、旋转等许多运算，支持JPG、PNG、TIFF等文件格式。

graph：处理离散数学中的图结构，并提供图、矩阵等数据结构上的泛型算法，可以看作是STL在非线性容器领域的扩展。

H

heap：类似于std::priority_queue的优先队列，但功能更多。

I

icl：提供了可以容纳"区间"的容器和其上的多种运算，不仅可以处理数学意义上的区间，也可以处理时间等其他领域的区间。

identity_type：使用宏包装类型，能够在其他宏里使用含有"逗号"的模板类。

integer*：一组有关整数处理的头文件和类，具有良好的可移植性，让C++能够更方便、更准确、更容易地处理整数类型。

interprocess：可移植的进程间通信（IPC）的功能，并提供了简洁易用的STL风格接口。

interval：处理"区间"概念相关的数学问题，把一般的算术运算扩展到了区间上，可以对区间执行各种运算。

intrusive：侵入式容器和算法，提供list、tree、map等与标准库几乎等价的容器。

in_place_factory：允许就地直接构造对象而不需要一个临时对象的拷贝，类似于 C++11 的 emplace 方法。

io state savers：可以简化恢复流状态的工作，保存流的当前状态，自动或者由程序员控制恢复流的状态。

iostreams：扩展了 C++标准库的流处理，建立了一个流处理框架，使得编写流式处理更加容易。

iterators：定义了一组新的迭代器概念、构造框架和有用的适配器，能够帮助程序员更轻松地实现迭代器模式。

L

lambda：为 C++引入了 lambda 表达式和函数式编程，可以就地创建小型的函数对象。

lexical_cast*：进行“字面值”的转换，类似 C 中的 atoi 函数，可以进行字符串、整数/浮点数之间的字面转换。

local_function：可以在函数内部定义嵌套函数，很像 C++11/14 的 lambda 表达式，但兼容 C++98。

locale：基于 std::locale 提供了强大的本地化功能，支持 unicode 和 utf，支持 C++11 的新字符类型 char16_t 和 char32_t，可以执行拼写检查、字符集转换、断词等许多有用的功能。

lockfree：非阻塞的队列和栈，可由多个生产者/消费者同时操作。

log：功能强大又简单易用的日志库，使用类似于 iostreams 的 source、sink、filter 等概念。

M

math*：包含了大量数学领域的模板类和算法。

math/common_factor：最大公约数和最小公倍数。

math/octonion：八元数。

math/quaternion：四元数。

math/special_functions：大量近现代数学函数。

math/statistical distributions：大量的统计分布和函数。

mem_fn：类似于 bind 的函数绑定器，用于绑定成员函数。

meta state machine：使用模板元编程技术实现的高性能的有限状态机。

minmax*：对标准库中的算法 min/max 和 min_element/max_element 的增强，可在一次处理中同时获取最大最小值。

move：使用包装类手法模拟实现了 C++11 标准中的转移（move）语义，兼容 C++98。

MPI：用于高性能分布式并行计算应用的开发，封装了标准的 MPI（消息传送接口）以便更好地支持现代 C++编程风格。

mpl：模板元编程框架，包含编译期的算法、容器和函数等完整的元编程工具。

multi_array*：多维容器，高效地实现了 STL 风格的多维数组，比使用原始多维数组或者 vector<vector>更好。

multi_index：实现了具有多个 STL 兼容访问接口（索引）的容器。

multiprecision：高精度的整数、有理数和浮点数。

N

numeric conversion：提供用于安全数字转型的一组工具。

O

odeint：解决常微分方程的初值问题。

operators*：只要用户在自己的类里定义少量的操作符（如 operator<），就可以自动生成其他操作符重载，而且保证正确的语义实现。

optional*：使用“容器”语义，包装了“可能产生无效值”的对象，实现了“未初始化”的概念，即将进入 C++17 标准。

P

parameter：提供了类似于 Python 等语言里的命名参数的特性（是在 C++标准化过程中曾经被拒绝的特性之一），可以使用参数名来指定函数参数值。

phoenix：另一个函数式编程的强大工具，与 lambda 类似。

pointer container：提供了与标准容器类似的若干种指针容器，性能较好且异常安全。

polygon：处理平面多边形的库，功能强大。

pool*：基于简单分隔存储思想实现了一个快速、紧凑的内存池，不仅能够管理大量的对象，还可以被用作 STL 的内存分配器。

predef：定义了一整套操作系统和编译器相关的宏，有利于简化条件编译。

preprocessor：类似于模板元编程的预处理元编程，发生在编译之前的预处理阶段。

program_options*：实现了非常完善的程序配置选项处理功能，不仅能够分析命令行，还能够从配置文件甚至环境变量中获取参数。

property map：提供 key-value 映射的属性概念定义，为从键到值的映射定义了一个通用接口。

property tree*：保存多个属性值的树形数据结构，可以用类似路径的简单方式访问任意节点的属性，而且每个节点都可以用类似于 STL 的风格遍历子节点。

proto：允许在 C++中构建专用领域嵌入式语言，基于表达式模板技术定义小型专用语言的“编译器”。

python：可以很容易地在 Python 和 C++之间自由转换，全面支持 C++和 Python 的各种特性，包括 C++到 Python 的异常转换、默认参数、关键字参数、引用和指针等。

R

random*：可以产生高质量的随机数，并提供随机数发生器、分布等很多有用的数学、统计学相关概念。

range：基于 STL 迭代器提出了“范围”的概念——容器的半开区间，相当于一个迭代器的 pair。

ratio*：编译期的有理数运算，使用了模板元编程技术。

rational*：实现了有理数概念，完善了 C++的数学域，运算时没有精度的损失。

ref*：应用代理模式，引入对象引用的包装器概念解决了拷贝引用的问题，是一个“智能引用”。

regex：正则表达式库，可用于字符串的匹配、查找和替换。

result_of：帮助程序员确定一个调用表达式的返回类型，主要用于泛型编程和其他 Boost 库组件。

S

scope_exit：使用 preprocessor 库的预处理技术实现在退出作用域时的资源自动释放，也可以执行任意的代码，等价于 shared_ptr<void>+lambda 表达式的用法。

serialization：实现了 C++数据结构的持久化，可以把任意的 C++对象整编为一串二进制字节流或者文本，然后再在需要的时候解整编恢复为原来的对象。

signals：实现了信号/插槽机制，是观察者模式的具体实践，现已经被废弃，请使用 signals2。

signals2*：线程安全的信号/插槽机制，无须编译即可使用。

smart_ptr*：提供 6 种智能指针，包括最“智能”的 shared_ptr。

sort：比标准库里的 sort 更好的排序算法。

spirit：面向对象的递归下降解析器的生成框架，它使用 EBNF 语法，是一个比正则表达式更强大的语法分析器。

statechart：功能完善的有限状态自动机框架，完全支持 UML 语义，可以从 UML 模型很方便地转换成 C++代码。

static_assert*：把断言的诊断时间由运行期提前到编译期，让编译器检查可能发生的错误，从而能够更好地增强程序的健壮性。

string_algo*：非常全面的字符串算法库，提供了大量的字符串操作函数，可以在不使用正则表达式的情况下处理大多数字符串的相关问题。

string_ref ：轻量级、高效的字符串常引用，类似于 const string&，但不需要构造一个临时字符串对象，其设计思想被引入 C++17 标准，但更名为 string_view。

swap*：增强了标准库提供的 std::swap，为交换两个变量值提供了便捷的方法，支持数组交换。

system*：使用轻量级的对象封装了操作系统底层的错误代码和错误信息，使调用底层接口的程序可以被很容易地移植到其他操作系统。

T

test*：提供用于单元测试的测试套件，还附带检测内存泄漏的功能，比其他的单元测试库更强大更方便好用。

thread*：为 C++增加了线程处理的功能，提供了简明清晰的线程、互斥量等概念，可以很容易地创建多线程应用程序。

timer*：提供简易的度量时间和进度显示功能，可以用于性能测试等需要计时的任务，对于大多数的情况而言它足够用。

tokenizer*：专门用于分词（token）的字符串处理库。

TR1：对 C++库扩展技术报告 TR1 的一个实现，现已经被废弃。

tribool*：类似于 C++内置的 bool 类型，但基于三态的布尔逻辑。

tti：类似于 type_traits，但使用宏的形式定义检查元函数，专门检查类内部成员（类型定义、成员变量、成员函数等）。

tuple*：定义了一个有固定数量元素的容器，其中的每个元素类型都可以不相同，是 std::pair 的泛化，可以从函数返回任意数量的值，也可以代替 struct 组合数据。

type_erasure：提供运行时的“类型擦除”，混合了动态多态（虚函数）和静态多态（模板）的特点，是功能更强的 any。

type_index：运行时/编译时的类型信息，比 type_info 和 std::type_index 更好。

type_traits：提供一组特征（trait）类，用于在编译期确定类型是否具有某些特征。

typeof：使用宏模拟了 C++11 新增的 auto 和 decltype 关键字，可以减少书写烦琐的变量类型声明的工作量，简化代码。

U

uBLAS：用于线性代数领域的数学库，比 std::valarray 要好得多，支持单位向量、稀疏向量、密集矩阵、稀疏矩阵、三角矩阵等许多线性代数概念。

units：实现了物理学的量纲处理，都是在编译期处理，没有运行时开销。

unordered*：完全符合 C++11 标准的散列容器实现。

utility*：集合了很多非常有用的小工具，如 checked_delete、declval。

uuid*：表示和生成 UUID。

V

value_initialized：可以保证变量在声明时被正确地初始化，拥有零值或者缺省值。

variant*：与 any 有些类似，是一种可变类型，是对 C/C++中 union 概念的增强和扩展。

VMD：预处理元编程工具，使用可变参数宏增强了 preprocessor 库。

W

wave：是使用 spirit 库开发的一个完全符合 C/C++标准的预处理器。

X

xpressive*：一个先进的、灵活的、功能强大的正则表达式库，提供了对正则表达式的全面支持，甚至可以用来构建语法解析器，有动态和静态两种用法。

附录 C

Boost 程序库安装简介

Boost 社区成立于 1998 年，目的是向 C++程序员提供免费的、同行审查的、可移植的高质量 C++源程序库。它强调程序库要与 C++标准库很好地共同工作，建立在“既有的实践”之上和提供参考实现，使得 Boost 库可以适合最后的标准化。

本部分简要介绍在 Linux 操作系统上安装 Boost 的方法。

虽然很多 Linux 发行版提供编译好的 Boost 程序库，可以使用 apt-get 或 yum 安装，但使用源码形式通常更好，能够及时获得 Boost 的最新版本。源码包可以从官方网站 http://www.boost.org 下载，本书使用的是 boost_1_60_0.tar.gz。

在解压缩后，首先要做的是基本配置（例如编译器检查）：

```
./bootstrap.sh                                  #编译前的配置工作
```

Boost 使用专门的 b2 工具来执行构建工作，使用如下的命令可以快捷安装：[1]

```
./b2 install                                    #快捷安装 Boost
```

这将指示 Boost 使用 release 模式，把头文件安装到/usr/local/include，把库文件安装到/usr/local/lib。

bootstrap.sh 和 b2 均支持很多的选项参数，可以对 Boost 的安装做更进一步的定制，例如：

```
./bootstrap.sh --show-libraries                 #列出需要编译的库组件
```

① b2 还支持使用 bjam 语言编写构建脚本，功能比 make、scons 更强大，语法也更方便，本书在 GitHub 上的示例源码均使用 b2 来编译构建。

```
./bootstrap.sh --prefix=/opt/boost                #安装到/opt/boost 目录

./b2 --with-date_time                             #仅编译 date_time 库
./b2 link=static install                          #编译安装静态库，作者推荐此方式
./b2 --buildtype=complete install                 #完全编译并安装
```

bootstrap.sh 和 b2 更多的选项可以使用--help 查看。

不过 Boost 程序库实在太过庞大，有的时候我们并不需要使用所有的功能，而只需要其中的一部分，这时使用 b2 完全安装就未免有些小题大做了。Boost 为此提供了另外一个工具：bcp。

顾名思义，bcp 的作用是拷贝 Boost 程序库，它可以将 Boost 库里的任意组件和相关的依赖提取到一个新的目录下，从而方便编译、部署或发布。

bcp 位于 Boost 源码包里的 tools/bcp 目录下，可以使用 b2 直接编译，即：

```
./b2 tools/bcp                                    #使用 b2 编译 bcp
```

编译后的 bcp 位于./dist/bin/目录下，并不会安装到系统目录里。

bcp 的用法很简单，指定 Boost 组件名或者头文件名就可以完成提取工作，例如：

```
#提取 noncopyable 组件及依赖
./dist/bin/bcp noncopyable ~/tmp/boost_noncopyable

#提取 smart_ptr.hpp 头文件及依赖
./dist/bin/bcp smart_ptr.hpp ~/tmp/boost_smart_ptr
```

bcp 在提取的时候还可以给 boost 名字空间重命名，需要使用选项--namespace：

```
#提取 array 组件，名字空间 boost 重命名为 one_punch
./dist/bin/bcp --namespace=one_punch array ~/tmp/boost_array
```

bcp 同样也有很多其他选项，读者可以使用--help 查看。

C++特色

C++具有良好的结构和绝佳的运行效率，可以开发系统级软件；它开创了许多新的现代编程范式，支持面向对象、泛型等技术，具有足够的抽象度，灵活方便，可以开发各种大型复杂的应用软件。在众多的编程语言中C++可称得上是“全能选手”，可上可下，大至企业级应用，小至嵌入式系统，几乎没有什么事情是C++做不到的。

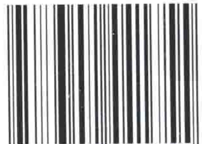
21世纪的第二个十年里，C++11/14标准终于“千呼万唤始出来”，基于现代编程理念对C++进行了大幅度的改造，从语言到标准库都面目一新。经过此番磨砺的C++焕发了“第二春”，变得更成熟稳重，正当壮年。现在，使用C++来开发程序，限制程序员能力的大概只有自己的想象力了。

C++11/14标准拥有大量的新特性，C++之父Bjarne Stroustrup是这样评价的：“一个全新的语言”（feels like a new language）。对于任何人——无论是初学者还是高级专家，C++11/14都值得去静下心来认真学习体会。

清华大学出版社数字出版网站
WQBook 文泉书局
www.wqbook.com

上架建议：C++

ISBN 978-7-302-44175-5



定价：79.00元