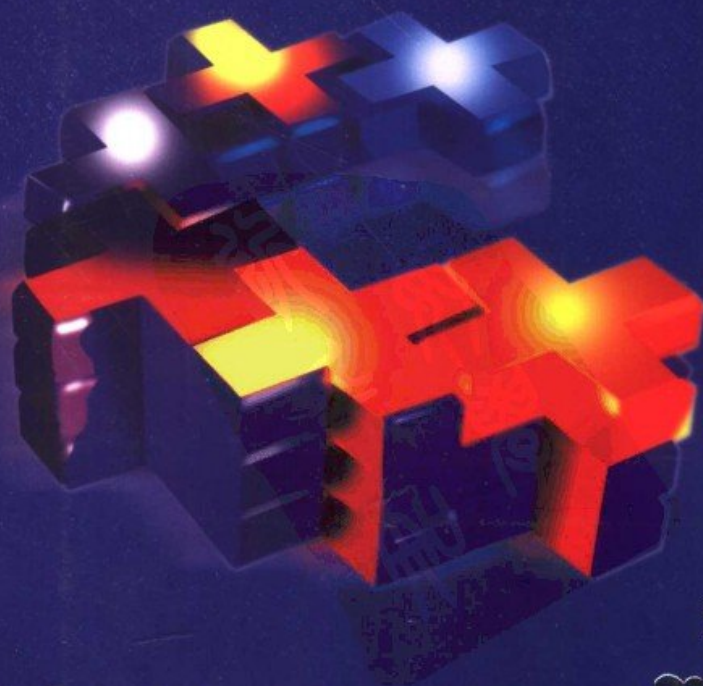


软件工程研究院

# 高质量 程序设计指南 ——C++/C语言 第3版



林 锐 韩永泉 编著  
飞思科技产品研发中心 监制



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



## 凝结众多项目经验 成就软件工程专家

把握软件工程各环节核心，在管理和技术的实践中，  
寻找符合我们真实需求的实用指南。

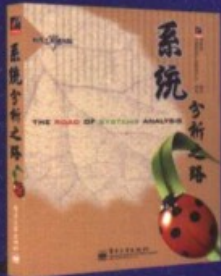
高质量程序设计是软件行业的薄弱环节，大部分企业只能依靠大量的测试和改错来提高软件产品的质量，为此付出了高昂的代价。因此，如何让程序员熟练地掌握编程技术和编程规范，在开发过程中内建高质量，是IT企业面临的主要挑战之一。

本书以轻松幽默的笔调向读者论述了高质量软件开发方法与C++/C编程规范。它是作者多年从事软件开发工作的经验总结，面向的主要读者对象是IT企业的程序员和项目经理，以及大专院校的本科生和研究生。

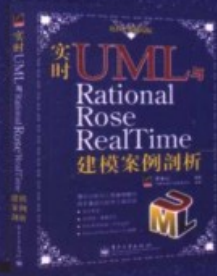
### ● 可以结合阅读的图书：



《C语言编程精要12讲》  
ISBN 7-121-03111-6  
定价：29.00元



《系统分析之路》  
ISBN 7-121-01137-9  
定价：49.00元



《实时UML与Rational Rose  
RealTime建模案例剖析》  
ISBN 978-7-121-03796-2  
定价：39.80元



《Project 2003在项目管理  
中的应用》  
ISBN 7-121-02075-0  
定价：35.00元

上架提示 C++/C /软件工程

飞思在线：<http://www.fecit.com.cn>  
飞思科技产品研发中心总策划



责任编辑：赵红梅

责任美编：李春瑞



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 978-7-121-04114-3



9 787121 041143 >

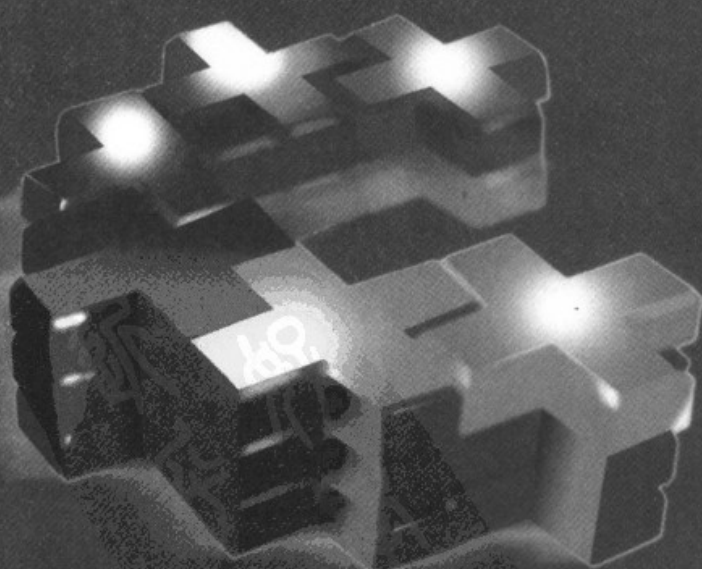
定价：39.80元



软件工程研究院

# 高质量 程序设计指南 ——C++/C语言 (第3版)

林 锐 韩永泉 编著  
飞思科技产品研发中心 监制



电子工业出版社  
Publishing House of Electronics Industry  
北京·BEIJING



## · 内 容 简 介 ·

高质量程序设计是软件行业的薄弱环节,大部分企业只能依靠大量的测试和改错来提高软件产品的质量,为此付出了高昂的代价。因此,如何让程序员熟练地掌握编程技术和编程规范,在开发过程中内建高质量代码,是 IT 企业面临的主要挑战之一。

本书以轻松幽默的笔调向读者论述了高质量软件开发方法与 C++/C 编程规范。它是作者多年从事软件开发工作的经验总结。本书共 17 章,第 1 章到第 4 章重点介绍软件质量和基本的程序设计方法;第 5 章到第 16 章重点阐述 C++/C 编程风格、面向对象程序设计方法和一些技术专题;第 17 章阐述 STL 的原理和使用方法。

本书第 1 版和第 2 版部分章节曾经在 Internet 上广泛流传,被国内 IT 企业的不少软件开发人员采用。本书的附录 C《大学十年》是作者在网上发表的一个短篇传记,文中所描述的充满激情的学习和生活态度,感染了大批莘莘学子。

本书的主要读者对象是 IT 企业的程序员和项目经理,以及大专院校的本科生和研究生。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

### 图书在版编目(CIP)数据

高质量程序设计指南: C++/C 语言 / 林锐, 韩永泉编著. 3 版. — 北京: 电子工业出版社, 2007.5  
(软件工程研究院)

ISBN 978-7-121-04114-3

I. 高… II. ①林…②韩… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2007) 第 038798 号

责任编辑: 赵红梅

印 刷: 北京智力达印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 25.75 字数: 659.2 千字

印 次: 2007 年 5 月第 1 次印刷

印 数: 6 000 册 定价: 39.80 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线: (010) 88258888。



## 作者简介



### ~ 林 锐 ~

1973年生。1990年至1996年，就读于西安电子科技大学，获硕士学位。1997年至2000年，就读于浙江大学计算机系，获博士学位。大学期间两度被评为中国百名跨世纪优秀大学生，1996年获电子工业部科技进步二等奖，1997年获首届中国大学生电脑大赛软件展示一等奖。2000年7月加入上海贝尔有限公司，从事软件工程和CMM的研究推广工作，2003年7月当选为 Alcatel 集团软件工程专家。2004年初创建上海漫索计算机科技有限公司（<http://www.chinaspis.com>），致力于创作适合国内企业需求的软件研发管理解决方案，包括方法论和软件产品。工作期间出版著作7部。



## 作者简介



### ~ 韩永泉 ~

1975年生。1994年至2001年就读于西安电子科技大学计算机系，获硕士学位。2001年4月加入上海大唐移动通信技术有限公司，担任高级软件工程师，从事电信设备网管软件的设计和开发。2004年加入北京新岸线软件科技有限公司，从事数字电视相关软件产品的设计、开发和管理工作，曾负责所在公司与上海漫索计算机科技有限公司合作开展的软件过程改进和研发管理解决方案的实施工作，现为公司软件项目经理。他是面向对象和面向组件软件开发技术及编程技术的爱好者。



## 第3版

大约在2005年年初，本书第2版（和第1版）已经售完。至今本书仍然受到软件公司和C++程序员的关注，不断有读者询问我从何处可以买到本书、什么时候再版。

说来惭愧，我从2002年写完本书第1版后，再也没有接触过C++编程，现在对C++已经很陌生了。2004年1月我离开上海贝尔，创办了上海漫索计算机科技有限公司，专注于IT企业的研发管理整体解决方案（包括软件产品和咨询服务）。我自己已经从技术专家转型为企业管理者，关注商务多于软件技术。对于出版本书第3版，我的确心有余而力不足。幸好第2版的作者韩永泉仍然从事应用软件开发，宝刀未老，他全面操办了第3版，我只是挂名而已。

在撰写第3版的时候，为了更进一步突出本书一贯强调的“高质量程序设计”理念，对原书第2版的内容做了一些调整：

首先是对第2版进行了全面的修订，改正了所有已经发现的错误，并对原有部分章节的内容进行了补充；

其次，删除了第2版的第2章和第17章（名字空间和模板）。根据我们的观察，除非是开发类库等通用程序，第17章的内容在现阶段对应用软件开发人员一般不具有实际指导价值；

最后，增加了大约10个小节的内容，分散在各章中。这些增加的内容是实际应用软件开发过程中经常会用到的技术，可以显著地提高编程效率，增强软件的健壮性和可移植性。

不论本书第1版和第2版是好是差，它都被过度地使用了，产生了令作者始料不及的影响。本书的试题被国内软件公司大面积地用于C++程序员招聘考试，结果事先看过答案的应试者考了高分而被录取，还真有人向我致谢；也有不少人未看过答案而考了低分未被录取，在网上把作者骂一通。本书的试题和答案早在2002年就公开了，不知有多少人看过，我很奇怪怎么到现在还被煞有介事地用于考试。

本书第3版即将出版，我希望读者正确地使用本书：请您学习和应用您（或公司）认为好的东西，不要把本书当做标准来看待，不要全部照搬，也不必花费很多时间去争议本书是好还是坏。如果您发现书中的错误或不妥之处，请及时告知作者韩永泉，或发邮件至 [northwest\\_wolf@sina.com](mailto:northwest_wolf@sina.com)，或直接上他的 Blog 与他交流：[http://blog.csdn.net/northwest\\_wolf/](http://blog.csdn.net/northwest_wolf/)。



2007年1月

上海漫索计算机科技有限公司

<http://www.mansuo.com>

[linrui@mansuo.com](mailto:linrui@mansuo.com)



## 第2版

《高质量程序设计指南——C++/C 语言》第1版上市后，一度成为畅销书。网上评论甚多，褒贬参半。我们分析了读者的批评和建议，总结了本书第1版的主要不足：

由于第1版原本是企业的培训教材，初衷是为了帮助程序员提高程序质量，假设读者已经熟悉 C++/C 语法，所以内容薄而精练、前后章节不连贯，看起来更像专题讲座。出版社的宣传工作做得很好，本书吸引了很多 C++初学者和高级程序员。由于书中不讲解入门知识，导致很多初学者看不下去。有一些大学生听了我的讲座后，为表敬意特地买书让我签名，翻阅之后就塞进书架当做纪念品了。对于那些高级程序员而言，本书的大部分内容他们早已经熟悉，好不容易看到几处精彩的章节，却翻了几页就没有了，真是不过瘾啊。

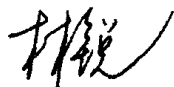
我的研究专长是软件工程和企业管理，而非程序设计。在 C++/C 编程方面自己仅仅是一名老工匠而已，我的确没有时间没有能力写出让初学者和高级程序员都喜欢的 C++/C 书籍。这本书炒作得过火，让我骑虎难下了。从2002年11月起，我就开始物色真正的 C++/C 高手来写本书的第2版。

恰好上海大唐移动通信设备有限公司的韩永泉正在为公司写 C++/C 培训教材，他也是本书第1版的读者。韩永泉提出了很有价值的建议和批评，并把他写的培训教材发给我看，真是自投罗网啊！

韩永泉是在西安电子科技大学计算机系读的本科和硕士，内功扎实。我和他碰头交谈了2小时，就把第2版托付给他了。两个月后，他把第2版的书稿交给我审阅。第2版的内容比第1版多了一倍，其广度符合我的设想，其深度完全出乎我的意料。为了阐述清楚 C++/C 程序之中的许多“为什么”，本书给出了大量的“提示、建议、规则”，并从编译器实现的角度论述原理。这种深度非一般教科书能比，我用了一个月时间才审阅并且学习完毕，删除了几十页过分深奥的内容（免得让我自己看昏倒）。我相信第2版可以让大多数高级程序员看过瘾了。

网上有一些忌世愤俗者认为计算机领域的每个分支都已经有了世界名著，不具有世界顶尖水平的中国人再写类似书籍都是欺世盗名的行为。这种极度自负和极度自卑的心态导致他们专爱骂国内作者。如果中国作者的书籍中的技术错误被他们抓住，经过放大、推理、演绎之后基本上就能断定作者是卑鄙之徒，于是砖头就拍过来了（简称“拍砖”）。拍砖者们遥相呼应，很快就能拍出江湖豪情，被拍的作者就成了倒霉蛋。有位好心的读者怕我经受不起，特意发给我一本拍砖大法——《拍砖十二流》以增强内功。

网上自由漫骂既是网络价值的体现又是民主的体现，这是物质文明和精神文明发展到一定境界的产物。《高质量程序设计指南——C++/C 语言》第2版即将出版，作者忐忑不安地等待第二轮“拍砖”。



2003年2月

上海贝尔阿尔卡特股份有限公司



## 第 1 版

软件质量是被许多程序员挂在嘴上而不是放在心上的东西！

除了完全外行和真正的编程高手外，初读本书，你最先的感受将是惊慌：“哇！我以前捏造的 C++/C 程序怎么会有那么多的毛病？”

别难过，作者只不过比你早几年、多几次惊慌而已。

请花几小时认真阅读这本经书，你将会受益匪浅，这是前面  $N-1$  个读者的建议。

### 编程老手与高手的误区

自从计算机问世以来，程序设计就成了令人羡慕的职业，程序员在受人宠爱之后容易发展成为毛病特多却常能自我臭美的群体。

如今在 Internet 上流传的“真正”的程序员据说是这样的：

(1) 真正的程序员没有进度表，只有讨好领导的马屁精才有进度表，真正的程序员会让领导提心吊胆。

(2) 真正的程序员不写使用说明书，用户应当自己去猜想程序的功能。

(3) 真正的程序员几乎不写代码的注释，如果注释很难写，它理所当然也很难读。

(4) 真正的程序员不画流程图，原始人和文盲才会干这事。

(5) 真正的程序员不看参考手册，新手和胆小鬼才会看。

(6) 真正的程序员不写文档也不需要文档，只有看不懂程序的笨蛋才用文档。

(7) 真正的程序员认为自己比用户更明白用户需要什么。

(8) 真正的程序员不接受团队开发的理念，除非他自己是头头。

(9) 真正的程序员的程序不会第一次就正确运行，但是他们愿意守着机器进行若干个 30 小时的调试改错。

(10) 真正的程序员不会在上午 9:00 到下午 5:00 之间工作，如果你看到他在上午 9:00 工作，这表明他从昨晚一直干到现在。

.....

具备上述特征越多，越显示程序员水平高，资格老。所以别奇怪，程序员的很多缺点竟然可以被当做优点来欣赏。就像在武侠小说中，那些独来独往、不受约束且带点邪气的高手最令人崇拜一样。我曾经也这样信奉，并且希望自己成为那样的“真正”的程序员，结果没有得到好下场。

我从读大学到博士毕业 10 年来一直勤奋好学，累计编写了数十万行 C++/C 代码。有这样的苦劳和疲劳，我应该称得上是编程老手了吧？

我开发的软件都与科研相关（集成电路 CAD 和 3D 图形学领域），动辄数万行程序，技术复杂，难度颇高。这些软件频频获奖，有一个软件获得首届中国大学生电脑大赛软件展示一等奖。在 1995 年开发的一套图形软件库到 2000 年还有人买。罗列出这些“业绩”，可以说明我算得上是编程高手了吧？

可惜这种个人感觉不等于事实。



读博期间我曾用一年时间开发了一个近 10 万行 C++ 代码的 3D 图形软件产品，我内心得意表面谦虚地向一位真正的软件高手请教。他虽然从未涉足过 3D 图形领域，却在几十分钟内指出该软件多处重大设计错误。让人感觉那套软件是用纸糊的华丽衣服，扯一下掉一块，戳一下破个洞。我目瞪口呆地意识到这套软件毫无实用价值，一年的心血白花了，并且害死了自己的软件公司。

人的顿悟通常发生在最心痛的时刻，在沮丧和心痛之后，我作了深刻反省，“面壁”半年，重新温习软件设计的基础知识。补修“内功”之后，又觉得腰板硬了起来。博士毕业前半年，我曾到微软中国研究院找工作，接受微软公司一位资深软件工程师的面试。他让我写函数 `strcpy` 的代码。

太容易了吧？

错！

这么一个小不点儿的函数，他从三个方面考查：

- (1) 编程风格；
- (2) 出错处理；
- (3) 算法复杂度分析（用于提高性能）。

在大学里从来没有人如此严格地考查过我的程序。我花了半小时，修改了数次，他还不尽满意，让我回家好好琢磨。我精神抖擞地进“考场”，大汗淋漓地出“考场”。这“高手”当得也太窝囊了。我又好好地反省了一次。

我把反省后的心得体会写成文章放在网上，引起了不少软件开发人员的共鸣。我因此有幸和国内大型 IT 企业如华为、上海贝尔、中兴等公司的同行们广泛交流。大家认为高质量与生产率是软件工程要解决的核心问题。高质量程序设计是非常重要的环节，毕竟软件是靠编程来实现的。

我们心目中的老手们和高手们能否编写出高质量的程序来？

不见得都能！

就我的经历与阅历来看，国内大学的计算机教育根本就没有灌输高质量程序设计的观念，教师们和学生们也很少自觉关心软件的质量。勤奋好学的程序员长期在低质量的程序堆中滚爬，吃尽苦头之后才有一些心得体会，长进极慢，我就是一例。

现在国内 IT 企业拥有学士、硕士、博士文凭的软件开发人员比比皆是，但他们在接受大学教育时就“先天不足”，岂能一到企业就突然实现质的飞跃。试问有多少软件开发人员对正确性、健壮性、可靠性、性能、易用性、清晰性、可扩展性、安全性、兼容性、可移植性等质量属性了如指掌？并且能在实践中运用自如？“高质量”可不是干活小心点就能实现的！

我们有充分的理由疑虑：

(1) 编程老手可能会长期用隐含错误的方式编程，习惯成自然后，被人指出发现毛病时都不愿相信那是真的！

(2) 编程高手可以在某一领域写出极有水平的代码，但未必能从全局把握软件质量的方方面面。

事实证明如此。我到上海贝尔工作后，陆续面试或测试过近百名“新”、“老”程序员的编程技能，合格率低于 50%。很少有人能够写出完全符合质量要求的 if 语句，很多程



序员对指针、内存管理一知半解……

领导们不敢相信这是真的。我做过现场试验：有一次部门新进 14 名硕士生，在开欢迎会之前对他们进行“C++/C 编程技能”摸底考试。我问大家试题难不难？所有的人都回答不难。结果没有一个人及格，有半数人得零分。

竞争对手如华为、中兴、大唐等公司的朋友们也做过试验，也是类似的结果。真的不是我“心狠手辣”或者要求过高（甚至变态），而是很多软件开发人员对自己的要求不够高。要知道这些大公司的员工素质在国内 IT 企业中是比较位列前茅前列的，倘若他们的编程质量都如此差的话，我们怎么敢期望中小公司拿出高质量的软件呢？连程序都编不好，还谈什么振兴民族软件产业。

多年来，我在软件开发过程中的苦头吃得实在太多了，现在总算被折磨清醒了。我打算定义编程老手和编程高手，请您别见笑。

定义 1：能长期稳定地编写出高质量程序的程序员称为编程老手。

定义 2：能长期稳定地编写出高难度、高质量程序的程序员称为编程高手。

根据上述定义，马上得到第一推论：我既不是高手也算不上是老手。

在写此书前，我阅读了不少程序设计方面的英文著作，越看越羞惭。因为发现自己在编程基本技能方面都未能全面掌握，顶多算是二流水平，还好意思谈什么老手和高手。希望和我一样在国内土生土长的程序员朋友们能够做到：

- (1) 知错就改；
- (2) 经常温故而知新；
- (3) 坚持学习，天天向上。

2002 年 4 月

上海贝尔阿尔卡特股份有限公司

《高质量程序设计指南——C++/C 语言》第 3 版终于完成了！第 2 版在 2003 年出版后，接着第 1 版的名声，仍然受到不少读者的青睐。但是，出版后发现了一些错误，不管这些错误是什么原因造成的，林锐博士都替我挨了不少“砖块”，这令我深感愧疚，深怕影响了林锐博士的好名声，更怕误人子弟，对热心读者产生误导。我觉得有责任将这些错误改正后予以发表，于是 2004 年年末，在诸多热心网友的指正下，整理出了一份勘误表并粘贴在了 CSDN 网站上，但是只有少数人知道并下载，现在偶尔还能收到热心网友的 E-mail 要求给他们发送一份勘误表。

这次应电子工业出版社之邀撰写本书第 3 版，说明本书还有很强的生命力，更重要的是：给了我一个全面纠正错误的机会，同时，这几年在工作中也有很多新的感受，我很想总结出来与大家分享。

在第 3 版完稿之际，特别要感谢为我指正错误的热心网友，他们是：方如坤、smart\_sonny、wfeng、fecita 等。感谢电子工业出版社飞思公司的编辑和策划，给我匡正失误、续写新篇的机会，没有他们的辛勤工作，本书现在不会出现在你的手上；这次帮我审稿的还有我的朋友陈亚明，他看到了许多我没有看到的问题，同时他也是 C++ 编程技术的爱好者，有他帮我审定稿件，相信有助于降低本版的错误率，感谢他的辛勤劳动；当然还有林锐博士，虽然我已穷思竭力避免错误产生，但毕竟金无足赤，新的错误仍然在所难免，林锐博士不免又要挡些“板砖”了。哈哈！还是欢迎大家多提批评意见。当然，有谁想夸我们两句，本人与林锐博士也是很愿意听到的☺。第 2 版也有不少网友对我们提出表扬，在此一并表示深深的感谢！

还有一个重要的感谢对象，那就是正怀着我孩子的老婆。虽然我不太希望我的孩子将来也从事 IT 行业，但我还是想用这本书的出版来庆祝我们的宝贝——兜兜——来到这个世界！

韩永泉

2007 年 1 月

北京新岸线软件科技有限公司

Yongquan.Han@nufrontsoft.com



<b>第1章 高质量软件开发之道</b> .....	1
1.1 软件质量基本概念 .....	1
1.1.1 如何理解软件的质量 .....	1
1.1.2 提高软件质量的基本方法 .....	3
1.1.3 “零缺陷”理念 .....	4
1.2 细说软件质量属性 .....	4
1.2.1 正确性 .....	4
1.2.2 健壮性 .....	5
1.2.3 可靠性 .....	5
1.2.4 性能 .....	6
1.2.5 易用性 .....	7
1.2.6 清晰性 .....	7
1.2.7 安全性 .....	7
1.2.8 可扩展性 .....	8
1.2.9 兼容性 .....	8
1.2.10 可移植性 .....	8
1.3 人们关注的不仅仅是质量 .....	9
1.3.1 质量、生产率和成本 之间的关系 .....	9
1.3.2 软件过程改进的基本概念 .....	11
1.4 高质量软件开发的基本方法 .....	13
1.4.1 建立软件过程规范 .....	13
1.4.2 复用 .....	15
1.4.3 分而治之 .....	16
1.4.4 优化与折中 .....	17
1.4.5 技术评审 .....	18
1.4.6 测试 .....	19
1.4.7 质量保证 .....	21
1.4.8 改错 .....	22
1.5 关于软件开发的一些常问的思考 .....	24
1.5.1 有最好的编程语言吗 .....	24
1.5.2 编程是一门艺术吗 .....	24
1.5.3 编程时应该多使用 技巧吗 .....	24
1.5.4 换更快的计算机还是 换更快的算法 .....	25

1.5.5 错误是否应该分等级 .....	25
1.5.6 一些错误的观念 .....	25
1.6 小结 .....	26
<b>第2章 编程语言发展简史</b> .....	27
2.1 编程语言大事记 .....	27
2.2 Ada 的故事 .....	30
2.3 C/C++ 发展简史 .....	31
2.4 Borland 与 Microsoft 之争 .....	32
2.5 Java 阵营与 Microsoft 的较量 .....	33
2.6 小结 .....	36
<b>第3章 程序的基本概念</b> .....	37
3.1 程序设计语言 .....	37
3.2 语言实现 .....	38
3.3 程序库 .....	40
3.4 开发环境 .....	40
3.5 程序的工作原理 .....	41
3.6 良好的编程习惯 .....	42
<b>第4章 C++/C 程序设计入门</b> .....	45
4.1 C++/C 程序的基本概念 .....	45
4.1.1 启动函数 main() .....	45
4.1.2 命令行参数 .....	47
4.1.3 内部名称 .....	48
4.1.4 连接规范 .....	49
4.1.5 变量及其初始化 .....	51
4.1.6 C Runtime Library .....	52
4.1.7 编译时和运行时的不同 .....	52
4.1.8 编译单元和独立编译技术 .....	54
4.2 基本数据类型和内存映像 .....	54
4.3 类型转换 .....	56
4.3.1 隐式转换 .....	56
4.3.2 强制转换 .....	58
4.4 标识符 .....	60
4.5 转义序列 .....	61
4.6 运算符 .....	62
4.7 表达式 .....	63
4.8 基本控制结构 .....	65
4.9 选择(判断)结构 .....	65

4.9.1 布尔变量与零值比较.....	66	6.13 使用 const 提高函数的 健壮性.....	118
4.9.2 整型变量与零值比较.....	67	6.13.1 用 const 修饰函数 的参数.....	118
4.9.3 浮点变量与零值比较.....	67	6.13.2 用 const 修饰函数的 返回值.....	119
4.9.4 指针变量与零值比较.....	69	<b>第 7 章 C++/C 指针、数组和 字符串</b> .....	121
4.9.5 对 if 语句的补充说明.....	70	7.1 指针.....	121
4.9.6 switch 结构.....	70	7.1.1 指针的本质.....	121
4.10 循环(重复)结构.....	71	7.1.2 指针的类型及其 支持的运算.....	123
4.10.1 for 语句的循环控制变量.....	72	7.1.3 指针传递.....	125
4.10.2 循环语句的效率.....	73	7.2 数组.....	126
4.11 结构化程序设计原理.....	78	7.2.1 数组的本质.....	126
4.12 goto/continue/break 语句.....	79	7.2.2 二维数组.....	128
4.13 示例.....	80	7.2.3 数组传递.....	129
<b>第 5 章 C++/C 常量</b> .....	85	7.2.4 动态创建、初始化和 删除数组的方法.....	131
5.1 认识常量.....	85	7.3 字符数组、字符指针和 字符串.....	133
5.1.1 字面常量.....	85	7.3.1 字符数组、字符串和 '\0'的关系.....	133
5.1.2 符号常量.....	86	7.3.2 字符指针的误区.....	134
5.1.3 契约性常量.....	87	7.3.3 字符串拷贝和比较.....	134
5.1.4 枚举常量.....	87	7.4 函数指针.....	135
5.2 正确定义符号常量.....	87	7.5 引用和指针的比较.....	137
5.3 const 与#define 的比较.....	88	<b>第 8 章 C++/C 高级数据类型</b> .....	141
5.4 类中的常量.....	89	8.1 结构(Struct).....	141
5.5 实际应用中如何定义常量.....	90	8.1.1 关键字 struct 与 class 的困惑.....	141
<b>第 6 章 C++/C 函数设计基础</b> .....	95	8.1.2 使用 struct.....	142
6.1 认识函数.....	95	8.1.3 位域.....	145
6.2 函数原型和定义.....	96	8.1.4 成员对齐.....	147
6.3 函数调用方式.....	97	8.2 联合(Union).....	159
6.4 认识函数堆栈.....	99	8.3 枚举(Enum).....	161
6.5 函数调用规范.....	100	8.4 文件.....	163
6.6 函数连接规范.....	101	<b>第 9 章 C++/C 编译预处理</b> .....	165
6.7 参数传递规则.....	102	9.1 文件包含.....	165
6.8 返回值的规则.....	104		
6.9 函数内部实现的规则.....	107		
6.10 存储类型及作用域规则.....	109		
6.10.1 存储类型.....	109		
6.10.2 作用域规则.....	110		
6.10.3 连接类型.....	111		
6.11 递归函数.....	113		
6.12 使用断言.....	116		



9.1.1 内部包含卫哨和 外部包含卫哨.....	165
9.1.2 头文件包含的合理顺序.....	166
9.2 宏定义.....	166
9.3 条件编译.....	169
9.3.1 #if、#elif 和 #else.....	169
9.3.2 #ifdef 和 #ifndef.....	170
9.4 #error.....	171
9.5 #pragma.....	171
9.6 #和##运算符.....	171
9.7 预定义符号常量.....	172
<b>第 10 章 C++/C 文件结构和 程序版式.....</b>	<b>175</b>
10.1 程序文件的目录结构.....	175
10.2 文件的结构.....	176
10.2.1 头文件的用途和结构.....	176
10.2.2 版权和版本信息.....	177
10.2.3 源文件结构.....	178
10.3 代码的版式.....	178
10.3.1 适当的空行.....	178
10.3.2 代码行及行内空格.....	179
10.3.3 长行拆分.....	180
10.3.4 对齐与缩进.....	181
10.3.5 修饰符的位置.....	182
10.3.6 注释风格.....	182
10.3.7 ADT/UDT 版式.....	183
<b>第 11 章 C++/C 应用程序 命名规则.....</b>	<b>185</b>
11.1 共性规则.....	185
11.2 简单的 Windows 应用 程序命名.....	186
<b>第 12 章 C++面向对象程序设计 方法概述.....</b>	<b>189</b>
12.1 漫谈面向对象.....	189
12.2 对象的概念.....	190
12.3 信息隐藏与类的封装.....	191
12.4 类的继承特性.....	195
12.5 类的组合特性.....	200
12.6 动态特性.....	201

12.6.1 虚函数.....	202
12.6.2 抽象基类.....	202
12.6.3 动态绑定.....	205
12.6.4 运行时多态.....	207
12.6.5 多态数组.....	208
<b>12.7 C++对象模型.....</b>	<b>215</b>
12.7.1 对象的内存映像.....	215
12.7.2 隐含成员.....	224
12.7.3 C++编译器如何 处理成员函数.....	225
12.7.4 C++编译器如何 处理静态成员.....	225
12.8 小结.....	226
<b>第 13 章 对象的初始化、 拷贝和析构.....</b>	<b>229</b>
13.1 构造函数与析构 函数的起源.....	229
13.2 为什么需要构造函数和 析构函数.....	230
13.3 构造函数的成员 初始化列表.....	232
13.4 对象的构造和析构次序.....	234
13.5 构造函数和析构函数的 调用时机.....	235
13.6 构造函数和赋值 函数的重载.....	236
13.7 示例：类 String 的 构造函数和析构函数.....	238
13.8 何时应该定义拷贝构造 函数和拷贝赋值函数.....	239
13.9 示例：类 String 的拷贝 构造函数和拷贝 赋值函数.....	240
13.10 用偷懒的办法处理拷贝 构造函数和拷贝赋值 函数.....	242
13.11 如何实现派生类的 基本函数.....	243

<b>第 14 章 C++函数的高级特性</b>	247
14.1 函数重载的概念	247
14.1.1 重载的起源	247
14.1.2 重载是如何实现的	247
14.1.3 当心隐式类型转换导致 重载函数产生二义性	249
14.2 成员函数的重载、 覆盖与隐藏	250
14.2.1 重载与覆盖	250
14.2.2 令人迷惑的隐藏规则	251
14.2.3 摆脱隐藏	253
14.3 参数的默认值	254
14.4 运算符重载	255
14.4.1 基本概念	255
14.4.2 运算符重载的特殊性	256
14.4.3 不能重载的运算符	257
14.4.4 重载++和--	257
14.5 函数内联	259
14.5.1 用函数内联取代宏	259
14.5.2 内联函数的编程风格	260
14.5.3 慎用内联	261
14.6 类型转换函数	261
14.7 const 成员函数	264
<b>第 15 章 C++异常处理和 RTTI</b>	267
15.1 为什么要使用异常处理	267
15.2 C++异常处理	268
15.2.1 异常处理的原理	268
15.2.2 异常类型和异常对象	269
15.2.3 异常处理的语法结构	270
15.2.4 异常的类型匹配规则	272
15.2.5 异常说明及其冲突	272
15.2.6 当异常抛出时局部 对象如何释放	273
15.2.7 对象构造和析构 期间的异常	273
15.2.8 如何使用好异常 处理技术	275
15.2.9 C++的标准异常	278
15.3 虚函数面临的难题	278

15.4 RTTI 及其构成	280
15.4.1 起源	280
15.4.2 typeid 运算符	281
15.4.3 dynamic_cast<>运算符	283
15.4.4 RTTI 的魅力与代价	285
<b>第 16 章 内存管理</b>	287
16.1 内存分配方式	287
16.2 常见的内存错误及 其对策	288
16.3 指针参数是如何 传递内存的	289
16.4 free 和 delete 把指针怎么啦	291
16.5 动态内存会被 自动释放吗	292
16.6 杜绝“野指针”	292
16.7 有了 malloc/free 为什么 还要 new/delete	293
16.8 malloc/free 的使用要点	295
16.9 new 有 3 种使用方式	296
16.9.1 plain new/delete	296
16.9.2 nothrow new/delete	297
16.9.3 placement new/delete	297
16.10 new/delete 的 使用要点	300
16.11 内存耗尽怎么办	301
16.12 用对象模拟指针	302
16.13 泛型指针 auto_ptr	305
16.14 带有引用计数的 智能指针	306
16.15 智能指针作为容器元素	310
<b>第 17 章 学习和使用 STL</b>	323
17.1 STL 简介	323
17.2 STL 头文件的分布	324
17.2.1 容器类	324
17.2.2 泛型算法	325
17.2.3 迭代器	325
17.2.4 数学运算库	325
17.2.5 通用工具	325



17.2.6 其他头文件.....	326
17.3 容器设计原理.....	326
17.3.1 内存映像.....	326
17.3.2 存储方式和访问方式.....	327
17.3.3 顺序容器和关联式容器的比较.....	328
17.3.4 如何遍历容器.....	331
17.3.5 存储空间重分配问题.....	332
17.3.6 什么样的对象才能作为 STL 容器的元素.....	333
17.4 迭代器.....	334
17.4.1 迭代器的本质.....	334
17.4.2 迭代器失效及其危险性.....	338
17.5 存储分配器.....	346
17.6 适配器.....	347
17.7 泛型算法.....	350
17.8 一些特殊的容器.....	354
17.8.1 string 类.....	354
17.8.2 bitset 并非 set.....	355
17.8.3 节省存储空间的 vector<bool>.....	357
17.8.4 空容器.....	358
17.9 STL 容器特征总结.....	360
17.10 STL 使用心得.....	362
附录 A C++/C 试题.....	365
附录 B C++/C 试题答案与评分标准.....	369
附录 C 大学十年.....	375
附录 D 《大学十年》后记.....	393
附录 E 术语与缩写解释.....	395
参考文献.....	397

# 第1章 高质量软件开发之道

本章讲述高质量软件开发的道理。

为了深入理解软件质量的概念，本章阐述了 10 个重要的软件质量因素，即正确性、健壮性、可靠性、性能、易用性、清晰性、安全性、可扩展性、兼容性和可移植性，并介绍了消除软件缺陷的基本方法。

人们开发软件产品的目的是赚钱。为了获得更多的利润，人们希望软件开发工作“做得好、做得快，并且少花钱”，所以软件质量并不是人们唯一关心的东西。本章论述了“质量、生产率和成本”之间的关系，并给出了能够“提高质量、提高生产率，并且降低成本”的软件开发方法。

## 1.1 软件质量基本概念

### 1.1.1 如何理解软件的质量

什么是质量？

词典的定义是：① 典型的或本质的特征；② 事物固有的或区别于其他事物的特征或本质；③ 优良或出色的程度。

CMM 对质量的定义是：① 一个系统、组件或过程符合特定需求的程度；② 一个系统、组件或过程符合客户或用户的要求或期望的程度。

上述定义很抽象，软件开发人员看了准会一脸迷惘。软件的质量不容易说清楚，但我们今天非得把它搞个水落石出不可。

就以健康做类比吧。早先人们以为长得结实、饭量大就是健康，这显然是不科学的。现代人总是通过考察多方面的生理因素来判断是否健康，如测量身高、体重、心跳、血压、血液、体温等。如果上述因素都合格，那么表明这人是健康的。如果某个因素不合格，则表明此人在某个方面不健康，医生会对症下药。同理，我们也可以通过考察软件的质量属性来评价软件的质量，并给出提高软件质量的方法。

一提起软件的质量属性，人们首先想到的是“正确性”。“正确性”的确很重要，但运行正确的软件就是高质量的软件吗？不见得。

这个软件也许运行速度很低，并且浪费内存，甚至代码写得一塌糊涂，除了开发者本人谁也看不懂，也不会使用。可见正确性只是反映软件质量的一个因素而已。

不贪污的官就是好官吗？不见得。

时下老百姓对一些腐败的地方政府深恶痛绝，对“官”不再有质量期望。只要当官的不贪污，哪怕毫无政绩也算是“好官”了。过去有一个县官，在上级面前标榜自己的清廉：“我下去视察最多只喝老百姓的一碗水”，上级回敬他：“那我不如放



一个泥胎在那儿，它连一碗水都不会喝！”

相比之下，搞软件开发是够幸福的了。因为我们能通过努力，由自己来把握软件的质量。让我们好好珍惜权利，不要轻易放弃提高软件质量的机会。

软件的质量属性很多，如正确性、精确性、健壮性、可靠性、容错性、性能、易用性、安全性、可扩展性、可复用性、兼容性、可移植性、可测试性、可维护性、灵活性等。还可以列出十几个，新词可谓层出不穷。

上述这些质量属性“你中有我，我中有他”，非常缠绵。如果开发人员每天要面对那么多的质量属性咬文嚼字，不久就会迂腐得像孔乙己，因此我们有必要对质量属性做些分类和整合。质量属性可分为两大类：“功能性”与“非功能性”，后者有时也称为“能力”（Capability）。

从实用角度出发，本章将重点论述“10大”质量属性，如表1-1所示。

其中，功能性质量属性有3个：正确性、健壮性和可靠性；非功能性质量属性有7个：性能、易用性、清晰性、安全性、可扩展性、兼容性和可移植性。

表1-1 “10大”软件质量属性

功 能 性	正确性（Correctness）
	健壮性（Robustness）
	可靠性（Reliability）
非 功 能 性	性能（Performance）
	易用性（Usability）
	清晰性（Clarity）
	安全性（Security）
	可扩展性（Extendibility）
	兼容性（Compatibility）
	可移植性（Portability）

为什么碰巧是“10大”呢？

不为什么，只是方便记忆而已（如同国际国内经常评“10大”那样）。

为什么“10大”里面不包括可测试性、可维护性、灵活性呢？它们不也是很重要的吗？

它们是很重要，但不是软件产品的卖点，所以挤不进“10大”行列。我认为如果做好了前述“10大”质量属性，软件将会自然而然地具备良好的可测试性、可维护性。人们很少纯粹地去提高可测试性和可维护性，勿要颠倒因果。至于灵活性，它有益处也有坏处，该灵活的地方已经被其他属性覆盖，而不该灵活的地方就不要刻意去追求。

根据经验，如果你想一股脑儿地把任何事情都做好，结果通常是什么都做不好，做事总是要分主次的。什么是重要的质量属性应当视具体产品的特征和应用环境而定，请读者不要受本书观点的限制。最简单的判别方式就是考察该质量属性是否被用户关注（即卖点）。

## 1.1.2 提高软件质量的基本方法

质量的死对头是缺陷，缺陷是混在产品中的人们不喜欢、不想要的东西。缺陷越多质量越低，缺陷越少质量越高。

Bug 是缺陷的形象比喻，人们喜欢说 Bug 是因为可以把 Bug 当做“替罪羊”。软件的缺陷明明是人造成的，有了 Bug 这个词后就可以把责任推给 Bug——“都是 Bug 惹的祸”。唉，当一只 Bug 真是太冤枉了！

软件存在缺陷吗？是的，有以下典故为证。

---

编程大师说：“任何一个程序，无论它多么小，总存在着错误。”

初学者不相信大师的话，他问：“如果有个程序小得只执行一个简单的功能，那会怎么样？”

“这样的程序没有意义，”大师说，“但如果这样的程序存在的话，操作系统最后将失效并产生错误。”

但初学者不满足，他问：“如果操作系统不失效，那会怎么样？”

“没有不失效的操作系统，”大师说，“但如果这样的操作系统存在的话，硬件最后将失效并产生错误。”

初学者仍不满足，继续问：“如果硬件也不失效，那会怎么样？”

大师长叹一声道：“没有不失效的硬件。但如果这样的硬件存在的话，用户就会想让那个程序做一件不同的事，这件事也是个错误。”

没有错误的程序世间难求。（摘自《编程之道》）

---

错误是严重的缺陷。医生犯的错误最终会被埋葬在地下，从此一了百了。但软件的错误不会自动消失，它会一直骚扰用户。据统计，对于大多数的软件产品而言，用于测试与改错的工作量和成本将占整个软件开发周期的 30%，这是巨大的浪费。如果不懂得如何有效地提高软件质量，项目会付出很高的代价，你（开发人员）不仅没有功劳，也没人欣赏你的苦劳，你拥有最多的将只是疲劳。

怎样才能提高软件的质量呢？

还是先来听一个中国郎中治病的故事吧！

---

在中国古代，有一家三兄弟全是郎中。其中有一人是名医，人们问他：“你们兄弟三人谁的医术最高？”

他回答说：“我常用猛药给病危者医治，偶尔有些病危者被我救活，于是我的医术远近闻名并成了名医。我二哥通常在人们刚刚生病的时候马上就治愈他们，临近村庄的人都知道他的医术。我大哥深知人们生病的原因，所以能够预防家人生病，他的医术只有我们家里才知道。”

---

提高软件质量的基本手段是消除软件缺陷。与上述三个郎中治病很相似，消除软件缺陷也有三种基本方式：

（1）在开发过程中有效地防止工作成果产生缺陷，将高质量内建于开发过程之中。这就是“预防胜于治疗”的道理，无疑是最佳方式，但是要求开发人员必须懂



得正确地做事(台阶比较高)。我们学习“高质量编程”的目的就是要在干活的时候一次性编写出高质量的程序,而不是在程序出错后才去修补。

(2) 当工作成果刚刚产生时马上进行质量检查,及时找出并消除工作成果中的缺陷。这种方式效果比较好,人们一般都能学会。最常用的方法是技术评审、测试和质量保证等(详见本章1.4节),已经被企业广泛采用并取得了成效。

(3) 当软件交付给用户后,用着用着就出错了,赶紧请开发者来补救,这种方式的代价最高。可笑的是,当软件系统在用户那里出故障了,那些现场补救成功的人倒成了英雄,好心用户甚至还寄来感谢信。☺

## 1.1.3 “零缺陷”理念

质量的最高境界是什么?是尽善尽美,即“零缺陷”。

“零缺陷”理念来源于国际上一些著名的硬件厂商。尽管软件的开发与硬件生产有很大的区别,但我们仍可以借鉴,从中得到启迪。

人在做一件事情时,由于存在很多不确定的因素,一般不可能100%地达到目标。假设平常人做事能完成目标的80%。如果某个人的目标是100分,那么他最终成绩可达80分;如果某个人的目标只是60分,那么他最终成绩只有48分。我们在考场上身经百战,很清楚那些只想混及格的学生通常都不会及格。即使学习好的学生也常有失误,因而捶胸顿足。

做一个项目通常需要多个人的协作。假设某系统的总质量是10个开发人员的工作质量之积,即最高值为1.0,最低值为0。如果每个人的质量目标是0.95,那么10个人的累积质量不会超过0.598。如果每个人的质量目标是0.9,那么10个人的累积质量不会超过0.35。只有每个人都做到1.0,系统总质量才会是1.0。只要其中一人的工作质量是0,那么系统总质量也就成了0。因系统之中的一个缺陷而导致机毁人亡的事件已不罕见。

上述比喻虽然严厉了一些,但从严要求只有好处没有坏处。如果不严以律己,人的堕落就会很快。如果没有“零缺陷”的质量理念,也许缺陷就会成堆。

从理念到行动还是有一定距离的,企业在开发产品时应当根据自身实力和用户的期望值来设定可以实现的质量目标。过低的质量目标会毁坏企业的声誉,而过高的质量目标则有可能导致成本过高而拖累企业(请参见本章1.3节)。

## 1.2 细说软件质量属性

### 1.2.1 正确性

正确性是指软件按照需求正确执行任务的能力。这里“正确性”的语义涵盖了“精确性”。正确性无疑是第一重要的软件质量属性。如果软件运行不正确,将会给用户造成不便甚至损失。技术评审和测试的第一关都是检查工作成果的正确性。

正确性说起来容易做起来难。因为从“需求开发”到“系统设计”再到“实现”,

任何一个环节出现差错都会降低正确性。机器不会主动欺骗人，软件运行出错通常都是人造成的，所以不要找借口埋怨机器有毛病。开发任何软件，开发者都要为“正确”两字竭尽全力。

### 1.2.2 健壮性

健壮性是指在异常情况下，软件能够正常运行的能力。正确性与健壮性的区别是：前者描述软件在需求范围之内行为，而后者描述软件在需求范围之外的行为。可是正常情况与异常情况并不容易区分，开发者往往要么没想到异常情况，要么把异常情况错当成正常情况而不做处理，结果降低了健壮性。用户才不管正确性与健壮性的区别，反正软件出了差错都是开发方的错。所以提高软件的健壮性也是开发者的义务。

健壮性有两层含义：一是容错能力，二是恢复能力。

容错是指发生异常情况时系统不出错误的能力，对于应用于航空航天、武器、金融等领域的这类高风险系统，容错设计非常重要。

容错是非常健壮的意思，比如 UNIX 的容错能力很强，很难使系统出问题。而恢复则是指软件发生错误后（不论死活）重新运行时，能否恢复到没有发生错误的状态的能力。

从语义上理解，恢复不及容错那么健壮。

例如，某人挨了坏蛋一顿拳脚，特别健壮的人一点事都没有，表示有容错能力；比较健壮的人，虽然被打倒在地，过了一会还能爬起来，除了皮肉之痛外倒也不用去医院，表示恢复能力比较强；而虚弱的人可能短期恢复不过来，得在病床上躺很久。

恢复能力是很有价值的。Microsoft 公司早期的窗口系统如 Windows 3.x 和 Windows 9x，动不动就死机，其容错性的确比较差。但它们的恢复能力还不错，机器重新启动后一般都能正常运行，看在这个份上，人们也愿意将就着用。

### 1.2.3 可靠性

可靠性不同于正确性和健壮性，软件可靠性问题通常是由于设计中没有料到的异常和测试中没有暴露的代码缺陷引起的。可靠性是一个与时间相关的属性，指的是在一定环境下，在一定的时间段内，程序不出现故障的概率，因此是一个统计量，通常用平均无故障时间（MTTF, mean-time to fault）来衡量。

可靠性本来是硬件领域的术语。比如某个电子设备在刚开始工作时挺好的，但由于器件在工作中其物理性质会发生变化（如发热、老化等），慢慢地系统的功能或性能就会失常。所以一个从设计到生产完全正确的硬件系统，在工作中未必就是可靠的。人们有时把可靠性叫做稳定性。

软件在运行时不会发生物理性质的变化，人们常以为如果软件的某个功能是正确的，那么它一辈子都是正确的。可是我们无法对软件进行彻底的测试，无法根除软件中潜在的错误。平时软件运行得好好的，说不准哪一天就不正常了，如有千年

等一回的“千年虫”问题、司空见惯的“内存泄露”问题、“误差累积”问题，等等。因此把可靠性引入软件领域是很有意义的。

软件可靠性分析通常采用统计方法，遗憾的是目前可供第一线开发人员使用的成果很少见，大多数文章限于理论研究。我曾买了一本关于软件可靠性的著作，此书充满了数学公式，我实在难以看懂，更不知道怎样应用。请宽恕我的愚昧，我把此“天书”给“供养”起来，没敢用笔画一处记号。

口语中的可靠性含义宽泛，几乎囊括了正确性、健壮性。只要人们发现系统有毛病，便归结为可靠性差。从专业角度讲，这种说法是不对的，可是我们并不能要求所有的人都准确地把握质量属性的含义。

有必要搞清楚“故障”和“错误”这两个容易混淆的概念。

在《现代英汉词典》里，“故障（Fault）”一词的定义是：使设备、部件或元件不能按所要求的方式运行的一种意外情况，可能是物理的也可能是逻辑的。

那些潜伏在代码中的错误往往是不明显的，之所以在测试的时候没有暴露，是因为测试时的环境和条件不足以使之暴露，更何况我们无法对代码进行最彻底的测试。由此可见，故障是在经过日积月累，满足了一定的条件之后才出现的。例如“千年虫”问题，“内存泄漏（吃内存）”导致内存耗尽，“误差累积”导致计算错误进而导致连锁反应，“性能开销累积”导致性能显著下降，等等。因此，故障通常都是不可预料的、灾难性的。

“错误”的含义要广泛得多，例如语法错误、语义错误、文件打开失败、动态存储分配失败等。一般说来，程序错误是可以预料的，因此可以预设错误处理程序，运行时这些错误一旦发生，就可以调用错误处理程序把它干掉，程序还可以继续运行。因此，错误的结果一般来说不是灾难性的。

## 1.2.4 性能

性能通常是指软件的“时间—空间”效率，而不仅是指软件的运行速度。人们总希望软件的运行速度快些，并且占用资源少些。旧社会地主就是这么对待长工的：干活要快点，吃得要少点。

程序员可以通过优化数据结构、算法和代码来提高软件的性能。算法复杂度分析是很好的方法，可以达到“未卜先知”的功效。

性能优化的目标是“既要马儿跑得快，又要马儿吃得少”，关键任务是找出限制性能的“瓶颈”，不要在无关痛痒的地方瞎忙乎。例如在大学里当教师，光靠卖力气地讲课或者埋头做实验，职称是升不快的。有些人找到了突破口，一年之内“造”它几十篇文章，争取破格升副教授、教授。在学术上走捷径，这类“学者”的质量真让人担忧。

性能优化就好像从海绵里挤水一样，你不挤，水就不出来，你越挤海绵越干。有些程序员认为现在的计算机不仅速度越来越快，而且内存越来越大，因此软件性能优化的必要性下降了。这种看法是不对的，殊不知随着机器的升级，软件系统也越来越庞大和复杂了，性能优化仍然大有必要。最具有代表性的是三维游戏软件，例如《Delta Force》、《古墓丽影》、《反恐精英》等，如果不对软件（关键是游戏引擎）



做精益求精的优化，要想在一台普通的 PC 上顺畅地玩游戏是不太可能的。

### 1.2.5 易用性

易用性是指用户使用软件的容易程度。现代人的生活节奏快，干什么事都想图个方便，所以把易用性作为重要的质量属性无可非议。

导致软件易用性差的根本原因是开发人员犯了“错位”的毛病：他以为只要自己用起来方便，用户也一定会满意。俗话说“王婆卖瓜，自卖自夸”。当开发人员向用户展示软件时，常会得意地讲：“这个软件非常好用，我操作给你看，……是很好用吧！”

软件的易用性要让用户来评价。如果用户觉得软件很难用，开发人员不要有逆反心理：哪里找来的笨蛋！

其实不是用户笨，是自己开发的软件太笨了。当用户真的感到软件很好用时，一股温暖的感觉就会油然而生，于是就会用“界面友好”、“方便易用”等词来夸奖软件的易用性。

### 1.2.6 清晰性

清晰意味着工作成果易读、易理解，这个质量属性表达了人们的一种质朴的愿望：让我花钱买它或者用它，总得让我看明白它是什么东西。我小时候的一个伙伴在读中学时就因搞不明白电荷为什么还要分“正”和“负”，觉得很烦恼，便早早地辍学当了工人。

开发人员只有在自己思路清晰的时候才可能写出让别人易读、易理解的程序和文档。可理解的东西通常是简洁的。一个原始问题可能很复杂，但高水平的人就能够把软件系统设计得很简洁。如果软件系统臃肿不堪，它迟早会出问题。所以简洁是人们对工作“精益求精”的结果，而不是潦草应付的结果。

在生活中，与简洁对立的是“里 唆”。废话大师有句名言：“如果我令你过于轻松地明白了，那你一定是误解了我的意思。”中国小说中最“婆婆妈妈”的男人是唐僧。有一项民意调查：如果世上只有唐僧、孙悟空、猪八戒和沙僧这四类男人，你要嫁给哪一类？请列出优先级。调查结果表明，现代女性毫不例外地把唐僧摆在最后。

很多人在读研究生时有一种奇怪的体会：如果把文章写得很简洁，让人很容易理解，投稿往往中不了；只有加上一些玄乎的东西，把本来简单的东西弄成复杂的，才会增加投稿的命中率。虽然靠这种做法能把文凭混到手，可千万不要把此“经验”应用到产品的开发中！

### 1.2.7 安全性

这里的安全性是指信息安全，英文是 Security 而不是 Safety。安全性是指防止系统被非法入侵的能力，既属于技术问题又属于管理问题。信息安全是一门比较深奥的学问，其发展是建立在正义与邪恶的斗争之上的。这世界似乎不存在绝对安全的

系统，连美国军方的系统都频频遭黑客入侵。如今全球黑客泛滥，真是“道高一尺，魔高一丈”啊！

对于大多数软件产品而言，杜绝非法入侵既不可能也没有必要。因为开发商和客户愿意为提高安全性而投入的资金是有限的，他们要考虑值不值得。究竟什么样的安全性是令人满意的呢？

一般地，如果黑客为非法入侵花费的代价（考虑时间、费用、风险等多种因素）高于得到的好处，那么这样的系统就可以认为是安全的。

## 1.2.8 可扩展性

可扩展性反映了软件适应“变化”的能力。在软件开发过程中，“变化”是司空见惯的事情，如需求、设计的变化，算法的改进、程序的变化等。

由于软件是“软”的，是否它天生就容易修改以适应“变化”？

关键要看软件的规模和复杂性。

如果软件规模很小，问题很简单，那么修改起来的确比较容易，这时就无所谓“可扩展性”了。要是软件的代码只有 100 行，那么“软件工程”也就用不着了。

如果软件规模很大，问题很复杂，倘若软件的可扩展性不好，那么该软件就像用卡片造成的房子，抽出或者塞进去一张卡片都有可能使房子倒塌。可扩展性是系统设计阶段重点考虑的质量属性。

## 1.2.9 兼容性

兼容性是指两个或两个以上的软件相互交换信息的能力。由于软件不是在“真空”里应用的，它需要具备与其他软件交互的能力。例如两个字处理软件的文件格式兼容，那么它们都可以操作对方的文件，这种能力对用户很有好处。国内金山公司开发的字处理软件 WPS 就可以操作 Word 文件。

兼容性的商业规则是：弱者设法与强者兼容，否则无容身之地；强者应当避免被兼容，否则市场将被瓜分。如果你经常看香港拍的“黑帮”影片，你就很容易明白这个道理。所以 WPS 一定要与 Word 兼容，否则活不下去。但是 Word 绝对不会与 WPS 兼容，除非 WPS 在中国称老大。

## 1.2.10 可移植性

软件的可移植性指的是软件不经修改或稍加修改就可以运行于不同软硬件环境（CPU、OS 和编译器）的能力，主要体现为代码的可移植性。编程语言越低级，用它编写的程序越难移植，反之则越容易。这是因为，不同的硬件体系结构（例如 Intel CPU 和 SPARC CPU）使用不同的指令集和字长，而 OS 和编译器可以屏蔽这种差异，所以高级语言的可移植性更好。

C++/C 是一种中级语言，因为它具有灵活的“位操作”能力（因此具有硬件操作能力），而且可以直接嵌入汇编代码。但是 C++/C 并不依赖于特定的硬件，因此比汇编语言可移植性好。

Java 是一种高级语言，Java 程序号称“一次编译，到处运行”，具有 100%的可

移植性。为了提高 Java 程序的性能，最新的 Java 标准允许人们使用一些与平台相关的优化技术，这样优化后的 Java 程序虽然不能“一次编译，到处运行”，仍然能够“一次编程，到处编译”。

一般地，软件设计时应该将“设备相关程序”与“设备无关程序”分开，将“功能模块”与“用户界面”分开，这样可以提高可移植性。

## 1.3 人们关注的不仅仅是质量

企业开发产品的目的是赚钱，为了使利润最大化，人们希望软件开发工作“做得好、做得快，并且少花钱”。用软件工程的术语来讲，即“提高质量、提高生产率，并且降低成本”。古代哲学家曾为“鱼与熊掌不可得兼”的问题费尽心思，我们现在却梦想鱼、熊掌、美酒三者兼得，现代人的欲望真是无止境啊。

让我们先谈谈质量、生产率和成本之间的关系。

### 1.3.1 质量、生产率和成本之间的关系

质量无疑是客户最关心的问题。客户即使不图物美价廉，也要求货真价实。软件开发商必须满足客户对质量的要求（不论是写在合同上的还是约定俗成的），否则做不成买卖。现在就连做盗版光盘生意的人也讲究质量，如果盘片不好，是可以退货的。高质量既是软件开发人员的技术追求，又是职业道德的要求。

在关注质量的同时，软件开发商又期望生产率能高些，并且成本能低些。老板和员工们谁不想用更少的时间赚更多的钱！

质量与生产率之间存在相辅相成的关系。高生产率必须以质量合格为前提。如果质量不合格，软件产品要么卖不出去，要么卖出去了再赔偿客户的损失。这种情况下“高生产率”变得毫无意义。别看开发商和客户双方的代表能在餐桌上谈笑风生，一旦出了质量问题，那就不会很亲热了。从短期效益看，追求高质量可能会延长软件开发时间，一定程度上降低了生产率。从长期效益看，追求高质量将使软件开发过程更加成熟和规范化。日积月累，当开发过程成熟到一定程度后，必将大大降低软件的测试和改错的代价，缩短产品的开发周期，实质上是提高了生产率，同时又获得了很好的信誉。所以质量与生产率之间不存在根本的对立。

提高质量与生产率需要一个过程，企业不可操之过急。一般地，软件过程能力比较低的企业（例如低于 CMM 2 级），应该将质量放在第一位，生产率放在第二位，只有这样才可能持久地提高质量和生产率（“能力成熟度模型 CMM”将在 1.4.2 节介绍）。

如果一个企业的软件过程能力低于 CMM 2 级，表明其开发能力与管理能力还很薄弱。就其目前的实力而言，无论下多大的决心去做，都不可能一开始就把质量与生产率改善得一样好。并不是我们刻意贬低生产率的“地位”，是公司的现实情况要求在质量与生产率之间分个“轻重缓急”。由于人们天生就有“急功近利”的倾向，



如果公司领导人认可“生产率第一、质量第二”，那么员工们做着做着必定会回到混乱的局面。这样的教训实在是太多了！老话说得好：磨刀不误砍柴工，用它类比上述理念最合适不过了。

俗话说“一分钱一分货”，人们买东西的时候大多认可“质量越好价格就越高”。除了垄断性的产品外，一般来说成本是影响价格的主要因素。

对于软件开发而言，质量与成本之间有什么关系？高质量必然会导致高成本吗？

经验表明，如果软件的“高质量”是“修补”出来的，毫无疑问会导致低生产率和高成本。如果能研制出某些好方法，将高质量与高生产率内建于开发过程之中，那么就能自然地降低开发成本，这是软件过程改进的目标。

要提醒大家的是，大公司与小公司对成本的关注程度是不尽相同的。

首先谈一下“市场价”(Marketing Price)与“成本价”(Cost Price)的概念。在某个领域，当市场上只出现尚未形成竞争格局的一个或几个产品时，产品价格基本上是由厂商自己制定的，这里称其为“市场价”。由于缺乏竞争，无论成本多高，产品总能获得高额利润。电影《大腕》里那个搞房地产的精神病人说“不求最好，但求最贵”，真是实话实说。

当产品之间形成竞争时，就会出现“杀价”现象。由于各家产品的功能、质量旗鼓相当，竞争实质上是在拼成本。谁的成本低，谁就有利可图。这时的产品价格称为“成本价”。

---

中国的彩电业是一个活生生的例子。若干年前彩电价格极高，彩电远离寻常百姓，一部分人即使买得起也买不到。如今连超市里都充斥着各种品牌的彩电，价格战打得呜呼哀哉，把厂商逼到“微利”的地步。然而商场里TOTO品牌的马桶价格却在2000~3000元，比同体积的国产纯平彩电还贵，并且利润高得多。唉，我们坐在这样的马桶上真的要为民族工业忧心忡忡啊！

---

由于“市场价”与“成本价”的差价悬殊，IT行业的大公司都想吃“市场价”这块肥肉。大公司的资金雄厚，销售力量强，只要能抢先推出产品，就不愁卖不出去。怎样才能达到目的呢？通常有两种方式。

一种方式是从别处购买快要成型的产品，改头换面，贴上大公司的标签就可以上市销售。所以IT行业的“公司收购”特别盛行。如果Cisco公司的网络产品全部让原班人马来开发，它很难那么快就发展成为网络业的霸主！

另一种方式是自行开发新产品，让公司的研发队伍加班加点地干活。这么辛苦是值得的，产品成功会让员工们有很大的成就感。

无论通过哪种方式抢先推出产品，前提条件都要求产品的质量合格。如果产品因质量不合格而被市场拒绝，那么损失的不仅仅是成本，更惨重的是失去机会和信誉。像Intel这样了不起的公司也会吃败仗。每当Intel公司的CPU芯片出现缺陷时，就会骂声一片。Intel公司不得不大量回收芯片并向用户道歉，此时竞争对手如AMD公司就会乘虚而入，抢走像IBM、Compaq这些大客户的部分订单。

在信息高度发达的社会里，你能想得到的产品别人也能想得到。只有少数大公司能够享受到“市场价”的利益，但是好景不会太长。大多数公司在大部分时间里开发的是“成本价”的产品。所以树立“降低开发成本”的理念仍然十分重要。

### 1.3.2 软件过程改进的基本概念

在 20 世纪 70~80 年代，软件工程的研究重点是需求分析、系统设计、编程、测试、维护等领域的方法、技术和工具，我们称为经典软件工程。

现代的软件技术、软件开发工具比几十年前不知好多少倍，而且几乎所有的开发人员都学习过分析、设计、编程、测试等技能，可是如今绝大多数软件项目依然面临着质量低下、进度延误、费用超支这些老问题。

人们逐渐意识到，由于机构管理软件过程的能力比较弱，常常导致项目处于混乱状态，过程的混乱使得新技术、新工具的优势难以体现。经典的软件工程不是不好，而是不够用。从 20 世纪 90 年代至今，软件过程改进成为软件工程学科的一个主流研究方向，其中 CMM 和 CMMI 是该领域举世瞩目的重大成果。

首先解释什么是过程。过程就是人们使用相应的方法、规程、技术、工具等将原始材料（输入）转化为用户需要的产品（输出）。过程的 3 个基本要素是：人、方法与规程、技术与工具，如图 1-1 所示。可以把过程比喻为 3 条腿的桌子，要使桌子平稳，这 3 条腿必须协调好。

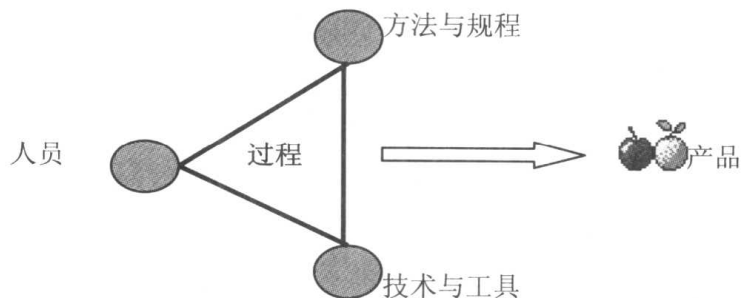


图 1-1 过程的要素

从图 1-1 可知，过程与产品存在因果关系。即好的过程才能得到好的产品，而差的过程只会得到差的产品。这个道理很朴实，但是很多人并未理解或者理解了却不执行。毕竟我们销售的是产品，而非过程。人们常常只把眼光盯在产品上，而忘了过程的重要性。例如，领导对员工们下达命令时总强调：“我不管你们如何做，反正时间一到，你们就得交付产品。”其实这是一句因果关系颠倒了的话，却在业界普遍存在。

在过程混乱的企业里，一批人马累死累活地做完产品后，马上又因质量问题被折腾得焦头烂额。这种现象反反复复地发生，让人疲惫不堪。怎么办？长痛不如短痛，应该下决心，舍得花精力与金钱去改进软件过程能力。

CMM (Capability Maturity Model) 是用于衡量软件过程能力的事实上的标准，同时也是目前软件过程改进最好的参考标准。CMM 是由美国卡内基—梅隆大学

(Carnegie-Mellon) 软件工程研究所 (Software Engineering Institute, SEI) 研制的, 其发展简史如下:

- ◇ CMM 1.0 于 1991 年制定。
- ◇ CMM 1.1 于 1993 年发布, 该版本应用最广泛。
- ◇ CMM 2.0 草案于 1997 年制定 (未广泛应用)。
- ◇ 到 2000 年, CMM 演化成为 CMMI (Capability Maturity Model Integration), CMM 2.0 成为 CMMI 1.0 的主要组成部分。
- ◇ CMMI-SE/SW 1.1 (CMMI for System Engineering and Software Engineering) 于 2002 年 1 月正式推出。

CMM 将软件过程能力分为 5 个级别, 最低为 1 级, 最高为 5 级。目前国内只有几家 IT 企业达到了 CMM 2 级或 CMM 3 级。鉴于 CMM 已经被美国、印度软件业广为采纳, 并且取得了卓著成效, 近两年来国内兴起了 CMM 热潮。CMM 受欢迎的程度远远超过了 ISO 同类标准。

国内 IT 企业采用 CMM 的目的大体有两种:

- (1) 主要想提高企业的软件过程能力, 但并不关心 CMM 评估。
- (2) 既要提高企业的软件过程能力, 又想通过 CMM 评估来提升企业的威望与知名度。

出于第一种考虑的企业占绝大多数, 它们主要是一些中小型 IT 企业。出于第二种考虑的一般是实力雄厚的大型 IT 企业。无论是哪类 IT 企业, 它们在实施 CMM 时遇到的共性问题都是“费用高、难度大、见效慢”。

企业做一次比较完整的 CMM 2~3 级咨询和评估大约要花费 60 万~100 万元。然而 CMM 咨询师只能起到“参谋”的作用, 解决实际问题还得靠自己。企业要组建软件工程过程小组 (Software Engineering Process Group, SEPG) 专门从事 CMM 研究与推广工作, SEPG 的成本并不比咨询费低。如果企业再购买一些昂贵的软件工程工具 (例如 Rational 的产品), 那么总成本会更高。

即使企业舍得花钱, 也不意味着就能够轻易地提高软件过程能力。目前国内通过 CMM 2-3 级评估的企业屈指可数, 而这些企业的实际能力也没有宣传的那么好。因为参加 CMM 评估的项目都是精心准备的, 个别项目或者事业部通过了 CMM 评估并不意味着整个企业达到了那个水平, 这里面的水分相当大。

曾经有一段时间, IT 人士经常争论“CMM 好不好”、“值不值得推广 CMM”等话题。现在业界关注的焦点则是“企业如何以比较低的代价有效地提高软件过程能力”, 攻克这个难题必将产生巨大的经济效益和社会效益。

一般地, 为了真正提高软件过程能力, 企业至少要做 3 件最重要的事情:

- (1) 制定适合于本企业的软件过程规范;
- (2) 对员工们进行培训, 指导他们依据规范来开发产品;
- (3) 购买或者开发一些软件工程和项目管理工具, 提高员工们的工作效率。

本书作者和合作者根据上述需求, 研制了一套“软件过程改进解决方案” (Software Process Improvement Solution, SPIS)。SPIS 的主要组成部分有:

- ◇ 基于 CMMI 3 级的软件过程改进方法与规范, 命名为“精简并行过程” (SPP)



(电子工业出版社已于 2003 年 1 月出版公布了 SPP, 详见 <http://www.chinaspis.com>)。

- ◇ 一系列培训教材, 包括软件工程、项目管理、高质量 C++/Java 编程等, 本书即为其中之一。
- ◇ 基于 Web 的集成化项目管理工具, 包括项目规划、项目监控、质量管理、配置管理、需求管理等功能, 命名为 Future。

## 1.4 高质量软件开发的基本方法

### 1.4.1 建立软件过程规范

人们意识到, 若想顺利开发出高质量的软件产品, 必须有条理地组织技术开发活动和项目管理活动。我们把这些活动的组织形式称为**过程模型**。软件企业应当根据产品的特征, 建立一整套在企业范围内通用的软件过程模型及规范, 并形成制度, 这样开发人员与管理人员就可以依照过程规范有条不紊地开展工作。

我们曾与国内很多研发人员和各级经理交流过, 大家都对软件开发的混乱局面表示了不满和无奈。尽管“游击队”的开发模式到处可见, 但是没有人真的喜欢混乱。“规范化”是区别“正规军”和“游击队”的根本标志, 大家无不渴望以规范化的方式开发产品, 这是现状, 是需求, 也是希望。

对软件开发模型的研究兴起于 20 世纪 60 年代末 70 年代初, 典型成果是 1970 年提出的瀑布模型。人们研制了很多的软件开发模型, 常见的还有“喷泉模型”、“增量模型”、“快速原型模型”、“螺旋模型”、“迭代模型”等。

这么多软件开发模型, 企业应该如何选择并应用呢?

企业在选择软件开发模型时, 不要太在乎学术上的“先进”与“落后”, 正如有才华的人并不一定要出自名牌大学或拥有高学历那样。关键是看该模型能否有效地帮助企业顺利地开发出软件产品, 并且要考虑员工们使用起来是否方便。简而言之, 就是考察模型是否“实用、好用”。

最早出现的软件开发模型是瀑布模型, 它太理想化、太单纯, 看起来已经落后于现代的软件开发模式。如今瀑布模型几乎被学术界抛弃, 偶尔被人提起, 都属于被贬的对象, 未留下一丝惋惜。说它如何如何地差, 为的是说明新模型是怎样怎样地好。

然而企业界不同于学术界。我认为瀑布模型对企业太有价值了, 我要为它声辩, 恢复它应有的名誉。瀑布模型的精髓是“线性顺序”地开发软件。我们应该认识到“线性化”是人们最容易掌握并能熟练应用的思想方法。当人们碰到一个复杂的“非线性”问题时, 总是千方百计地将其分解或转化为一系列简单的线性问题, 然后逐个解决。一个软件系统的整体可能是复杂的, 而细分后的子程序总是简单的, 可以用“线性化”的方式来实现, 否则干活就太累了。

让我们引用爱因斯坦的话作为信条——“任何事物都应该尽可能地简洁”。“线性”

是一种简洁，简洁就是美。当我们领会了“线性”的精神，就不要再呆板地套用“线性”的外表，而应该用活它。例如，增量模型实质上就是分段的线性模型。螺旋模型则是迭代的弯曲了的线性模型。在其他模型中大都能够找到“线性”的影子。

瀑布模型是如此地简洁，所有的软件开发人员一学就会（如果学不会，那他就别干软件这一行了）。所以瀑布模型特别适合于企业，请大家别轻易地贬低它。

软件开发模型只关注技术开发活动，并不考虑项目管理，这对开发产品而言是不够的，所以开发模型只是软件过程模型的一部分。奇怪的是，迄今为止我尚未找到论述软件过程模型的软件工程书籍，我就自己创作了一个基于 CMMI 3 级的软件过程模型，称为“精简并行过程”（Simplified Parallel Process, SPP）。

SPP 模型如图 1-2 所示。“精简并行过程”的含义是：

- （1）对 CMMI 3 级以内的过程域及关键实践做了“精简”处理；
- （2）项目管理过程、技术开发过程和机构支撑过程“并行”开展。

SPP 模型把产品生命周期划分为 6 个阶段：

- ◇ 产品概念阶段，记为 PH0。
- ◇ 产品定义阶段，记为 PH1。
- ◇ 产品开发阶段，记为 PH2。
- ◇ 产品验证阶段，记为 PH3。
- ◇ 用户验收阶段，记为 PH4。
- ◇ 产品维护阶段，记为 PH5。

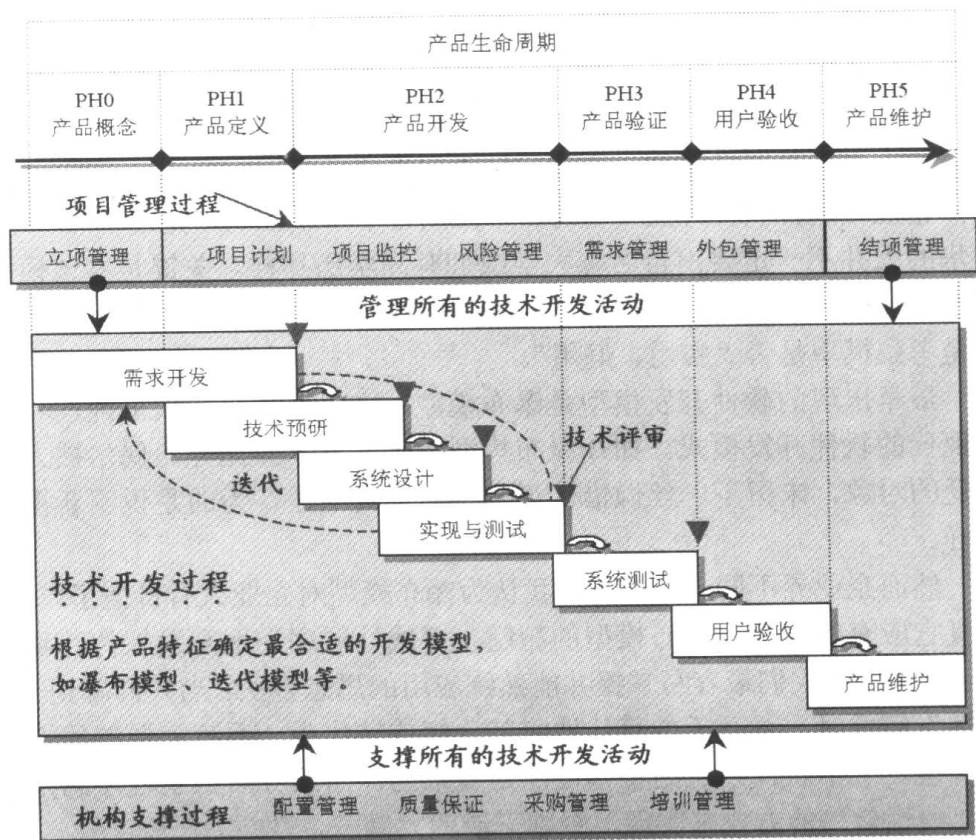


图 1-2 精简并行过程（SPP）模型

在 SPP 模型中，一个项目从 PH0 到 PH5 共经历 19 个过程域 (Process Area)，它们被划分为三大类过程，如表 1-2 所示。其中项目管理过程含 6 个过程域，技术开发过程含 8 个过程域，支撑过程含 5 个过程域。

表 1-2 SPP 过程域分类

过 程 类 别	项目管理过程	技术开发过程	支 撑 过 程
过 程 域	立项管理	需求开发	配置管理 质量保证 采购管理 外包管理 培训管理
	结项管理	技术预研	
	项目计划	系统设计	
	项目监控	实现与测试	
	风险管理	系统测试	
	需求管理	用户验收	
		产品维护	
		技术评审	

SPP 模型的主要优点有：

(1) 模型直观。SPP 模型是三层结构，上层是项目管理过程的集合，中层是技术开发过程的集合，下层是支撑过程的集合。这种模型很直观，高级经理、项目经理、开发人员、质量保证员等根据 SPP 模型很容易知道自己“应该在什么时候做什么事情，以及按照什么规范去做事情”。SPP 模型有助于使各个过程的活动有条不紊地开展。

(2) 便于用户裁剪 SPP 模型。项目管理过程和支撑过程对绝大多数软件产品开发而言都是适用的。需求开发、技术预研、系统设计、编程、测试、技术评审、维护都是技术开发过程中必不可少的环节，用户可以根据产品的特征确定最合适的开发模型（例如瀑布模型、快速原型模型、迭代模型等）。

(3) 便于用户扩充 SPP 模型。如果产品同时涉及软件、硬件开发的话，可将产品生命周期、软件开发过程和硬件开发过程集成起来。

#### 1.4.2 复用

复用就是指“利用现成的东西”，文人称其为“拿来主义”。被复用的对象可以是有形的物体，也可以是无形的知识成果。复用不是人类懒惰的表现，而是智慧的表现。因为人类总是在继承了前人的成果，不断加以利用、改进或创新后才会进步。所以当我们欢度国庆时，要清楚祖国远不止 50 来岁，我们今天享用到的财富还有历史上几千年来中国人民的贡献。

复用有利于提高质量，提高生产率，并降低成本。由经验可知，通常在一个新系统中，大部分的内容是成熟的，只有小部分内容是创新的。一般可以相信成熟的东西总是比较可靠的（即具有高质量），而大量成熟的工作可以通过复用来快速实现（即具有高生产率）。勤劳并且聪明的人们应该把大部分的时间用在小比例的创新工作上，而把小部分的时间用在大比例的成熟工作中，这样才能把工作做得又快又好。



把复用的思想用于软件开发,称为软件复用。技术开发活动与管理活动中的任何成果都可以被复用,如思想方法、经验、程序、文档,等等。据统计,世上已有一千多亿行程序,无数功能被重写了成千上万次,真是浪费啊!面向对象(Object Oriented)学者的口头禅就是“请不要再发明相同的车轮子了”。

具有一定集成度并可以重复使用的软件组成单元称为软构件(Software Component)。软件复用可以表述为:构造新的软件系统可以不必每次从零做起,直接使用已有的软构件,即可在组装或加以合理修改后成为新的系统。

复用方法合理化并简化了软件开发过程,减少了总的开发工作量与维护代价,既降低了软件的成本,又提高了生产率。另一方面,由于软构件是经过反复使用验证的,自身具有较高的质量,因此由软构件组成的新系统也具有较高的质量。

软件复用不仅要使自己拿来方便,还要让别人拿去方便,是“拿来拿去主义”。这想法挺好,但现实中执行得并不如意。企业的业务各式各样,谁也不能坐等着天上掉下可以大规模复用的东西,一般要靠“日积月累”方能建设可以复用的软件库。从理论上讲,这项工作没有不可逾越的技术障碍。真正的障碍是它“耗时费钱”,前期投入较多,缺乏近期效益。大部分的公司都注重近期效益,不是它天生目光短浅,而是为了生存必须这么做。有些处境艰难的公司下一顿饭还不知道在何处着落,更别提软件复用这样的“长久之计”了。所以软件复用对大多数公司来说不是“最高优先级”。

我到公司工作的最初安排是从事电信领域“可复用软件工厂”的开发。领导在招聘时跟我讲,这个想法已经有数年了,一直没有落实,希望我能做好。待我正式上班时,马上就改成做其他短期的研发项目了。要知道我所在的公司人力与财力相当充足,非国内普通中小型IT企业所能比。即便我们有如此好的条件,软件复用也只是挂在嘴上,没有实际行动。

所以我建议:随时随地尽可能地复用你能复用的东西,不要等待公司下达复用的行政命令,因为你很难等到那一天,即使等到了也没有多大的意义。

## Q 1.4.3 分而治之

分而治之是指把一个复杂的问题分解成若干个简单的问题,然后逐个解决。这种朴素的思想来源于人们的生活和工作经验,完全适合于技术领域。

分而治之说起来容易,做好却难,最糟糕的现象是“分是分了”,却“治不了”。

软件的分而治之不可以“硬分硬治”。不像为了吃一个西瓜或是一只鸡,挥刀斩成 $n$ 块,再把每块塞进嘴里粉碎搅拌,然后交由胃肠来消化吸收,象征复杂问题的西瓜或是鸡也就此消失了。

软件的“分而治之”应该着重考虑:复杂问题分解后,每个问题能否用程序实现?所有程序能否最终集成为一个软件系统并有效解决原始的复杂问题?图1-3表示了软件的“分而治之”策略。软件的模块化设计就是分而治之的具体表现。

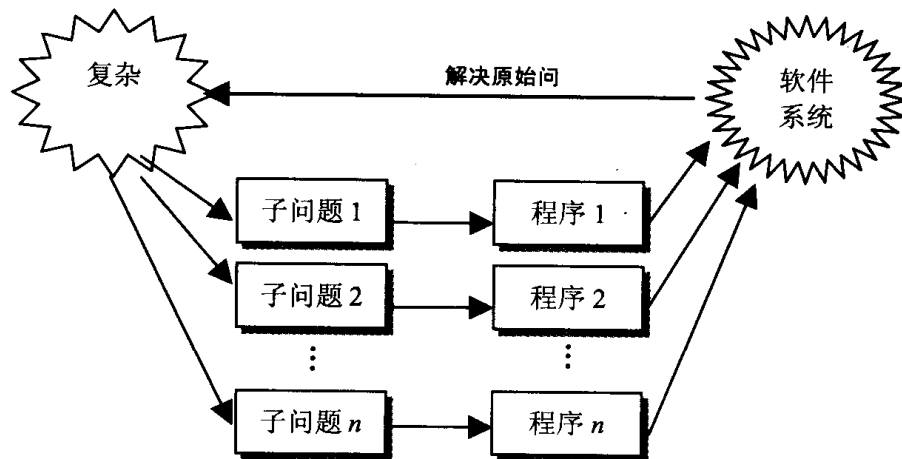


图 1-3 软件的分而治之策略

#### 1.4.4 优化与折中

软件的优化是指优化软件的各个质量属性，如提高运行速度，提高对内存资源的利用率，使用户界面更加友好，使三维图形的真实感更强，等等。想做好优化工作，首先要让开发人员都有正确的认识：优化工作不是可有可无的事情，而是必须做的事情。

当优化工作成为一种责任时，开发人员才会不断改进软件中的数据结构、算法和程序，从而提高软件质量。

著名的 3D 游戏软件 Quake，能够在 PC 机上实时地绘制具有高度真实感的复杂场景。Quake 的开发者能把很多成熟的图形技术发挥到极致，例如把 Bresenham 画线、多边形裁剪、树遍历等算法的速度提高近一个数量级。我的博士研究方向是计算机图形学，我第一次看到 Quake 时不仅感到震动，而且深受打击。这个 PC 游戏软件的技术水平已经远胜于我所见识到的国内领先的图形学相关科研成果。这对国内日益盛行的“点到为止”的学术研究真是莫大的讽刺！

所以当我们开发出来的软件表现出很多不可救药的病症时，不要埋怨机器差。真的都是我们自己没有把工作做好。写不好字不要嫌笔钝。

假设我们经过思想教育后，精神抖擞，随时准备为优化工作干上六天七夜时，要清醒地知道愿意做并不意味着就能把事情做好。优化工作的复杂之处是很多目标之间存在千丝万缕的关系，可谓“剪不断理还乱”。当不能够使所有的目标都得到优化时，就需要“折中”。

软件中的“折中”是指协调各个质量属性，实现整体质量的最优。就像某些当官者扮演的角色：“……为了使整个组织具有最好的战斗力，我们要重用几个人，照顾一些人，在万不得已的情况下委屈一批人”。

软件折中的重要原则是不能使某一方损失关键的功能，更不可以像“舍鱼而取熊掌”那样抛弃一方。例如 3D 动画软件的瓶颈通常是速度，但如果为了提高速度而在程序中取消光照明计算，那么场景就会丧失真实感，3D 动画也就不再有意义了。

人都有惰性，如果允许滥用折中的话，那么一旦碰到困难，人们就会用“拆东墙补西墙”的方式去折中，不再下苦功去做有意义的优化。所以我们有必要为折中制定严正的立场：在保证其他质量属性不差的前提下，使某些重要质量属性变得更好。

下面让我们用优化与折中的思想解决“鱼与熊掌不可得兼”的难题。

问题提出：假设鱼每千克 10 元，熊掌每千克 10 000 元。有个倔脾气的人只有 20 元钱，非得要吃上一公斤美妙的“熊掌烧鱼”，怎么办？

解决方案：花 9 元 9 角 9 分钱买 999 克鱼肉，花 10 元钱买 1 克熊掌肉，可做一道“熊掌戏鱼”菜。剩下的那一分钱还可建立基金，用于推广优化与折中方法。

## 1.4.5 技术评审

技术评审（Technical Review, TR）的目的是尽早地发现工作成果中的缺陷，并帮助开发人员及时消除缺陷，从而有效地提高产品的质量。

技术评审最初是由 IBM 公司为了提高软件质量和提高程序员生产率而倡导的。技术评审方法已经被业界广泛采用并收到了很好的效果，它被普遍认为是软件开发的最佳实践之一。技术评审能够在任何开发阶段执行，它可以比测试更早地发现并消除工作成果中的缺陷。技术评审的主要好处有：

- ◇ 通过消除工作成果的缺陷而提高产品的质量。
- ◇ 越早消除缺陷就越能降低开发成本。
- ◇ 开发人员能够及时地得到同行专家的帮助和指导，无疑会加深对工作成果的理解，更好地预防缺陷，在一定程度上能提高开发效率。

技术评审有两种基本类型：

- ◇ 正式技术评审（FTR）：FTR 比较严格，需要举行评审会议，参加评审会议的人员比较多。
- ◇ 非正式技术评审（ITR）：ITR 的形式比较灵活，通常在同伴之间开展，不必举行评审会议，评审人员比较少。

从理论上讲，为了确保产品的质量，产品的所有工作成果都应当接受技术评审。现实中为了节约时间，允许人们有选择地对工作成果进行技术评审。技术评审方式也视工作成果的重要性和复杂性而定。例如，将重要性、复杂性各分“高、中、低”3 个等级，重要性—复杂性的组合与技术评审方式的对应关系如表 1-3 所示。

表 1-3 工作成果重要性—复杂性组合与技术评审方式的对应关系

工作成果的重要性—复杂性组合	技术评审方式（FTR, ITR）
高高	FTR
高中	FTR
高低	FTR 或 ITR 均可
中中	FTR 或 ITR 均可
中低	ITR
低低	ITR

正式技术评审的一般流程如图 1-4 所示。

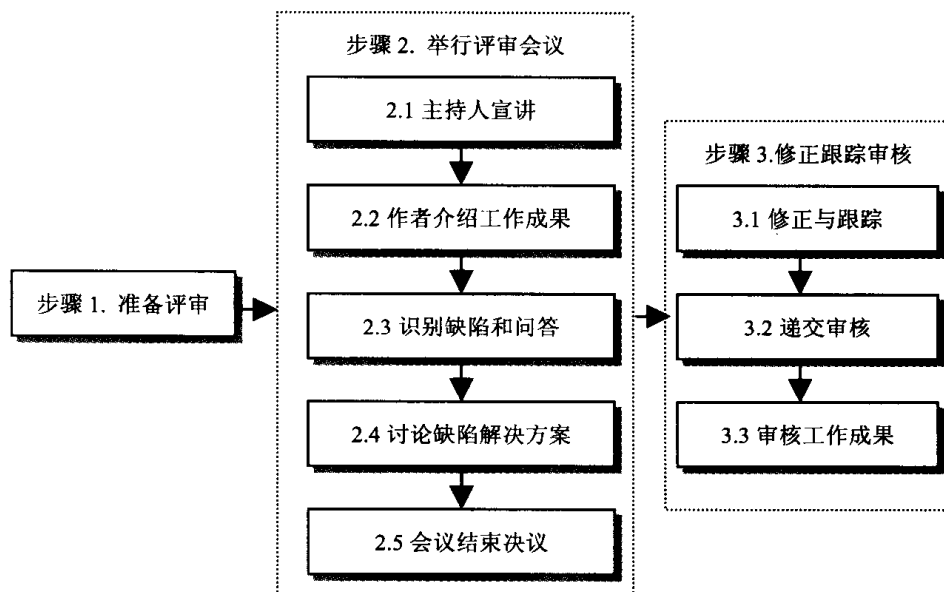


图 1-4 正式技术评审的一般流程

技术评审的注意事项：

- ✧ 评审人员的职责是发现工作成果中的缺陷，并帮助开发人员找到消除缺陷的办法，而不是替开发人员消除缺陷。
- ✧ 技术评审应当“就事论事”，不要打击有失误的开发人员的工作积极性，更不准搞人身攻击（如挖苦、讽刺等）。
- ✧ 在会议评审期间要限制过多的争论，以免浪费他人的时间。

对技术评审的一些建议：

- ✧ 对于重要性和复杂性都很高的工作成果，建议先在项目内部进行“非正式技术评审”，然后再进行“正式技术评审”。
- ✧ 技术评审应当与质量保证有机地结合起来，最好请质量保证人员参加并监督正式技术评审。
- ✧ 技术评审应当与配置管理有机地结合起来，例如规定：工作成果在成为基准（Baseline）之前必须先通过技术评审。
- ✧ 建议机构采用统一的缺陷跟踪工具，使得技术评审所发现的缺陷能够及时地消除，不致遗漏。

#### 1.4.6 测试

测试是通过运行测试用例（Test Case）来找出软件中的缺陷。测试与技术评审的主要区别是：前者要运行软件而后者不必运行软件（动态检查和静态检查）。

在软件开发过程中，编程和测试是紧密相关的技术活动，缺一不可。理论上讲两者不分贵贱，同等重要。但在大多数软件企业中，程序员的待遇普遍要高于专职的测试人员。即使不考虑待遇问题，大多数人认为开发工作比测试工作更有趣、有成就感、有前途。所以计算机专业人员通常会把编程当成一种看家本领，舍得下功



夫学习和钻研,但极少有人以这种态度对待软件测试。这种意识导致软件测试被过于轻视。不仅学生们在读书时懒得学习测试(目前国内高校似乎没有“软件测试”的课程),就连有数年工作经验的软件开发人员也未必懂得测试。

不少开发人员虽然不敌视测试工作,但有“临时抱佛脚”的坏习惯,往往事到临头才到处找测试资料,向人请教。经常有人向我要文档模板,用来对付测试。

你以为有了文档模板就懂得如何测试了么?

测试虽然并不深奥,但是学好并不容易。不懂得“有效”测试的项目小组往往面临这样的问题:计划中的时间很快就用完了,即使有迹象表明软件中还有不少缺陷,也只好草草收场,把麻烦留给将来。

测试的主要目的是为了发现尽可能多的缺陷。可是这个观念不容易被人接受。

正确理解测试的目的十分重要。如果认为测试的目的是为了说明软件中没有缺陷,那么测试人员就会向这个目标靠拢,因而下意识地选用一些不易暴露错误的测试示例。这样的测试是不真实的。如果为了说明软件有多么好,那么应当制作专门的演示。千万不要将“测试”与“演示”混为一谈。

看来测试并不单单是个技术问题,还是个职业道德问题。

根据测试的目的,可以得出一个推论:成功的测试在于发现了迄今尚未发现的缺陷。所以测试人员的职责是设计出这样的测试用例,它能有效地揭示潜伏在软件里的缺陷。

如果测试工作很全面、很认真,但是的确没有发现新的缺陷,那么这样的测试是否毫无价值?

不,那不是测试的过失,应当反过来理解:软件的质量实在是太好了,以至于这样的测试发现不了缺陷。

所以,如果产品通过了严格的测试,大家不要不吭气,应当好好地宣传一下,把测试的成本捞回一些。

软件测试的常规分类如表1-4所示,测试的一般流程如图1-5所示。如果对表1-4和图1-5的内容做深入论述,大约还需要几十页的篇幅,这显然超出了本章的范畴。这里就点到为止吧,请读者参考软件测试的有关文献。

表 1-4 测试的常规分类

测试的常规分类	
测试阶段	单元测试、集成测试、系统测试、验收测试
测试方式	白盒测试、黑盒测试
测试内容	功能测试、健壮性测试、性能测试、用户界面测试、安全性测试、压力测试、可靠性测试、安装/反安装测试, .....

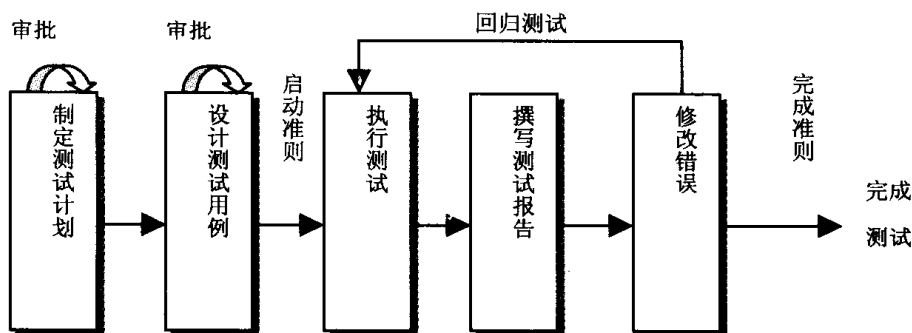


图 1-5 测试的一般流程

### 测试经验谈：

- ✧ 测试能提高软件的质量，但是提高质量不能依赖测试。
- ✧ 测试只能证明缺陷存在，不能证明缺陷不存在。“彻底地测试”难以成为现实，要考虑时间、费用等限制，不允许无休止地测试。我们只好祈祷：软件的缺陷在产品被淘汰之前一直没有机会发作！
- ✧ 测试的主要困难是不知道如何进行有效的测试，也不知道什么时候可以放心地结束测试。
- ✧ 每个开发人员应当测试自己的程序（分内之事），但是不能作为该程序已经通过测试的依据（所以项目才需要独立的测试人员）。
- ✧ 2-8 原则：80%的缺陷聚集在 20%的模块中，经常出错的模块改错后还会经常出错。

## 1.4.7 质量保证

质量保证（Quality Assurance, QA）的目的是提供一种有效的人员组织形式和管理方法，通过客观地检查和监控“过程质量”与“产品质量”，从而实现持续地改进质量。质量保证是一种有计划的、贯穿于整个产品生命周期的质量管理方法。

过程质量与产品质量存在某种因果关系，通常“好的过程”产生“好的产品”，而“差的过程”将产生“差的产品”。人们销售的是产品而不是过程，用户关心的是最终产品的质量，而软件开发团队既要关心过程质量又要关心产品质量。

质量保证的基本方法是通过有计划地检查“工作过程及工作成果”是否符合既定的规范，来监控和改进“过程质量”与“产品质量”。如果“工作过程及工作成果”不符合既定的规范，那么产品的质量肯定有问题。基于这样的推理，质量保证人员即使不是技术专家，他也能够客观地检查和监控产品的质量，这是质量保证方法富有成效的一面。但是“工作过程及工作成果”符合既定的规范并不意味着产品的质量一定合格，因为仅靠规范无法识别出产品中可能存在的大量缺陷，这是质量保证方法的不足之处。所以单独的“质量保证”其实并不能保证质量。

技术评审与测试关注的是产品质量而不是过程质量，两者的技术强度比质量保证要高得多。技术评审和测试能弥补质量保证的不足，三者是相辅相成的质量管理方法。我们在实践中不能将质量保证、技术评审和测试混为一谈，也不能把三者孤

立起来执行。建议让质量保证人员参加并监督重要的技术评审和测试工作，把三者有机地结合起来，才能提高工作效率和降低成本。

质量保证小组（Quality Assurance Group, QAG）有如下特点：

- ✧ 质量保证小组在行政上独立于任何项目。这种独立性有助于质量保证小组客观地检查和监控产品的质量。
- ✧ 质量保证小组有一定的权利，可以对质量不合格的工作成果做出处理。这种权利使得质量保证小组的工作不会被轻视，并有助于加强全员的质量意识。需要强调的是，提高产品质量是全员的职责，并非只是质量保证小组的职责。

质量保证过程域的主要活动如图 1-6 所示。

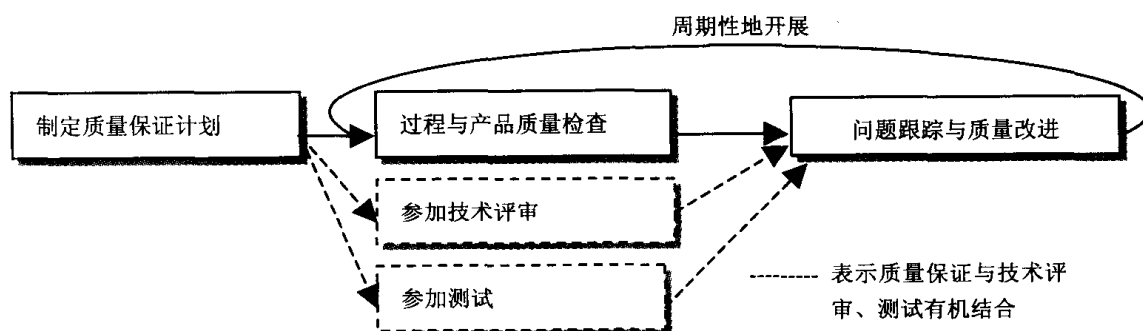


图 1-6 质量保证过程域示意图

## 1.4.8 改错

改错是个大悲大喜的过程，一天之内可以让人在悲伤的低谷和喜悦的巅峰之间跌荡起伏。

我从大三开始真正接受改错的磨炼，已记不清楚多少次汗流浹背、湿透板凳。改不了错误时，恨不得撞墙。改了错误时，比女孩子朝我笑笑还开心。

在做本科毕业设计时，一天夜里，一哥们儿流窜到我的实验室，合不拢嘴地对我嚷嚷：“你知道什么叫茅塞顿开吗？”

我像文盲似地摇摇头。

他说：“今天我花了十几小时没能干掉一个错误，刚才我去了厕所五分钟，一切都解决了。”

他一定要请我吃“肉夹馍”，那得意劲儿仿佛同时谈了两个女朋友。

软件中的错误通常只有开发者自己才能找出并改掉，如果因畏惧而拖延，会让你终日心神不定，食无味，睡不香。所以长痛不如短痛，要集中精力对付错误。

东北有个林场工人马永顺，工作勤奋，一个人能干几个人的活。前 30 年是伐树劳模，受到周恩来总理的接见。忽有一天醒悟过来，觉得自己太对不起森林，决心补救错误。后 30 年成了植树劳模，受到朱镕基总理的接见。若能以此大勇来改错，真是无往而不胜也。我们软件开发人员应当向这位可敬的林场工人学习。

改错过程很像侦破案件，有些坏事发生了，而仅有的信息就是它的确发生了。我们必须从结果出发，逆向思考。改错的第一步是找出错误的根源，如同医生治病，必须先找出病因才能“对症下药”。

---

有人问阿凡提：“我肚子痛，应该用什么药？”

阿凡提说：“应该用眼药水，因为你眼睛不好，吃了脏东西才肚子痛。”

---

根据软件错误的症状推断出根源并不是件容易的事儿，因为：

- (1) 症状和根源可能相隔很远。也就是说，症状可能在某一个程序单元中出现，而根源实际上在很远的另一个地方。高度耦合的程序结构加剧了这种情况。
- (2) 症状可能在另一个错误被纠正后暂时消失。
- (3) 症状可能并不是由某个程序错误直接引发的，如误差累积。
- (4) 症状可能是由不太容易跟踪的人工错误引起的。
- (5) 症状可能时隐时现，如内存泄漏。
- (6) 很难重新产生完全一样的输入条件，难以重现“错误现场”。
- (7) 症状可能分布在许多不同的任务中，难以跟踪。

改错的最大忌讳是“急躁蛮干”。人们常说“急中生智”，我不信，我认为大多数人着了急就会蛮干，早把“智”丢到脑后去了。不仅人如此，动物也如此。

---

我们经常看到，蜜蜂或者苍蝇想从玻璃窗中飞出，它们会顶着玻璃折腾几个小时，却不晓得从旁边轻轻松松地飞走。我原以为蜜蜂和苍蝇长得太小，视野有限，以至看不见近在咫尺的逃生之窗，所以只好蛮干。可是有一天夜里，有只麻雀飞进我的房间，它的逃生方式竟然与蜜蜂一模一样。我用灯光照着那扇打开的窗户为其引路，并向它打手势，对它说话，均无济于事。它是到天亮后才飞走的，这一宿我和它都没有休息好。

---

我们把寻找错误根源的过程称为调试 (Debugging)。调试的基本方法是“粗分细找”。对于隐藏得很深的 Bug，我们应该运用归纳、推理、“二分”等方法先“快速、粗略”地确定错误根源的范围，然后再用调试工具仔细地跟踪此范围的源代码。如果没有调试工具，那么只好用“土办法”：在程序中插入打印语句如 `printf(...)`，观看屏幕的输出。

有些时候，世界上最好的调试工具恐怕是那些有经验的人。我们经常会长时间地追踪某个 Bug，苦恼万分。恰好有高手路过，被他一语“道破天机”，顿时沮丧的阴云就被驱散，你不得不说“I 服了 You”。

#### 修改代码错误时的注意事项：

- ✧ 找到错误时，不要急于修改，先思考一下：修改此代码会不会引发其他问题？如果没有问题，则可以放心修改；如果有问题，那么可能要改动程序结构，而不止一行代码。
- ✧ 有些时候，软件中可能潜伏同一类型的许多错误（例如由不良的编程习惯引起的），好不容易逮住一个，应当乘胜追击，全部歼灭。



- ✧ 在改错之后一定要马上进行回归测试，以免引入新的错误。有人在马路上捡到钱包后得意忘形，不料自己却被汽车撞倒。改了一个程序错误固然是喜事，但要防止乐极生悲。更加严格的要求是：不论原有程序是否绝对正确，只要对此程序做过改动（哪怕是微不足道的），都要进行回归测试。
- ✧ 上述事情做完后，应当好好反思：我为什么会犯这样的错误？怎么能够防止下次不犯相似的错误？最好能写一下心得体会，与他人共享经验教训。

## 1.5 关于软件开发的一些常识和思考

### 1.5.1 有最好的编程语言吗

**作者的观点：**程序员在最初学习 BASIC、Fortran、Pascal、C、C++等语言时会感觉一个比一个好，不免有喜新厌旧之举。而如今的 Visual Basic、Delphi、Visual C++、Java 等语言各有所长，真的难分优劣。能很好地解决问题的编程语言就是好语言。开发人员应该根据实际情况，选择业界推荐的并且是自己擅长的编程语言来开发软件，才能保证有较好的质量与效率。

编程是一件自由与快乐的事情，不要发誓忠于某某语言而自寻烦恼。

### 1.5.2 编程是一门艺术吗

**作者的观点：**水平高到一定程度后，干啥事都能感受到“艺术”，编程也不例外。但在技术行业，人们通常认为“艺术”是随心所欲、不可把握的东西。如果程序员都把编程当成“艺术”来看待，准会把公司老板吓昏过去。

大部分人开发软件是为了满足客户的需求，而不是为了自己享受。本书提倡规范化编程。规范化能够提高质量与效率，最具实用价值，尽管它在一定程度上压抑了“艺术”。编程艺术是人们对高水平程序创作的一种感受，但只可意会，不可言传，不能成为软件公司的一个指导方针。

### 1.5.3 编程时应该多使用技巧吗

**作者的观点：**就软件开发而言，技巧的优点在于能另辟蹊径地解决一些问题，缺点是技巧并不为人熟知。若在程序中使用太多的技巧，可能会留下错误隐患，别人也难以理解。一个局部的优点对整个系统而言是微小的，而一个错误则可能对整个系统是致命的。我建议用自然的方式编程，不要滥用技巧。我们有时确实不知道自己的得意之举究竟是锦上添花，还是画蛇添足。就像蒸出一笼馒头，在上面插一朵鲜花，本想弄点诗情画意，却让人误以为那是一堆热气腾腾的牛粪。

小时候读的《狼三则》故事启示我们，失败的技巧被讽刺为“伎俩”。当我们编程时无法判断用的是技巧还是伎俩的情况下，那就少用。《卖油翁》的故事又告诉我们“熟能生巧”，表明技巧是自然而然产生的，不是卖弄出来的。

#### 1.5.4 换更快的计算机还是换更快的算法

如果软件运行较慢，是换一台更快的计算机，还是设计一种更快的算法？

**作者的观点：**如果开发软件的目的是为了学习或是研究，那么应该设计一种更快的算法。如果该软件已经用于商业，则需谨慎考虑。若换一台更快的计算机能解决问题，则是最快的解决方案。改进算法虽然可以从根本上提高软件的运行速度，但可能引入错误并延误进度。

技术狂毫无疑问会选择后者，因为他们觉得放弃任何可以优化的机会就等于犯罪。类似的争议还有：是买现成的程序，还是彻底由自己开发？技术人员和商业人士常常会有不同的决策。

#### 1.5.5 错误是否应该分等级

微软的一些开发小组将错误分成 4 个等级（Cusumano, p354~p355）：

- ◇ 一级严重：错误导致软件崩溃。
- ◇ 二级严重：错误导致一个特性不能运行并且没有替代方案。
- ◇ 三级严重：错误导致一个特性不能运行但有替代方案。
- ◇ 四级严重：错误是表面化的或是微小的。

**作者的观点：**将错误分等级的好处是便于统计分析，仅此而已。但上述分类带有较重的技术倾向，并不是普遍适用的。假设某个财务软件有两个错误：错误 A 使该软件死掉，错误 B 导致工资计算错误。按上述分类，错误 A 属一级严重，错误 B 属二级严重。但事实上 B 要比 A 严重。工资算多了或者算少了，将会使老板或员工遭受经济损失。而错误 A 只是使操作员感到厌烦，并没有造成经济损失。再例如航空软件操作手册写错了，按上述分类则属四级严重，但这种错误可能导致机毁人亡，难道还算微小吗？

开发人员应该意识到：所有的错误都是严重的，不存在微不足道的错误。只有这样才能少犯错误。

#### 1.5.6 一些错误的观念

**错误观念之一：**我们拥有一套讲述如何开发软件的书籍，书中充满了标准与示例，可以帮助我们解决软件开发中遇到的任何问题。

**作者的观点：**好的参考书无疑能指导我们的工作。充分利用书籍中的方法、技术和技巧，可以有效地解决软件开发中大量常见的问题。但实践者并不能因此依赖于书籍，因为：

（1）在现实中，由于工作条件千差万别，即使是相当成熟的软件工程规范，也常常无法套用。

（2）软件技术日新月异，没有哪一种标准能长盛不衰。祖传秘方在某些领域很吃香，而在软件领域可能意味着落后。

**错误观念之二：**我们拥有充足的资源和经费，可以买最好的设备，一定能做出优秀的软件产品。

**作者的观点：**大公司经常有这种夜郎自大的心态。良好的开发环境只是产出成果的必要条件，而不是充分条件。如果拥有好环境的是一群庸人或者是一群勾心斗角的聪明人，难保他们不干出南辕北辙的事情。

**错误观念之三：**如果进度落后于计划，可以增加更多的程序员来解决问题。

**作者的观点：**软件开发不同于传统的农业生产，人多不见得力量大。如果给落后于计划的项目增添新手，可能会更加延误项目。因为：

(1) 新手会产生很多新的错误，给项目添麻烦。

(2) 老手向新手解释工作及交流思想都要花费时间，使实际开发时间更少。

所以精确地制定项目计划很重要，不在乎计划中的进度看起来有多么快，计划要恰如其分。

**错误观念之四：**只要干活小心点，就能提高软件的质量。

**作者的观点：**软件开发是一种智力创作活动，世上最小心翼翼、最踏实的程序员未必就能开发出高质量的软件来。程序员必须了解软件质量的方方面面（称为质量属性），一定要先搞清楚怎样才能提高质量，才可以在进行需求开发、系统设计、编程、测试时将高质量内建其中。

## 1.6 小结

软件质量属性之间并非完全独立的，而是互相交织、互相影响的。因此，程序设计中要同时兼顾几个质量属性，使程序达到整体最优。要把质量属性牢记在心，这样才能在程序设计时一次性地编写出高质量的、错误较少的代码来，同时也可以减轻查错和调试的负担。

经典的软件工程书籍厚得像砖头，或让人望而却步，或让人看了心事重重。请宽恕作者的幼稚，本章试图用聊天、说理的方式来解释软件工程的道理。软件工程的观念、方法和规范都是朴实无华的，平凡之人皆可领会，但只有实实在在地用起来才有价值。我们不可以把软件工程方法看成是诸葛亮的锦囊妙计——在出了问题之后才打开看看，而应该事先预料将要出现的问题，控制每个实践环节，防患于未然。

研究软件工程永远做不到像理论家那样潇洒：定理证明了，就完事儿。

## 第2章 编程语言发展简史

编程语言之于程序员就如枪之于军人。编程语言不仅是程序员的谋生工具，它们还让我们拥有了“从士兵到将军”的职业发展梦想。让我们先向历史上伟大的编程语言、伟大的人物、伟大的企业致敬。

本章讲述编程语言发展简史，穿插了一些有趣的故事。如今的编程语言比起几十年前算是高度发达了，所以程序员的日子一天比一天好过，真可谓“前人种树后人乘凉”。

### 2.1 编程语言大事记

1822年，英国人 Charles Babbage 设计了差分机。该差分机利用卡片输入程序和数据，类似于百年后的电子计算机。

1834年，Babbage 设计了一台分析机，在穿孔卡片（只读存储器）中存储程序和数据，基本实现了控制中心（类似于今天的 CPU）和存储程序的设想。而且程序可以根据条件进行跳转，这有些类似于今天的程序控制。

1848年，英国数学家 George Boole 创立了二进制代数学，约提前一个世纪为现代二进制计算机铺平了道路。此后，计算机的研制大约沉寂了 40 年，自然也没有什么人来设计程序。

1890年，美国进行人口普查。由于 1880 年的普查用了 7 年的时间进行统计分析，这意味着 1890 年的统计分析可能会超过 10 年。人口普查部门希望能有一台机器，帮助他们提高统计分析的效率。Herman Hollerith 借鉴了 Babbage 的设计，用穿孔卡片存储数据和程序，并制造了处理机器。结果该机器仅仅用了 6 周就得出了人口普查的统计分析结果。Herman Hollerith 因此大发横财，他的公司后来发展成了 IBM 公司，真乃时势造英雄。

1896年，Herman Hollerith 创办了 IBM 公司的前身，开始大量制造穿孔卡片处理机。当真正意义上的电子计算机出现时，穿孔卡片自然地成为最早的程序载体。

1906年，美国的 Lee De Forest 发明了电子管。在这之前的计算机都基于机械运行方式，而在这之后计算机开始进入电子时代。

1924年2月，IBM 公司成立了。从那时起直到今天，IBM 公司始终在计算机工业界占据重要地位。

1937年，英国剑桥大学的 Alan M. Turing 出版了他的论文。没错，就是那个著名的图灵，他在论文中提出了“图灵机”数学模型。现在几乎所有的编程语言都建立在图灵机模型之上。

1937年，贝尔实验室的 George Stibitz 首先用继电器来表示二进制。如果你是那



个时代的先知，也许可以预见到今天的编码方式。

1939年1月1日，加利福尼亚的 David Hewlet 和 William Packard 在他们的车库里造出了 Hewlett-Packard 计算机。机器的名字是两人用投硬币的方式决定的。这两个人后来成立了著名的 HP 公司。

1943年，从这一年开始到1959年，出现了大量使用真空管的计算机，通常被称为第一代计算机。ENIAC (Electronic Numerical Integrator and Computer) 是第一台真正意义上的数字电子计算机。它于1943年开始研制，完成于1946年2月。重30吨，占地170平方米，体积3000立方英尺，用了18000个电子管，功率25KW，主要用于计算弹道和研制氢弹。它的负责人是 John W. Mauchly 和 J. Presper Eckert。

如果你有幸成为 ENIAC 的程序员，你将不得不用机器码和穿孔卡片编写所有的程序，并且直接在内存中读写指令和数据，安排和维护内存的分配。即使增加一行代码，也必须重新考虑所有指令和数据在内存中的分配。编制的程序完全像天书，全由0和1组成。

1949年的 EDVAC (Electronic Discrete Variable Computer) 是第一台使用磁带的计算机。这是一个突破，专家们可以在其上多次编写和存储程序。不过你还是必须使用机器码。这一年的科学杂志做了一个大胆的预测：“未来的计算机不会超过1.5吨”。

1952年，对于程序设计来说是具有重要里程碑意义的一年。MIT (美国麻省理工学院) 在 Whirlwind 系统上使用了符号地址，开始使用汇编语言来编写程序。Whirlwind 被美国空军用于控制实时防御系统。

1954年，IBM 公司的 John Backus 和他领导的研究小组开始研制 Fortran (Formula Translation) 语言，这是一种用于科学计算的编程语言。Fortran 语言于1957年研制完成。Fortran 支持一些最常用的编码方式，如算术表达式、逻辑运算、过程调用、循环和条件等。相对于汇编语言来说，Fortran 可以被称为高级语言，它提高了程序员的编程效率。Fortran 历经变迁，如今演变成为 Visual Fortran。

1958年，Robert Noyce (Intel 公司的创始人) 发明了集成电路。1959年 Grace Murray Hopper 开始研制 COBOL (Common Business-Oriented Language) 语言，并于1961年完成。COBOL 在银行系统和许多大型企业中得到了广泛的应用。直到今天，仍然有许多用 COBOL 编制的程序在大型机上运行。

1960年，来自丹麦、英国、法国、德国、荷兰、瑞士和美国的13名代表举行了一次国际会议，会后在计算机权威刊物 CACM 上发表了《关于算法语言 Algol 60 的报告》。Algol 是一种用日常英语及与常用数学表达式相近的形式表现算法的语言，没有输入输出语句，全部以过程的形式进行描述，并以块结构为基础。Algol 是第一个结构化编程语言。

1961年，IBM 的 Kenneth Iverson 推出 APL 编程语言，专门用于矩阵运算。

1965年，Thomas E. Kurtz 和 John Kemeny 研制了 BASIC (Beginners All Purpose Symbolic Instruction Code) 语言。BASIC 特别适合于计算机教育和初学者使用，后来发展成为 Visual Basic，为 Microsoft 公司挣了很多钱。

1967年，Niklaus Wirth 开始在 Algol 基础之上开发 Pascal 语言，于1971年研制

完成。Pascal 后来成为 Borland 公司用来对抗 Microsoft 公司的利器。但是这个时候，无论是 Microsoft 还是 Borland 都还没有出世。

1968 年，Seymour Paper 和他的研究小组在 MIT 开发了 LOGO 语言。LOGO 语言非常有趣，适用于教育领域。LOGO 语言的标志是一个有趣的忍者神龟。

1969 年，ARPANET 计划启动（Advanced Research Projects Agency Network），这是现代 Internet 的雏形。1970 年，许多大学和商业部门开始接入 ARPANET。Internet 的发展又带动了一批新的语言，但这是 20 年之后的事了。

1970 年，Ken Thomson 和 Dennis Ritchie 开始研制 UNIX 操作系统。

1971 年 11 月 15 日，Intel 公司的 Marcian E. Hoff 研制成功第一块微处理器 4004。它包含 2300 个晶体管，是一个 4 位系统，时钟频率 108kHz，每秒执行 6 万条指令。

1972 年，贝尔实验室发明了 C 语言。C 兼有低级语言和高级语言的功能，被人们称为中级语言。C 是一个功能强大的编程语言，它最初因被用于开发 UNIX 系统而闻名于世。到 20 世纪 80 年代，贝尔实验室又发明了 C++ 语言。C 和 C++ 被誉为是程序员的“正宗编程语言”，它们的广泛应用极大地推动了软件业的发展。

1974 年这一年发生了许多重大的事件。4 月 1 日 Intel 发布了 8 位微处理器芯片 8080。12 月，MITS 发布了 Altair 8800，这是第一台商用个人计算机，价值 397 美元，内存只有 256 字节。同年，Bill Gates 和 Paul Allen 开始开发第一个在 MITS 的 Altair 计算机上运行的 BASIC 程序，他们手头甚至没有 Altair 计算机。

1975 年，Bill Gates 和 Paul Allen 创办了 Microsoft 公司。要是那个时候人们买了 Microsoft 公司的股票该有多好啊！

1976 年，Zilog 推出 Z80 处理器，这是一个 8 位的微处理器。CP/M 就是基于 Z80 的操作系统。

1979 年，Jean Ichbiah 研制了 Ada 语言，被广泛用于美国军方。同年，IBM 公司眼看着个人计算机市场被苹果等电脑公司占有，决定开发自己的个人计算机。Microsoft 公司不但提供了用于 IBM-PC 的 BASIC 语言，还承担了操作系统的开发。

1981 年 8 月，IBM 推出了首款 IBM-PC，同时也为 Microsoft 的崛起铺平了道路。在 IBM-PC 发布的同时，MS-DOS 1.0 和 PC-DOS 1.0 也一起发布。Microsoft 受 IBM 委托开发 DOS 操作系统，他们从 Tim Paterson 那里购买了一个叫 86-DOS 的程序并加以改进。从 IBM 卖出去的叫 PC-DOS，从 Microsoft 卖出去的叫 MS-DOS，Microsoft 精明地保留了继续开发的权利。DOS 的最初版本里 Bug 很多，以至于被称为“Dirty Operation System”，但这却是 Microsoft 独霸 PC 操作系统的开始。

1983 年，Borland 公司成立，其创始人是 Philippe Kahn 和 Anders Hejlsberg，他们合作研制了 Turbo Pascal，并在著名的 Byte 杂志上登广告。售价 49.99 美元的 Turbo Pascal 是一个革命性的产品，它能够在 RAM 中常驻运行，又具有闪电般的编译速度，成为当时 PC 上最流行的开发工具。Borland 也由此迈上了其影响 PC 软件开发工具十几年的道路。

1985 年，Microsoft 发布了 Windows 1.0。最初的 Windows 存在很多严重的 Bug，不仅少有人用而且被人讥笑。一直熬到 1993 年，Windows 3.1 才获得成功。Windows 的图形用户界面与 Apple 公司的类似，以致被 Apple 公司控告。诉讼一直持续到 1997

有人绘制了一张比较直观的编程语言关系图，如图 2-1 所示。

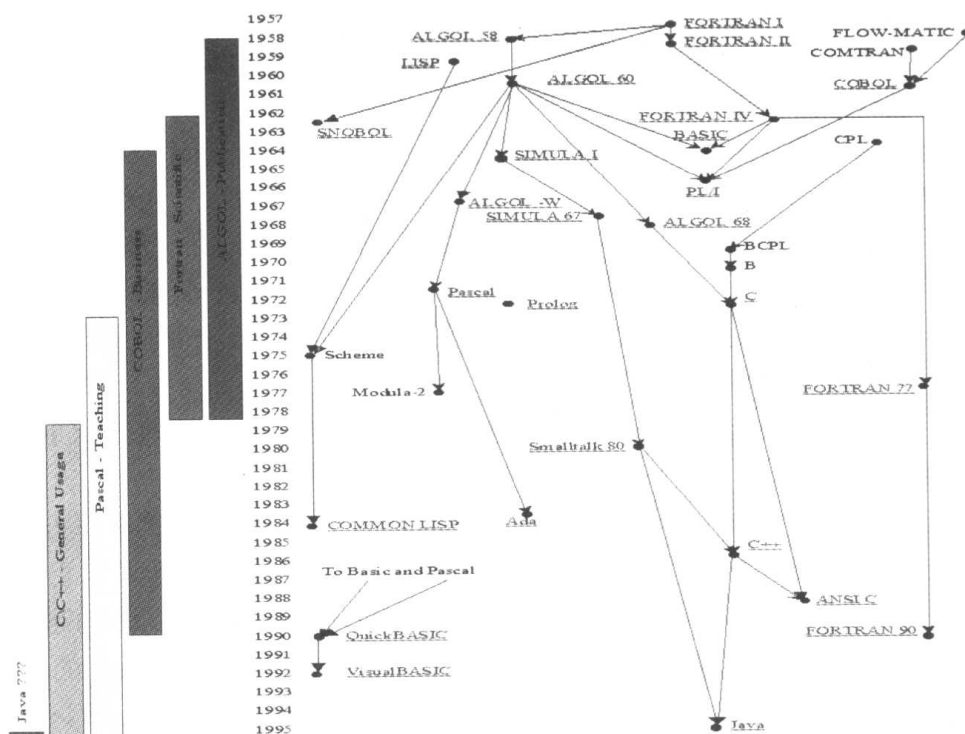


图 2-1 编程语言关系图

## 2.2 Ada 的故事

拜伦是位天才诗人，但并不是一位称职的丈夫，他和妻子的关系极不和谐。在 Ada 出生五个月时，拜伦离家出走，据说主要原因是拜伦不欣赏妻子的出众才华。拜伦的妻子有极高的数学天分，曾学过代数、几何，甚至天文学，这在当时上流社会

的妇女中是极少见的。想不到欧洲也曾流行“女子无才便是德”。

Ada 是一位美丽而有修养的女子，她不幸患了小儿麻痹症，以致双腿瘫痪。Ada 竭尽全力同疾病斗争，终于在 13 岁时她又可以独立行走了。并且在三年养病期间，她还研读了天文学和形而上学。

1834 年，Ada 和 Babbage 首次相遇，那时 Babbage 正试图制造分析机（一种机械式计算机）。尽管 Ada 只有 18 岁，可她被 Babbage 的工作深深地吸引了。Ada 为 Babbage 的分析机编制了程序，于是成为世界上第一位程序员。

Ada 不仅仅为历史上第一台机械式计算机开发了程序，她还预见了计算机的广泛应用，她甚至认为可以用计算机来作曲。她的思想在当时显得太过于超前。1852 年，Ada 在与癌症苦苦搏斗中去世，年仅 37 岁。

1977 年，美国国防部计划开发一种新的编程语言，以替换美国陆、海、空三军使用的五花八门的编程语言。新语言应该叫什么名字呢？五角大楼的一位军官突然想起了多年前的一位年轻女子，历史上第一位程序员，她的名字是 Ada。

Ada 就这样成为一种编程语言的名称。

## 2.3 C/C++发展简史

C 语言由贝尔实验室的 Dennis Ritchie 从两个早期语言 BCPL 和 B 发展而来，并于 1972 年在 DEC PDP-11 型机上实现。C 语言最强大之处在于提供灵活的数据类型和硬件操作能力（位操作能力），因此被广泛用做操作系统软件和设备驱动程序设计的首选语言。UNIX 操作系统就是使用 C 语言开发的第一个大型软件系统。实际上，现在的大多数操作系统都是用 C/C++ 语言实现的。C 语言与硬件无关，因此经过认真的设计，程序员编写出的程序可以移植到大多数不同的硬件平台上。

随着 C 语言的广泛应用，出现了许多不兼容的版本，这对于 C 的可移植性来说是一个非常严重的问题。1983 年，ANSI 开始组织 C 的标准化工作。1989 年，ANSI C 语言标准正式颁布。此后，ANSI 与 ISO 合作在全世界范围内对 C 语言进行标准化和推广。1990 年，标准文档出版，ANSI/ISO C 正式诞生。C 语言的最新情况是：1999 年，ISO 批准了一个新的 C 语言版本，即 C99。

1979 年，贝尔实验室的一个年轻工程师 Bjarne Stroustrup 为了使 C 语言成为一种更好的实现大型软件工程的编程语言而展开了实验，实验的内容就是对 C 进行扩展。那时，一般项目都需要十几万行的代码。现在，光是 Windows 2000（前身是 NT 5.0）就有 3000 多万行代码。当软件规模超过了 100 000 行代码的时候，C 语言的缺陷达到了让人无法忍受的地步。

最初对 C 的扩展就是给它增加类的概念，从而产生了“带类的 C”，它可以很好地支持封装和信息隐藏。1983 年，又对带类的 C 做了几处修改和扩展，于是“C++”这个名字产生了。

Bjarne Stroustrup 是 C++ 的鼻祖，受到无数程序员的敬仰。按中国人的说法，Bjarne 果真是聪明绝顶。当初他要是把“++”注册了专利，不仅可以发一笔大财，而且也



不会导致现在“++”满天飞。

1985 到 1989 年间, C++经历了最主要的革新: 相继增加了保护成员、保护继承、模板, 以及有争议的多重继承等特性。C++到了需要标准化的时候。

C++的标准化不同于 C 的标准化, 它从一开始就是国际性的合作。ANSI C 委员会使用 Kernighan 和 Ritchie 写的《The C Programming Language》作为起点, 而 ANSI C++委员会使用 Ellis 和 Bjarne 写的《Annotated C++ Reference Manual (ARM)》作为基础文档。1998 年, 在做了一些次要的修改后, ANSI/ISO C++产生了。C++的国际性工作使得该标准在世界范围内被广泛接受。

关于 C++的设计和演化, 在 Bjarne Stroustrup 的著作《The Design and Evolution of C++》中有详细的叙述, 其中总结了 C++的一些设计原则:

- ◇ C++的每一步演化和发展必须是由于实际问题所引起的。
- ◇ C++是一门编程语言, 而不是一个完整的系统。
- ◇ 不能无休止地一味追求完美。
- ◇ C++在其存在的“当时”必须是有用处的。
- ◇ 每一种语言特性必须有明确的实现方案。
- ◇ 总能提供一种变通的方法。
- ◇ 不强制于人。

.....

从一开始, C++面向的就是那些从事软件开发工作的程序员。所谓的“完美”被认为是不可能达到的。在 C++语言的演化过程中, 来自用户的反馈和语言实现者们积累的经验是最为重要的。

## 2.4 Borland 与 Microsoft 之争

Borland 公司成立于 1983 年, 曾经是世界第三大软件公司。在软件开发工具领域, Borland 公司几乎是高品质的代名词。

Borland 和 Microsoft 曾经围绕软件开发工具展开了一场没有硝烟的持久战。

在 20 世纪 90 年代初期, 那时还是 DOS 时代。Borland 公司的 Turbo C/C++在编译性能和易用性方面可谓独领风骚, Turbo 系列风靡全球。相比之下, Microsoft 的 C/C++产品越发显得笨拙。那个时期是 Microsoft 的 C/C++产品部门最难过的日子, Microsoft 的员工们都认为公司里最笨的人全集中在 C/C++部门了。还有一个滑稽的事件, 在 Microsoft 的开发工具刊物上, 出现了一个名字叫做 Buck Forland 的作者, 不断发表各种文章嘲笑 Borland 公司的产品, 引起了 Borland 公司及其拥护者的强烈不满。许多人推测该作者应该是 Microsoft 的软件工程师, 用这个笔名来发泄不满。

在 Windows 3.x 流行的那几年里, Microsoft 的 Visual C++ 1.0 仍然不敌 Borland C/C++。Microsoft 毕竟比 Borland 财大气粗, 既然明着斗不过, 那就挖 Borland 公司的墙脚。据说后来 Visual C++小组的成员有 60%是从 Borland 公司跳过来的。

自从 Windows 9x 推出之后, Visual C++ 逐渐占据了上风, 因为没有人能比 Microsoft 更好地利用 Windows 操作系统的特性。

虽然在 C/C++ 领域败下阵来, 但是 Borland 公司还握有 Pascal 这张王牌。与 C++ 有 ANSI 标准不同的是, Pascal 几乎是被 Borland 公司独家拥有。1995 年, Borland 公司推出了 Pascal 的现代版本 Delphi 1.0。Delphi 是个卓越快速应用软件开发工具 (RAD), 迟来的 Visual Basic 3.0 在 Delphi 面前一败涂地。从此, Delphi 赢得了令人肃然起敬的绰号“VB 杀手”(VB Killer)。这大概又会成为 Microsoft Visual Basic 小组“心中的痛”。Microsoft 只好继续使用挖人的手段, 甚至把 Delphi 的首席设计师 Anders Hejlsberg 也挖走了。若干年之后, Microsoft 为了反击 Java 而开发了 C#, 而 C# 首席设计师就是从 Borland 公司挖过来的 Anders Hejlsberg, 这真是“它山之石可以攻玉”。

从技术上讲, Borland 的开发工具的确称得上是无与伦比的卓越。Borland 每次推出新产品或者新版本时, 都会赢得程序员们的赞誉。可以说 Borland 的产品是被 Microsoft 公司的 Windows 垄断地位和大批量挖人的方式打压下去的。Microsoft 胜之不武, 但这是商业竞争。

20 世纪 90 年代初期的 C/C++ 程序员对 Borland 公司有深厚的仰慕之情。本书作者在读大学时用 Turbo C 2.0 和 Borland C++ 3.1 开发过不少软件。我曾对同伴们说: 等我以后挣钱了, 一定要买 Borland 的正版软件。

大约 5 年后的今天, 在我的项目购买 Borland 的 CORBA 产品时, 我对 Borland 销售人员讲了我在读大学时候的那个心愿, 并希望购买 Turbo C 2.0 和 Borland C++ 3.1 作为留念, 可惜这两个产品已经没有了。

Borland 犹如楚楚动人的少女, 带着美丽和忧伤步入了红尘, 经受着岁月对它的侵蚀。这些回忆仿佛触动了尘封多年的初恋情节, 令人一丝丝心痛。

我们温柔地期待 Borland 走好。

## 2.5 Java 阵营与 Microsoft 的较量

1995 年 5 月 23 日, Sun 公司的 John Gage 和 Netscape 公司执行副总裁 Marc Andressen 联合推出了 Java 技术。当时 Netscape 的浏览器是进入 Internet 世界的最主要工具, Netscape 大有“微软终结者”的姿态。

Java 最初只是 Sun 公司一个秘密项目的编程工具。Java 的开发者们当初甚至没有想到把它发展成一种编程语言的“野心”, 更不用说后来的 Java 平台了。那是 1991 年, 那个秘密项目叫做“Green Project”, 项目人数最多的时候也只有 13 人。Sun 公司启动这个秘密项目的动机是, 他们认为计算模式将转向基于消费类设备的分布式计算。

“Green Project”小组开发了一个叫做 StarSeven 的数字设备。当时“Green Project”成员所在办公室的电话系统将任何电话切过来的操作方式是\*7。StarSeven 是一个不同数字设备之间的代理 (Agent)。StarSeven 可以控制多种不同平台的设备, 这得益

于 Gosling 发明的一种叫做“OAK”的跨平台编程工具，这个“OAK”就是后来大名鼎鼎的 Java。OAK 的名字来自于 Gosling 办公室窗外的一棵橡树。这些人无意之中发明了可以改变计算机计算模式的技术。

Internet 的迅速普及拉开了 Java 时代的帷幕……

和绝大多数公司一样，Microsoft 也取得了 Java 的开发许可证。但在随后推出的 Visual J++ 中，Microsoft 提供了依赖于 Windows 平台的 Java 实现，这与 Sun 推崇的“一次编译，到处运行”的口号大相径庭。于是 Sun 发起了一场“100% Pure Java”的宣传运动，其声势浩大，几乎成为了一种文化。

Sun 还对 Microsoft 提出诉讼，要求赔偿 35 亿美金。Microsoft 最后被判决赔偿 2 000 万美金。Microsoft 咽不下这口气，随之做出了 Windows XP 系统不再支持 Java 的决定。Sun 又针对 Microsoft 的这个决定打起了官司，指责 Microsoft 利用垄断的力量阻碍 Java 的发展。

正如评论所说，Sun 和 Microsoft 的这场官司，真正受害的是广大开发者。

现在，Java 阵营的最新平台是 J2EE，Microsoft 则推出 .NET 与之对抗。最有名的一次交锋是“Pet Store”（宠物商店）性能竞赛。

Pet Store 是 Sun 公司推出的一个范例应用程序，用来帮助人们在 J2EE 平台上开发应用软件。完整的 Pet Store 提供了一个基于浏览器的 B2C 购物环境（当然，你只能买宠物）、一个基于 XML 的网站管理工具和一个基于 Web Service 的 B2B 交易系统。

除了展示 J2EE 的功能之外，它还展现了 Java 平台的其他功能。看看图 2-2，注意到不同的国旗了吗？这代表了 Pet Store 的不同语言的版本。在 Java 中实现多语言是一件比较容易的事。

Pet Store 虽小，但是五脏俱全，常用的搜索、账号和购物车功能都实现了。许多 J2EE 应用服务器的供应商都在其产品中提供了 Pet Store 的相应实现。因此在 J2EE 社区里，Pet Store 被奉为圣典。Pet Store 对应于 J2EE 的版本，现在已经推出了 1.3 版，增加了许多新功能，其无限版本也发布了，这个好东西是免费的。



图 2-2 Pet Store 的界面

Pet Store 是一个如此有名的范例，以至于 Microsoft 在推出 .NET 平台时，专门实现了一个 Pet Store 的 .NET 版本，并且宣称这个版本比 Sun 的快了 28 倍。这在当时激起了轩然大波，无数人都在讨论这个话题。

但是，很快证明这只不过是 Microsoft 的营销手段而已。Microsoft 大概是高兴得昏了头，居然公开了源代码。于是真相大白了，Microsoft 采用了一种与 Sun 完全不同的体系结构。Pet Store 的 Sun 版本是为了揭示 J2EE 平台的各种用法，因此采用了一种复杂的、极具扩展性的体系结构，涵盖了 J2EE 的方方面面。而 Microsoft 的 Pet Store 完全出于宣传的考虑，采用了一种极简单的结构：在 ASP 页面中直接调用存储过程，速度自然快了很多。数年前 Microsoft 在比较 Delphi 与 Visual Basic 时也采用了同样的手段，他们在 Delphi 示例程序中使用了一个远程数据库调用，而在 VB 的代码中却使用了本地调用。然后宣传 Visual Basic 的性能比 Delphi 的优越。

最具讽刺意义的是，Oracle 不久也公开宣称，它们实现了一个 J2EE 平台的 Pet Store，比 Microsoft 的又快了 3 倍，不知道 Oracle 用了什么伎俩，事情发展到了这一步，变成了一出闹剧。

.NET 和 Java 渊源还是很深的。微软公司挖来 Borland 的天才 Anders Hejlsberg 后，他推出的 Windows 平台下的 Java 虚拟机的效率甚至超过了 C 语言程序的效率。由于 Sun 的起诉，微软在推出 Visual J++ 后直接把它转移到了 .NET 上，甚至超越了语言。如果 Sun 不打官司，也许我们现在只学一门 Java 语言就够了。本质上 Java 和 .NET 都是虚拟运行时平台上的开发语言。虽然 .NET 现在好像只支持 Windows，但我想这应该只是微软的市场策略而已——为了巩固微软 Windows 的市场，不支持在其他操作系统上运行，现在能够影响全世界开发人员的，微软舍我其谁？！

抛开大公司之间的竞争话题不提，平心静气地看 Microsoft 的 .NET 和 Sun 的 J2EE，尽管两者方法不同，但都具备许多优秀的特征，两者难分优劣。

.NET 和 J2EE 的可移植性都非常好。虽然 .NET 的核心只能工作在 Windows 环境下，但从理论上讲可以支持多种语言开发，只要这些语言的子集已经定义好，并为它们建立了 IL (Intermediate Language) 编译器。对于 J2EE 来说，只要遵循 Java VM 规则和一组平台需要的服务，就可以在任何平台上工作。由于 J2EE 平台的所有规范都已经向公众公布，许多供应商可以提供 J2EE 的兼容产品和开发环境。

.NET 并不是一种精巧的标志，而是意味着 Microsoft 产品战略的重大转移。Java 清除了平台的障碍，但是为了用 J2EE 来做开发，用户必须在 Java 环境下工作。而 .NET 是想让用户使用自己喜欢的语言来建造 .NET 应用软件，这个设想是十分美妙的。

对于 Windows 环境下的软件开发商而言，.NET 是一个好的架构，用户可以将许多事情交给 .NET 去做。例如 ASP.NET 比 ASP 好，ADO.NET 比 ADO 和 DCOM 出色，C# 比 C/C++ 更好用。不过，虽然 .NET 平台描绘了美好的蓝图，但其设想要全部成为现实还有较长的路要走。例如 IL 公共语言的运行，目前还有某些明显的不足。想要把每一种语言和元件在运行时集成起来，必须定义这种语言的子集或超集，并清晰地映射到 IL 上；此外必须定义结构，以便提供 IL 需要的元数据；还有，必须开发适用于两种编译语言结构的编译器并集成到 IL 部件的代码中；同时还要生成对现有 IL 元件的语言专用接口。

由于历史原因,若想在 Java 语言中使用其他语言,必须开发非 Java 语言到 Java VM 的众多转换器。因此,在 Java 环境中编写代码,就必须承受额外的翻译工作。如果目标环境是 J2EE,程序员通常会选择 Java 来编程。如果目标环境是.NET,那么程序员将会选择 C#……

## 2.6 小结

编程语言发展到今天,已经越来越平台化。掌握一门编程语言,不仅要求懂得语法,还要能熟练使用该语言的集成开发环境和相应的库函数。

世上不存在最好的编程语言,每一种语言都有其优点和缺点,能够很好地解决应用问题的编程语言就是好语言。开发人员应当根据待开发产品的特征,选择业界推荐的并且是自己擅长的编程语言来开发软件。

语言之间存在一定的相似性,学好一门语言后再学其他语言就容易得多,所以精通一门编程语言将使你长期受益。



## 第3章 程序的基本概念

不少程序员虽然能够使用 C++/C 语言来编写程序，却无法准确地阐述什么是程序设计语言、语言实现、程序库、开发环境及程序的工作原理等概念，以致走入了语言学习的误区。本章将尽量把这些纠缠不清的概念和它们之间的关系阐述清楚，为本书后面的章节奠定基础。

### 3.1 程序设计语言

程序设计语言实际上就是一套规范的集合，主要包括该语言使用的字符集、直接和间接支持的数据类型集合、运算符集合、关键字集合、指令集合、语法规则，以及对特定构造的支持，例如函数（过程）的定义、抽象数据类型的定义、继承、模板、异常处理等。这些内容就是一个语言的构造或者说特征集。可见，语法只是语言的一部分，它指导程序员如何把语言的各种构造组合起来形成一系列可以解决实际问题的可执行命令，这就是程序。一种语言对于它的各种构造的支持是通过各种关键字集合及其语法规则来实现的。

就拿标准 C 语言来说，它支持函数设计，但是语言本身并没有提供任何现成的函数可以直接调用（你可能认为 `sizeof` 是一个函数，其实它是一个运算符）；它支持用户定义 `struct`、`union`、`enum` 等，但是它本身并没有提供任何具体的 `struct`、`union`、`enum` 类型供程序员使用。

有人会问，我们学习 C 语言的时候总是首先学习它的“格式化 I/O”，以便看到自己程序的运行成果，难道“格式化 I/O”不是 C 语言的组成部分吗？

确实不是！

标准 C 语言没有提供 I/O 的实现，只是定义了标准的 I/O 函数接口，所有的 I/O 工作都是通过库函数来完成的，在这一点上它不同于 BASIC。标准 C++ 语言继承了 C 的 I/O 库函数接口，并且重新定义了自己的面向对象的 I/O 系统。

现在你该明白了，I/O 系统并不是 C++/C 语言本身的组成部分，函数库和类库也不是它们的组成部分。那么为什么要提供它们呢？实际上应该说是语言实现（参见本章 3.2 节）按照标准接口提供了它们的一个实现版本，算是语言实现的附加产品。它们都是用语言支持的基本特征开发出来的（有的直接使用了汇编语言），目的是方便程序员进行程序设计，使他们从烦琐的底层硬件操作中解脱出来而专注于高层的业务逻辑处理。

作为一种语言，它必须定义它能够支持的所有合法的语法结构及其组合应用，而语言实现同样要能够接受所有可能的合法的代码。但是对于面向实际应用的程序员而言，完全没有必要学习和使用那些过于复杂的、高难度的和罕见的语法结构及

其组合。就像 C++ 之父 Bjarne Stroustrup 说的那样：“你使用一个语言特征是因为你需要它，而不是因为它存在”。

学习一门程序设计语言，并不需要掌握其全部的语法，关键是要学习使用语言来解决实际问题的方法。例如，C 语言的格式化 I/O 非常复杂，有不少程序员努力去记住那么多的格式控制符号，其实完全没有必要！还有 C 运算符的优先级和结合率，也没有必要把它们完全搞清楚，遇到这种问题时只需要按照自己要求的计算顺序多使用“()”就可以解决。很多人在学习程序设计语言时常常沉迷于语法，这是学习的误区！

如果记不住很多语法细节，你可以查阅手册，但是程序设计的道理、解决实际问题的方法是没有地方可查阅的。如果你所掌握的语法和程序设计方法能够高效地解决实际工作中的各种问题，那么表明你已经掌握了这门语言。

## 3.2 语言实现

虽然 C++/C 并没有提供对 I/O 的直接支持，但是由于类似的函数库和标准的类库对于实际开发工作太重要了，所以 C++/C 标准 (ANSI/ISO C++/C) 规定了这些库的标准接口，但是并没有提供其实现，因为这是语言实现的任务。

那么什么是语言实现呢？语言实现就是具体地实现一种语言的各种特征并支持特定编程模式的技术和工具。一般地讲，编程语言的实现就是编译器 (compiler) 和连接器 (linker) (编译 - 连接模式) 或者解释器 (interpreter) (解释模式) 的实现，即用来分析你的源代码并生成最终的可执行机器指令集合的技术和工具，以及一套标准库实现。

语言最终要表现为某个或某些具体的实现版本，但是语言实现并不就是语言本身，因为对于那些允许扩展的标准化语言，它们的实现往往会对那些可扩展的部分进行必要的修改和扩展，这就产生了方言 (dialect)。C++/C 就是允许扩展的语言，即它的标准规范文本中定义和说明了许多实现相关的及平台相关的特征细节，这就要求语言实现根据具体的平台环境和实现技术来修改或定义自己的解决办法。比如在 C++/C 标准规范中，程序的行为就有一种“实现定义的行为 (即依赖于实现的行为)”，就是程序在运行时某些情况下的行为要靠具体的语言实现来定义，而标准规范不会强加定义。C++/C 语言有许多的方言，像 Turbo C++/C、Borland C++/C、Microsoft C++/C、GNU C++/C 等，都是 C++/C 不同的实现版本。所以，要根据你所使用的 C++/C 语言实现来确定那些“实现定义的行为”的具体含义。

另外，语言标准可能会要求实现提供可用的标准化库 (Standard Library)。库增加了语言的灵活性和可扩展性。各种库的实现方法可能不同，但是其接口必须是标准化的 (一致的)，否则会给用户带来不便。编译器开发商在提供语言实现的同时可能还会提供集成开发环境 (IDE)，不论是可视化的还是非可视化的 IDE，其目的都是帮助程序员提高编程效率。

一个具体的语言实现必须支持语言规范所定义的核心特征，除此之外的特征和

对核心特征的修改都属于扩展。例如 Microsoft C++/C 就分别对 ANSI/ISO C++/C 进行了必要的扩展，具体表现在关键字、类型转换、单行注释、变长参数列表、作用域规则及增加的编译器选项等方面（详见其编译器文档）。此外，随着技术的发展，解决了一些问题，而到了需要对语言本身进行修订的时候，例如增加一些新的特征和删除一些过时的特征，这就需要对语言的核心进行扩展，当然语言的实现也要做出相应的修订。例如 C++ 希望将来能够增加对象持久性、并行处理等特征，而这些恐怕无法通过库来实现，必须对语言的核心特征进行必要的扩展才能解决，但是现在的技术水平还无法完成这个任务。

由于存在着“依赖于实现的行为”等诸多因素，不同语言实现之间可能并不完全兼容。这种不兼容包括代码的不兼容、特征实现的不兼容、库的不兼容、编译器和连接器的不兼容，以及它们生成的中间文件的不兼容即二进制不兼容，等等。就拿标识符重命名 (Name-Mangling) 方案来说，Microsoft C++ 采用的方案就和 Borland C++ 的不同 (C++ 标准并没有规定这些，但是 C 标准规定了)，从而导致一方编译生成的目标文件 (.obj) 拿到另一方的连接器上无法连接的问题；还有不能同时使用不同实现的库等问题。

虽然一种语言可能存在不同的实现，但是学好标准语言本身无疑是最重要的！过去经常听说“某某人在钻研 Visual C++”，我不知道他是在学习 C++ 语言还是在学习 Visual C++ 的 IDE 和 MFC 或者兼而有之。有些人甚至错误地认为学会使用 Visual C++ IDE 就是学会了 C++ 语言。这又是一个误区！

**首先掌握语言的特征及其使用方法，再学习具体的语言实现才是语言学习的正道！**

实际上，语言实现要解决的根本问题是如何才能高效地使用这种语言开发程序，这不仅包括对实现一个程序（编译—连接）的支持，还包括对程序开发设计活动的支持。比如 C++/C 头文件与实现文件的分离、多源文件程序的组织、单独编译技术、调试技术等都是特定的语言实现开发出来的标准技术，现在已经被广泛采用，它们并非作为语言的 C++/C 本身所关心的问题。

我们不能因为 Bjarne 说过那样的话就不去使用甚至不去学习 C++/C 的高级特性。你可能认为：即使不使用它们，也照样能够实现需要的功能。是的，你能够！因为任何问题都不可能只存在一种解决办法。但是能够做一件事情和做好一件事情是完全不同的两码事儿！你是否认真考虑过你这样做的代价和结果呢？

要想高效地使用一种语言，毫无疑问你首先必须了解它的各个特征和它们的使用方法，以及使用它们有什么样的利弊。此外，你还必须学习几种编程方法和编程模式，以及高效地和高质量地把它们组合起来的技巧，比如结构化编程、模块化编程及过程式编程是基本的编程方法和编程模式，而基于对象、面向对象、面向组件、泛型编程乃至事件驱动编程和基于规则的编程则是高级的编程模式。可以预见，模块化的面向对象程序设计模式、面向组件的程序设计模式、泛型编程都将是未来的主流模式。这还不够，最后你要学习各种编程环境和标准库或其他库的使用方法，要充分利用现有的资源而不要做重复劳动，这样才能用好一门语言，才能开发出高质量的程序。



**【教你一招】:** 你要学会在特定的平台下学习具体的语言实现,除了要了解其编译器手册和帮助外,你还可以编写一些简短的程序来测试 C++/C 的每一个特性,观察其输出结果(其他语言也是类似的方法)。

最后要说的是,仅仅学习一门程序设计语言是不够的,你必须至少学会两门语言并逐步培养运用它们解决实际问题的能力和水平,才能在实际工作中做到游刃有余。

## 3.3 程序库

程序库(Library)是由具体的语言实现提供的,它使用语言本身的基本构造开发而成。可重用的库不仅是“软件重用”思想的体现,也是“面向对象编程”的目标之一。语言与库的关系在前面几节里已经讨论了很多。

库的例子有很多,例如,只要你使用 C++/C 编程就必然会用到的 C Runtime Library 及 STL 等。现在主流的 C++实现及其集成开发环境(IDE)大都提供了不止一个库,除了 C++标准库(包括 I/O 和 STL 等)外,它们的开发商还提供了支持可视化事件驱动编程的类库,如 MFC、OWL、VCL 等,这几种库并不冲突,可以同时使用。此外还有一些第三方开发的程序库和类库,如果它们和其他库一起使用,就要考虑是否存在二进制兼容性问题。

一般说来库是可替换的,即你可以安全地把 IDE 的缺省库卸载,然后安装另一个库,应该也可以正常工作。对于以源代码形式提供的库,必须使用当前的编译器对其重新编译;如果是二进制级的库,除非它的开发商保证该库的实现与 IDE 的缺省库是二进制兼容的,否则不能使用。

**【规则 3-1】:** 尽量采用标准库中提供的函数和类来编程,而不要创建自制的版本。这不仅可以提高开发效率和程序的性能,而且可以改善程序的可移植性。因为这些库不仅是每一个语言实现必须提供的,而且它们经过精心的设计、调试和测试,可以保证高效而正确地执行。

## 3.4 开发环境

编程序就像是在写文章。写文章要求你首先会一门语言(如同程序设计语言),要有内容(如同代码),还要有各种工具,如桌子、笔墨纸砚等(如同你的工作平台)。文章刚写出来时不能马上交给读者看,还需要排版、校对、印刷和发行(如同代码调试、编译连接和发布)。

通过上面的比喻,你应该明白语言与开发环境和开发工具的区别了吧!不过还是有些人喜欢把开发环境当做语言来学习,以为学会了开发环境(和类库)就学会了语言,就可以编写出高质量的程序来!

**开发环境**泛指支持软件开发的一切工具，例如操作系统、代码编辑器、编译器、连接器、调试器，等等，典型的 C++/C 开发环境如图 3-1 所示。**集成开发环境 (IDE)**则是把编辑器、编译器、连接器及调试器等各种工具集成到一个工作空间中。例如，Visual C++ 的 IDE 不仅提供了默认的编译器 (CL.EXE)、连接器 (LINK.EXE)，还集成了调试器、跟踪器和剖视器等，并可以设置工程选项、编译器选项和连接器选项等。如果没有集成开发环境，就得手工编辑编译连接的命令行或者 makefile，手工编辑它们的参数设置，光是这些工作就会把你搞得头晕眼花。

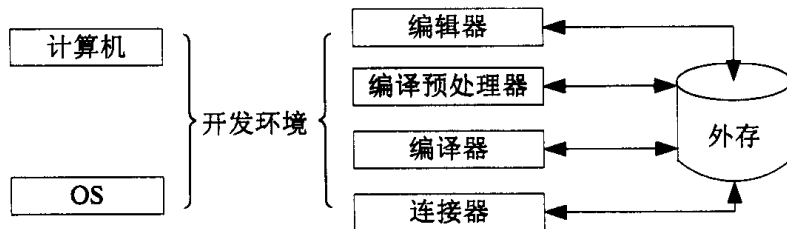


图 3-1 C++/C 开发环境

## 3.5 程序的工作原理

**程序**既可以指开发完成的可执行文件及其相关文件和数据，也可以指正处于开发阶段的源代码及其相关文件和数据 (IDE 称之为程序工程)。程序工程包含一个或多个编译单元、资源文件、静态库及配置文件等。把一个 C++/C 程序工程转变为一个可执行程序要经历编译预处理、编译、连接等过程。一个可执行文件的结构不一定就是它被加载到内存中运行时的内存映像，但是至少包含代码段、静态数据段、堆栈段这三个部分。其中代码段包含源程序中的可执行语句序列，静态数据段存放全局变量、静态对象、符号表等，堆栈段留给函数和线程使用。堆 (heap) 和自由存储空间不属于程序，而是属于操作系统，但是应用程序可以通过动态内存分配指令来获得它们的使用权。

现在的计算机仍然遵循冯·诺依曼的“**存储程序控制**”原理。本质上，任何一个**程序**都是由待处理的数据和一系列处理它们的指令 (操作) 组成的，这些指令通过内存地址来访问待处理的数据。程序中任何复杂的操作 (例如显示复杂的图形或窗口) 最终都被转换为简单的加法运算让计算机来执行。程序在运行的时候首先要求把内存操作数的地址通过数据总线 (DB) 传递到 CPU 寄存器中，然后 CPU 指示将它送到地址总线 (AB) 上，接着内存单元的数据就会“流”入 CPU 的接收寄存器中；然后取第二个操作数；最后执行加法运算。函数调用则是首先提取函数体的首地址到 CPU 寄存器中，然后将 CPU 指令指针修改为这个寄存器中的值，CPU 从内存提取下一条指令时就可以取到函数的第一条指令，这样就实现了函数跳转。

典型的 C++/C 运行环境如图 3-2 所示。



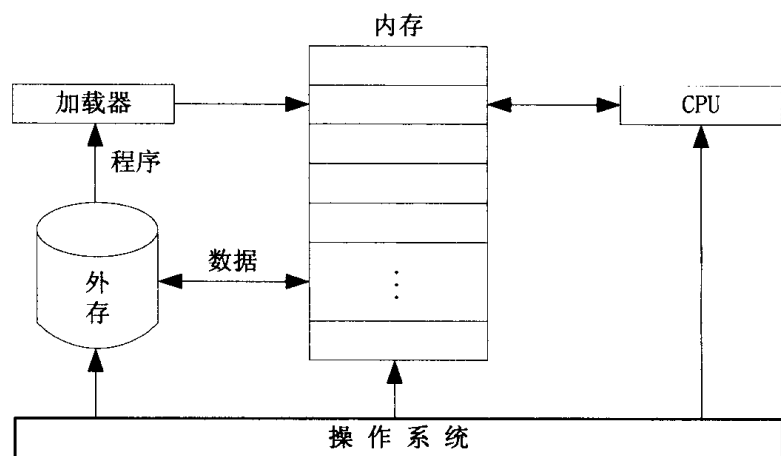


图 3-2 C++/C 程序运行环境

存在于二进制可执行程序中的只是指令、地址和数据，没有别的东西（实际上地址也是一种特殊的数据），这就是“运行时”的本质。像标识符（类型名、变量名、函数名等）、类型定义、`const` 关键字、访问限定符 `public/private/protected`、引用（&）等只是存在于源代码中，它们不会被带入二进制可执行程序中，这是“编译时”的本质。C++/C 源代码中的语句、指针和变量都将被转换成二进制程序中的指令、地址和数据。因此，通过名字直接引用一个变量、对象及其成员，这样的代码在编译和连接完成后，实际上都被转换成了通过变量、对象或成员变量的地址（即内存单元的地址）进行访问。

## 3.6 良好的编程习惯

C++/C 语言灵活而又复杂，有经验的 C++/C 程序员往往会因为自己能够以一些古怪的、烦琐而又令人困惑的方式使用它而沾沾自喜。但这并非就是良好的编程习惯，也不能产生高质量的代码。

**【建议 3-1】：** 应该用简单而直接的方式编写 C++/C 程序，这就叫做 KIS（Keep It Simple）编程准则。不要毫无章法地“滥用”语言。

高质量的代码源于高质量的设计和好的编程风格。

在设计上要追求简单和低耦合。许多遗留的程序错误解决不了，在很大程度上是因为设计本身存在不必要的复杂部分。什么是好的设计？一个好的设计应该恰如其分地反映和解决一个具体的问题，即解决方案与问题一致，没有不必要的特性。在面向对象分析和设计（OOA&D）中，耦合性用来衡量不同对象之间和不同模块之间的依赖程度。好的设计应该尽量降低对象之间和模块之间的耦合性。松耦合的程序易于理解、易于实现、易于测试和维护。

一个良好的编程习惯可以用于任何编程语言中，用它生产的代码不仅容易阅读、容易理解，而且易于调试和测试。良好的编程风格是良好的编程习惯的具体表现。

书写的代码结构安排要合理，版式要清晰一致，给标识符取个“自说明”的名字，不要使用过于复杂的语句和表达式，使用清晰的注释等，这些都是建立良好的编程风格的指导原则。

有的程序员认为，越是能够正确地使用复杂语句书写代码，这个人的水平就越高，这种观念太偏激。伟大的软件不一定就是用复杂的编程技巧实现的。实际上，软件的质量乃至代码的质量与编程技巧没有必然的联系。且不说复杂语句是否真的正确，光是调试和测试这样的语句就很困难，而且维护代码的人不一定能正确地理解它们的语义。

这就好比我们骑自行车上班，本来在人行道上走得好好的，突然蹿到了机动车道上，虽然你的车技很好，机动车道也不是不能走，但是难保不发生交通事故。

良好的编程习惯不是一天就能养成的。本指南从第5章开始将在这方面给出一些通用的指导和建议。

**【提示3-1】:**

虽然 C++/C 与硬件无关，具有很高的可移植性，但是不同的语言实现及不同的硬件平台之间必然存在许多兼容性问题，它们增加了 C++/C 程序的移植难度。所以，并不是说用 C++/C 编写的程序就能保证绝对的可移植性，程序员需要进行针对编译器和机器平台的良好可移植性设计，并把这种可移植性内建在程序代码中。



## 第4章 C++/C 程序设计入门

本章讲解 C++/C 程序设计的入门知识，篇幅较长，但是重点不是罗列语法，而是阐述被一般教科书所忽略的编译器实现原理（技术内幕）和编程规范。

本书附录的一些试题取自本章，的确曾经考败过一大批程序员。所以入门并不意味着简单，实际上本章内容出乎意料地深入。作者并非想给读者（尤其是初级程序员）当头一棒，而是想强调“良好的开端是成功的一半”。

### 4.1 C++/C 程序的基本概念

#### 4.1.1 启动函数 main()

C++/C 程序的可执行部分都是由函数组成的，main()就是所有程序中都应该提供的一个默认全局函数——主函数——所有的 C++/C 程序都应该从函数 main()开始执行。但是语言实现本身并不提供 main()的实现（它也不是一个库函数），并且具体的语言实现环境可以决定是否用 main()函数来作为用户应用程序的启动函数，这是标准赋予语言实现的权利（又是一个“实现定义的行为”☺）。

虽然 main 不是 C++/C 的保留字（因此你可以在其他地方使用 main 这个名字，比如作为类、名字空间或者成员函数等的名字），但是你也不可以修改 main()函数的名字。如果修改了 main()的名字，比如改为 mymain，连接器就会报告类似的连接时错误：“unresolved external symbol \_main”。这是因为 C++/C 语言实现有一个启动函数，例如 MS C++/C 应用程序的启动函数为 mainCRTStartup()或者 WinMainCRTStartup()，同时在该函数的末尾调用了 main()或者 WinMain()，然后以它们的返回值为参数调用库函数 exit()，因此也就默认了 main()应该作为它的连接对象，如果找不到这样一个函数定义，自然会报错了。如此看来，main()其实就是一个回调函数。main()由我们来实现，但是不需要我们提供它的原型，因为我们并不能在自己的程序中调用它，这又和普通的回调函数有所不同。

基于应用程序框架（Application Framework，例如 MFC）生成的源代码中往往找不到 main()，这并不是说这样的程序中就不需要 main()，而是应用程序框架把 main()的实现隐藏起来了，并且它的实现具有固定的模式，所以不需要程序员来编写。在应用程序的连接阶段，框架会将包含 main()实现的 library 加进来一起连接。

由于 main()函数如此重要，C++标准特别规定了标准 main()函数的原型（参见 ISO/IEC 14882:1998 3.6.1 节）：

---

“...It shall have a return type of type `int`, but otherwise its type is implementation-defined. All implementations shall allow both of the following definitions of `main()` :

```
int main() { /* ..... */ }
```

and

```
int main( int argc, char *argv[] ) { /* ..... */ }
```

...It is recommended that any further(optional) parameters be added after `argv`.”

“...`main()`应该返回 `int`, 但是具体返回什么类型可以由实现来定义 (注: 即可由实现来扩展)。不过所有实现版本都应该至少允许下面两种形式的 `main()` 函数:

```
int main() { /* ..... */ }
```

和

```
int main( int argc, char *argv[] ) { /* ..... */ }
```

...并允许实现在参数 `argv` 后面增加任何需要的也是可选的参数 (注: 这也是可扩展的)。”

---

也就是说, 上述两种形式是最具有可移植性的正确写法, 其他形式都是特定实现的扩展形式, 比如 MS C++/C 允许 `main()` 返回 `void`, 以及增加第三个参数 `char* env[]` 等。读者可参考编译器的帮助文档, 以了解当前的编译器支持怎样的扩展形式。

关于 `main()` 函数的返回值问题, C++ 标准如是说:

---

“...A return statement in `main()` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `exit()` with the return value as the argument. If control reaches the end of `main()` without encountering a return statement, the effect is that of executing

```
return 0;”
```

“...`main()`中的 `return` 语句的作用是离开 `main()` (返回到 C 运行时库的启动模块, 并启动销毁过程, 销毁任何具有自动存储生命期的对象), 就像其他函数一样, 并且用其返回值作为参数调用 `exit()` 返回操作系统。如果控制到达 `main()` 的结尾, 却没有遇到任何 `return` 语句, 则效果相当于执行一条 `return 0;` 语句。”

---

当 `main()` 返回 `int` 类型时, 不同的返回值具有不同的含义。当返回 0 时, 表示程序正常结束; 返回任何非 0 值表示错误或者非正常退出。`exit()` 用 `main()` 的返回值作为返回操作系统的代码, 以指示程序执行的结果 (当然你也可以在 `main()` 或其他函数内直接调用 `exit()` 来结束程序)。

特别地, C++ 标准对 `main()` 有几个不同于一般函数的限制:

(1) 不能重载;



- (2) 不能内联;
- (3) 不能定义为静态的;
- (4) 不能取其地址;
- (5) 不能由用户自己调用;

.....

### 4.1.2 命令行参数

我们可能希望可执行程序具有处理命令行参数的能力, 例如常用的“dir X:\document /p /w”等 DOS 或 UNIX 命令。标准 C++/C 规定, 可以在 main() 函数中添加形式参数以接收程序在启动时从命令行中输入的各个参数。这里需注意, 不要把程序启动时的“命令行参数”与调用 main() 的“函数实参”的概念混淆了, 命令行参数是由启动程序截获并打包成字符串数组后传递给 main() 的一个形参 argv 的, 而包括命令字 (即可执行文件名称) 在内的所有参数的个数则被传递给形参 argc。

以上所述, 包括 main() 的连接规范 (Linkage Specification) 和调用约定 (Calling Convention) 在内, 不同的语言实现很可能是不同的。具体可参考编译器文档或者 C Runtime Library 和类库的帮助文档, 甚至类库的源代码也可拿来读一读 (例如 Visual C++ 的 crtexe.c 和 internal.h 等)。

示例 4-1 是一个文件拷贝的程序, 和 DOS 内部命令 copy 的功能一样。

示例 4-1

---

```
// mycopy.c : copy file to a specified destination file.
#include <stdio.h>
int main(int argCount, char* argValue[])
{
    FILE *srcFile = 0, *destFile = 0;
    int ch = 0;
    if (argCount != 3) {
        printf("Usage: %s src-file-name dest-file-name\n", argValue[0]);
    } else {
        if ((srcFile = fopen(argValue[1], "r")) == 0) {
            printf("Can not open source file \"%s\" !", argValue[1]);
        } else {
            if ((destFile = fopen(argValue[2], "w")) == 0) {
                printf("Can not open destination file \"%s\"!", argValue[2]);
                fclose(srcFile); /*!!!*/
            } else {
                while((ch = fgetc(srcFile)) != EOF) fputc(ch, destFile);
                printf("Successful to copy a file!\n");
                fclose(srcFile); /*!!!*/
                fclose(destFile); /*!!!*/
            }
        }
    }
}
```

```
        return 0;        /*!!!*/
    }
}
return 1;
}
```

// 用法示例:

```
mycopy C:\file1.dat C:\newfile.dat
```

如果你还不了解文件操作，没有关系，你不妨输入这个程序，把源文件名改为 `mycopy.c` (或 `.cpp`)，编译连接。在 DOS 命令行方式下随便找几个文件测试一下，是不是很有成就感呢？

## 4.1.3 内部名称

请注意 4.1.1 节的连接错误信息中的 “`_main`”，这是编译器为 `main` 生成的内部名称。C 和 C++ 语言实现都会按照特定的规则把用户（指程序员）定义的标识符（各种函数、变量、类型及名字空间等）转换为相应的内部名称。当然，这些内部名称的命名方法还与用户为它们指定的连接规范有关，比如如果使用 C 的连接规范，则 `main` 的内部名称就是 `_main`。

内部名称是否多此一举呢？非也！

在 C 语言中，所有函数不是局部于编译单元（文件作用域）的 `static` 函数，就是具有 `extern` 连接类型和 `global` 作用域的全局函数，因此除了两个分别位于不同编译单元中的 `static` 函数可以同名外，全局函数是不能同名的；全局变量也是同样的道理。其原因是 C 语言采用了一种及其简单的函数名称区分规则：仅在所有函数名的前面添加前缀 “`_`”，从唯一识别函数的作用上来说，实际上和不添加前缀没什么不同。

但是，C++ 语言允许用户在不同的作用域中定义同名的函数、类型、变量等，这些作用域不仅仅限于编译单元，还包括 `class`、`struct`、`union`、`namespace` 等；甚至在同一个作用域中也可定义同名的函数，即 **重载函数**。那么编译器和连接器如何来区分这些同名且又都会在同一编译单元中被引用的程序元素呢？在示例 4-2 中，假设下面的两个类都定义在同一个作用域中，且都定义了名为 `foo` 的成员函数。

示例 4-2

```
class Sample_1
{
    char m_name[16];
public:
    void foo(char *newName);
    void foo(int age);
};
```

```
class Sample_2
{
    char m_name[16];
public:
    void foo(char *newName);
    void foo(bool sex);
};
```

由于成员函数并不属于某一个对象，那么编译器如何区分下面这些语句分别调用的是哪个函数呢？

```
Sample_1 a;  
Sample_2 b;  
a.foo("aaa");  
a.foo(100);  
b.foo("bbb");  
b.foo(false);
```

你也许会说“通过它们各自的对象和成员标识符就可以区分”。是的，你说得没错，但这只是源代码级或者说是形式上的区分。在连接器看来，所有函数都是全局函数，能够用来区分不同函数调用的除了作用域外就是函数名称了。但是，上面的调用显然都是合理合法的。因此，如果不对它们进行重命名，就会导致连接二义性。在 C++ 中，重命名称为“**Name-Mangling**”（名字修饰或名字改编）。例如在它们的前面分别添加所属各级作用域的名称（class、namespace 等）及重载函数的经过编码的参数信息（参数类型和个数等）作为前缀或者后缀，产生全局名字 Sample\_1\_foo@pch@1、Sample\_1\_foo@int@1、Sample\_2\_foo@pch@1 和 Sample\_2\_foo@int@1，这样就可以区分了。关于这方面更详细的信息请参考 Lippman 的《Inside The C++ Object Model》相关章节，你也可以从 MS C++/C 编译器输出的 MAP 文件了解一下它所 Mangling 出来的函数的内部名称。

另外，标准 C++ 的不同实现会采取不同的 Name-Mangling 方案（标准没有强制规定），这正是导致不同语言实现之间的连接器不能兼容的原因之一。

#### 4.1.4 连接规范

在使用不同编程语言进行软件联合开发的时候，需要统一全局函数、全局变量、全局常量、数据类型等的连接规范（Linkage Specification），特别是在不同模块之间共享的接口定义部分。为什么呢？因为连接规范关系到编译器采用什么样的 Name-Mangling 方案来重命名这些标识符的问题，而如果同一个标识符在不同的编译单元或模块中具有不一致的连接规范，就会产生不一致的内部名称，这肯定会导致程序连接失败。

同样道理，在开发程序库的时候，明确连接规范也是必须要遵循的一条规则。通用的连接规范则属 C 连接规范：extern “C”，其使用方法如下。

- (1) 如果是仅对一个类型、函数、变量或常量等指定连接规范：

```
extern "C" void WinMainCRTStartup();  
extern "C" const CLSID CLSID_DataConverter;  
extern "C" struct Student{.....};  
extern "C" Student g_Student;
```

- (2) 如果是对一段代码限定连接规范：

```
#ifdef __cplusplus  
extern "C" {  
#endif  
const int MAX_AGE = 200;  
#pragma pack(push, 4)
```

```
typedef struct _Person
{
    char *m_Name;
    int   m_Age;
} Person, *PersonPtr;
#pragma pack(pop)
Person g_Me;
int     __cdecl memcmp (const void *, const void *, size_t);
void *  __cdecl memcpy (void *, const void *, size_t);
void *  __cdecl memset (void *, int, size_t);
#ifdef __cplusplus
}
#endif
```

(3) 如果当前使用的是 C++ 编译器, 并且使用了 `extern "C"` 来限定一段代码的连接规范, 但是又想令其中某行或某段代码保持 C++ 的连接规范, 则可以编写如下代码 (具体要看你的编译器是否支持 `extern "C++"`):

```
#ifdef __cplusplus
extern "C" {
#endif
const int MAX_AGE = 200;
#pragma pack(push, 4)
typedef struct _Person
{
    char *m_Name;
    int   m_Age;
} Person, *PersonPtr;
#pragma pack(pop)
Person g_Me;

#if _SUPPORT_EXTERN_CPP_
extern "C++" {
#endif
int     __cdecl memcmp (const void *, const void *, size_t);
void *  __cdecl memcpy (void *, const void *, size_t);
#if _SUPPORT_EXTERN_CPP_
}
#endif
void *  __cdecl memset (void *, int, size_t);
#ifdef __cplusplus
}
#endif
```

(4) 如果在某个声明中指定了某个标识符的连接规范为 `extern "C"`, 那么也要为其对应的定义指定 `extern "C"` 连接规范, 如下所示:

```
#ifdef __cplusplus
```

```
extern "C" {  
#endif  
int __cdecl memcmp (const void *, const void *, size_t); // 声明  
#ifdef __cplusplus  
}  
#endif  
#ifdef __cplusplus  
extern "C" {  
#endif  
int __cdecl memcmp (const void *p, const void *a, size_t len)  
{  
    ..... // 功能实现  
}  
#ifdef __cplusplus  
}  
#endif
```

但是对 COM 接口方法（Interface Methods，Interface 中的 pure virtual functions）使用的 C 复合数据类型来说（它们也是 COM 对象接口的组成部分），是否采用统一的连接规范，对 COM 对象及组件的二进制数据兼容性和可移植性都没有影响。因为即使接口两端（COM 接口实现端和接口调用端）对接口数据类型的内部命名不同，只要它们使用了一致的成员对齐和排列方式、一致的调用规范、一致的 virtual function 实现方式，总之就是一致的 C++ 对象模型，并且保证 COM 组件升级时不改变原来的接口和数据类型定义，则所有方法的运行时绑定和参数传递都不会存在问题（所有方法的调用都被转换为通过对象指针对 vptr 和 vtable 及函数指针的访问和调用，这种间接性不再需要任何方法名即函数名的参与，而接口名和方法名只是为了让客户端的代码能够顺利通过编译，但是连接时就全部不再需要了）。

#### 4.1.5 变量及其初始化

变量就是用来保存数据的程序元素，它是内存单元的别名，取一个变量的值就是读取其内存单元中存放的值，而写一个变量就是把值写入到它代表的内存单元中。在 C++/C 中，全局变量（extern 或 static 的）存放在程序的静态数据区中，在程序进入 main() 之前创建，在 main() 结束之后销毁，因此在我们的代码中根本没有机会初始化它们，于是语言及其实现就提供了一个默认的全局初始化器 0。如果你没有明确地给全局变量提供初值，编译器会自动地将 0 转换为所需要的类型来初始化它们。函数内的 static 局部变量和类的 static 数据成员都具有 static 存储类型，因此最终被移到程序的静态数据区中，因此也会默认初始化为 0，除非你明确地提供了初值。但是自动变量的初始化则是程序员的责任，因为它们是运行时在堆栈上创建的并且可以在运行时由程序员来初始化的，不要指望编译器会给它一个默认的初值。

**全局变量的声明和定义应当放在源文件的开头位置。**

**【提示 4-1】：**要区分初始化和赋值的不同。前者发生在对象（变量）创建的同时，而后者是在对象创建后进行的。要区分什么是编译器的责任，什么是程序



员的责任,不可错把程序员的责任推给编译器,否则结果可能出乎意料!例如全局变量的初始化、数据类型的隐式转换、类的隐含成员的初始化等都是编译器的责任,而局部变量的初始化、强制类型转换、类的非静态数据成员的初始化等都是程序员的责任。

在一个编译单元中定义的全局变量的初始值不要依赖定义于另一个编译单元中的全局变量的初始值。这是因为:虽然编译器和连接器可以决定同一个编译单元中定义的全局变量的初始化顺序保持与它们定义的先后顺序一致,但是却无法决定当两个编译单元连接在一起时哪一个的全局变量的初始化先于另一个编译单元的全局变量的初始化。也就是说,这一次编译连接和下一次编译连接很可能使不同编译单元之间的全局变量的初始化顺序发生改变。例如:

<pre>// file1.c int g_x = 100; .....</pre>	<pre>// file2.c double g_d = 3.14159; .....</pre>
--	---

当这两个文件编译完成并连接时,在最后的可执行程序启动时,到底是先初始化 `g_x` 还是先初始化 `g_d` 呢?答案是:我们无法预料,连接器也不会给你保证一个顺序!所以下面的做法是不当的:

<pre>// file1.c int g_x = 100; .....</pre>	<pre>// file2.c extern int g_x; double g_d = g_x + 10; .....</pre>
--	--

如果 `g_x` 初始化被排在 `g_d` 的前面,那么 `g_d` 就会被初始化为 110;但是如果反过来,那么 `g_d` 的初始值就无法预料了。

## 4.1.6 C Runtime Library

一般来说,一个 C++/C 程序不可能不使用 C 运行时库,即使你没有显式地调用其中的函数也可能间接地调用,只是我们平时没有在意罢了。例如启动函数、I/O 系统函数、存储管理、RTTI、动态决议、动态链接库 (DLL) 等都会调用 C 运行时库中的函数。我们在每一个程序开头包含的 `stdio.h` 头文件中的许多 I/O 函数就是它的一部分。C 运行时库有多线程版和单线程版,开发多线程应用程序时应该使用多线程版本的库,仅在开发单线程程序时才使用单线程版本。另外,同一软件的不同模块最好使用一致的运行时库,否则会出现连接问题。

## 4.1.7 编译时和运行时的不同

我们把编译预处理器、编译器和连接器工作的阶段合称“编译时”。语言中有些构造仅在编译时起作用,而有些构造则是在“运行时”起作用的,分清楚这些构造对于程序设计很重要。例如预编译伪指令、类 (型) 定义、外部对象声明、函数原型、标识符、各种修饰符号 (`const`、`static` 等) 及类成员的访问说明符 (`public`、`private`、

protected) 和连接规范、调用规范等仅在编译器进行语法检查、语义检查和生成目标文件(.obj 或.o 文件)及连接的时候起作用的,在可执行程序中不存在这些东西。容器越界访问、虚函数动态决议、函数动态连接、动态内存分配、异常处理和 RTTI 等则是在运行时才会出现和发挥作用的,因此运行时出现的程序问题大多与这些构造有关。

我们举两个例子来说明编译时和运行时的不同,见示例 4-3 和示例 4-4。

示例 4-3

---

```
int *pInt = new int[10];
pInt += 100;           // 越界,但是还没有形成越界访问
cout << *pInt << endl; // 越界访问!可能行,也可能不行!
*pInt = 1000;          // 越界访问!即使偶尔不出问题,但不能确保永远不出问题!
```

---

上述代码在编译时绝对没有问题,但是运行时会出现错误!千万不要写出这样的代码来!

示例 4-4

---

```
class Base {
public:
    virtual void Say(){ cout<< "Base::Say() was invoked!\n"; }
};
class Derived : public Base {
private:    // 改变访问权限,合法但不是好风格!
    virtual void Say(){ cout << "Derived::Say() was invoked!\n"; }

};
// 测试
Base *p = new Derived;
p->Say();    // 输出: Derived::Say() was invoked!
            // 出乎意料地绑定到了一个 private 函数身上!
```

---

示例 4-4 在编译时没有问题,在运行时也不会出现错误,但是违背 private 的用意。

我们在程序设计时就要对运行时的行为有所预见,通过编译连接的程序在运行时不见得就是正确的。虽然你能够一时“欺骗”编译器(因为编译器还不够聪明),但是由此造成的后果要你自己来承担。这里我们引用 Bjarne Stroustrup 的一段话来说明这一问题:“C++的访问控制策略是为了防止意外事件而不是防止对编译器的故意欺骗。任何程序设计语言,只要它支持对原始存储器的直接访问(例如 C++的指针),就会使数据处于一种开放的状态,使所有有意按照某种违反数据项原有类型安全规则所描述的方式去触动它的企图都能够实现,除非该数据项受到操作系统的直接保护。”

## 4.1.8 编译单元和独立编译技术

语言实现和开发环境支持的独立编译技术并非语言本身所规定的。每一个源代码文件（源文件及其递归包含的所有头文件展开）就是一个最小的编译单元，每一个编译单元可以独立编译而不需要知道其他编译单元的存在及其编译结果。例如，一个编译单元在单独编译的时候根本无法知道另一个编译单元在编译的时候是否已经定义了一个同名的 `extern` 全局变量或全局函数，所以每个编译单元都能够通过编译，但是如果另一个编译单元也定义了同名的 `extern` 全局变量或全局函数，那么当把两个目标文件连接到一起的时候就会出错。

独立编译技术最大的好处就是公开接口而隐藏实现，并可以创建预定义的二进制可重用程序库（函数库、类库、组件库等），在需要的时候用连接器把用户代码与库代码连接成可执行程序。另一方面，独立编译技术可以大大减少代码修改后重新编译的时间。

## 4.2 基本数据类型和内存映像

数据类型用来定义变量的值的类型，每种数据类型对应特定的字节数。例如在 32 位操作系统上，`int` 类型的变量就拥有 4 字节的内存单元，而 `double` 类型的变量占据 8 字节的内存单元。

### 【提示 4-2】:

字节是内存编址的最小单位。因为语言必须支持对一个变元（基本类型或复合类型的变量或对象）进行取地址运算（&），而且这个地址必须是有效的内存单元的地址，所以最小的对象（包括空对象）也至少会占据 1 字节的内存空间。请使用 `sizeof()` 确定数据类型在不同系统上的大小。

标准 C 语言支持的基本（内建）数据类型有 `int`、`long`、`float`、`double`、`char`、`void`，以及它们和 `signed`、`unsigned`、`*`、`&` 等的组合（有些组合是不支持的，例如 `void&`）。标准 C++ 在这些类型的基础上增加了 `bool` 类型，并同时增加了两个内置的符号常量 `true` 和 `false`（关键字）。

`void` 是“空”类型（无值型），意思是这种类型的大小无法确定。显然不存在 `void` 类型的对象，所以你也就不可能声明 `void` 类型的对象或是将 `sizeof()` 运算符用于 `void` 类型，C++/C 语言不能对一个大小未知的对象直接操作。`void` 通常用来定义函数的返回类型、参数列表（无参）或者 `void` 指针。`void` 指针可以作为通用指针，因为它可以指向任何类型的对象。

### 【提示 4-3】:

注意要分清 `void` 类型指针和 `NULL` 指针值之间的区别。`NULL` 是可以赋值给任何类型指针的值 0，在 C 语言环境中它的类型为 `void*`，而在标准 C++ 语言环境中由于允许从 0 到任何指针类型的隐式转型，因此 `NULL` 就是整数 0。即：

```
#ifdef __cplusplus
```

```

#define NULL    0
#else
#define NULL    ((void *)0)
#endif

```

一个 `void*` 类型的指针是一个合法的指针，常用于函数参数中来传递一个函数与其调用者之间约定好类型的对象地址，例如在线程函数中；而一个值等于 `NULL` 的指针虽也是一个合法的指针，但不是一个有效的指针。

虽然 `bool` 类型的变量只存在两种可能的值：`true` 和 `false`，按理说只需要一个 bit 就可以表示了。但是字节是内存编址的最小单位，而计算机从内存中提取一个变量的值是通过其地址进行的，所以一个 `bool` 变量也占据 1 字节内存，即 `sizeof(bool)` 等于 1，浪费了 7 bit。

标准 C 语言中没有 `bool` 类型，但是某些实现通过库提供了其映射，并且定义了相应的常量。例如：

```

typedef int  BOOL;
#define TRUE  1
#define FALSE 0

```

在标准 C 中，`int` 为默认类型，也就是说如果你不明确指定函数的形参类型或函数的返回值类型，则它们的类型为 `int`。标准 C++ 不支持默认类型，但是在模板中有“默认类型参数”的概念。

**【提示 4-4】：** 无论是 C 还是 C++ 程序，都不要使用默认数据类型。一定要明确指出函数每一个形参的类型和返回值类型。

某些基于 RISC（精简指令集计算机）的 CPU 比如 SPARC、PowerPC 等，对内存中基本数据类型的变量采用高字节（BYTE）和高字（WORD）在低地址存放、低字节和低字在高地址存放的 Big Endian 存储格式（即高字节、高字在前，或地址大的字节结尾），并且把最高字节的地址作为变量的首地址。在这种自然的存储格式中，要求变量在内存中的存放位置必须自然对齐，否则 CPU 会报告异常。所谓自然对齐，就是基本数据类型（主要是 `short`、`int` 和 `double`）的变量不能简单地存储于内存中的任意地址处，它们的起始地址必须能够被它们的大小整除。例如：在 32 位平台下，`int` 和指针类型变量的地址应该能被 4 整除，而 `short` 变量的地址都应该是偶数，`bool` 和 `char` 则没有特别要求。所以，基于这种 CPU 架构的平台，编译器将按照自然对齐的要求来为每个变量生成逻辑地址，C++/C 编译器亦如此。例如 `short` 型变量 `x` 和 `int` 型变量 `y` 的内存布局及其首地址如图 4-1 所示。

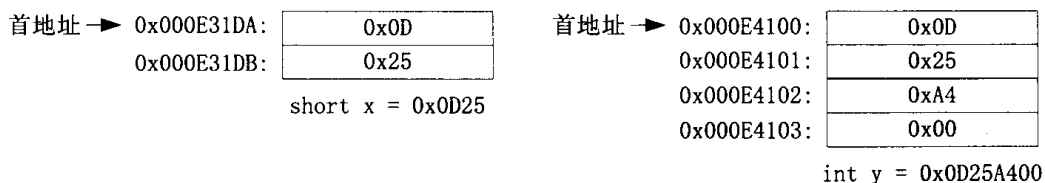


图 4-1 Big Endian 和自然对齐

Intel 系列 CPU 采用 Little Endian 存储格式来存放基本类型变量，即低字节和低

字在低地址存放、高字节和高字在高地址存放（即低字节、低字在前，或地址小的字节结尾），并且把最低字节的地址作为变量的首地址。在这种硬件平台上，上述两个变量  $x$  和  $y$  在内存中的布局将如图 4-2 所示。

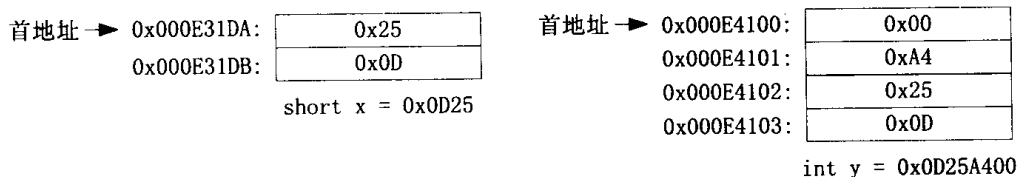


图 4-2 Little Endian 和自然对齐

在 Intel 系列 CPU 这种硬件平台上，并不要求基本类型变量在内存中必须自然对齐，同样也不会要求复合类型变量必须自然对齐。如果变量没有自然对齐，可能会在一定程度上降低 CPU 访问该变量的性能，但并不会影响程序的正确性。

对于任意类型的数组，我们知道下面的断言都应该是真的：

```
SomeType x, y[100];
assert(sizeof(y) == sizeof(x) * 100);    // true
```

编译器必须确保不仅第一个对象元素要自然对齐，而且以后的每一个对象元素也要对齐才行，所以数组的自然对齐要求和单个元素的对齐要求是一样的。关于复合数据类型的对齐问题，我们将在本书 8.1.4 节详细讲解。

## 4.3 类型转换

当 C 程序中某个地方所需要的数据类型并非你所提供的类型时，通常要进行数据类型转换，这是 C 语言强制类型检查机制的具体表现。例如在一个同时出现了不同类型的操作数的复合算术表达式中，一般占用内存较少的类型会隐式地转换为表达式中占用内存最多的操作数类型。C++ 也是一样，甚至提供了比 C 更严格的静态类型检查系统。

**【提示 4.5】：**从本质上来说，C++/C 不会直接对两个类型不同的操作数进行运算，如果操作数类型不同，编译器就会试图运用隐式类型转换规则或者按照用户要求进行强制类型转换。类型转换并不是改变原来的类型和值，而是生成了新的临时变元，其类型为目标类型。

### 4.3.1 隐式转换

所谓隐式转换，就是编译器在背后帮程序员做的类型转换工作，程序员往往察觉不到。既然是编译器自动进行的，那么这种类型转换必须具有足够的安全性，这是编译器的责任。反过来，凡是不安全的类型转换，编译器都应该能够检查出来并给出错误或警告信息，而不会默默地执行。如果程序员确实想做这样的转换，那么

需要显式地使用强制类型转换，由此可能造成的安全隐患由程序员负责。

这里安全性主要包括两个方面：内存单元访问的安全性和转换结果的安全性，主要表现为内存访问范围的扩张、内存的截断、尾数的截断、值的改变和溢出等。

下面我们具体分析一下基本数据类型之间的转换，以及 C++ 基类和派生类之间的转换中可能出现的安全性问题。

基本数据类型之间存在如下的兼容关系：char is-a int、int is-a long、long is-a float、float is-a double，并且 is-a 关系是传递的。但是存在于基本数据类型之间的 is-a 关系不同于 C++ 派生类和基类之间的 is-a 关系，因为一个高级基本数据类型（占据内存字节多的数据类型）并不是由一个或多个低级基本数据类型（占据内存字节少的数据类型）子对象构造而成的，一个低级基本数据类型也不是派生自多个高级基本数据类型的。

一个低级数据类型对象总是优先转换为能够容纳得下它的最大值的、占用内存最少的高级类型对象。例如 100 这个字面常量（类型为 int）如果转换为 long 型就能满足编译器的要求，那就不会转换为 double 型。比如下面的两个重载函数：

```
void f(long l);  
void f(double d);
```

如果存在调用 f(100)，则必然调用 f(long l) 而不是 f(double d)，除非不存在 f(long l) 才会连接到 f(double d)。

示例 4-5 中的转换是安全的，并不需要强制。编译器首先隐式地将 100 提升为 double（作为它的整数部分）的一个临时变量，然后将这个临时变量赋值给 d1；同样，i 也会首先隐式地提升为 double（其值作为它的整数部分）的一个临时变量，然后才赋值给 d2。当编译器认为这些临时变量不再需要时就会适时地把它们销毁。

示例 4-5

```
double d1 = 100;  
int i = 100;  
double d2 = i;
```

由于派生类和基类之间的 is-a 关系，可以直接将派生类对象转换为基类对象，这虽然会发生内存截断，但是无论从内存访问还是从转换结果来说都是安全的。这得益于 C++ 的一个保证：派生类对象必须保证其基类子对象的完整性，即其中的基类子对象的内存映像必须和真正的基类对象的内存映像一致，如图 4-3 所示。程序见示例 4-6。

示例 4-6

```
class Base  
{  
private:  
    int m_a;  
    int m_b;
```

```
// 示例  
Derived objD1;  
Base objB1 = objD1; // 见图 4-3 左图
```



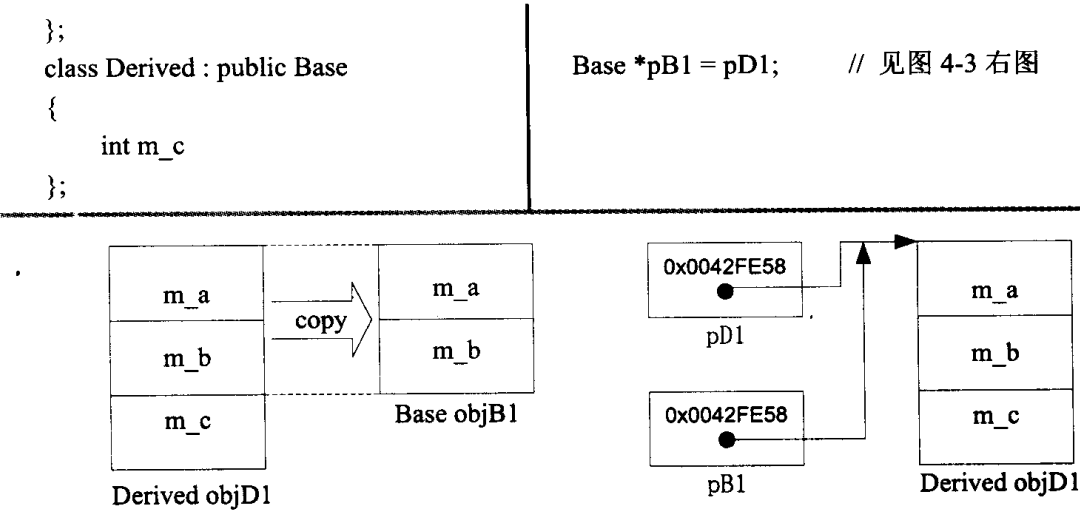


图 4-3 基类和派生类之间的隐式转换

**【提示 4-6】:** 标准 C 语言允许任何非 void 类型指针和 void 类型指针之间进行直接的相互转换。但在 C++ 中，可以把任何类型的指针直接指派给 void 类型指针，因为 void\* 是一种通用指针；但是不能反过来将 void 类型指针直接指派给任何非 void 类型指针，除非进行强制转换。因此，在 C 语言环境中我们就可以先把一种具体类型的指针如 int\* 转换为 void\* 类型，然后再把 void\* 类型转换为 double\* 类型，而编译器不会认为这是错误的。然而这样的做法确实存在着不易察觉的安全问题（内存扩张和截断等），这是标准 C 语言的一个缺陷。

4.3.2 强制转换

我们来看看强制类型转换可能导致的安全问题。首先来看基本数据类型及其指针的转换，见示例 4-7。

示例 4-7

```
double d3 = 1.25e+20;
double d4 = 10.25;
int i2 = (int)d3;
int i3 = (int)d4;
```

按照从浮点数到整型数的转换语义，结果应该是截去浮点数的小数部分而保留其整数部分，因此 i3 会得到 10，而 i2 会溢出，因为 d3 的整数部分远远超出了一个 int 所能表示的范围，结果当然不是我们所期望的。

基本数据类型之间的指针转换一般说来必然会造成内存截断或内存访问范围的扩张，除非两种类型具有相同的字节大小。例如在 32 位系统中，int、long、float 都具有 4 字节的空间，虽然不会造成内存截断或内存扩张，但是它们之间的指针转换改变了编译器对指针所指向的内存单元的解释方式，因此结果必然是错误的，见示例 4-8。

示例 4-8

```
double d5 = 1000.25;
int *pInt = (int*)&d5;
int i4 = 100;
double *pDbl = (double*)&i4;
```

从内存访问角度来说,你通过 `pInt` 访问它指向的 `double` 类型变量 `d5` 是安全的(后面的 4 字节被“截断”了,可访问内存范围缩小),但是其值绝对不会是 `d5` 的整数部分 1000,而是位于 `d5` 开头 4 字节中的内容,并解释为 `int` 类型数,这个数是不可预料的;同样,你通过 `pDbl` 访问 `int` 类型变量 `i4`,得到的数据不一定就是 100,况且造成了可访问内存范围的“扩张”。如果你往里面写数据就会产生运行时错误,如图 4-4 所示。

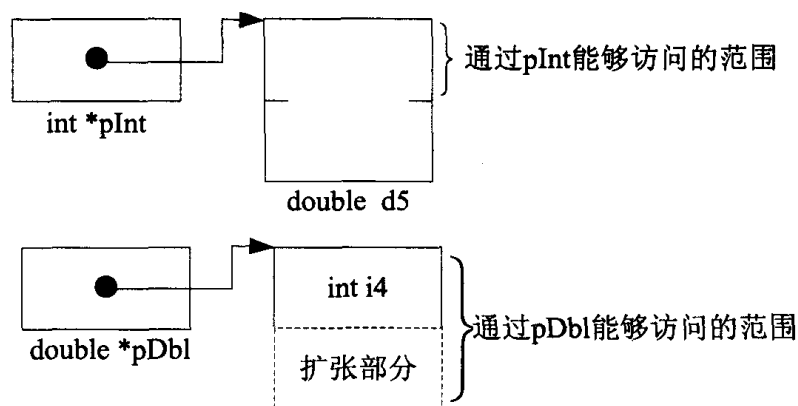


图 4-4 基本数据类型的指针之间的强制转换

C++基类和派生类之间的指针强制转换同样存在安全隐患,如示例 4-9 所示。

示例 4-9

```
Base objB2;
Derived *pD2 = (Derived*)&objB2;
```

存在的问题是:通过 `pD2` 能够访问的内存范围“扩张”了 4 字节,如果访问 `m_c` 就可能引发运行时错误,因为 `pD2` 指向的对象根本就没有成员 `m_c` 的空间,如图 4-5 所示。

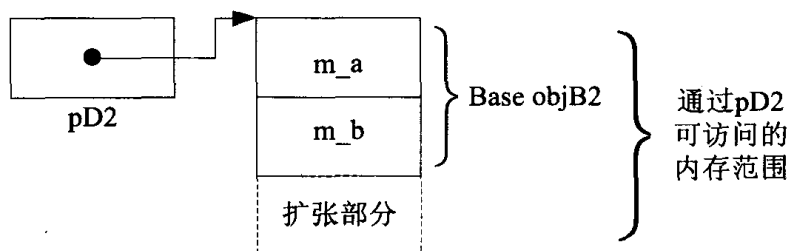


图 4-5 派生类和基类之间的指针强制转换

对于类层次结构中的转换, C++提供了新的类型转换操作符及能够起类型转换作用的函数, 我们会在后面讲到。

**【提示 4-7】:**

- (1) 不可以把基类对象直接转换为派生类对象, 无论是直接赋值还是强制转换, 因为这不是“自然的”;
- (2) 对于基本类型的强制转换一定要区分值的截断与内存截断的不同;
- (3) 如果你坚持要使用强制转换, 必须同时确保内存访问的安全性和转换结果的安全性;
- (4) 如果确信需要数据类型转换, 请尽量使用显式的(即强制)数据类型转换, 让人们知道发生了什么事, 避免让编译器静悄悄地进行隐式的数据类型转换。

**【提示 4-8】:**

尽量避免做违反编译器类型安全原则和数据保护原则的事情, 例如在有符号数和无符号数之间转换, 或者把 `const` 对象的地址指派给非 `const` 对象指针, 等等。

## 4.4 标识符

C++/C 的标识符是由字母、数字和下划线( `_` )组成的字符序列, 用来标识一个程序元素, 例如变量、函数、宏、类型名等。标识符可以任意长, 但是标准 C 语言规定, 编译器只取前 31 个字符作为有效的标识符; 而标准 C++ 则取前 255 个字符作为有效的标识符。

每一个标识符都具有如下的几个属性: 值、值的类型、名字、存储类型、作用域范围、连接类型(可见性)、生存期等。例如 C 函数, 函数名其实就是函数体代码在内存中的首地址, 在编译时就可以确定其值, 因此是一个常量, 这是它的值; 值的类型就是函数指针类型; 存储类型默认为 `extern`, 除非声明为 `static`; 作用域范围为文件作用域; 连接类型默认为外连接, 除非声明为 `static`; 生存期为永久(即静态)。其他的几个属性我们将在后面章节适当的地方详细介绍。

**【提示 4-9】:**

- (1) 避免使用前导“`_`”和“`__`”来定义你自己的标识符, 因为语言及其实现使用它来定义一些内部名称或预定义的宏, 如果你也使用它, 就有可能造成命名冲突;
- (2) 给标识符起一个有意义的名字, 要能够“望文生义”。如果是变量, 最好能体现出它的值的类型(例如使用类型名缩写作为前缀)。这样的标识符具有“自说明”能力, 具体参见本书第 11 章;
- (3) 使用长的标识符名字并不会增大可执行代码的体积, 因此不要使用过于简单的名字, 但也不要使用过长的名字。标识符名字的长度应该遵循“用最短的名字包含最多的信息量”的原则。

## 4.5 转义序列

在 C++/C 中,有些字符在程序代码中具有特殊的含义,例如“%”表示取余,在字符串中表示 I/O 格式控制,“”表示字符串的开始和结束,“?”是三元运算符 ? 的一分子,等等。那么,现在想把这些字符本身输出到终端上,尤其是当它们出现在普通字符串或格式控制字符串中的时候,就需要做一些特殊处理。一般说来有两种办法:使用转义序列或者直接引用 ASCII 码值。转义序列是用反斜线 (\) 后跟一个特定转义字符组成的。常见的转义序列见表 4-1。

表 4-1 转义序列

转 义 序 列	ASCII 码值	说 明
\'	0x27	输出单引号字符本身(可以不加\"来输出)
\"	0x22	输出双引号字符本身
\?	0x3F	输出问号字符本身(可以不加\"来输出)
\\	0x5C	输出反斜线字符本身
\a	0x07	触发扬声器发出蜂鸣声(特殊作用)
\b	0x08	使光标退一格
\f	0x0C	输出换页(特殊作用)
\n	0x0A	换行字符或功能(特殊作用)
\r	0x0D	把终端光标移到行首(回车字符或功能)(特殊作用)
\t	0x09	输出水平制表符(特殊作用)
\v	0x0B	输出垂直制表符(特殊作用)
\0	0x00	空字符(特殊作用)

其实由%引导的 I/O 格式控制字符序列,例如%d、%f、%%等,也都是转义序列。转义序列用在 I/O 格式控制字符串中才会体现出它的用途,而在其他场合下的普通字符串中并不表现其转义语义。

在字符串中可以使用“\0000”或“\xHH”来引用 ASCII 码表中的任何一个字符,其中 000 和 HH 分别表示该字符的八进制数据和十六进制数据 ASCII 码值。

### 【提示 4-10】:

要区分“换行”与“回车”的语义。首先,它们的 ASCII 码值不同。“换行”字符一般用于文件,即把从键盘输入的“回车”字符转换为“换行”字符来保存而不是直接保存“回车”字符;“换行”还用于程序的输出控制,即输出一个“换行”字符以指示终端输出从新行开始。而“回车”是键盘功能,用于输入控制,例如代替“鼠标左击”和表示输入的结束或从新行输入,它不能输出。因此要记住:输出“换行”,输入“回车”。不过有些字符输入函数可以把键盘输入的“回车”字符自动转换为“换行”字符返回,例如 getchar()。

## 4.6 运算符

C++/C 基本上有 3 种运算符：算术运算符、关系运算符和逻辑运算符，还有一些其他运算符，如函数调用、类型转换、成员选择等。C++ 新提供了几个类型转换运算符和运行时类型识别运算符（typeid），以及作用域解析（::）、动态内存分配和释放、类成员指针等运算符，我们在后面“C++运算符重载”中再介绍。运算符用来构成表达式并指示计算机执行计算，其基本特性就是优先级和结合律。在没有使用小括号确定一个复合表达式中各运算符的计算顺序的情况下，编译器将使用它们的优先级和结合律来确定计算顺序。优先级越高的运算符越先计算；相反，优先级越低的越后计算；相同优先级的运算符之间或同一运算符之间的计算顺序按照结合律来确定。

运算符和表达式都是属于 C++/C 程序的基本组成元素，它们看似简单，但使用时隐患还是比较多的。

常见运算符的优先级与结合律见表 4-2。注意一元运算符 +、-、\* 的优先级高于对应的二元运算符。

表 4-2 运算符优先级和结合律

优 先 级	运 算 符	结 合 律
从 高 到 低 排 列	() [] -> .	从左至右
	! ~ ++ -- (类型) sizeof + (取正) - (取负) * (反引用) & (取地址)	从右至左
	* / %	从左至右
	+ (加) - (减)	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
		从左至右
	&&	从左至右
		从右至左
	?:	从右至左
	= += -= *= /= %= &= ^=  = <<= >>=	从左至右
	,	从左至右

由于将上表熟记是比较困难的，为了防止产生歧义并提高可读性，应当用括号确定表达式的操作顺序。例如：

```
word = (high << 8) | low  
if ((a | b) && (a & c))
```

**【提示 4-11】** 如果代码行中的运算符比较多，用括号确定表达式中每一个子表达式的计算顺序，避免使用默认的优先级。

条件运算符“?:”是 C++/C 中惟一的三元运算符，其语法为：

条件表达式 ? 表达式 1: 表达式 2;

其语义是：如果“条件表达式”为 TRUE，则整个表达式的值就是“表达式 1”的值，“表达式 2”忽略；否则整个表达式的值就是“表达式 2”的值，“表达式 1”忽略。它的语义等价于如下的 if/else 结构：

```
ResultType retValue;  
if (条件表达式) {  
    retValue = 表达式 1;  
} else {  
    retValue = 表达式 2;  
}
```

**【提示 4-12】** 当单独对一个变量使用++、--运算符时，它们的前置版本和后置版本效果一样；只有当变量用在较复杂的表达式中时，它们的前置版本和后置版本才具有不同的效果。在后面的“C++运算符重载”中我们再详细讨论它们的前置版本和后置版本的语义和区别。

当心那些视觉上不易分辨的操作符，以免发生书写错误。我们经常会把“==”误写成“=”，像“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

## 4.7 表达式

表达式其实是一个递归定义的概念：

- (1) 一个单独的标识符（变量、常量、函数等）是一个表达式。
- (2) 由表达式和运算符按照语法规则构成的更加复杂的表达式也是表达式。

通俗地讲，表达式就是使用运算符和标识符（操作数）按照语法规则连接起来的算式，因此任何表达式都是有值的。常见的表达式有常量表达式、算术表达式、关系表达式和逻辑表达式及复合表达式。此外还有一些不常用的表达式，例如逗号表达式、条件运算符(?:)表达式、位运算表达式，等等。

**【提示 4-13】** 不要把数学中的表达式与计算机语言支持的表达式相混淆。例如：

```
if (a < b < c) //a < b < c 是数学表达式而不是程序表达式
```

并不表示：

```
if ((a < b) && (b < c))
```

而是成了令人费解的 if((a < b) < c)。



常量表达式就是全部由常量（字面常量、符号常量、枚举常量、布尔常量等）和运算符构成的表达式。由于常量在运行时不能改变值，所以常量表达式没有必要等到运行时才计算，编译器在编译时就可以对它求值。因此，常量表达式也可以定义为：在编译时就可以求值的表达式。

**【提示 4-14】：**能够在编译时求值的程序元素是否需要分配运行时的存储空间呢？要看它是什么类型的程序元素。例如基本数据类型的字面常量、枚举常量、sizeof()、常量表达式等就不需要分配存储空间，因此也没有存储类型；但是字符串常量、const 常量（尤其是 ADT/UDT 的 const 对象）都要分配运行时的存储空间，即有特定的存储类型。

算术表达式就是由算术运算符（+、-、\*、/、%、|、&等）和标识符构成的表达式。除了由赋值运算符（=）和算术运算符结合在一起组成的运算符外（例如=、+=、-=、\*=、/=、%=、|=、&=等），不要使用算术表达式作为单独的语句，因为你并没有保存和使用它的计算结果却要计算机白白地耗费时间去计算它。

关系表达式就是由关系运算符（>、<、>=、<=、==、!=）和标识符构成的表达式，常用来构造条件表达式。所谓条件表达式就是指示计算机执行判断从而有所选择地执行语句序列的表达式。很显然，关系运算符都是二元运算符，两个操作数才能产生关系，单独一个操作数能和谁有关系呢？关系表达式总是返回 TRUE（非 0 值）或 FALSE（0）。

逻辑表达式则是由逻辑运算符（&&、||、!）和标识符构成的表达式，其中“!”是一元运算符。逻辑运算符也常用来构造条件表达式。逻辑表达式总是返回 TRUE（非 0 值）或 FALSE（0）。

**【建议 4-1】：**在使用运算符“&&”的表达式中，要尽量把最有可能为 FALSE 的子表达式放在“&&”的左边；同样在使用运算符“||”的表达式中，要尽量把最有可能为 TRUE 的子表达式放在“||”的左边。因为 C++/C 对逻辑表达式的判断采取“突然死亡法”（猝死法）：如果“&&”左边的子表达式计算结果为 FALSE，则整个表达式就为 FALSE，后面的子表达式没有必要再计算；如果“||”左边的子表达式计算结果为 TRUE，则整个表达式就为 TRUE，因此后面的子表达式没有必要再计算。这种方法可以提高程序的执行效率。

由简单表达式通过算术的、关系的和逻辑的运算组合而成的表达式就是复合表达式（混合表达式），这是我们重点讨论的对象，因为它最有可能存在隐患。

允许复合表达式存在的理由是：

- （1）书写简洁，使用简单表达式完成相同功能需要更多周折。
- （2）生成的可执行代码更加高效。

但要防止滥用复合表达式。

**【提示 4-15】** 不要编写太复杂的复合表达式。例如：

```
i = a >= b && c < d && c + f <= g + h; // 复合表达式过于复杂
```

不要编写多用途的复合表达式。例如：

```
d = (a = b + c) + r;
```

该表达式既求 a 值又求 d 值。应该拆分为两个独立的语句：

```
a = b + c;
```

```
d = a + r;
```

需要注意的是，任何表达式都可以作为单独的语句来用，我们叫做表达式语句。例如赋值表达式也叫做赋值语句。

## 4.8 基本控制结构

早期的程序控制转移（选择、循环和无条件转移等）都是通过 goto 或者类似的语句来完成的，但是后来的事实表明毫无节制的控制权转移不仅会加大软件的开发和维护难度，而且极有可能导致程序的失控。结构化程序设计概念的出现改变了这种状况，Bohm 和 Jacopini 研究证明：任何程序只用 3 种控制结构就可以实现，它们是顺序结构、选择结构和循环结构。因此，结构化程序设计完全可以说是“无 goto”的。

顺序结构当然是使用最多的，也是最简单的控制结构，无论是在 C++/C 语言中还是在其他中高级语言中都是如此。实际上，计算机本来就是逐条执行程序语句的，因此顺序结构都是内置在语言中的，也是其他控制结构的基础。

## 4.9 选择（判断）结构

计算机在本质上就是能够执行运算和做出逻辑判断的机器，它的这种能力可以通过编程语言中的选择结构（即判断结构）表现出来。C++/C 有 3 种基本的选择结构：if 结构（单选择）、if/else 结构（双重选择）和 switch 结构（多重选择）。

if 结构和 if/else 结构的语法分别如下：

<pre>if(条件表达式) {     语句序列 }</pre>	<pre>if(条件表达式) {     语句序列 1 } else {     语句序列 2 }</pre>
-----------------------------------	---

C++/C 也支持下面的 if/else 结构：

```
if(...){...}
```

```
else if(...){...}  
else if(...){...}  
else{...}
```

因为它相当于如下的嵌套结构：

```
if(...) {  
    ...  
} else {  
    if(...) {  
        ...  
    } else {  
        if(...) {  
            ...  
        } else {  
            ...  
        }  
    }  
}
```

因为每一个 else 分支里面仅有一条 if 语句，故省略了 {}。

**【建议 4-2】：** 在 if/else 结构中，要尽量把为 TRUE 的概率较高的条件判断置于前面，这样可以提高该段程序的性能。

按理说，if 语句是 C++/C 语言中比较简单和常用的语句，然而很多程序员用隐含错误的方式书写 if 语句，最常见的就是变量与零值的比较。

## 4.9.1 布尔变量与零值比较

假设布尔变量名字为 flag，它与零值比较的标准 if 语句如下：

```
if(flag)           // 表示 flag 为真  
if(!flag)          // 表示 flag 为假
```

**【提示 4-16】：** 根据布尔类型（boolean）的语义，0 为“假”，任何非 0 值都是“真”。可用 TRUE 和 FALSE 来表示“真”和“假”两个概念。语言实现必须通过可比的值来区分二者，比如 0 和非 0 就可以担当此任。但是 TRUE 的值究竟是什么并没有统一的标准，不同的语言可能采用不同的方案。具体到 C++ 语言，标准化以前的某些实现并不支持 bool 这种内置类型，也没有 TRUE/FALSE 这两个内置常量。If 语句判断其条件表达式的真假并不是通过把它的计算结果转换为布尔类型的临时变量来进行的，而是将其结果直接和 0 进行相等性比较，如果不等于 0 则表示真，否则为假。许多语言都采用了这个比较通用的做法。标准化后的某些 C++ 实现可能对 if 语句的处理做了增强或者调整，因为现在 bool 是标准类型了。由于历史的原因，Visual C++ 将 TRUE 定义为 1，而 Visual Basic 则将 TRUE 定义为 -1。

标准 C++ 规定的 `bool` 类型常量和整数、指针等之间的转换规则如下：

```
false → 0, true → 1;  
0 → false, 任何非 0 值 → true;
```

但是不同的实现对 `true` 的表示可能不同(可能不符合标准), 因此下面这样的语句:

```
int flag = -1;  
if ( flag == true ) {}  
else {}
```

在不同的实现下行为可能不一致。但是 `false` 的值是确定的, 因此应该总是和 `false` 比较。

不要将布尔变量 `flag` 直接与 `TRUE` 或者 `1`、`-1`、`0` 等进行比较。下列 `if` 语句都属于不良用法:

```
if(flag != TRUE)    // 错误用法  
if(flag == TRUE)    // 错误用法  
if(flag == 1)       // 错误用法  
if(flag != 1)       // 错误用法  
if(flag == 0)       // 不良用法, 让人误以为 flag 是整数  
if(flag != 0)       // 不良用法, 让人误以为 flag 是整数
```

#### 4.9.2 整型变量与零值比较

假设整型变量为 `value`, 它与零值比较的标准 `if` 语句如下:

```
if(value == 0)  
if(value != 0)
```

不可以模仿 `bool` 变量的风格而写成:

```
if(value)    // 会让人误以为 value 是布尔变量  
if(!value)
```

#### 4.9.3 浮点变量与零值比较

计算机表示浮点数 (`float` 或 `double` 类型) 都有一个精度限制。对于超出了精度限制的浮点数, 计算机会把它们的精度之外的小数部分截断。因此, 本来不相等的两个浮点数在计算机中可能就变成相等的了。例如:

```
float a = 10.222222225, b = 10.222222229;
```

在数学上 `a` 和 `b` 是不相等的, 但是在 32 位计算机中它们就是相等的。

**【提示 4-17】:** 如果两个同符号浮点数之差的绝对值小于或等于某一个可接受的误差 (即精度), 就认为它们是相等的, 否则就是不相等的。精度根据具体应用要求而定。不要直接用 “=” 或 “!=” 对两个浮点数进行比较, 虽然 C++/C 语言支持直接对浮点数进行 “=” 和 “!=” 的比较操作, 但是由于它们采用的精度往往比我们实际应用中要求的精度高, 所以可能导致不符合实际需求的结果甚至错误。

假设有两个浮点变量  $x$  和  $y$ ，精度定义为  $\text{EPSILON} = 1\text{e-}6$ ，则错误的比较方式如下：

```
if(x == y)           // 隐含错误的比较
if(x != y)           // 隐含错误的比较
```

应该转化为正确的比较方式：

```
if(abs(x - y) <= EPSILON)  // x 等于 y
if(abs(x - y) > EPSILON)  // x 不等于 y
```

同理， $x$  与零值比较的正确方式为：

```
if(abs(x) <= EPSILON)      // x 等于 0
if(abs(x) > EPSILON)       // x 不等于 0
```

从数学意义上讲，两个不同的数字之间存在着无穷个实数。计算机只能区分至少 1bit 不同的两个数字，并且使用较少的位（32 或 64 位）来表示一个很大范围内的数字，因此浮点表示只能是一种近似结果。在针对实际应用环境编程时，总是有一个精度要求，而直接比较一个浮点数和另外一个值（浮点数或者整数）是否相等（==）或不等（!=）可能得不到符合实际需要的结果。

例如：假设在某光学精密仪器制造工业应用中，要求该仪器各零部件尺寸精度达到  $1\mu\text{m}$ ，即  $10^{-6}\text{m}$ 。那么下面两个数从数学意义上来说应该是不相等的：

```
d1 = 1.123 456 2(m)    d2 = 1.123 456 8(m)
```

但是在精度要求为  $10^{-6}\text{m}$  的情况下，它们应该被视为相等。而如果使用 == 和 != 对  $d1$  和  $d2$  直接进行比较，结果将可能如下：

```
if(d1 == d2).....    // FALSE
if(d1 != d2).....    // TRUE
```

原因就是这样的直接比较的精度比我们要求的  $10^{-6}\text{m}$  要高。

同样道理，一个浮点数和一个整数之间的直接判等也存在类似的偏差。例如：

```
d3 = 1.000 000 1(m)    d4 = 1(m)
```

同样在精度要求为  $10^{-6}\text{m}$  的情况下，它们应该被视为相等。而如果使用 == 和 != 对  $d3$  和  $d4$  直接进行比较，结果可能恰恰相反。

所以，在实际应用环境下，如果直接比较浮点数和另一个数（整数或浮点数）是否相等（==）或不等（!=），可能会产生错误的结果，进而导致软件出错。

直接比较浮点数和另一个数（整数或浮点数）是否相等（==）或不等（!=），其结果可能依赖于具体的编译环境和平台（如操作系统和硬件系统结构），因为每一个编译平台都有自己默认的精度，对浮点数直接进行 == 和 != 的比较采用的就是这个默认精度，而不是按照内存中两个数仅有某个 bit 不同（其他所有 bit 都相同）来判断两个数是否相等的。

例如:

```
float x = 0.0f, y = 0.0f;
...
if (abs(x - y) <= numeric_limits<float>::epsilon()) {
    // x == y
} else {
    // x != y
}

if (abs(x) <= numeric_limits<float>::epsilon()) {
    // x == 0
} else {
    // x != 0
}
```

虽然不建议直接使用==和!=比较浮点数,但是可以直接比较浮点数谁大谁小,即可将“<”和“>”直接应用于浮点数之间的比较及浮点数和整数的比较。但是,对内置类型来说,“!(a>b) && !(a<b)”与“a==b”的语义是等价的,因此在针对实际应用环境编程时也不建议使用“!(a>b) && !(a<b)”来判断浮点数相等与否。

#### 4.9.4 指针变量与零值比较

指针变量的零值是“空值”(记为 NULL),即不指向任何对象。尽管 NULL 的值与 0 相同,但是两者意义不同(类型不同):

```
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
```

假设指针变量的名字为 p,它与零值比较的标准 if 语句如下:

```
if(p == NULL)    // p 与 NULL 显式比较,强调 p 是指针变量
if(p != NULL)
```

而不要写成:

```
if(p == 0)        // 容易让人误以为 p 是整型变量
if(p != 0)
```

或者:

```
if(p)            // 容易让人误以为 p 是布尔变量
if(!p)
```



## 4.9.5 对 if 语句的补充说明

有时候我们可能会看到 `if (NULL == p)` 这样古怪的格式。不是程序写错了，而是程序员为了防止将 `if (p == NULL)` 误写成 `if (p = NULL)`，有意把 `p` 和 `NULL` 颠倒。编译器认为 `if (p = NULL)` 是合法的，但是会指出 `if (NULL = p)` 是错误的，因为 `NULL` 不能被赋值。类似的还有 `if (100 == i)` 等。

程序中有时会遇到 `if/else/return` 的组合，应该将如下不良风格的程序：

```
if(condition)
    return x;
return y;
```

改写为：

```
if(condition) {
    return x;
} else {
    return y;
}
```

或者改写成更加简练的：

```
return condition ? x : y;
```

## 4.9.6 switch 结构

有了 `if/else` 语句为什么还要 `switch` 语句？

`switch` 是多分支选择语句，而 `if/else` 语句只有两个分支可供选择。虽然可以用嵌套的 `if` 语句来实现多分支选择，但那样的程序冗长难读，更重要的是可能在匹配到某一个分支前执行多次无谓的比较。相反，`switch` 的效率比 `if/else` 结构高，这正是 `switch` 语句存在的理由。

`switch` 语句的基本格式是：

---

```
switch (表达式)
{
    case 常量表达式 1:
        语句序列
        break;
    case 常量表达式 2:
        语句序列
        break;
    ...
    default :
        ...
        break;
}
```

---

- 【提示 4-18】:** (1) switch 没有自动跳出的功能, 每个 case 子句的结尾不要忘了加上 break, 否则当表达式与某一个 case 子句匹配并执行完它的语句序列后, 将接着执行下面 case 子句的语句序列, 这就导致了多个分支重叠 (除非有意让多个分支共享一段代码)。break 语句只是一个“jmp”指令, 其作用就是跳到 switch 结构的结尾处;
- (2) 不要忘记最后那个 default 子句。即使程序真的不需要 default 处理, 也应该保留语句 default : break;, 这样做并非多此一举, 而是为了防止别人误以为你忘了 default 处理, 以及出于清晰性和完整性的考虑。

我们举一个选择结构的例子 (见示例 4-10): 输入一个年份, 判断是否为闰年。判断闰年的方法是: 如果该年能被 4 整除但不能被 100 整除, 或者能被 400 整除, 就是闰年。

示例 4-10

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    unsigned long year;
    printf("Input a year : ");
    scanf("%lu", &year);
    if ( (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0) ) {
        printf("Year %lu is a leap year.\n", year);
    } else {
        printf("Year %lu is not a leap year.\n", year);
    }
    return 0;
}
```

## 4.10 循环 (重复) 结构

循环结构在本质上是一种包含选择结构 (循环条件的判断) 的重复执行的一组语句序列。标准 C++/C 语言提供了 3 种循环结构: do/while、while 和 for。循环结构基本上可以分为确定循环、不确定循环和无限循环, 它们分别又可以叫做计数器控制的循环、标志控制的循环和死循环 (busy-loop)。

这 3 种循环结构的基本语法如下:

do	while (条件表达式)	for ( 循环控制变量初始化表达式; 循环控制条件表达式; 循环控制变量修改语句)
{ 语句序列 } while (条件表达式);	{ 语句序列 }	{ 语句序列 }

它们都在条件表达式为 TRUE (非 0 值) 时执行体内的语句序列。但是 do/while 结构在执行语句序列之后才测试条件表达式的真假, 因此它至少要执行体内的语句序列一次, 除非像下面这样来用它:

```
do {  
    if(条件表达式 is FALSE) break;  
    语句序列  
} while (条件表达式);
```

但是我们强烈建议不要这样使用。

另外两个结构在一开始就判断条件表达式的值, 如果为 TRUE 则执行体内语句序列, 否则退出循环。

for 结构等价于下面的 while 结构:

```
循环控制变量初始化语句;  
while (循环控制条件表达式) {  
    if (循环控制变量不等于其初始值)  
        修改循环控制变量;  
    语句序列  
}
```

如果循环体中出现了 continue 语句, 则要防止它跳过循环控制变量的修改语句。for 结构中把循环控制变量的修改放在前面而不是后面就是这个道理。

可以使用它们中的任何一种来编写确定循环或不确定循环, 但是我们建议: 如果你的循环是确定的, 最好使用 for 结构, 否则使用 while 结构, do/while 结构不常用。确定循环还有一个不常见的用法: 起到延迟作用。

**【提示 4-19】:** 标准 C++/C 语言允许在 for 语句中声明 (定义) 变量, 并且它的作用范围为该 for 语句块内。不过有些语言实现可能没有遵循这个语义, 例如 Visual C++ 就会把这样声明的变量挪到 for 语句块外, 从而它的作用域与 for 语句相同, 在 for 语句块结束后它仍然有效。这一点要引起注意。

## 4.10.1 for 语句的循环控制变量

不要使用浮点数做计数器来控制循环, 因为它可能是不精确的。

不要在循环体内修改循环变量——除了循环控制变量的递增递减, 防止循环失去控制, 尤其是 for 循环。在后面讲到容器的时候也有一条类似的规则: 不要在遍历 (迭代) 容器的过程中对容器进行插入、删除元素的操作。

**【建议 4-3】:** 如果计数器从 0 开始计数, 则建议 for 语句的循环控制变量的取值采用“前闭后开区间”写法。要防止出现“差 1”错误。

例如示例 4-11 中的两种风格:

示例 4-11

<pre>for(int x = 0; x &lt; N; x++) {     ... }</pre>	<pre>for(int y = 0; y &lt;= N-1; y++) {     ... }</pre>
--	---

其中  $x$  属于前闭后开区间  $[0, N)$ ，起点到终点的间隔为  $N$ ，循环次数为  $N$ 。 $y$  属于闭区间  $[0, N-1]$ ，起点到终点的间隔为  $N-1$ ，循环次数为  $N$ 。相比之下，前一种写法更加直观，尽管两者的功能是相同的。

### 4.10.2 循环语句的效率

标准 C++/C 循环语句中，for 语句使用频率最高，while 语句其次，do/while 语句很少用。我们着重讨论一下使用 for 语句遍历多维数组的效率问题。

**【建议 4-4】：** 对于多维数组来说，正确的遍历方法要看语言以什么顺序来安排数组元素的存储空间。比如 FORTRAN 是以“先列后行”的顺序在内存中连续存放数组元素，而 C++/C 则是以“先行后列”的顺序来连续存储数组元素。因此，以“先列后行”存储的数组，就应该以“先列后行”的顺序遍历，以“先行后列”存储的数组就应该以“先行后列”的顺序遍历。

如示例 4-12 所示。

示例 4-12

<pre>double array[1024][512]; for (int r = 0; r &lt; 1024; ++r)     for (int c = 0; c &lt; 512; ++c) {         std::cout &lt;&lt; array[r][c] &lt;&lt; std::endl;     }</pre>
---

就 C++/C 多维数组来说，“先行后列”遍历效率肯定好于“先列后行”遍历，不论其行数远大于列数还是情况相反甚至接近，即使在最坏的情况下也至少与“先列后行”遍历的效率相当。影响效率的实际上主要是大型数组导致的内存页面交换次数以及 cache 命中率的高低，而不是循环次数本身。

我们以二维 int 类型数组为例来说明这种情况，其他类型的二维数组或二维以上的数组也是同样的道理。

在 C++ 中，二维数组以“先行后列”的顺序存储在连续的内存中。例如 `int a[10][3]` 在内存中的映像如图 4-6 所示（假设一个 int 大小为 4 字节，这是在 MSVC 下某次执行时打印出来的逻辑地址）。

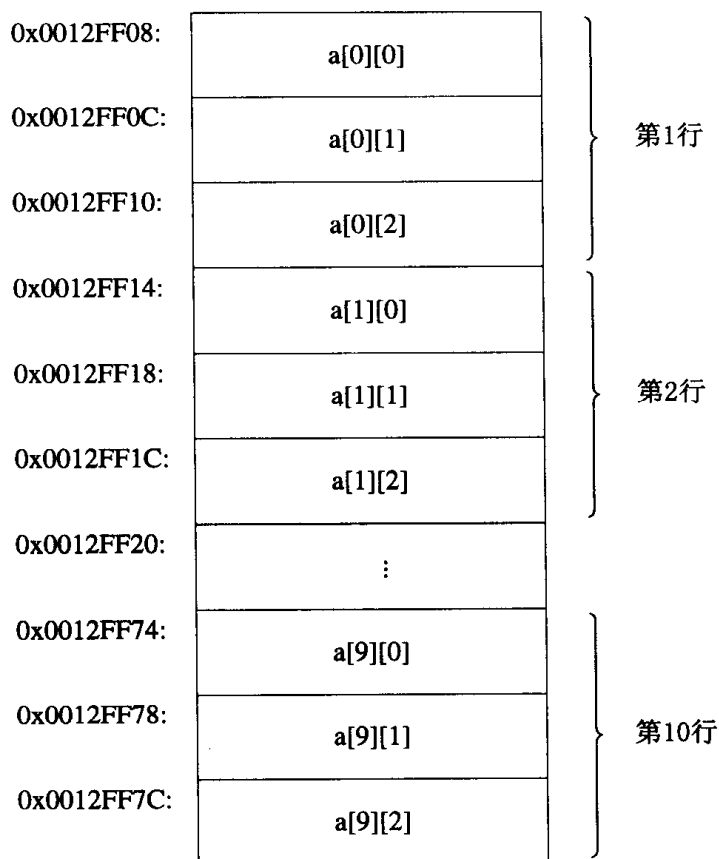


图 4-6 C++/C 数组的内存映像

我们来比较一下“先行后列”遍历和“先列后行”遍历的过程。

“先行后列”：

```
for (int i = 0; i < 10; ++i)
    for (int j = 0; j < 3; ++j)
    {
        a[i][j] = 0;
    }
```

即外层循环走过每一个行，在走到特定的某一行时，内层循环走过该行包含的每一列的所有元素。显然内层循环执行时地址跳跃的幅度很小，都是连续地依次访问每一个内存位置相邻的元素，没有跨行存取。

“先列后行”：

```
for (int j = 0; j < 3; ++j)
    for (int i = 0; i < 10; ++i)
    {
        a[i][j] = 0;
    }
```

外层循环依次走访每一列，对于其中的某一列，内层循环依次访问该行包含的每一个元素，但是这些元素的内存位置不再是相邻的，而是跨行的。很显然，地址

跳跃的幅度较大。

数组元素的访问是真正的随机访问（直接地址计算）。如果整个数组能够在一个内存页中容纳下，那么在对整个数组进行操作的过程中至少不会为了访问数组元素而出现缺页中断、页面调度和页面交换等情况，只需要一次外存读取操作就可以将数组所在的整个页面调入内存，然后直接访问内存就可以了。此时“先行后列”和“先列后行”的遍历效率差不多（仅有的差别是 cache 命中率不同，但是差别不大）。

但是如果整个数组内容需要多个内存页来容纳的话，情况就大不一样了，尤其是列数较大的情况下。我们就考虑一种特殊的情况：假设一个内存页的大小为 4096 字节，并且定义一个 int 数组，其列数为 1024（即每一行的元素恰好占用一页），行数远小于列数，假设为 128 行。虽然这个假设比较特殊，但是足以说明问题。我们来计算一下“先行后列”遍历和“先列后行”遍历分别需要进行多少次页面交换。

数组定义：

```
int b[128][1024];
```

这就是说，该数组每行包含 1024 个 int 类型数，正好占据一个内存页的空间，因此整个数组将占据 128 个内存页。

“先行后列”：

```
for (int i = 0; i < 128; ++i)
    for (int j = 0; j < 1024; ++j)
    {
        b[i][j] = 0;
    }
```

外层循环走过每一个行，在走到特定的某一行时，内层循环走过该行的 1024 个元素，正好走完一页，没有发生页面调度，且 cache 命中率高；当循环进入下一行时操作系统从外存调入下一页。因此可以得知，遍历完整个数组发生页面调度的次数最多为 128 次（假设一次调度一页）。

“先列后行”：

```
for (int j = 0; j < 1024; ++j)
    for (int i = 0; i < 128; ++i)
    {
        b[i][j] = 0;
    }
```

在遍历某一系列的时候，由于它包含的每一个元素都恰好分别位于每一个行内，因此内层循环每执行一次就会发生一次页面调度，遍历一列下来就会发生 128 次页面调度，显然总共发生的页面调度次数将可能达到  $1024 \times 128$  次（假设一次调度一页），也就是“先行后列”遍历的 1024 倍。不过实际情况并没有这么坏，因为如果物理内存足够，页面调度和页面交换的次数往往会减少很多。但是可以肯定的是：“先列后行”遍历发生的页面交换次数要比“先行后列”多，且 cache 命中率相对也低。这恰恰就是导致“先列后行”遍历效率降低的真正原因。



结论 (针对大型数组, 大规模矩阵):

(1) 当行数远大于列数的时候 ( $m \gg n$ ), “先行后列”遍历的效率高于“先列后行”遍历的效率, 这是显然的, 但可能不明显;

(2) 反之当列数远大于行数的时候 ( $n \gg m$ ), “先行后列”遍历的效率必然高于“先列后行”遍历的效率, 这是从上面分析可以得出的结论;

(3) 即使行数和列数接近的时候 ( $m \approx n$ ), “先行后列”的效率还是高于“先列后行”遍历的效率。

事实胜于雄辩! 下面我们就来做一个实际测试, 测试代码如示例 4-13 所示。

示例 4-13

---

```
const int MAX_ROW = 500;
const int MAX_COLUMN = 10000;
int (*a)[MAX_COLUMN] = new int[MAX_ROW][MAX_COLUMN];
std::cerr << "int array(MAX_ROW = " << MAX_ROW \
    << ", MAX_COLUMN = " << MAX_COLUMN << ")\n";

// 先行后列
DWORD start = GetTickCount();
for (int i = 0; i < MAX_ROW; ++i)
    for (int j = 0; j < MAX_COLUMN; ++j)
        a[i][j] = 1;
DWORD end = GetTickCount();
std::cerr << "Stepthrough by row-first-column-last takes " \
    << (end - start) << "毫秒\n";

delete []a;
```

---

```
const int MAX_ROW = 500;
const int MAX_COLUMN = 10000;
int (*a)[MAX_COLUMN] = new int[MAX_ROW][MAX_COLUMN];
std::cerr << "int array(MAX_ROW = " << MAX_ROW \
    << ", MAX_COLUMN = " << MAX_COLUMN << ")\n";

// 先列后行
DWORD start = GetTickCount();
for (int j = 0; j < MAX_COLUMN; ++j)
    for (int i = 0; i < MAX_ROW; ++i)
        a[i][j] = 1;
DWORD end = GetTickCount();
std::cerr << "Stepthrough by column-first-row-last takes " \
    << (end - start) << " 毫秒\n";

delete []a;
```

---

表 4-3 是某次不同行列配对下的一组测试结果 (单位: ms), 测试平台: Intel CPU 1.7GHz + Windows2000 + MSVC++6.0。

表 4-3 多维数组遍历测试结果

Row/Col 遍历顺序	Row = 500 Col = 10000	Row = 10000 Col = 500	Row = 2000 Col = 2500	Row = 2500 Col = 2000	Row = 2000 Col = 2000
先行后列	63 ms	78 ms	62 ms	78 ms	47 ms
先列后行	188 ms	735 ms	844 ms	828 ms	672 ms

比较一下测试结果，看看每种情况下所花费时间的相对大小，一切就都明白了。

**【建议 4-5】：** 如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。

在示例 4-14 中，左边的写法比右边的写法多执行了  $N-1$  次逻辑判断，并且前者的逻辑判断打断了循环的“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果  $N$  非常大，最好采用右边的写法，可以提高效率。如果  $N$  非常小，两者效率差别并不明显，采用左边的写法比较好，因为程序更加简洁。

示例 4-14

<pre> for(i = 0; i &lt; N; i++) {     if(condition)         DoSomething();     else         DoOtherthing(); } </pre>	<pre> if(condition) {     for(i = 0; i &lt; N; i++)         DoSomething(); } else {     for(i = 0; i &lt; N; i++)         DoOtherthing(); } </pre>
--	--

我们举一个循环结构的例子来结束本节：求两个整数的最大公约数和最小公倍数。

求最大公约数可采用辗转相除法，其流程如图 4-7 所示。最小公倍数就是两个整数的乘积除以其最大公约数，程序见示例 4-15。

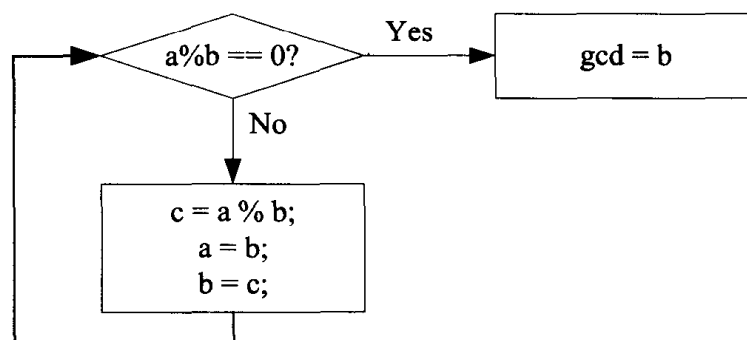


图 4-7 求最大公约数和最小公倍数的流程

示例 4-15

```
int main()
{
    unsigned long a, b, c = 0;      /* 两个整数和临时变量 */
    unsigned long lcm = 0, gcd = 0; /* 最小公倍数和最大公约数 */

    printf("Input two positive integers: ");
    scanf("%lu %lu", &a, &b);
    unsigned long ra = a, rb = b;   /* 保存原始数据 */

    while (a % b != 0){
        c = a % b;
        a = b;
        b = c;
    }

    gcd = b;
    lcm = (ra * rb) / gcd;          /* 使用原始数据计算 lcm */

    printf("Their Greatest Common Divisor is %lu.\n", gcd);
    printf("Their Least Common Multiple is %lu.\n", lcm);

    return 0;
}

// testing
Input two positive integers: 1240 85
Their Greatest Common Divisor is 5.
Their Least Common Multiple is 21080.
Input two positive integers: 85 1240
Their Greatest Common Divisor is 5.
Their Least Common Multiple is 21080.
```

## 4.11 结构化程序设计原理

我们已经讲完了基本的控制结构，但是光有基本结构还不行，还需要把它们组合起来形成更大的结构，解决更复杂的问题。

我们可以证明，任何复杂的选择结构都可以化为最简单的 if 结构，任何复杂的循环结构都可以化为最简单的 while 结构。三种基本控制结构只能有两种组合方式，那就是堆叠和嵌套。堆叠就像是搭积木一样，把需要的每一个控制结构按照先后顺序拼接起来，它们的顺序只与具体业务有关；嵌套的意思就是“我中有你或你中有我”，任何两种结构都可以嵌套。结构化编程就是利用这三种控制结构和两种组合方

式来编写程序的。在第 7 章我们将讲述以函数为基础的模块化程序设计，但是结构化编程是模块化编程的基础，任何函数都是通过结构化编程和函数调用来实现的。

## 4.12 goto/continue/break 语句

自从提倡结构化程序设计以来，goto 就成了有争议的语句。首先，由于 goto 语句可以实现无条件跳转，如果不加限制，它的确会破坏结构化程序设计风格。其次，goto 语句经常带来错误和隐患。它可能跳过某些对象的构造、变量的初始化、重要的计算等语句，如示例 4-16 所示。

示例 4-16

---

```
goto state;
String s1, s2;      // 被 goto 跳过
int sum = 0;        // 被 goto 跳过
...
state:
...
```

---

如果编译器不能发觉此类错误，每用一次 goto 语句都可能留下隐患。

很多人建议废除 C++/C 的 goto 语句，以绝后患。但老实说，错误是程序员自己造成的，不是 goto 的过错（你不用它，难道它还会跳出来骚扰你不成？）。goto 语句至少有一处可显神通，它能从多层嵌套的循环体中嗖地一下子跳到外面，用不着写很多次的 break 语句，从而提高了效率，见示例 4-17。

示例 4-17

---

```
for(...)
{ ...
    for(...)
    { ...
        for(...)
        { ...
            goto error;
        }
    }
}
error: // 错误处理代码
```

---

就像楼房失火了，来不及从楼梯一级一级往下走，可以从窗口跳出来。所以我们主张少用、慎用 goto 语句，而不是禁用。

## 4.13 示例

现在我们给出几个具体的示例程序,它们都使用了本章讲述的C++/C 基础知识,供C++/C 初级程序员学习用。

**例 1:** 从键盘输入一行字符,直到按下“回车”键,然后将所有字符转换成大写字符并反序输出。

从键盘读入字符使用 `getchar()` 库函数,读入字符串使用 `gets()` 库函数,因此需要预先申请在内存中保存字符串的缓冲区。但是我们无法预知用户到底会输入多少个字符才结束,因此要预留足够的空间。转换工作使用库函数 `toupper()` 完成,输出字符使用 `putchar()` 库函数。完整的程序见示例 4-18。

示例 4-18

---

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    char temp[255] = { '\0' };
    printf("Input a string, then press <CR> : \n");
    gets(temp);
    /*
     * 或者使用: int i = 0;
     *          while( ( temp[i] = getchar() ) != '\n' ) i++;
     *          temp[i] = '\0';
     */
    unsigned int strLen = strlen(temp);
    for(int j = strLen - 1; j >= 0; j--) {
        temp[j] = toupper(temp[j]);
        putchar(temp[j]);
    }
    putchar('\n');
    return 0;
}
```

---

**例 2:** 计算正整数  $n$  的阶乘。可以使用迭代法计算也可以使用递归法计算,我们将在第 7 章中给出递归法的程序。注意这种计算可能导致结果“上溢”(试着输入 20 并查看输出)。程序见示例 4-19。

示例 4-19

---

```
#include <stdio.h>
int main(void)
```

---

```

{
    unsigned long n = 0, fact = 1;
    printf("Input a positive number : ");
    scanf("%d", &n);
    for(unsigned int k = 1; k <= n; k++)
        fact = fact * k;
    printf("The factorial of %d is %d: \n", n, fact);
    return 0;
}

```

**例3：**输入一个正整数  $n$ ，计算斜边长在  $n$  以内的所有可能的边长为整数的直角三角形的边长组合。最笨的办法是使用三重循环，每一重循环都迭代  $n$  次。不过这样检测出来的三角形有重复的（例如  $\langle 5, 3, 4 \rangle$  和  $\langle 5, 4, 3 \rangle$  都是同一个三角形），而且性能太低，执行了许多无谓的循环检测。我们仅检测斜边和一条直角边固定的情况下，另一条直角边大于或等于第一条直角边并小于斜边的情况就可以了。不过这样还是执行了一些无用的检测。我们知道，如果斜边和第一条直角边固定，则第二条直角边必定小于等于  $\sqrt{(\text{斜边})^2 - (\text{第一条直角边})^2}$  的整数部分，因此我们按照这个算法来搜索。程序见示例 4-20，读者可能还有更高效率的解法，请编程试一试。

示例 4-20

```

#include <stdio.h>
#include <math.h>
int main(void)
{
    unsigned int n = 0, count = 0;
    printf("Input a positive number : ");
    scanf("%u", &n);
    printf("The list of right-angled triangles: \n");
    for( unsigned int r = 1; r <= n; r++) { /*斜边 r 从 1 到 n 迭代*/
        unsigned long rsquare = r * r;
        for( unsigned int p = 1; p < r; p++) { /*第一条直角边 p 从 1 到 r 迭代*/
            unsigned int psquare = p * p;
            unsigned int w = (unsigned int)sqrt(rsquare - psquare);
            for( unsigned int q = p; q <= w; q++) { /*第三条直角边 q 从 p 到 w 迭代*/
                if( rsquare == (psquare + (q * q)) ) {
                    printf("%u\t\t%u\t\t%u \n", r, p, q);
                    count++;
                }
            }
        }
    }

    if(count == 0) {

```



```
        printf("No such right-angled triangles!\n");
    }else {
        printf("There are %d right-angled triangles in total!\n", count);
    }
    return 0;
}
```

计算机不仅可以解决数学领域的复杂计算，还可以通过计算来解决逻辑推理难题。现在的“人工智能”就是利用计算机“不知疲倦”的计算能力来进行定理证明、逻辑推理和决策支持等工作的。

**例 4：**协助破案。假设已经查清，有 A、B、C、D、E 五个嫌疑人可能参与制造了一起抢劫银行案，但是不知道其中哪几个人是真正的案犯。不过，有确凿证据表明：

- ① 如果 A 参与了作案，则 B 一定也会参与。
- ② B 和 C 两人中只有一人参与了作案。
- ③ C 和 D 要么都参与了作案，要么都没有参与。
- ④ D 和 E 两个人中至少有一人参与作案。
- ⑤ 如果 E 作案，则 A 和 D 一定参与作案。

是不是有点绕口？我们可以用数理逻辑中的正规表达式来表示上述论断：

- ①  $A \rightarrow B$
- ②  $(B \wedge \neg C) \vee (\neg B \wedge C)$
- ③  $(C \wedge D) \vee (\neg C \wedge \neg D)$
- ④  $(D \wedge E) \vee (\neg D \wedge E) \vee (D \wedge \neg E)$
- ⑤  $E \rightarrow (A \wedge D)$

我们现在用 1 (TRUE) 表示作案，0 (FALSE) 表示未作案，则每个人的取值范围就是 {0, 1}。然后我们在 5 个人取值的所有可能的组合空间中进行搜索，同时满足这五条线索的一个组合就是本案的答案了。于是，上述正规表达式可进一步表示为下列 C++/C 逻辑表达式：

- ①  $A == 0 \parallel (A == 1 \ \&\& \ B == 1)$
- ②  $B + C == 1$
- ③  $C == D$
- ④  $D + E >= 1$
- ⑤  $E == 0 \parallel (E == 1 \ \&\& \ A == 1 \ \&\& \ D == 1)$

我们用另一个变量 count 来表示组合空间中某一个组合能够满足几条论断，如果出现了这样一个组合：它同时满足了这五条论断，那么它就是我们要找的组合。程序见示例 4-21。

示例 4-21

```
#include <stdio.h>
int main(void)
{
```

```

int A, B, C, D, E;
int count = 0;
for( A = 0; A < 2; A++) {
    for( B = 0; B < 2; B++) {
        for( C = 0; C < 2; C++) {
            for( D = 0; D < 2; D++) {
                for( E = 0; E < 2; E++) {
                    count = 0;           // 计数器清 0
                    count += ( A == 0 || ( A == 1 && B == 1 ) );
                    count += ( ( B + C ) == 1 );
                    count += ( C == D );
                    count += ( ( D + E ) >= 1 );
                    count += ( E == 0 || ( E == 1 && A == 1 && D == 1 ) );
                    if ( count == 5 ) // 找到一个满足所有条件的组合
                        goto finish;
                }
            }
        }
    }
}

```

finish:

```

printf ("Suspect A is %s.\n", ( A == 1 ) ? "a criminal" : "not a criminal");
printf ("Suspect B is %s.\n", ( B == 1 ) ? "a criminal" : "not a criminal");
printf ("Suspect C is %s.\n", ( C == 1 ) ? "a criminal" : "not a criminal");
printf ("Suspect D is %s.\n", ( D == 1 ) ? "a criminal" : "not a criminal");
printf ("Suspect E is %s.\n", ( E == 1 ) ? "a criminal" : "not a criminal");
return 0;
}

```

输出结果如下:

```

Suspect A is not a criminal.
Suspect B is not a criminal.
Suspect C is a criminal.
Suspect D is a criminal.
Suspect E is not a criminal.

```

在这个例子中，goto 语句还是很有用的！

最后我们举一个数值计算的例子：使用迭代法计算超越方程  $f(x) = 0$  的近似解。

理论证明：最高次幂不高于 4 的一元方程的根可用根式来表示，高于 4 次的则一般不存在根的解析表达式，对于超越方程更不太可能存在。此时要借助数值计算来寻找方程的近似解。

已知方程  $x^6 - 8x - 4 = 0$ ，求它的近似解，要求误差小于  $10^{-10}$ 。

迭代法是这样的：首先将方程  $f(x) = 0$  改写为如下的等价形式：

$$x = g(x)$$

然后在可能的根区间  $[a, b]$  上任取一点  $x_0$ ，代入  $g(x)$ ，得到  $x_1 = g(x_0)$ ，一般说来

$x_1 \neq x_0$ ; 再把  $x_1$  代入  $g(x)$ , 得到  $x_2 = g(x_1)$ ; .....这样就计算出了一系列值:

$x_0, x_1, x_2, \dots, x_{n-1}, x_n, \dots$

当  $(x_n - x_{n-1})$  的绝对值小于给定的误差时就可以认为  $x_n$  是所求的根了。

我们把方程改写为  $x = \frac{x^6}{8} - \frac{1}{2}$ ,  $g(x) = \frac{x^6}{8} - \frac{1}{2}$ 。取  $x_0 = 1$ , 程序见示例 4-22。

示例 4-22

```
#include <stdio.h>
#include <math.h>
double g(double x);
int main(void)
{
    double x0 = 1, x1 = 0;
    int count = 0;
    printf("x%d = %10fn", count, x0);
    while( fabs( x1 - x0 ) >= 1e-10 ){
        x1 = x0;
        x0 = g(x1);
        count++;
        printf("x%d = %10fn", count, x0);
    }
    printf("approximate x = %10fn", x0);
    return 0;
}
double g(double x)
{
    return pow(x, 6) / 8.0 - 0.5;
}
x0 = 1.000000
x1 = -0.375000
x2 = -0.499652
x3 = -0.498055
x4 = -0.498092
x5 = -0.498091
x6 = -0.498091
x7 = -0.498091
x8 = -0.498091
approximate x = -0.498091
```

我们看到, 在第 5 次迭代后就已经找到了满足误差条件的解, 所以后面的迭代就是不必要的了。

## 第5章 C++/C 常量

我们在程序中经常使用常量，例如 `for` 语句中循环控制变量的上下限值，用来初始化指针的 `NULL`，用来初始化字符数组的 `'\0'`。语言实现也在隐含地使用着一些常量，例如初始化全局变量和静态变量的 `0` 等。此外，还有一些我们平时没有“感觉”的常量，例如函数地址（即函数名）、静态数组的名字、字符串常量的地址，等等。

常用的常量可以分为：字面常量、符号常量、契约性常量、布尔常量和枚举常量等。本章将逐一揭示每一种常量的本质及其用法。由于布尔常量及其使用已经在第4章讨论过，本章不再赘述。

### 5.1 认识常量

#### 5.1.1 字面常量

字面常量也许是在程序中使用得最多的常量了，例如直接出现的各种进制的数字、字符（"括住的单个字符"）或字符串（"括住的一系列字符"）等。实际上，只存在基本数据类型的字面常量。示例 5-1 是一些字面常量的例子。

示例 5-1

```
x = -100.25f;
#define OPEN_SUCCESS 0x00000001
char c = 'a';
char *pChar = "abcdef"; //取字符串常量的地址
int *pInt = NULL;
```

由于字面常量只能引用，不能修改，所以语言实现一般把它保存在程序的符号表里而不是一般的数据区中。符号表是“只读”的，其实它是一种访问保护机制，千万不要理解为只读存储器（ROM）。除了字符串外，你无法取一个字面常量的地址，例如 `int *p = &5`；这类语句是错误的。当你试图通过常量字符串的地址修改其中的字符时就会报告“只读”错误。例如：

```
*(pChar + 2) = 'k'; // 错误，不能修改字面常量的内存单元
```

如果程序中到处都充斥着各种各样的字面常量，那么可能存在相同的常量，我们应该把相同的常量合并，以减少内存消耗。有些连接器自动执行常量合并，而有的连接器则提供了常量合并的开关，以优化程序的效率。如果你的编译链接环境支持常量合并，那么请将它打开。

## 5.1.2 符号常量

存在两种符号常量：用#define 定义的宏常量和用 const 定义的常量。由于#define 是预编译伪指令，它定义的宏常量在进入编译阶段前就已经被替换为所代表的字面常量了，因此宏常量在本质上是字面常量。在 C 语言中，用 const 定义的常量其实是值不能修改的变量，因此会给它分配存储空间（外连接的）；但是在 C++ 中，const 定义的常量要具体情况具体对待：对于基本数据类型的常量，编译器会把它放到符号表中而不分配存储空间，而 ADT/UDT 的 const 对象则需要分配存储空间（大对象）。还有一些情况下也需要分配存储空间，例如强制声明为 extern 的符号常量或取符号常量的地址等操作，都将强迫编译器为这些常量分配存储空间，以满足用户的要求。

你可以取一个 const 符号常量的地址：对于基本数据类型的 const 常量，编译器会重新在内存中创建它的一个拷贝，你通过其地址访问到的就是这个拷贝而非原始的符号常量；而对于构造类型的 const 常量，实际上它是编译时不允许修改的变量，因此如果你能绕过编译器的静态类型安全检查机制，就可以在运行时修改其内存单元，见示例 5-2。

示例 5-2

```
const long lng = 10;
long *pl = (long*)&lng;           // 取常量的地址
*pl = 1000;                       // “迂回修改”
cout << *pl << endl;              // 1000, 修改的是拷贝内容!
cout << lng << endl;              // 10, 原始常量并没有变!

class Integer {
public:
    Integer(): m_lng(100) {}
    long m_lng;
}

const Integer int_1;
Integer *pInt = (Integer*)&int_1;  // 去除常数属性
pInt->m_lng = 1000;
cout << pInt->m_lng << endl;       // 1000, 修改 const 对象
cout << int_1.m_lng << endl;      // 1000, “迂回修改”成功
```

这个例子说明：const 符号只是编译时（源代码级或语言层面）强类型安全检查机制的一种手段，以帮助程序员发现无意中要修改它们的代码并进行纠正，而在运行时（二进制层面）无法阻止恶意的修改。也可以说就是“防君子不防小人”。

### 【提示 5-1】:

从理论上讲，只要你手中握有一个对象的指针（内存地址），你就可以设法绕过编译器随意修改它的内容，除非该内存受到操作系统的保护。也就是说，C++ 并没有提供对指针有效性的静态检查，而是把它丢给了操作系统，这正是使用指针的危险所在。

### 【提示 5-2】:

在标准 C 语言中，const 符号常量默认是外连接的（分配存储），也就是说你不能在两个（或两个以上）编译单元中同时定义一个同名的 const



符号常量（重复定义错误），或者把一个 `const` 符号常量定义放在一个头文件中而在多个编译单元中同时包含该头文件。但是在标准 C++ 中，`const` 符号常量默认是内连接的，因此可以定义在头文件中。当在不同的编译单元中同时包含该头文件时，编译器认为它们是不同的符号常量，因此每个编译单元独立编译时会分别为它们分配存储空间，而在连接时进行常量合并。

### 5.1.3 契约性常量

契约性 `const` 对象的定义并未使用 `const` 关键字，但被看做是一个 `const` 对象，见示例 5-3。

示例 5-3

```
void ReadValue(const int& num)
{
    cout << num;
}
int main(void)
{
    int n = 0;
    ReadValue(n); // 契约性 const, n 被看做是 const
}
```

### 5.1.4 枚举常量

C++/C 的构造类型 `enum` 实际上常用来定义一些相关常量的集合。标准 C++/C 规定枚举常量的值是可以扩展的，并非受限于一般的整型数的范围（见示例 5-4）。

示例 5-4

```
enum Gigantic
{
    SMALL = 10,
    GIGANTIC = 3000000000000
};
```

至于底层如何实现，则依赖于具体的环境和编译器厂商，可能会有不同的语义，请查看编译器文档。

## 5.2 正确定义符号常量

如果不使用符号常量，而是直接在程序中填写数字或字符串，将会有哪些麻烦呢？

（1）程序的可读性（可理解性）变差。程序员自己会忘记哪些数字或字符串代表什么意思，用户则更不知它们从何而来、表示什么；

- (2) 在程序的很多地方输入同样的数字或字符串, 难保不发生书写错误;
- (3) 如果要修改数字或字符串, 则需要同时在很多地方改动, 既麻烦又容易出错。

**【建议 5-1】:** 所以尽量使用含义直观的符号常量来表示那些将在程序中多次出现的数字或字符串。

例如:

```
#define      MAX    100           // 宏常量
const int    MAX = 100;          // const 常量
const float  PI = 3.14159;       // const 常量
```

**【规则 5-1】:** (1) 所以在 C++ 需要对外公开的常量放在头文件中, 不需要对外公开的常量放在定义文件的头部。为便于管理, 可以把不同模块的常量集中存放在一个公用的头文件中;

(2) 如果某一常量与其他常量密切相关, 应在定义中包含这种关系, 而不应给出一些孤立的值。例如:

```
const float  RADIUS = 100;
const float  DIAMETER = RADIUS * 2;
```

## 5.3 const 与#define 的比较

C++ 语言可以用 `const` 来定义常量, 也可以用 `#define` 来定义常量。但是前者比后者具有更多的优点:

(1) `const` 常量有数据类型, 而宏常量没有数据类型。编译器可以对前者进行静态类型安全检查; 而对后者只进行字符替换, 没有类型安全检查, 并且在字符替换时可能会产生意料不到的错误 (边际效应);

(2) 有些集成化的调试工具可以对 `const` 常量进行调试, 但是不能对宏常量进行调试。

**【提示 5-3】:** 所以在 C++ 程序中应尽量使用 `const` 来定义符号常量, 包括字符串常量。

要注意的是, `const` 不仅仅用于定义符号常量, 凡是需要编译器帮助我们预防无意中修改数据的地方, 都可以使用 `const`, 比如 `const` 数据成员、`const` 成员函数、`const` 返回类型、`const` 参数等。

**【提示 5-4】:** `const` 是 `constant` 的缩写, 是“恒定不变”的意思。被 `const` 修饰的东西都受到 C++/C 语言实现的静态类型安全检查机制的强制保护, 可以预防意外修改, 能提高程序的健壮性。请参考 6.13 节的论述。



## 5.4 类中的常量

有时我们希望某些常量只在类中有效。由于 `#define` 定义的宏常量是全局的，不能达到目的，于是想当然地觉得应该用 `const` 修饰数据成员来实现。`const` 数据成员的确是存在的，但其含义却不是我们所期望的。非静态 `const` 数据成员是属于每一个对象的成员，只在某个对象的生存期限内是常量，而对于整个类来说它是可变的，除非是 `static const`。因为类可以创建多个对象，不同的对象其 `const` 数据成员的值可以不同。

不能在类声明中初始化非静态 `const` 数据成员。示例 5-5 的用法是错误的，因为在类的对象被创建之前，编译器无法知道 `SIZE` 的值是什么。

示例 5-5

---

```
class A
{
    ...
    const int SIZE = 100;           // 错误，企图在类声明中初始化 const 数据成员
    int array[SIZE];               // 错误，未知的 SIZE
};
```

---

非静态 `const` 数据成员的初始化只能在类的构造函数的初始化列表中进行，见示例 5-6。

示例 5-6

---

```
class A
{
    ...
    A(int size);                   // 构造函数
    const int SIZE ;
};
A::A(int size) : SIZE(size)       // 构造函数的初始化列表
{
    ...
}
A a(100);    // 对象 a 的 SIZE 值为 100
A b(200);    // 对象 b 的 SIZE 值为 200
```

---

那么，怎样才能建立在整个类中都恒定的常量呢？别指望 `const` 数据成员了，应该用类中的枚举常量来实现，见示例 5-7。

示例 5-7

---

```
class A
{
```

---

```
...
enum
{
    SIZE1 = 100,           // 枚举常量
    SIZE2 = 200
};

int array1[SIZE1];
int array2[SIZE2];
};
```

枚举常量不会占用对象的存储空间，它们在编译时被全部求值，更何况它定义的是一个匿名枚举类型。枚举常量的缺点是不能表示浮点数（如  $\text{PI}=3.14159$ ）和字符串。

还可以使用另一种方法来定义类的所有对象都共享的常量，即 `static const`，见示例 5-8。

示例 5-8

```
class A
{
public: // 有些语言实现可能不支持这样的初始化，如 Visual C++
    static const int SIZE1 = 100;    // 静态常量成员
    static const int SIZE2 = 200;    // 静态常量成员

private:
    int array1[SIZE1];              // 普通成员
    int array2[SIZE2];              // 普通成员
};
```

5.5 实际应用中如何定义常量

如果要为程序中多个编译单元或者多个模块定义统一的符号常量，在 C 程序中你常常会把它定义在哪里呢？在 C++ 程序中你又常常会把它定义在哪里呢？如果是只为某一个编译单元定义符号常量，又如何？

表 5-1 是我们常用的做法。

表 5-1 常用做法

	多个编译单元或模块公用	只为一个编译单元使用
C 程 序	[方法一：] 在某个公用头文件中将符号常量定义为 <code>static</code> 并初始化，例如：  // CommonDef.h	[方法一：] 同左边方法一。

(续表)

	多个编译单元或模块公用	只为一个编译单元使用
	<p><code>static const int MAX_LENGTH = 1024;</code></p> <p>然后每一个使用它的编译单元<code>#include</code> 该头文件即可; 或者在头文件中使用宏定义。</p> <p>[方法二: ]</p> <p>在某个公用的头文件中将符号常量声明为 <code>extern</code> 的, 例如:</p> <pre>// CommonDef.h extern const int MAX_LENGTH; 并且 在某个源文件中定义一次: const int MAX_LENGTH = 1024; 然后 每一个使用它的编译单元#include 上述头文件即可。</pre> <p>[方法三: ]</p> <p>如果是整型常量, 在某个公用头文件中定义 <code>enum</code> 类型, 然后每一个使用它的编译单元<code>#include</code> 该头文件即可。</p>	<p>[方法二: ]</p> <p>直接于该编译单元 (源文件) 开头位置将符号常量定义为 <code>static</code> 并初始化, 例如:</p> <pre>// foo.C static const int MAX_LENGTH = 1024;</pre> <p>[方法三: ]</p> <p>同左边方法三, 或者直接于该编译单元 (源文件) 开头位置定义 <code>enum</code> 类型</p>
C++ 程 序	<p>[方法一: ]</p> <p>在某个公用的头文件中直接在某个名字空间中或者全局名字空间中定义符号常量并初始化 (有无 <code>static</code> 无所谓), 例如:</p> <pre>// CommonDef.h const int MAX_LENGTH = 1024; 然后 每一个使用它的编译单元#include 该头文件即可。</pre> <p>[方法二: ]</p> <p>在某个公用头文件中并且在某个名字空间中或者全局名字空间中将符号常量声明为 <code>extern</code> 的, 例如:</p> <pre>// CommonDef.h extern const int MAX_LENGTH; 并且 在某个源文件中定义一次并初始化: const int MAX_LENGTH = 1024; 然后 每一个使用它的编译单元#include 上述头文件即可。</pre> <p>[方法三: ]</p> <p>如果是整型常量, 在某个公用头文件中定义 <code>enum</code> 类型, 然后每一个使用它的编译单元<code>#include</code> 该头文件即可。</p>	<p>[方法一: ]</p> <p>同左边方法一。</p> <p>[方法二: ]</p> <p>直接于该编译单元 (源文件) 开头定义符号常量并初始化 (有无 <code>static</code> 无所谓), 例如:</p> <pre>// foo.C const int MAX_LENGTH = 1024;</pre> <p>[方法三: ]</p> <p>同左边方法三, 或者直接于该编译单元 (源文件) 开头位置定义 <code>enum</code> 类型。</p>

(续表)

	多个编译单元或模块公用	只为一个编译单元使用
	<div>[方法四：] 定义为某一个公用类的 static const 数据成员并初始化，或者定义为类内的枚举类型，例如：  // Utility.h  class Utility{ public:      static const int MAX_LENGTH;      enum {          TIME_OUT = 10      };  };  // Utility.cpp  const int Utility::MAX_LENGTH = 1024;  每一个使用它的编译单元#include 该类的定义即可。</div>	<div>[方法四：] 同左边方法四</div>

显然，在 C 程序和 C++程序中定义符号常量是有区别的。我们仅讨论“多个编译单元公用符号常量”的声明和定义。

在 C 程序中，const 符号常量定义的默认连接类型（Linkage）是 extern 的，即外连接（external linkage），就像全局变量一样。因此，如果要在头文件中定义，必须使用 static 关键字，这样每一个包含该头文件的编译单元就会分别拥有该常量的一份独立定义实体（如同直接在每一个源文件中分别定义一次），否则会导致“redefinition”的编译器诊断信息；如果在源文件中定义，除非明确改变它的连接类型为 static（实际上是存储类型为 static，连接类型为内连接）的，否则其他编译单元就可以通过 extern 声明来访问它。

但是在 C++程序中，const 符号常量定义的默认连接类型却是 static 的，即内连接（internal linkage），就像 class 的定义一样，这就是在头文件中定义而不需要 static 关键字的原因。

搞清了 C 和 C++符号常量的区别，就不难理解上表所列的几种方法了。

这些做法各有优缺点，如果是整型常量或者浮点常量，那么优缺点都不是那么明显。对于 C++程序，我们比较一下方法二、四和方法一、三。

表 5-2 方法比较

	方法二、四	方法一、三
优点	<div>（1）节约存储，每一个编译单元访问的都是这个唯一的定义；  （2）修改初值后只需重新编译定义所在编译单元即可，影响面很小</div>	<div>维护方便</div>

(续表)

	方法二、四	方法一、三
缺点	如果要改变初值, 需修改源文件	(1) 如果修改常量初值, 则将影响多个编译单元, 所有受影响的编译单元必须重新编译; (2) 每一个符号常量在每一个包含了它们的编译单元内都存在一份独立的拷贝内容, 每个编译单元访问的就是各自的拷贝内容, 因此浪费存储空间

实际上, 在大型应用开发过程中, 修改常量初值的事情并不是经常发生, 甚至极少发生, 即使是在维护阶段, 也不太可能变来变去。因此方法二、四较方法一、三的优点就主要体现在存储空间上了。如果是整型和浮点型常量, 它们浪费的那点空间对一般应用来说还不至于达到无法容忍的地步, 不过对嵌入式应用的开发来说也许是一件大事儿。

那什么常量最浪费空间呢? 是字符串常量, 尤其是较长的字符串常量。

字符串常量的定义和整型常量的定义差不多, 但是其类型为 `const char *`, 因此我们常常这样定义它们:

```
const char* const ERR_DESP_NO_MEMORY = "There is no enough memory!";
```

可以在头文件中定义并初始化, 也可以在源文件中定义并初始化, 但是二者差别较大:

- ◇ 如果在头文件中定义并初始化, 那么包含了该头文件的每一个编译单元不仅会为每一个常量指针常量 (`const char * const`) 创建一个独立的拷贝项, 而且也会为那个长长的字符串字面常量创建一个独立的拷贝项, 就相当于在每一个编译单元内分别定义和初始化每一个常量时, 一次一个样。这是与整型或浮点型常量的定义不同的 (它们在初始化完后不再需要那个字面常量)。因此, 每一个编译单元内访问的字符串常量都是它自己单独创建的拷贝。空间的开销就体现在每一个字符串字面常量的独立拷贝项上;
- ◇ 如果采用方法二, 在头文件中声明所有常量指针常量, 而在源文件中定义并初始化它们, 则每一个包含该头文件的编译单元访问的不仅是常量指针常量的唯一实体, 而且字符串字面常量也是唯一的实体。这就大大节约了内存, 而且不失效率。

当然, 我们完全可以把常量合并的优化交给编译器和连接器来完成, 但是我们还是提倡由自己来优化常量的定义。





## 第6章 C++/C 函数设计基础

函数是 C++/C 程序的基本功能单元，是模块化程序设计的基础，其重要性不言而喻。仅使函数的功能正确是不够的，因为函数设计的细微缺点很容易导致该函数被错用。

本章首先介绍函数的基本知识，比较堆栈与堆的相似点和不同点；然后重点论述函数的接口设计和内部实现的一些规则，它们不仅用于一般的 C 函数设计，也用于 C++ 的类成员函数设计；接着介绍一些比较高级的内容，如存储类型、递归函数等；最后介绍断言、const 的使用，用于提高程序的健壮性。

### 6.1 认识函数

一个函数实际上就是“输入—处理—输出”模型的一种具体表现。编写 C 程序实际上主要就是编写各种各样的函数。在结构化程序设计中，一个大程序可划分为若干个具有不同功能的模块，每个模块都由一个或多个函数组成，模块之间的通信和模块内部功能就靠函数调用来实现。函数对于构建程序并不是必需的，但却是非常重要的，它极大地增强了代码的模块性，使程序更易于开发和维护。

在语言实现中，函数分为库函数和用户自定义函数，二者在本质上是一样的，只是前者是预先编译好的。在使用一种语言实现时，应该了解它提供哪些库，例如函数库、类库及系统调用等，并学习它们的用法。在程序中积极地使用现有的库不仅可以节省时间，还可以提高程序的质量。

**【提示 6-1】:** 在你需要某种功能的函数时，首先查看现有的库中是否提供了类似的函数。不要编写函数库中已有的函数，不仅因为这是重复劳动，而且自己编写的函数在各个质量属性方面一般都不如对应的库函数。库函数是经过严格测试和实践检验的。

对于静态链接库的函数库或类库，如果你调用了其中的函数（无论是直接调用还是间接调用），那么连接器会从相应的库中提取这些函数的实现代码并把它们连接到你的应用程序中；如果你没有调用库中的某些函数，则连接器是不会把它们的实现代码连接进来的，即使该库包含了成千上万个函数。如果你使用的是动态链接库（DLL），则运行时必须将所有 DLL 都拷贝到运行环境的相应目录下。

连接的本质就是把一个名字的实现（代码）绑定到对它的每一个引用语句上（在编译时或在运行时），如果你根本就没有引用这个名字，那么它的实现将和谁去绑定呢？

同样，如果程序中任何地方都没有调用你自己编写的某个函数的话，编译器也不会为该函数生成可执行代码。明白了这个道理，你就可以消除一些不必要的担心。

例如我们在开发一些通用类的时候，应该设计并实现其完整的功能，而不必担心它的使用者仅使用其中一小部分功能，却要包含整个类定义而会导致代码体积增大。C++模板实例化及类库的设计和实现都是这个道理。

## 6.2 函数原型和定义

早先的 C 语言存在**函数前置声明**的概念。因为在 C 语言环境中，同一个作用域中不能出现同名的全局函数，所以函数前置声明足以表明一个函数定义的存在性和惟一性。函数前置声明的格式是：

---

函数返回类型 函数名 ( ) ;

---

有了函数前置声明，即使把函数的定义体放在函数调用后面的任何地方也无妨，连接器在连接时能够找到它。但是函数前置声明并没有给出函数可接受的参数类型和个数，于是编译器无法对函数调用语句执行静态类型安全检查（即检查实参与形参的个数、类型及顺序等是否匹配）。这可能导致不正确的参数传递，从而出现运行时错误甚至破坏堆栈的情况。

解决这一问题的方法就是使用**函数原型**（ANSI/ISO C 从 C++借鉴了函数原型）。函数原型能够告诉编译器一个函数的作用域、返回值的数据类型、调用规范、连接规范、函数名称、可以接受的参数个数、类型及其顺序，以便编译器执行静态类型安全检查。

函数原型的格式如下：

---

[作用域] [函数的连接规范] 返回值类型 [函数的调用规范] 函数名(类型 1 [形参名 1], 类型 2 [形参名 2],...);

---

**形参**指的是：在函数原型或定义及 `catch` 语句的参数列表中被声明的对象或指针、宏定义中的参数、模板定义中的类型参数等。

**实参**指的是：函数调用语句中以逗号分隔的参数列表中的表达式、宏调用语句中以逗号分隔的列表中一个或多个预处理标识符的序列、`throw` 语句的操作数、表达式的操作数、模板实例化时的实际类型参数等。

函数调用中参数传递的本质就是用**实参来初始化形参而不是替换形参**，记住这一点很重要。尽管 C++/C 标准对形参和实参做了清晰的区分，但还是经常被人混用。当讨论函数和模板时，这两者之间的区别尤其重要（见示例 6-1）。

示例 6-1

---

```
void f(int n, char *p);           // n 和 p 是形参
template<class _T> class C{ ... }; // T 是形参
int main(void)
{
    char *q = "abcd";
```

---

```
f(5, q);           // 5 和 q 是实参
C<int> a;          // int 是实参
return 0;
}
```

**【提示 6-2】:** 应当在函数原型中写出形参名称，虽然编译器会忽略它们。这样做的目的是使函数具有“自说明”和“自编档”能力。不要在函数体内定义与形参同名的局部变量，因为形参也被看做本地变量。

函数原型的另一个作用就是指导编译器把实参隐式地转换为形参的类型，如果实参与形参类型不匹配，但是符合类型转换规则或者存在从实参类型到形参类型的转换函数的话；否则将拒绝编译。也就是说，函数原型可以强迫你使用正确的参数类型来调用函数，以避免运行时可能出现的错误或堆栈破坏。

函数定义的语法如下：

```
[函数的连接规范] 返回值类型 [函数的调用规范] 函数名(形参列表)
{
    函数体语句序列
}
```

如果该函数至少被调用一次的话，编译器将为函数定义体生成对应的可执行代码。

## 6.3 函数调用方式

一般函数都支持三种调用方式：像过程一样调用（或在表达式中调用），嵌套调用及递归调用。如果函数没有返回值（即返回 `void`）则不能在表达式中调用。递归函数在稍后讨论。除了递归外，函数调用必须是线性的，不能出现调用环路。

现在先看一个嵌套调用的例子。

求 3 个整数  $a$ 、 $b$ 、 $c$  的最小公倍数。显然，3 个整数的最小公倍数等于其中两个数的最小公倍数和第三个数的最小公倍数。即：

$$\text{lcm3}(a, b, c) = \text{lcm}(\text{lcm}(a, b), c)$$

程序见示例 6-2。

示例 6-2

```
#include <stdio.h>
unsigned long lcm (unsigned long a, unsigned long b); //求两个整数的最小公倍数
// 求 3 个整数的最小公倍数
unsigned long lcm3 (unsigned long a, unsigned long b, unsigned long c);
int main (int argc, char* argv[])
{
```

```
    unsigned long x, y, z;
    printf("Input 3 positive numbers: ");
    scanf("%lu %lu %lu", &x, &y, &z);
    printf("The LCM of this three numbers is : %lu\n", lcm3(x, y, z));
    return 0;
}

unsigned long lcm3(unsigned long a, unsigned long b, unsigned long c)
{
    return lcm(lcm(a, b), c);
}

unsigned long lcm(unsigned long a, unsigned long b)
{
    unsigned long multiple, c = 0;
    multiple = a * b;
    while(a % b != 0){
        c = a % b;
        a = b;
        b = c;
    }
    return (multiple / b);
}
```

实际上, 还有一种不常用的函数调用方式: 回调函数。这在事件驱动程序和多线程应用程序中很常见。狭义的“回调函数”就是由用户来实现, 而由系统调用的函数, 这是与 API 函数或库函数不同的地方。也就是说, 回调函数往往是具体应用领域相关的实现, 而 API 函数则是应用无关的通用功能实现。

回调函数最典型的例子就是系统定时器回调函数和线程函数。首先为一个定时器注册一个回调函数, 当定时器每次超时的时候系统就会自动触发对这个回调函数的调用, 在函数体内可以做你想做的事情。回调函数和普通的函数没有什么本质区别, 也可以由用户自己显式地调用, 只是当它作为回调函数的时候其运行方式和普通函数有所不同。

例如同一个函数可以同时作为几个线程的函数。此时由于线程上下文切换, 这个函数的多个执行流就可能出现重叠, 而且每个执行流退出的时机不可预料, 一个执行流的退出可能会破坏另一个执行流的堆栈。因此, 线程函数并不使用普通的函数堆栈, 而是使用线程自己的堆栈。线程堆栈是线程在每次启动时动态分配的, 这样就可以在线程函数每次执行时使用不同的堆栈, 从而避免线程函数执行流重叠时可能出现的堆栈破坏。此外, 如果一个函数可能会被多个线程调用的话, 对于它们共享的数据要进行同步访问保护, 并且避免使用 static 局部变量。

常使用的几个用来注册回调函数的系统函数有: set\_new\_handler()、atexit()、set\_terminate()、beginthread()、set\_unexpected()、signal()等。它们的具体用法可参考具体语言实现的帮助文档。

## 6.4 认识函数堆栈

我们知道函数调用必须通过堆栈来完成，但是许多人对堆栈的具体工作原理不清楚，因此产生了许多担心。

函数堆栈实际上使用的是程序的堆栈段内存空间，虽然程序的堆栈段是系统为程序分配的一种静态数据区，但是函数堆栈却是在调用到它的时候才动态分配的。也就是说，你不能在编译时就为函数分配好一块静态空间作为堆栈。一是因为无法在编译时确定一个函数运行时所需的堆栈大小；二是因为函数在调用完后如果还为其保留堆栈空间就会浪费内存，一个软件系统中成千上万的函数就会耗尽内存了！

当你对某些现象“百思不得其解”的时候，不妨设计一些小程序并打印出变量的地址，从中也许可以窥探到程序内部的秘密。例如想知道一个复合对象的各个成员在内存中是如何排列的，先声明的成员在高地址区还是在低地址区，或者 class 的各个访问区段的成员之间是如何排列的。那么我们可以设计如示例 6-3 所示的测试程序。但是要注意：这种测试的结果可能是平台相关的。

示例 6-3

```
#include <stdio.h>
#include <iostream>
using namespace std;
class TestArrange {
public:
    long m_lng;
    char m_ch1;
    TestArrange() { // constructor
        m_lng = 0;
        m_ch1 = 'a';
        m_int = 0;
        m_ch2 = 'a';
    }
    const int* GetIntAddr(){ return &m_int; }
    const char* GetChar2Addr(){ return &m_ch2; }
private:
    int m_int;
    char m_ch2;
};
int main(void)
{
    TestArrange test;
    cout << "Address of test object : " << &test << endl ;
    cout << endl ;
    cout << "Address of m_lng : " << &(test.m_lng) << endl ;
```

```
printf("Address of m_ch1 : %p\n", &(test.m_ch1));  
cout << "Address of m_int : " << test.GetIntAddr() << endl ;  
cout << "Address of m_ch2 : " << (void *)test.GetChar2Addr() << endl ;  
return 0;  
}  
Address of test object : 0012FF70  
Address of m_lng : 0012FF70  
Address of m_ch1 : 0012FF74  
Address of m_int : 0012FF78  
Address of m_ch2 : 0012FF7C
```

在 C 语言的格式化 I/O 中，常使用 %p 来输出内存地址值；而在 C++ 中，除了字符串的地址无法直接输出外，其他类型对象的地址都可以使用 “&” 输出。此时，我们可以采用一个通用的方法：把任何类型的地址或指针强制转换成 void\* 就可以输出了，包括字符串的指针。此外，还可以使用联合（union）来测试多字节数据类型对象（例如 int、double）的各个字节（高字节和低字节）在内存中的地址排列顺序。

堆栈是自动管理的，也就是说局部变量的创建和销毁、堆栈的释放都是函数自动完成的，不需要程序员的干涉。局部变量在程序执行流到达它的定义的时候创建，在退出其所在程序块的地方销毁，堆栈在函数退出的时候清退（还给程序堆栈段）。显然，由于函数堆栈在预先分配好的内存空间上创建，不需要运行时的搜索，因此比动态内存分配速度快而且安全。这是堆栈与堆和自由存储空间的区别所在（即是否显式分配和释放）。

函数堆栈主要有三个用途：在进入函数前保存环境变量和返回地址，在进入函数时保存实参的拷贝，在函数体内保存局部变量。

## 6.5 函数调用规范

与函数堆栈使用密切相关的就是函数调用规范，即调用约定（Calling Convention）。函数调用规范决定了函数调用的实参压栈、退栈及堆栈释放的方式，以及函数名改编（Name-Mangling）的方案，也即命名规范（Naming Convention）。Windows 环境下常用的调用规范有：

（1）\_\_cdecl：这是 C++/C 函数的默认调用规范，参数从右向左依次传递并压入堆栈，由调用函数负责堆栈的清退，因此这种方式利于传递个数可变的参数给被调用函数（因为只有调用函数才知道它给被调用函数传递多少个参数及它们的类型）。如 printf() 就是这样的函数（函数的实现不需要知道实参的具体类型和个数）。

（2）\_\_stdcall：这是 Win API 函数使用的调用规范。参数从右向左依次传递并压入堆栈，由被调用函数负责堆栈的清退。该规范生成的函数代码比 \_\_cdecl 更小，但当函数有可变个数的参数时会转为 \_\_cdecl 规范。在 Windows 中，宏 WINAPI、CALLBACK 都定义为 \_\_stdcall。

（3）\_\_thiscall：是 C++ 非静态成员函数的默认调用规范，不能使用个数可变的



参数。当调用非静态成员函数的时候，`this` 指针直接保存在 `ECX` 寄存器中而非压入函数堆栈。其他方面与 `__stdcall` 相同。

(4) `__fastcall`: 该规范所修饰的函数的实参将被直接传递到 CPU 寄存器中而不是内存堆栈中（这就是“快速调用”的含义）。堆栈清退由被调用函数负责。该规范不能用于成员函数。

函数必须指定一个调用规范。特别是在模块之间的逻辑接口中，每一个函数原型的调用规范必须与其实现的调用规范保持一致，否则会出现编译连接错误。特别地，如果你调用了在某个 DLL 中实现的 COM 对象的方法，而这些方法在创建时却没有显式地指定调用规范，那么它们会使用环境默认的调用规范，而此时如果你的程序使用的默认调用规范与它们的默认调用规范不一致的话，虽然你的程序可以通过编译和连接（因为不需要 COM 对象的导出库），但是在运行时就可能导致程序崩溃。

所以，凡是接口函数都必须显式地指定其调用规范，除非接口函数是类的非静态成员函数（non-static member function）。如果不显式指定调用规范，类的静态成员函数和全局函数都将采用 C/C++ 默认的函数调用规范或者由工程设置指定的调用规范，因此最好也为静态成员函数显式地指定调用规范。注意：类的静态成员函数的默认调用规范不是 `thiscall`，类的友元函数的调用规范也不是 `thiscall`，它们都是由函数本身指定或者由工程设定的。特别地，COM 接口的方法都指定 `__stdcall` 调用规范，而我们自己开发 COM 对象及其接口时也可以指定其他的调用规范。

## 6.6 函数连接规范

在使用不同编程语言进行软件联合开发的时候，需要统一函数、变量、数据类型、常量等的连接规范（Linkage Specification），特别是在不同模块之间共享的接口部分。当开发程序库的时候，明确连接规范也是必须遵循的一条规则。

对 COM 接口（Interface）及其使用的数据类型来说，是否采用统一的连接规范，对其二进制兼容性和可移植性都没有影响。因为连接规范主要影响到名字改编方案的不同，这样即使接口两端（COM 接口实现端和客户端）对接口的解释不同，只要它们使用一致的成员对齐方式和布局方案、一致的函数调用规范、一致的 virtual function 实现方式，总之就是一致的 C++ 对象模型，并且保证 COM 组件升级（DLL 透明替换）时不改变原来的接口和数据类型定义，则所有方法（methods，这里指 Interface 中的 pure virtual functions）的运行时绑定都不会存在问题（所有方法的调用都被转换为通过对象指针对 `vpitr` 和 `vtable` 及函数指针的操作，这种间接性不再需要任何方法名即函数名的参与，而接口名和方法名只是为了让客户端的代码能够顺利通过编译，但是连接时就全部不再需要了）。

但是对于定义于普通静态连接库（.Lib）和动态连接库（.DLL）中的全局数据类型、全局函数、全局变量甚至全局常量，它们的连接规范必须在两端（库和调用端）保持一致，否则客户程序会出现连接问题（linking error: unresolved external symbol

等),这是因为普通的封装为 DLL 的函数库或者类库,客户程序在创建时一般都需要与它们的导出库(.Lib)进行连接,除非使用 LoadLibrary()和 GetProcAddress()函数来获得 DLL 中函数的地址。通用的连接规范要属 C 连接规范:extern “C”。

具体使用方法如下。

- (1) 如果是仅对一个类型、函数、变量或常量指定连接规范:

```
extern "C" void WinMainCRTStartup();
extern "C" const CLSID CLSID_DataConvert;
extern "C" struct Student{.....};
extern "C" Student g_Student;
```

- (2) 如果是对一段代码指定连接规范:

```
#ifdef __cplusplus
extern "C" {
#endif
const int MAX_AGE = 200;
#pragma pack(push, 4)
typedef struct _Person
{
    char *m_Name;
    int m_Age;
} Person, *PersonPtr;
#pragma pack(pop)
Person g_Me;

int __cdecl memcmp (const void *, const void *, size_t);
void * __cdecl memcpy (void *, const void *, size_t);
void * __cdecl memset (void *, int, size_t);

#ifdef __cplusplus
}
#endif
```

## 6.7 参数传递规则

函数接口的两个要素是参数和返回值。C 语言中,函数的参数和返回值的传递方式有两种:值传递(pass by value)和地址传递(即指针传递,pass by pointer)。C++ 语言中增加了引用传递(pass by reference)。由于引用传递的效果像指针传递,而使用方式却像值传递,初学者常常迷惑不解,容易引起混乱,请阅读第 7 章的“引用与指针的比较”。

### **【规则 6-1】:**

不论是函数的原型还是定义,都要明确写出每个参数的类型和名字,不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数,那么使用 void 而不要空着,这是因为标准 C 把空的参数列表解释为可以接受

任何类型和个数的参数；而标准 C++ 则把空的参数列表解释为不可以接受任何参数。在移植 C++/C 程序时尤其要注意这方面的不同。

例如：

```
void SetValue(int width, int height);    // 良好的风格
void SetValue(int, int);                // 不良的风格
float GetValue(void);                  // 良好的风格
float GetValue();                      // 不良的风格
```

**【规则 6-2】：** 参数命名要恰当，输入参数和输出参数的顺序要合理。

若编写字符串拷贝函数 `StringCopy`，它有两个参数，如果把参数名字起为 `str1` 和 `str2`，函数原型为：

```
void StringCopy(char *str1, char *str2);
```

那么我们很难搞清楚究竟是把 `str1` 拷贝到 `str2` 中，还是相反。

应当把参数名字起得更更有意义，如叫 `strSource` 和 `strDestination`。这样从名字上就可以看出应该把 `strSource` 拷贝到 `strDestination`。

还有一个问题，这两个参数哪一个该在前，哪一个该在后？参数的顺序要遵循程序员的习惯。一般地，输出参数放在前面，输入参数放在后面，并且不要交叉出现输入输出参数。

如果将函数 `StringCopy` 声明为：

```
void StringCopy(char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式而导致出错：

```
char str[20];
StringCopy(str, "Hello World"); // 参数顺序颠倒
```

**【规则 6-3】：** 如果参数是指针，且仅做输入用，则应在类型前加 `const`，以防止该指针指向的内存单元在函数体内无意中被修改。

例如：

```
void StringCopy(char *strDestination, const char *strSource);
```

如果输入参数以值传递的方式传递对象，则宜改用“const &”方式来传递，因为引用的创建和销毁不会调用对象的构造和析构函数，从而可提高效率。

**【建议 6-1】：** (1) 应避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型和顺序搞错。此时，可以将这些参数封装为一个对象并采用地址传递或引用传递方式；

(2) 尽量不要使用类型和数目不确定的参数列表。C 标准库函数 `printf` 是采用不确定参数列表的典型代表，其原型为：

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

这种风格的函数在编译时丧失了严格的静态类型安全检查。

其实从本质上来讲,不存在什么传址调用和引用调用,所有调用都是传值调用,关键是看传递的是什么。例如下面的函数及其调用语句:

```
bool func(double *p);
double d = 100;
bool retVal = func(&d);
```

传递的是变量 `d` 的地址(假设为 `0x004dfe08`),而地址本身也是一个值。如果以指针来看的话,它仍然是传值调用;而如果以指针所指向的对象来看的话,就是传址调用(即引用调用)。在函数内部我们通过 `p`(值为 `0x004dfe08`) 引用到的对象就是 `d`。从二进制角度讲,地址(指针)才是内存单元真正的引用。至于引用调用,其本质(即在二进制层面上)就是传址调用,只是在源代码层面的使用方式与传址调用不同而已。

## 6.8 返回值的规则

我们已经看到了,从函数返回值有两种途径:使用 `return` 语句和使用输出参数。尤其是当需要从函数中同时返回不止一个结果的时候,仅使用 `return` 语句无能为力,只好借助输出参数。

**【规则 6-4】:** 不要省略返回值的类型。如果函数没有返回值,应声明为 `void` 类型。

在标准 C 语言中,凡不加类型说明的函数,一律自动按 `int` 类型处理。这样做不会有什么好处,却容易被误解为返回 `void` 类型。

C++语言有很严格的静态类型安全检查,不允许上述情况发生。由于 C++程序可以调用 C 函数,为了避免混乱,我们规定任何 C++/C 函数都必须有返回值类型。

**【规则 6-5】:** 函数名字与返回值类型在语义上不可冲突。

违反这条规则的典型代表是 C 标准库函数 `getchar()`。例如:

```
char c;
c = getchar();
if (c == EOF)
...
```

按照 `getchar` 名字的意思,将变量 `c` 声明为 `char` 类型是很自然的事情。但不幸的是, `getchar()` 的返回值类型的确不是 `char` 类型,而是 `int` 类型,其原型如下:

```
int getchar(void);
```

由于 `c` 是 `char` 类型,取值范围是 `[-128, 127)`,如果宏 `EOF` 的值在 `char` 的取值范围之外,那么 `if` 语句将总是失败,这种“危险”人们一般预料不到!导致本例错误的责任并不在用户,是函数 `getchar()` 误导了使用者。

**【建议 6-2】:** 不要将正常值和错误标志混在一起返回。建议正常值用输出参数获得，而错误标志用 **return** 语句返回。

回顾上例，C 标准库函数的设计者为什么要将 `getchar()` 声明为令人迷糊的 `int` 类型呢？他会那么傻吗？

在正常情况下，`getchar()` 的确返回单个字符（一般的可见字符的 ASCII 码值都是大于 0 的）。但如果 `getchar()` 碰到文件结束或发生读错误，它必须返回一个标志 EOF。为了区别于正常的字符，只好将 EOF 定义为负数（通常为 -1）。因此函数 `getchar()` 就成了 `int` 类型。

我们在实际工作中，经常会碰到上述令人为难的问题。为了避免误解，我们应该将正常值和错误标志分开，即正常值用输出参数获得，而错误标志用 **return** 语句返回。此外，你还可以使用另外一种方案：将正常情况下的返回值和错误标志绑定成一个值对（value pair）返回，比如 STL 库中 `std::map<>` 的 `insert()` 方法的返回值就是一个 `<iterator, bool>` 值对。

如果把函数 `getchar()` 改写成 `bool GetChar(char *c)`；就好多了。

虽然 `getchar()` 比 `GetChar()` 灵活，例如 `putchar(getchar())`，但是如果 `getchar()` 都用错了，它的灵活性又有什么用呢！

**【提示 6-3】:** 有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

例如字符串拷贝函数 `strcpy` 的原型：

```
char *strcpy(char *strDest, const char *strSrc);
```

`strcpy` 函数将 `strSrc` 拷贝至输出参数 `strDest` 中，同时函数的返回值又是 `strDest`。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];  
int length = strlen( strcpy(str, "Hello World") );
```

但是注意不要把返回指针的函数用做左值，例如：

```
double* func(double * p)  
{  
    return p;  
}  
double d= 100;  
*func(&d) = 200;
```

虽然编译器可以接受，但是最后这个语句非常难以理解，应避免类似写法。

**【提示 6-4】:** 如果函数的返回值是一个对象，有些场合下可以用“返回引用”替换“返回对象值”，这样可以提高效率，而且还可以支持链式表达。而有些场合下只能用“返回对象值”而不能用“返回引用”，否则会出错，见示例 6-4。

示例 6-4

```
class String
{
    ...
    // 赋值函数
    String& operator = (const String &assign);
    // 相加函数, 如果作为成员函数来重载, 则只许有一个参数
    friend String operator + (const String &lh, const String &rh);
private:
    char *m_data;
}
```

对于赋值函数, 应当用“返回引用”的方式返回 String 对象 (即 this 对象), 见示例 6-5。

示例 6-5

```
String& String::operator = (const String &assign)
{
    if (this == &assign)
        return *this;
    char *p = new char[strlen(assign.m_data) + 1];
    strcpy(p, assign.m_data);
    delete m_data;
    m_data = p;
    return *this;    // 返回的是 *this 的引用, 没有内容拷贝的过程
}
```

如果赋值函数采用“返回对象值”的方式, 虽然功能仍然正确, 但由于 return 语句要把 \*this 拷贝到保存返回值的外部存储单元之中, 增加了不必要的开销, 降低了赋值函数的效率 (如示例 6-6 所示)。

示例 6-6

```
String a, b, c;
...
a = b;           // 如果赋值函数采用“返回对象值”, 将产生一次 *this 拷贝
a = b = c;       // 如果赋值函数采用“返回对象值”, 将产生两次 *this 拷贝
```

String 的相加函数 operator + 的实现见示例 6-7。

示例 6-7

```
String operator + (const String &lh, const String &rh)
{
    String temp;
    temp.m_data = new char[strlen(s1.m_data) + strlen(s2.m_data) + 1];
```

```
strcpy(temp.m_data, s1.m_data);
strcat(temp.m_data, s2.m_data);
return temp;    //执行 string 对象及其字符串内容的拷贝
}
```

对于相加函数，应当用“返回对象值”的方式返回 String 对象，这将把局部对象 temp 及其真正的字符串值拷贝一份给调用环境的接收者。如果改用“返回引用”，那么函数返回值是一个指向局部对象 temp 的“引用”（即地址），而 temp 在函数结束时被自动销毁，将导致返回的“引用”无效，例如：

```
c = a + b;
```

此时 a + b 并不返回期望值，c 什么也得不到，于是留下了隐患。

## 6.9 函数内部实现的规则

不同功能的函数其内部实现各不相同，看起来似乎无法就“内部实现”达成一致的观点。但根据经验，我们可以在函数体的“入口处”和“出口处”从严把关，从而提高函数的质量。

**【规则 6-6】：** 在函数体的“入口处”，对参数的有效性进行检查。

很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”（assert）来防止此类错误（详见 6.12 节“使用断言”）。

**【规则 6-7】：** 在函数体的“出口处”，对 return 语句的正确性和效率进行检查。

如果函数有返回值，那么函数的“出口处”是 return 语句。我们不要轻视 return 语句。如果 return 语句写得不好，函数要么出错，要么效率低下。

注意事项如下：

（1）return 语句不可返回指向“堆栈内存”的“指针”或者“引用”，因为该内存单元在函数体结束时被自动释放，见示例 6-8。

示例 6-8

```
char * Func(void)
{
    char str[] = "hello world";           // str 数组创建在函数堆栈上，并用字符串
                                           // 常量来初始化，在末尾自动添加'\0'

    cout << sizeof(str) << endl;          // 12
    cout << strlen(str) << endl;          // 11
    return str;                           // 该语句存在隐患，str 指向的内存单元将被释放
}

但以下程序则是正确的：
const char * Func(void)
```



```
{
    const char *p = "hello world";           // 字符串常量存放在程序的静态数据区
                                              // 末尾自动添加'\0'
    cout << sizeof(p) << endl;               // 4
    cout << strlen(p) << endl;               // 11
    return p;                                // 返回字符串常量的地址
}
```

- (2) 要搞清楚返回的究竟是“对象的值”、“对象的指针”，还是“对象的引用”；  
(3) 如果函数返回值是一个对象，要考虑 `return` 语句的效率。例如：

```
return String(s1 + s2);
```

这是临时对象的构造函数语法，表示“创建一个临时对象并返回它”。不要以为它与“先创建一个局部对象 `temp` 并返回它的结果”是等价的，如：

```
String result(s1 + s2);
return result;
```

实则不然，上述代码将发生三件事情：首先，`result` 对象被创建，同时调用相应的构造函数完成初始化；然后调用拷贝构造函数，把 `result` 拷贝到保存返回值的外部存储单元中；最后，`result` 在函数结束时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编译器可以直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的开销，提高了效率。

类似地，我们不要把：

```
return (x + y);    // int x, y; 创建一个临时变量并返回它
```

写成：

```
int temp = x + y;
return temp;
```

由于内部数据类型如 `int`、`float`、`double` 的变量不存在构造函数与析构函数，虽然该“临时变量的语法”不会提高多少效率，但是程序更加简洁易读。

- 【建议 6-3】：** 函数的功能要单一，即一个函数只完成一件事情，不要设计多用途的函数。函数体的规模要小，尽量控制在 50 行代码之内。
- 【建议 6-4】：** 不仅要检查输入参数的有效性，还要检查通过其他途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。
- 【建议 6-5】：** 用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。
- 【建议 6-6】：** 尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易于理解又不利于测试和维护。在 C++/C 语言中，函数的 `static` 局部变量是函数的“记忆”存储器。建议尽量少用 `static` 局部变量，除非必需，例如，`static` 局部变量可以实现

全局常量的封装（见示例 6-9）。

示例 6-9

```
inline const char* GetWindowsPath(void)
{
    static const char *p = "C:\\\\Windows\\";
    return p;
}
```

## 6.10 存储类型及作用域规则

我们知道，任何可执行程序最终都会变成一系列的机器指令和数据。在把一个程序的源代码转变为机器指令序列的过程中，一些程序元素被剔除了，另一些程序元素则保留在可执行代码中。

被保留的程序元素将在最终的可执行程序中占用一定的存储空间。然而，不同的程序元素在运行时的存储方式不尽相同，这就引出了“存储类型”的概念。同时，不同的程序元素在编译阶段具有不同的作用范围，这就是标识符“作用域”的概念。在本节最后我们还要顺便介绍一下标识符“连接类型”的概念。

### 6.10.1 存储类型

标准 C 语言为变量、常量和函数等定义了 4 种存储类型，即：`extern`、`auto`、`static`、`register`，它们分别用一个关键字（存储类型说明符）来说明。一个程序元素的存储类型与它的作用域、生存期限及连接类型具有某种微妙的关系，但是一个具有作用域和连接类型的标识符不一定就具有存储类型。

这 4 种存储类型可分为两种生存期限：永久的（即在整个程序执行期间都存在）和临时的（即暂时保存在堆栈和寄存器中）。

`extern` 和 `static` 用来标识永久生存期限的变量和函数，而 `auto` 和 `register` 则用来标识临时生存期限的变量（注意，只有变量才能具有临时生存期限）。一个变量或函数只能具有一种存储类型，当然也只能有一种生存期限。

默认情况下，全局变量和全局函数的存储类型是 `extern` 的，能够被定义在它们之后的同一个编译单元内的函数所调用。如果变量和函数被显式地加上 `extern` 声明，那么其他编译单元中的函数也能调用它们。

显式地声明为 `static` 的全局变量和全局函数具有 `static` 存储类型，只能被同一个编译单元内的函数调用。

局部变量默认具有 `auto` 存储类型，除非用 `static` 或 `register` 来定义。但不管如何，它们的作用域都是程序块作用域，连接类型都是内连接，在进入函数的时候创建，在函数退出的时候销毁。`register` 和 `auto` 只能用于声明局部变量和局部常量。

全局常量的默认存储类型为 `static` 的，除非在定义了它的编译单元之外的其他编译单元中显式地用 `extern` 声明，否则不能被访问。

局部符号常量（注意，不是函数内出现的字面常量）的默认存储类型为 `auto`，除非显式地定义为 `static` 或 `register`。

函数的形参是局部变量，因此与一般局部变量的存储类型相同，但是最好不要声明为 `static` 的。

用 `register` 修饰的变量会被直接加载到 CPU 寄存器中，如果寄存器足以容纳得下它的话。把那些经常使用的变量例如循环计数器等直接放到 CPU 寄存器中，可以避免在寄存器和内存之间频繁地交换数据，因此能够提高程序的运行效率。现在有一些聪明的编译器，它们会对程序中出现的变量进行使用频率的评估，把使用频率较高、体积较小的变量直接放入 CPU 寄存器中，因此一般情况下不需要程序员显式地使用 `register` 说明符。

实际上还有一种存储类型，但它不是变量或函数的属性，而是存储空间的属性，那就是自由存储（或堆存储）。不过，这种存储类型与我们这里讨论的存储类型已经是完全不同的概念了，它的生存期限不属于上述任何一种，是自由的，即用户根据需要显式地分配和释放，它的生存期限就是分配和释放之间这一段时间。

## 6.10.2 作用域规则

所谓作用域就是一个标识符能够起作用的程序范围。在标准 C 语言中，这些范围包括文件、函数、程序块和函数原型，而在标准 C++ 中除了这 4 种外还有两个作用域类型：类和名字空间，其中名字空间是可以跨文件的。

标号（`label`）是具有函数作用域的惟一一种标识符。这就是说，不论你的标号定义在函数中的哪一行上，也不论定义在函数内嵌套得多么深的程序块内，它都能够在函数体内任何一个地方访问到，因此也叫做函数级的标识符。标号一般用在 `goto` 语句中，如果 `goto` 语句没有使用到该标号，那么该标号将被忽略。

有些人可能会说，局部变量的作用域应该是函数作用域啊！局部变量不是创建在函数堆栈上的吗？局部变量是创建在函数堆栈上不假，但是由于在嵌套的程序块中内层程序块定义的变量不能在其外层程序块中访问，所以局部变量不具有函数作用域，而是具有程序块作用域（由 `{}` 决定）。

例如，在函数中嵌套的程序块可以定义相同名字的变量，在内层的变量会遮蔽外层的同名变量。如果局部变量具有函数作用域的话，那么这两个同名标识符就会引起“重复定义”错误，这也有力地说明了局部变量不能具有函数作用域。即使局部变量的存储类型声明为 `static`，它仍然具有程序块作用域。

**【建议 6-7】：** 尽管语法允许，但是请不要在内层程序块中定义会遮蔽外层程序块中的同名标识符，否则会损害程序的可理解性。

当局部变量与某一个全局变量同名时，在函数内部将遮蔽该全局变量。此时在函数内部我们可以通过一元作用域解析运算符（`::`）来引用全局变量，例如：`::g_iCount++`。

在任何函数、类型定义及名字空间定义之外的标识符具有文件作用域，包括函数定义、类型定义本身，它们从其定义位置开始直到所在文件结尾都是可见的，因

此也叫做文件级的标识符。

函数原型中的形参名称是具有函数原型作用域的唯一一种标识符，不会与其他函数的同名形参名称冲突，该形参名称会被编译器忽略。但是函数定义中的形参是局部变量，具有程序块作用域。

C++ 的类不同于 C 的用户定义类型 (UDT)，它是有行为能力的抽象数据类型 (ADT)，但是 C++ 的类作用域概念其实包含了 C 的 UDT 成员的作用域概念。ADT/UDT 的所有成员都具有类作用域。在类作用域中，类的非静态成员函数可以直接访问类的其他任何成员；但在类作用域外，我们只能通过类的对象、对象指针及对象引用来访问类成员（这时类成员其实转换成了对象成员）。另外，成员函数中定义的局部变量具有程序块作用域。如果某一个成员函数内定义了与类的某一个数据成员同名的局部变量，那么这个局部变量将遮蔽该同名数据成员。此时，我们有两种方法来访问同名的数据成员：使用 `this` 和二元作用域解析运算符 `::`，例如 `this->flag = flag`；或 `ClassName::flag = flag`。

### 6.10.3 连接类型

连接类型分为外连接、内连接及无连接 3 种。连接类型表明了一个标识符的可见性，因此容易和作用域概念相混淆。

如果一个标识符能够在其他编译单元中或者在定义它的编译单元中的其他范围内被调用，那么它就是外连接的。外连接的标识符需要分配运行时的存储空间，见示例 6-10。

示例 6-10

```
void f(bool flag){ ... }           // 函数定义是外连接的
int g_int;                         // 全局变量 g_int 是外连接的
extern const int MAX_LENGTH = 1024; // MAX_LENGTH 变成外连接的
namespace NS_H
{
    long count;                   // NS_H::count 是外连接的
    bool g();                     // NS_H::g 是外连接的，但原型是内连接的
}
```

如果一个标识符能在定义它的编译单元中的其他范围内被调用，但是不能在其他编译单元中被调用，那么它就是内连接的（见示例 6-11）。

示例 6-11

```
static void f2(){ ... }           // f2 为内连接的
union                             // 匿名联合的成员为内连接的
{
    long count;
    char *p;
};
class Me{ ... };                  // Me 是内连接的
```

```
const int MAX_LENGTH = 1024;    // 常量为内连接的
typedef long Integer;           // typedef 为内连接的
```

一个仅能够在声明它的范围内被调用的名字是**无连接的**（见示例 6-12）。

示例 6-12

```
void f()
{
    int a;                // a 是无连接的
    class B{ ... };       // 局部类是无连接的，具有程序块作用域
}
```

为了更好地理解连接类型与作用域的区别，假设我们在两个编译单元中分别定义了同名全局函数 `function` 和同名的全局变量 `var`，见示例 6-13。

示例 6-13

```
// 文件 1: file1.cpp
long var = 1;
bool function(void)
{
    printf("Aha?");
    return true;
}
```

```
// 文件 2: file2.cpp
long var = 2;
bool function(void)
{
    printf("Oh! I think so.");
    return true;
}
```

那么，当你单独编译每一个源文件的时候可以正常通过，但是当你 Build 整个程序的时候就会“连接失败”。为什么呢？就是因为它们的连接类型都是外连接的，以及其存储类型为 `extern` 所致，结果导致连接出现二义性。这并不是它们的作用域所致。如果同名变量（或者同名函数）的其中一个定义为 `static` 的话（变成内连接的了），那么就可以正常连接了。

表 6-1 综合比较了程序元素的存储类型、作用域、生存期限及连接类型。

表 6-1 程序元素的存储类型、作用域、生存期和连接类型

程序元素	存储类型	作用域	生存期限	连接类型
全局 ADT/UDT 定义		文件		内连接
嵌套的 ADT/UDT 定义		类		内连接
局部 ADT/UDT 定义		程序块		无连接
非静态全局函数和全局变量	<code>extern</code>	文件	永久	外连接
静态全局函数和全局变量	<code>static</code>	文件	永久	内连接
局部非静态变量/常量	<code>auto</code>	程序块	临时	无连接
局部静态变量/常量	<code>static</code>	程序块	永久	无连接
静态全局常量	<code>static</code>	文件	永久	内连接

(续表)

程序元素	存储类型	作用域	生存期限	连接类型
非静态全局常量	C 和 C++有所不同			
类的静态成员	static	类	永久	内连接
类的非静态成员		类		内连接
名字空间的成员	不确定	名字空间	不确定	外连接
外部函数原型		文件		内连接
程序块中的函数原型		程序块		内连接
宏定义		文件		内连接

## 6.11 递归函数

一般情况下，我们都是使用层次分明的、没有环路的函数调用链来编写程序。函数除了能够嵌套调用外，还可以调用自己，这样的函数就是递归函数。递归函数的概念来源于数学领域函数的递归定义。常见的例子如乘幂  $x^n$ 、 $n!$ 、等差等比数列、斐波那契数列等：

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

$$a_n = \begin{cases} 1 & \text{假设等差数列首项 } a_0 = 1, \text{ 步长为 } d \\ a_{n-1} + d & n > 0 \end{cases}$$

$$\text{fibonacci}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & n > 1 \end{cases}$$

实际上，递归函数是通过解决基本问题进而解决复杂问题的。像上面这些数学问题中的  $n = 0$ （或  $n = 1$ ）的情况就是基本问题，或者叫做基本条件、终止条件；而  $n > 0$ （或  $n > 1$ ）的情况就叫做递归步，或叫做条件递归。递归步必须与原始问题相似，但规模要小于原始问题，这样才能使递归函数最终收敛于基本条件。也就是说，递归必须是有条件的递归，每次递归都要能简化原始问题并最终能够收敛于基本条件。如果出现无条件递归或递归不能简化原始问题的状况，就可能导致无限递归，最终导致程序崩溃（比死循环还要严重）。

递归函数的实现首先必须检测基本条件：如果基本条件出现，则返回基本值；否则修改规模并调用自己进入下一轮递归。我们以  $n!$  为例来实现其递归函数。程序见示例 6-14。

示例 6-14

```
#include <stdio.h>
unsigned long factorial(unsigned long n);
int main(void)
{
    unsigned int n;
    printf("Input a positive number: ");
    scanf("%u", &n);
    printf("%u! = %lu", n, factorial(n));
    return 0;
}
unsigned long factorial(unsigned long n)
{
    if(n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

只要问题能够用递归来定义, 就能够用递归函数来实现。递归函数为什么能够进行下去? 主要有 3 个原因:

(1) 函数在进入下一轮递归的时候并没有退出, 因此当前堆栈的内容并没有销毁, 从而每次递归进入和退出时能够保证有序地入栈和出栈操作, 不会混乱;

(2) 函数内的局部变量都是动态创建的, 即每次调用进入函数时才创建(压栈), 而当函数返回时才销毁(出栈), 因此能够保证每一次递归返回时局部变量有序地销毁;

(3) 函数堆栈是自动增长的, 理论上只要内存足够, 它就会按需增长, 直到达到最大堆栈限制为止(堆栈溢出)。

**【提示 6-5】:** 任何能够用递归函数实现的解决方案都可以用迭代来实现, 但是对于某些复杂的问题, 将递归方案展开为迭代方案可能比较困难, 而且程序的清晰性会下降。例如经典的“汉诺塔”问题。关于如何把递归函数展开为迭代, 请参考《数据结构》之类的书籍。

那么在什么情况下选择递归, 什么情况下选择迭代呢?

**【提示 6-6】:** 由于递归使用了函数的反复调用并占用了大量堆栈空间, 所以其运行时的开销非常大; 而迭代只是发生在一个函数内部, 反复使用局部变量进行计算, 因此运行时系统开销比递归要小得多。但是递归函数能够直观地反映使用递归方法定义的问题, 因此编写出来的代码易于理解和阅读。我们要在程序的性能与清晰性之间做一个选择。

例如, 计算费波纳契数列的第  $n$  项需要执行  $2^n$  次递归调用, 这种指数复杂度的



问题需要极大的时间和空间开销，当  $n$  很大时就可能使系统崩溃。

**【建议 6-8】：** 不要使用间接递归，即一个函数通过调用另一个函数来调用自己，因为它会损害程序的清晰性（见示例 6-15）。

示例 6-15

---

```
int f(int x)
{
    ...
    return g(x);
}
int g(int x)
{
    ...
    return f(x);
}
```

---

第 4 章的 4.10 节用循环结构来求解“两个整数的最大公约数和最小公倍数”，此处用递归函数来实现，见示例 6-16。

示例 6-16

---

```
#include <stdio.h>
// 求两个整数 a 和 b 的最大公约数
unsigned long gcd_2(unsigned long a, unsigned long b)
{
    if(b == 0) {
        return a;
    } else {
        return gcd_2(b, a % b);
    }
}
int main(void)
{
    unsigned long a, b;           /*两个整数*/
    unsigned long lcm = 0, gcd = 0; /*最小公倍数和最大公约数*/
    printf("Input two positive integers: ");
    scanf("%lu %lu", &a, &b);
    gcd = gcd_2(a, b);
    lcm = (a * b) / gcd;
    printf("Their Greatest Common Divisor is %lu.\n", gcd);
    printf("Their Least Common Multiple is %lu.\n", lcm);
    return 0;
}
```

---

## 6.12 使用断言

断言 (assert) 的语义如下: 如果表达式的值为 0 (假), 则输出错误消息并终止程序的执行 (一般还会出现提示对话框, 说明在什么地方引发了 assert); 如果表达式为真, 则不进行任何操作。因此断言失败就表明程序存在一个 bug。

C++/C 的宏 `assert(expression)` 就是这样的断言, 当表达式为假时, 调用库函数 `abort()` 终止程序。

程序一般分为 Debug 版本和 Release 版本, Debug 版本用于内部调试, Release 版本发行给用户使用。由于 `assert(expression)` 的宏体全部被条件编译伪指令 `#ifdef _DEBUG` 和 `#endif` 所包含, 因此 `assert()` 只在 Debug 版本里有效。

`assert` 不是一个仓促拼凑起来的宏。为了不在程序的 Debug 版本和 Release 版本中造成差别, `assert` 不应该带来任何副作用。所以 `assert` 不是函数, 而是宏。程序员可以把 `assert` 看成一个在任何系统状态下都可以安全使用的无害测试手段。所以不要把程序中的 `assert` 语句删除掉。

**【提示 6-7】:** 如果程序在 `assert` 处终止了, 并不是说含有该 `assert` 的函数有错误, 而是调用函数出了差错, `assert` 可以帮助我们追踪到错误发生的原因。

**【建议 6-9】:** 在函数的入口处, 建议使用断言来检查参数的有效性 (合法性)。

例如内存拷贝函数 `memcpy` (见示例 6-17), 如果 `assert` 的表达式为假, 那么程序就会中止。这可以检查你的程序对 `memcpy()` 函数的调用是否正确。

示例 6-17

```
void * memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    // 使用断言, 防止 pvTo 或 pvFrom 为 NULL
    assert((pvTo != NULL) && (pvFrom != NULL)); typedef char byte;
    byte *pbTo = (byte *)pvTo;           // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *)pvFrom;       // 防止改变 pvFrom 的地址
    while (size-- > 0) {                  // 不要与字符串拷贝混淆!
        *pbTo++ = *pbFrom++;
    }
    return pvTo;
}
```

很少有比跟踪到程序的断言却不知道该断言的作用更让人沮丧的事了, 花了很多时间, 不是为了排除错误, 而只是为了弄清楚这个错误到底是什么。有的时候, 程序员偶尔还会设计出有错误的断言。所以, 如果搞不清楚断言检查的是什么, 就很难判断错误是出现在程序中, 还是出现在断言中。

**【建议 6-10】:** 请给 `assert` 语句加注释, 告诉人们 `assert` 语句究竟要干什么。

幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。

**【规则 6-8】:**

要注意 `assert` 语句仅仅在 `Debug` 版本中才有效，而在 `Release` 版本中无效。如果在 `Debug` 版本中 `assert` 没有捕捉到非法情况（即表达式为真），那么表明在 `Release` 版本的对应处程序也是合法的，所以不必再检查合法性。但是合法的程序并不见得就是正确的程序。

使用断言的目的是捕捉在运行时不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是程序运行过程中自然存在的并且是一定要主动做出处理的。

举个常见的例子来解释。用 `malloc` 申请动态内存时，如果系统没有足够的内存可用，那么 `malloc` 返回 `NULL`。动态内存申请失败不是非法情况，而是错误情况。所以我们要用 `if` 语句捕捉错误情况并给出错误处理代码，而不应该使用 `assert`（否则在 `Release` 版本失效），见示例 6-18。

示例 6-18

// 错误的示例	//正确的示例
<pre>int *pBuf = (int *)malloc(sizeof(int) * 1000); assert(pBuf != NULL); //错误地使用 assert ... // do something</pre>	<pre>int *pBuf = (int *)malloc(sizeof(int) * 1000); if(pBuf == NULL) {     ... // 错误处理代码 }else {     ... // do something }</pre>

**【提示 6-8】:**

一般教科书鼓励程序员们进行防错性程序设计，但要知道这种编程风格可能会隐瞒错误。当进行防错性设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

**【提示 6-9】:**

要区分断言（`assert`）和跟踪语句（`tracer`）的不同。后者是指一些用于报告程序执行过程中当前状态的输出语句，但它们并不一定就是 `bug`。

**【提示 6-10】:**

程序通过了断言的检查，并不保证就万无一失了。例如 `memcpy()` 中的断言，如果给它传入两个未初始化的野指针（地址不为 0 但是没有指向合法的内存块），那么 `assert()` 就失去了作用。

## 6.13 使用 const 提高函数的健壮性

看到 `const` 关键字, C++ 程序员首先想到的可能是 `const` 常量, 这可不是良好的条件反射。如果只知道用 `const` 定义常量, 那么相当于把火药仅用于制作鞭炮。`const` 更大的魅力是它可以修饰函数的参数和返回值, 甚至函数的定义体。

`const` 是 `constant` 的缩写, 是“恒定不变”的意思。被 `const` 修饰的东西都受到 C++/C 语言实现的静态类型安全检查机制的保护, 可以预防意外修改, 能提高程序的健壮性。所以很多 C++ 程序设计书籍建议: “Use `const` whenever you need”。

### 6.13.1 用 `const` 修饰函数的参数

如果参数用于输出, 不论它是什么数据类型, 也不论它采用“指针传递”还是“引用传递”, 都不能加 `const` 修饰, 否则该参数将失去输出功能。`const` 只能修饰输入参数。

**【建议 6-11】:** 如果输入参数采用“指针传递”, 那么加 `const` 修饰可以防止意外地改动该指针指向的内存单元, 起到保护的作用。

例如 `StringCopy` 函数:

```
void StringCopy(char *strDest, const char *strSrc);
```

其中 `strSrc` 是输入参数, `strDes` 是输出参数。给 `strSrc` 加上 `const` 修饰符后, 如果函数体内的语句试图改动 `strSrc` 指向的内存单元, 编译器将指出错误。

如果还想保护指针本身, 则可以声明指针本身为常量, 防止该指针的值被改变, 例如:

```
void OutputString(const char * const pStr);
```

**【提示 6-11】:** 如果输入参数采用“值传递”, 由于函数将自动用实参的拷贝初始化形参, 因此即使在函数内部修改了该参数, 改变的也只是堆栈上的拷贝而不是实参, 所以一般认为不需要用 `const` 修饰。

虽然形参在语义上等价于局部变量, 但是由于它包含了特殊的初值, 所以与普通的局部变量还是不能“等价齐观”的。如果函数内部不止一次地使用形参的初值, 那么在该形参前面加 `const` 也就顺理成章了。

例如可把函数 `void Func1(int value)` 写成 `void Func1(const int value)`, 如果 `Func1` 内部多次使用 `value` 的初值的话, 可以保证写出的代码不会无意中修改它。

对于 ADT/UDT 的参数而言, 像 `void Func(A a)` 这样声明的函数注定效率比较低。因为函数体内将产生 `A` 类型的临时对象用于拷贝参数 `a`, 而临时对象的构造、拷贝、析构过程都将消耗时间。

为了提高效率, 可以将函数声明改为 `void Func(A &a)`, 因为“引用传递”仅借

用一下参数的别名而已（本质上是传递地址），不需要产生临时对象。但是函数 `void Func(A &a)` 存在一个缺点：“引用传递”有可能改变参数 `a`，这是我们不期望的。解决这个问题很容易，加 `const` 修饰即可，因此函数最终成为 `void Func(const A &a)`。

依次类推，是否应将 `void Func(const int x)` 改写为 `void Func(const int &x)`，以便提高效率呢？完全没有必要！因为基本数据类型的参数不存在构造、析构的过程，而拷贝也非常快，“值传递”和“引用传递”的效率几乎相当。

问题是如此地纠缠不清，我们只好将“`const &`”修饰输入参数的用法总结一下。

**【提示 6-12】** 对于 ADT/UDT 的输入参数，应该将“值传递”改为“`const &`传递”，目的是提高效率。例如，将 `void Func(A a)` 改为 `void Func(const A &a)`。  
对于基本数据类型的输入参数，不要将“值传递”的方式改为“`const &`传递”，否则既达不到提高效率的目的，又让人费解。例如，不要把 `void Func(const int x)` 改为 `void Func(const int &x)`。

### 6.13.2 用 `const` 修饰函数的返回值

如果给“指针传递”的函数返回值加 `const` 修饰符，那么函数返回值是一种契约性常量，不能被直接修改，并且该返回值只能被赋值给加 `const` 修饰的同类型指针（除非强制转型）。例如函数：

```
const char * GetString(void);
```

则如下语句将出现编译错误：

```
char *str = GetString();
```

正确的用法是：

```
const char *str = GetString();
```

如果函数返回值采用“值传递”的方式，在一般情况下由于函数会把返回值拷贝到外部临时的存储单元中，所以加 `const` 修饰没有什么意义。例如函数：

```
int GetInt(void);
```

一般作为右值来调用：

```
int x = GetInt();
```

但是当函数以值传递方式返回 ADT/UDT 对象、返回引用或者返回指针时，有人可能会把该函数作为左值来调用，例如如下调用 `A GetA(void)`：

```
A x;  
GetA() = x;
```

显然不是好风格。为了防止这种错误，可以用 `const` 来修饰返回值，即：

```
const A GetA(void);
```

其中 `A` 为用户自定义的数据类型。如此一来，编译器就可以检查出上述错误。但是

这种情况不是很常见，所以不必拘泥于此。

函数返回值采用“引用传递”的方式并不常见，这种方式一般出现在类的赋值函数中，目的是实现链式表达式，见示例 6-19。

示例 6-19

---

```
class A{
public:
    ...
    A& operate=(const A &other);    // 赋值函数
};
A a, b, c;                          // a、b、c 为 A 的对象
...
a = b = c;                          // 正常的链式赋值
(a = b) = c;                        // 不正常的链式赋值，但合法
```

---

如果将赋值函数的返回值加 `const` 修饰，那么该返回值的内容不允许被修改。上例中，语句 `a = b = c` 仍然正确，但是语句 `(a = b) = c` 则是错误的。

虽然函数返回值采用“`const &`”方式能提高效率，例如：

```
const A& GetA(void);
```

但是千万要小心，一定要搞清楚函数究竟是想返回一个对象的“拷贝”还是仅返回“别名”（即引用）就可以了，否则程序会出错的。

## 第 7 章 C++/C 指针、数组和字符串

在 C++/C 程序中一定会用到指针、数组和字符串，但是有许多人无法非常清晰地说出它们之间的区别和联系，总是处于一种糊里糊涂的状态之中，因此产生了许多认识错误和编程错误。本章通过分析指针、数组、字符串、引用的内存映像，来澄清这些概念。

很多程序员惧怕指针，其实不是指针本身有什么可怕的，而是因为没有掌握使用指针的正确方法，导致了各种“百思不得其解”的问题，从而吓住了他们。如果程序员不了解指针的本质，不能熟练使用指针的话，那么他就无法管理动态内存，很难想象这样的程序会带来什么质量问题。。

### 7.1 指针

#### 7.1.1 指针的本质

我们知道，可执行程序是由指令、数据和地址组成的。当 CPU 访问内存单元时，不论是读取还是写入，首先必须把内存单元的地址加载到地址总线（AB）上，同时将内存电路的“读写控制”设为有效，然后内存单元中的数据就通过数据总线（DB）“流”向了接收寄存器中，或者结果寄存器中的值“流”向了目标内存单元中。这就是一个“内存读写周期”。

内存单元的地址就是我们马上要讲的“指针”的值。首先，指针是变量，它和我们平常使用的整型变量、字符变量、浮点变量等各种变量并没有本质的差异，不同的只是它们的类型和值的含义，即解释方式。在二进制层面，指针的值就是内存单元的地址，而变量又是引用内存单元的值的别名，因此在语言层面指针的值就是变量的地址。我们使用下面的程序（运行于 32 位系统）来打印一些变量的地址以探究其中的秘密（见示例 7-1）。

示例 7-1

```
#include <stdio.h>
#include <iostream>
using namespace std;
int main(void)
{
    int    i = 100;
    char   c = 'a';
    float  f = i;
    double d = f;
```



```

bool    b = true;
int *pI = &i;      // 声明指针变量并初始化为 i 的地址
int **ppI = &pI;   // 指针的指针
cout << "Address of i      : 0x " << &i << endl;
printf("Address of c      : 0x %p\n", &c);
cout << "Address of f      : 0x " << &f << endl;
cout << "Address of d      : 0x " << &d << endl;
cout << "Address of b      : 0x " << &b << endl;
cout << "Value of pI       : 0x " << pI << endl;
cout << "Address of pI     : 0x " << &pI << endl;
cout << "Value of ppI      : 0x " << ppI << endl;
cout << "Address of ppI    : 0x " << &ppI << endl;
return 0;
}

```

输出结果:

```

Address of i      : 0x 0012FF64
Address of c      : 0x 0012FF60
Address of f      : 0x 0012FF5C
Address of d      : 0x 0012FF54
Address of b      : 0x 0012FF50
Value of pI       : 0x 0012FF64
Address of pI     : 0x 0012FF4C
Value of ppI      : 0x 0012FF4C
Address of ppI    : 0x 0012FF48

```

观察输出结果。可以看出，指针的值是 32 位的正整数。变量 `pI` 的值就等于整型变量 `i` 的地址，而 `pI` 本身的地址则为 `0x0012FF4C`；`ppI` 的值就是 `pI` 的地址，而 `ppI` 本身的地址则是 `0x0012FF48`；如此下去。我们甚至可以声明 `N` 级指针。图 7-1 表示了 `i`、`pI`、`ppI` 的内存映像及它们之间的关系。

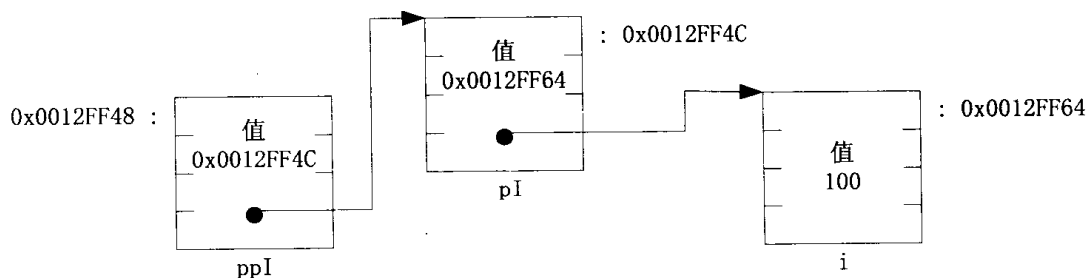


图 7-1 指针变量与所指对象的内存映像

只要我们始终掌握着对象的内存地址（也就是它的指针），我们就可以在任何地方对其指向的内存单元进行操作，无论是读写其数据还是调用其函数。这种灵活性正是指针危险（可能指向无效内存对象）和不易掌握的主要原因。如果你访问了非法的或无效的内存单元，就会导致运行时错误。

### 7.1.2 指针的类型及其支持的运算

指针的类型为一个类型名和字符“\*”的组合，我们平时说“一个 int 类型的指针”，其实是在说“一个类型为 int\* 的指针”。编译器认为这样的指针指向的内存单元为 4 字节，并将其内容解释为 int 类型的值。

当一个类型名和字符“\*”结合以声明变量的时候，就产生了一种新的类型，可以利用这种类型来声明一个指针（见示例 7-2）。

示例 7-2

```
int *pInt;
int **ppInt;
int ***pppInt;
char *pChar;
void *pVoid;
```

pInt 的类型就是 int\*，ppInt 的类型就是 int\*\*。可见并不存在一种固定的“指针类型”（不像 int、long、char 等是固定类型），不同的类型和“\*”组合起来就可以产生不同的指针类型。这也许就是语言本身不直接支持指针的类型名字的原因吧！我们可以使用 typedef 把不同的指针类型定义下来，然后就可以使用这些自定义“类型名”来定义成批的指针变量（见示例 7-3）。

示例 7-3

```
typedef int *      IntPtr;
typedef int **     IntPtrPtr;
typedef IntPtrPtr* IntPtrPtrPtr;
typedef char *     CharPtr;
typedef void *     VoidPtr;
typedef CharPtr*   CharPtrPtr;

IntPtr    pa, pb, pc;
IntPtrPtr ppa, ppb, ppc;
CharPtr   pChar1, pChar2;
VoidPtr   pVoid;
CharPtrPtr ppChar1, ppChar2;
```

#### 【提示 7-1】:

虽然类型名和“\*”的组合是一种指针类型，但是编译器解释的时候，“\*”是和其后的变量名结合的。例如：

```
int* a, b, c;
```

编译器会理解为：

```
int *a, b, c;
```

即只有 a 是 int 类型的指针，而 b 和 c 仍然是 int 类型的变量。

全局指针变量的默认初始值是 NULL。而对于 non-static 局部指针变量 p，必须显式地指定其初值，否则 p 的初始值是不可预测的（不是 NULL）。当你第一次使用它的时候就可能会用 if (p != NULL) 来检查 p 的有效性，然而此时 if 语句的确不起作用。如果你忘记了给 p 赋初值，那么你第一次使用 p 的时候就会导致运行时错误。

**【规则 7-1】：** 不管指针变量是全局的还是局部的、静态的还是非静态的，应当在声明它的同时初始化它，要么赋予它一个有效的地址，要么赋予它 NULL。

既然指针的值是 int 类型，那么就可以给它赋以任何整数值，包括负值，见示例 7-4。

示例 7-4

```
int *pInt = NULL;
pInt = reinterpret_cast<int*>(-0x0024FFE4); // 把负数解释为地址
cout << pInt << endl;                    // FFDB001C
cout << reinterpret_cast<unsigned int>(pInt); // 4292542492
cout << UINT_MAX << endl;                 // unsigned int 的最大值: 4294967295
```

4292542492 的十六进制表示正是 FFDB001C。可见，编译器并不是把指针值解释为 signed int 类型数，而是解释为 unsigned int 类型数。同样的道理，指针可以参与整数能够参与的任何算术运算，但是除了下列的运算外，其他运算都是没有实际意义的。

- (1) 指针自增 (++), 表示它指向了序列中的后一个元素。
- (2) 指针自减 (--), 表示它指向了序列中的前一个元素。
- (3) 指针加一个正整数 i, 表示它向后递进 i 个元素。
- (4) 指针减一个正整数 i, 表示它向前递进 i 个元素。
- (5) 两个同类型指针相减, 表示计算它们之间的元素个数。
- (6) 指针赋值, 把一个指针值指派给另一个指针。
- (7) 指针比较 (>、<、==、!=、>=、<=), 最常用的是==和!=。
- (8) 取地址 (&) 和反引用 (\*).

**【提示 7-2】：** 指针加/减一个正整数 i, 其含义并不是在其值上直接加/减 i, 还要包含指针所指对象的字节数信息。例如：

```
int *pInt = new int[100];
pInt += 50; // 编译器改写为 pInt += 50 * sizeof (int);
pInt++;    // 即 pInt += 1;
```

因此, void\* 类型的指针不能参与算术运算, 只能进行赋值、比较和 sizeof() 操作。

**【提示 7-3】：** (1) 当把 “&” 用于指针时, 就是在提取指针变量的地址。不能在一个指针前面连续使用多个 “&”, 例如 &&p, 因为 &p 已经不是一个变量了, 不能把 “&” 单独用于一个非变量的东西。不能对字面常量使用 “&” 来取其地址, 因为字面常量保存在符号表中, 它们没有地址概念;

(2) 当把 “\*” 用于指针时, 就是在提取指针所指向的变量。因此在将 “\*” 用于指针时一定要确保该指针指向一个有效的和合法的变量。不能对 void\* 类型指针使用 “\*” 来取其所指向的变量。

### 7.1.3 指针传递

我们可以把函数的参数或返回值类型声明为指针类型。此时, 函数接受的参数或返回值就是地址, 而不是指针所指的内存单元的值, 见示例 7-5。

示例 7-5

```
void f( int * p )
{
    cout << p << endl;      // 输出局部指针变量 p 的值
    cout << &p << endl;     // 输出 p 在堆栈上的地址
    cout << *p << endl;     // p 所指内存单元的值 0
    *p = 100;                // 修改 p 所指内存单元 (即 iCount) 的值
    cout << *p << endl;     // 100
    p = NULL;
}
int iCount = 0;
f(&iCount);
cout << iCount << endl;    // 100
```

假设 iCount 的地址为 0x0024FF42, 则在调用语句 f(&iCount) 中传入的就是这个地址值。至于 p 的地址则是堆栈地址。总之, 指针传递其实是在传递一个地址而不是该地址指向的对象。

顺着这样的思路, 不难理解双指针的传递和返回问题, 无非是比单指针又多增加了一层间接性而已。如果你还是理解不了, 建议你使用指针的引用, 见示例 7-6。

示例 7-6

```
void Allocate(char *& p, int size)
{
    p = (char *) malloc (size);
}
void Test(void)
{
    char *str = NULL;
    Allocate (str, 100);
    strcpy (str, "Hello World!");
    printf (str);
    free (str);
}
```

【提示 7-4】:

我们可以把一个对象的地址在整个程序中的各个函数之间传来传去，只要保证每次使用时它都指向合法的和有效的内存单元。这就是指针容易被错误使用的一个原因：在指针传递的过程中，你可能把它指向的内存释放掉了却还继续使用该指针，或者传递了一个局部对象的地址，而该对象在函数结束时就销毁了。

7.2 数组

7.2.1 数组的本质

任何数组，不论是静态声明的还是动态创建的，其所有元素在内存中都是连续字节存放的，也就是说保存在一大块连续的内存区中。我们在后面讲到泛型容器的时候会看到 `vector<T>` 的所有元素对象（类型为 `T`）在内存中也是连续存放的，只不过 `vector` 是一种可变长度的数组。

图 7-2 表示数组 `int a[10]` 的内存映像（其中的地址排列可能与平台有关，这里仅是示意）。

0x004284F0:	11	←a[0]
0x004284F4:	28	←a[1]
0x004284F8:	-5	←a[2]
0x004284FC:	45	←a[3]
	⋮	
0x00428514:	19	←a[8]
0x00428518:	-40	←a[9]

图 7-2 一维数组的内存映像

数组元素的下标编号从 0 开始，最后一个元素的下标等于元素个数减 1。下标必须是整数或整数的表达式。通过下标引用一个数组的元素，在本质上和引用一个同类型的变量没什么区别，编译器通过下标值来计算你所引用的元素在内存中的地址。因此，在语义上，下标操作符返回的是一个元素的引用。例如：

```
a[3] = 100;
```

编译器计算地址 `a + 3 * sizeof(int)`，得到 `0x004284FC`，并返回该地址所指对象的引用而不是返回“45”这个值。它在语义上等价于如下的语句：

```
int &ri = a[3];  
ri = 100;
```

**【提示 7-5】:**

其实，当你使用“[]”来引用数组元素的时候，编译器必须把它转换为同类型的指针表示形式，然后再进行编译。例如：

```
a[3] = 100;           // 转换为 *(a + 3) = 100;  
cout << a[3] << endl; // 转换为 cout << *(a + 3) << endl;
```

这是因为 C++/C 数组本身不会保存下标值与元素对象之间的对应关系，因此无法直接通过下标索引来定位真正的数组元素对象。如果在程序中直接使用指针方式，虽然可以缩短编译时间，但是会损害程序的清晰性，所以要使用“[]”。

数组名字本身就是一个指针，是一个指针常量，即 `a` 等价于 `int * const a`，因此你不能试图修改数组名的值。数组名的值就是数组第一个元素的内存单元首地址，即 `a == &a[0]`，在图 7-2 中 `a` 为 `0x004284F0`。这样，我们就可以通过同类型的指针迭代（++/--）来遍历整个数组，但是你不能妄想仅通过数组名（不使用下标和迭代）就想达到访问整个数组的目的，除非它是带有“\0”结束符的字符数组（即字符串）。基于这个原因，任何两个数组之间不能直接赋值，即使是同类型数组，示例 7-7 的做法是错误的。

示例 7-7

```
int a[10] = {0};  
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
a = b;    // 不仅语义不对，何况 a 也不能被修改
```

要想把 `b` 的内容指派给 `a`，必须按元素逐个赋值，或者使用内存拷贝函数 `memcpy()`。

在声明数组时，一般有如下 3 种方式：

- （1）明确地指出它的元素个数，编译器会按照给定的元素个数来分配存储空间；
- （2）不指定元素个数而直接初始化，编译器会根据你提供的初始值的个数来确定数组的元素个数；
- （3）同时指定元素个数并且初始化。但是不允许既不指定元素个数，又不初始化，因为编译器不知道到底该为数组分配多少存储空间。

**【提示 7-6】:**

编译器在给数组分配内存空间的时候总是以用户指定的元素个数为准。如果初始值的个数多于指定的个数，则报错；如果用户没有指定元素个数，则编译器计算初始值的个数作为数组的元素个数；如果初始值个数小于指定的数组元素个数，则后面的元素全部自动初始化为 0。

示例 7-8 展示了初始化数组的方法。

示例 7-8

```
int b[100];           // sizeof(b) = 400 bytes, 未初始化
int c[] = {1, 2, 3, 4, 5}; // 元素个数为 5, sizeof(c) = 20 bytes, 初始化
int d[5] = {1, 2, 3, 4, 5, 6, 7}; // 错误! 初始值越界
int e[10] = {5, 6, 7, 8, 9}; // 元素个数为 10, 指定了前 5 个元素的初始值, 剩
                             // 下的元素全部自动初始化为 0
int f[10] = {5, , 12, , 2}; // 错误! 不能跳过中间的某些元素
```

不存在元素个数为 0 的数组，这是很显然的。但是存在元素个数为 0 的容器，即空容器（参见本书第 17 章）。

标准 C++/C 不会对用户访问数组是否越界进行任何检查，无论是静态的（编译时）检查还是动态的（运行时）检查。

**【提示 7-7】：** 虽然数组自己知道它有多少个元素，但是由于可以使用整型变量及其表达式作为数组的下标，而变量的值在编译时是无法确定的，所以语言无法执行静态检查。

从理论上讲，C++/C 可以在运行时进行数组的越界访问检查，这是因为数组大小的信息保存在程序中的某个地方，一般就是放在数组第一个元素位置的前面，占用一个 int 变量的字节数，它的地址为 `a-sizeof(int)`。但是如果在每次访问数组元素的时候都要去这个地方取数组的大小并与下标值比较，这会极大地增加运行时开销（增加了代码长度和运行时时间）。另外，C++/C 允许通过同类型指针来访问数组的每一个元素，此时又如何确定指针是否越界了呢？显然，毫无办法，因为指针除了它的类型和值外，不带有任何其他信息。所以 C++/C 不会对用户访问数组是否越界进行任何检查。

因此，你可以访问超出数组范围的元素。换句话说，你可以把数组存储空间以外的内存单元当做这个数组的元素来访问，例如 `a[15] = 100;`，系统可能不会报错，那是因为 `a[15]` 可能恰好是位于一块空闲内存中，但是由此造成的后果要由你的程序来承担。

## 7.2.2 二维数组

二维数组在 C++/C 中都是以“行序优先”来存储元素的，而在 Fortran 中则是以“列序优先”来存储的。图 7-3 是二维数组 `int a[5][3]` 的内存映像。



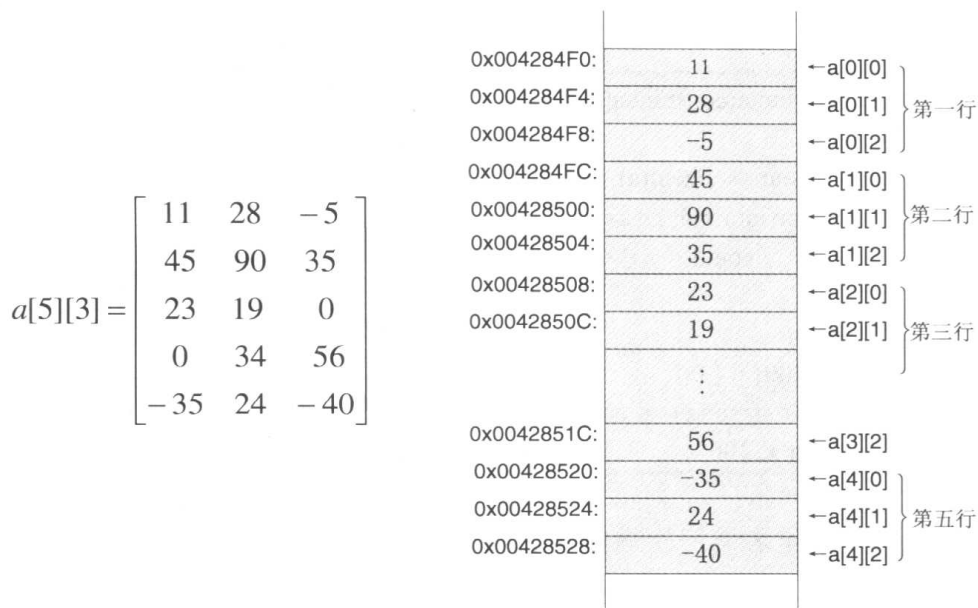


图 7-3 二维数组的内存映像

当我们访问数组元素 `a[4][1]` 时，实际上编译器转换为 `*((a + 4 * 3) + 1)`，其中 3 就是该数组的列数，而 4 和 1 是用户提供的下标值。在访问二维数组中的某个元素时，编译器为了计算其地址必须知道该数组的列数，但并不需要知道这个数组总共有多少行。

**【提示 7-8】:** 数组实际上也是一个可以递归定义的概念：任何维数的数组都可以看做是由比它少一维的数组组成的一维数组。例如 `int a[3][4][5]` 可以看做是由二维数组 `int b[4][5]` 组成的一维数组，其长度为 3；而 `b[4][5]` 又可以看做是由一维数组 `int c[5]` 组成的一维数组，其长度为 4；`c[5]` 则是 `int` 元素的数组，其长度为 5。

数组和指针之间存在如下的等价关系。

(1) 一维数组等价于元素的指针，例如：

`int a[10] ⇔ int * const a;`

(2) 二维数组等价于指向一维数组的指针，例如：

`int b[3][4] ⇔ int (* const b)[4];`

(3) 三维数组等价于指向二维数组的指针，例如：

`int c[3][4][5] ⇔ int (* const c)[4][5];`

由于多维数组的概念与二维数组的相似，本节不再多讲。关于二维或多维数组的遍历方法，请参考本书 4.10.2 节。

7.2.3 数组传递

数组是不能从函数的 `return` 语句返回的，但是数组可以作为函数的参数，见示例 7-9。

示例 7-9

```
void output(const int a[ ], int size)
{
    cout << sizeof(a) << endl;    // 是 4 而不是 400
    for(int i = 0; i < size; i++)
        cout << a[i] << endl;
}
// main
int x[100] = { 0 };
cout << sizeof(x) << endl;    // 400
output( x, 100 );
```

**【提示 7-9】:**

数组名实际上就是该数组的首地址。把数组作为参数传递给函数的时候并非把整个数组的内容传递进去，此时数组退化为一个同类型的指针。也就是说，`output` 函数原型中声明的数组并非真正的数组，编译器会把它改写为 `void output (const int * const a, int size)`。当你在函数内部使用下标来访问数组元素的时候，编译器把它转换为 `*(a + i)`。

C++/C 为什么要把数组传递改写为指针传递呢？主要原因如下：

- ◇ 数组在内存中是连续字节存放的，因此编译器可以通过地址计算来引用数组中的元素；
- ◇ 出于性能考虑。如果把整个数组中的元素全部传递进去，不仅需要大量时间来拷贝数组，而且这个拷贝还会占用大量的堆栈空间。

对于多维数组传递，情况有所不同，你必须指出除第一维之外的所有维的长度（元素个数）。我们以二维数组为例，见示例 7-10。

示例 7-10

```
void output(const int a[ ][20], int line)
{
    cout << sizeof(a) << endl;    // 4
    for( int i = 0; i < line; i++ ) {
        for( int j = 0; j < 20; j++ ) {
            cout << a[i][j] << endl;
        }
    }
}
// main
int x[10][20] = { { 100 } };
cout << sizeof(x) << endl;    // 800
output( x, 10 );
```

**【提示 7-10】:** 对于多维数组，C++/C 并不像一维数组那样可以简单地转换为同类型指针，而是转换为与其等价的数组指针。例如 `int a[m][n]` 就转换为 `int (*a)[n]`，就是说 `a` 是一个指向一维数组的指针，而该一维数组具有  $n$  个元素，`a` 是指向原来数组的第一行的指针，`a+1` 就是指向第二行的指针，依次类推。下面的 4 种表达方法是等价的：

- (1) `a[i][j]`
- (2) `*(a[i] + j)`
- (3) `*(a + i)[j]`
- (4) `*(*(a + i) + j)`

要注意，上述 4 个表达式中的 `a` 是指向一维数组的指针，而不是单纯地指向 `int` 类型元素的指针，因此 `*(a + i) + j` 的值实际上是 `((a + i * sizeof(int) * n) + j * sizeof(int))`。基于上述认识，`output()` 函数实际上就被转换成了：

```
void output(const int (*a)[20], int line);
```

因此函数内部对 `a[i][j]` 的访问转换为 `*(a + (i * 20 + j) * sizeof(int))`。

**【提示 7-11】:** 为什么在向函数传递多维数组的时候不需要说明第一维的长度而必须说明其他所有维的大小呢？原因有三：

- (1) 数组元素在内存中按照“行优先”规则存储，所以需要列数来确定一行有多少个元素；
- (2) 编译器在计算元素的地址时不需要数组第一维的长度，但是需要所有其他维的长度信息，参见【提示 7-8】；
- (3) C++/C 不对数组进行越界访问检查，因此对编译器来说不需要知道第一维的长度。

**【提示 7-12】:** 数组传递在 C++/C 中默认就是地址传递。如果你想按值来传递数组，可以把数组封装起来，例如放到 `struct` 或 `class` 里面作为一个成员，因为结构和类对象默认都是按值传递的。

#### 7.2.4 动态创建、初始化和删除数组的方法

数组尤其是多维数组，其动态创建、初始化和删除的正确使用方法不像静态分配的或自动分配的数组那样轻松。

数组类型和指针类型具有如下对应关系：

```
ElemType a[m][n][o]...[x][y][z]
⇔
ElemType (*const pa)[n][o]...[x][y][z]
```

多维数组的使用必须遵从这个等价关系才能做到类型安全。

对于字符数组，其动态创建的正确方法为：

```
char *p = new char[1025];    // 分配空间
```

...

动态创建的字符数组的释放要注意：因为对于字符数组来说，它并不在乎自己的结尾有无'\0'结束标志，所以不能把它当做字符串。正确的释放方法是：

```
delete []p;    // 删除数组空间
```

它不会删除不属于它的内存单元，也不会泄漏哪怕是一个字节的内存单元。因为你明白地告诉了编译器：这是在释放一个字符数组，“请”它去取 p 指向的字符数组的大小信息（数组的元素个数，它被编译器保存在程序的某个地方了），然后按照这个大小来释放动态内存。这就是数组的释放(delete[])的语义。

对于其他类型的一维数组，其创建的方法与一维字符数组相同，释放时必须使用 delete []，否则就会造成内存泄漏。例如：

```
int *q = new int[1024];
```

.....

此时，如果你使用：

```
delete q;
```

来释放 q 指向的 1024 个 int 内存空间，结果将是：它只释放了该数组的第一个元素的空间，后面的 1023 个 int 空间全丢失了！这是因为：编译器不会把 q 看做是指向 int 类型数组的指针，而仅仅是一个 int 指针。从这里也可见字符指针和字符数组之间的微妙关系。删除 q 的正确方法应该是：

```
delete []q;    // 正确的数组释放语义
```

根本原因在于：delete 语句执行时确认 q 是否指向一个数组以及查找数组的大小信息会极大地影响它的执行效率，所以 C++ 做了一个妥协，即“只有在 '[' 出现在指针前面时编译器才会去某个地方（某个固定位置或者关联数组）查找数组的大小信息，否则它便假设用户只删除一个对象”。

多维数组的动态创建稍复杂些：你不能简单地使用一个元素类型的指针来接收动态创建的多维数组的返回地址。这是因为：一个多维数组在语义上并不等价于一个指向其元素类型的指针，相反它等价于一个“指向数组的指针”。举例如下：

```
char *p1 = new char[5][3];    // ERROR! 语义不等价
int  *p2 = new int[4][6];     // ERROR! 语义不等价
char (*p3)[4] = new char[5][4]; // OK! 退化第一维，语义等价
int  (*p4)[5] = new int[3][5];  // OK! 退化第一维，语义等价
char (*p5)[5][7] = new char[20][5][7]; // OK! 退化第一维，语义等价
.....
```

同样道理，像下面这样删除动态创建的多维数组的方法就是错误的：

```
delete [][]p3;    // 不存在这样的语法构造！
delete [][]p4;    // 不存在这样的语法构造！
delete [][][]p5;  // 不存在这样的语法构造！
```

我们必须从多维数组的语义出发来确定删除它们的正确方法，即把多维数组转化为等价的一维数组，然后就可以使用 `delete[]` 来删除。例如，删除 `p3`、`p4`、`p5` 的唯一具有可移植性、正确且标准的方法应该分别为：

```
delete []p3;    // 删除 p3
delete []p4;    // 删除 p4
delete []p5;    // 删除 p5
```

除此之外的其他删除方法其结果都是未定义的。

## 7.3 字符数组、字符指针和字符串

### 7.3.1 字符数组、字符串和 '\0' 的关系

许多人理不清字符数组和字符指针及字符串之间的关系，其实编译器把它们分得很清楚。

字符数组就是元素为字符变量的数组，而字符串则是以 '\0'（ASCII 码值为 0x00）为结束字符的字符数组。可见，字符数组并不一定就是字符串。

而对于字符串来说，它是可变长的，因此它无法记录自己的长度。但是，如何来表示字符串的结束呢？它本身又没有长度信息，因此必须用一个字符来标记字符串的结束，这就是 '\0' 的来历。由于字符串的连续性，编译器没有必要通过它的长度信息来提取整个字符串，仅通过一个指向其开头字符的字符指针就能实现对整个字符串的引用。

**【提示 7-13】** 如果用一个字符串字面常量来初始化一个字符数组，数组的长度至少要比字符串字面常量的长度大 1，因为还要保存结束符 '\0'。例如：

```
char array[] = "Hello";
```

数组 `array` 的元素为 {'H','e','l','l','o','\0'}。

**【提示 7-14】** 对于字符数组来说，它并不在乎中间或末尾有没有 '\0' 结束字符，因为数组知道它自己有多少个元素，况且 '\0' 字符对它来说是一个合法的元素。问题就在于：你可能会把字符数组当做字符串来使用，可能使用字符指针（例如 `char *p`）来引用一个字符数组。在这种情况下，用来操作字符串的库函数（例如 `strlen`、`strcpy` 等）并不知道这个字符串是来自一个字符数组，因为你传递进去的仅仅是一个字符指针，而字符指针除了它的类型和值外并不包含其他任何信息。这些库函数总是假定你提供的字符指针指向的内存空间中的某个字节里存放着那么一个 '\0'，它们会很傻地直到找到第一个 '\0' 字符时才会罢休。于是危险出现了：如果此时你的字符数组中并没有 '\0' 结束标志，那么把它当做字符串来用时就会导致“内存访问冲突”或者篡改了其他的内存单元。

**【提示 7-15】:** 如果你能够保证总是使用数组下标来访问字符数组中的每个元素，那么就没有必要在字符数组的结尾放进一个'\0'。但是我们都无法做出这样的保证，因为大多数情况下总是把字符数组用做字符串的缓冲区，所以应当在字符数组的结尾处放入这个麻烦的'\0'。

程序见示例 7-11。

示例 7-11

```
char arrChar_1[] = {'a','b','\0','d','e'};
char arrChar_2[] = "hello";
char *p = "hello";
cout << sizeof(arrChar_1) << endl;           // 5, 表示该数组占 5 字节
cout << strlen(arrChar_1) << endl;           // 2, 表示字符串长度为 2
cout << sizeof(arrChar_2) << endl;           // 6, 表示数组占 6 字节
cout << strlen(arrChar_2) << endl;           // 5, 表示字符串长度为 5
cout << sizeof(p) << endl;                   // 4, 表示指针 p 占 4 字节
cout << strlen(p) << endl;                   // 5, 表示字符串长度为 5
```

## 7.3.2 字符指针的误区

**【提示 7-16】:** 当你使用字符指针来引用一个字符变量的时候，千万要小心，因为 C++/C 默认 char \*表示字符串。例如：

```
char ch = 'a';           // 用字符'a'来初始化字符变量 ch
char *pChar = &ch;       // 字符指针指向字符变量
cout << pChar << endl;    // 错把字符指针当做字符串
```

正确的用法是：

```
cout << *pChar << endl;  // 取一个字符
```

## 7.3.3 字符串拷贝和比较

字符串的拷贝请使用库函数 strcpy 或 strncpy，不要企图用 “=” 对字符串进行拷贝，因为那是字符指针的赋值。

同理不要用 “==”、“>=”、“!=” 符号直接比较两个字符串，字符串的比较应该使用 strcmp、strncmp 等库函数。

**【提示 7-17】:** 对字符串进行拷贝时，要保证函数结束后目标字符串的结尾有'\0'结束标志。某些字符串函数并不会自动在目标字符串结尾追加'\0'，例如 strncpy 和 strncat，除非你指定的 n 值比源串的长度大 1，strcpy 和 strcat 会把源串的结束符一并拷贝到目标串中。

## 7.4 函数指针

在注册一个回调函数的时候，我们常常使用函数指针。C++/C 的连接器在连接程序的时候必须把函数体的首地址绑定到对该函数的调用语句上，因此函数地址必须在编译时就确定下来，也就是编译器为函数生成代码的时候。这样的话，函数地址就是一个编译时常量。函数指针就是指向函数体的指针，其值就是函数体的首地址。而在源代码层面，函数名就代表函数的首地址，所以可以把函数名直接指派给一个同类型的函数指针而不需要使用“&”运算符，也可以直接使用函数名来注册回调函数，见示例 7-12。

示例 7-12

```
typedef int __cdecl (* FuncPtr)( const char * ); // 定义一种函数指针类型
FuncPtr fp_1 = strlen ;
FuncPtr fp_2 = puts ;
double __cdecl (*fp_3)( double ) = sqrt ;
```

通过函数指针来调用函数有两种方式：

- (1) 直接把函数指针变量当做函数名，然后填入参数；
- (2) 将函数指针的反引用作为函数名，然后填入参数。

例如：

```
int len = fp_1("I am a software engineer.");
double d = ( *fp_3 )( 10.25 );
```

可以通过函数指针数组实现同类型函数的批量调用。在 C++ 动态决议的虚拟机制中使用的 `vtable` 就是一个用来保存虚成员函数地址的函数指针数组。函数指针数组的声明和初始化见示例 7-13。

示例 7-13

```
double __cdecl (* fp[5])( double ) = { sqrt, fabs, cos, sin, exp };
for ( int k = 0; k < 5; k++ )
{
    cout << "Result : " << fp[k]( 10.25 ) << endl ;
}
```

我们来简单分析一下函数指针数组是如何实现函数连接（绑定）的。我们知道，数组的一个元素是一个特殊的“变量”，而函数名就是函数的地址，是一个编译时的常量；我们又知道，“函数连接”的本质就是把函数地址绑定到函数的调用语句上（实际上就是一个跳转指令），并且一般的函数调用语句可以在编译时就完成这个绑定（叫做静态决议或静态连接）。然而由于数组下标 `k` 是一个变量，它在编译时是没有值的，因此 `fp[k]` 在编译时自然也就没有确定的值，你说它等于 `sqrt`、`fabs`、`cos`，还



是等于 `sin` 或 `exp`？都不是！这就是说，“`fp[k](10.25)`”这个调用在编译时无法绑定到一个具体确定的函数体上。这正是函数指针数组不同于一般函数调用的地方。于是，连接器只能把这个通过函数指针的调用推迟到运行时再绑定（叫做运行时连接）。在运行时的某一时刻，`k` 有了一个确定的值，例如 1，从而 `fp[k]` 也就确定了下来，例如 `fp[1]` 即 `sin`，于是这个调用就跳转到 `sin` 函数并执行起来；而在另一个时刻，`k` 具有了另一个值，`fp[k]` 就跳到了另一个函数体……

关于类成员函数的指针这个主题就比较复杂了，我们在这里简单地和函数指针比较一下。

类的成员函数有 4 种类型：`inline`、`virtual`、`static`、`normal`。`inline` 函数在运行时会展开，虽然语言允许取其地址，但是没有太大意义。`virtual` 成员函数的地址指的是其在 `vtable` 中的位置；`static` 成员函数的地址和普通全局函数的地址没有任何区别；普通成员函数的地址和一般函数的地址也没有区别，就是函数代码在内存中的真实地址，但是由于它的调用要绑定到一个实实在在的对象上，因此无论是其函数指针的声明方式还是其地址的获取方法都比较特别，见示例 7-14。

示例 7-14

```
class CTest {
public:
    void f( void ) { cout << "CTest::f()" << endl; }           // 普通成员函数
    static void g( void ) { cout << "CTest::g()" << endl; }    // 静态成员函数
    virtual void h( void ) { cout << "CTest::h()" << endl; }   // 虚拟成员函数
    //...
private:
    //...
};

void main()
{
    typedef void (*GFPtr)( void );           // 定义一个全局函数指针类型
    GFPtr fp = CTest::g;                     // 取静态成员函数地址的方法和
                                           // 取一个全局函数的地址相似
    fp();                                    // 通过函数指针调用类静态成员函数
    typedef void (CTest::*MemFuncPtr)( void ); // 声明类成员函数指针类型
    MemFuncPtr mfp_1 = &CTest::f;           // 声明成员函数指针变量并初始化
    MemFuncPtr mfp_2 = &CTest::h;           // 注意获取成员函数地址的方法
    CTest theObj;
    (theObj.*mfp_1)();                       // 使用对象和成员函数指针调用成员函数
    (theObj.*mfp_2)();
    CTest *pTest = &theObj;
    (pTest->*mfp_1)();                       // 使用对象指针和成员函数指针调用成员函数
    (pTest->*mfp_2)();
}

输出结果:
```

```
CTest::g()  
CTest::f()  
CTest::h()  
CTest::f()  
CTest::h()
```

类的静态成员函数不依赖于类的对象而存在，也不依赖于类的对象而调用，因此它和普通的全局函数没什么两样，只是其作用域变成了类作用域。我们可以像取全局函数的地址那样直接把类静态成员函数名指派给一个普通全局函数类型的指针，并且直接用这个指针就可以实现对该静态成员函数的调用。关于类的静态成员函数我们还会在后面详细讲述。

其他两种成员函数虽然也不依赖于类的对象而存在，但是其调用必须绑定到具体的对象。为了与静态成员函数区别，取 `virtual` 函数和普通成员函数的地址需要使用“&”运算符（取静态成员函数的地址也可以使用它，但是没有必要）。

为什么能够取得一个成员函数的地址呢？实际上，任何成员函数的代码体都是独立于类的对象而存在的，只是非静态成员函数在调用的时候需要与具体的对象建立绑定关系而已（即 `this` 指针）。C++/C 编译器最终会把所有的成员函数经过 `Name-Mangling` 的处理后转换成全局函数，并且增加一个入参 `this` 作为第一个参数，供所属类的所有对象共享。因此成员函数的地址实际上就是这些全局函数的地址，因而也是编译时常量。由于纯虚函数没有实现体，而非纯虚函数有实现体，且虚函数都是通过 `vptr` 和 `vtable` 来间接调用的，因此取虚函数的地址将得到该虚函数实现体在 `vtable` 中的索引号。要想得到虚函数实现体的真实地址，还需要首先从对象入手，找到 `vptr` 的位置，进而找到 `vtable` 的所在，然后根据函数指针的大小和虚函数的索引，取出虚函数的真实地址。

## 7.5 引用和指针的比较

引用“&”是 C++ 新增的概念，注意这里的“&”不是取地址的语义。以下程序中 `n` 是变量 `m` 的一个引用（reference），`m` 是被引用物（referent）：

```
int m;  
int& n = m;
```

`n` 相当于 `m` 的别名（绰号），对 `n` 的任何操作就是对 `m` 的操作。例如有人名叫王小毛，他的绰号是“三毛”。说“三毛”如何如何，其实就是对王小毛说三道四。所以 `n` 既不是 `m` 的拷贝，也不是指向 `m` 的指针，其实 `n` 就是 `m` 自己。

初学者容易把引用和指针混淆。引用的特性及其与指针的比较如下：

（1）引用在创建的同时必须初始化，即引用到一个有效的对象；而指针在定义的时候不必初始化，可以在定义后面的任何地方重新赋值；

（2）不存在 `NULL` 引用，引用必须与合法的存储单元关联；而指针则可以是

NULL。如果把一个引用初始化为 0，例如：

```
const int& rInt = 0;
```

其语义并非是把引用初始化为 NULL，而是创建一个临时的 int 对象并用 0 来初始化它，然后再用它来初始化引用 rInt，而该临时对象将一直保留到 rInt 销毁的时候才会销毁。所以，不要用字面常量来初始化引用：

(3) 引用一旦被初始化为指向一个对象，它就不能被改变为对另一个对象的引用（即“从一而终、矢志不渝”）；而指针在任何时候都可以改变为指向另一个对象。给引用赋值并不是改变它和原始对象的绑定关系，例如：

```
int a = 10, b = 1000;
int& rInt = a;          // rInt 引用到 a, rInt 等于 10
rInt = b;               // rInt 和 a 都变成了 1000
```

这并非是让 rInt 改变初衷而引用到变量 b，而是把 b 的值赋给变量 a，以后对 rInt 的使用仍然是在使用 a 而不是 b。相反，指针就大不一样了，例如：

```
int *pInt = &a;         // pInt 指向变量 a
pInt = &b;
```

现在 pInt 指向了变量 b，以后对 \*pInt 的使用就是在使用 b 而不是 a；

(4) 引用的创建和销毁并不会调用类的拷贝构造函数和析构函数；

(5) 在语言层面，引用的用法和对象一样；在二进制层面，引用一般都是通过指针来实现的，只不过编译器帮我们完成了转换。

这看起来有点像是在玩文字游戏，没有体现出引用的价值。引用的主要用途是修饰函数的形参和返回值。C++语言中，函数的参数和返回值的传递方式有三种：值传递、指针传递和引用传递。引用既具有指针的效率，又具有变量使用的方便性和直观性。

以下是“值传递”的示例程序。由于 Func1 函数体内的 x 是外部变量 n 的一个拷贝，改变 x 的值不会影响 n，所以 n 的值仍然是 0（见示例 7-15）。

示例 7-15

---

```
void Func1( int x )
{
    x = x + 10;          // 修改的是 n 在堆栈中的拷贝 x
}
...
int n = 0;
Func1( n );
cout << "n = " << n << endl; // n = 0
```

---

以下是“指针传递”的示例程序。由于 Func2 函数体内的 x 是指向外部变量 n 的指针，改变该指针指向的内存单元的内容将导致 n 的值改变，所以 n 的值为 10（见示例 7-16）。

示例 7-16

---

```
void Func2( int  *x )
{
    (* x) = (* x) + 10;           // 修改指针 x 指向的内存单元的值
}
...
int n = 0;
Func2( &n );
cout << "n = " << n << endl;    // n = 10
```

---

以下是“引用传递”的示例程序。由于 Func3 函数体内的 x 是外部变量 n 的引用，x 和 n 是同一个东西，改变 x 就是在改变 n，所以 n 的值为 10（见示例 7-17）。

示例 7-17

---

```
void Func3( int& x )
{
    x = x + 10;                   // 修改的是 x 引用到的对象 n
}
...
int n = 0;
Func3( n );
cout << "n = " << n << endl;    // n = 10
```

---

对比上述几个示例程序，你会发现：“引用传递”的性质像“指针传递”，而书写方式像“值传递”。实际上“引用”可以做的任何事情，“指针”都能够做，为什么还要“引用”呢？

答案是：“用适当的工具做恰如其分的工作”。“引用”体现了**最小特权原则**，即给予程序元素足以完成其功能的最小权限。

指针能够毫无约束地操作内存中的任何东西，尽管功能强大，但是非常危险。就像一把刀，它可以用来砍树、裁纸、修指甲、理发等，但谁敢这样用？

如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。



## 第 8 章 C++/C 高级数据类型

C 语言中的构造数据类型如结构、联合、枚举等在 C++ 中仍然有效。由于 C++ 新增了一种类型名 `class`，许多人错误地认为 `struct` 只能用来包装数据，或者 `class` 必须定义成员函数。

C++ 对 C 的结构、联合、枚举等进行了必要的改造和增强，本章比较分析了异同点，总结了使用要点，对于那些正在从 C 语言向 C++ 语言过渡的程序员有较好的参考价值。

### 8.1 结构 (Struct)

如果只能使用基本数据类型来编程，那将是一件痛苦的事情。C 语言支持把基本数据类型组合起来形成更大的构造数据类型，这就是 C 语言的 `struct`，有时也称为用户自定义数据类型 (User defined Type, UDT)。构造数据类型还可以嵌套 (对象嵌入) 和引用 (对象关联)，实际上，构造数据类型是一个递归的定义：

- (1) 由若干基本数据类型组合而成的类型是构造数据类型；
- (2) 由若干基本数据类型和构造数据类型组合而成的数据类型是构造数据类型；
- (3) 由若干构造数据类型组合而成的数据类型是构造数据类型。

语言本身的这种能力使我们能够定义非常复杂的数据结构，例如树 (tree)、链表 (list) 和映射 (map) 等。

#### 8.1.1 关键字 `struct` 与 `class` 的困惑

C++ 语言对 C 语言的 `struct` 进行了改造，使其也可以像 `class` 那样支持成员函数的声明和定义，从而使 `struct` 变成真正的抽象数据类型 (Abstract Data Type, ADT)，这使得许多人对 `struct` 和 `class` 倍感困惑。

当语言支持某种特征时，是否使用这种特征则完全取决于程序员。因此，并不是说 `class` 支持成员函数的定义，我们就一定要在每一个 `class` 中都定义成员函数；也并不是说 `struct` 过去不支持成员函数定义，我们就非得用 `class` 完全取代 `struct`。实际上就 C++ 语言本身来讲，`struct` 和 `class` 除了“默认的成员访问权限”这一点不同外，没有任何区别。

**【提示 8-1】** 在 C++ 语言中，如果不特别指明，`struct` 成员的默认访问限定符为 `public`，而 `class` 成员的默认访问限定符为 `private`。

因此，在 C++ 程序中，只要你明确地声明每一个成员的访问权限，那么完全可以用 `struct` 取代 `class`，也完全可以用 `class` 取代 `struct`，见示例 8-1。

示例 8-1

<pre>struct SA { public:     const char * GetName( ) const; private:     char *m_name;     int m_height;     int m_weight; };</pre>	<pre>class CA { public:     const char * GetName( ) const; private:     char *m_name;     int m_height;     int m_weight; };</pre>
---	--

本例中 SA 和 CA 这两个类型在 C++ 中没有任何不同。就像 Lippman 所说的那样，“在 C++ 中，选择使用关键字 `struct` 还是 `class` 来定义 UDT 或 ADT 完全是一种观念上的差异，而关键字本身并没有代表这种差异”。

我们再看一看 C++ 鼻祖 Bjarne Stroustrup 是如何说的：“带类的 C 和 C 语言几乎是‘代码兼容’的，并且也是连接兼容的。C 的函数可以在带类的 C 程序中调用，带类的 C 函数也可以在 C 程序中调用；带类的 C 程序中的 `struct` 和 C 中的 `struct` 在两个语言里的布局都一致，所以可以在两个语言之间传递简单对象或组合对象。这种连接兼容性一直保持到 C++ 中。”

C++ 仍然支持 C 风格的 `struct` 并且还做了增强，主要原因是为了兼容遗留的 C 代码以使它们可以在新的 C++ 环境下重新编译而继续“发挥余热”，可以让“过程化和结构化思想根深蒂固”的 C 程序员比较容易地过渡到面向对象的 C++ 语言。关于这个问题更具哲学性的讨论请参考 Lippman 所著《Inside The C++ Object Model》一书。

**【建议 8-1】：**为了不使程序产生混乱和妨碍理解，建议还使用 `struct` 定义简单的数据集合；而定义一些具有行为的 ADT 时最好采用 `class`，如果采用 `struct` 似乎感觉不到面向对象的味道了。

## 8.1.2 使用 struct

在 C++ 环境中，我们把 C 风格的 `struct` 叫做 POD (Plain Old Data) 对象，从字面上你也可以知道它仅包含一些数据成员，这些数据成员可以是基本数据类型变量、任何类型的指针或引用、任何类型的数组及其他构造类型的对象等，见示例 8-2。

示例 8-2

```
struct Student
{
    unsigned long ID;
    char firstName[15];
    char lastName[15];
    char email[20];
};
```



**【提示 8-2】:**

虽然把数组当做参数传递给函数的时候，数组将自动转换为指针，但是包装在 struct/class 中的数组其内存空间则完全属于该 struct/class 的对象所有。如果把 struct/class 当做参数传递给函数时，默认为值传递，其中的数组将全部拷贝到函数堆栈中。例如：

```
void func (Student s)
{
    cout << sizeof(s) << endl;    // 56
}
Student s0;
func (s0);
```

因此，当你的 UDT/ADT 中包含数组成员的时候，最好使用指针或引用传递该类型的对象，并且一定要防止数组元素越界，否则它会覆盖后面的结构成员。

任何 POD 对象的初始化都可以使用 memset() 函数或者其他类似的内存初始化函数。假设 s 是 Student 的一个对象，用 memset() 初始化 s 的方法如下：

```
memset (&s, 0x00, sizeof (Student));
```

C 风格的构造类型对象也可以在定义的时候指定初始值。我们可以仅指定第一个成员的初值来初始化 POD 对象，后面的成员将全部自动初始化为 0，就像数组的初始化一样。例如：

```
Student s = { 0 };
```

结构可以嵌套定义，也就是在一个结构的定义体内定义另一个结构，见示例 8-3。

示例 8-3

```
struct Student
{
    struct _Name
    {
        char firstName[15];
        char lastName[15];
    };
    unsigned long ID ;
    _Name name ;
    char email[20];
};
```

**【提示 8-3】:**

构造类型虽然可以嵌套定义，但是嵌套定义的类型其对象不一定存在包含关系，存在包含关系的对象类型也不一定是嵌套定义的。例如上例中的 \_Name 类型完全可以挪到 Student 定义的外面某处，而它们的对象之间的包含关系不会改变。当一个类型 A 只会在另一个类型 B 中被使用的时

候, 就可以把 A 定义在 B 的定义体内, 这样可以减少暴露在外面的用户自定义类型的个数。

所谓对象之间的包含是指一个类型的对象充当了另一个类型定义的数据成员, 从而也就充当了它的对象的成员, 即两个对象之间存在 **has-a** 关系。但是要注意: 一个对象不能自包含, 无论是直接的还是间接的, 因为编译器无法为它计算 `sizeof` 值, 也就不知道该给这样的对象分配多少存储空间, 见示例 8-4。

示例 8-4

<pre>struct A {     int i;     B b; };</pre>	<pre>struct B {     char ch;     A a; };</pre>
--	--

假设 A 定义在 B 的前面, 于是计算 A 的大小就需要知道 B 的大小, 而计算 B 的大小又需要 A 的大小, …… , 于是陷入了“鸡生蛋还是蛋生鸡”的怪圈! 这样的代码在编译的时候肯定通不过。

虽然对象不能自包含, 但可以自引用, 而且两个类型可以交叉引用, 这种关系称为 **holds-a** 关系。因为任何类型的指针的大小都一样, 给指针分配存储空间的时候不需要知道它指向的对象的类型细节, 见示例 8-5。

示例 8-5

<pre>struct A {     int count;     char *pName;    // A holds-a string     B *pb;          // A holds-a B };</pre>	<pre>struct B {     char ch;     A *pa;          // B holds-a A     B *pNext;       // B 自引用 };</pre>
--	---

上面的两个结构可以组成一个链表, A 是链表头的类型, B 是链表节点的类型。通过链表头节点可以遍历整个链表, 每个链表节点还可以指向另一个链表, …… , 这样就形成了一个庞大的链式结构。

利用对象之间的引用关系, 我们就可以实现链表、树、队列等复杂的数据结构, 或者实现一些复杂的对象管理, 比如对象之间的索引和定位。

**【提示 8-4】:** C++和 C 都支持相同类型对象之间的直接赋值操作 (默认的 `operator=` 语义, 就是对象按成员拷贝语义), 但是不能直接比较大小和判等。

这是因为, 相同类型对象的各数据成员在内存中的布局是一致的, 编译器执行默认的位拷贝也是符合赋值操作语义的。而出于对齐 (将大小调整到机器字的整数倍) 的考虑, 每个对象的存储空间中可能会存在填补字节, 这些字节单元不会初始化而是具有上次使用留下的“脏值” (随机值)。显然每个对象填补字节的内容是不会相同的。这就是说, 如果编译器支持使用逐位比较的默认方法来比较同类型对象,

结果肯定是不对的，而有意义的大小关系是与具体应用相关的，显然编译器并不对应应用领域的东西做任何假设。例如：

```
Student a, b;  
cout << ((a.ID > b.ID) ? "a larger than b" : "a less than b") << endl;
```

所以，当默认的赋值语义不能满足我们的要求的时候，就需要定义自己的赋值语义。在 C 语言中只有定义一些函数来完成这样的功能，而 C++ 则提供了运算符重载机制可以解决赋值和比较等问题（本质上仍然是函数调用，只是形式不同而已！）。

### 8.1.3 位域

显然在多数情况下，使用以字节为基本单位的存储模式会浪费大量的内存空间，我们在前面讲过的 `bool` 类型就是浪费存储空间的“大户”（浪费程度达 87.5%）。有些设备提供的存储空间很有限（例如嵌入式系统），因此必须对存储开销“精打细算”，这可以使用位域和位运算来解决。

位域以单个的位（bit）为单位来设计一个 `struct` 所需要的存储空间，因此你可以根据数据成员的有效取值范围来仔细规划它们各自所需要的位数，见示例 8-6。

示例 8-6

```
struct DateTime  
{  
    unsigned int year ;  
    unsigned int month : 4 ;  
    unsigned int day : 5 ;  
    unsigned int hour : 5 ;  
    unsigned int minute : 6 ;  
    unsigned int second : 6 ;  
};  
cout << sizeof (DateTime) << endl; // 8
```

C 语言位域各成员的类型必须是 `int`、`unsigned int`、`signed int` 等类型，C++ 还允许使用 `char`、`long` 等类型。不允许使用指针类型或浮点类型作为位域的成员类型，因为它们可能导致无效的值。关于 C 和 C++ 在位域上的不同请参考具体语言实现的相关文档。

`signed int` 类型数据的正负符号要占用一位，因此该类型的位域成员长度应该至少为 2。

**【提示 8-5】：** 不要定义超越类型最大位数的位域成员（这与平台相关）。例如：

```
struct DateTime  
{  
    unsigned int year : 33 ; // 位越界  
    //...  
};
```

如果允许这样做，不仅会导致结果上溢，还会导致一个成员跨越两个字

节的边界却又不能占满整个字节，这又是巨大的浪费。C++/C 语言干脆拒绝这样的定义。

可以定义非具名的位域成员，其作用是相当于占位符，可用来隔离两个相邻的位域成员。如示例 8-7，由于第二个位域成员没有名字，因此不能直接访问它所在的位。

示例 8-7

```
struct DateTime
{
    //...
    unsigned int day      : 5;
    unsigned int          : 2;
    unsigned int hour     : 5;
    //...
};
```

可以定义长度为 0 的位域成员，其作用是迫使下一个成员从下一个完整的机器字（Word）开始分配空间，见示例 8-8。

示例 8-8

```
struct DateTime
{
    //...
    unsigned int day      : 5;
    unsigned int          : 0;
    unsigned int hour     : 5;
    //...
};

cout << sizeof(DateTime) << endl; // 12
```

**【提示 8-6】:**

(1) 位域成员的访问方法和结构成员的访问没什么区别，要防止修改位域成员的值时可能出现的上溢；

(2) 不能取一个位域对象的数据成员的地址，即使该成员完全与字节边界对齐，因为字节是编址的最小单位而不是位；但是可以取位域对象的地址，即使位域所有成员的位数总和达不到整字节的倍数，位域对象也会对齐到机器字长；

(3) 不要把位域成员当做位的数组，因此不能使用访问数组元素的方法（[]）来访问位域成员的单个位。如果有这种需要，请使用位运算符（~、&、|、>>、<<、^及其与=的组合运算符）或者使用 std::bitset<N>。

**【建议 8-2】:**

在设计位域的时候，最好不要让一个位域成员跨越一个不完整的字节来存放，因为这样会增加计算机运算的开销。例如前面的 DateTime 的设计就不好。比较好的设计应该是：

```

struct DateTime
{
    unsigned int year ;
    unsigned int month      : 8 ;
    unsigned int day        : 8 ;
    unsigned int hour        : 8 ;
    unsigned int minute     : 8 ;
    unsigned int second     : 8 ;
};

```

请使用 `sizeof()` 运算符来计算位域的大小而不要自己估算，因为你很容易估算错误。

#### 【提示 8-7】:

使用位域节省存储空间会导致程序运行速度的下降，因为计算机无法直接寻址到单个字节中的某些位，必须通过额外的代码来实现。这种矛盾是由计算机的基本原理决定的，在“内存空间”和“运行速度”无法同时优化的情况下，由应用需求来决定优化哪一个。

### 8.1.4 成员对齐

我们在本书第 4.2 节讲过“自然对齐”的概念，本节将在此基础上讲解关于成员对齐的问题。结构的成员对齐方式是否一致，将影响到整个程序运行期间的稳定性甚至正确性，特别是模块之间的接口部分共用的结构数据类型。

试考虑下面的 `struct` 定义：

```

typedef unsigned char BYTE;
enum Color{ RED = 0x01, BLUE, GREEN, YELLOW, BLACK};
struct Sedan    // 私家车
{
    bool        m_hasSkylight;    // 是否有天窗
    Color       m_color;          // 颜色
    bool        m_isAutoShift;    // 是否是自动挡
    double      m_price;          // 价格（元）
    BYTE        m_seatNum;        // 座位数量
};

```

你能说出 `Sedan` 的对象在内存中实际占据的字节数是多少吗？或者更直接地说，这个结构的大小是多少呢？这个结构定义中的数据成员的声明顺序或者说这个结构的成员布局是不是合理的呢？为什么？如果不合理，怎样调整才能既不损失数据成员的访问效率，又能使对象的内存占用量最少呢？

许多人会说：把所有成员的大小加在一起不就行了（即 15 字节）吗？果真如此简单吗？更多的人会立刻说：用 `sizeof()` 运算符让编译器帮我们计算不就行了吗？的确，任何时候都应该用 `sizeof()` 运算符来计算一个类或者对象的大小，而不要自己猜测。但是如果仅仅想到这一步还不够！

既不损失 CPU 对对象的访问效率，又要尽可能地压缩对象的内存占用量，这一

直是我们追求的对象设计目标之一。在特定的运行环境下，一个应用程序可以使用的资源是有限的（比如内存），特别是在嵌入式应用程序的设计和开发中，在保证程序正确性的前提下，尽可能节省内存消耗同时提高程序的性能（提高性价比）无疑是最重要的考虑因素，而 C++/C 对象的大小则是影响这个性价比的一个重要因素（另一个因素当然是代码是否简单、精练和高效，有无重复代码）。尤其是当这些程序中需要定义大量的对象数组时，如果对象本身的设计浪费内存的话，这些对象数组必然浪费大量内存资源。

CPU 对对象的访问效率与什么有关系呢？是它们的地址的特点。

对于复合类型（一般指结构和类）的对象，如果它的起始地址能够满足其中要求最严格（或最高）的那个数据成员的自然对齐要求，那么它就是自然对齐的；如果那个数据成员又是一个复合类型的对象，则依次类推，直到最后都是基本类型的数据成员。

什么是“自然对齐要求最严格”呢？举例来说吧：`double` 变量的地址要能够被 8 整除，而 `int` 变量的地址只需能被 4 整除即可，一个 `bool` 变量的地址则只需能被 1 整除。所以 `double` 类型的自然对齐要求就要比 `int` 类型严格，`int` 类型的对齐要求又比 `bool` 类型严格，因为能够被 8 整除的地址肯定能被 4 整除，但是反过来就不一定了。在 C++/C 的基本数据类型中，如果不考虑 `enum` 可能的最大值所需的内存字节数，`double` 就是对齐要求最严格的类型了，其次是 `int` 和 `float`，然后是 `short`、`bool` 和 `char`。

例如，考虑 `Sedan` 的自然对齐要求。由于 `Sedan` 的所有成员都是基本类型，显然 `double` 成员 `m_price` 的对齐要求最严格，因此 `Sedan` 的对象的地址应该能被 8 整除。此外，如果编译器按照自然对齐的要求布局 `Sedan` 的内存映像的话，`m_price` 的偏移量还必须是 8 的倍数才能确保也总是自然对齐的，在这里应该是 16 字节；其他成员的起始地址也需要满足各自的自然对齐要求。

在复合类型的对象中，各个数据成员在内存中是如何排列的呢？一般说来，没有编译器会故意自找麻烦而把用户定义的数据成员声明顺序打乱来构造对象，都会直接依照声明顺序来存放，即使复合类型中存在多个访问段（即 C++ 类中的每个 `public`、`private` 和 `protected` 访问限定符），至少也会保证每个段内的所有数据成员是按照声明顺序来存放的。至于先声明的成员会被放在高地址还是低地址处，完全是由编译器实现来决定的，而且一般都会采用“按照声明的先后顺序从低地址到高地址依次布放各个成员”的方案。同时，为了满足各个成员的对齐要求，各个成员之间甚至对象的末尾可能会插入一定量的填充字节，因此对象的实际大小往往比把各个成员的大小简单加在一起要大。为什么有的对象会在末尾插入一定量的填充字节呢？因为编译器在考虑一个类型的大小的时候，不仅要考虑一个对象的对齐要求，还要考虑该类型对象数组的对齐要求，这样才能保证用户在使用对象数组时也具有和单个对象一样的访问效率。注意：绝对不会在对象开头插入填充字节。

基于上述认识，假设 `Sedan` 的对象 `s` 的起始地址为 `0x00031D10`，则 `s` 的内存布局将可能如图 8-1 所示（假设按照声明的先后顺序从低地址到高地址依次布放各个成员）。

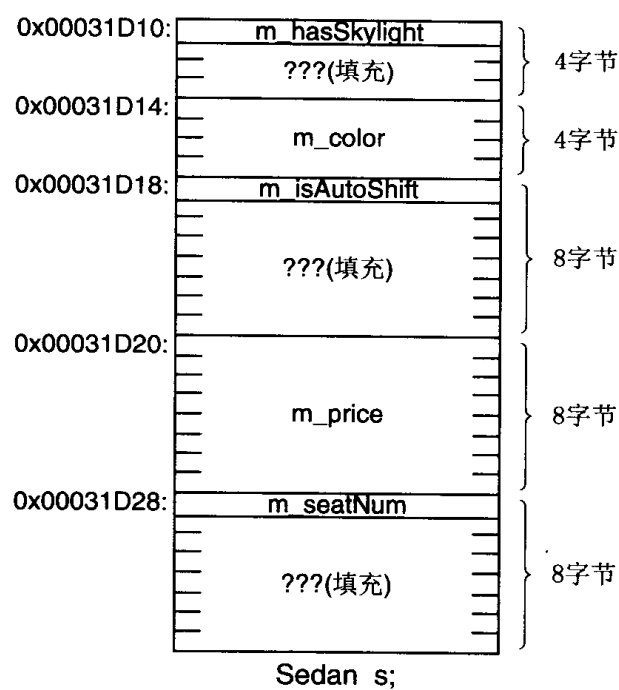


图 8-1 Sedan 的内存布局（8 字节边界对齐）

从图中可见，为了满足每一个成员的自然对齐要求，对象 s 的大小达到了 32 字节，其中填充字节竟然达到了 17 字节，比所有数据成员本身加在一起还大！bool 成员 m\_isAutoShift 竟然占据了 8 字节内存空间！那么为什么 m\_seatNum 后面还需要 7 个字节的填充呢？那是为了满足 Sedan 对象数组的对齐要求而必须加上的。试想，如果砍掉末尾这 7 个字节，对象 s 的大小就变为 25 字节，而数组的每个元素都是连续存放的，于是 Sedan 的数组中除第一个元素是自然对齐的外，后面的元素都有可能无法自然对齐，就会影响数组的访问效率。

下面定义了几个复合数据类型并列出了它们的自然对齐方式及其大小，如示例 8-9 所示。

示例 8-9

<pre>struct X {     char m_ch;     char *m_pStr; };</pre>	<pre>struct Y {     char m_ch;     int m_count; };</pre>	<pre>struct Z {     bool m_ok;     char m_name[6]; };</pre>
<pre>sizeof(X) = 4 按 4 字节对齐</pre>	<pre>sizeof(Y) = 8 按 4 字节对齐</pre>	<pre>sizeof(Z) = 7 按 1 字节对齐</pre>
<pre>struct R {     char m_ch;     double m_width;     char m_name[6]; };</pre>	<pre>struct T {     int m_no;     R m_r; };</pre>	<pre>struct U {     bool m_ok;     T m_t; };</pre>



sizeof(R) = 24 按 8 字节对齐	sizeof(T) = 32 按 8 字节对齐	sizeof(U) = 40 按 8 字节对齐
----------------------------	----------------------------	----------------------------

读者可以尝试画出上图中每一个复合类型的内存映像。

基于 Intel 系列 CPU 的 C++/C 编译器都会支持多种结构成员对齐方式，并允许用户选择其中任意一种对齐方式来创建复合类型对象，或者为不同的复合类型选择不同的对齐方式（但一种类型只能使用一种对齐方式，且一旦确定就不能再改变）。

复合类型对象在内存中创建后，每个成员本身的地址取决于它们相对于对象起始地址的偏移字节数，而这个偏移量并不仅仅与排在它前面的成员的大小有关，还与用户为这个对象类型指定的成员对齐方式有关。例如 MS C++/C 支持用户在代码中使用 `#pragma` 编译指令为某些复合类型定义显式指定结构成员对齐方式，或者使用编译器的命令行参数 `/Zp` 为个别文件中定义的复合类型或者整个工程中所有的复合类型指定对齐方式，可用的对齐方式有 1、2、4、8、16 字节地址边界对齐。具体用法可参考 MSDN 中关于 `struct member alignment` 的说明。而且它将按照声明的先后顺序从低地址到高地址依次布放各个成员。

## 【提示 8-8】:

在实际应用中，最好能够直接在代码中使用编译器支持的方法来为每一个复合数据类型指定对齐方式，或者为多个复合类型共同指定一个对齐方式。例如：

<pre> #ifdef _MSC_VER #pragma pack (push, 8) //按 8 字节边界对齐 #endif struct Sedan {     bool        m_hasSkylight;     Color       m_color;     bool        m_isAutoShift;     double      m_price;     BYTE        m_seatNum; }; #ifdef _MSC_VER #pragma pack (pop) #endif </pre>	<pre> #ifdef _MSC_VER #pragma pack (push, 4) //按 4 字节边界对齐 #endif struct Sedan {     bool        m_hasSkylight;     Color       m_color;     bool        m_isAutoShift;     double      m_price;     BYTE        m_seatNum; }; #ifdef _MSC_VER #pragma pack (pop) #endif </pre>
--	--

## 【提示 8-9】:

指定对齐方式的编译器指令可能是不可移植的。某些环境可能不支持用户为自定义复合数据类型特别指定对齐方式，编译器采用一种默认的、统一的对齐方式，特别是对于没有自然对齐要求的平台；或者指定成员对齐方式的指令有所不同。在具体开发时请查阅编译器的帮助文档。

显然，声明顺序和对齐方式一经确定，每个成员的地址偏移量就确定了，不随对象的不同而改变。因此，即使一个复合类型的对象在内存中满足了自然对齐的要求，其某些数据成员本身却可能不是自然对齐的。就拿 Sedan 来说吧，当按照 4 字节

地址边界对齐时，其大小将变为 24 字节，而成员 `m_price` 的偏移量将变为 12 字节，如图 8-2 所示。

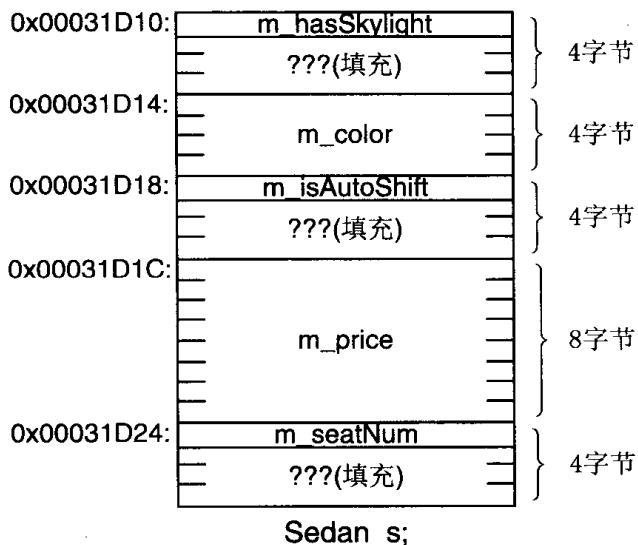


图 8-2 Sedan 的内存布局（4 字节边界对齐）

可见，即使 `s` 的起始地址仍然能够被 8 整除，`m_price` 的地址也不一定能被 8 整除（此处 `0x00031D1C` 就不能被 8 整除）。

**【提示 8-10】:**

想知道任意复合类型中每个数据成员的偏移字节数，有两种方法：

（1）使用 `offsetof` 宏。这个宏专门用来计算数据成员相对于对象起始地址的真实偏移量，它会把所有隐含成员也计算进去，比如虚函数表指针 `vptr`。其用法可参考 MSDN 中关于 `offsetof` 的说明及其定义。其他的编译器可能也定义了类似的工具来计算成员偏移，虽然其实现方法可能不同。如果你想追踪某个 C++ 编译器对虚拟机制的实现，使用这个工具很方便地就能推算出 `vptr` 的位置，读者不妨一试。例如：

```
std::cout << "offsetof(Sedan, m_hasSkylight) = " << offsetof(Sedan, m_hasSkylight);
std::cout << "offsetof(Sedan, m_color) = " << offsetof(Sedan, m_color);
std::cout << "offsetof(Sedan, m_isAutoShift) = " << offsetof(Sedan, m_isAutoShift);
std::cout << "offsetof(Sedan, m_price) = " << offsetof(Sedan, m_price);
std::cout << "offsetof(Sedan, m_seatNum) = " << offsetof(Sedan, m_seatNum);
```

（2）随便定义一个对象，依次打印出对象的起始地址及其每一个成员的地址，据此就可计算出每一个成员的偏移，或者直接将两个地址相减。例如：

```
Sedan s;
std::cout << "Address of s = " << (void*)&s;
std::cout << "offset of m_hasSkylight = " << ((char*)&s.m_hasSkylight - (char*)&s);
std::cout << "offset of m_color = " << ((char*)&s.m_color - (char*)&s);
std::cout << "offset of m_isAutoShift = " << ((char*)&s.m_isAutoShift - (char*)&s);
std::cout << "offset of m_price = " << ((char*)&s.m_price - (char*)&s);
std::cout << "offset of m_seatNum = " << ((char*)&s.m_seatNum - (char*)&s);
```

## **【注意】:**

千万不可使用“类成员的地址”这种方法来计算数据成员的实际偏移，因为这种方法计算出来的偏移并没有把隐含成员（例如 `vptr`）考虑在内，所以仅是源代码层面的逻辑偏移。而且用这种方法计算第一个用户声明的数据成员的字节偏移的话，将得到 1，而不是我们认为的 0，这是为了区分“一个没有指向任何数据成员的类成员指针变量的值”和“一个指向了第一个数据成员的类成员指针变量的值”这两个概念而故意如此。例如（假设按 4 字节地址边界对齐）：

```
bool Sedan::*pmSkylight = 0;           // 不指向任何成员
pmSkylight = &Sedan::m_hasSkylight;
```

// 值为 1，指向第一个成员（实际偏移为 0）

```
Color Sedan::*pmColor = &Sedan::m_color;
// 值为 4，指向第二个成员（实际偏移为 4）
```

```
s.*pmColor = BLACK;                   // s.m_color = BLACK;
```

```
bool Sedan::*pmAuto = &Sedan::m_isAutoShift; // 值为 8，指向第三个成员
```

```
double Sedan::*pmPrice = &Sedan::m_price;    // 值为 12，指向第四个成员
```

```
BYTE Sedan::*pmSeat = &Sedan::m_seatNum;     // 值为 20，指向第五个成员
```

但是，在 MS VC++ 6.0 上测试该方法，却得到所有成员的字节偏移都是 1 的结果，也许是该编译器做了特殊处理的缘故。

从上述分析可见，编译器不会随便地在任意一个逻辑内存地址上来创建 C++/C 的变量和对象，它们在内存中的起始地址需要满足一定的条件，数据成员也并非一定是挨在一起的，而且每个数据成员的地址也不是随便安排的，都需要经过编译器的精心规划和计算，这样才能提高对象及其成员的访问效率。

但是从 Sedan 的内存映像看，如果我们完全满足它的自然对齐要求，同时也满足其每一个成员的自然对齐要求，那将非常浪费内存！那么怎样才能节省内存呢？显然要设法减少对象中的空洞，宁愿让末尾留下空洞也不要让中间留下空洞，尽量使所有成员连续存放，并且减少末尾的填充字节。其实方法也很简单，那就是：按照从大到小的顺序从前到后依次声明每一个数据成员，并且尽量使用较小的成员对齐方式。按此法我们修改 Sedan 的定义如下所示，内存布局如图 8-3 所示（在 MS VC++6.0 上测试通过）。

```
#ifdef _MSC_VER
#pragma pack(push, 8) //按 8 字节边界对齐
#endif
struct Sedan
{
    double    m_price;
    Color     m_color;
    bool      m_hasSkylight;
    bool      m_isAutoShift;
    BYTE      m_seatNum;
};
```

```

#ifdef _MSC_VER
#pragma pack (pop)
#endif

```

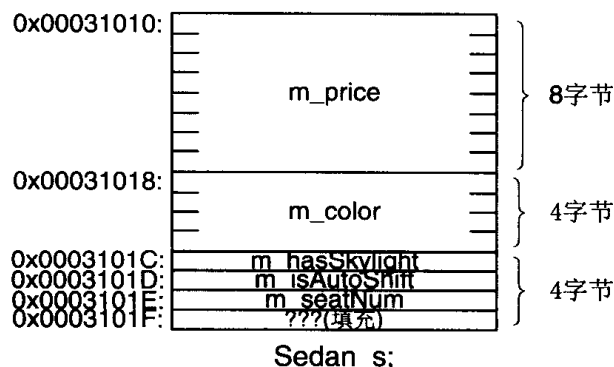


图 8-3 Sedan 的内存布局（调整成员顺序）

可以看出，Sedan 的大小立刻就变为 16 字节了，且末尾只有 1 字节的填充。如果继续减小对齐字节数到 4 或 2，其大小和布局将仍然保持与上图相同，直到对齐字节数减小到 1 时，才会把末尾的填充字节去掉，Sedan 的大小才会变为 15 字节，但是所有成员的偏移却不会改变。

综上所述，类的数据成员类型的选择、声明顺序即排列、采用的成员对齐方式都将影响对象的实际大小和访问效率。

虽然编译器已经帮我们处理了这些发生在背后的事情，但是程序员了解一些关于对齐的细节还是很有好处的：

- (1) 可以对对象的大小及其布局心中有数；
- (2) 可以帮助你优化复合类型的数据成员及其布局，吸收末尾的填充字节；
- (3) 至少还可以应付 IT 公司的那些计算对象大小的面试题☺！

如何吸收掉末尾的填充字节呢？以 Sedan 为例，把 m\_seatNum 的类型修改为 short 如何？因为除非你使用按 1 字节边界对齐方式，否则在要求自然对齐的平台上，这个末尾的 1 字节填充是无法消除的。与其留着不用，不如用 short 把它吸收掉，然后把顺序再调整一下。使用 short 不仅吸收了多余字节，而且扩大了 m\_seatNum 的值的范围，更不容易溢出。现在的布局如图 8-4 所示。

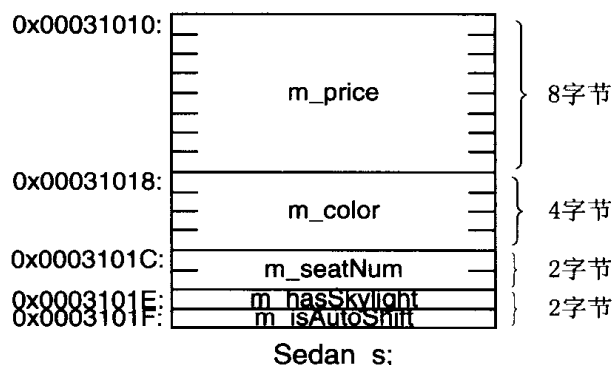


图 8-4 Sedan 的内存布局（吸收填充字节）

再比如下面的类定义：

```
class String
{
    char          *m_buf;
    unsigned short m_length;
public:
    size_t get_length() const { return m_length; }
    .....
};
```

如果按照自然对齐的要求为 String 对象分配内存，将占用 8 字节空间，末尾有 2 字节的填充，所以使用 short 并不能节省空间。如果想节省空间，就指定其对齐方式为 2 字节边界对齐，否则与其浪费 2 字节，不如把成员 m\_length 修改为 int 类型，从而吸收末尾的填充字节，这样不仅充分利用了内存，提高了效率，而且还扩大了 m\_length 的取值范围，如图 8-5 所示。

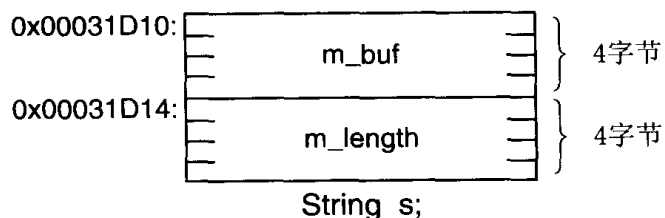


图 8-5 String 的内存布局

可见，数据成员也并非越小越好，关键是看其具体的需求。

那么为什么要“从大到小”排列复合类型的数据成员而不是“从小到大”排列呢？

从上面的几个例子可以看出，由于编译器绝对不会在对象的头部插入填充字节，因此当从小到大排列数据成员时，就可能会在对象的中间留下空洞。例如：如果 Sedan 按照从小到大的顺序排列数据成员，则在按照 8、4 或 2 字节边界对齐的情况下，其内存布局都将如图 8-6 所示。

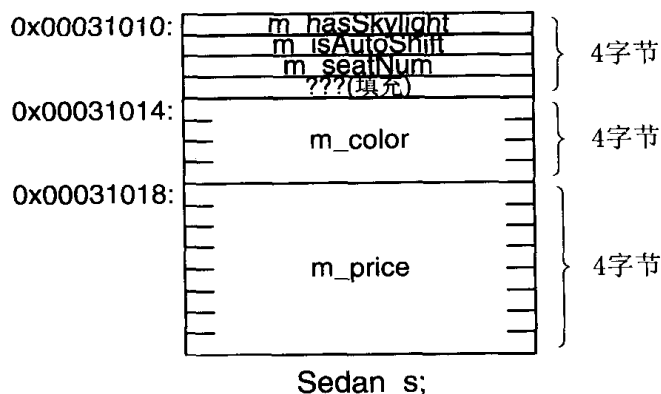


图 8-6 Sedan 的内存布局（从小到大排列）

一旦在对象的中间留下空洞，这些空洞就很可能随着对复合类型对齐方式的调整而被压缩甚至吸收掉，从而导致部分数据成员的偏移发生改变（成员前移）。例如：如果 Sedan 按照从小到大的顺序排列数据成员，并且现在按照 1 字节边界对齐，则其内存布局将如图 8-7 所示。

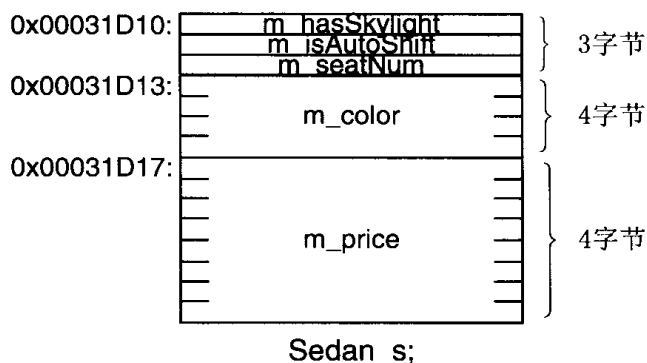


图 8-7 Sedan 的内存布局（1 字节边界对齐）

从上图可见，m\_color 和 m\_price 的相对位置都比原来前移了 1 字节，不再是自然对齐的了。

从上面的分析可知，如果所有数据成员都能从大到小排列，则只会在对象的末尾追加填充字节。这是因为，对齐要求最严格的成员被放在了对象的开头，其地址就是对象的首地址，如果满足了整个对象的对齐要求（实际上也就是满足了这个打头成员的对齐要求），那么排在后面的所有成员的对齐要求都自然满足，只要挨着依次存放即可，绝对不会在中间留下空洞。因此，对象的布局及各个成员的偏移就不会随着对齐方式的改变而改变，即其布局非常稳定。这实际上在一定程度上增强了复合类型的可移植性及其对象的二进制兼容性。这一点在那些包含多个模块的应用中定义模块之间的接口数据类型时是很重要的。

Sedan 足以说明问题，但是在 NotAlign 上，对象大小及成员偏移受对齐方式的影响却表现得更加明显。图 8-8 列出了分别在 8、4、2、1 字节对齐方式下 NotAlign 的布局。NotAlign 的定义如下：

```
struct NotAlign
{
    bool    m_b;
    double  m_d;
};
```

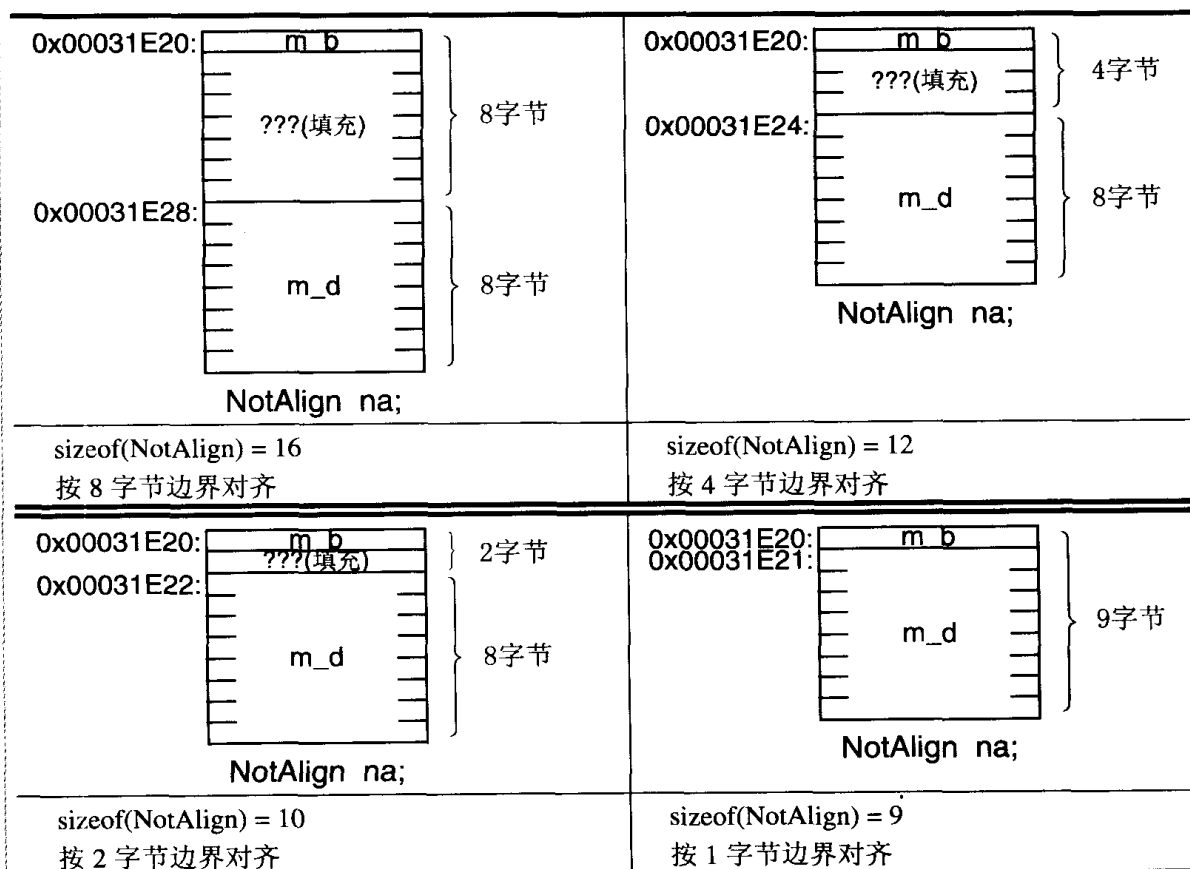


图 8-8 NotAlign 的内存布局

对齐方式的指定还关系到模块之间接口的语义一致性和对象的二进制兼容性，不一致的对齐方式极有可能导致程序运行时产生错误的结果甚至崩溃！

在一个多模块的应用程序中，可能包含一个可执行程序、若干个静态链接库和若干个动态链接库。不管是静态库还是动态库，一般来说，模块之间除了函数接口外，还会有一些共享的复合数据类型定义，它们也是模块接口的一部分。从上面的分析可知，使用不同的对齐方式，这些接口数据类型的对象在内存中很可能具有不同的布局，某些成员的偏移会发生变化。如果不同模块恰好使用了不同的对齐方式，而模块间共享的复合数据类型没有显式地指定对齐方式，那么程序出错甚至崩溃的风险就会增加。为什么呢？下面举个例子来说明这个问题。

假设有个应用程序由一个主程序（main.exe）和一个静态链接库（PrintSedan.lib）连接而成，这两个模块之间共享一个用户自定义复合数据类型 Sedan 和一个函数 print\_Sedan\_1()。两个模块共享的接口头文件定义如下：

```
// sedan.h
//
#ifndef _SEDAN_H_
#define _SEDAN_H_

enum Color{ RED = 0x01, BLUE, GREEN, YELLOW, BLACK};
typedef unsigned char  BYTE;
```



```
struct Sedan //私家车
{
    bool        m_hasSkylight; // 是否有天窗
    Color        m_color;      // 颜色
    bool        m_isAutoShift; // 是否是自动档
    double       m_price;       // 价格 (元)
    BYTE        m_seatNum;      // 座位数量
}; // 注意: 这里没有指定对齐方式!

#ifdef __cplusplus
extern "C" {
#endif
void __cdecl print_Sedan_1(const Sedan *p);
#ifdef __cplusplus
}
#endif

#endif // _SEDAN_H_
```

---

PrintSedan.lib 的实现如下:

---

```
// sedan.cpp
//
#include "sedan.h"
#include <iostream>
using namespace std;

#ifdef __cplusplus
extern "C" {
#endif
void __cdecl print_Sedan_1(const Sedan *p)
{
    cout << "print_Sedan_1() in library PrintSedan.lib : " << endl;
    cout << "// Sedan : \n{\n";
    cout << "\t m_hasSkylight = " << (p->m_hasSkylight ? "yes" : "no") << endl;
    cout << "\t m_color = " << p->m_color << endl;
    cout << "\t m_isAutoShift = " << (p->m_isAutoShift ? "yes" : "no") << endl;
    cout << "\t m_price = " << p->m_price << endl;
    cout << "\t m_seatNum = " << (int)(p->m_seatNum) << endl;
    cout << "}\n\n";
}
#ifdef __cplusplus
}
#endif
```

---

我们设置 sedan.cpp 的成员对齐方式为 8 字节边界对齐, 然后编译生成 PrintSedan.lib。

而 main.exe 也实现了一个 print\_Sedan\_2():

---

```
// main.cpp
//
#include "sedan.h"          /* 包含库头文件 */
#include <iostream>
using namespace std;

void __cdecl print_Sedan_2(const Sedan *p)
{
    cout << "print_Sedan_2() in main.cpp : " << endl;
    cout << "// Sedan : \n{\n";
    cout << "\t m_hasSkylight = " << (p->m_hasSkylight ? "yes" : "no") << endl;
    cout << "\t m_color = " << p->m_color << endl;
    cout << "\t m_isAutoShift = " << (p->m_isAutoShift ? "yes" : "no") << endl;
    cout << "\t m_price = " << p->m_price << endl;
    cout << "\t m_seatNum = " << (int)(p->m_seatNum) << endl;
    cout << "}\n\n";
}

int main()
{
    Sedan s;
    s.m_hasSkylight = true;
    s.m_color = GREEN;
    s.m_isAutoShift = false;
    s.m_price = 100000;
    s.m_seatNum = 4;
    print_Sedan_2(&s);
    print_Sedan_1(&s);
    return 0;
}
```

---

我们设置 main.cpp 的成员对齐方式为 4 字节边界对齐, 然后编译并与 PrintSedan.lib 连接, 生成 main.exe。运行时输出结果如下:

---

```
print_Sedan_2() in main.cpp :
// Sedan :
{
    m_hasSkylight = yes
    m_color = 3
    m_isAutoShift = no
    m_price = 100000
    m_seatNum = 4
}

print_Sedan_1() in library PrintSedan.lib :
```

```
// Sedan :  
{  
    m_hasSkylight = yes  
    m_color = 3  
    m_isAutoShift = no  
    m_price = 3.18272e-023    // !!!  
    m_seatNum = 76           // !!!  
}
```

对比这两个结果：同样一个对象，在传递给 `PrintSedan.lib` 后，`m_price` 和 `m_seatNum` 的值就不对了。什么原因呢？就是因为两个模块使用了不一致的对齐方式，导致它们对 `Sedan` 的解释不一致，主要就是理解的对象大小不一致，某些成员的偏移不一致，`main.exe` 认为 `Sedan` 的大小为 24 字节，并且给 `PrintSedan.lib` 传递了一个 24 字节大小的对象，而 `PrintSedan.lib` 认为 `Sedan` 的大小为 32 字节，因此把传递进来的对象当做 32 字节来操作，当然会导致访问错误的内存和输出错误的结果。这就是二进制不兼容问题。如果这样的代码在 `PrintSedan.lib` 中被用于计算而不是简单的输出，那后果可想而知！

如果能够完全按照从大到小排列每一个数据成员，以及每一个嵌套的复合数据成员，虽然可以大大降低这种不一致的风险，但是定义一致的对齐方式将可从根本上解决这个问题。能够 100% 保证一致的方法就是直接在代码中使用编译器提供的方法指定每一个接口数据类型的对齐方式，而不是依赖于命令行参数设置或者其他途径。

COM 规范要求不得在接口中定义数据成员，其原因之一就是这可能导致不同模块使用不同编译器而出现二进制不兼容。

## 8.2 联合（Union）

联合也是一种构造数据类型，它提供了一种使不同类型数据成员之间共享存储空间的方法，同时可以实现不同类型数据成员之间的自动类型转换。但是与结构不同的是，联合对象在同一时间只能存储一个成员的值（即只有一个数据是活跃的）。因此，如果你同时访问一个联合对象的多个成员，那么其中最多只有一个值是正确的，见示例 8-10。

示例 8-10

```
union KeyCode  
{  
    short    keyNum ;  
    char     byteArr[2] ;  
};
```

**【提示 8-11】:**

联合的内存大小取决于其中字节数最多的成员，而不是累加，联合也会进行字长对齐。与位域不同的是，使用联合不会带来任何额外的运行时开销。联合中存储的数据的值完全取决于对它的解释方式。用一个成员存入数据而用另一个成员来提取它是可以的，但结果可能并非你想要的。在定义联合变量的时候可以指定初始值，但是只能指定一个初始值，而且该初始值的类型必须与联合的第一个成员的类型匹配。可以取一个联合变量的地址，也可以取一个联合变量的任一成员的类型地址，它们总是相等的。你还可以在同类型的联合变量之间赋值。但是不能比较两个联合变量的大小，不只是因为可能存在填补字节，而且这两个变量可能存储着不同类型的成员，此时它们实际上代表两个类型不同的变量。

C++对C的union进行了扩展，除了数据成员外还可以定义成员的访问说明符，可以定义成员函数，甚至还可以定义构造函数和析构函数。但是联合不能包含虚拟成员函数和静态数据成员，不能作为其他类型的基类或者派生自其他类型。此外，C++还支持匿名联合。

比如定义一个用于标识社会中的“人”的类型，并且假设不会同时使用身份证号码和姓名，于是就可以像示例 8-11 这样来定义。

示例 8-11

```
#include <cassert>
#include <iostream>
#include <iomanip>
using namespace std;
class Person {
public:
    Person(): type(true){ id.ID = 0; }
    void SetID(unsigned long double newId) {
        id.ID = newId;
        type = true;
    }
    unsigned long double GetID() const {
        assert( type != false);
        return id.ID;
    }
    void SetName(const char *name) {
        strcpy(id.name, name);
        type = false;
    }
    const char* GetName() const {
        assert( type == false );
        return id.name;
    }
    bool IDIsSelected() const { return type; }
```

```
bool NameIsSelected() const { return (! Type); }
private :
    union Choice
    {
        unsigned long ID;           // 身份证号码
        char name[20];              // 姓名
    };
    Choice id;                      // ID/name
    bool type;                      // 标识当前存放的是 ID(true)还是 name(false)
};
int main(int argc, char* argv[])
{
    Person p1;
    p1.SetID( 123456 );
    cout << p1.GetID( ) << endl;
    p1.SetName( "HYQ" );
    cout << p1.GetName( ) << endl;
    if( p1.NameIsSelected( ) )
        cout << "Now, id is type of name." << endl;
    else
        cout << "Now, id is type of ID." << endl;
    return 0;
}
输出:
123456
HYQ
Now, id is type of name.
```

联合的另一个妙用就是用来解析一个寄存器或多字节内存变量的高低字节的值，而不需要我们手工使用位运算符来解析它们。比如上面定义的联合类型 `KeyCode` 就可用来自动获取按键编码的高低字节，因为它们分别代表不同的击键含义。当你用键盘敲入一个字符的时候，计算机内部把它转换为一个双字节的整数编码，其中 `byteArr[1]` 存放的是高字节的值，而 `byteArr[0]` 存放的就是低字节的值。一般的 ASCII 码就保存在低字节中，而键盘扩展码保存在高字节中。

关于联合作为类型自动转换的工具，可参考 MFC 应用框架中的消息映射表的设计。

## 8.3 枚举 (Enum)

C++/C 枚举类型允许我们定义特定用途的一组符号常量，它表明这种类型的变量可以取值的范围。当你定义一个枚举类型的时候，如果不特别指定其中标识符的值，则第一个标识符的值将为 0，后面的标识符将比前面的标识符依次大 1；如果你指定了其中某一个标识符的值，那么它后面的标识符自动在前面的标识符值的基础上依次加 1，除非你也同时指定了它们的值。例如：

```
enum Week { Sun, Mon = 125, Tue, Wed, Thu = 140, Fri, Sat };
```

枚举类型 `Week` 中各符号常量的值依次为: 0, 125, 126, 127, 140, 141, 142, 这些值就是 `Week` 的变量可以具有的合法的值。例如:

```
Week weekday = Sun;
```

你也可以把某些枚举常量初始化为相等的常量, 这可能在某些具体的应用中有用。例如:

```
enum ABC { A = 1, B = 1, C = 10 };
```

在标准 C 中, 枚举类型的内存大小等于 `sizeof(int)`。但是在标准 C++ 中, 枚举类型的底层表示并非必须是一个 `int`——它可以更小或更大。换句话说, 如果一个枚举变量的取值范围小到足以用一个 `short` 或 `byte` 来表示, 那么这个枚举变量的底层表示就可能采用 `short` 或 `byte`; 相反如果一个枚举变量的取值范围大到必须用一个比 `int` 更大的类型来表示的话, 那编译器允许使用更大的类型来表示枚举变量。例如:

```
enum { SIZE = 123456789012 };
```

这在标准 C++ 中是允许的 (不过要看具体的语言实现是否支持定义这么大的枚举常量)。

**【提示 8-12】:** 枚举变量和常量都可以参与整型变量能够参与的某些运算, 但注意不要给枚举变量赋予一个不在枚举常量列表中的值, 例如, 不要对枚举变量使用 `++`、`--`、`+=`、`-=` 等操作, 除非你为它特别重载了这些运算符。

虽然枚举是和整数类型兼容的数据类型, 但是它们之间的转换还是有一些需要注意的地方。枚举类型变量一般可以直接转换成某种整数类型, 除非其值超出了这种整数类型可以表示的范围。但是一个整型变量在强制转换成枚举类型后就不一定具有一个有效的值了, 这是因为: 整型数是连续的, 而枚举类型变量的取值很可能是不连续的, 因此当你把一个值不等于任何一个枚举常量的整型变量强制转换成这种枚举类型时, 其结果就不得而知了。

此外, 枚举类型还可以是匿名的。匿名的枚举类型就相当于直接定义的 `const` 符号常量, 可以作为全局枚举, 也可以放在任何类定义或名字空间中。

**【建议 8-2】:** 使用匿名枚举来定义程序中出现的相关常量集合是一个好主意, 它可以取代宏常量和 `const` 符号常量。例如:

```
enum
{
    OBJECT_CREATION      = 0x10,
    OBJECT_DELETION      = 0x11,
    STATE_CHANGE         = 0x12,
    ATTR_VALUE_CHANGE    = 0x13,
    NEW_ALARM            = 0x20,
```

```
ALARM_CLEARED      = 0x21,  
ALARM_CHANGED      = 0x22  
};
```

## 8.4 文件

文件操作属于一种 I/O 操作，I/O 操作并不是 C++/C 语言的组成部分，它是通过标准的 I/O 函数库来实现的。操作系统甚至把设备也作为文件来看待，例如键盘是字符设备文件，磁盘是块设备文件等。本节仅简单讨论常用的磁盘文件的操作方法，需要对文件系统有全面了解的读者请参考其他的资料。

内存中的任何对象都可以看做是由一些字节序列组成的实体。对计算机来说，它并不知道某个对象在实际应用中代表的含义，在它看来任何对象除了内存空间和值以外没有任何区别。

当你把内存中的对象（无论是 ASCII 字符串还是二进制对象）写入磁盘文件的时候，计算机只是把它们内存映像（值）拷贝到磁盘文件中；反过来，当你从磁盘文件读取数据（无论是 ASCII 字符串还是一些二进制数据）到内存中的时候，所读数据的含义完全取决于你如何解释它（即把它赋值给一个什么类型的内存对象）。因此，我们完全可以把一个 ASCII 文件当做二进制文件来读取，或者把一个二进制文件当做 ASCII 文件来读取，计算机并不在乎这样做有什么危险，相反，危险在你的应用领域：因为使用时这些数据是错误的！

上面的分析表明：任何对象和文件只有与具体的应用相关联才有含义，否则就是一些单纯的字节序列。也就是说，并不存在一种固定的“文件记录”，任何有意义的文件格式都是由具体的应用领域决定的。

在 C++/C 中，文件操作是通过和“流”这种对象关联而进行的。流的意思就是字节流。当我们打开一个文件的时候，操作系统就建立一个流对象并与该文件关联。操作系统维护了一个保存当前系统中所有打开文件的文件控制块（FCB）的数组，并利用每一个 FCB 来管理对每一个文件的操作，数组的上限就是操作系统允许你同时打开的文件个数的上限。在 C 语言中，一个 FILE 结构中包含了打开文件的描述符（即文件句柄，是一个整数），它就是用来检索这个文件控制块数组的下标。我们用图 8-9 来表示它们之间的关系（假设这是打开的第 10 个文件）。

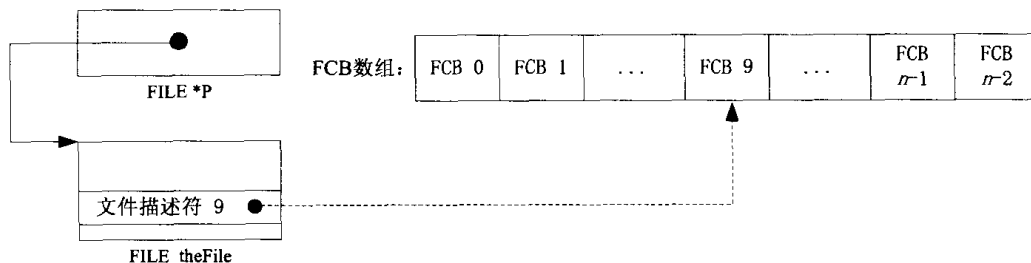


图 8-9 文件系统：文件控制块与 FILE 结构的关系



我们要做的第一件事情是声明一个 FILE 结构的指针，然后调用库函数 `fopen()`。`fopen()` 动态创建一个 FILE 结构对象并分配一个文件句柄，从磁盘文件中读入 FCB 结构并填入 FCB 数组中，然后返回这个 FILE 结构的地址。此后你就可以拿着这个地址调用文件操作库函数来完成特定的任务。最后调用 `fclose()` 函数销毁动态创建的 FILE 结构对象，同时释放文件句柄并刷新缓冲区等，因为其他应用程序有可能也要使用这些资源。

C++ 实现了面向对象的 I/O 系统，不再让用户直接使用“文件指针”这种原始的设施，而是把任何文件看做一个对象，对该对象的操作就是在对一个文件进行操作，同时允许用户为自定义数据类型定制 I/O 操作，这正是 C++ 扩展性的表现。

**【提示 8-13】:**

虽然在 C++ 程序中可以混用 C 的 I/O 操作方式和 C++ 的 I/O 操作方式，但还是应该尽量使用后者，因为它是类型安全的。如果使用 C 的 I/O 操作方式，即使你的格式控制字符串与输出数据类型完全不匹配，编译器也不会帮你检查出来（因为它们是字符串常量），因此它是类型不安全的。

由于 I/O 是一个比较复杂的系统，涉及到大量的格式控制方法，因此不便在此详述，具体使用方法请参考其他有关资料。

## 第9章 C++/C 编译预处理

C++/C 的编译预处理器对预编译伪指令进行处理后生成中间文件作为编译器的输入, 因此所有的预编译伪指令都不会进入编译阶段。预编译伪指令一般都以 # 打头, 且其前面只能出现空白字符。预编译伪指令不是 C++/C 语句, 但是它们可以出现在程序中的任何地方, 只要展开后符合语法规则并且是有效的, 例如头文件中、函数定义体内、控制结构中、类定义和名字空间定义中。本章讲述 C++/C 程序中经常用到的一些预编译伪指令, 如文件包含、宏定义、条件编译及一些不常使用的预编译伪指令和符号常量, 使用它们对提高代码的可移植性和清晰性很有好处。

### 9.1 文件包含

#include 伪指令用于包含一个头文件, 头文件中存放的一般是模块接口, 编译预处理器在扫描到该伪指令后就用对应的文本内容替换它。文件包含有两种语法形式:

- (1) #include <头文件名称>
- (2) #include "头文件名称"

第一种形式一般用来包含开发环境提供的库头文件, 它指示编译预处理器在开发环境设定的搜索路径中查找所需的头文件。第二种形式一般用来包含自己编写的头文件, 它指示编译预处理器首先在当前工作目录下搜索头文件, 如果找不到的话再到开发环境设定的路径中去查找。

使用该伪指令时, 头文件前面可以加相对路径或者绝对路径 (此处的 “\” 并不解释为转义字符)。例如:

```
#include ".\myinclude\abc.h"
#include "C:\myproject\test1\source\include\abc.inl"
```

#### 9.1.1 内部包含卫哨和外部包含卫哨

为了避免同一个编译单元包含同一个头文件的内容超过一次 (这会导致类型重复定义错), 我们需要在头文件里面使用**内部包含卫哨**。内部包含卫哨实际上是使用预处理器的一种标志宏。有了内部包含卫哨, 我们就可以放心地在同一个编译单元及其包含的头文件中多次包含同一个头文件而不会造成重复包含。例如:

```
// stddef.h
#ifndef _STDDEF_H_INCLUDED_
#define _STDDEF_H_INCLUDED_
..... // 头文件的内容
#endif // !_STDDEF_H_INCLUDED_
```

```
// xxx.cpp
#include "stddef.h"
#include "stddef.h"      // No problem!
```

当包含一个头文件的时候,如果能始终如一地使用外部包含卫哨,可以显著地提高编译速度,因为当一个头文件被一个源文件反复包含多次时(常常是因为递归的#include指令展开后所致),可以避免多次查找和打开头文件的操作。例如:

```
#if !defined(_INCLUDED_STDDEF_H_)
#include <stddef.h>
#define _INCLUDED_STDDEF_H_
#endif  // !_INCLUDED_STDDEF_H_
```

建议外部包含卫哨和内部包含卫哨使用同一个标志宏,这样就可以少定义一个宏。例如:

```
#if !defined(_STDDEF_H_INCLUDED_)
#include <stddef.h>
#endif  // !_STDDEF_H_INCLUDED_
```

因为文件 `stddef.h` 中的内部包含卫哨就是 `_STDDEF_H_INCLUDED_`, 所以这里不需要再次定义。

可以仅在头文件中包含其他头文件时使用外部包含卫哨,源文件中可以不使用,也基本不会影响编译速度。

## 9.1.2 头文件包含的合理顺序

无论是在头文件中还是源文件中,在文件开始部分包含其他的头文件时需要遵循一定的顺序。如果包含顺序不当,有可能出现包含顺序依赖问题,甚至引起编译时错误。推荐的顺序如下:

在头文件中:

- (1) 包含当前工程中所需要的自定义头文件(顺序自定);
- (2) 包含第三程序库的头文件;
- (3) 包含标准头文件。

在源文件中:

- (1) 包含该源文件对应的头文件(如果存在);
- (2) 包含当前工程中所需要的自定义头文件;
- (3) 包含第三程序库的头文件;
- (4) 包含标准头文件。

## 9.2 宏定义

可以使用 `#define` 伪指令来定义一个宏。宏分为不带参数的宏和带参数的宏。宏定义以 `#define` 关键字后面出现的第一个连续字符序列作为宏名,剩下的部分作为宏

体。宏定义具有文件作用域，不论宏定义出现在文件中的哪个地方，例如函数体内、类型定义内部、名字空间内部等，在它后面的任何地方都可以引用宏。

宏的几个特点和注意事项如下。

(1) 宏定义不是 C++/C 语句，因此不需要使用语句结束符“;”，否则它也被看做宏体的一部分。例如：

```
#define OUTPUT(word)  cout << #word << endl
```

使用方法为：

```
OUTPUT(I like swimming very much.);
```

替换结果就是：

```
cout << "I like swimming very much." << endl;
```

(2) 任何宏在编译预处理阶段都只是进行简单的文本替换，不做类型检查和语法检查，这个工作留给编译器进行。参数替换发生在宏扩展之前。

(3) 宏定义可以嵌套。例如：

```
#define PI      3.14  
#define PI_2   (2 * PI)
```

(4) 宏不可以调试，因为宏不会进入符号表（符号表是编译器创建的，在编译时宏已经消失了），即使宏替换后出现了语法错误，编译器也会将错误定位到源程序中而不是定位到具体的某个宏定义中。

(5) 程序里使用双引号括起来的字符串中即使出现了与宏同名的子串，预处理过程也不进行替换。

(6) 定义带参数的宏时，宏名和左括号之间不能出现空格，否则使用时会出现问题，但是编译器不会检查出这种错误。例如，宏 TEXT 如果定义为：

```
#define TEXT (str)#str
```

则宏引用语句：

```
cout << TEXT(Hello World);
```

将扩展为：

```
(str)#str(Hello World);
```

这肯定是错误的。

(7) 带参数的宏体和各个形参应该分别用括号括起来，以免造成意想不到的错误。例如：

```
#define SQUARE(x)((x) * (x))
```

如果写成：

```
#define SQUARE(x)x * x
```

那么语句:

```
a = SQUARE(3 + 5);
```

将被扩展为:

```
a = 3 + 5 * 3 + 5;
```

这不是我们所期望的。

(8) 不要在引用宏定义的参数列表中使用增量和减量运算符, 否则将导致变量的多次求值。例如宏引用:

```
int n = 5;
int x = SQUARE(n++);
```

其结果将是 30 而不是期望的 25, 这是因为展开的结果为:

```
int x = ((n++) * (n++));
```

(9) 带参数的宏定义不是函数, 因此没有函数调用的开销, 但是其每一次扩展都会生成重复的代码, 结果使可执行代码的体积增大。

(10) inline 函数不可能完全取代宏, 各有各的好处。用宏来构造一些重复的、数据和函数混合的、功能较特殊的代码段的时候, 其优点就显示出来了。例如, MFC 自己实现的运行时类型识别机制 DECLARE\_DYNCREATE 宏定义, 见示例 9-1。

示例 9-1

```
#define DECLARE_DYNAMIC(class_name) \
protected: \
    static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
    static const AFX_DATA CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const; \
#define DECLARE_DYNCREATE(class_name) \
DECLARE_DYNAMIC(class_name) \
static CObject* PASCAL CreateObject();
```

几乎每一个 MFC 类定义都在开头插入了对宏 DECLARE\_DYNCREATE()或 DECLARE\_DYNAMIC()的引用, 并以该类的名字作为参数, 这样就避免了在每一个新创建的类中都编写大段的类似代码。

(11) 当我们不再使用某一个宏时, 可以使用#undef 来取消其定义。例如:

```
#undef TEXT
```

否则简单地删除宏定义会带来许多编译错误。

**【建议 9-1】:** (1) 虽然宏定义很灵活, 并且通过彼此结合可以产生许多变形用法, 但是 C++/C 程序员不要定义很复杂的宏, 宏定义应该简单而清晰;

(2) 宏名采用大写字母组成的单词或其缩写序列, 并在各单词之间使用

“\_”分隔;

(3) 如果需要公布某个宏, 那么该宏定义应当放置在头文件中, 否则放置在实现文件(.cpp)的顶部;

(4) 不要使用宏来定义新类型名, 应该使用 **typedef**, 否则容易造成错误;

(5) 给宏添加注释时请使用块注释 (`/* */`), 而不要使用行注释。因为有些编译器可能会把宏后面的行注释理解为宏体的一部分;

(6) 尽量使用 **const** 取代宏来定义符号常量;

(7) 对于较长的使用频率较高的重复代码片段, 建议使用函数或模板而不要使用带参数的宏定义; 而对于较短的重复代码片段, 可以使用带参数的宏定义, 这不仅是出于类型安全的考虑, 而且也是优化与折衷的体现;

(8) 尽量避免在局部范围内(如函数内、类型定义内等)定义宏, 除非它只在该局部范围内使用, 否则会损害程序的清晰性。

## 9.3 条件编译

使用条件编译可以控制预处理器选择不同的代码段作为编译器的输入, 从而使得源程序在不同的编译条件下产生不同的目标代码。条件编译伪指令就如同程序控制结构中的选择结构, 不同的是前者只在编译预处理阶段即在正式编译之前发挥作用(不生成运行时代码), 而后者则是在运行时起作用的。条件编译为程序的移植和调试带来了极大的方便, 可以用它来暂时或永久性地阻止一段代码的编译。我们将在下面看到它的价值。条件编译伪指令主要包括 `#if`、`#ifdef`、`#ifndef`、`#elif`、`#else`、`#endif`、`defined`。每一个条件编译块都必须以 `#if` 开始, 以 `#endif` 结束, `#if` 必须与它下面的某一个 `#endif` 配对; `defined` 必须结合 `#if` 或 `#elif` 使用, 而不能单独使用。条件编译块可以出现在程序代码的任何地方。

### 9.3.1 `#if`、`#elif` 和 `#else`

在编程过程中, 当需要暂时放弃编译一段代码的时候, 我们习惯于使用块注释来把它屏蔽掉。但是如果这段代码本身就含有块注释时, 那么双重注释很麻烦。此时我们可以使用条件编译伪指令 `#if` 来屏蔽这段代码, 如下代码所示。如果要使这段代码生效, 只需把 0 改为任何一个非 0 的值(例如 1)即可。但是千万要记住: 不要企图用 `#if` 来代替块注释, 两者根本不是同一种用途。

```
#if 0
... /*...*/    // 希望禁止编译的代码段
... /*...*/    // 希望禁止编译的代码段
#endif
```

由于条件编译由编译预处理器来处理, 显然预编译伪指令无法计算有变量参与

其中的表达式或 `sizeof` 表达式，因此只能用常量表达式。如果常量表达式的值非 0，则条件为真；否则条件为假（见示例 9-2）。

示例 9-2

```
#define FLAG_DOS2
#define FLAG_UNIX 1
#define FLAG_WIN0
#define OS 1
#if OS == FLAG_DOS
    cout << "DOS platform" << endl;
#elif OS == FLAG_UNIX
    cout << "UNIX platform" << endl;
#elif OS == FLAG_WIN
    cout << "Windows platform" << endl;
#else
    cout << "Unknown platform" << endl;
#endif
```

## 9.3.2 `#ifdef` 和 `#ifndef`

预编译伪指令 `#ifdef XYZ` 等价于 `#if defined(XYZ)`，此处 `XYZ` 称为调试宏。如果前面曾经用 `#define` 定义过宏 `XYZ`，那么 `#ifdef XYZ` 表示条件为真，否则条件为假。例如：

```
#define XYZ
...
#ifdef XYZ
    DoSomething();
#endif
```

如果你不想让 `DoSomething();` 语句被编译，那么删除 `#define XYZ`，或者在其后用 `#undef XYZ` 取消该宏即可。

预编译伪指令 `#ifndef XYZ` 等价于 `#if !defined(XYZ)`。示例 9-3 演示了内部包含卫哨的用法。

示例 9-3

```
#ifndef GRAPHICS_H // 防止 graphics.h 被重复引用
#define GRAPHICS_H
#include "myheader.h" // 引用非标准库的头文件
#include <math.h> // 引用标准库的头文件
... // 一些函数声明和类型定义
#endif
```



## 9.4 #error

编译伪指令#error 用于输出与平台、环境等有关的信息（见示例 9-4）。

示例 9-4

```
#if !defined(WIN32)
    #error ERROR: Only Win32 platform supported!
#endif
#ifdef __cplusplus
    #error MFC requires C++ compilation (use a .cpp suffix)
#endif
```

当预处理器发现应用程序中没有定义宏 WIN32 或者\_\_cplusplus 时,那么把#error 后面的字符序列输出到屏幕后即终止,程序不会进入编译阶段。

## 9.5 #pragma

编译伪指令#pragma 用于执行语言实现所定义的动作（见示例 9-5），具体参考你所使用的编译器的帮助文档。

示例 9-5

```
#pragma pack(push, 8)          /* 对象成员对齐字节数 */
#pragma pack(pop)
#pragma warning(disable:4069)  /* 不要产生第 C4069 号编译警告 */
#pragma comment(lib, "kernel32.lib")
#pragma comment(lib, "user32.lib")
#pragma comment(lib, "gdi32.lib")
```

## 9.6 #和##运算符

构串操作符#只能修饰带参数的宏的形参,它将实参的字符序列（而不是实参代表的值）转换成字符串常量。例如:

```
#define STRING(x)    #x    #x    #x
#define TEXT(x)      "class" #x "Info"
```

那么宏引用:

```
int abc = 100;
STRING(abc)
```

TEXT(abc)

展开后的结果分别为：

"abcabcabc"  
"classabcInfo"

合并操作符 `##` 将出现在其左右的字符序列合并成一个新的标识符（注意，不是字符串）。例如：

```
#define CLASS_NAME(name) class##name
#define MERGE(x, y) x##y##x
```

则宏引用：

```
CLASS_NAME(SysTimer)
MERGE(me, To)
```

将分别扩展为如下两个标识符：

```
classSysTimer
meTome
```

使用合并操作符`##`时，产生的标识符必须预先有定义，否则编译器会报“标识符未定义”的编译错误。

9.7 预定义符号常量

C++继承了ANSI C的预定义符号常量（见表 9-1），预处理器在处理代码时将它们替换为确定的字面常量。这些符号不能用`#define`重新定义，也不能用`#undef`取消。

表 9-1 C++/C 预定义符号常量

符 号 常 量	解 释
LINE	引用该符号的语句的代码行号
FILE	引用该符号的语句的源文件名称
DATE	引用该符号的语句所在源文件被编译的日期（字符串）
TIME	引用该符号的语句所在源文件被编译的时间（字符串）
TIMESTAMP	引用该符号的语句所在源文件被编译的日期和时间（字符串）
STDC	标准 C 语言环境都会定义该宏以标识当前环境

表 9-1 中的预定义符号常量可以被直接引用，常用来输出调试信息和定位异常发生的文件及代码行（见示例 9-6）。

示例 9-6

```
double * const pDouble = new(nothrow) double[10000000];
if( pDouble == NULL ) {
```

```
cerr << "allocate memory failed on line " << (__LINE__ - 2) \
    << "in file " << __FILE__ << endl;
}
```

---

如果你不习惯它们的名称，你可以自己定义喜欢的名称。例如：

```
#define THIS_FILE    __FILE__
#define THIS_LINE    __LINE__
```

某些编译器在此基础上也会定义一些自己的宏，例如 `__cplusplus` 等。如果你要使用这些宏，请参考具体的编译器文档。



# 第 10 章 C++/C 文件结构和程序版式

C++/C 的文件结构和程序版式并不影响功能，也无多少技术含量，但是能够反映出开发者的职业化程度。我们当然希望自己的程序看上去是专业级的。

版式虽然不会影响程序的功能，但是会影响清晰性。程序的版式追求清晰、美观，是程序风格的重要因素。可以把程序的版式比喻为“书法”，好的“书法”可让人对程序一目了然，看得兴致勃勃。程序员们学习程序的“书法”，弥补大学计算机教育的漏洞，实在很有必要。

## 10.1 程序文件的目录结构

一个正在开发中的程序工程不仅包含源代码文件，还包含许多资源文件、数据文件、库文件及配置文件等。理论上，这些文件在磁盘上的存放位置是很自由的，即使是 IDE 也没有什么强制规定。但是为了便于开发与维护，最好有一目了然的组织方式（即目录结构）。例如，我们创建一个 C++/C 程序工程 TestProj，其文件可以参照图 10-1 所示的结构来组织。

图 10-1 中目录的用途如下。

(1) Include 目录存放应用程序的头文件 (.h)，还可以再细分子目录。

(2) Source 目录存放应用程序的源文件 (.c 或 .cpp)，还可以再细分子目录。

(3) Shared 目录存放一些共享的文件。

(4) Resource 目录存放应用程序所用的各种资源文件，包括图片、视频、音频、图标、光标、对话框等，还可以再细分子目录。

(5) Debug 目录存放应用程序调试版本生成的中间文件。

(6) Release 目录存放应用程序发行版本生成的中间文件。

(7) Bin 目录存放程序员自己创建的 lib 文件和 dll 文件。



图 10-1 开发目录结构

### 【提示 10-1】

分清楚编译时相对路径和运行时相对路径的不同，这在编写操作 DLL 文件、INI 文件及数据文件等外部文件的代码时很重要，因为它们的“参照

物”不同。例如下面的代码行:

```
#include "..\include\abc.h"
```

是相对于当前工程所在目录的路径,或者是相对于当前文件所在目录的路径,在编译选项的设置中也有这样的路径。

而下面一行代码:

```
OpenFile("..\\abc.ini");
```

则是相对于运行时可执行文件所在目录的路径,或者是相对于你为当前程序设置的工作目录的路径。

## 10.2 文件的结构

C++/C 源程序通常分为两类文件。一类文件用于保存程序的声明,称为头文件;另一类文件用于保存程序的实现,称为源文件(或者定义文件)。

C++/C 程序的头文件以“.h”为后缀,C 程序的源文件以“.c”为后缀,C++程序的源文件通常以“.cpp”为后缀(也有一些系统以“.cc”或“.cxx”为后缀)。

C++/C 编译器在扫描到一条函数调用语句时首先应当知道该函数的原型或定义,函数原型一般都放在头文件中,函数定义则放在源文件中,当源文件或头文件通过#include 指令包含另一个头文件的时候,编译预处理器用头文件的内容取代#include 伪指令。这就是说,头文件的所有内容最终都会被合并到某一个或某几个源文件中,如此将每一个包含的头文件递归地展开后形成的源文件就叫编译单元。

### 10.2.1 头文件的用途和结构

早期的编程语言如 BASIC、Fortran 没有头文件的概念,C++/C 语言的初学者虽然会使用头文件,但常常不明其理。这里对头文件的作用略做解释。

#### 【提示 10-2】:

(1) 通过头文件来调用库功能。在很多场合,源代码不便(或不准)向用户公布,只要向用户提供头文件和二进制的库即可。用户只需按照头文件中的接口声明来调用库函数,而不必关心接口是怎么实现的。连接器会从库中提取相应的代码,并和用户的程序连接生成可执行文件或者动态连接库文件;

(2) 头文件能加强类型安全检查。如果某个接口被实现或被使用时的方式与头文件中的声明不一致,编译器就会指出错误,这一简单的规则能大大减轻程序员调试、改错的负担;

(3) 头文件可以提高程序的可读性(清晰性)。

头文件中的元素比较多,其顺序(结构)一般应安排如下:

- (1) 头文件注释(包括文件说明、功能描述、版权声明等)(必须有);
- (2) 内部包含卫哨开始(#ifndef XXX/#define XXX)(必须有);

- (3) `#include` 其他头文件 (如果需要);
- (4) 外部变量和全局函数声明 (如果需要);
- (5) 常量和宏定义 (如果需要);
- (6) 类型前置声明和定义 (如果需要);
- (7) 全局函数原型和内联函数的定义 (如果需要);
- (8) 内部包含卫哨结束: `#endif` // XXX (必须有);
- (9) 文件版本及修订说明。

上述排列顺序并非绝对, 也不存在对错之分, 可根据具体情况灵活安排。

如果程序中需要内联函数, 那么内联函数的定义应当放在头文件中, 因为内联函数调用语句最终被扩展开来而不是采用真正的函数调用机制。

### 10.2.2 版权和版本信息

你知道为什么要在源程序中加入版权和版本信息吗?

如果你在企业里工作, 那么请记住: 你已不再是学生了, 你编写的程序属于企业。所以要给每个程序打上企业的“烙印”, 即版权和版本声明。

版权、版本信息的主要内容有:

- (1) 版权信息;
- (2) 文件名称、简要描述、创建日期和作者;
- (3) 当前版本信息和说明;
- (4) 历史版本信息和修订说明。

见示例 10-1。

示例 10-1

```

/*****
*   Copyright (c) 2002-2005   Company Name.
*   All rights reserved.
*
*   文件名称: filename.h
*   简要描述: 简要描述该文件的内容、功能
*
*   当前版本: 2.0
*   作者/修改者:
*   完成日期:
*   修订说明:
*
*   取代版本: 1.0
*   修 改 人:
*   完成日期:
*   修订说明:
*****/

```

**【提示 10-3】** 任何源程序都要使用配置管理工具（例如 SourceSafe、ClearCase 和 CVS 等）进行管理（例如版本控制、变更控制等）。源程序中版本信息仅仅起到提示作用，所以不必太详细（免得太长而喧宾夺主）。

## 10.2.3 源文件结构

C 和 C++ 源文件主要保存函数的实现和类的实现，其结构一般如下：

- (1) 源文件注释（包括文件说明、功能描述、版权声明等）（必须有）；
- (2) 预处理指令（如果需要）；
- (3) 常量和宏定义（如果需要）；
- (4) 外部变量声明和全局变量定义及初始化（如果需要）；
- (5) 成员函数和全局函数的定义（如果需要）；
- (6) 文件修改记录。

同样，上述这种排列顺序并非绝对的，也不存在对错之分，可以根据具体情况灵活安排。

## 10.3 代码的版式

### 10.3.1 适当的空行

写文章要分段落，写代码也要分段落。空行起着分割段落的作用，空行得体（不过多也过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得，所以不要舍不得用空行。

但是空行也不能过度使用。段落之间一般留一个或两个空行，如果连续空出  $n$  行（ $n > 2$ ），反而使代码看上去不够紧凑，给人“七零八落”的感觉。

**【建议 10-1】** 哪些地方该使用空行呢？一般约定如下：

- (1) ADT/UDT 定义之间要留空行。ADT 内部的各个访问段（**public**、**private** 等）之间要留空行；每一段内按照相关性分组的，建议在各组之间留空行，没有明显相关性的不需要分组；
- (2) 函数定义之间要留空行。在函数体内，完整的控制结构及单独的语句块之间要分别留出空行，它们与其他段落之间也要留出空行以示区分；逻辑上密切相关的语句序列之间不要留空行（例如初始化数据成员的一系列语句）；最后一条 **return** 语句前要留空行，除非该函数只有这一条语句；控制结构、语句块、条件编译块等遵循同样的规则；
- (3) 注释行或注释块与被它们注释的程序元素之间不要留空行。



### 10.3.2 代码行及行内空格

- 【建议 10-2】:**
- (1) 不要书写复杂的语句行，一行代码只做一件事情，如只定义一个变量或只写一条语句，这样书写出来的代码容易阅读，并且便于写注释；
  - (2) **if**、**elseif**、**for**、**do**、**while** 等语句各自占一行，其他语句不得紧跟其后；不论该语句块中有多少行语句都要用“**{}**”括起来，这样可以防止书写失误。当你准备开始书写语句块时，首先写下一对“**{}**”，然后再在里面书写语句，这样可以避免不易察觉的逻辑错误；
  - (3) 局部变量在定义的同时应当初始化，而且要在同一行内初始化。这是因为，对于局部变量，系统不会自动初始化它们，在运行时它们的内存单元将保留上次使用以来留下的脏值。再者，如果变量的引用处距其定义处较远，变量的初始化就容易被遗忘。如果引用了未初始化的变量，很有可能导致程序运行错误。

见示例 10-2。

示例 10-2

风格不良的代码行	风格良好的代码行
<code>int width, height, depth; // 宽度高度深度</code>	<pre>int width = 10;    // 宽度 int height = 10;   // 高度 int depth = 10;    // 深度</pre>
<pre>x = a + b;   y = c + d;   z = e + f;  if(width &lt; height) dosomething();</pre>	<pre>x = a + b; y = c + d; z = e + f; if(width &lt; height) {     dosomething(); }</pre>
<pre>for(initialization; condition; update)     dosomething(); other();</pre>	<pre>for(initialization; condition; update) {     dosomething(); } // 空行 other();</pre>

关于使用空格的建议如下。

- 【建议 10-3】:**
- (1) 关键字之后要留空格。像 **const**、**virtual**、**inline**、**case** 等关键字之后至少要留一个空格，否则无法辨析关键字。像 **if**、**elseif**、**for**、**while**、**switch** 等关键字之后应留一个空格再跟左括号“(”，以突出关键字；
  - (2) 函数名之后不留空格，无论是在原型、定义还是在调用中；
  - (3) “(”、“[”向后紧跟，“]”、“.”、“;”、“)”向前紧跟，紧跟处不留空格；“,”之后要留空格，如 `f(x, y, z)`；如果“;”不是一行的结束符，

则后面也要留空格，如 `for (initialization; condition; update);`

(4) 预编译指令中 `#` 和保留字之间不要留空格；文件包含伪指令中文件名与两端的 `<`、`>` 或 `"`、`"` 之间不留空格；

(5) 二元运算符如 `=`、`+=`、`>=`、`<=`、`+`、`*`、`%`、`&&`、`||`、`<<`、`^` 等的前后应当加空格；

(6) 一元运算符如 `!`、`~`、`++`、`--`、`-`、`&`（取地址运算符）、`*`（反引用）等与所作用的操作数之间不加空格；

(7) `.`、`->`、`.*`、`->*`、`::` 这类运算符前后不加空格；`?`、`:` 前后要加空格。

见示例 10-3。

示例 10-3

<code>void Func1(int x, int y, int z);</code>	//良好的风格
<code>void Func1(int x,int y,int z);</code>	//不良的风格
<code>if(year &gt;= 2000)</code>	//良好的风格
<code>if(year&gt;=2000)</code>	//不良的风格
<code>if((a &gt;= b) &amp;&amp; (c &lt;= d))</code>	//良好的风格
<code>if(a&gt;=b&amp;&amp;c&lt;=d)</code>	//不良的风格
<code>for(i = 0; i &lt; 10; i++)</code>	//良好的风格
<code>for(i=0;i&lt;10;i++)</code>	//不良的风格
<code>x = a &lt; b ? a : b;</code>	//良好的风格
<code>x=a&lt;b?a:b;</code>	//不好的风格
<code>int *x = &amp;y;</code>	//良好的风格
<code>int * x = &amp; y;</code>	//不良的风格
<code>array[5] = 0;</code>	//不要写成 <code>array [ 5 ] = 0;</code>
<code>a.MemFunc();</code>	//不要写成 <code>a . MemFunc ();</code>
<code>b-&gt;MemFunc();</code>	//不要写成 <code>b -&gt; MemFunc ();</code>

### 10.3.3 长行拆分

代码行最大长度宜控制在 70 至 80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。有时候会碰到比较长的语句，则可以在一些“断点”处折行。

**【建议 10-4】：**长表达式要在低优先级运算符处拆分为多行，运算符放在新行之首（以示突出）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。这些运算符常见的有二元逻辑运算符、输入输出运算符等（见示例 10-4）。

示例 10-4

```
if ( (very_longer_variable1 >= very_longer_variable12)
    && (very_longer_variable3 <= very_longer_variable14)
    && (very_longer_variable5 <= very_longer_variable16) )
```

```

    {
        dosomething();
    }
virtual CMatrix CmultiplyMatrix( CMatrix leftMatrix,
                                CMatrix rightMatrix );

for ( very_longer_init-exp;
      very_longer_condition;
      very_longer_update )
{
    dosomething();
}

```

#### 10.3.4 对齐与缩进

**【建议 10-5】：** 程序的分界符“{”和“}”应独占一行并且位于同一列，同时与引用它们的语句左对齐。{} 之中的代码块在“{”右边数格处左对齐，建议采用一个“\t”字符或 4 个空格。各层嵌套请使用统一的缩进（见示例 10-5）。在另外一些情况下，为了不使语句块和{} 看起来有被架空的感觉，可将{} 和其内部的语句块左对齐，并且都向右缩进 4 格，如示例 10-4 所示。

示例 10-5

为了节约版面的对齐与缩进方式	编程推荐的对齐与缩进方式
<pre>void Function(int x){     ... // program code }</pre>	<pre>void Function(int x) {     ... // program code }</pre>
<pre>if(condition){     ... // program code }else {     ... // program code }</pre>	<pre>if(condition) {     ... // program code } else {     ... // program code }</pre>
<pre>for(initialization; condition; update){     ... // program code }</pre>	<pre>for(initialization; condition; update) {     ... // program code }</pre>
<pre>while (condition){     ... // program code }</pre>	<pre>while(condition) {     ... // program code }</pre>
	<p>如果出现嵌套的 {}，则使用缩进对齐，并且每层缩进的格数要相同。如：</p> <pre>{</pre>

	...
	{
	...
	}
	...
	}

## 10.3.5 修饰符的位置

修饰符 `*` 和 `&` 应该靠近数据类型名还是靠近变量名，是一个有争议的话题。若将修饰符 `*` 靠近数据类型，例如：

```
int*    px = NULL;
```

从语义上讲这种写法比较直观，即 `px` 是 `int` 类型的指针。但是此写法在某些情况下容易引起误解，例如：

```
int*    x = NULL, y = NULL;
```

此处 `y` 被误解为指针变量，其实它是 `int` 类型的变量（不该初始化为 `NULL`）。虽然将 `x` 和 `y` 分行定义可以避免误解，但并非人人都愿意这样做。

**【建议 10-6】：** 建议将修饰符 `*` 和 `&` 紧靠变量名，例如：

```
char    *name;
int     *x, y;    // 此处 y 不会被误解为指针
```

或者使用 `typedef` 做一个类型映射，例如：

```
typedef int*  PINT;
typedef int&  RINT;
PINT  p1, p2;    // 两个指针
RINT  r1 = i, r2 = j; // 两个引用
```

## 10.3.6 注释风格

C 语言的注释符为 `/*...*/`。在 C++ 语言中，程序块的注释常采用 `/*...*/`，而行注释一般采用 `//...`。不过，注释的标志是可扩展的语言特征，有些 IDE 就可以支持在 C 代码中使用 `//...` 注释，例如 Microsoft C。

注释通常用于：

- (1) 版本、版权声明；
- (2) 函数接口说明；
- (3) 重要的代码行或段落提示。

使用注释的建议如下。

**【建议 10-7】：** (1) 注释是对代码的“提示”，而不是文档。程序中的注释不可喧宾夺主，注释太多了会让人眼花缭乱。注释的花样要少，不要用注释拼图案；  
(2) 如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌

烦。例如：

```
i++;      //i 加 1 （多余的注释）
```

(3) 边写代码边加注释，修改代码的同时修改相应的注释，以保证注释与代码的一致性；

(4) 不再有用的注释要删除掉。注释应当准确、易懂，防止出现二义性。错误的注释不但无益而且有害；

(5) 注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

见示例 10-6。

示例 10-6

<pre>/*  * 函数头注释  */ void Function(float x, float y, float z) {     ... }</pre>	<pre>if(...) {     ...     while(...)     {         ...     } // end of while     ... } // end of if</pre>
---	--

函数头注释主要用来说明函数的功能及其参数和返回值的信息。对于那些十分简单的函数，可以不加注释或只用一行注释来说明其作用。对于比较复杂的函数，建议采用示例 10-7 的函数头注释。

示例 10-7

<pre>/******  * 函数名称：函数名  * 功能描述：简要描述该函数实现的功能  * 参数列表：param1——描述；  *             param2——描述；  *             param3——描述；  * 返回结果：描述返回值的情况  *****/</pre>
--

### 10.3.7 ADT/UDT 版式

ADT/UDT（例如类）的版式主要有两种形式。

(1) 将 `private` 限定的成员写在前面，而将 `public` 限定的成员写在后面。采用这

种版式的程序员一般是主张“以数据为中心”设计 ADT/UDT，重点关注其内部结构；

(2) 将 `public` 限定的成员写在前面，而将 `private` 限定的成员写在后面。采用这种版式的程序员一般是主张“以行为为中心”设计 ADT/UDT，重点关注其提供的接口（或服务）。

见示例 10-8。

估计很多 C++ 教科书受到 Bjarne Stroustrup 第一本著作（《The C++ Programming Language》）的影响，不知不觉地采用了“以数据为中心”的方式来编写类，这样做并不见得有多少道理。

**【建议 10-8】：** 建议采用“以行为为中心”的方式来编写类，即首先考虑类应该提供什么样的接口（即函数）。这是很多人的经验——“这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。因为用户最关心的是接口，谁愿意先看到一堆私有数据成员！”

示例 10-8

以数据为中心的版式	以行为为中心的版式
<pre>class A {   private:     int   i;     float x;     ...    public:     void Func1(void);     void Func2(void);     ... }</pre>	<pre>class A {   public:     void Func1(void);     void Func2(void);     ...    private:     int   i;     float x;     ... }</pre>

## 第 11 章 C++/C 应用程序命名规则

每个人都有名字，为了让子女的名字好听并且有意义，多少父母翻破字典、挖空心思啊！在编程过程中，给标识符取个合适的名字也是很重要的，需要程序员动动脑筋。

在软件领域，比较著名的命名规则当推 Microsoft 公司的“匈牙利”命名法，该命名规则的中心思想是“在标识符中加入能够表明其类型和作用域属性的前缀，以增进人们对标识符的理解”。例如所有的字符变量均以 `ch` 为前缀，若是指针变量则加前缀 `p`。如果一个变量由 `ppch` 开头，则表明它是指向字符指针的指针。

“匈牙利”命名法最大的缺点就是烦琐。例如下面的定义：

```
int    i, j, k;  
float  x, y, z;
```

倘若采用“匈牙利”命名法，则应当写成：

```
int    iI, iJ, iK;           // 前缀 i 表示 int 类型  
float  fX, fY, fZ;           // 前缀 f 表示 float 类型
```

如此烦琐的命名法会让大多数程序员无法忍受。

据考查，没有一种命名规则可以适合所有程序员的口味，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败攸关”的事，我们不要花太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

### 11.1 共性规则

本节讨论的共性规则是为大多数程序员所接受的，我们应当在首先遵循这些共性规则的前提下再扩充特定的规则（参见 11.2 节）。

**【规则 11-1】** 标识符的名字应当直观且可以拼读，可望文知义，不必进行“解码”。

标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不要太复杂，而且用词应当准确。例如不要把 `currentValue` 写成 `nowValue`。

**【规则 11-2】** 标识符的长度应当符合“min-length & max-information”原则。

几十年前 ANSI C 规定名字不准超过 6 个字符，现今的 C/C++ 不再有此限制。一般说来，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符已不足为怪。那么名字是否越长越好？不一定！例如变量名 `maxValue` 就比 `maxValueUntilOverflow` 好用。单字符的简单名字也是有用的，常见的如 `i, j, k, m, n, x`,

y, z 等，它们通常用做程序块内的局部变量名，例如循环计数器。

有时候标识符采用的英文单词太长，几个单词组合后会更长，此时应该采用一些通用而合理的缩写或者应用领域专业术语的缩写。通常这些缩写会在编程规范中给出，可作为参考。

**【规则 11-3】：** 程序中不要出现仅靠大小写来区分的相似标识符，虽然 C++/C 是大小写相关的。

例如：

```
int x, X;           // 变量 x 与 X 容易混淆
void foo(int x);     // 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

**【规则 11-4】：** 不要使程序中出现局部变量和全局变量同名的现象，尽管由于两者的作用域不同而不会发生语法错误，但会使人误解。

**【规则 11-5】：** 变量的名字应当使用“名词”或者“形容词 + 名词”的格式来命名。

例如：

```
float value;
float oldValue;
float newValue;
```

**【规则 11-6】：** 全局函数的名字应当使用“动词”或者“动词 + 名词”（动宾词组）。类的成员函数应当只使用“动词”，被省略掉的名词就是对象本身。

例如：

```
DrawBox();          // 全局函数
box.Draw();          // 类的成员函数
```

**【规则 11-7】：** 用正确的反义词组命名具有相反意义的变量或相反动作的函数等。

例如：

```
int minValue;
int maxValue;
int SetValue(...);
int GetValue(...);
```

**【建议 11-1】：** 尽量避免名字中出现数字编号，如 value1, value2 等，除非逻辑上的确需要如此。这是为了防止程序员偷懒，不肯为动脑筋而用无意义的名字（因为用数字编号最省事），就像没有人会给子女起名叫张三或李四一样。

## 11.2 简单的 Windows 应用程序命名

作者对“匈牙利”命名规则做了合理的简化，下述的命名规则简单易行，比较适合于 Windows 应用程序的开发。



**【建议 11-2】：** 类型名和函数名均以大写字母开头的单词组合而成。

例如：

```
class Node;           // 类名
class LeafNode;       // 类名
void Draw(void);      // 函数名
void SetValue(int value); // 函数名
```

**【建议 11-3】：** 变量名和参数名采用第一个单词首字母小写而后面的单词首字母大写的单词组合。

例如：

```
BOOL flag;
int drawMode;
```

**【建议 11-4】：** 符号常量和宏名用全大写的单词组合而成，并在单词之间用单下划线分隔，注意首尾最好不要使用下划线。

例如：

```
const int MAX = 100;
const int MAX_LENGTH = 1024;
```

**【建议 11-5】：** 给静态变量加前缀 `s_`（表示 `static`）。

例如：

```
void Init(...)
{
    static int s_initValue; // 静态变量
    ...
}
```

**【建议 11-6】：** 如果不得已需要全局变量，这时全局变量加前缀 `g_`（表示 `global`）。

例如：

```
int g_howManyPeople; // 全局变量
int g_howMuchMoney;  // 全局变量
```

**【建议 11-7】：** 类的数据成员加前缀 `m_`（表示 `member`），这样可以避免数据成员与成员函数的参数同名。

例如：

```
void Object::SetValue(int width, int height)
{
    m_width = width;
    m_height = height;
}
```

**【建议 11-8】：** 为了防止某一软件库中的一些标识符和其他软件库中的冲突，可以统一为各种标识符加上能反映软件性质的前缀。

例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。还有更好的办法是使用名字空间。

# 第 12 章 C++面向对象程序

## 设计方法概述

会用 C++ 的程序员就一定懂得面向对象程序设计吗？

不会用 C++ 的程序员就一定不懂得面向对象程序设计吗？

两者都未必。

---

我曾经和许多 C++ 程序员一样，在享用到 C++ 语法的好处时便以为自己已经明白了面向对象程序设计方法。就这样糊里糊涂地编写了十几万行 C++ 程序，如此使用 C++，就像挤掉牙膏卖牙膏皮那样，真是暴殄天物呀。

---

本章目的不是阐述面向对象的理论，而是用浅显的示例来解说面向对象程序设计的一些重要概念，如封装、继承、组合、虚函数、抽象基类、动态绑定和多态性等。本章对本书后面章节有指导意义。

### 12.1 漫谈面向对象

第一次世界大战结束前夕，一个维也纳人在战俘营里写了一本《逻辑哲学论》。这本 75 页的小册子提出了对象的观点：

世界可以分解为事实；  
事实是由原子事实组成的；  
一个原子事实是多个对象的组合；  
对象是简单的；  
对象形成了世界的基础。

50 年之后，面向对象（Object-Oriented, OO）方法论火起来了，现在“对象”真的成为了软件世界的基础。

面向对象分析和设计（OOAD）方法兴起于 20 世纪 80 年代，从 20 世纪 90 年代起至今它已经在分析设计领域占据了无可争议的主流地位。作者在读本科（1990 年至 1994 年）时就充分感受到了人们对“面向对象”的狂热，有关“面向对象”的课堂、学术报告常常人满为患，搞软件开发的人都“言必谈对象”，并引以为荣。

面向对象分析设计领域有一些比较著名的学派，如：

◇ Coad 和 Yourdon 学派

- ◇ Booch 学派
- ◇ Jacobson 学派
- ◇ Rumbaugh 学派

有趣的是, 这些学派的掌门人就像上帝、真主、如来佛, 他们用各自的方式定义了这个世界, 并留下一堆经书来解释这个世界。这种混乱的局面被学术界称为“百家争鸣”, 于是每年都会诞生许多论著和教授。叫苦的是软件企业和开发人员: 没有统一的方法, 不好干活呀!

终于等到了那么一天, Rational 公司招纳了 Booch、Jacobson 和 Rumbaugh, 这三位“面向对象”业界的老大强强联手, 制定了“统一建模语言”(UML)。1997 年 11 月, UML 被国际对象管理组织 (OMG) 采纳, 此后 UML 成为 OOAD 建模语言的国际标准。

同样有趣的是, 面向对象编程语言却比 OOAD 方法论更早问世。最早的面向对象编程语言是 Smalltalk, 由施乐公司研究中心于 1970 年研制成功。在软件开发领域, 编程实践往往先行于相应的理论。就如人类的进化: 先学会讲话, 后来才产生文字。用程序员的行话讲, 这叫“编程是硬道理”。

六七年前, 作者刚“热恋”面向对象时急切地想知道什么是面向对象, 于是买了一堆书来阅读。不少书籍建议这样找“对象”: 分析一个句子的语法, 找出名词和动词, 名词就是对象, 动词则是对象的方法 (即函数)。

天哪! 这可不是程序员们喜欢的做法!

建议程序员应当先学习用 C++ 或者 Java 编写程序, 当对面向对象程序设计有了感性认识之后, 再阅读面向对象理论书籍 (其中最著名的书当推 Lippman 所著《Inside The C++ Object Model》), 这样才能深入理解面向对象方法的精髓。

面向对象编程语言很多, 如 Smalltalk、Ada、Eiffel、C++ 和 Java 等。C++ 语言最受程序员欢迎, 因为它兼容 C 语言, 所以应用最广泛。Java 是一种纯面向对象语言, 它诞生之初曾红极一时, 不少人叫喊着“用 Java 革 C++ 的命”。

## 12.2 对象的概念

面向对象思想把整个世界看做是由具有行为的各种对象组成的, 任何对象都具有某种特征和行为。比如人是一种对象, 计算机也是一种对象; 人有姓名、年龄、身高和体重等特征, 计算机有颜色、重量和品牌等特征; 人能够吃、喝、拉、撒、睡, 计算机能够开机、关机、播放音乐、播放 VCD……

OOAD 把一个对象的特征称为属性, 而把其行为称为服务或方法。于是可以借用著名的“客户机/服务器 (C/S)”体系结构模型来描述面向对象系统中各个对象之间的协同工作原理。由于对象具有行为, 并且可以接受外界的信息 (参数传递), 因此对外表现为能够提供一定的服务。当我们向一个对象传递参数并调用对应的函数时, 就是在请求其提供服务。我们总是通过一个对象 (大到整个进程对象, 小到一个 ADT 对象) 向另一个对象请求服务, 于是发出请求的对象就是“客户机”, 而提

供服务的对象就是“服务器”。但是这种“客户机/服务器”的概念并非绝对的，客户机在某个时候可能成为服务器，而服务器在某个时候也可能成为客户机。这样，对象之间就可以通过它们能够提供的服务来交流，进而合作完成特定的任务。

搞清楚对象的概念对于理解面向对象系统尤其是分布式面向对象系统非常重要，比如 COM 和 CORBA。

在面向对象软件领域，完全可以把对象看成是一种软件模块，整个软件就是由各种各样的运行时对象构成的一个系统；甚至还可以把软件系统的一个线程、进程乃至子系统当做对象来封装、设计和实现。

## 12.3 信息隐藏与类的封装

在一节不和谐的课堂里，老师叹气道：“要是坐在后排聊天的同学能像中间打牌的同学那么安静的话，就不会影响到前排睡觉的同学了。”

这个故事告诉我们，如果不想让坏事传播开来，就应该把坏事隐藏起来，因为向来就是“好事不出门，坏事传千里”。“家丑不可外扬”就是这个道理。

对于软件设计而言，为了尽量避免某个模块的行为干扰同一系统中的其他模块，应该让模块仅仅公开必须让外界知道的内容，而隐藏其他一切内容。

“信息隐藏”这种设计理念产生了 C++ 类的封装特性。

C++ 对 C 的最根本的改变就是把函数放进了结构之中，从而产生了 C++ 类。类把数据和函数捆绑在一起，其中数据表示类的属性（数据成员），函数表示类的行为，也称为成员函数、方法或者服务。C++ 提供了关键字 `public`、`private` 和 `protected` 用于声明哪些数据和函数是可以公开访问的、私用的或者是受保护（受限访问）的，这样就达到了信息隐藏的目的，即让类仅仅公开必须让外界知道的内容，而隐藏其他一切内容（见示例 12-1）。

示例 12-1

```
class WhoAmI {  
public:  
    void GetMyName(void);    // 名字是可以公开的  
protected:  
    void GetMyAsset(void);    // 财产是受保护的，只有我和继承者可以使用  
private:  
    void GetMyGuilty(void);  // 罪过是要保密的，只有自己才能偷看  
    ...  
};
```

类的封装特性是 C++ 的基本语法之一，易学易用。但要注意的是，不可以滥用类的封装特性，不要把不相干的数据和函数封装到类里头，不要把类当成火锅，什么东西都往里扔。

为了更好地体会封装的特点，下面举两个例子。同步队列的实现见示例 12-2，

循环缓冲区的实现见示例 12-3。

示例 12-2

```
class CriticalSection {
public:
    CriticalSection() { ::InitializeCriticalSection(&m_critSection); } // WIN32 API
    ~CriticalSection() { ::DeleteCriticalSection(&m_critSection); } // WIN32 API
    void lock() { ::EnterCriticalSection(&m_critSection); } // WIN32 API
    void unlock() { ::LeaveCriticalSection(&m_critSection); } // WIN32 API
private:
    CRITICAL_SECTION m_critSection; // Windows 临界区
private:
    CriticalSection(const CriticalSection&); // 不可拷贝
    void operator=(const CriticalSection&); // 不可赋值
};

template<class SyncObject>
class GenericLocker {
public:
    explicit GenericLocker(const SyncObject& so)
        : m_refObj(const_cast<SyncObject*>(so)) { m_refObj.lock(); }
    ~GenericLocker() { m_refObj.unlock(); }
private:
    GenericLocker(const GenericLocker<SyncObject>&); // 不可拷贝
    void operator=(const GenericLocker<SyncObject>&); // 不可赋值
private:
    SyncObject& m_refObj;
};

template<typename T>
class SyncQueue {
public:
    typedef typename std::deque<T>::size_type size_type;
    typedef typename std::deque<T>::value_type value_type;
    typedef GenericLocker<CriticalSection> _QueueLocker;
    size_type size() const {
        _QueueLocker guard(m_mutex);
        return (m_deque.size());
    }
    bool empty() const {
        _QueueLocker guard(m_mutex);
        return (m_deque.empty());
    }
    void clear() {
        _QueueLocker guard(m_mutex);
        m_deque.clear();
    }
    void push(const value_type& x) {
```

```

        _QueueLocker guard(m_mutex);
        m_deque.push_back(x);
    }
    bool pop(value_type& value) {
        _QueueLocker guard(m_mutex);
        if (m_deque.empty())
            return false;
        value = m_deque.front();
        m_deque.pop_front();
        return true;
    }
protected:
    std::deque<T>      m_deque;           // 用 deque 来实现同步队列
    CriticalSection    m_mutex;
};

```

示例 12-3

```

typedef unsigned char  BYTE;
template<unsigned int N /*容量（字节数）*/>
class RingBuffer {
public:
    typedef size_t      size_type;
    typedef GenericLocker<CriticalSection> _BufferLocker;
    RingBuffer() : m_pushPos(0), m_popPos(0), m_count(0) {
        m_pRingBuffer = new BYTE[N];
    }
    ~RingBuffer() { delete []m_pRingBuffer; }
    bool is_full() const {
        _BufferLocker guard(m_mutex);
        return (m_count == N);
    }
    bool is_empty() const {
        _BufferLocker guard(m_mutex);
        return (m_count == 0);
    }
    size_type size() const {
        _BufferLocker guard(m_mutex);
        return m_count;
    }
    size_type capacity() const { return N; }
    size_type push(const BYTE *data, size_type length) {
        _BufferLocker guard(m_mutex);
        assert(data != NULL);
        if (length == 0 || length > (N - m_count))
            return 0;
        size_type rearLen = N - m_pushPos;           // 尾部剩余空间

```

```

        if (length <= rearLen) {
            ::memmove(&m_pRingBuffer[m_pushPos], data, length);
            m_pushPos += length;
            m_pushPos %= N;                // 调整新的 push 位置
        } else {
            ::memmove(&m_pRingBuffer[m_pushPos], data, rearLen);
            ::memmove(m_pRingBuffer, data + rearLen, length - rearLen);
            m_pushPos = length - rearLen;    // 调整新的 push 位置
        }
        m_count += length;
        return (length);
    }

    size_type pop(BYTE *buf, size_type length) {
        _BufferLocker guard(m_mutex);
        assert(buf != NULL);
        if (length == 0 || length > m_count)
            return 0;
        size_type rearLen = N - m_popPos;    // 尾部剩余数据
        if (length <= rearLen) {
            ::memmove(buf, &m_pRingBuffer[m_popPos], length);
            m_popPos += length;
            m_popPos %= N;                // 调整新的 pop 位置
        } else {
            ::memmove(buf, &m_pRingBuffer[m_popPos], rearLen);
            ::memmove(buf + rearLen, m_pRingBuffer, length - rearLen);
            m_popPos = length - rearLen;    // 调整新的 pop 位置
        }
        m_count -= length;
        return (length);
    }

    void clear() {
        _BufferLocker guard(m_mutex);
        m_pushPos = 0, m_popPos = 0, m_count = 0;
    }

private:
    RingBuffer(const RingBuffer<N>&);
    void operator=(const RingBuffer<N>&);

private:
    BYTE          *m_pRingBuffer;    // buffer
    size_type      m_pushPos;         // 新的 push 位置: pushPos=(popPos+count)% N
    size_type      m_popPos;         // 新的 pop 位置
    size_type      m_count;          // 有效字节数
    CriticalSection m_mutex;
};

```



## 12.4 类的继承特性

对象是类的一个实例 (Instance)。如果将对象比做一个个房子, 那么类就是房子的设计图纸。所以面向对象分析和设计的重点是类的分析和设计。在口语中, 人们常把对象和类混为一谈, 我们可以根据上下文判断人们所说的究竟是“类”还是“对象”。

对于 C++ 程序而言, 设计孤立的类是比较容易的, 比较困难的是正确设计基类及其派生类。

如果 A 是基类, B 是 A 的派生类, 那么 B 将继承 A 的数据和函数 (见示例 12-4)。

示例 12-4

---

```
class A {
public:
    void  Func1(void);
    void  Func2(void);
};
class B : public A {
public:
    void  Func3(void);
    void  Func4(void);
};
main()
{
    B  b;
    b.Func1();    // B 从 A 继承了函数 Func1
    b.Func2();    // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

---

这个简单的示例程序说明了这样一个事实: C++ 的“继承”特性可以提高程序的可复用性。类的可复用性还表现在派生类可以调用基类的函数来实现自己的函数, 即使这些函数是虚函数, 这在许多类库程序如 MFC 中很常见, 如示例 12-5 所示。

示例 12-5

---

```
class A {
public:
    void  Func1(void);
    void  Func2(void);
};
class B : public A {
```

---

```
public:
    void Func3(void) {
        A::Func1(); // 先调用基类的函数
        ....      // 自己的实现代码
    }
    void Func4(void) {
        ....      // 自己的实现代码
        A::Func2(); // 后调用基类的函数
    }
};
main()
{
    B b;
    b.Func3();
    b.Func4();
}
```

正因为“继承”太有用、太容易用，才要防止乱用。我们应当给“继承”设立一些规矩。

**【规则 12-1】:** 如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多一些而让 B 继承 A 的功能和属性。不要觉得“不吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。

**【规则 12-2】:** 若在逻辑上 B 是 A 的“一种” (is-a-kind-of)，则允许 B 继承 A 的功能和属性。例如男人 (Man) 是人 (Human) 的一种，男孩 (Boy) 是男人的一种。那么类 Man 可以从类 Human 派生，类 Boy 可以从类 Man 派生 (见示例 12-6)。

示例 12-6

```
class Human {           // Human 是基类
    ...
};
class Man : public Human { // Man 是 Human 的派生类
    ...
};
class Boy : public Man    { // Boy 是 Man 的派生类
    ...
};
```

**【规则 12-2】** 看起来很简单，但是实际应用时可能会有意外。继承的概念在程序世界和现实世界中并不是完全相同的。

例如从生物学角度讲，鸵鸟 (Ostrich) 是鸟 (Bird) 的一种，按理说类 Ostrich 应该可以从类 Bird 派生。但是鸵鸟不能飞，那么 Ostrich::Fly () 是什么东西？ (见示例 12-7)

示例 12-7

---

```

class Bird {
public:
    virtual void Fly();    // 鸟能飞行
    ...
};
class Ostrich : public Bird {    // 鸵鸟是鸟的一种
public:
    virtual void Fly();    // 如何让鸵鸟飞起来?
    ...
};

```

---

再比如，从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），按理说类 Circle 应该可以从类 Ellipse 派生。但是椭圆有长轴和短轴之分，如果圆继承了椭圆的长轴和短轴，岂不是画蛇添足？所以更加严格的继承规则应当是：

**【规则 12-3】** 若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。

C++语言可不管你让谁继承自谁，只要两个类之间具有 public 继承关系，那么它们之间就存在 is-a 关系。编译器理解的 is-a 关系并不总是等价于我们所想要的 is-a 关系，所以我们不要乱用继承。

如果一个事物同时具有另外几个事物的特点（多重特点），在程序世界中该如何来表示它呢？C++有办法，那就是使用多重继承。例如我们家中常使用的沙发床，它可以当沙发来用，展开后就是一张床，所以它既是沙发又是床（双重 is-a），显然我们不能把一个沙发和一张床放在一起（组合）叫做一个沙发床。所以，沙发床这个类应该按如下来定义（见示例 12-8）。

示例 12-8

---

```

class Sofa {
public:
    virtual void Seating(Man&);    // 人可就座
    ...
};
class Bed {
public:
    virtual void Lying(Man&);    // 人可躺下
    ...
};
class Sofabed : public Sofa, public Bed {    // 既是沙发又是床
public:
    virtual void Seating(Man&);

```

---

```
virtual void Lying(Man&);  
...  
};
```

类似的例子还有很多，例如电话传真机、在外国很流行的房车及现在很流行的带有照相功能的手机等，都可以用多重继承来为它们建模。

多重继承另一个比较重要的用途就是在已有接口和实现类的基础上创建自己的接口和实现类，如示例 12-9 和图 12-1 所示。当基于别人开发的类库来开发应用程序时这常常很有用。

示例 12-9

```
class Socket {    // 通用 Socket 接口  
public:  
    virtual ~Socket(){}  
    virtual void Read(BYTE *buff, int length) = 0;           // 读数据  
    virtual void Write(const BYTE *buff, int length) = 0;    // 读数据  
    virtual std::string GetPeerIP() const = 0;                // 获得对端 IP  
    virtual int GetPeerPort() const = 0;                      // 获得对端端口  
    virtual int GetLocalPort() const = 0;                     // 获得本地端口  
    virtual void Close() = 0;  
};  
class SocketImpl : virtual public Socket {                    // 通用 Socket 实现  
public:  
    SocketImpl(){ m_socketId = ::socket(...); }  
    virtual ~SocketImpl() { Close(); }  
    virtual void Read(BYTE *buff, int length) { ... }  
    virtual void Write(const BYTE *buff, int length) { ... }  
    .....  
    virtual void Close() { ::closesocket(m_socketId); }  
private:  
    SOCKET  m_socketId;  
};  
class ClientSocket : virtual public Socket {                  // 客户机 Socket 接口  
public:  
    virtual void Connect(const std::string& svrIp, int listenPort) = 0;  
};  
class ClientSocketImpl : public ClientSocket, private SocketImpl { // 客户机 Socket 实现  
public:  
    virtual void Read(BYTE *buff, int length) { SocketImpl::Read(buff, length); }  
    virtual void Write(const BYTE *buff, int length) { SocketImpl::Write(buff, length); }  
    virtual void Connect(const std::string& svrIp, int listenPort){ ... }  
    .....  
    virtual void Close() { SocketImpl::Close(); }  
};
```

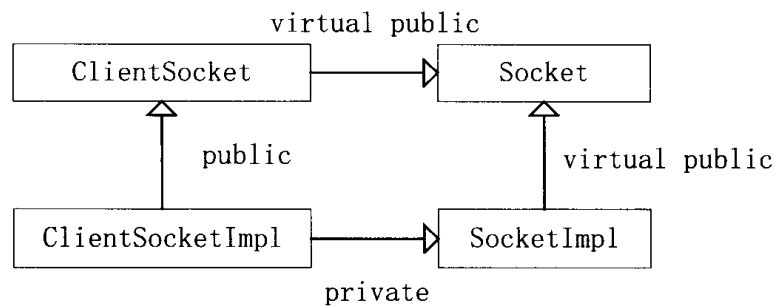


图 12-1 接口继承与实现继承

最后，在 COM 中多重继承被作为定义 COM 对象类的一种方法：一个 COM 对象类可以同时从多个 COM 接口（Interface）继承。这实际上体现了面向对象设计的一个原则——接口隔离原则（ISP）。但是此法不是只在开发 COM 组件时才能用。示例 12-10 展示了这种用法。

示例 12-10

```

class InterfaceA : public IUnknown {                // IID_InterfaceA
public:
    virtual void foo() = 0;
};
class InterfaceB : public IUnknown {                // IID_InterfaceB
public:
    virtual void bar() = 0;
};
class AB : public InterfaceA, public InterfaceB {    // 接口实现类
public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        REFIID riid, void **ppvObject)
    {
        if (riid == IID_IUnknown || riid == IID_InterfaceA) {
            *ppvObject = static_cast<InterfaceA*>(this);
        } else if (riid == IID_InterfaceB) {
            *ppvObject = static_cast<InterfaceB*>(this);
        } else {
            *ppvObject = NULL;
            return E_NOINTERFACE;
        }
        this->AddRef();
        return S_OK;
    }
    virtual ULONG STDMETHODCALLTYPE AddRef( void) {
        return ::InterlockedIncrement(&m_refCount);
    }
    virtual ULONG STDMETHODCALLTYPE Release( void) {
        ULONG result = ::InterlockedDecrement(&m_refCount);
    }
}
    
```

```
        if (result == 0) delete this;
        return result;
    }
    virtual void foo(){ ... }
    virtual void bar(){ ... }
private;
    int m_refCount;                // 接口引用计数器
};
```

在这种用法中，由于各个接口之间一般情况下没有任何关系（除非接口之间有继承关系），所以如果接口的实现类对其客户程序不可见的话，客户程序将无法直接创建实现类的对象，也就无法从一个接口导航到另一个接口。为此，实现类必须提供另外的方法以便客户程序能够得到实现类的对象，比如使用工厂方法模式或者提供一个全局 `Create` 函数，它们创建实现类的对象并返回其中任意一个接口指针，此后客户程序就可以调用该接口的方法，或者调用 `QueryInterface()` 来获得其他接口的指针。

## 12.5 类的组合特性

组合（Composition）也是一种类的复用技术，用于表示类的“整体与部分”关系，例如主机、显示器、键盘、鼠标组合成一台计算机。继承则表示类的“一般与特殊”关系。继承与组合显然不是相似的概念，我们在编程时切勿把继承与组合混为一谈。

**【规则 12-4】** 若在逻辑上 A 是 B 的“一部分”（is-a-part-of），则不允许 B 从 A 派生，而是要用 A 和其他部分组合出 B。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生而成（见示例 12-11）。

示例 12-11

```
class Eye {
public:
    void Look(void);
};
class Nose {
public:
    void Smell(void);
};
class Mouth {
public:
    void Eat(void);
};
```

```

class Ear {
public:
    void Listen(void);
};
// 正确的设计，虽然代码冗长
class Head {
public:
    // 调用传递
    void Look(void){ m_eye.Look(); }
    void Smell(void){ m_nose.Smell(); }
    void Eat(void){ m_mouth.Eat(); }
    void Listen(void){ m_ear.Listen(); }
private:
    Eye      m_eye;
    Nose     m_nose;
    Mouth    m_mouth;
    Ear      m_ear;
};

```

但是如果让 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能。程序如下：

```

class Head : public Eye, public Nose, public Mouth, public Ear
{
    .....
};

```

虽然采用继承方法来实现的 Head 类不仅代码十分简短并且运行正确，但是这种设计方法却是不对的。“运行正确”的程序不见得是高质量的程序，此处就是一个例证。

实际上，类的组合特性具体表现为两种：聚合（has-a）和关联（holds-a）。我们上面举的例子就是聚合，而关联就是类之间的引用，请参看第 8 章关于结构的论述。

许多刚刚接触 C++ 的程序员恨不得在所有的地方都使用继承，然后得意洋洋地宣称已经充分利用了面向对象的好处。这让人想起了电影《没完没了》的精彩对话：

---

一只公鸡使劲地追打一只刚下了蛋的母鸡，你知道为什么吗？  
因为母鸡下了鸭蛋。

---

## 12.6 动态特性

在绝大多数情况下，程序的功能是在编译的时候就确定下来的，我们称为静态特性。反之，如果程序的功能是在运行时刻才确定下来的，则称为动态特性。

动态特性是面向对象语言最强大的功能之一，因为它在语言层面上支持程序的

可扩展性，而可扩展性则是软件设计追求的重要目标之一。

C++虚函数、抽象基类、动态绑定（Dynamic binding）和多态（Polymorphism）构成了出色的动态特性。

## 12.6.1 虚函数

假定几何形状的基础类为 Shape，其派生类有 Circle、Rectangle、Ellipse 等，每个派生类都能够绘制自己所代表的形状。不管派生类的形状如何，我们希望用统一的方式来调用绘制函数，最好是使用 Shape 定义的接口函数 Draw()，并让程序在运行时动态地确定应该使用哪一个派生类的 Draw() 函数。

为了使这种行为可行，我们把基类 Shape 中的函数 Draw() 声明为虚函数，然后在派生类中重新定义 Draw() 使之绘制正确的形状，这种方法叫覆盖（Override）。虚函数的声明方法是在基类的函数原型之前加上关键字 virtual。要注意的是，虚函数并不是实现上述动态特性的唯一手段，也许其他语言还会采用其他方法呢！

一旦类的一个函数被声明为虚函数，那么其派生类的对应函数也自动成为虚函数，这样一级一级传递下去。虽然如此，但是为了提高程序的清晰性，建议你厌其烦地在每一个派生层次中将它显式地声明为虚函数（即加 virtual 关键字），见示例 12-12。

示例 12-12

```
class Shape {
public:
    virtual void Draw(void);    // Draw()为虚函数
};
class Rectangle : public Shape {
public:
    virtual void Draw(void);    // Draw()为虚函数
    ...
}
```

## 12.6.2 抽象基类

当我们把类看成是一种数据类型时，通常会认为该类肯定是要被实例化为一个或多个对象的。但是在很多情况下，定义那些不能实例化出对象的类也是很有用的，这种类就称为抽象类（Abstract Class），而能够被实例化为对象的类称为具体类（Concrete Class）或实现类（Implementation Class）（那些把所有构造函数都声明为 private 函数的类也是不能实例化的类，但不属于我们这里要讨论的话题）。抽象类的唯一目的就是让其派生类继承并实现它的接口方法（Method），因此它通常也被称为抽象基类（Abstract Base Class）。

如果将基类的虚函数声明为纯虚函数，那么该类就被定义为了抽象基类。纯虚函数是在声明时将其“初始化”为 0 的函数，例如：



```
class Shape {                                // Shape 是抽象基类
public:
    virtual void Draw(void) = 0;           // Draw()为纯虚函数
};
```

抽象基类 Shape 的纯虚函数 Draw()根本不知道自己应该怎么绘制出一个“形状”来，具体功能必须由代表具体形状的派生类对应的 Draw()函数来实现。

我们知道，函数名就是函数的地址，将一个函数初始化为 0 意味着函数的地址将为 0，这就是在告诉编译器：不要为该函数编址，从而阻止该类的实例化行为。在 C++中只有虚函数才可以被初始化为 0。

很多良好的面向对象系统中，基类层次结构的顶部通常都是抽象基类，甚至可以有好几层的抽象类。例如几何形状类结构可分为三层（如图 12-2 所示），顶层是抽象基类 Shape，第二层也是抽象基类 Shape2D 和 Shape3D，在第三层才是可以被实例化为对象的具体类，如二维形状类 Circle、Rectangle 和 Ellipse，三维形状类 Cube、Cylinder 和 Sphere。

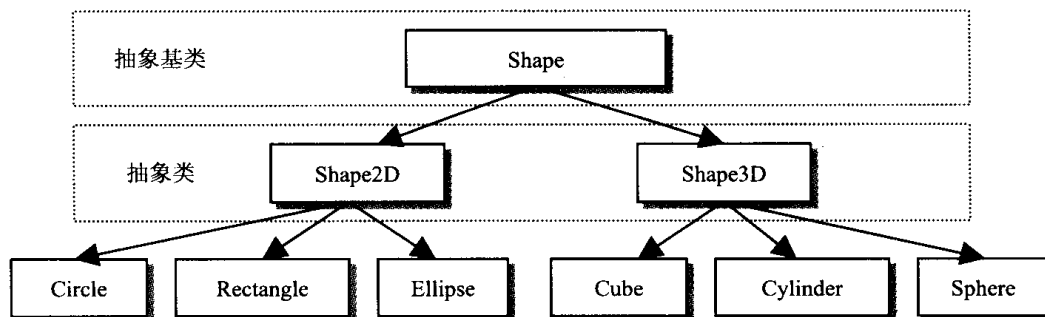


图 12-2 几何形状类层次结构

抽象基类的主要用途是“接口与实现分离”：不仅要把数据成员（信息）隐藏起来，而且还要把实现完全隐藏起来，只留一些接口给外部调用。即使将来实现改变了，接口仍然可以保持不变。

一般的信息隐藏就是把类的所有数据成员声明为 private 或者 protected 的，并提供相应的 get 和 set 函数来访问对象的数据。抽象基类则更进一步，它把数据和函数实现都隐藏在实现类中，而在抽象基类中提供丰富的接口函数供调用，这些函数都是 public 的纯虚函数。这样的抽象基类叫做接口类（Interface）。

这种彻底的封装思想充分利用了面向对象的动态绑定技术，被 COM、CORBA 等体系结构普遍采用。在使用面向对象技术构造的软件系统中，这样的接口技术很适合于定义各个模块之间的接口，更详细的讨论请参考 COM、CORBA 等技术书籍。

示例 12-13 中的 IRectangle 为接口类，而 RectangleImpl 则是一个实现类。

示例 12-13

```
class IRectangle {
public:
    virtual ~IRectangle(){}
    virtual float GetLength() const = 0;
```

```

virtual void SetLength( float newLength) = 0 ;
virtual float GetWidth() const = 0 ;
virtual void SetWidth(float newWidth) = 0 ;
virtual RGB GetColor() const = 0 ;           // RGB : unsigned long
virtual void SetColor(RGB newColor) = 0 ;
virtual float CalculateArea() const = 0 ;
virtual void Draw() = 0 ;
static IRectangle* __stdcall CreateRectangle(); //入口函数
void Destroy(){ delete this; }

};

class RectangleImpl : public IRectangle {
public:
    RectangleImpl() : m_length(1), m_width(1), m_color(0x00FFEC4D){}
    virtual ~RectangleImpl(){}
    virtual float  GetLength() const { return m_length; }
    virtual void   SetLength(float newLength) { m_length = newLength; }
    virtual float  GetWidth() const { return m_width; }
    virtual void   SetWidth(float newWidth) { m_width = newWidth; }
    virtual RGB    GetColor() const { return m_color; }
    virtual void   SetColor(RGB newColor) { m_color = newColor; }
    virtual float  CalculateArea() const { return m_length * m_width; }
    virtual void   Draw() { cout << "RectangleImpl::Draw()" << endl; }

private:
    float  m_length;
    float  m_width;
    RGB    m_color;
};

```

由于抽象基类不能实例化，并且实现类被完全隐藏，所以必须以其他的途径使用户能够获得实现类的对象，比如提供入口函数来动态创建实现类的对象。入口函数可以是全局函数，但最好是静态成员函数，见示例 12-14。

示例 12-14

```

IRectangle* __stdcall IRectangle::CreateRectangle()
{
    // 返回的是 RectangleImpl 对象的地址，接受该地址的则是接口指针
    return new(nothrow) RectangleImpl;
}

void main(void)
{
    IRectangle *pRect = IRectangle::CreateRectangle();
    if(pRect == NULL) exit(-1);
    pRect->SetLength(100.5);
    pRect->SetWidth(35);
    pRect->SetColor(0x00FFFFFF);
    cout << "Length : " << pRect->GetLength() << endl;
}

```

```

    cout << "Width : " << pRect->GetWidth() << endl;
    cout << "Area : " << pRect->CalculateArea() << endl;
    cout << "Color : " << pRect->GetColor() << endl;
    pRect->Draw();
    pRect->Destroy();
    pRect = NULL;
}

```

输出结果如下:

```

Length : 100.5
Width : 35
Area : 3517.5
Color : 16777215
RectangleImpl::Draw()

```

### 12.6.3 动态绑定

如果将基类 Shape 的函数 Draw() 声明为 virtual 的, 然后用指向派生类对象的基类指针或引用来调用 Draw(), 那么程序会在运行时选择该派生类的 Draw() 函数而不是 Shape::Draw(), 这种特性称为运行时绑定 (或动态绑定、晚绑定), 见示例 12-15。

示例 12-15

```

Shape *pShape = NULL;
Circle aCircle;
Cube aCube;
Sphere aSphere;
pShape = &aCircle;
pShape->Draw();           // 调用 Circle::Draw(), 绘制一个 circle
(*pShape).Draw();        // 调用 Circle::Draw(), 绘制一个 circle
pShape = &aCube;
pShape->Draw();           // 调用 Cube::Draw(), 绘制一个 cube
(*pShape).Draw();        // 调用 Cube::Draw(), 绘制一个 cube
pShape = &aSphere;
pShape->Draw();           // 调用 Sphere::Draw(), 绘制一个 Sphere
(*pShape).Draw();        // 调用 Sphere::Draw(), 绘制一个 Sphere

```

动态绑定可以使独立软件供应商 (ISV) 在不透露技术秘密的情况下发行软件包, 即只发行头文件和二进制文件, 不必公开源代码 (实现代码)。软件开发 (软件包的用户) 可以利用继承机制从 ISV 提供的类库中派生出新的类。和 ISV 类库一起运行的软件也能够和新的派生类一起运行, 并且能够通过动态绑定使用新派生类改写的虚函数。

那么 C++ 的动态绑定技术是如何实现的呢? 程序之所以能够在运行时选择正确的虚函数, 必定隐藏了一段运行时进行对象类型判断或是函数寻址的代码。事实正是如此!

每一个具有虚函数的类都叫做多态类, 这个虚函数或者是从基类继承来的, 或

者是自己新增加的。C++编译器必须为每一个多态类至少创建一个虚函数表(vtable)，它其实就是一个函数指针数组，其中存放着这个类所有的虚函数的地址及该类的类型信息，其中也包括那些继承但未改写(Overrides)的虚函数。

没有必要在每一个对象的内存空间中都保留一份类型信息，因为同类对象的类型信息完全相同，所以只需要在该类的 vtable 中保留一份就可以了。每一个多态对象都有一个隐含的指针成员，它指向所属类型的 vtable，这就是 vptr。显然，类型信息应该保存在 vtable 中固定的位置上，否则就无法实现统一的检索操作。

然而，实际上虚函数的动态绑定并没有使用到这个类型信息，而是采用了运行时函数寻址技术，该类型信息主要用在 RTTI 技术上。你可以暂时先跳到 12.7.1 节看一下多态对象的内存布局，这样有助于理解上面这段话。

程序中通过基类指针或引用对虚函数的调用语句都会被编译器改写成下面这种形式：

```
(*p->_vptr[slotNum])(p, arg-list); // 指针当作数组来用，最后改写为指针运算
```

其中，p 是基类型指针，vptr 是 p 指向的对象的隐含指针，而 slotNum 就是调用的虚函数在 vtable 中的编号，这个数组元素的索引号在编译时就确定了下来，并且不会随着派生层次的增加而改变。

由于这种运行时才进行的函数寻址，以及各个虚函数的参数列表(signature)都不尽相同，编译时根本无法对一个具体的虚函数调用执行静态的参数类型检查。那么 C++编译器是不是把虚函数调用语句的参数类型检查推迟到了运行时进行了呢？

标准 C++实现没有采取这种策略，而是采用了最严格也是最简单和最有效的规则，那就是：派生类定义中的名字(对象或函数名)将义无反顾地遮蔽(即隐藏)掉基类中任何同名的对象或函数。基于这样的规则，如果派生类定义了一个与其基类的虚函数同名的虚函数，但是参数列表有所不同，那么这就不会被编译器认为是对基类虚函数的改写(Overrides)，而是隐藏，所以也不可能发生运行时绑定。相反，要想达成运行时绑定的效果，派生类和基类中同名的虚函数必须具有相同的原型，也即相同的 Signature(返回类型可以不同，这是 C++的一个特征——协变)。如示例 12-16 中，声明// (1) 不是对基类 Draw() 函数的改写，而是新增的 virtual 函数，声明// (2) 才是对基类 Draw() 函数的改写。

示例 12-16

```
class IRectangle {
public:
    virtual ~IRectangle(){}
    ...
    virtual void Draw() = 0;
    ...
};
class RectangleImpl : public IRectangle {
public:
    RectangleImpl() : m_length(1), m_width(1), m_color(0x00FFEC4D){}
```

```

virtual ~RectangleImpl(){}
virtual void Draw(int scale) { cout << "RectangleImpl::Draw(int)" << endl; } // (1)
virtual void Draw() { cout << "RectangleImpl::Draw()" << endl; } // (2)
private:
    float m_length;
    float m_width;
    RGB m_color;
};
void main(void)
{
    IRectangle *pRect = IRectangle::CreateRectangle();
    if (pRect == NULL) exit(-1);
    pRect->Draw(); // 由于 pRect 的静态类型为 IRectangle*, 所以使用 IRectangle::
                  // Draw()执行静态类型检查, 但由于 pRect 指向的对象实际是
                  // RectangleImpl 对象, 因此将绑定到 RectangleImpl::Draw()!
    pRect->Draw(200); // 同理, 由于 IRectangle 类并没有此类原型的函数, 因此拒绝
                     // 编译, 除非 pRect 的类型为 RectangleImpl*. 此 Draw()非彼
                     // Draw()!

    pRect->Destroy();
    pRect = NULL;
}

```

此外, 在示例 12-16 中, 如果 RectangleImpl 不重定义 Draw()函数, 那么下面的代码:

```

RectangleImpl *pRectImpl = new RectangleImpl;
pRectImpl->Draw(); // (3)
pRectImpl->Draw(200); // OK!

```

将无法编译, 因为//(3)处调用的 Draw()是基类的函数, 它被 RectangleImpl 中的同名函数 Draw(int)隐藏了, 看不见了!

关于成员函数的重载、覆盖(改写)和隐藏的规则请参考本书 14.2 节。

C++的虚拟机制如此复杂, 如果你想详细地了解其工作原理的内幕, 请参考《Inside The C++ Object Model》或《Thinking In C++》, 或者也可以看一看《The Design and Evolution of C++》; 如果想详细地了解 RTTI 的内幕, 请看本书第 15 章、《Thinking In C++》或者《The C++ Programming Language》。

## 12.6.4 运行时多态

当许多派生类因为继承了共同的基类而建立 is-a 关系时, 每一个派生类的对象都可以被当成基类的对象来使用, 这些派生类对象能对同一函数调用做出不同的反应, 这就是运行时多态, 见示例 12-17。

示例 12-17

```
void Draw(Shape *pShape)    // 多态函数
{
    pShape->Draw();          // 或者: (*pShape).Draw();
}

main()
{
    Circle  aCircle;
    Cube    aCube;
    Sphere  aSphere;
    ::Draw(&aCircle);        // 绘制一个 circle
    ::Draw(&aCube);          // 绘制一个 cube
    ::Draw(&aSphere);        // 绘制一个 Sphere
}
```

实际上, C++支持运行时多态特性的手段有两种, 这里讲的虚函数机制是其中主要的一种, 另一种是 RTTI, 将在本书第 15 章讲解。我们将 C++支持多态的方法总结如下:

- ◇ 经过隐含的转型操作, 令一个 **public** 多态基类的指针或者引用指向它的一个派生类的对象 (见示例 12-14 和示例 12-15);
- ◇ 通过这个指针或引用调用基类的虚函数, 包括通过指针的反引用调用虚函数, 因为反引用一个指针将返回所指对象的引用 (见示例 12-15 和示例 12-17, 以及本书 16.12、16.13 和 16.14 节);
- ◇ 使用 `dynamic_cast<>` 和 `typeid` 运算符, 参见本书第 15 章。

综合 C++的“虚函数”和“多态”, 有如下突出优点:

- ◇ 应用程序不必为每一个派生类编写功能调用, 只需要对基类的虚函数进行改写或扩展即可。这一招叫“以不变应万变”, 可以大大提高程序的可复用性和可扩展性;
- ◇ 派生类的功能可以被基类指针引用, 这叫向后兼容。以前写的程序可以被将来写的程序调用, 这不足为奇; 但是将来写的程序可以被以前写的程序调用那可了不起, 这正是动态特性的妙处!

现在的许多分布式中间件比如 CORBA、COM 等充分利用了虚函数动态绑定这一优点, 把底层的网络通信功能全部封装起来, 而在上层预留一些回调接口由用户来实现, 这些接口往往表现为虚函数, 它们的调用则是由底层服务来安排的 (一般是由远程网络请求触发的)。用户不仅可以直接实现这些接口, 还可以进一步派生出自己的实现类。

## 12.6.5 多态数组

用惯了 C 数组的人在刚开始学习 C++虚函数和多态特性时一般都会产生这样的念头: 如果能够在数组里放置一些多态对象的话, 就可以通过一致的接口来动态地

调用它们自定义的虚函数实现了。其实这个想法是很好的，但是在具体操作的过程中就会产生一些没有意识到的问题。

假设我们定义了如图 12-3 所示的类层次结构，类定义见示例 12-18。

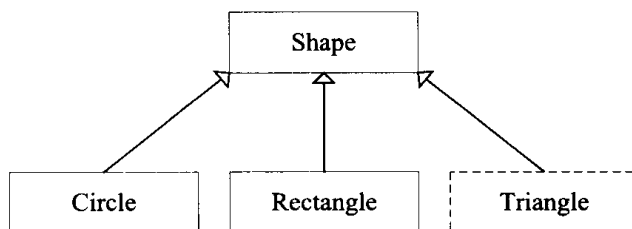


图 12-3 Shape 类层次结构

示例 12-18

```

// two-dimension point
struct Point {
    Point(int x, int y) : m_x(x), m_y(y){ }
    int    m_x;    // horizontal ordinate
    int    m_y;    // vertical ordinate
};

ostream& operator<<(ostream& os, const Point& p)
{
    cerr << "\n\tHorizontalOrdinate = " << p.m_x \
        << "\n\tVerticalOrdinate   = " << p.m_y << "\n\n";
    return os;
}

class Shape {
    Point m_origin;    // origin point
public:
    Shape() : m_origin(0, 0){ }
    explicit Shape(const Point& origin) : m_origin(origin){ }
    virtual ~Shape() { cerr << "Shape::~~Shape()" << endl; }
    Point GetOrigin() const{ return m_origin; }
    virtual void Draw() const { // draw a point only
        cerr << "Shape::Draw()" << endl;
        cerr << "Origin Point : " << m_origin << endl;
    }
};

class Circle : public Shape {
    int m_radius; // radius
public:
    Circle() : m_radius(1){ }
    Circle(const Point& origin, int radius) : Shape(origin), m_radius(radius){ }
    virtual ~Circle() { cerr << "Circle::~~Circle()" << endl; }
    int GetRadius() const{ return m_radius; }
    virtual void Draw() const { // draw circle

```

```
        cerr << "Circle::Draw()" << endl;
        cerr << "Origin Point : " << GetOrigin() \
            << "Radius = " << m_radius << endl << endl;
    }
};

class Rectangle : public Shape {
    Point  m_rightBottom; // right-bottom point, m_origin as left-top point
public:
    Rectangle() : m_rightBottom(1, 1){ }
    Rectangle(const Point& leftTop, const Point& rightBottom)
        : Shape(leftTop, m_rightBottom(rightBottom)){ }
    virtual ~Rectangle() { cerr << "Rectangle::~~Rectangle()" << endl; }
    Point GetLeftTop() const{ return GetOrigin(); }
    Point GetRightBottom() const{ return m_rightBottom; }
    virtual void Draw() const { // draw rectangle
        cerr << " Rectangle::Draw()" << endl;
        cerr << "Left-top Point : " << GetLeftTop() \
            << "Right-bottom Point : " << m_rightBottom << endl;
    }
};

void DrawShapes(const Shape shapes[], int numOfShapes)
{
    for (int i = 0; i < numOfShapes; ++i)
        shapes[i].Draw();
}
```

如果直接使用多态对象数组会存在什么问题呢？我们看下面的例子，示例 12-19 在基类型对象数组中存放派生类对象。

示例 12-19

```
Shape    a(Point(1, 1));
Circle   b(Point(2, 2), 5);
Rectangle c(Point(3, 3), Point(4, 4));
Shape    myShapes[3];
myShapes[0] = a;
myShapes[1] = b;
myShapes[2] = c;
for (int i = 0; i < 3; ++i)
    myShapes[i].Draw();
```

你可能期望它会依次调用 `Shape::Draw()`、`Circle::Draw()` 和 `Rectangle::Draw()`，但是实际上它不幸地失败了！为什么呢？

首先，数组 `myShapes` 的元素类型是 `Shape`，而不是 `Shape&` 或者 `Shape*`，因此它会按照 `Shape` 的大小来分配内存空间。因此通过 `Shape` 对象调用 `Draw()` 根本不会进行动态绑定，三次调用的都将是 `Shape::Draw()`；



其次,把派生类对象赋值给 myShapes 数组元素时发生了对象切割,因为派生对象比基类对象大(一般情况下都是如此),数组里保存的并非是每次指派给它的对象,而都是 Shape 对象。如果 Shape 是一个抽象类,那么根本就不能直接定义 Shape 类型的数组;

最后,循环体内 myShapes[i] 会转化为一个指针算术运算:

$$\text{myShapes}[i] \Leftrightarrow *(\text{myShapes} + i)$$

而 (myShapes + i) 的值实际上等于 ((BYTE\*)myShapes + (i \* sizeof(Shape))), 它始终指向的是 Shape 对象的起始地址,而不是派生类对象。下面是该例运行结果,该结果充分说明了这一事实。

---

```
Shape::Draw()
Origin Point :
{      HorizontalOrdinate = 1
      VerticalOrdinate   = 1
}
Shape::Draw()
Origin Point :
{      HorizontalOrdinate = 2
      VerticalOrdinate   = 2
}
Shape::Draw()
Origin Point :
{      HorizontalOrdinate = 3
      VerticalOrdinate   = 3
}
```

---

在示例 12-20 中,语句 //(1) 能够正常工作,但结果和示例 12-19 一样。语句 //(2) 会导致程序悲惨地失败,虽然能够顺畅地编译。为什么会如此呢?

示例 12-20

---

```
DrawShapes(myShapes, 3);           // (1)
Rectangle  d(Point(3, 3), Point(4, 4));
Rectangle  e(Point(5, 5), Point(6, 6));
Rectangle  f(Point(7, 7), Point(8, 8));
Rectangle  myRects[3];
myRects[0] = d;
myRects[1] = e;
myRects[2] = f;
DrawShapes(myRects, 3);           // (2)
```

---

函数 void DrawShapes(const Shape shapes[], int numOfShapes); 接受 Shape 数组为输入参数,而一维数组等价于指向元素的指针类型,即:

$$\text{Shape shapes[]} \Leftrightarrow \text{Shape} * \text{const shapes}$$

因此，该函数等价于：

```
void DrawShapes(const Shape * const shapes, int numOfShapes);
```

同样道理，`Rectangle myRects[3]`则等价于 `Rectangle * const myRects`，并且在作为参数传递时会执行这种退化，由于 `Rectangle` 为 `Shape` 的 `public` 派生类 (Is-a)，因此调用语句(2)是合法的，即传入的实际上是数组 `myRects` 的第一个元素对象的地址，而不是整个数组的拷贝。但是在调用语句(2)的内部发生了什么事情呢？

在循环：

```
for (int i = 0; i < numOfShapes; ++i)
    shapes[i].Draw();
```

内，`shapes[i]`转化为指针算术运算`*(shapes + i)`，而在函数 `DrawShapes()`内部编译器并不知道 `shapes` 指向的实际是 `Rectangle` 对象数组，而会认为 `shapes` 里面都是 `Shape` 对象，因为 `shapes` 的静态类型为 `Shape`。因此`(shapes + i)`实际上就被转换成了`((BYTE*)shapes + (i * sizeof(Shape)))`，然而这个指针值此时指向的也并非总是 `Shape` 对象（当 `i == 0` 时指向的是 `Shape` 对象）。结果可想而知：除第一次循环能够正确执行外，剩余的循环都失败了（实际上第二个循环就已经导致程序失败了），因为它们的地址完全错位了。从图 12-4 可以看出这一事实。

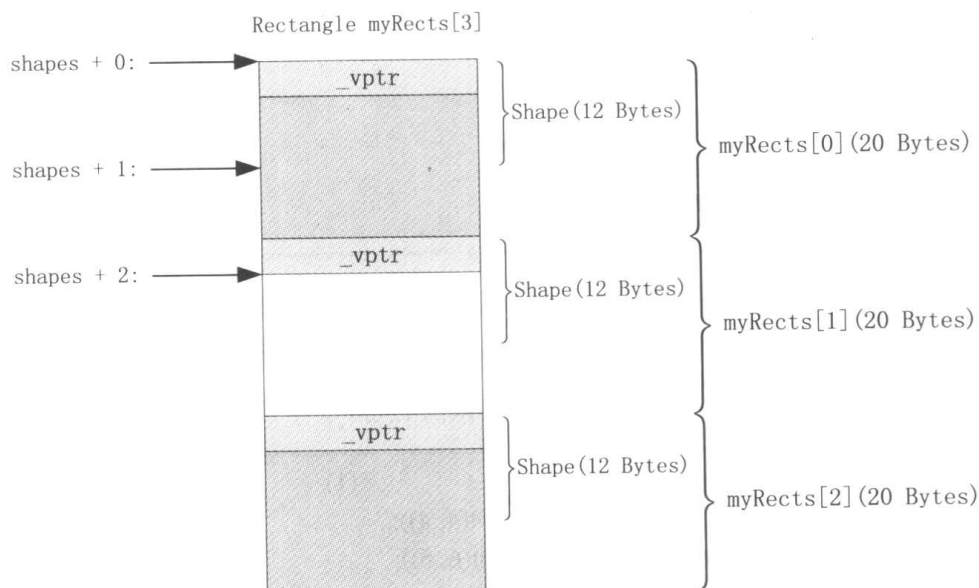


图 12-4 Rectangle 数组的内存映像

实际上，有一个解决办法可以让调用语句(2)正确地运行，那就是借助 `RTTI`。例如我们可以这样重新编写函数 `DrawShapes()`，见示例 12-21。

示例 12-21

```
void DrawShapes(const Shape shapes[], int numOfShapes)
{
    for (int i = 0; i < numOfShapes; ++i) {
```

```

Shape *p = (Shape*)shapes;
if (typeid(*p) == typeid(Shape)) {
    p->Draw();
    p++;
} else if (typeid(*p) == typeid(Circle)) {
    Circle *q = (Circle*)p;
    q->Draw();
    p = ++q;
} else if (typeid(*p) == typeid(Rectangle)) {
    Rectangle *r = (Rectangle*)p;
    r->Draw();
    p = ++r;
} else
    ;
}
}

```

现在可以如你期望的那样运行了！但是这显然不是什么好办法，C++编译器也不会这样来实现以帮助你正确地使用所谓的“多态数组”。主要是因为依赖 RTTI 会导致严重的效率低下；其次，如果打算从 `Shape` 派生新的类，比如 `Triangle`，那么必须修改 `DrawShapes()` 的实现，否则无法应用于这些新的派生类，因此丧失了通用性和可扩展性。

另外，删除一个多态对象数组也需要多加注意，见示例 12-22，通过基类指针删除派生类对象数组。

示例 12-22

```

Shape *pShapes = new Circle[3];
...          // using pShapes
delete []pShapes;

```

这里调用的是运算符 `delete` 的数组版本而不是单元素版本，因此 `delete[]` 的实现里面照样包含指针的算术运算，并且需要依次调用每一个元素的析构函数，然后释放掉它们的内存。编译器将 `delete []pShapes` 翻译成类似下面的语句：

```

for (int i = 从某处取出 pShapes 指向的数组的元素个数 - 1; i >= 0; --i)
{
    Shape *q = pShapes + i;    // 从后向前释放
    q->~Shape();                // pShapes 的静态类型为 Shape*
    _delete(q);                // 释放 q 指向的内存块儿;
}

```

显然，`q` 一开始指向的对象根本就不是一个真正的 `Shape` 对象，当然也不是一个 `Circle` 对象，因此根本没有调用到我们期望的析构函数 `~Circle()`，就可能造成资源泄漏（如果 `Circle` 的析构函数中会释放某种资源的话）；其次，如果派生类对象比基类

对象大, 该循环必然造成内存泄漏。通过图 12-4 可证实这一点。

此时, 正确的删除方法应该是:

```
for (int i = 2; i >= 0; --i)      // 以与创建相反的顺序删除
{
    Shape *q = (Circle*)pShapes + i;
    delete q; // 动态绑定析构函数
}
```

或者直接将 pShapes 转换为 Circle\* 然后用 delete[] 来删除。

- 【提示 12-1】:**
- (1) 通过基类指针删除一个由派生类对象组成的数组, 结果未定义 (C++ 标准);
  - (2) 多态和指针算术运算不能混合运用, 而数组操作几乎总是会涉及到指针运算, 因此多态和数组不应该混合运用 (Scott Meyers)。

针对上述问题, 有没有更好的解决办法呢? 办法是有的, 那就是: **不要在数组中直接存放多态对象, 而是换之以基类指针或者基类的智能指针。**直接使用基类的指针可能需要自己负责删除它们指向的对象, 但是使用智能指针则不需要操这份心! 使用普通指针数组见示例 12-23, 使用智能指针数组见示例 12-24。关于 SmartPtr 参见本书第 16.14 节。

示例 12-23

```
Shape *p = new Shape(Point(1, 1));
Shape *q = new Circle(Point(2, 2), 5);
Shape *r = new Rectangle(Point(3, 3), Point(4, 4));
Shape *shapes[3];
shapes[0] = p;
shapes[1] = q;
shapes[2] = r;
for (int i = 0; i < 3; ++i)
    shapes[i]->Draw();
for (int j = 0; j < 3; ++j)
    delete shapes[j]; // 释放内存
```

运行结果如下:

```
Shape::Draw()
Origin Point :
{   HorizontalOrdinate = 1
    VerticalOrdinate   = 1
}
Circle::Draw()
Origin Point :
{   HorizontalOrdinate = 2
    VerticalOrdinate   = 2
```

```

    }
    Radius = 5
    Rectangle::Draw()
    Left-top Point :
    {
        HorizontalOrdinate = 3
        VerticalOrdinate    = 3
    }
    Right-bottom Point :
    {
        HorizontalOrdinate = 4
        VerticalOrdinate    = 4
    }
}

```

示例 12-24

```

typedef SmartPtr<Shape>  ShapeSmartPtr;
ShapeSmartPtr shapes[3];
ShapeSmartPtr  p(new Shape(Point(1, 1)));
ShapeSmartPtr  q(new Circle(Point(2, 2), 5));
ShapeSmartPtr  r(new Rectangle(Point(3, 3), Point(4, 4)));
shapes[0] = p;
shapes[1] = q;
shapes[2] = r;
for (int i = 0; i < 3; ++i)
    shapes[i]->Draw();

```

示例 12-24 的运行结果与示例 12-23 相同。

如果确实需要使用多态数组，请使用 STL 容器配合普通指针或者智能指针。

## 12.7 C++对象模型

什么是 C++ 的对象模型？要想理解它，必须把 C++ 语言层面的面向对象特征和概念与对象的底层实现技术结合起来进行分析。前者如构造函数、拷贝和赋值函数、析构函数、静态成员、虚函数、继承、组合、动态创建对象、RTTI 等；后者则是语言实现隐藏起来的细节，如对象的内存映像、vtable 的构造、vptr 的插入和初始化时机、构造和析构函数的自动调用时机、对象的构造和析构次序、临时对象的创建和销毁、RTTI 的底层实现技术，等等。

对象模型涉及的内容非常之多，构成了一套完整的语言实现体系。我们不可能讲述其中的每一个主题，仅进行粗浅的介绍，以起到抛砖引玉的作用。如果读者朋友们想详细而深入地了解对象模型的知识，可参阅 Lippman 的《Inside The C++ Object Model》这本书。

### 12.7.1 对象的内存映像

了解对象模型首先要知道对象在内存中的布局，也就是对象在内存中是如何存

放和表示的。我们在本书 4.2 节和 8.1.4 节分别讲解过基本类型和复合类型对象的内存布局，这里我们将简单阐述一下 C++ 类的内存映像。

我们先考虑示例 12-25 中的简单的非多态类 `Rectangle` 的对象的内存映像（如图 12-5 所示）。

示例 12-25

```
class Rectangle {
public:
    Rectangle() : m_length(1), m_width(1) { ... }
    ~Rectangle(){...}
    float GetLength() const { return m_length; }
    void SetLength(float length) { m_length = length; }
    float GetWidth() const { return m_width; }
    void SetWidth(float width) { m_width = width; }
    void Draw(){...}
    static unsigned int GetCount(){ return m_count; }
protected:
    Rectangle(const Rectangle& copy){...}
    Rectangle& operator=(const Rectangle& assign){...}
private:
    float m_length;           // 长
    float m_width;           // 宽
    static unsigned int m_count; // 对象计数
};
```

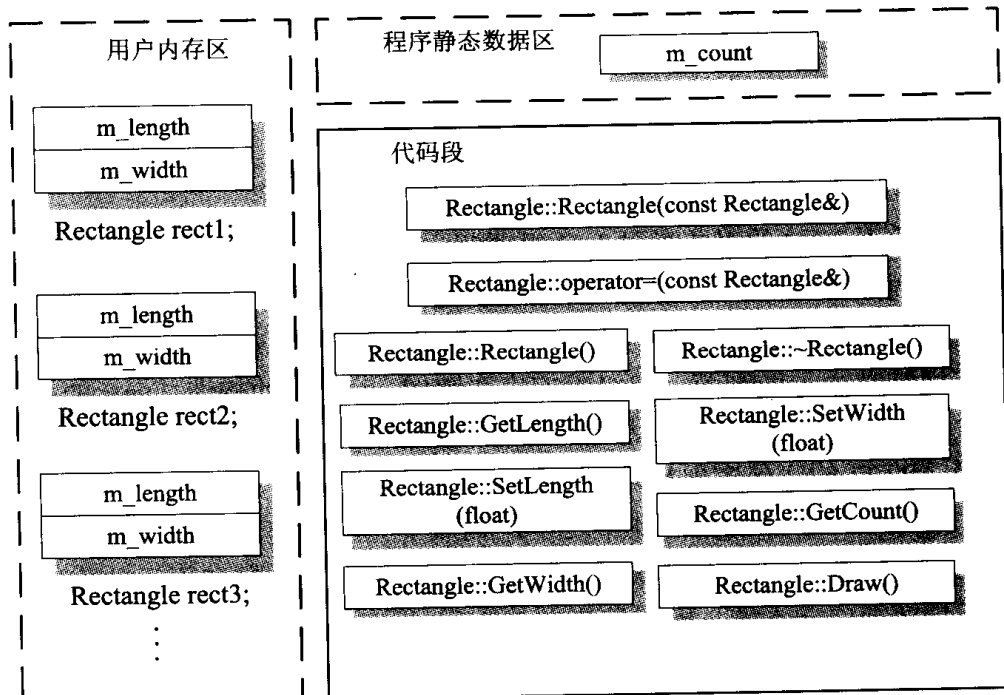


图 12-5 `Rectangle` 对象的内存映像

`Rectangle` 这种基本的 C++ 对象模型有下述几个规则：

- ✧ 非静态数据成员被放在每一个对象体内作为对象专有的数据成员;
- ✧ 静态数据成员被提取出来放在程序的静态数据区内为该类所有对象共享, 因此仅存在一份;
- ✧ 静态和非静态成员函数最终都被提取出来放在程序的代码段中并为该类的所有对象共享, 因此每一个成员函数也只存在一份代码实体;
- ✧ 类内嵌套定义的各种类型 (typedef、class、struct、union、enum 等) 与放在类外面定义的类型除了作用域不同外没有本质区别。

因此, 构成对象本身的只有数据, 任何成员函数都不隶属于任何一个对象, 非静态成员函数与对象的关系就是绑定, 绑定的中介就是 this 指针。

成员函数为该类所有对象共享, 不仅是出于简化语言实现、节省存储的目的, 而且是为了使同类对象具有一致的行为。虽然同类对象的行为一致, 但是操作不同对象的数据成员, 就会使各个对象具有不同的状态。

现在我们假设 Rectangle 派生自抽象基类 Shape, Shape 有一个属性 m\_color, 并且把 Draw() 移到 Shape 中作为纯虚函数, 见示例 12-26。

示例 12-26

---

```
class Shape {
public:
    Shape() : m_color(0){ }
    virtual ~Shape(){ }
    float GetColor() const { return m_color; }
    void SetColor(float color) { m_color = color; }
    virtual void Draw() = 0;
private:
    float m_color;    // 颜色
};

class Rectangle : public Shape {
public:
    .....
private:
    .....
};
```

---

增加了继承和虚函数的类的对象模型变得更加复杂, 规则如下:

- ✧ 派生类继承基类的非静态数据成员, 并作为自己对象的专用数据成员;
- ✧ 派生类继承基类的非静态成员函数并可以像自己的成员函数一样访问;
- ✧ 为每一个多态类创建一个虚函数指针数组 vtable, 该类的所有虚函数 (继承自基类或者新增的) 的地址都保存在这张表里;
- ✧ 多态类的每一个对象 (如果有) 中安插一个指针成员 vptr, 其类型为指向函数指针的指针, 它总是指向所属类的 vtable, 也就是说: vptr 当前所在的对象是什么类型的, 那么它就指向这个类型的 vtable。vptr 是 C++ 对象的隐含数据成员之一 (实际上它被安插在多态类的定义中);

- ✧ 如果基类已经插入了 `vptr`，则派生类将继承和重用该 `vptr`；
- ✧ 如果派生类是从多个基类继承或者有多个继承分支（从所有根类开始算起），而其中若干个继承分支上出现了多态类，则派生类将从这些分支中的每个分支上继承一个 `vptr`，编译器也将为它生成多个 `vtable`，有几个 `vptr` 就生成几个 `vtable`（每个 `vptr` 分别指向其中一个），分别与它的多态基类对应；
- ✧ `vptr` 在派生类对象中的相对位置不会随着继承层次的逐渐加深而改变，并且现在的编译器一般都将 `vptr` 放在所有数据成员的最前面；
- ✧ 为了支持 RTTI，为每一个多态类创建一个 `type_info` 对象，并把其地址保存在 `vtable` 中的固定位置（一般为第一个位置）（这一条取决于具体编译器的实现技术，标准并没有规定）。

现在的 `Rectangle` 的对象模型如图 12-6 所示。

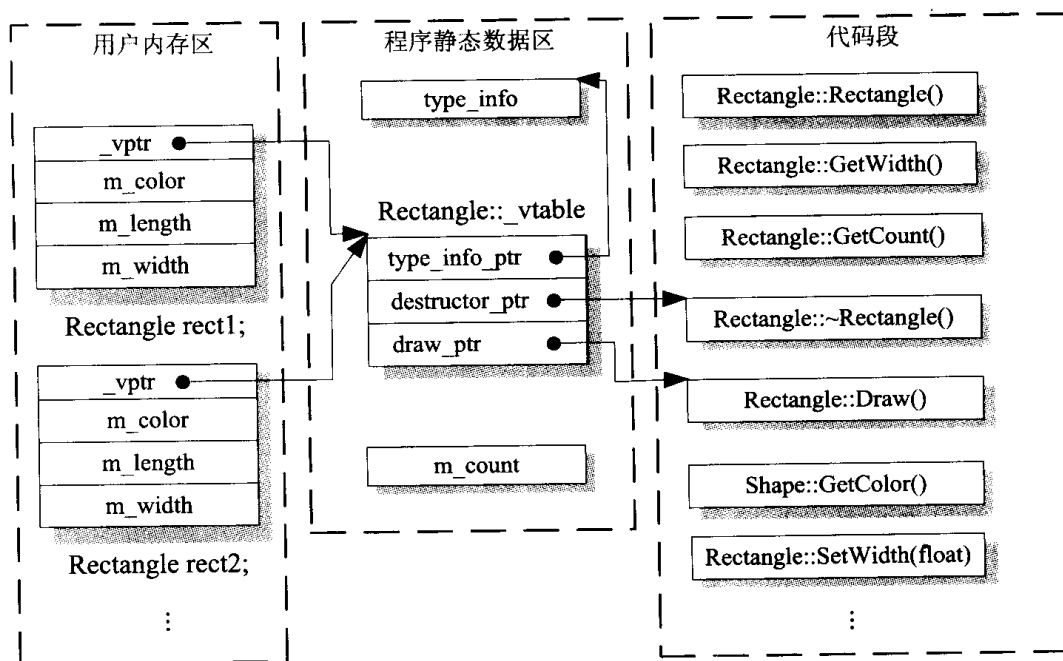


图 12-6 加上继承和多态特性后的 `Rectangle` 对象模型

该模型有如下特点：

- ✧ 从一个派生类对象入手，可以直接访问到基类的数据成员，因为基类的数据成员被直接嵌入到了派生类对象中（保持基类子对象的完整性）；
- ✧ 不论派生层次有多深，派生类对象访问基类对象的数据成员和成员函数时，与访问自己的数据成员和成员函数没有任何效率差异；
- ✧ 由于派生类数据成员和基类数据成员的这种紧密关系，当基类定义发生改变时（例如增加或删除成员）派生类必须重新编译后才能正确使用；
- ✧ 派生类新增数据成员和继承来的基类数据成员按照对象的构造顺序来组合，并且每层派生的新增数据成员要么统一放在基类子对象的前面，要么统一放在后面；
- ✧ 只有虚函数访问需要经过 `vptr` 的间接寻址，增加了一层间接性，因此带来了一些额外的运行时开销。



关于 vtable 和 vptr 还有两个问题需要解释，一个就是它们的类型问题，另一个就是它们的初始化问题。

我们知道，从语言层面讲，只有相同类型的对象才能被放入同一个数组中（或者一个容器中），也就是说，你无法声明一个可以同时包含多种不同类型对象的数组，对函数指针数组来说也是如此。我们在本书 7.4 节已经讲解过函数指针数组并列举了一两个例子，但是在那里并没有阐述关于 vtable 的更多细节。

vtable 也是一个函数指针数组，按理说也只能存放类型相同的函数指针。可是，一个 class 中可能有各种各样的虚函数，它们的原型不可能都一样，因此也不可能都是一种函数指针类型，不同 class 的虚函数那就更不可能了。那怎么定义这个虚函数表呢？

这里我想到了 MFC 实现的消息映射（Message Map）和传递机制中所使用的几个类型和结构，以及那个消息分派（Message Dispatch）的源头函数 CWnd::OnWndMsg()，这几个类型和结构分别为：DECLARE\_MESSAGE\_MAP、BEGIN\_MESSAGE\_MAP（和 END\_MESSAGE\_MAP）、AFX\_MSGMAP\_ENTRY、AFX\_PMSG（以及 AFX\_PMSGT 和 AFX\_PMSGW）、MessageMapFunctions、AfxSig，以及各种 ON\_WM\_XXX、ON\_COMMAND\_XXX、ON\_NOTIFY\_XXX 宏等。它们共同撑起了消息映射、分派和传递的整个流程。部分定义如下：

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;           // windows message id
    UINT nCode;              // control code or WM_NOTIFY code
    UINT nID;                // control ID (or 0 for windows messages)
    UINT nLastID;            // used for entries specifying a range of control id's
    UINT nSig;               // signature type (action) or pointer to message #
    AFX_PMSG pfn;            // routine to call (or special value)
};
union MessageMapFunctions // 用于成员函数指针自动转型
{
    AFX_PMSG pfn;          // 通用的成员函数指针

    // 处理 WM_COMMAND 和 WM_NOTIFY 消息的特殊成员函数指针
    void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND)();
    BOOL (AFX_MSG_CALL CCmdTarget::*pfn_bCOMMAND)();
    void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_RANGE)(UINT);
    BOOL (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_EX)(UINT);
    .....
    BOOL (AFX_MSG_CALL CWnd::*pfn_bwsp)(UINT, short, CPoint);
    void (AFX_MSG_CALL CWnd::*pfn_vws)(UINT, LPCTSTR);
};

#define DECLARE_MESSAGE_MAP() \
private: \
```

```
static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
static AFX_DATA const AFX_MSGMAP messageMap; \
static const AFX_MSGMAP* PASCAL _GetBaseMessageMap(); \
virtual const AFX_MSGMAP* GetMessageMap() const;

#define ON_WM_CLOSE() \
{ WM_CLOSE, 0, 0, 0, AfxSig_vv, \
  (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL \
    CWnd::*)(void))&OnClose },
```

可以看出，在构建一个 class 的消息映射表时，首先通过类似于 ON\_WM\_XXX 等的宏将消息处理函数（某些成员函数）的地址强制转换为 AFX\_PMSG 类型，然后在 BEGIN\_MESSAGE\_MAP 的配合下填充消息映射表 \_messageEntries[] 的每一项。当一个消息到达后，首先在消息映射表中找到对应的表项，然后根据指定的 Signature（即 UINT nSig，其取值参见枚举类型 AfxSig）利用 union 的功能将其中的 AFX\_PMSG pfn 成员执行一个反转换（参见 CWnd::OnWndMsg() 函数的实现），这个反转换则由联合 MessageMapFunctions 自动完成（这里可见 union 的自动转型作用）。更具体的细节请参考 MFC 源代码。

受此启发，我们可以设想 C++ 编译器构建 vtable 的方法是这样的：

（1）定义如下通用的虚函数指针类型（实际上是经过 Name-Mangling 处理后对应的全局函数指针类型）和 vtable 的类型。

```
typedef void (__cdecl *PVFN)(void);           // 通用的虚函数指针类型
typedef struct {
    type_info * _pTypeInfo;
    PVFN _arrayOfPvfn[];                      // 虚函数个数由初始化语句确定
} VTABLE;
```

（2）在每一个继承分支中的第一个多态类中插入 vptr，而在每一个多态类中都插入 vtable 的声明，如示例 12-27 所示。

示例 12-27

```
class Shape {
    PVFN *_vptr;
    static VTABLE _vtable;
public:
    Shape() : m_color(0) { }
    virtual ~Shape() { }
    float GetColor() const { return m_color; }
    void SetColor(float color) { m_color = color; }
    virtual void Draw() = 0;
private:
    float m_color;    // 颜色
};
```

```

class Rectangle : public Shape {
    static VTABLE _vtable;    // 在实现文件中初始化
public:
    Rectangle() : m_length(1), m_width(1) { ... }
    virtual ~Rectangle(){...}
    .....
    virtual void Draw(){...}
    static unsigned int GetCount(){ return m_count; }
protected:
    Rectangle(const Rectangle& copy){...}
    Rectangle& operator =(const Rectangle& assign){...}
private:
    float m_length;           // 长
    float m_width;            // 宽
    static unsigned int m_count; // 对象计数, 在实现文件中初始化
};

```

编译器在编译完一个 class 后, 把 class 中所有虚函数的指针 (实际就是被转换为全局函数后的地址, 参见 12.7.3 节) 都强制转换为 PVFN 类型, 并用它们初始化 \_vtable 中的 \_arrayOfPvfn[], 就像 ON\_WM\_XXX 所做的那样; 而 \_vptr 则将被初始化为指向 \_vtable 对象或者 \_vtable.\_arrayOfPvfn[0], 具体指向哪里也取决于编译器的实现。在图 12-6 中, Rectangle::\_vtable 可能没有 type\_info 对象, 两个 Rectangle 对象的 vptr 也可能指向 vtable 中的第一个虚函数指针。vptr 的初始化和改写在 class 的各个构造函数和析构函数中完成。

但是, C++编译器显然不可能像 MFC 枚举所有可能的消息处理函数类型那样来枚举所有的虚函数类型。因此, 在遇到通过指针或引用调用虚函数的语句时, 首先根据指针或引用的静态类型来判断所调虚函数是否属于该 class 或者它的某个 public 基类, 然后进行静态类型检查。例如:

```

Shape *pShape = new Rectangle;
pShape->Draw();           // 根据 Shape::Draw()执行类型检查
delete pShape;           // 根据 Shape::~~Shape()执行类型检查

```

检查通过后, 就剩下最后一步工作了: 改写虚函数调用语句。那怎么改写呢? 在 12.6.3 节中, 我们已经知道应该改写为类似下面的样子:

```

(*(pShape->_vptr[2]))(pShape);    // pShape->Draw();
(*(pShape->_vptr[1]))(pShape);    // delete pShape;

```

它们的具体索引值取决于 vtable 的构造。

但是, 语句(pShape->\_vptr[n])从 vtable 中取出来的函数指针类型应该是 PVFN, 与实际调用的虚函数的类型一般是不匹配的 (编译器知道我们定义的每一个虚函数的类型), 所以还应该有一个反向类型强制转换的过程。最后的结果应该类似下面这样 (仅作示意):

```

typedef void (__cdecl *PVFN_Draw)(void);

```

```
typedef void (__cdecl *PVFN_~Shape)(void);
(*(PVFN_Draw)(pShape->_vptr[2]))(pShape);      // pShape->Draw();
(*(PVFN_~Shape)(pShape->_vptr[1]))(pShape);      // delete pShape;
```

可以看出,在整个过程中并没有派生类改写的虚函数 `Rectangle::Draw()` 和 `Rectangle::~~Rectangle()` 参与其中。那怎么会调用到这两个改写的虚函数呢?奥妙就在 `vtable` 的构造及 `pShape` 当前实际指向的对象。

虽然 `pShape` 的静态类型是 `Shape*`,但是在运行时它却指向一个 `Rectangle` 对象,而该对象的 `vptr` 成员指向 `Rectangle::_vtable`,而不是 `Shape::_vtable`;这个 `vtable` 中存放的也都是 `Rectangle` 改写过的虚函数或者新加的虚函数的地址,而不是 `Shape` 的虚函数的地址,除非有的虚函数 `Rectangle` 没有改写。

关于 `vtable` 中虚函数指针的排列顺序,我们可以总结出如下规律:

- ◇ 一个虚函数如果在当前 `class` 中是第一次出现(如果它在其某一个基类中已经出现过,则不能算是第一次出现),则将其地址插入到该 `class` 的每一个 `vtable` 的尾部;
- ◇ 如果派生类改写了基类的虚函数,则这个函数的地址在派生类 `vtable` 中的位置与它在其基类 `vtable` 中的位置一致,而与它在派生类中声明的位置无关;也就是说虚函数第一次出现时它在 `vtable` 中的位置一旦确定,就不会随派生层次的增加而改变,除非改变了它和其他虚函数在 `class` 中第一次声明的顺序;
- ◇ 派生类没有改写的基类虚函数被继承下来并插入派生类 `vtable` 中(与该虚函数所在基类对应下来的那个 `vtable`),且在派生类 `vtable` 中的位置与其在基类 `vtable` 中的位置相同;
- ◇ 显然,派生类的 `vtable` 布局应该兼容其基类的 `vtable`;
- ◇ 在一个 `class` 中,虽然标准没有明确规定所有第一次出现的虚函数在其 `vtable` 中的排列顺序,但是一般都会按照声明的先后顺序排列,特别是如果两个编译器要想做到二进制连接兼容,在 `vtable` 和 `vptr` 的实现技术上必须保持高度一致,例如宣称都支持 COM 规范和 COM 对象开发的编译器就应如此;
- ◇ 多重继承情况下派生类 `vtable` 的布局特点可根据单重继承情况的布局一一推导出来。
- ◇ 虚拟继承及虚拟多重继承情况下的对象模型和 `vptr`、`vtable` 布局就更复杂了,不同编译器的实现方法更是不同。

例如,如果我们按如下方式定义 `Shape` 和 `Rectangle` 类(见示例 12-28),则它们的 `vtable` 将如图 12-7 所示。

示例 12-28

```
class Shape {
public:
    Shape(): m_color(0) {}
    virtual ~Shape() {}
    float GetColor() const { return m_color; }
```

```

void SetColor(float color) { m_color = color; }
virtual void Draw() = 0;
private:
    float m_color;                // 颜色
};
class Rectangle : public Shape {
public:
    Rectangle() : m_length(1), m_width(1) { ... }
    virtual int ObliqueAngle() { return 90; }    // 新增虚函数
    virtual void Draw() { ... }
    .....
    virtual ~Rectangle() { ... }
    static unsigned int GetCount() { return m_count; }
protected:
    Rectangle(const Rectangle& copy) { ... }
    Rectangle& operator =(const Rectangle& assign) { ... }
private:
    float m_length;                // 长
    float m_width;                // 宽
    static unsigned int m_count;    // 对象计数，在实现文件中初始化
};

```

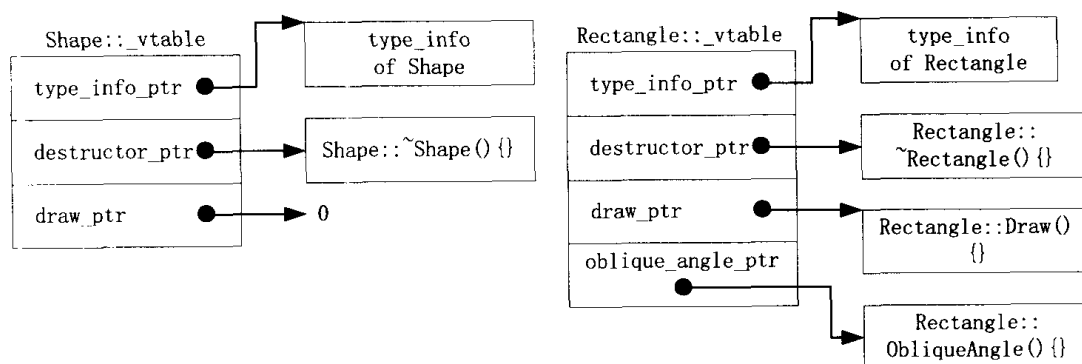


图 12-7 派生类 vtable 兼容基类 vtable

说了这么多，现在可以下结论了：由于 `vptr` 在对象中的偏移不会随派生层次的增加而改变，而且改写的虚函数在派生类 `vtable` 中的位置与它在基类 `vtable` 中的位置始终保持一致，有了这两条保证，再加上被改写虚函数与其基类中对应虚函数的原型和调用规范都保持一致，自然就能轻松地调用起实际所指对象的虚函数了。实际上，在定义 COM 接口时，为了保证接口方法（Interface Method）在实现方（COM 对象的实现库）和调用方（COM 对象的客户程序）之间保持调用规范的一致性，都要显式指定每个方法的调用规范，因为调用规范决定了函数参数压栈、退栈的顺序及由谁来释放堆栈。

试想：如果语句 `(pShape->_vptr[2])` 从 `vtable` 中取出来的函数地址指向的实际函数的类型与所调函数的静态类型 `PVFN_Draw` 不一致的话，比如原型不一致或者仅仅是调用规范不一致，那么程序运行的结果就可想而知了！所有虚函数都是如此。由此

也可以理解 12.6.3 节所述为什么“C++标准要采取隐藏(或遮蔽)策略而不是放宽虚函数改写(Overrides)的规则”的原因了。

最后,我们来想一想:是否有必要在最终的二进制代码(LIB, DLL 和 EXE)中为一个抽象基类产生实际的 vtable 呢?

.....(往下看之前思考一下☺)

显然,根据本节的分析可以看出:没有必要为抽象基类产生实际的 vtable,因为抽象基类不可能实例化,也就不可能有指针或引用指向它的对象,也不可能有 vptr 指向它的 vtable,所以它只存在于概念中。但是在具体的实现中,是否为抽象基类产生 vtable,则取决于编译器的实现。我们头脑中只当是存在的即可。

理解 C++ 对象模型是进一步理解 COM 原理的基础。

## 12.7.2 隐含成员

由于存在隐含成员和填补字节,导致“一个内存对象的实际大小不像它看上去的那样”。从应用编程的角度讲,我们根本没有必要了解这些底层的技术细节,因为它们是语言实现所关心的事情;但是如果你了解了这些知识,可以使你在编程的时候头脑中始终保持一个清晰的“运行时对象视图”,有助于一次性编写出正确的代码来。

一个 C++ 的复合类型对象,其可能的隐含成员包括:若干 vptr、默认构造函数、默认拷贝构造函数、析构函数和默认拷贝赋值函数。至少在现阶段是这样的。

那么在什么情况下,一个类的对象才会含有上述这些隐含成员呢?

对 vptr 而言,当一个类表现多态特性时,其对象就会含有至少一个 vptr 成员,这包括下面的情况:

- ✧ 该类含有虚函数,无论是自己定义的还是从基类继承下来的;
- ✧ 该类的继承链中至少有一个基类是多态类;
- ✧ 该类至少有一个虚基类(virtual base class);
- ✧ 该类包含了多态的成员对象,但是该类不一定是多态类。

显然,当创建一个对象的时候,其隐含的成员 vptr 必须被初始化为指向正确的 vtable,而且这个初始化工作只能在运行时完成,所以这个任务自然就交给了构造函数。我们将在第 13 章详细讨论上述 4 个默认的成员函数的作用。

C++ 对象模型要充分考虑对象数据成员的空间效率和访问速度,以优化性能。另外,每一个对象必须占据足够大的内存空间以便容纳其所有非静态数据成员。因此,对象的实际大小可能比简单地把各个成员的大小加在一起的结果还大,而且与具体的环境有关。造成这种结果的原因主要有两条:

- (1) 由编译器自动安插的额外隐含数据成员,以支持对象模型,例如 vptr;
- (2) 出于对存取效率的考虑而增加的填补字节,以使对象的边界能够对齐到机器字长(WORD),即为 WORD 的整数倍。

所以建议你使用 sizeof() 计算一个对象的真实大小。对于各种情况下的隐含成员和对象的边界调整的详细论述,请参考本书第 8.1.4 节或其他书籍,这里不再赘述。

### 12.7.3 C++编译器如何处理成员函数

在编译器眼中，同一个函数只存在一个实现（函数代码），不管是全局函数还是成员函数。对于在两个编译单元中分别定义的两个完全相同的 `static` 全局函数，由于编译器认为它们是不同的函数，因此会分别为它们生成可执行代码。

那么 C++ 是如何处理类的成员函数的呢？

实际上，C++ 通过 `Name-Mangling` 技术把每一个成员函数都转换成了名字唯一的全局函数，并把通过对象、指针或引用对每一个成员函数的调用语句改写为相应的全局函数调用语句。

我们以 12.7.1 节定义的 `Rectangle` 类为例，其每一个非静态成员函数都会被添加一个本类对象的指针作为第一个参数，这就是 `this` 指针的由来；然后再运用 `Mangling` 技术处理。例如函数 `SetLength` 被编译器改写后的样子可能是：

```
void _SetLength@Rectangle$2F$pf@GS(Rectangle *this, float length) // 全局函数
{
    this->m_length = length;    // 绑定的本质
}
```

因此调用语句：

```
rect1.SetLength(100.5);
```

将被编译器改写为：

```
::_SetLength@Rectangle$2F$pf@GS(&rect1, 100.5);
```

编译器对虚函数也会执行这样的改写。例如 `Rectangle::Draw()` 在改写了从 `Shape` 继承下来的 `Draw()` 后，它就变成了一个全新的函数，本质上与 `Shape::Draw()` 不再有任何关系。编译器在把 `Rectangle::Draw()` 函数 `Mangling` 后将其地址放进了 `Rectangle::_vtable` 的 2 号 slot 中，即：

```
Rectangle::_vtable._arrayOfPvfn[1] = _Draw@Rectangle$1P@GS; // 伪码
```

对调用语句的改写参见 12.7.1 节。

编译器对数据成员也会进行 `Mangling` 处理，目的是区分派生类和基类中可能的同名成员，但是不会提到 `class` 外。

不同的 C++ 编译器对 `class` 的数据成员、成员函数和全局函数等的 `Name-Mangling` 方案是不同的，这是造成不同编译器之间存在二进制连接兼容性的主要原因之一。如果有兴趣，可以打开任意一个 Visual C++ 工程编译后导出的 `map` 文件，其中就罗列了本工程引用的所有函数的内部名称。

### 12.7.4 C++编译器如何处理静态成员

在 C++ 中，凡是使用 `static` 关键字声明和定义的程序元素，不论其作用域是文件、函数或是类，都将具有 `static` 存储类型，并且其生存期限为永久，即在程序开始运行时创建而在程序结束时销毁。因此，类的静态成员（静态数据成员和静态成员函数）都不依赖于对象的存在而存在，也就不需要通过对象来访问，在本质上就是一种全

局变量或函数。

类的静态数据成员可以在 class 的定义中直接初始化，但是要清楚：这只是声明并给它提供了一个初值而已，还必须在某一个编译单元中把它定义一次（即分配内存）。

静态成员函数的设计思想是这样的：如果一个成员函数访问了对象的数据成员，那么给它传入一个 this 指针是必需的；但是有时候一个成员函数不会访问对象的任何数据成员，比如它只是返回该类的名字信息，那么给它传递 this 指针就毫无意义了，因此也没有必要非得把它的调用绑定到一个具体的对象身上。这种想法后来就演变为 C++ 的静态成员函数。

静态成员函数像其他成员函数一样也要经过 Name-Mangling 处理并提出到 class 之外，所不同的是它们不需要 this 指针参数。例如编译器对 Rectangle::GetCount 和 m\_count 改写的结果可能为：

```
unsigned int _GetCount@Rectangle$$@GS( ){ return  ::_m_count@Rectangle$I@GS; }
```

虽然基类的静态成员也会被派生类继承，但这种继承并不是继承它们的实体，而是使得它们在派生中也是可以直接访问的。

静态成员的最大特点就是没有 this 指针，因此可以通过作用域解析运算符 (::) 直接引用。例如：

```
cout << "Current number of Rectangle : " << Rectangle::GetCount() << endl;
```

静态成员的访问规则如图 12-8 所示。

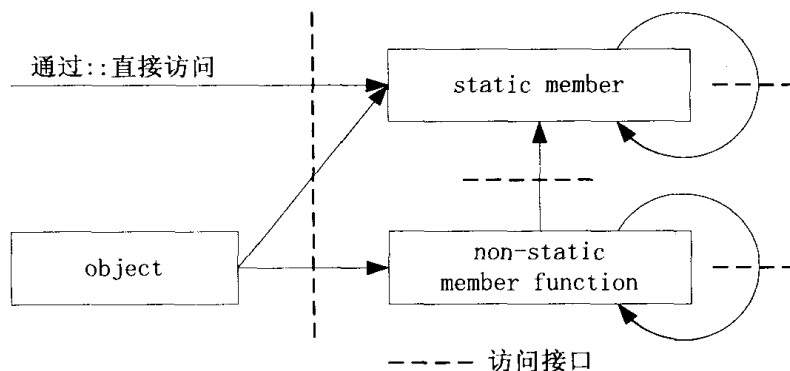


图 12-8 静态成员的访问规则

## 12.8 小结

C++ 是应用最广泛的面向对象编程语言，在笔者心目中 C++/C 是程序员的正宗语言。学好 C++/C 后，再学习其他编程语言如 Visual Basic、Java 就非常容易。

C++ 基本的面向对象特性有封装、继承、多态、运行时绑定等，高级的面向对象编程特性有多重继承、虚拟继承、静态的多态和动态的多态等。C++ 还支持泛型编程，



但它与面向对象编程没有直接的关系。虽然这两种模式的出发点截然相反：泛型编程的初衷是将数据集合（容器）与算法分开并独立发展，而面向对象编程则强调封装和信息隐藏，但是这并不妨碍将它们同时应用于实际软件开发项目中。事实上，泛型编程也可以在面向过程的语言中实现。然而，泛型编程和面向对象编程的组合使 C++ 成为功能非常强大的语言。

面向对象不会是程序设计方法的终点。我们现在不知道 OO 之后的“XO”是什么东西（不是一种酒喔☺），但至少可以推知，“XO”的核心概念必然高于并包容“对象”这一概念，正如对象高于并包容了函数和变量一样。

C++/C 程序设计技术如同少林寺的武功一样博大精深，笔者练了十年，大概只有五成功力。所以，无论什么时候都不要觉得自己的编程水平很高，要虚心学习。

如果你会编写 C++/C 程序，不要因此而得意洋洋，这只是程序员的基本技能而已。如果把系统分析和系统设计比做“战略和战术”，那么编程充其量只是“格斗技能”。所以程序员不要把眼光只盯在程序上，要让自己博学多才。



# 第 13 章 对象的初始化、拷贝和析构

构造函数、析构函数与赋值函数是每个类最基本的函数。它们太普通以至于让人容易麻痹大意，其实这些貌似简单的函数就像没有顶盖的下水道那样危险。

每个类只有一个析构函数，但可以有多多个构造函数（包含一个拷贝构造函数，其他的为普通构造函数）和多个赋值函数（包含一个拷贝赋值函数，其他的为普通赋值函数）。一般地，对于任意一个类 A，如果程序员不显式地声明和定义上述函数，C++编译器将自动为 A 产生 4 个 public inline 的默认函数，这 4 个函数最常见的形式为：

A()	// 默认构造函数
A(const A&);	// 默认拷贝构造函数
~A();	// 析构函数
A& operator = (const A& a);	// 默认赋值函数

这不禁让人疑惑，既然能自动生成函数，为什么还要程序员编写？原因如下。

（1）如果使用“默认的无参数构造函数”和“默认的析构函数”，等于放弃了自主“初始化”和“清除”的机会，C++发明人 Bjarne Stroustrup 的好心好意白费了；

（2）“默认的拷贝构造函数”和“默认的赋值函数”均采用“按成员拷贝”的默认方式来实现。也就是说，它们会依次调用每个数据成员的默认拷贝构造函数和默认赋值函数，除非为它们显式定义了新的构造函数和赋值函数；如此递归下去，直到所有成员都是基本类型为止。但是我们知道，基本类型变量（包括指针）的拷贝和赋值实际上都是按 bit 进行的，因此假如类中含有指针成员，这两个默认函数的执行结果将是两个对象中的对应指针成员指向相同的对象，而这在一般情况下并非是我们所期望的，除非你确实想如此。

对于那些没有吃够苦头的 C++程序员，如果他说编写构造函数、析构函数和赋值函数很容易，可以不用动脑筋，表明他的认识还比较肤浅，水平有待提高。

## 13.1 构造函数与析构函数的起源

作为比 C 更先进的语言，C++提供了更好的机制来增强程序的安全性。C++编译器具有严格的类型安全检查功能，它几乎能找出程序中所有的语法问题，这的确帮了程序员的大忙。但是程序通过了编译检查并不表示已经没有错误了，在“错误”

的大家庭里,“语法错误”的地位只能算是小弟弟,而“有水平”的错误通常隐藏得很深,就像狡猾的罪犯,想“捉”住它可不容易。根据经验,不少难以察觉的程序错误是由于变量没有被正确初始化或清除造成的,而初始化和清除工作很容易被人遗忘。Bjarne Stroustrup 在设计 C++语言时充分考虑到了这个问题并很好地加以解决。

把对象的初始化工作放在构造函数中,把清除工作放在析构函数中。当创建对象时,构造函数被自动执行;当对象消亡时,析构函数被自动执行。这样就不用担心忘记对象的初始化和清除工作了。

构造函数与析构函数的名字不能随便起,必须让编译器认得出(或者约定俗成)才可以被自动执行。Bjarne Stroustrup 的命名方法既简单又合理:

让构造函数、析构函数与类同名,由于析构函数的目的与构造函数的目的相反,就加前缀“~”以示区别(取“求反”之意)。

除了名字外,构造函数与析构函数的另一个特别之处就是没有返回值类型,这与返回值类型为 void 的函数是不同的。构造函数与析构函数的使命非常明确,就像出生与死亡,光溜溜地来光溜溜地去。如果它们有返回值类型,那么编译器将不知所措,为了防止节外生枝,干脆规定没有返回值类型(以上典故取自参考文献《C++编程思想》([Eekel, p55~p56])。

本章将用很多篇幅阐述构造函数和析构函数,我们首先应当搞清楚构造函数和析构函数应该做什么事情。

**【提示 13-1】:** 不要在构造函数内做与初始化对象无关的工作,不要在析构函数内做与销毁一个对象无关的工作。也就是说,构造函数和析构函数应该做能够满足正确初始化和销毁一个对象的最少工作量,否则会降低效率,甚至可能会让人误解。

比如,对于一个用于消息发送和接收的类来说,不应该在构造函数内打开一个 socket 连接,同样不应该在析构函数内断开一个 socket 连接,而应该把打开和断开 socket 连接放到另外的成员函数内来完成,因为我们在创建该类的一个对象时并不一定想马上发送消息,否则别人还会误以为没有打开 socket 连接就发送消息,没有断开连接就删除了对象,显然不符合 socket 编程习惯。

## 13.2 为什么需要构造函数和析构函数

编译器无法预期一个程序在执行过程中会在何时创建一些什么对象,而只能根据当时的上下文要求来创建。对象的初始化最好能够通过运行时执行一个函数来完成,而且是在对象创建的同时,这个函数就是构造函数。同样,对象在完成其使命的时候能够通过一个函数来销毁,这就是析构函数。

怎样才算一个对象已经被创建起来了呢?当给一个对象分配好原始内存空间(raw-memory)的时候这个对象就应该算创建起来了,只不过它还处于一种“原始状态”,即未初始化的、不良的状态。如果把这样的内存直接拿来使用,除非第一个

操作是赋值，否则极有可能出错，见示例 13-1。

示例 13-1

---

```

long lng1;                // 局部变量
cout << lng1 << endl;    // 访问原始内存
char * pStr = (char*)malloc(1024);
cout << pStr << endl;    // 访问未初始化内存

```

---

因此，创建一个变量或动态对象时一定不要忘记初始化。有人说：我在声明一个变量后，在第二行立刻就用赋值语句来初始化了，这总可以吧。这样做的人显然还没有搞清楚“赋值”和“初始化”在语义上的区别。

**【提示 13-2】：** 初始化就是在对象创建的同时使用初值直接填充对象的内存单元，因此不会有数据类型转换等中间过程，也就不会产生临时对象；而**赋值**则是在对象创建好后任何时候都可以调用的而且可以多次调用的函数，由于它调用的是“=”运算符，因此可能需要进行类型转换，即会产生临时对象。

C++对象可以使用构造函数来初始化，构造函数是任何对象创建时自动调用的第一个成员函数，也是为每个对象仅调用一次的成员函数。所以构造函数的作用就是：当对象的内存分配好后把它从原始状态变为良好的可用的状态。

有的程序员可能认为：虽然我没有在构造函数中初始化数据成员（甚至没有定义构造函数），但是我在声明一个对象后会马上调用它的 `set_XXX()` 函数来初始化它的每一个成员，效果也是一样的。

你说这是何苦呢？构造函数能够自动调用，你却使用 `set` 函数逐个初始化数据成员，不仅烦琐而且增加了运行时开销，何况也无法保证每创建一个对象时就必定记得调用这一堆 `set` 函数！

**【建议 13-1】：** 最好为每个类显式地定义构造函数和析构函数，即使它们暂时空着，尤其是当类含有指针成员或引用成员的时候。

**【提示 13-3】：** 构造函数的另一重要用途就是给一些可能存在的隐含成员如 `vptr` 创建一个初始化的机会，否则虚拟机制将不能保证实现。每当此时，如果程序员没有为一个多态类显式地定义默认构造函数、拷贝构造函数、析构函数或拷贝赋值函数，那么编译器会自动地生成相应的函数，它们都是 `public inline` 的，并在其中插入正确初始化或修改 `vptr` 数据成员值的代码，从而确保基类对象和派生类对象构造时及在它们之间拷贝时 `vptr` 能够指向或重新指向恰当的 `vtable`。这样的四个函数就分别叫做非平凡默认构造函数、非平凡拷贝构造函数、非平凡析构函数和非平凡拷贝赋值函数。

## 13.3 构造函数的成员初始化列表

我们一般习惯在构造函数体内来初始化数据成员，然而这不是真正意义上的初始化，而是赋值。不过，由于构造函数是创建一个对象时自动调用的第一个成员函数，因此我们也愿意把构造函数体内的赋值语句当成初始化来看待。

真正的初始化是使用所谓的“初始化表达式表”（简称初始化列表）进行的。初始化列表位于构造函数参数表之后，在函数体 `{}` 之前。这说明该列表里的初始化工作发生在函数体内的任何代码被执行之前，编译器也确实是这样做的。

构造函数初始化列表的使用规则如下：

（1）如果类存在继承关系，派生类可以直接在其初始化列表里调用基类的特定构造函数以向它传递参数，因为我们不能在初始化对象时访问基类的数据成员，见示例 13-2。

示例 13-2

---

```
class A {  
    ...  
public:  
    A(int x);           // A 的构造函数  
};  
class B : public A {  
    ...  
    B(int x, int y);    // B 的构造函数  
};  
B::B(int x, int y) : A(x)    // 在初始化列表里调用 A 的构造函数  
{  
    ...  
}
```

---

（2）类的非静态 `const` 数据成员和引用成员只能在初始化列表里初始化，因为它们只存在初始化语义，而不存在赋值语义（参见本书第 7.5 节）；

（3）类的数据成员的初始化可以采用初始化列表或函数体内赋值两种方式，这两种方式的效率不完全相同，见示例 13-3。

示例 13-3

---

```
class A {  
    ...  
    A(void);           // 默认构造函数  
    A(const A& other);  // 拷贝构造函数  
    A& operator =(const A& other); // 赋值函数
```

---

```
};

class B {
public:
    B(const A& a);           // B 的构造函数
private:
    A m_a;                 // 成员对象
};

// (1) 采用初始化列表的方式初始化
B::B(const A& a) : m_a(a)
{
    ...
}

// (2) 采用函数体内赋值的方式初始化
B::B(const A& a)
{
    m_a = a;
    ...
}
```

本例第一种方式，类 B 的构造函数在其初始化列表里调用了类 A 的拷贝构造函数，从而将成员对象 m\_a 初始化。

本例第二种方式，类 B 的构造函数在函数体内用赋值的方式将成员对象 m\_a 初始化。我们看到的只是一条赋值语句，但实际上 B 的构造函数干了两件事：先暗地里创建 m\_a 对象（调用了 A 的默认构造函数），再调用类 A 的赋值函数，才将参数 a 赋给 m\_a。

显然第一种方式的效率比第二种高。

对于内部数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但第二种方式的程序版式似乎更清晰些，见示例 13-4。

示例 13-4

```
class F {
public:
    F(int x, int y);        // 构造函数
private:
    int m_x;
    int m_y;
};

// (1) 采用初始化列表的方式初始化
F::F(int x, int y) : m_x(x), m_y(y)
{
}

// (2) 采用函数体内赋值的方式初始化
```

```
F::F(int x, int y)
{
    m_x = x;
    m_y = y;
}
```

## 【提示 13-4】:

当使用成员初始化列表来初始化数据成员时，这些成员真正的初始化顺序并不一定与你在初始化列表中为它们安排的顺序一致，编译器总是按照它们在类中声明的次序来初始化的。因此，最好是按照它们的声明顺序来书写成员初始化列表：

- (1) 调用基类的构造函数，向它们传递参数；
- (2) 初始化本类的数据成员（包括成员对象的初始化）；
- (3) 在函数体内完成其他的初始化工作。

如果各个成员的初始化存在依赖关系，要注意顺序问题。可以调整一下数据成员的声明顺序来避免这个问题，或者把存在初始化依赖关系的数据成员分别放在初始化列表和构造函数体内来初始化。

## 13.4 对象的构造和析构次序

这里的构造和析构次序并非是说明给对象分配内存空间有什么次序，而是说对象初始化和销毁的次序。如果一个类没有基类，那么它的构造过程很简单，仅仅把自己的数据成员初始化就可以了；但是如果一个类是派生类，那么它的构造函数将首先调用每一个基类的构造函数，然后调用成员对象的构造函数；而每一个基类的构造函数又将首先调用它们各自基类的构造函数，……，直到最根类。因此，任何一个对象总是首先构造最根类的子对象，然后逐层向下扩展，直到把整个对象构造起来。

但是由于一个类可以存在多个构造函数，却只能有一个析构函数，那么对象在析构的时候如何能够匹配每一个构造函数呢？不用担心，因为：

(1) 析构会严格按照与对象构造相反的次序执行，该次序是唯一的，否则编译器将无法自动执行析构过程；

(2) 数据成员的初始化次序完全不受它们在初始化列表中出现次序的影响，只由它们在类中声明的次序决定，因为这个顺序是唯一的。显然，每个构造函数的初始化列表中各成员出现的顺序不可能完全相同，如果数据成员按照初始化列表的次序进行构造，将导致析构函数无法得到唯一的逆序。

对象的构造和析构过程就如同网络消息的打包和解包过程，如图 13-1 所示。



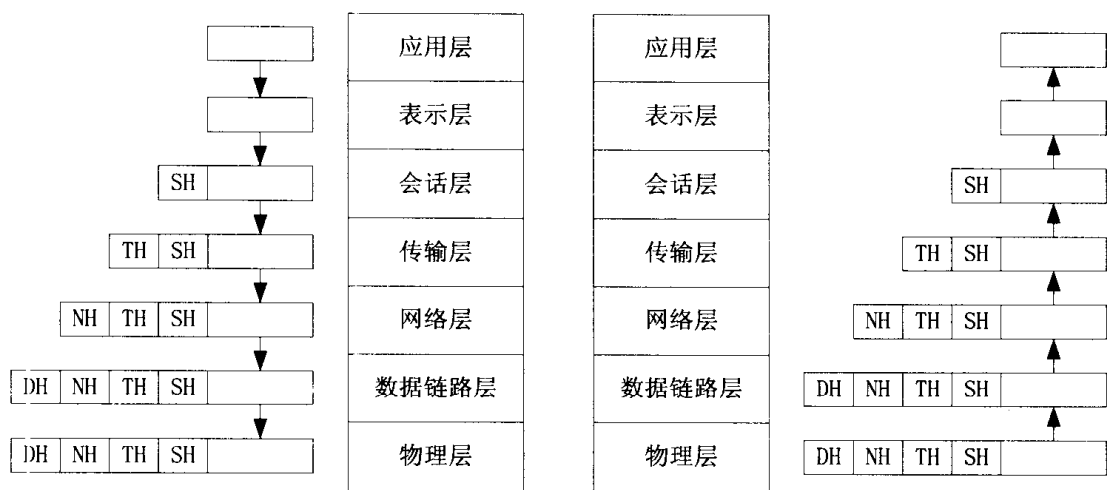


图 13-1 网络消息的打包和解包过程

### 13.5 构造函数和析构函数的调用时机

许多人虽然知道构造函数、赋值函数和析构函数的作用，但是并不清楚这些函数在什么情况下和什么时候才会调用。表 13-1 总结了构造函数和析构函数的调用时机。

表 13-1 构造函数和析构函数的调用时机

对象类型	构造函数、赋值函数和析构函数的调用时机
非静态的局部对象，如 T obj; T obj = ...; Const T obj = ...;	<div><div>➤</div>在程序执行到该对象的定义处时，创建对象并调用相应的构造函数</div> <div><div>➤</div>如果在定义对象时没有提供初始值，则会暗中调用默认构造函数；如果没有默认构造函数，则该对象不会自动初始化</div> <div><div>➤</div>如果在定义对象时提供了初始值，则会暗中调用类型匹配的带参数的构造函数（包括拷贝构造函数）。如果没有定义这样的构造函数，编译器就可能报错</div>

(续表)

对象类型	构造函数、赋值函数和析构函数的调用时机
全局对象，如 T g_obj; T g_obj = ...; static T g_obj; static T g_obj = ...;	<ul style="list-style-type: none"><li>➤ 在程序进入 <code>main()</code> 之前自动调用它们的相应构造函数来初始化，但是初始化的顺序不确定</li><li>➤ 如果在定义对象时没有提供初始值，则会暗中调用默认构造函数；如果没有默认构造函数，则自动初始化为 0</li><li>➤ 如果在定义对象时提供了初始值，则会暗中调用类型匹配的带参数的构造函数（包括拷贝构造函数）。如果没有定义这样的构造函数，编译器就可能报错</li><li>➤ 直到 <code>main()</code> 结束后才会调用析构函数</li></ul>
类的静态数据成员对象	<ul style="list-style-type: none"><li>➤ 等同于全局对象的情况</li></ul>
对象引用	<ul style="list-style-type: none"><li>➤ 其初始化和销毁都不会调用构造函数和析构函数，这就是引用传递比值传递高效的原因</li></ul>
动态创建对象	<ul style="list-style-type: none"><li>➤ 当调用 <code>new</code> 运算符动态创建对象时，自动调用类型匹配的构造函数</li><li>➤ 当接收返回地址的指针为静态局部指针变量时，等同于静态局部对象的情况</li><li>➤ 当接收返回地址的指针为非静态局部指针变量时，等同于非静态局部对象的情况</li><li>➤ 当调用 <code>delete</code> 运算符删除对象时，自动调用对象的析构函数</li></ul>
对象赋值	<ul style="list-style-type: none"><li>➤ 调用类型匹配的 <code>operator=</code> 重载函数，因此如果没有定义这样的赋值函数，编译器将可能报错</li></ul>

## 13.6 构造函数和赋值函数的重载

C++ 允许为类定义多个构造函数，即重载构造函数，目的在于可以用不同的方式来初始化对象。同样，也可以为类重载赋值函数（`operator=`）。

构造函数分为三类：默认构造函数、拷贝构造函数和其他带参数的构造函数。

所谓默认构造函数是这样的构造函数：要么它没有参数（即参数列表为 `void`），要么所有参数都有默认值。

**【提示 13-5】** 不能同时定义一个无参数的构造函数和一个参数全部有默认值的构造函数，否则会造成二义性。

拷贝构造函数是这样的构造函数：第一个参数为本类对象的引用、`const` 引用、`volatile` 引用或 `const volatile` 引用，并且没有其他参数，或者其他参数都有默认值。

拷贝构造函数的参数必须是同类对象的引用，而不能是对象值。如果允许定义值传递的拷贝构造函数，就会与引用传递的构造函数产生二义性，见示例 13-5。

示例 13-5

---

```
class A
{
public:
    A(A copy){...}           // (1)
    A(const A& other){...}   // (2)
    //...
};
A a;
A b = a;
```

---

实际上，C++不可能允许// (1)这种形式的拷贝构造函数，即它是非法的。原因如下：这个拷贝构造函数在参数传递的过程中又要调用拷贝构造函数本身，因为是“值传递”，而调用拷贝构造函数时又要先进行参数传递，参数传递又要调用拷贝构造函数……于是陷入不停地分配堆栈的无限递归中，而每一次压栈过程中又嵌套了压栈，致使每一次的压栈都不能完成，因此是一个错误。我设计了一个模拟这种“拷贝构造函数”形式的行为奇怪的函数（因为这个“拷贝构造函数”不允许编译☺），见示例 13-6。

示例 13-6

---

```
#include <iostream>
using namespace std;
class Test;
void StrangeFunction(Test x);    // 值传递
static unsigned long g_iCount = 0;
class Test{
public:
    Test(){}
    Test(const Test& copy)
    {
        cerr << "Test(const Test& copy) was called " \
              << ++g_iCount << " times." << endl;
        ::StrangeFunction(copy);
    }
private:
    double m_dArray[10];
};
inline void StrangeFunction(Test x){}
int main()
{
    Test a;
    StrangeFunction(a);
    return 0;
}
```

---

函数 `StrangeFunction` 在效果上和“拷贝构造函数”`A(A copy);`一样，它最终会导致类 `Test` 的拷贝构造函数无限递归地调用自己，直到堆栈溢出。在作者的机器上，试着把数组 `Test::m_dArray` 的大小调整为 10 000，而函数堆栈大小默认为 1MB，这样类 `Test` 的拷贝构造函数只递归了 11 次便退出了。改变堆栈的设置，或者参数的大小，就可以改变递归的次数。

最后要说明的是：在 C++ 语言中，`A(A copy);` 是一个语法错误的函数，而并非一个无限递归函数；而 `StrangeFunction` 却是一个间接递归函数。

**【提示 13-6】** 如果没有显式地定义默认构造函数却定义了带参数的构造函数，那么后者的存在就会阻止编译器生成前者，于是类就没有默认构造函数。此时如果定义该类型对象就会导致编译错误。

**【建议 13-2】** 一般来说，重载的构造函数的行为都差不多，因此必然存在重复代码片段。当我们为类定义多个构造函数时，设法把其中相同任务的代码片段抽取出来并定义为一个非 `public` 的成员函数，然后在每一个构造函数中适当的地方调用它。

类的赋值函数 `operator=()` 也是一种拷贝函数，当然也可以重载。如果赋值函数的参数类型就是当前类，那么就是类的拷贝赋值函数。拷贝赋值函数可有如下几种形式（不考虑返回类型）：

```
A& operator=(const A& copy);    // (1): 编译器默认形式
A& operator=(A& copy);         // (2)
A& operator=(A copy);          // (3)
```

(2) 和 (3) 实际上是 `operator=` 的重载形式，其中 (3) 又会调用当前类的拷贝构造函数。

## 13.7 示例：类 `String` 的构造函数和析构函数

本节和后面两节将以类 `String` 的设计与实现为例，深入阐述被很多教科书忽视了的道理（见示例 13-7）。

注意，在构造函数中，`m_data` 被初始化为空字符串（只有“\0”）而不是 `NULL`。因为 C++ 中的任何字符串的长度至少为 1（即至少包含一个结束符“\0”）。空字符串也是有效的字符串，它的长度为 1，因此它代表一块合法的内存单元而不是 `NULL`（注意：STL 的 `std::string` 不一定包含“\0”）。

示例 13-7

```
class String {
public:
    String(const char *str = "");           // 默认构造函数
    String(const String& copy);             // 拷贝构造函数
    ~String();                             // 析构函数
```

```

String& operator = (const String& assign); // 赋值函数
private:
    size_t    m_size;                // 保存当前长度
    char      *m_data;              // 指向字符串的指针
};
// String 的默认构造函数
String::String(const char *str)
{
    if(str == NULL) {
        m_data = new char[1];
        *m_data = '\0';
        m_size = 0;
    } else {
        int length = strlen(str);
        m_data = new char[length+1];
        strcpy(m_data, str);
        m_size = length;
    }
}
// String 的析构函数
String::~String(void)
{
    delete [] m_data;
}

```

## 13.8 何时应该定义拷贝构造函数和拷贝赋值函数

由于并非所有的对象都会使用拷贝构造函数和拷贝赋值函数，程序员可能对这两个函数有些轻视。请先记住以下提示，在阅读下文时就会留心。

**【提示 13-7】：**本章开头讲过，如果不主动编写拷贝构造函数和拷贝赋值函数，编译器将以“按成员拷贝”的方式自动生成相应的默认函数。倘若类中含有指针成员或引用成员，那么这两个默认的函数就可能隐含错误。

以类 String 的两个对象 a、b 为例。假设 a.m\_data 的内容为“Hello”，b.m\_data 的内容为“world”。现将 a 赋值给 b，默认赋值函数的“按成员拷贝”意味着执行 b.m\_data = a.m\_data。这将造成 3 个错误：

- (1) b.m\_data 原持有的内存没有被释放，造成内存泄漏；
- (2) b.m\_data 和 a.m\_data 指向同一块内存，a 或 b 任何一方变动都会影响另一方；
- (3) 在对象被析构时，m\_data 被 delete 了两次。

拷贝构造也是一样的道理。这个过程如图 13-2 所示。所以我们应该主动改变这种默认语义，具体做法参见 13.9 节。

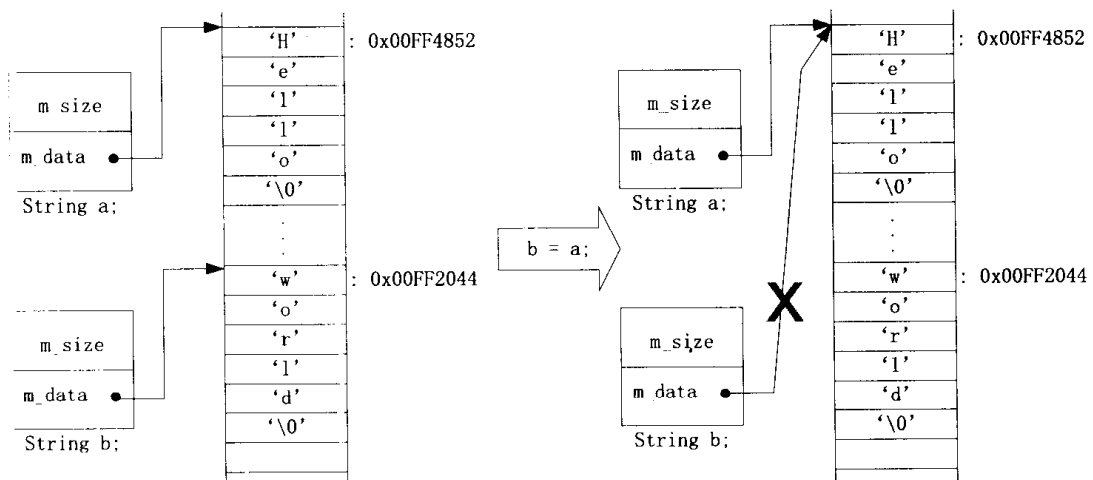


图 13-2 指针按位拷贝引起的错误

## 【提示 13-8】:

拷贝构造函数和拷贝赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对象被创建并用另一个已经存在的对象来初始化它时调用的；而赋值函数只能把一个对象赋值给另一个已经存在的对象，使得那个已经存在的对象具有和源对象相同的状态。

以下程序中，第三个语句和第四个语句很相似，你分得清楚哪个调用了拷贝构造函数，哪个调用了拷贝赋值函数吗？

```
String a("hello");
String b("world");
String c = a;           // 调用拷贝构造函数，最好写成 c(a);
c = b;                  // 调用赋值函数
```

本例中第三个语句的风格较差，宜改写成 `String c(a)`，以区别于第四个语句。

## 13.9 示例：类 String 的拷贝构造函数和拷贝赋值函数

示例 13-8 展示了 String 类的拷贝构造函数和拷贝赋值函数的一种简单实现。

示例 13-8

```
// 拷贝构造函数
String::String(const String& other)
{
    // 提示：允许操作 other 的私有成员 m_data
    size_t len = strlen(other.m_data);
    m_data = new char[len + 1];
    strcpy(m_data, other.m_data);
    m_size = len;
}
```

```
// 赋值函数
String& String::operator=(const String& other)
{
    // (1) 检查自赋值
    if(this != &other) {
        // (2) 分配新的内存资源，并拷贝内容
        char *temp = new char[strlen(other.m_data) + 1];
        strcpy(temp, other.m_data); // copy '\0' together
        // (3) 释放原有的内存资源
        delete []m_data;
        m_data = temp;
        m_size = strlen(other.m_data);
    }
    // (4) 返回本对象的引用
    return *this;
}
```

类 String 拷贝构造函数与默认构造函数（参见 13.7 节源代码）的区别是：在函数入口处无需与 NULL 进行比较，这是因为“引用”不可能是 NULL，而“指针”可以为 NULL。

类 String 的赋值函数比构造函数复杂得多，分四步实现：

（1）检查自赋值。你可能会认为多此一举，难道有人会愚蠢到写出 `a = a` 这样的自赋值语句？的确不会！但是间接的自赋值仍有可能出现，如示例 13-9 所示。

示例 13-9

```
// 间接自赋值
p = &a;
...
a = *p;
```

自赋值的缺点就是耗费不必要的时间。

注意不要将检查自赋值的 if 语句：

```
if(this != &other) // 地址相等才认为是同一个对象
```

错写成：

```
if(*this != other) // 值相等不能作为自赋值的判断依据
```

（2）分配新的内存资源，并拷贝字符串。在这里，如果先把原有的内存资源释放，万一后来的内存重分配操作失败，结果就惨了！所以，先分配内存交给一个临时指针，万一分配失败，也不会改变 `this` 对象，这是为了异常安全起见！

**【提示 13-9】** 注意函数 `strlen` 返回的是有效字符串长度，不把结束符“\0”计算在内。函数 `strcpy` 会连“\0”也一起拷贝。

（3）用 `delete` 释放原有的内存资源。

**【提示 13-10】** 如果现在不释放内存，以后就没机会了，将造成内存泄漏。

(4) 返回本对象的引用，目的是为了实现在像  $a = b = c$ ；这样的链式表达式。

**【提示 13-11】** 不要将 `return *this`；错写成 `return this`；。那么能否写成 `return other`；呢？效果不是一样吗？绝对不可以！因为我们不知道参数 `other` 的生存期，`other` 有可能是一个临时对象，在赋值结束后它马上消失，那么 `return other`；返回的将是垃圾。

除了示例 13-8 给出的最基本的实现方法外，为了提高赋值函数的异常安全性，类 `String` 的拷贝赋值函数实际上还可以这样来实现（见示例 13-10）。

示例 13-10

```
void String::swap(String& other)           // 交换两个 String 对象的成员
{
    std::swap(m_data, other.m_data);      // 异常安全的！
    std::swap(m_size, other.m_size);      // 异常安全的！
}
String& String::operator=(const String& other)
{
    String temp = other;                  // 调用 String 的拷贝构造函数
    temp.swap(*this);                     // 当前对象与临时对象交换，异常安全的！
    return (*this);
}
String& String::operator=(String other)    // 值传递将调用拷贝构造函数
{
    other.swap(*this);                     // 直接与临时对象交换，异常安全的！
    return (*this);
}
```

通过调用拷贝构造函数来实现拷贝赋值函数，可以确保在拷贝构造函数抛出异常的情况下立即终止赋值操作，因此不会修改左值对象。

## 13.10 用偷懒的办法处理拷贝构造函数和拷贝赋值函数

如果我们实在不想编写拷贝构造函数和拷贝赋值函数（不想拷贝对象），又不允许别人使用编译器自动生成的默认函数，该怎么办？

偷懒的办法是：只需将拷贝构造函数和拷贝赋值函数声明为 `private`，并且不实现它们，但是不能像虚函数那样置为 0（它只能用来初始化纯虚函数）。由于显式声明的这两个函数会阻止编译器自动生成相应的默认函数，因此这样做不仅可以防止



用户直接调用它们，而且也防止了用户通过其他成员函数或友元函数来间接地调用它们。例如：

```
class A {
    //...
    const char * GetType();           // 未实现
private:
    A(const A& a);                   // 私有的拷贝构造函数
    A& operator=(const A& a);        // 私有的拷贝赋值函数
};
```

如果有人试图编写如下程序：

```
A  b(a);                           // 调用了私有的拷贝构造函数
b = a;                             // 调用了私有的赋值函数
```

编译器将指出错误，因为外界不可以操作 A 的私有函数。

甚至可以把类的所有构造函数和赋值函数都声明为 **private**，这样就彻底阻止了该类的实例化；或者把默认构造函数声明为 **private**，而把其他带参数的构造函数声明为 **public**，这样就强迫用户使用带参数的构造函数来声明和定义对象。

## 13.11 如何实现派生类的基本函数

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述基本函数时应注意以下事项：

(1) 派生类的构造函数应在其初始化列表里显式地调用基类的构造函数（除非基类的构造函数不可访问）；

(2) 如果基类是多态类，那么必须把基类的析构函数定义为虚函数，这样就可以像其他虚函数一样实现动态绑定；否则有可能造成内存泄漏，见示例 13-11。

示例 13-11

```
#include <iostream.h>
class Base {
public:
    virtual ~Base() { cout << "Base::~~Base()" << endl ; }
    //...
};
class Derived : public Base {
public:
    virtual ~Derived() {
        delete p_test;
        cout << "Derived::~~Derived()" << endl ;
    }
    //...
```

```
private:
    char *p_test;
};
void main(void)
{
    Base *pB = new Derived; // upcast
    delete pB;
}
```

输出结果为:

Derived::~Derived()

Base::~Base()

如果析构函数不是虚函数, 那么输出结果将是:

Base::~Base

这就是说, 如果基类的析构函数不是虚函数, 派生类对象拥有的内存单元就不会释放, 因此造成内存泄漏。

(3) 在编写派生类的赋值函数时, 注意不要忘记对基类的数据成员重新赋值, 这可通过调用基类的赋值函数来实现, 见示例 13-12。

示例 13-12

```
class Base {
public:
    //...
    Base& operator=(const Base& other); // Base 的拷贝赋值函数
private:
    int m_i, m_j, m_k;
};
class Derived : public Base {
public:
    //...
    Derived& operator=(const Derived& other); // Derived 的拷贝赋值函数
private:
    int m_x, m_y, m_z;
};
Derived& Derived::operator=(const Derived& other)
{
    // (1) 检查自赋值
    if(this != &other) {
        // (2) 对基类的数据成员重新赋值
        Base::operator=(other); // 因为不能直接操作基类的私有数据成员
        // (3) 对派生类的数据成员赋值
        m_x = other.m_x;
        m_y = other.m_y;
        m_z = other.m_z;
    }
    // (4) 返回本对象的引用
}
```

```
return *this;
```

```
}
```

---

### ❧ 一些心得体会 ❧

---

有些 C++ 程序设计书籍称构造函数、析构函数和赋值函数是类的“Big-Three”，它们的确是任何类最重要的函数，不容轻视。

也许你认为本章的内容已经够多了，学会了就能平安无事，笔者不能做这个保证。如果希望学透“Big-Three”，请仔细阅读参考文献[Cline]、[Meyers]和[Murry]。

---



## 第 14 章 C++函数的高级特性

对比于 C 语言的函数，C++增加了重载（overload）、内联（inline）、const 和 virtual 四种新机制。其中重载和内联机制既可用于全局函数，也可用于类的成员函数，const 与 virtual 机制仅用于类的成员函数。

重载和内联肯定有其好处才会被 C++语言采纳，但是不可以当成免费的午餐而滥用。本章将探究重载和内联的优点与局限性，说明什么情况下应该采用或不应该采用，以及要如何防止错用。

### 14.1 函数重载的概念

#### 14.1.1 重载的起源

自然语言中，一个词可以有許多不同的含义，即该词被重载了。人们可以通过上下文来判断该词到底是哪种含义。“词的重载”可以使语言更加简练，如“吃饭”的含义十分广泛，人们没有必要每次非得说清楚具体吃了什么。

在 C++程序中，可以将语义、功能相似的几个函数用同一个名字来表示，即函数名被重载。函数重载便于记忆，提高了函数的易用性，这是 C++语言采用重载机制的一个理由。例如下面示例中的函数 EatBeef、EatFish、EatChicken 可以用同一个函数名 Eat 表示，用不同类型的参数加以区别，见示例 14-1。

示例 14-1

void EatBeef(...);	// 可以改为	void Eat(Beef ...);
void EatFish(...);	// 可以改为	void Eat(Fish ...);
void EatChicken(...);	// 可以改为	void Eat(Chicken ...);

C++语言采用重载机制的另一个理由是：类的构造函数需要重载机制，C++规定构造函数必须与类名相同，因此只能有一个名字。如果想用几种不同的方法来创建对象该怎么办？别无选择，只能用重载机制来实现。所以类可以有多个同名的构造函数。

还有一个理由就是：使各种运算符能够支持对象语义，因为 C++中运算符的语义都是固定的，而且就那么几个运算符，所以要想让同一个运算符同时能支持对不同类型对象的操作就必须使用重载机制。

#### 14.1.2 重载是如何实现的

几个同名的重载函数仍然是不同的函数，但是调用者可不管，反正都是使用一

个名字。然而在最终的二进制可执行程序中是不允许有同名函数出现的，因为所有的函数最终都将转换成等效的全局函数，调用语句也会做相应的转换。所以，作为语言实现的编译器必须为每一个被重载的函数生成唯一的内部名称，才能把它们彼此区分开。这是如何办到的呢？我们自然想到函数接口的两个要素：参数列表与返回值。

如果同名函数的参数不同（包括类型、顺序或数量不同），那么很容易识别出它们是不同的函数。

如果同名函数仅仅是返回值类型不同，有时可以区分，有时却不能。例如：

```
void Function(void);  
int  Function(void);
```

上述两个函数，第一个没有返回值，第二个的返回值类型为 `int`。对于如下语句：

```
int  x = Function();
```

则可以判断出 `Function` 为第二个函数。问题是：在 C++/C 程序中，我们可以像调用过程那样来调用函数（即忽略函数的返回值），此时编译器和程序员都不知道哪个 `Function` 将被调用。

**【提示 14-1】** 所以只能靠参数列表而不能仅靠返回值类型的不同来区分重载函数。编译器根据参数列表为每个重载函数产生不同的内部标识符。

例如编译器为 14.1.1 节示例中的 3 个 `Eat` 函数产生像 `_eat_beef`、`_eat_fish`、`_eat_chicken` 之类的内部标识符，这就叫做 Name-Mangling 技术，即名称修饰或者重命名机制。C++ 标准并没有规定一种统一的重命名方案，因此不同的编译器对重载函数可能产生不同风格的内部标识符，这就是不同厂商的 C++ 编译器和连接器不能兼容的一个主要原因。

如果 C++ 程序要调用已经被编译的 C 函数，该怎么办？假设某个 C 函数的声明如下：

```
void __cdecl foo(int x, int y);
```

该函数被 C 编译器编译后在库中的名字为 `_foo`，而 C++ 编译器则会产生像 `_foo_int_int` 之类的内部名字用来支持函数重载。由于编译后的名字不同，C++ 程序不能直接调用编译后的 C 函数。C++ 提供了一个 C 连接交换指示符 `extern "C"` 来解决这个问题。例如：

```
#ifdef __cplusplus  
extern "C" {  
#endif  
void __cdecl foo(int x, int y);  
... // 其他 C 函数  
#ifdef __cplusplus  
}  
#endif
```

或者写成:

```
#ifdef __cplusplus
extern "C" {
#endif
#include "myheader.h"
... // 其他 C 头文件
#ifdef __cplusplus
}
#endif
```

这就是连接规范的概念。上述代码是在告诉 C++编译器, 函数 foo 是个 C 连接函数, 应该为它生成名字 \_foo 而不是 \_foo\_int\_int, 并指示连接器到 C 程序库中去找该函数的定义。C++编译器开发商已经对 C 标准库的头文件做了 extern "C" 处理, 所以我们可以用 #include 直接引用这些头文件。

**【提示 14-2】** 并不是两个函数的名字相同就能构成重载。全局函数和类的成员函数同名不算重载, 因为它们的作用域不同。

例如:

```
void Print(...);           // 全局函数: 文件作用域
class A
{
    ...
    void Print(...);       // 成员函数: 类作用域
}
```

不论两个 Print 函数的参数是否相同, 如果 A::Print 要调用全局的 Print 函数, 为了与成员函数 Print 区别, 应该在函数名前加一元作用域解析运算符 (::), 否则成员函数将遮蔽同名的全局函数。如:

```
void A::Print(...)
{
    ::Print(...);         // 表示调用的是全局函数而不是成员函数
}
```

### 14.1.3 当心隐式类型转换导致重载函数产生二义性

示例 14-2 的代码中, 第一个 output 函数的参数为 int 类型, 第二个 output 函数的参数是 float 类型。语句 output (0.5) 将产生编译错误, 因为 0.5 的类型为 double, 而一个 double 临时变量既可以转换为 int, 也可以转换为 float, 二者都是标准转换, 根据规则, 两个转换一样好。隐式类型转换在很多地方可以简化程序的书写, 但是也可能留下隐患。

示例 14-2

```
#include <iostream.h>
```

```
void output( int x ); // 函数声明
void output( float x ); // 函数声明
void output( int x )
{
    cout << " output int " << x << endl ;
}
void output( float x )
{
    cout << " output float " << x << endl ;
}
void main(void)
{
    int    x = 1;
    float y = 1.0;
    output(x);           // output int 1
    output(y);           // output float 1
    output(1);           // output int 1
    output(0.5);         // 错误! 不明确的调用, 因为自动类型转换
    output(int(0.5));     // output int 0
    output(float(0.5));   // output float 0.5
}
```

## 14.2 成员函数的重载、覆盖与隐藏

成员函数的重载、覆盖 (override, 也叫改写) 与隐藏 (也叫遮蔽) 很容易混淆, 程序员必须搞清楚概念, 否则调用结果可能与期望不符。

### 14.2.1 重载与覆盖

成员函数被重载的特征是:

- ✧ 具有相同的作用域 (即同一个类定义中);
- ✧ 函数名字相同;
- ✧ 参数类型、顺序或数目不同 (包括 const 参数和非 const 参数);
- ✧ virtual 关键字可有可无。

覆盖是指派生类重新实现 (或者改写) 了基类的成员函数, 其特征是:

- ✧ 不同的作用域 (分别位于派生类和基类中);
- ✧ 函数名称相同;
- ✧ 参数列表完全相同;
- ✧ 基类函数必须是虚函数。

示例 14-3 的代码中, 函数 `Base::f(int)` 与 `Base::f(float)` 构成重载, 而 `Base::g(void)` 则被 `Derived::g(void)` 覆盖。



示例 14-3

```
#include <iostream.h>
class Base {
public:
    void f(int x){ cout << "Base::f(int) " << x << endl; }
    void f(float x){ cout << "Base::f(float) " << x << endl; }
    virtual void g(void){ cout << "Base::g(void)" << endl; }
};
class Derived : public Base {
public:
    virtual void g(void){ cout << "Derived::g(void)" << endl; }
};
void main(void)
{
    Derived d;
    Base *pb = &d;
    pb->f(42);           // Base::f(int) 42
    pb->f(3.14f);        // Base::f(float) 3.14
    pb->g();              // Derived::g(void)
}
```

- 【提示 14-3】**
- (1) virtual 关键字告诉编译器, 派生类中相同的成员函数应该放到 vtable 中, 并替换基类相应成员函数的槽位;
  - (2) 虚函数的覆盖有两种方式: 完全重写和扩展。扩展是指派生类虚函数首先调用基类的虚函数, 然后再增加新的功能。

### 14.2.2 令人迷惑的隐藏规则

本来仅仅区别重载与覆盖并不算困难, 但是 C++ 的隐藏规则使问题的复杂性陡然增加。这里“隐藏”是指派生类的成员函数遮蔽了与其同名的基类成员函数, 具体规则如下:

- ✧ 派生类的函数与基类的函数同名, 但是参数列表有所差异。此时, 不论有无 virtual 关键字, 基类的函数在派生类中将被隐藏(注意别与重载混淆);
- ✧ 派生类的函数与基类的函数同名, 参数列表也相同, 但是基类函数没有 virtual 关键字。此时, 基类的函数在派生类中将被隐藏(注意别与覆盖混淆)。

有时候我们把这种现象叫做“跨越类边界的重载”, 即程序员可能期望它们构成重载, 然而 C++ 却没有满足他们的要求, 因为构成“重载”的重要标志是“函数位于同一个作用域中”。实际上, 覆盖是一种特殊的隐藏。至于 C++ 为什么最终选择了这种策略, 可以参考 Bjarne Stroustrup 的《The Design and Evolution of C++》3.5 节。

示例 14-4 的类定义中:

- ✧ 函数 Derived::f(float) 覆盖了 Base::f(float);
- ✧ 函数 Derived::g(int) 隐藏了 Base::g(float), 而不是重载;

◇ 函数 `Derived::h(float)` 隐藏了 `Base::h(float)`，而不是覆盖。

示例 14-4

```
#include <iostream.h>
class Base {
public:
    virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
    void g(float x){ cout << "Base::g(float) " << x << endl; }
    void h(float x){ cout << "Base::h(float) " << x << endl; }
};
class Derived : public Base {
public:
    virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
    void g(int x){ cout << "Derived::g(int) " << x << endl; }
    void h(float x){ cout << "Derived::h(float) " << x << endl; }
};
```

很多 C++ 程序员没有意识到有“隐藏”这回事。由于认识不够深刻，“隐藏”的发生可谓神出鬼没，常常产生令人迷惑的结果。

示例 14-5 的程序使用上述类定义，`pb` 和 `pd` 指向同一个对象，按理说运行结果应该是相同的，可事实并非如此。

示例 14-5

```
void main(void)
{
    Derived d;
    Base *pb = &d;
    Derived *pd = &d;
    // 好：程序的行为仅依赖于对象的真实类型
    pb->f(3.14f);    // 动态绑定: Derived::f(float) 3.14
    pd->f(3.14f);    // 动态绑定: Derived::f(float) 3.14
    // 不好：程序的行为依赖于指针的静态类型
    pb->g(3.14f);    // 静态绑定: Base::g(float) 3.14
    pd->g(3.14f);    // 静态绑定: Derived::g(int) 3 (奇怪吗？不奇怪！)
                    // 3.14 强制转换为一个 int 类型临时变量
    // 不好：程序的行为依赖于指针的静态类型
    pb->h(3.14f);    // 静态绑定: Base::h(float) 3.14 (奇怪吗？不奇怪！)
    pd->h(3.14f);    // 静态绑定: Derived::h(float) 3.14
}
```

**【提示 14-4】**

如果你确实想使用所谓的“跨越类边界的重载”，可以在派生类定义中的任何地方显式地使用 `using` 关键字，见示例 14-6。

示例 14-6

```
class Derived : public Base {
```

```
public:
    using Base::g;
    void g(int x){ ... }
};
```

### 14.2.3 摆脱隐藏

隐藏规则引起了不少麻烦。示例 14-7 的代码中，程序员可能期望语句 `pd->f(10)` 调用函数 `Base::f(int)`，但是 `Base::f(int)` 不幸被 `Derived::f(char*)` 隐藏了。由于数字 10 不能被隐式地转换为字符串类型，所以在编译时出错。

示例 14-7

```
class Base {
public:
    void f(int x);
};
class Derived : public Base {
public:
    void f(char *str);
};
void Test(void)
{
    Derived *pd = new Derived;
    pd->f(10);    // error!
}
```

看来隐藏规则似乎很愚蠢，但是隐藏规则的存在至少有两个理由。

(1) 写语句 `pd->f(10)` 的人可能真的想调用 `Derived::f(char*)` 函数，只是他误将参数写错了。有了隐藏规则，编译器就可以明确地指出错误，这未必不是好事，否则，编译器会静悄悄地将错就错，程序员将很难发现这个错误，于是留下祸根。

(2) 假如类 `Derived` 有多个基类（多重继承），有时搞不清楚哪些基类定义了函数 `f`。如果没有隐藏规则，那么 `pd->f(10)` 可能会调用一个出乎意料的基类函数 `f()`。尽管隐藏规则看起来不怎么有道理，但它的确能消除这些意外。

如果语句 `pd->f(10)` 确实想调用函数 `Base::f(int)`，那么有两个办法：其一就是使用 `using` 声明；其二就是把类 `Derived` 修改为示例 14-8 的样子。

示例 14-8

```
class Derived : public Base {
public:
    void f(char *str);
    void f(int x) { Base::f(x); }    // 调用传递
};
```

## 14.3 参数的默认值

有一些参数的值在每次函数调用时都相同，书写这样的语句会使人厌烦。C++语言采用参数的默认值使书写变得简捷（在编译时，默认值由编译器自动插入）。

**【规则 14-1】：** 参数默认值的使用规则：把参数默认值放在函数的声明中，而不要放在定义体中。

例如：

```
void Foo(int x = 0, int y = 0);    // 正确，默认值出现在函数的声明中
void Foo(int x = 0, int y = 0)    // 错误，默认值出现在函数的定义体中
{
    ...
}
```

为什么会这样？我想有两个原因：一是函数的实现（定义）本来就与参数是否有默认值无关，所以没有必要让默认值出现在函数的定义体中；二是参数的默认值可能会改动，显然修改函数的声明比修改函数的定义更为方便。

**【规则 14-2】：** 如果函数有多个参数，参数只能从后向前依次默认，否则将导致函数调用语句怪模怪样。

正确的示例：

```
void Foo(int x, int y = 0, int z = 0);
```

错误的示例：

```
void Foo(int x = 0, int y, int z = 0);
```

对于编译器而言，使用参数默认值和没有使用默认值没有什么本质区别，并没有赋予函数新的功能，仅仅是使书写变得简捷一些。

函数参数的默认值可能会提高函数的易用性，但也可能会降低函数的清晰性。所以我们只能适当地使用参数的默认值，要防止使用不当而产生负面效果。示例 14-9 的代码中，不恰当地使用参数的默认值从而导致重载函数 `output` 产生二义性。

示例 14-9

```
#include <iostream.h>
void output( int x );
void output( int x, float y = 0.0 );
void output( int x )
{
    cout << " output int " << x << endl ;
}
void output( int x, float y )
```

```

{
    cout << " output int " << x << " and float " << y << endl ;
}
void main(void)
{
    int x = 1;
    float y = 0.5;
    output(x);           // 错误！模棱两可的调用
    output(x, y);        // 输出整型数 1 和实型数 0.5
}

```

## 14.4 运算符重载

### 14.4.1 基本概念

在 C++语言中,可以用关键字 `operator` 加上运算符来表示函数,叫做**运算符重载**。运算符重载函数是一种特殊形式的函数,即运算符本身就是函数名,并且不改变它们作为内置运算符时的使用方法。例如两个复数相加函数:

```
Complex Add(const Complex& a, const Complex& b);
```

可以用运算符重载来定义:

```
Complex operator +(const Complex& a, const Complex& b);
```

运算符与普通函数在调用时的不同之处在于:对于普通函数,实参出现在圆括号内;而对于运算符,实参出现在其两侧(或一侧)。例如:

```

Complex a, b, c;
...
c = Add(a, b);    // 调用普通函数
c = a + b;        // 调用重载的运算符 "+"

```

如果运算符被重载为全局函数,那么只有一个参数的运算符叫做**一元运算符**,有两个参数的运算符叫做**二元运算符**。

如果运算符被重载为类的成员函数,那么一元运算符没有参数(但是++和--运算符的后置版本除外),二元运算符只有一个右侧参数,因为对象自己成了左侧参数。

从语法上讲,运算符既可以定义为全局函数,也可以定义为成员函数。文献[Murray]中 p44~p47 对此问题做了较多的阐述,并总结了如表 14-1 所示的规则。

表 14-1 运算符的重载规则

运 算 符	规 则
所有的一元运算符	建议重载为非静态成员函数
=、()、[]、->、*	只能重载为非静态成员函数

(续表)

运 算 符	规 则
$+=$ 、 $-=$ 、 $/=$ 、 $*=$ 、 $&=$ 、 $ =$ 、 $\sim=$ 、 $\%=$ 、 $>>=$ 、 $<<=$	建议重载为非静态成员函数
所有其他运算符	建议重载为全局函数

只能重载为成员函数的运算符确保了调用它们的对象（左边操作数）必须是重载了它们类的对象，否则重载就没有意义了。

不能重载为成员函数的运算符一般是因为发起调用的对象（左边操作数）并非必须是重载它们类的对象，或者是这些运算符的特殊性质（例如左右操作数可互换）使然。

**【提示 14-5】** C++语言支持函数重载，所以才能将运算符当成函数来用，C语言就不行了。我们要以平常心来对待运算符重载：

- (1) 不要过分担心自己不会用，它的本质仍然是程序员熟悉的函数。
- (2) 不要过分热心地使用，如果它不能使代码变得更加易读易写，那就别用，否则会自找麻烦。

运算符重载是支持数据抽象和泛型编程的利器。比如，一般说来，凡是用作容器元素类型的 `class`（包括 `struct`）都需要重载 “=”、“==” 和 “<” 等运算符，因为容器可能会使用它们来排序或者拷贝元素，某些泛型算法都是以假设元素类型已经重载了 “=” 和 “<” 为前提的。

## 14.4.2 运算符重载的特殊性

C++/C 都有一套固定的运算符集合，其中不乏比较特殊的运算符例如 “++” 和 “--”。虽说本质上运算符重载和函数重载没有什么不同，但是还是有一些特殊性需要注意：

- 【提示 14-6】**
- (1) 如果重载为成员函数，则 `this` 对象发起对它的调用。
  - (2) 如果重载为全局函数，则第一个参数发起对它的调用。
  - (3) 禁止用户发明该语言运算符集合中不存在的运算符。
  - (4) 除了函数调用运算符 “( )” 外，其他运算符重载函数不能有默认参数值。
  - (5) 不要试图改变重载运算符的语义，要与其内置语义保持一致。如果你把 “++” 运算符重载后让它执行递减操作，重载 “<<=” 和 “>>=” 执行 I/O 操作等，都会损害程序的可理解性，得不偿失。
  - (6) 某些运算符之间可以互相推导，比如在逻辑运算符之间和关系运算符之间，这样我们可以只实现少数几个运算符，然后再用它们来实现其他运算符。例如，我们可以用 “!” 和 “&&” 来实现 “||”（“`a&& b`” 等价于 “`!(a||b)`”）；我们也可以利用 “<” 来实现 “==”、“>”、“>=”、“<=”、“!=”（“`a== b`” 与 “`!(a<b)&&!(a>b)`” 等价），或者单独实现 “==”，然后和 “<” 组合来实现其他运算符等。

### 14.4.3 不能重载的运算符

在 C++运算符集合中,有一些运算符是不允许重载的,一方面是因为它们的右侧操作数是一个名字(比如成员名)而不是对象,另一方面是出于安全性的考虑,可防止产生错误和混乱。

- ✧ 不能重载“.”,因为它在 ADT/UDT 中对任何成员都有意义,已成为标准用法。
- ✧ 不能重载反引用类成员指针“.\*”。
- ✧ 不能重载作用域解析运算符“::”。
- ✧ 不能重载那个唯一的三元运算符——条件运算符“?:”。
- ✧ 不能重载 sizeof()和 typeid()。
- ✧ 不能重载 C++的新式类型转换运算符: static\_cast<>、dynamic\_cast<>、const\_cast<>、reinterpret\_cast<>。
- ✧ 不能重载“#”和“##”等预处理操作符。

### 14.4.4 重载++和--

这两个运算符令许多人迷惑不解,关键问题是它们都存在两个版本:前置版本和后置版本。许多教科书中是这样讲述“++”和“--”运算符的语义的:

“++”的前置版本表示先对它的操作数执行“+1”运算,然后再使用它的值;“++”的后置版本表示先使用其操作数的值,然后再对它的操作数执行“+1”运算。“--”的语义类似。

按照这种解释,那么可以推出下面的等价关系:

<code>int b = ++a;</code>	$\Leftrightarrow$	<code>a += 1; int b = a;</code>
---------------------------	-------------------	-------------------------------------

而:

<code>int b = a++;</code>	$\Leftrightarrow$	<code>int b = a; a += 1;</code>
---------------------------	-------------------	-------------------------------------

显然,第二个等价关系是错误的,可见教科书对“++后置版本”的语义解释并不完全正确。

我们知道,“++”的优先级要高于“=”,并且“a++”是一个完整的表达式,而“int b = a++;”是一个复合表达式语句,因此“a++”的计算要优先于“=”,即 b 应该是使用表达式“a++”的返回值来初始化才对。而表达式“a++”的值就等于 a,即:

<code>int b = a++;</code>	$\Leftrightarrow$	<code>int temp = a; a += 1; int b = temp; temp.~int();</code>
---------------------------	-------------------	---

在 C++程序中,你可能要为某些类重载“++”和“--”运算符,正确理解它们的语义是非常重要的。C++标准规定:当为一个类型重载“++”/“--”的前置版本时,

不需要参数；当为一个类型重载 “++” / “--” 的后置版本时，需要一个 `int` 类型的参数作为标志（即哑元，非具名参数）。至于为什么会是这种样子，而不是通过一个特殊的关键字来标识这种位置关系，请参考《The Design and Evolution of C++》。

我们举一个例子来说明如何重载这两个运算符（见示例 14-10）。

示例 14-10

```
class Integer {
public:
    Integer(long data) : m_data(data) {}
    Integer& operator++() {           // 前置版本：返回引用
        cout<< "Integer::operator++() called!" << endl;
        m_data++;
        return *this;
    }
    Integer operator++(int) {         // 后置版本：返回对象的值
        cout<< "Integer::operator++(int) called!" << endl;
        Integer temp = *this;
        m_data++;                    // 或： ++(*this);
        return temp;                 // 返回 this 对象的旧值
    }
    // 其他成员
    ...
private:
    long m_data;                     // 对 long 的封装
};

void main(void)
{
    Integer x = 1;                    // call Integer(long)
    ++x;                              // call operator++()
    x++;                              // call operator++(int)
}

输出结果：
Integer::operator++() called!
Integer::operator++(int) called!
```

## 【提示 14-7】

当 “++” / “--” 应用于基本类型数据时，前置版本和后置版本在效率上没有多大差别。然而，当应用于用户定义类型，尤其是大对象的时候，前置版本就会比后置版本的效率高许多。后置版本总是要创建一个临时对象，在退出函数时还要销毁它，而且返回临时对象的值时还会调用其拷贝构造函数。所以，如果你可以选择的话，请尽量使用前置版本。



## 14.5 函数内联

### 14.5.1 用函数内联取代宏

C++ 语言支持函数内联，其目的是为了提<sub>高</sub>函数的执行效率（速度）。

在 C 程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但使用起来像函数。编译预处理器用拷贝宏代码的方式取代函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高了速度。使用宏代码最大的缺点是容易出错，预处理器在拷贝宏代码时常常产生意想不到的边际效应。例如：

```
#define MAX(a, b) (a) > (b) ? (a) : (b)
```

语句：

```
result = MAX(i, j) + 2;
```

将被预处理器扩展为：

```
result = (i) > (j) ? (i) : (j) + 2;
```

由于运算符“+”比运算符“?:”的优先级高，所以上述语句并不等价于期望的：

```
result = ((i) > (j) ? (i) : (j)) + 2;
```

如果把宏代码改写为：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

则可以解决由优先级引起的错误。但是即使使用修改后的宏代码也不是万无一失的，例如语句：

```
result = MAX(i++, j);
```

将被预处理器解释为：

```
result = (i++) > (j) ? (i++) : (j); // 在同一个表达式中 i 被两次求值
```

宏的另一个缺点就是不可调试，但是内联函数是可以调试的。内联函数不是也像宏一样进行代码展开吗？怎么能够调试呢？其实内联函数的“可调试”不是说它展开后还能调试，而是在程序的调试（Debug）版本里它根本就没有真正内联，编译器像普通函数那样为它生成含有调试信息的可执行代码。在程序的发行（Release）版本里，编译器才会实施真正的内联。有的编译器可以设置函数内联开关，例如 Visual C++。

对于 C++ 而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。

让我们看看 C++ 的“函数内联”是如何工作的：

**【提示 14-8】** 对任何内联函数，编译器在符号表里放入函数的声明，包括名字、参数类型、返回值类型（符号表是编译器用来收集和保存字面常量和某些符号常

量的地方)。如果编译器没有发现内联函数存在错误,那么该函数的代码也被放入符号表里。在调用一个内联函数时,编译器首先检查调用是否正确(进行类型安全检查,或者进行自动类型转换,当然对所有的函数都一样)。如果正确,内联函数的代码就会直接替换函数调用语句,于是省去了函数调用的开销。这个过程与预处理器有显著的不同,因为预处理器不能进行类型安全性和自动类型转换。假如内联函数是成员函数,对象的地址(this)会被放在合适的地方,这也是预处理器办不到的。

C++语言的函数内联机制既具备宏代码的效率,又增加了安全性,而且可以自由操作类的数据成员。所以在C++程序中应该尽量用内联函数来取代宏代码,断言assert恐怕是唯一的例外。assert是仅在Debug版本中起作用的宏,它用于检查“不应该”发生的情况。为了不在程序的Debug版本和Release版本之间引起差别,assert不应该产生任何副作用。如果assert是函数,由于函数调用会引起内存、代码的变动,那么将导致Debug版本与Release版本存在某些差异,这并非我们所期望的。所以assert不是函数,而是宏(参见6.12节“使用断言”)。

内联函数的另一个优点就是:函数被内联后,编译器就可以通过上下文相关的优化技术对结果代码执行更深入的优化,而这种优化在普通函数体内是无法单独进行的,因为一旦进入函数体内它也就脱离了调用环境的上下文。

## 14.5.2 内联函数的编程风格

**【提示14-9】:** 关键字inline必须与函数定义体放在一起才能使函数真正内联,仅把inline放在函数声明的前面不起任何作用。

如下风格的函数Foo不能成为内联函数:

```
inline void Foo(int x, int y);    // inline 仅与函数声明放在一起
void Foo(int x, int y)
{
    ...
}
```

而如下风格的函数Foo则会成为内联函数:

```
void Foo(int x, int y);
inline void Foo(int x, int y)    // inline 与函数定义体放在一起
{
    ...
}
```

所以说,inline是一种“用于实现的关键字”,而不是一种“用于声明的关键字”。一般情况下,用户可以阅读函数的声明,但是却看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了inline关键字,但作者认为inline不应该出现在函数的声明中。这个细节虽然不会影响函数的功能,但是体现了高质量C++/C程序设计风格的一个基本原则:声明与定义不可混为一谈,用户没有必要也

不应该知道函数是否需要内联。

定义在类声明之中的成员函数将自动地成为内联函数，例如：

```
class A {  
public:  
    void Foo(int x, int y) { ... }    // 自动地成为内联函数  
};
```

但是编译器是否会将它真正内联则要看 Foo 函数如何定义。

### 14.5.3 慎用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数呢？

如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

**【提示 14-10】** 内联不是万灵丹，它以代码膨胀（拷贝）为代价，仅仅省去了函数调用的开销，从而提高程序的执行效率。注意：这里所说的“函数调用开销”并不包括执行函数体所需要的开销，而是仅指参数压栈、跳转、退栈和返回等操作。如果执行函数体内代码的时间比函数调用的开销大得多，那么 inline 的效率收益会很小。另一方面，每一处内联函数的调用都要拷贝代码，将使程序的总代码量增大，消耗更多的内存空间。

以下情况不宜使用内联：

- ◇ 如果函数体内的代码比较长，使用内联将导致可执行代码膨胀过大。
- ◇ 如果函数体内出现循环或者其他复杂的控制结构，那么执行函数体内代码的时间将比函数调用的开销大得多，因此内联的意义并不大。

**【提示 14-11】** 不少人误以为让类的构造函数和析构函数成为内联函数更有效。殊不知，构造函数和析构函数会隐藏一些行为，如“偷偷地”调用基类或成员对象的构造函数和析构函数。所以不要轻易让构造函数和析构函数成为内联函数。

实际上，inline 在实现的时候就是对编译器的一种请求，因此编译器完全有权利取消一个函数的内联请求。一个好的编译器能够根据函数的定义体，自动取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中），或者自动地内联一些没有 inline 请求的函数。因此，编译器往往选择那些短小而简单的函数来内联（内联候选函数）。

## 14.6 类型转换函数

**【提示 14-12】** 类型转换的本质是创建新的目标对象，并以源对象的值来初始化，所以源对象没有丝毫改变。不要把类型转换理解为“将源对象的类型转换为目标类型”。

从某种意义上讲, 类的构造函数提供了一种类型转换的手段, 因此我们可以把带有一个参数的构造函数看做一种类型转换函数 (见示例 14-11)。

示例 14-11

<pre>class Point{ public:     Point(float x);     ... private:     float m_x; };</pre>	<pre>class Point2D : public Point{ public:     Point2D(const Point&amp; p);     ... private:     float m_y; };</pre>	<pre>class Point3D : public Point2D{ public:     Point3D(const Point2D&amp; p2d);     ... private:     float m_z; };</pre>
--	--	--

Point 的构造函数以一个 float 数为参数来构造一个 Point 对象, 如:

```
Point p1 = 10.5; // 直接调用 Point::Point(float)
p1 = 20.5;       // 先调用 Point::Point(float)把 20.5 转换成一个 Point 临时对象,
                // 然后再调用默认的 operator=()完成拷贝赋值
```

因此, Point::Point(float x)就可以看做一个类型转换函数, 当我们在一个表达式中混合使用 Point 对象和 float 数时就会自动完成转换 (编译器帮我们调用它)。同理, Point2D 的构造函数 Point2D::Point2D(const Point&)可以将一个 Point 对象自动转换为一个 Point2D 对象, Point3D 的构造函数 Point3D::Point3D(const Point2D&)可以把一个 Point2D 对象自动转换为一个 Point3D 对象。

由于你不能把基类对象直接转换为派生类对象 (无论是直接赋值还是强制转换, 因为这不是“自然的”), 所以通过构造函数来完成这一转换成了必然的选择。但是有时候通过构造函数进行的隐含类型转换可能会损害程序的可理解性, 因为它可能执行用户本不期望的转换却不让用户察觉。例如:

```
class MyString {
public:
    MyString(size_t size, char c = '\0');
    //...
private:
    char *m_data;
};
```

这个构造函数的本意是: 构造一个长度为 size 的字符串, 并用字符 c 来初始化其内存单元。但是它无形中起了类型转换的作用, 把一个 size\_t 类型的整数转换为 MyString 对象。例如:

```
MyString strName = 20;
strName = 40;      // 莫名其妙!
```

尤其是当定义这样的函数的时候:

```
void f(MyString str);
```

对于如下的调用:

```
f(100);
```

如果当前程序中没有其他重载的 `f()` 函数，编译器就会将它暗中转换为：

```
f(MyString(100));
```

当你的类定义中出现类似的情况时，可以在构造函数前面添加关键字 `explicit` 将其声明为显式的，意即要求用户必须显式地调用该构造函数来初始化对象，以明确表明他的意图。

构造函数可以把其他类型对象转换为 `this` 对象（向内转换）。C++ 还提供了相反方向的转换手段，即自定义类型转换运算符，它可以把 `this` 对象转换为其他类型对象（向外转换或数据萃取），见示例 14-12。

示例 14-12

<pre>class Point { public:     Point(float x);     operator float() const  /**     { return m_x; }      ... private:     float m_x; };</pre>	<pre>class Integer { public:     Integer(const long lng);     operator long() const; /**     { return m_data; }      ... private:     long m_data; };</pre>
--	---

自定义类型转换运算符的规则如下：

**【规则 14-3】** 类型转换运算符定义以 `operator` 关键字开始，紧接着目标类型名和 `()`。它没有参数，实际上 `this` 就是参数；也没有返回类型，实际上函数名就是返回类型。类型转换运算符只能定义为非静态成员函数。

为一个类定义类型转换运算符的好处在于，该类型的对象可以直接用在需要目标类型的地方，编译器自动调用转换运算符来完成转换。例如：

```
Point    p2(100.25);
Integer  aInt(100);
cout << p2 << endl;      // 100.25
cout << aInt << endl;     // 100
```

这样，我们就不需要为 `Point` 和 `Integer` 重载 “<<” 运算符了。

与带参数的构造函数类似，自定义类型转换运算符也可能导致二义性和损害程序的可理解性，特别是当你为一个类同时提供了这两个转换函数的情况下，因此要谨慎使用。

此外，C++ 新增了 4 个类型转换运算符，它们是：

- ◇ `static_cast<dest_type>(src_obj)`，作用相当于 C 风格的强制转换，但是在多重继承的情况下，它会正确地调整指针的值，而 C 风格的强制转换则不会调整；它可以遍历继承树来确定 `src_obj` 与 `dest_type` 的关系，但是只在编译时

进行（此所谓静态）；如果使用它来做 `downcast` 操作，则会存在隐患。

- ✧ `const_cast<dest_type>(src_obj)`，用于去除一个对象的 `const/volatile` 属性。
- ✧ `reinterpret_cast<dest_type>(src_obj)`，我们可以借助它把一个整数转换成一个地址，或者在任何两种类型的指针之间转换。使用该运算符的结果很危险，请你不要轻易使用。
- ✧ `dynamic_cast<dest_type>(src_obj)`，在运行时遍历继承树（类层次结构）来确定 `src_obj` 与 `dest_type` 的关系，具体在本书第 15 章讲述。

**【提示 14-13】** 在 C++ 程序中尽量不要再使用 C 风格的类型转换，除非源对象和目标类型都是基本类型的对象或指针，否则很不安全。C++ 的类型转换运算符在需要的时候会进行指针调整，因此结果比较安全。不过，如果不了解其用法就胡乱使用就不安全了。请参考编译器帮助文档。

## 14.7 const 成员函数

任何不会修改数据成员的成员函数都应该声明为 `const` 类型。如果在编写 `const` 成员函数时不慎写下了试图修改数据成员的代码，或者调用了其他非 `const` 成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

见示例 14-13 的程序中，类 `stack` 的成员函数 `GetCount` 仅用于计数，从逻辑上讲 `GetCount` 应当为 `const` 函数。编译器将指出 `GetCount` 函数中的错误。

示例 14-13

```
class Stack {
public:
    void    Push(int elem);
    int     Pop(void);
    int     GetCount(void) const; // const 成员函数
private:
    int     m_num;
    int     m_data[100];
};

int Stack::GetCount(void) const
{
    ++m_num;                // 编译错误，企图修改数据成员 m_num
    Pop();                  // 编译错误，企图调用非 const 成员函数
    return m_num;
}
```

`const` 成员函数的声明看起来怪怪的：`const` 关键字只能放在函数声明的尾部，大概是因为其他地方都已经被占用了。

**【提示 14-14】:** 不要混淆 `const` 成员函数和成员函数返回 `const` 类型。实际上，这两者之间没有必然联系。也就是说，`const` 成员函数并非一定要返回 `const` 类型，非 `const` 成员函数也并非必须返回非 `const` 类型。

如果不恰当地定义和使用 `const` 成员函数，编译器会报告一大堆的错误，有时候不一定能清楚地理出个头绪来。

**【提示 14-15】:** `static` 成员函数不能定义为 `const` 的。这是因为 `static` 成员函数只是全局函数的一个形式上的封装，而全局函数不存在 `const` 一说；何况 `static` 成员函数不能访问类的非静态成员（没有 `this` 指针），修改非静态数据成员又从何说起呢？！

`const` 成员函数的访问规则如图 14-1 所示。

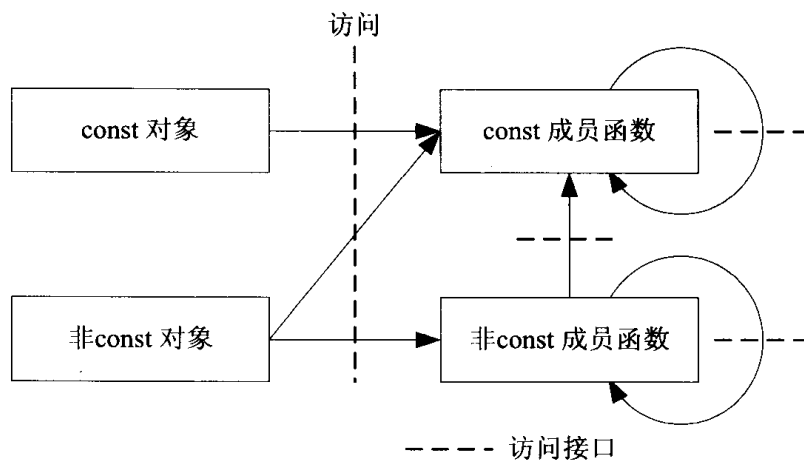


图 14-1 `const` 成员函数访问规则

## 一些心得体会

C++语言中的重载、默认参数、内联、隐式转换等机制展现了很多优点，但是这些优点的背后都隐藏着一些隐患。但是不能消极躲避，越是不学不用，就只会停留在“C++中的 C”的水平上。正如人们的饮食，少食和暴食都不可取，应当恰到好处。我们要辩证地看待 C++的新机制，应该恰如其分地使用它们，虽然这会让我们编程时多费一些心思，少了一些痛快，但这才是编程的艺术。





## 第 15 章 C++异常处理和 RTTI

健壮性是指软件在异常环境下仍然能够正常运行的能力，健壮性是非常重要的软件质量属性。提高 C++ 应用软件健壮性的基本手段之一就是使用异常处理技术。

C++ 的异常处理技术是一种面向对象的运行时错误处理机制，其思路完全不同于 C 语言旧式的返回值检查策略。本章主要探讨 C++ 异常处理机制的构成及其使用策略，并介绍一些 RTTI (Run-time Type Identification) 的知识。

### 15.1 为什么要使用异常处理

如果不处理运行时可能发生的各种异常情况，很难想象这样的软件怎么能放心地卖出去。人生病了总是通过吃药、打针或者手术来治疗的，然而，人们都明白“预防胜于治疗”的道理。这个道理应用到软件开发之中，那就是预防运行时错误的发生，即在错误即将发生前通过检测触发它的条件来阻止它，从而防止造成危害，努力使软件正常运行。

我们知道，C 语言没有内置的运行时错误处理机制，实际上早期的 C++ 也没有。它们无非是使用如下 3 种方法来处理可能的运行时错误：

- ◇ 函数返回彼此协商后统一定义的状态编码来表示操作成功、失败或其他不同类型的错误信息；
- ◇ 使用全局变量来保存错误编码，每一个使用到它的函数在开始的时候都检查它的值，并且每一个函数的操作结果都写到这个全局变量中，例如用 `errno` 表示任何一个函数调用返回后产生的错误码；
- ◇ 出错时终止程序运行。

显然，这些传统的错误处理方法存在很大的缺点：

(1) 状态编码和错误编码难以形成统一的标准，例如同样用途的程序库，不同的开发商就可能采用不同的错误编码方案来标识不同类型的错误。

(2) 这种方法将正常代码和错误处理程序紧紧地绑在了一起，虽然能够精确地掌握程序执行的流程，但是要求程序员在调用每一个具有返回错误码的函数后都要立刻检查其返回值并输出用户可理解的错误信息（有些函数可能会有多个返回值），可以想象这是一项很烦人的工作，我想每个程序员都有这种体会。这种方法不仅会增大程序控制的复杂度，而且会增大代码体积，甚至降低程序的可读性。

(3) 有些函数根本就没有返回值，例如构造函数和析构函数，那么怎样才能检测到对象构造或析构失败的情况？虽然在实际应用中对象构造失败的概率很小，但是对于一套完整的语言系统来说，它必须解决这个问题。

(4) 如果使用全局的错误变量，那么不仅要在函数的开头检查它，有时还要把它清零，这是很容易被人疏忽的。

(5) 在发生错误的时候使用 `exit()` 或 `abort()` 粗暴地结束程序运行，这是用户很不愿意接受的，除非万不得已。尤其是在金融、军事、生命保障等关键系统中的软件，不能因为运行时有错误就停止工作；

.....

为了解决这些问题，Bjarne Stroustrup 在设计 C++ 异常处理机制的时候就想达到下列目标：

- ◇ 允许从异常抛出点把任意数量的信息以类型安全的方式传递到异常处理器；
- ◇ 对于没有抛出异常的代码段，它不会带来任何额外的时间和空间开销；
- ◇ 保证所抛出的任何一个异常都能够被适当的处理器捕获；
- ◇ 通过一种组合方式可以使程序员写出处理一组异常的处理器；
- ◇ 能够直接应用到多线程程序中；

.....

## 15.2 C++ 异常处理

### 15.2.1 异常处理的原理

传统的错误处理是用不同的数值来表示不同类型的错误，其表达能力很有限，因为一个数字包含的信息量太少。而 C++ 异常处理机制将异常类型化，显然一个类型要比一个数字包含的信息量大得多。

比如我们常用的函数 `fopen()`，当打开文件失败时返回 `NULL`。按照传统的错误处理方法，在调用 `fopen()` 后立即检查其返回值，如果为 `NULL` 就进行错误处理。如果将返回 `NULL` 改为抛出异常 `OpenFailed`，那么我们就不用在调用 `fopen()` 后马上检查返回值，而是在调用函数内部或者更高层的调用者那里设置异常处理器来捕获这个异常。C++ 保证：如果一个异常在抛出点没有得到处理，那么它将一直被抛向上层调用者，直至 `main()` 函数，直到找到一个类型匹配的异常处理器，否则调用 `terminate()` 结束程序。

可以看出：异常处理机制实际上是一种运行时通知机制。示例 15-1 是一个简单的例子。

示例 15-1

```
class DevidedByZero{ };
double Devide(double a, double b)
{
    if (abs(b) < std::numeric_limits<double>::epsilon())
        throw DevidedByZero();    // 提前检测异常发生条件并抛出自定义异常
```

```

        return a / b;                                // 这才是可能真正发生错误的地方!
    }
    void test()
    {
        double x = 100, y = 20.5;
        try {
            cout << Devide(x, y) << endl;    // 可能抛出异常 DevidedByZero
        }
        catch(DevidedByZero&) {
            cerr << "Devided by zero!" << endl;
        }
    }
}

```

从这个例子可以看出异常处理机制的本质:

**【提示 15-1】:** 在真正导致错误的语句即将执行之前, 并且异常发生的条件已经具备时, 使用我们自定义的软件异常(异常对象)来代替它, 从而阻止它。因此, 当异常抛出时, 真正的错误实际上并未发生。

在一个大型的层次结构模式的软件系统中, 上层的功能最终都是通过调用底层 API 实现的。通常, 运行时错误几乎都发生在底层 API 调用上, 例如计算结果溢出、事务处理失败等。通常底层 API 只知道这是一些很低级的错误, 但是它不知道如何与上层应用领域的逻辑错误对应起来。你不能直接在用户界面上显示“结果上溢”或“事务失败”这样的错误信息, 而应该把这样的底层错误“翻译”为类似“燃油不足, 发动机即将关闭”和“该账户余额不足, 无法转账”这样的信息, 否则用户会感到莫名其妙的。但是高层组件如何知道下层发生了什么错误呢? 这就需要底层代码通知高层组件“这里发生了 XXX 错误”, 通知的方法就是抛出一个异常对象。C++异常处理技术可以使用户的正常代码与错误处理代码分离开来。

## 15.2.2 异常类型和异常对象

任何一种类型都可以当做异常类型, 因此任何一个对象都可以当做异常对象, 包括基本数据类型的变量、常量、任何类型的指针、引用、结构等, 甚至空结构或空类的对象。这是因为异常仅仅通过类型而不是通过值来匹配的, 否则就又退回到了传统的错误处理技术上, 见示例 15-2。

示例 15-2

<pre> try {     if(failed) throw 0; } catch(int) {     cerr &lt;&lt; "exception!" &lt;&lt; endl; } </pre>	<pre> try {     if(failed) throw "an exception occurred!"; } catch(const char *) {     cerr &lt;&lt; "exception!" &lt;&lt; endl; } </pre>
---	---

但是我们常常不直接使用基本数据类型的对象作为异常, 因为它们表示异常类

型的能力不足；相反，总是自定义一些异常类来具体描述我们需要的异常类型（见示例 15-3）。

示例 15-3

```
class DevidedByZero {
public:
    DevidedByZero(const char *p);
    const char* description();
    //...
private:
    char * desp;
};

double Devide(double a, double b)
{
    if(abs(b) < std::numeric_limits<double>::epsilon())
        throw DevidedByZero("The divisor is 0.");
    return a / b;
}

void test()
{
    double x = 100, y = 20.5;
    try {
        cout << Devide(x, y) << endl;
    }
    catch(const DevidedByZero& ex) {
        cerr << ex.description() << endl;
    }
}
```

## 15.2.3 异常处理的语法结构

异常处理也是一种程序控制结构，它包括 4 个组成部分：抛出异常、提炼异常、捕获异常及异常对象本身。它们对应了 C++ 语言的 3 个关键字及其构造，即 `throw`、`try{}、catch{}。``throw` 只是一条语句，而 `try` 和 `catch` 各自引导着一个程序块，即使它们是空的。`try{}块` 包含了可能会有异常抛出的代码段，而 `catch{}块` 则包含用户定义的异常处理代码，即异常处理器（handler）。一条 `throw` 语句只能抛出一个异常，一个 `catch` 子句也只能捕获一种异常。需要注意的是：异常抛出点常常和异常捕获点距离很远，异常抛出点可能深埋在底层软件模块内，而异常捕获点常常在高层组件中；异常捕获却必须和异常提炼（`try` 块）结合使用，并且可以通过异常组合在一个地点捕获多种异常。见示例 15-4。

示例 15-4

```
class OverFlow{};    // 空类作为异常类
class UnderFlow{};
```

```
void test(int index)
{
    double x = 100, y = 20.5;
    int x[20] = { 0 };
    try {
        cout << Devide(x, y) << endl;           // 可能抛出异常 DevidedByZero
        if(index >= 20) throw OverFlow();        // 抛出点位于当前 try 块内
        if(index < 0) throw UnderFlow();         // 抛出点位于当前 try 块内
    }
    catch(const DevidedByZero& ex) {
        cerr << ex.description() << endl;
    }
    catch(const OverFlow&) {                     // 省略参数名称
        cerr << "Overflow occurred!" << endl;
    }
    catch(const UnderFlow&) {
        cerr << "Underflow occurred!" << endl;
    }
    catch(...) {                                 // 捕获其他所有可能的异常
        cerr << "Unexpected exception occurred!" << endl;
    }
}
```

一个 `catch()` 子句就相当于带有一个参数的函数，而 `throw` 语句则相当于函数调用语句。虽然如此，但是它们的真正行为并非像一个函数一样。`catch` 处理完后不会返回到 `throw` 语句那里，而是要么退出函数，要么执行后面的正常代码。因此，`throw` 语句在行为上更像一个 `goto` 语句。

**【提示 15-2】:**

(1) 异常抛出点位于 `try` 块内有两种情况：`throw` 语句位于当前 `try` 块内；或者含有异常抛出点的函数被上层函数调用，而上层函数调用者位于 `try` 块内。当可能抛出异常的代码没有位于 `try` 块内，或者即使位于 `try` 块内但是当前没有类型匹配的 `catch` 块，则该异常将继续向上层调用者抛出，并且当前函数从异常抛出点退出，于是此后的语句都不会被执行。

(2) 在一个函数内尽量不要出现多个并列的 `try` 块，也不要使用嵌套的 `try` 块，否则不仅会导致程序结构复杂化，增加运行时的开销，而且容易出现逻辑错误。

每一个 `try` 块后面必须至少跟一个 `catch` 块。当异常抛出时，C++ 异常处理机制将从碰到的第一个 `catch` 块开始匹配，直到找到一个类型符合的 `catch` 块为止，紧接着执行该 `catch` 块内的代码。当异常处理完毕后，将跳过后面的系列 `catch` 块，接着执行后面的正常代码。从这一点上看，组合在一起的 `catch` 块类似于嵌套的 `if...else...` 结构。

(3) 对于每一个被抛出的异常，你总能找到一个对应的 `throw` 语句，只是有些是位于我们的程序之中，而另一些则位于标准库之中，这就是我

们常常能 catch 到一个异常却看不到它在哪里被 throw 的原因。

(4) 由于异常处理机制采用类型匹配而不是值判断, 因此 catch 块的参数可以没有参数名称, 只需要参数类型, 除非确实要使用那个异常对象。

由于异常处理机制只有在程序运行起来后确有异常抛出的时候才起作用, 因此如果运行时没有任何异常抛出, 那么异常处理机制不会给程序增加任何额外负担。

**【提示 15-3】** 虽然异常对象看上去像局部对象, 但是它并非创建在函数堆栈上, 而是创建在专用的异常堆栈上, 因此它才可以跨接多个函数而传递到上层, 否则在堆栈清退的过程中就会被销毁。不要企图把局部对象的地址作为异常对象抛出, 因为局部对象会在异常抛出后函数堆栈清退的过程中被销毁。

## 15.2.4 异常的类型匹配规则

C++异常处理机制必须保存每一个 throw 语句抛出的异常对象的类型信息和每一个 catch 子句的参数类型信息, 其目的就是在运行时执行异常对象与异常处理器的类型匹配。那么一个对象按照什么规则和一个类型进行匹配呢?

C++规定: 当一个异常对象和 catch 子句的参数类型符合下列条件时, 匹配成功:

- ✧ 如果 catch 子句参数的类型就是异常对象的类型或其引用;
- ✧ 如果 catch 子句参数类型是异常对象所属类型的 public 基类或其引用;
- ✧ 如果 catch 子句参数类型为 public 基类指针, 而异常对象为派生类指针;
- ✧ catch 子句参数类型为 void\*, 而异常对象为任何类型指针;
- ✧ catch 子句为 catch-all, 即 catch(...).

## 15.2.5 异常说明及其冲突

在使用了 C++异常处理机制的环境中, 应当使用函数异常说明, 有两个基本原因:

(1) 清楚地告诉函数的调用者, 该函数可能会抛出哪些类型的异常, 以使用户能够编写合适的异常处理器;

(2) 用户一般无法看到函数的实现 (例如库函数), 因此用户只能浏览函数原型才能知道一个函数可能会抛出哪些类型的异常。

函数异常说明如示例 15-5 所示。

示例 15-5

```
double Devide(double x, double y) throw(DevidedByZero); // (1) 只可能抛出一种异常
bool func(const char *) throw(T1, T2, T3);             // (2) 可能抛出 3 种异常
void g() throw();                                       // (3) 不抛出任何异常
void k();                                               // (4) 可能抛出任何异常, 也可能不抛出任何异常
```

注意异常说明的两种极端情况: 不抛出任何异常和可以抛出任何异常。前者就是使用空列表的 throw() 语句, 如本例中的 (3) 所示; 后者没有任何说明, 如本例中

的 (4) 所示。如果不加任何说明, 就表示可能抛出任何异常, 当然也可能不抛出任何异常, 这就会使调用者不知所措!

使用函数异常说明的好处是: 不仅可以约束函数的实现者, 防止抛出异常说明列表中没有说明的异常, 而且可以指导函数的调用者编写正确的异常处理程序。

#### 【提示 15-4】:

(1) 函数原型中的异常说明要与其实现中的异常说明一致, 否则容易引起异常冲突。由于异常处理机制是在运行时有异常抛出时才发挥作用的, 因此如果函数的实现中抛出了没有在其异常说明列表中列出的异常, 则编译器并不能检查出来。但是当运行时如果真的抛出了这样的异常, 就会导致异常冲突。因为你没有提示函数的调用者: 该函数会抛出一种没有被说明的即不期望的异常, 于是异常处理机制就会检测到这个冲突并调用标准库函数 `unexpected()`。`unexpected()` 的默认行为就是调用 `terminate()` 来结束程序。

(2) 显然, 我们要尽量避免运行时的异常冲突。但是在大型软件开发过程中要绝对避免是不太可能的, 此时可以在程序的开始处使用标准库函数 `set_unexpected()` 来预设一个回调函数, 当发生异常冲突或者未捕获异常时系统就会自动调用该函数来执行我们定制的异常处理功能。实际上, `set_unexpected()` 使用自定义的异常处理功能取代标准库函数 `unexpected()` 的默认行为, 即把调用 `terminate()` 函数改为调用用户提供的回调函数。该函数的使用方法请参考编译器帮助文档。

### 15.2.6 当异常抛出时局部对象如何释放

`throw` 语句看上去像是一个 `goto` 语句, 因此它本质上也是一种程序控制手法。在正常情况下, 每一个函数在结束前一刻都会调用其创建在堆栈上的每一个对象的析构函数来释放其占有的资源。但是, 当函数执行过程中抛出异常时, 要么进入当前函数的一个 `catch` 块, 要么将异常对象传递到上层调用者, 同时当前函数结束, 那么该函数内的局部对象该如何处置呢?

Bjarne Stroustrup 已经考虑到了这一点, 他引入了 “resource acquisition is initialization” 思想 (在初始化阶段请求资源, 也就是要在对象销毁时释放资源), 即在构造函数中请求资源 (不光是内存资源, 可能的资源有通信链路、端口、锁、文件、外设等), 而在析构函数中释放资源。当异常抛出时, 异常处理机制保证: 所有从 `try` 到 `throw` 语句之间构造起来的局部对象的析构函数将被自动调用 (以与构造相反的顺序), 然后清退堆栈 (就像函数正常退出那样)。如果一直上溯到 `main()` 函数后还没有找到匹配的 `catch` 块, 那么系统会调用 `terminate()` 终止整个程序, 这种情况下不能保证所有局部对象会被正确地销毁。

### 15.2.7 对象构造和析构期间的异常

对象在构造和析构的过程中也可能出现错误, 但是它们没有返回值来表示运行时的错误信息, 最合适的方式就是抛出异常。从构造函数和析构函数中抛出异常时

要多做些考虑。

静态创建的对象没有大问题，只要在 `try` 块内创建对象就可以捕获构造函数抛出的异常。问题在于动态创建对象的异常处理，它容易让人迷糊。我们知道，动态创建一个对象要执行两个操作：先在内存堆中分配一定数量的原始内存，如果成功的话就接着调用构造函数在这块内存上初始化一个对象。如果在分配内存时失败了，就会抛出 `std::bad_alloc` 异常。如果内存分配成功了，但是在构造函数中抛出了异常，那么是否会造成内存泄漏呢？

不必担心！如果构造函数抛出异常（即构造失败）的话，前面分配好的内存空间将被释放，这得益于 `new` 运算符内部的实现，因此不会造成内存泄漏。

例如：

```
T *p = new T;
```

将被编译器转换成类似下面的样子：

---

```
//第一步：分配原始内存，如果失败就抛出 std::bad_alloc 异常
T *p = reinterpret_cast<T*>(__new char[sizeof( T )]);
try {
    //第二步： T::T(); 在分配成功的内存块上构造对象
    new (p) T;           // placement new: 只调用 T 的构造函数
}
catch(...) {           // catch all: 构造函数中有异常抛出
    delete []p;        // 释放内存
    throw;             // 重抛异常让应用程序处理！
}
```

---

从析构函数中抛出异常是极其危险的，这是因为析构函数不仅会在正常情况下当对象生命期结束时被调用，而且当发生异常从而函数堆栈清退时也会被调用（如 15.2.6 节所述）。前一种情况下抛出异常不会有无法预料的结果，可以正常捕获；但是后一种情况就不容乐观了。如果一个函数在运行时抛出了异常，于是异常处理机制调用局部对象的析构函数（清退堆栈），而如果此时某一个析构函数恰巧也要抛出一个异常，那么这个异常将由谁来处理呢？没有办法，异常处理机制只好调用 `terminate()`。因此，如果你真的不得不从析构函数内抛出异常的话，你应该首先检查一下看当前是否有一个未捕获的异常正要被处理，如果没有，说明该析构函数的调用并非由一个外部异常引起，而是正常的销毁，于是你就可以抛出一个异常让上层程序来捕获，见示例 15-6。

示例 15-6

---

```
class DisconnectError{};
Client::~Client()throw(DisconnectError) // 异常说明
{
    if (Disconnect() == failed) {
        if (!uncaught_exception()) throw DisconnectError();
    }
}
```

---



```
    }  
}
```

然而,更好的方法是在析构函数中就地处理掉所有异常而不要让它们传播出去。

**【提示 15-5】:** 全局对象在程序开始运行之前构造,因此如果它们的构造函数中有异常抛出的话,将永远不会被捕获。全局对象的析构函数也是一样,因为它们在程序结束后才会被调用。这些异常只有操作系统才能捕获,应用程序无能为力。

### 15.2.8 如何使用好异常处理技术

想用好异常处理技术不是一件容易的事情,原因之一是它确实难用,行为比较怪异。你必须搞清楚与异常处理相关的 4 个流程:

- (1) 没有异常抛出时程序的正常执行流程。
- (2) 当抛出异常,但是如果在当前函数范围内异常没有被捕获,此时程序的执行流程。
- (3) 当抛出异常并且在当前函数范围内异常被捕获后程序的执行流程。
- (4) 以及当异常被处理后的程序执行流程。

看晕了吧!因此不少 C++ 程序员干脆就不用它。

原因之二就是它会带来额外的开销,主要是异常对象的存储、传递及类型匹配。但是请不要把“惹不起还躲得起”这种思想用到这里,只有大胆地使用才能了解它,最终才能运用自如。

不同的编译器和平台对异常处理机制的实现技术彼此不同。一些编译器可以设置异常处理支持开关,当关闭异常处理支持后,附加的数据结构、查找表、额外的代码都不会生成。但是即使你不直接使用异常处理也会隐含地使用它,例如 `new` 和许多操作符都可以抛出异常。STL 容器及标准库的其他函数也会抛出它们自己的异常,第三方供应商的代码库也可能使用异常处理机制。因此,当我们把纯 C 代码移植到 C++ 编译环境中来的时候才可以安全地关闭异常处理支持。

**【建议 15-1】:** 一般情况下不要把异常处理机制当做正常的程序控制流程来使用,如果不使用异常处理机制就能够安全而高效地消除错误,那么就不要再使用异常处理。但是由于不同编译器在实现异常处理机制时采用的技术一般不同,而且异常处理确实也是一种流程控制的手段,所以可能会应用在某些安全领域比如反跟踪。

见示例 15-7。

示例 15-7

```
class NotFound{...};  
size_t FindChar(const char *str, char ch)  
{  
    size_t idx = 0;
```

```
while (str[idx] != '\0' && str[idx] != ch)
    idx++;
if (str[idx] == '\0') throw NotFound(); // 没有必要使用异常!
return idx;
}
```

示例 15-7 的方法并不好，因为我们完全可以返回 `size_t(-1)` 来表明没有找到指定的字符，而使用异常处理机制既不好理解又要承担不必要的开销。

**【规则 15-1】:** `catch` 块的参数应当采用引用传递而不是值传递。原因之一：异常对象可能会在调用链中上溯好几个层次才能遇到匹配的处理块，显然引用传递比值传递的效率高得多；原因之二：这样可以利用异常对象的多态性，因为异常类型可能是多态类。你也可以抛出一个异常对象的地址，那么 `catch` 块中的参数就应该是异常类型的指针。

**【规则 15-2】:** 在异常组合中，要合理安排异常处理的层次：一定要把派生类的异常捕获放在基类异常捕获的前面，否则派生类异常匹配永远也不会执行到。

**【提示 15-6】:** 在异常抛出后，当找到第一个类型匹配的 `catch` 子句时，编译器就认为该异常已经被处理（识别）了。至于在 `catch` 块内如何编写异常处理代码，那就是程序员的事情了。

**【建议 15-2】:** 如果实在无法判断到底会有什么异常抛出，那就使用“一网打尽”的策略好了：`catch(void*)`和`catch(...)`。但是要记住：`catch(void*)`和`catch(...)`必须放在异常组合的最后面，并且`catch(void*)`放在`catch(...)`的前面。

见示例 15-8。

示例 15-8

```
void MemoryAllocate()
{
    try {
        double *pDouble = new double[10000000];
    }
    catch(const std::bad_alloc&) { // 派生异常类型
        cout << "Memory allocation failed!" << endl;
    }
    catch(const std::exception& ex) { // 基类型异常
        cout << ex.what() << endl;
    }
    catch(...) { // catch-all
        cout << "unknown exception occurred!" << endl;
    }
}
```

基于以上认识，最好在 `main()` 函数的结尾处使用 `catch(...)` 来一网打尽所有可能

抛出的异常。

**【规则 15-3】** 当你编写异常说明的时候，要确保派生类成员函数的异常说明和基类成员函数的异常说明一致，即派生类改写的虚函数的异常说明至少要和对应的基类虚函数的异常说明相同，甚至更加严格、更特殊（见示例 15-9）。

示例 15-9

```
class T1 {};  
class T2 {};  
class T3 {};  
class T4 : public T1 {};  
class T5 : public T2 {};  
class Base {  
public:  
    void f() throw(T1, T2);  
    void g() throw(T3);  
    void h() throw(T2);  
};  
class Derived : public Base {  
public:  
    void f() throw(T1, T2);    // OK!  
    void g() throw(T4);       // ERROR!  
    void h() throw(T5);       // OK!  
};
```

如何编写异常处理代码呢？实际上，你可以在 `catch` 块内做你想做的任何事情，例如用 `return` 语句简单地返回，或者打印错误消息、重新抛出这个异常、抛出另一个异常，甚至什么也不做，等等。

要注意**异常重抛**（`rethrow`）。你可以在 `catch` 块中使用一个空 `throw` 语句来达到此目的，但是当异常重新抛出后程序立刻退出 `try / catch` 范围而进入上一层范围，一般就是上一层调用者，除非使用了嵌套的 `try / catch` 结构。

我们可以在 `catch` 块内抛出一个不同于当前异常类型的异常对象，这样可以实现异常转换，并让上层调用者来进一步处理。在有些情况下确实有这种需要，特别是当异常说明与实际抛出的异常不符的时候（见示例 15-10）。

示例 15-10

```
class AllocFailed{};  
char* GetMemory(size_t size) throw(AllocFailed)  
{  
    try {  
        char *p = new char[size];    // 可能失败  
    }  
    catch(const std::bad_alloc&) {  
        throw AllocFailed();        // 抛出另一种异常  
    }  
}
```

```
    }  
    return p;  
}
```

还有一种办法来对付那些“漏网之鱼”，那就是使用 `set_unexpected()` 来注册一个回调函数，这在前面已经说过。

15.2.9 C++的标准异常

把 C++ 的多态特性用在异常处理中无疑是很有吸引力的，这样可以只编写一个异常处理过程来处理各种不同的特殊异常。C++ 的标准异常层次结构已经为此打好了基础，如表 15-1 所示。我们可以放心地从它们派生出自定义的异常类型，并改写成成员函数 `what()` 以实现定制的错误信息输出功能，当然你不改写而直接使用基类的 `what()` 函数也是可以的。关于标准异常类型的具体实现和使用方法可以参考其他书籍或 Visual C++ 提供的源代码。

表 15-1 标准异常及其头文件

头 文 件	异 常 类 型
<exception>	exception, bad_exception
<new>	bad_alloc
<typeinfo>	bad_cast, bad_typeid
<stdexcept>	logic_error, runtime_error, domain_error, invalid_argument, length_error, out_of_range, range_error, overflow_error, underflow_error

**【建议 15-3】：** 如果标准异常类型能够满足你的需要，就直接使用它们。这不仅减少了工作量，而且增强了代码的可移植性。

15.3 虚函数面临的难题

在介绍 RTTI 之前先来看一看虚函数的不足之处。

一般来说，虚成员函数可以满足对象的动态类型匹配的需要。一个定义良好的类层次结构应该为基类中声明的每一个虚成员函数定义有意义的操作。然而事情并非总是这样美妙。

我们举一个例子，假设有一个家用电器管理系统，可以管理电扇、电视机等。为了能够实现统一管理，我们设计一个抽象基类 `HomeElectricDevice` 来定义一些公共操作（见示例 15-11）。

示例 15-11

```
class HomeElectricDevice {  
public:  
    virtual void Open() = 0;           // 打开
```

```

        virtual void Close() = 0;                // 关闭
        virtual void Adjust(bool updown) = 0;    // 调节温度或者音量等
        // Other methods...
    private:
        // common attributes...
};

```

在基类的下面有一些派生类来管理不同的电器，如 `ElectricFan`, `Television`，它们实现了抽象基类的方法，并增加了一些特殊的操作（见示例 15-12）。

示例 15-12

```

class ElectricFan : public HomeElectricDevice {    // 电扇
public:
    virtual void Open(){...}                      // 打开
    virtual void Close(){...}                    // 关闭
    virtual void Adjust(bool updown) {...}        // 调节温度
    // ...
private:
    // attributes...
};

class Television : public HomeElectricDevice {    // 电视机
public:
    virtual void Open(){...}                      // 打开
    virtual void Close(){...}                    // 关闭
    virtual void Adjust(bool updown) {...}        // 调节音量
    // ...
    virtual void PlayVCD();                      // 播放 VCD
    // ...
private:
    // attributes...
};

```

显然，电扇和电视机有一些不同，例如电视机可以播放 VCD 而电扇不能。假设有一个控制类 `DeviceControl` 统一控制这些设备（见示例 15-13）。

示例 15-13

```

enum Command{ OPEN, CLOSE, ADJUST, PLAY_VCD, ... };
class DeviceControllor {    // 控制类
public:
    Command GetCommand(){ ... }                // 从其他渠道获取命令
    void ControlThem(HomeElectricDevice&);    // 控制方法
    // ...
};

```

在该系统中，统一的控制方法应该能够对每一个控制命令做出恰当的反应，但是由于 `HomeElectricDevice` 是一个不能实例化的抽象类，因此该函数只能接受派生类

的对象。其实现见示例 15-14。

示例 15-14

```
void DeviceController::ControlThem(HomeElectricDevice& device)
{
    Command cmd = GetCommand();
    switch(cmd)
    {
        case OPEN:
            device.Open();
            break;
        case CLOSE:
            device.Close();
            break;
        case ADJUST:
            device.Adjust(updown);
            break;
        case PLAY_VCD:
            // 这里遇到了难题！该如何实现？
            ...
    }
}
```

虽然我们初步实现了一个多态类层次结构，并且 `Open()`、`Close()` 等函数并不依赖于参数的确切类型。但是我们注意到，在实现 `PlayVCD()` 时遇到了困难，因为 `Television` 的成员函数 `PlayVCD()` 并非公共接口。你可能会说“把它提升到根类中”，但是如果这样做就是一个设计概念上的错误，因为并不是每一种电器都支持播放功能。但是当控制电视机时 `ControlThem()` 必须能够支持播放功能。此时，以虚函数形式表现的普通多态就显得力不从心了。`ControlThem()` 只知道它的参数是 `HomeElectricDevice` 派生类的对象，然而这个信息不足以告诉它当前待处理的具体对象能否播放 VCD。在这种情况下仅有静态类型检查和虚函数机制已经不足以解决所有的问题。怎么办呢？

显然，为了正常地处理所有可能的对象，`ControlThem()` 需要知道更多的关于参数运行时的类型信息。这正是需要 RTTI（运行时类型信息，运行时类型识别）的地方。在正确实现 `ControlThem()` 之前，我们先来看一下 RTTI 的构成。

## 15.4 RTTI 及其构成

### 15.4.1 起源

异常处理机制在 1989 年加入 C++ 语言，由于它是在运行时当异常抛出时才会执行具体的异常类型匹配操作，而不是在编译时就匹配好的，因此它需要一种能够在

运行时检查一个对象类型信息的手段。何况一个异常类型可能是多态类型，当抛出的异常对象为派生类对象时，如果找不到确切的处理过程就应该尝试去匹配其基类的处理过程，这也要在运行时检索一个对象的类型信息。于是直接导致了 RTTI (Run-time Type Identification) 机制的诞生。RTTI 后来的实现不仅解决了异常处理机制中的问题，它还能够完成虚函数不能完成的工作。

我们在第 12 章“C++对象模型”这一节提到过 `type_info` (类型信息)，最初的 C++ 并没有 `type_info`，是出于几个方面的考虑：

(1) 保持对 C 语言的向后兼容。因为如果支持运行时对一个对象确切类型的检索，就需要修改对象的内存映像，比如插入一个指针来指向其类型信息。虚拟机制出现后，类型信息就被放到了 `vtable` 中。

(2) RTTI 不光需要额外的内存开销（为每一个类型增加一个 `type_info` 对象），而且运行时检索对象的类型信息需要一定的时间开销。

(3) 在多数情况下虚函数的功能对于运行时对象类型的匹配已经足够了。

然而，C++ 还支持多重继承及虚拟继承这些更加复杂的特征，此时静态类型检查和虚函数动态绑定就更加不能满足对正确判断一个对象确切类型的要求了，因此 RTTI 和 `type_info` 被加入了 C++ 中。

**【提示 15-7】** RTTI 和虚函数并非一回事儿！实际上虚函数的动态绑定并没有使用对象的 `type_info` 信息，因此两者不相交，你可以从编译器对虚函数调用语句的改写手法上看出这一点。请参考《Inside The C++ Object Model》4.2 节 (Stanley B. Lippman 著)：“早在 RTTI 特征于 1993 年被引入 C++ 之前，C++ 对多态的唯一支持就是对虚函数调用的动态绑定操作。有了 RTTI 后，就能够在运行时查询一个多态指针或引用指向的具体对象的类型了”。

为了能够在运行时获得对象的类型信息 `type_info`，C++ 增加了两个运算符：`typeid` 和 `dynamic_cast<>`。`type_info` 常用的 3 个成员函数为 `operator==()`、`operator!=()` 和 `name()`，请参考标准头文件 `<typeinfo>`。

#### 15.4.2 typeid 运算符

`typeid()` 运算符就像 `sizeof()` 一样是 C++ 语言直接支持的，它以一个对象或者类型名作为参数，返回一个匹配的 `const type_info` 对象，它表明该对象的确切类型。比如我们要确认一下 `ControlThem()` 的参数 `device` 在运行时是不是一个 `Television` 对象，就可以像示例 15-15 这样来做。

示例 15-15

```
void DeviceControllor::ControlThem(HomeElectricDevice& device)
{
    Command cmd = GetCommand();
    switch(cmd)
    {
        case OPEN:
```

```

...
case PLAY_VCD:
    if (typeid(device) == typeid(Television)) {
        Television *pTemp = static_cast<Television*>(&device);
        pTemp->PlayVCD();
    } else {
        MsgBox("This device cannot play VCD!");
    }
    break;
...
}

```

表达式 `typeid(device)` 返回一个 `type_info` 对象的引用——该对象持有与 `device` 对象相关的必要的运行时类型信息；而表达式 `typeid(Television)` 则返回与类 `Television` 相关的类型信息。当 `typeid()` 作用于一个类名而不是一个对象的时候，它总是返回一个对应于该类名的 `type_info` 对象。`type_info` 类已经重载了 `operator==`，把左边表达式返回的 `type_info` 对象与右边表达式返回的 `type_info` 对象相比较。如果运行时 `device` 确实是类 `Television` 的一个实例对象，则整个逻辑表达式的值为 `true`，于是可以播放 VCD。

再解释一下本例的 `static_cast<>` 的用途：`device` 只能是 `HomeElectricDevice` 的派生类的对象（抽象基类不可实例化！）。虽然我们一眼就看出 `device` 不是 `ElectricFan` 就是 `Television` 的对象，然而编译器还没有聪明到能够自动识别这一点的地步，因此你不得不使用 `static_cast<>` 来明白地告诉编译器这个 `device` 到底是什么。

你也可以使用 `typeid()` 来检索非多态类型对象和基本数据类型对象的类型信息，只不过此时它不会去检索对象的 `vptr` 甚至 `vtable`（它们根本就没有这些设施），其结果仍然是操作数静态类型对应的 `type_info` 对象（见示例 15-16）。

示例 15-16

```

#include <typeinfo>
#include <iostream>
#include <string>
using namespace std;
typedef unsigned int UINT;
void f()
{
    cout << typeid(UINT).name() << endl;           // "unsigned int"
    cout << typeid(string).name() << endl;         // "string"
}

```

关于 `typeid()` 运算符更详细的用法及其异常，请参考编译器帮助文档或者其他书籍。



## 15.4.3 dynamic\_cast&lt;&gt;运算符

可以看出, typeid()不具备可扩展性,因为它返回一个对象的确切类型而不是基类型。一个派生类对象在语义上也应该是其基类型的对象(如果是 public 继承),然而 typeid()不具备这种判断能力。

假设一个人的收入增加了,他想买一套“家庭影院”,那么上例程序该如何扩展?你可能已经想到了:从 Television 派生出 FamilyCinema,就可以重用已经实现的行为,包括播放 VCD,并且只需实现家庭影院所支持的特殊功能,见示例 15-17。

示例 15-17

```
class FamilyCinema : public Television {
public:
    virtual void Open(){...}           // 打开
    virtual void Close(){...}         // 关闭
    virtual void Adjust(bool updown) {...} // 调节音量
    // ...
    virtual void PlayVCD();           // 播放 VCD
    // ...
private:
    // attributes...
};
```

但是,当运行时给函数 DeviceControllor::ControlThem()传入一个 FamilyCinema 对象时问题又出现了,见示例 15-18。

示例 15-18

```
void DeviceControllor::ControlThem(HomeElectricDevice& device)
{
    Command cmd = GetCommand();
    switch(cmd)
    {
        case OPEN:
            ...
        case PLAY_VCD:
            if (typeid(device) == typeid(Television)) { // 只能识别 Television 对象
                Television *pTemp = static_cast<Television*>(&device);
                pTemp->PlayVCD();
            } else {
                // 当 device 为 FamilyCinema 时将弹出对话框,这显然不合逻辑!
                MsgBox("This device cannot play VCD!");
            }
            break;
        ...
    }
}
```

`typeid(device)`返回了参数的确切类型信息（即类 `FamilyCinema` 的 `type_info` 对象），显然与类 `Television` 的 `type_info` 对象不相等，于是 `if` 语句的逻辑表达式为 `false`，从而被告知“该设备不能播放 VCD”。这显然是不对的！

每当你新增一种家用电器时上述问题都可能出现。看来仅有虚函数和 `typeid()` 还是不足以解决问题，要是有一种能够自动识别对象与类型间 `is-a` 关系的设施那就好了！

C++ 的 `dynamic_cast<>` 就是解决这个问题的运算符。`dynamic_cast<>` 用来执行运行时类型识别和转换，其语法如下：

```
dynamic_cast<dest_type>(src);
```

其中，`dest_type` 就是转换的目标类型，而 `src` 则是被转换的对象。其行为可以这样描述：如果运行时 `src` 和 `dest_type` 确实存在 `is-a` 关系，则转换可进行；否则转换失败。

**【提示 15-8】** (1) `dynamic_cast<>` 可以用来转换指针和引用，但是不能转换对象。当目标类型是某种类型的指针（包括 `void*`）时，如果转换成功则返回目标类型的指针，否则返回 `NULL`；当目标类型为某种类型的引用时，如果成功则返回目标类型的引用，否则抛出 `std::bad_cast` 异常，因为不存在 `NULL` 引用。

(2) `dynamic_cast<>` 如何能够知道一个基类型指针指向的对象是不是一个派生类对象呢？显然，它一定是通过该对象的 `vptr` 检查位于其类型的 `vtable` 第一个 `slot` 的 `type_info` 对象而得知的（见图 12-5），或者不同的编译器对 RTTI 有不同的实现手段。因此，`dynamic_cast<>` 只能用于多态类型对象（拥有虚函数或虚拟继承），否则将导致编译时错误。

`dynamic_cast<>` 可以实现两个方向的转换：`upcast` 和 `downcast`。

- ◇ `upcast`：把派生类型的指针、引用转换成基类型的指针或引用（实际上这可以隐式地进行，不必显式地转换）；
- ◇ `downcast`：把基类型的指针或引用转换成为派生类型的指针或引用。如果这个基类型的指针或引用确实指向一个这种派生类的对象，那么转换就会成功；否则转换就会失败。

**【提示 15-9】** (1) 为了支持 `dynamic_cast<>` 运算符，RTTI 机制必须维护一棵继承树，即 `base class table` 模型（或者类似的索引表格）。只有这样，`dynamic_cast<>` 才能够通过遍历其继承树来确定一个待转换的对象和目标类型之间是否存在 `is-a` 关系。

(2) `typeid()` 运算符不需要遍历继承树，实际上调用 `typeid()` 运算符的开销与虚函数动态绑定的开销是相等的。

(3) 可以看出，虚函数的动态绑定机制进行的是“精确的”对象类型匹配，而 `dynamic_cast<>` 进行的则是“模糊的”对象类型匹配。

#### 15.4.4 RTTI 的魅力与代价

我们用 `dynamic_cast<>` 来解决示例 15-18 的难题，见示例 15-19。

示例 15-19

```
void DeviceControllor::ControlThem(HomeElectricDevice& device)
{
    Command cmd = GetCommand();
    switch(cmd)
    {
        case OPEN:
            ...
        case PLAY_VCD:
            try {
                Television tv = dynamic_cast<Television &>(device);
                tv.PlayVCD(); // 现在能够处理 Television 对象和任何派生自它的对象!
            }
            catch(std::bad_cast&) {
                MsgBox("This device cannot play VCD!");
            }
            break;
        ...
    }
}
```

现在，当你从 `Television` 派生新类时不再需要修改 `case PLAY_VCD:` 这段代码。这显示了 `dynamic_cast<>` 的可扩展性，同时解决了虚函数不能解决的问题，这正是 RTTI 的魅力所在。

**【提示 15-10】** 当你使用 RTTI 时要注意如下事项：

- ✧ 如果你的编译器没有打开 RTTI 支持，那么请你打开它（请参考编译器手册）；
- ✧ 要想使用 RTTI，对象所属类型必须是多态类；
- ✧ 如果要用 `dynamic_cast<>` 转换一个引用，你要保证程序有一条 `catch()` 语句来处理 `std::bad_cast` 异常；
- ✧ 如果试图用 `typeid` 来检索 `NULL` 指针所指对象的类型信息，像这样  

```
typeid(*p); // p == NULL
```

将抛出 `std::bad_typeid` 异常；
- ✧ 当用 `dynamic_cast<>` 转换一个指针的时候，要记住检查返回值是否为 `NULL`。

RTTI 不是免费的午餐，它不仅在执行速度上而且在程序体积上都带来了额外开销。为了估计它在性能上的开销，有必要了解一下隐藏在背后的实现机制。

无论基本类型还是用户定义类型，都需要额外的内存来存放 `type_info` 对象。理想情况下，语言实现为每一种类型都生成一个独立的 `type_info` 对象。然而并非总是如此，有时候语言实现需要为每一种类型产生多个 `type_info` 对象。

请回顾一下第12章关于多态类的 `type_info` 对象是如何存放的：要给每一个多态类增加一个指针成员、一个 `type_info` 对象，以及给虚函数表增加一项。这种方法对任何多态类都是一样的，而与程序中对象的个数无关。因此，使用 `typeid()` 来检索每个对象运行时的类型信息所花时间是一样的（通过 `vptr` 间接完成），当然不如直接取用数据成员来得快了，但至少与虚函数的动态绑定开销是相等的。同任何其他对象一样，`type_info` 对象的创建过程也需要时间。

此外，RTTI 要求程序维护一棵继承树，`dynamic_cast` 能够判断源对象与目标类型之间是否具有 is-a 关系，这需要在运行时遍历继承树，并且其开销会随着源对象类型与目标类型之间距离（层次）的增大而增大。

## 第 16 章 内存管理

欢迎进入内存这片“雷区”。Bill Gates 曾经失言：

“640K ought to be enough for everybody.”

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本章的内容比一般教科书要深入得多，读者需细心阅读，做到真正地通晓内存管理。

### 16.1 内存分配方式

内存分配方式有 3 种：

(1) 从静态存储区域分配。内存在程序编译的时候就已经分配好了（即已经编址），这些内存在程序的整个运行期间都存在。例如全局变量，static 变量等。

(2) 在堆栈上分配。在函数执行期间，函数内局部变量（包括形参）的存储单元都创建在堆栈上，函数结束时这些存储单元自动释放（堆栈清退）。堆栈内存分配运算内置于处理器的指令集中，效率很高，并且一般不存在失败的危险，但是分配的内存容量有限，可能出现堆栈溢出。

(3) 从堆（heap）或自由存储空间上分配，亦称动态内存分配。程序在运行期间用 malloc()或 new 申请任意数量的内存，程序员自己掌握释放内存的恰当时机（使用 free()或 delete）。动态内存的生存期由程序员决定，使用非常灵活，但也最容易产生问题。

一般的原则是：如果使用堆栈存储和静态存储就能满足应用要求，那么就不要使用动态存储。这是因为，在堆上动态分配内存需要很可观的额外开销：

- ✧ 应用程序将调用操作系统中内存管理模块的堆管理器，搜索其中是否有符合要求的空闲的连续字节内存块。特别是在经过多次动态分配后，堆会变得“千疮百孔”——出现大量的闲散内存碎片，此时可能需要首先进行碎片合并，然后才能分配成功，在这种情况下动态分配需要很长时间。
- ✧ 如果动态分配失败，需要检查返回值或者捕获异常，这也需要额外开销。
- ✧ 动态创建的对象可能被删除多次，甚至在删除后还会继续使用，或者根本就不会被删除，于是出现运行时错误或程序“吃”内存的现象，这些问题并不是小心编程就可以避免的。

## 16.2 常见的内存错误及其对策

发生内存使用错误是一件非常麻烦的事情，因为编译器不能自动发现这些错误，它们通常在程序运行时才会蹦出来。这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，却没有发现程序运行有任何问题，但你一走，错误又发作了。

常见的内存错误及其对策如下。

(1) 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配还会失败（这些人有必要补习一下操作系统内存管理模块的知识）。常用的解决办法是，在使用内存之前检查指针是否为 NULL。如果指针 `p` 是函数的参数，那么在函数的入口处用 `assert(p != NULL)` 进行检查以避免输入非法参数。如果是用 `malloc()` 或 `new` 来申请内存，应该用 `if (p == NULL)`、`if (p != NULL)` 或者捕获异常来进行错误处理。

(2) 内存分配虽然成功，但是尚未初始化就使用它。

犯这种错误主要有两个原因：一是没有初始化的意识；二是误以为分配好的内存会自动初始化为全零，导致引用初值错误（例如数组和局部变量）。

C++ 语言并没有对内存的默认初值究竟是什么做出统一的规定，尽管有些时候会自动初始化为零值（比如全局对象或静态对象及其数组），但是我们自己应该在潜意识中认为它们没有初值。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

(3) 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 `for` 循环语句中，循环次数很容易搞错，导致数组元素访问越界。

(4) 忘记了释放内存或者只释放了部分内存，因此造成内存泄漏。含有这种错误的函数每调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统提示“内存耗尽”。这种情况在使用动态分配的对象数组时极有可能出现，比如你无意中修改了指向动态数组的指针，致使释放出错，或者使用了错误的动态数组释放语法等（参见本书 7.2.4 节）。

另外，动态内存的申请与释放必须配对，程序中 `malloc()` 与 `free()` 的使用次数一定要相同，否则肯定有错误（`new/delete` 同理）。

(5) 释放了内存却还在继续使用它。有以下几种情况：

- ✧ 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象是否已经被释放，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面；
- ✧ 函数的 `return` 语句写错了，注意不要返回指向“栈内存”的指针或者引用，因为该内存存在函数结束时被自动释放；
- ✧ 使用 `free()` 或 `delete` 释放了内存后，没有将指针设置为 `NULL`，产生“野指针”；

◇ 多次释放同一块内存。

**【规则 16-1】:** 用 `malloc` 或 `new` 申请内存之后, 应该立即检查指针值是否为 `NULL` 或者进行异常处理, 以防止使用值为 `NULL` 的指针。

**【规则 16-2】:** 不要忘记初始化指针、数组和动态内存, 防止将未初始化的内存作为右值使用。

**【规则 16-3】:** 避免数组或指针的下标越界, 特别要当心发生“多 1”或者“少 1”操作。

**【规则 16-4】:** 动态内存的申请与释放必须配对, 防止内存泄漏。

**【规则 16-5】:** 用 `free` 或 `delete` 释放了内存之后, 立即将指针设置为 `NULL`, 防止产生“野指针”。

## 16.3 指针参数是如何传递内存的

如果函数的参数是一个指针, 不要指望用该指针去申请动态内存。示例 16-1 中, `Test` 函数的语句 `GetMemory(str, 200)` 并没有使 `str` 获得期望的内存, `str` 依旧是 `NULL`, 为什么?

示例 16-1

```
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100);    // str 仍然为 NULL
    strcpy(str, "hello");   // 运行时错误;
}
```

问题出在函数 `GetMemory()` 中。编译器总是要为函数的每个参数制作临时副本, 指针参数 `p` 的副本是 `_p`, 编译器使 `_p = p`。如果函数体内的程序修改了 `_p` 指向的内容, 就导致参数 `p` 指向的内容也被做了相应的修改 (因为它们指向同一块内存空间)。这就是指针可以用做输出参数的原因。但是在本例中, `_p` 申请了新的内存, 只是把 `_p` 本身的值改变了, 即指向了新的内存空间, 但是 `p` 本身丝毫未变 (即修改了 `_p` 本身的值而不是 `_p` 指向的对象)。所以函数 `GetMemory()` 并不能输出任何东西。事实上, 每执行一次 `GetMemory()` 就会泄漏一块内存, 因为没有用 `free()` 释放内存。

如果非得要用指针参数去申请内存, 那么应该改用“指向指针的指针”或者“指针的引用”, 见示例 16-2。

示例 16-2

---

```
void GetMemory2(char **p, int num)           // 或者是 char *& rp
{
    *p = (char *)malloc(sizeof(char) * num);  // rp = ...
}
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100);                    // 注意参数是&str, 而不是 str
    strcpy(str, "hello");                      // ok
    cout << str << endl;
    free(str);                                 // ok
}
```

---

由于“指向指针的指针”及“指针的引用”这些概念不容易理解，我们可以用函数返回值来传递动态内存，这种方法更加简单，见示例 16-3。

示例 16-3

---

```
char * GetMemory3(int num)
{
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}
void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout << str << endl;
    free(str);
}
```

---

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 `return` 语句用错。这里强调不要用 `return` 语句返回指向“栈内存”的指针或引用，因为该内存存在函数结束时将自动释放，见示例 16-4。

示例 16-4

---

```
char * GetString(void)
{
    char p[] = "hello world";                // 用字符串常量来初始化数组的内存空间
    return p;                                // 编译器将提出警告
}
void Test4(void)
{

```

---



```

char *str = NULL;
str = GetString();           // str 的内容是垃圾
cout << str << endl;
}

```

用调试器逐步跟踪 Test4(), 发现执行 `str = GetString()` 语句后 `str` 不再是 NULL 指针, 但是 `str` 的内容不是 “hello world” 而是垃圾。

如果把示例 16-4 改写成示例 16-5, 会怎么样?

示例 16-5

```

char * GetString2(void)
{
    char *p = "hello world";
    return p;
}
void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout << str << endl;
}

```

函数 Test5() 运行时不会出错, 不过不提倡这样使用。函数 GetString2() 内的 “hello world” 是常量字符串, 位于静态存储区, 它在程序生命期内始终有效。无论什么时候调用 GetString2(), 返回的始终是同一个 “只读” 内存块的地址, 不可试图对它进行写操作, 可以把返回值改为 `const char*` 以避免无意的修改。

## 16.4 free 和 delete 把指针怎么啦

别看 `free()` 和 `delete` 的名字恶狠狠的 (尤其是 `delete`), 它们只是把指针所指的内存给释放掉, 并没有把指针本身删掉。

用调试器跟踪示例 16-6, 发现指针 `p` 被 `free` 以后其地址仍然不变 (不等于 NULL), 只是该地址对应的内存是垃圾——`p` 成了 “野指针”! 如果此时不把 `p` 设置为 NULL, 会让人误以为 `p` 是个有效的指针。

如果程序比较长, 我们有时记不住 `p` 所指的内存是否已经被释放, 在继续使用 `p` 之前, 通常会用语句 `if (p != NULL)` 进行防错处理。很遗憾, 此时 `if` 语句起不到防错作用, 因为即便 `p` 不是 NULL 指针, 它也可能不再指向合法的内存块。

示例 16-6

```

char *p = (char *) malloc(100);
strcpy(p, "hello");
free(p);           // p 所指向的内存被释放, 但是 p 本身仍然不变

```

```
                // 如果程序比较长, 我们有时记不住 p 所指的内存是否已经被释放
if(p != NULL)   // 没有起到防错作用
{
    strcpy(p, "world");    // 运行时错误
}
```

可见, 不等于 NULL 的指针不一定就是有效的指针, 所以再次强调一定不要忘记初始化指针变量为 NULL 或者有效地址。

## 16.5 动态内存会被自动释放吗

函数体内的局部变量在函数结束时自动消亡。很多人误以为示例 16-7 是正确的, 理由是 p 是局部的指针变量, 它消亡的时候会让它所指的动态内存一起完蛋。这是错觉!

示例 16-7

```
void Func(void)
{
    char *p = (char *) malloc(100);    // 动态内存会自动释放吗?
}
```

正是由于指针变量和其指向的内存空间不是同一回事, 才导致指针看上去具有一些不易理解的特征:

- (1) 指针消亡了, 并不表示它所指向的内存会被自动释放。
- (2) 内存被释放了, 并不表示指针会消亡或者成了 NULL。

这表明释放内存并不是一件可以草率对待的事。也许有人不服气, 一定要找出可以草率行事的理由:

如果程序终止了运行, 一切指针都会消亡, 动态内存会被操作系统回收。既然如此, 在程序临终前, 就可以不必释放内存、不必将指针设置为 NULL 了。终于可以偷懒而不会发生错误了吧?

想得倒美! 程序如果长期不结束, 那要“吃掉”多少内存啊! 再说了, 如果别人把那段程序取出来用到其他地方, 会带来什么样的后果呢?

## 16.6 杜绝“野指针”

“野指针”不是 NULL 指针, 而是指向“非法”内存的指针。人们一般不会错用 NULL 指针, 因为用 if 语句很容易判断。但是“野指针”是很危险的, if 语句对它可能不起作用。

“野指针”的成因主要有 3 种。

(1) 没有初始化指针变量。任何指针变量刚被创建时不会自动成为 NULL 指针，它的默认值是随机的，它会乱指一气。所以，指针变量在创建的同时应当初始化，要么将指针设置为 NULL，要么让它指向有效的内存。例如：

```
char *p = NULL;           // NULL 为合法指针值
char *str = (char *) malloc(100); // 指向有效内存区
```

(2) 指针 p 被 free() 或者 delete 之后，没有置为 NULL，让人误以为 p 仍然是一个有效的指针。

(3) 指针操作超越了变量的作用范围，这种情况让人防不胜防，见示例 16-8。

示例 16-8

```
class A {
public:
    void Func(void){ cout << "A::func()" << endl; }
};
void Test(void)
{ // 外层程序块
    A *p = NULL;
    { // 内层程序块
        A a;
        p = &a; // 注意 a 的生命期
    }
    p->Func(); // p 是“野指针”
}
```

函数 Test() 在执行语句 p->Func() 时对象 a 已经不存在了，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是运行这个程序时居然没有出错。

有位读者指出可能的原因：成员函数 Func() 不是虚函数，它的地址是在编译时刻就决定了，而且 Func() 中没有引用任何数据成员，当然不会出错啦！

实际上，根本原因是 a 虽然退栈了，但仅仅是调用了一下析构函数而已，而我们知道析构函数其实没干什么重要的事情，它并没有清除 a 的内存（a 的内存空间仍然在函数堆栈上），所以 a 仍然好好地在那里放着，只是你无法在程序块外直接访问而已。该问题不在我们的讨论范围之内。

## 16.7 有了 malloc/free 为什么还要 new/delete

malloc() 与 free() 是 C++/C 语言的标准库函数，new/delete 是 C++ 的运算符，它们都可用于申请和释放动态内存。

对于非内部数据类型（如 ADT/UDT）的对象而言，光用 malloc()/free() 无法满足动态对象的要求：对象在创建的同时要自动调用构造函数，对象在销毁的时候要自

动调用析构函数。由于 `malloc()/free()` 是库函数而不是运算符，不在编译器控制权限之内，不能把调用构造函数和析构函数的任务强加给它们。因此 C++ 语言需要一个能够完成动态内存分配和初始化工作的运算符 `new`，以及一个能够完成清理与释放内存工作的运算符 `delete`。

**【提示 16-1】** `new/delete` 并非库函数，而是语言实现直接支持的运算符，就像 `sizeof()` 和 `typeid()` 及 C++ 新增的 4 个类型转换运算符也不是库函数一样。

我们先看一看 `malloc()/free()` 和 `new/delete` 如何实现对象的动态内存管理，参示例 16-9。

示例 16-9

```
class Obj {
public:
    Obj(){ cout << "constructor" << endl; }
    ~Obj(){ cout << "destroy" << endl; }
    void Initialize(void){ cout << "initialize" << endl; }
    void Destroy(void){ cout << "destroy" << endl; }
};

void usemallocfree(void)
{
    Obj *a = (Obj *)malloc(sizeof(Obj)); // 申请动态内存
    a->Initialize();                     // 初始化
    //...
    a->Destroy();                         // 清除工作
    free(a);                             // 释放内存
}

void UseNewDelete(void)
{
    Obj *a = new Obj;                    // 申请动态内存并调用构造函数来初始化
    //...
    delete a;                           // 调用析构函数清除并且释放内存
}
```

类 `Obj` 的函数 `Initialize()` 模拟了构造函数的功能，函数 `Destroy()` 模拟了析构函数的功能。在函数 `usemallocfree()` 中，由于 `malloc()/free()` 不能调用构造函数和析构函数，必须调用成员函数 `Initialize()` 和 `Destroy()` 来完成初始化和清除工作。函数 `UseNewDelete()` 则简单得多。

所以，我们不要企图用 `malloc()/free()` 来完成动态对象的内存管理，应该用 `new/delete`。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 `malloc()/free()` 和 `new/delete` 是等价的。

既然 `new/delete` 的功能完全覆盖了 `malloc()/free()`，为什么 C++ 不把 `malloc()/free()` 淘汰出局呢？这是因为：

(1) C++ 程序经常要调用 C 函数，而且有些 C++ 实现可能使用 `malloc()/free()`

来实现 new/delete，而 C 程序只能用 malloc()/free() 来管理动态内存。

(2) 使用 new/delete 更加安全。因为 new 可以自动计算它要构造的对象的字节数量（包括成员边界调整而增加的填补字节和隐含成员），而 malloc() 则不能；new 直接返回目标类型的指针，不需要显式地转换类型，而 malloc() 则返回 void\*，必须首先显式地转换成目标类型后才能使用。

(3) 我们可以为自定义类重载 new/delete，实现富有“个性”的内存分配和释放策略。而 malloc()/free() 不能被任何类重载，否则就改变了标准的定义。

(4) 在某些情况下，malloc()/free() 可以提供比 new/delete 更高的效率，因此某些 STL 实现版本的内存分配器会采用 malloc()/free() 来进行存储管理。

如果用 free() 释放 new 创建的动态对象，那么该对象可能会因无法执行析构函数导致程序出错。C++ 标准并不保证 new 的底层实现会使用 malloc()，更有甚者，在一些实现中 malloc() 和 new 使用不同的堆。同样，如果用 delete 释放 malloc() 申请的动态内存，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc()/free() 也是一样。

## 16.8 malloc/free 的使用要点

函数 malloc() 的原型如下：

```
void * malloc(size_t size);
```

用 malloc() 申请一块长度为 length 的整型数组的内存，程序如下：

```
int *p = (int *)malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

- ✧ malloc() 函数返回值的类型是 void\*，所以在调用 malloc() 时要显式地进行类型转换，将 void\* 转换成所需要的指针类型。
- ✧ malloc() 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住 int、float 等数据类型变量的确切字节数。例如，int 变量在 16 位系统下是 2 字节，在 32 位系统下是 4 字节，而 float 变量在 16 位系统下是 4 字节，在 32 位系统下也是 4 字节。最好用以下程序做一次测试：

```
cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

在 malloc() 函数的 “()” 中使用 sizeof 运算符是良好的风格，但要当心有时昏了头，写出 `p = malloc(sizeof(p))` 这样的错误程序来！

函数 free 的原型如下：

```
void free( void * memblock );
```

为什么 free() 函数不像 malloc() 函数那样复杂呢？这是因为指针 p 的类型及它所指的内存容量事先都是知道的，语句 free(p) 能正确地释放内存。如果 p 是 NULL 指针，那么函数 free() 对 p 无论操作多少次都不会出问题。如果 p 不是 NULL 指针，那么函数 free() 对 p 连续操作两次就会导致程序运行时出现错误。

## 16.9 new 有 3 种使用方式

也许你还不知道，new 有 3 种使用方式，它们是：plain new、nothrow new 及 placement new。这 3 种形式极大地扩展了动态内存分配的灵活性，下面我们来分别介绍。

### 16.9.1 plain new/delete

从字面上就可以看出它的意思：这是普通的 new，也就是我们通常使用的那种 new，没有任何附加成分。它们在 <NEW> 中是这样定义的：

```
void * operator new(std::size_t) throw(std::bad_alloc);  
void operator delete(void *) throw();
```

**【提示 16-2】** 在未标准化时，C++ 中的 new 运算符总是返回 NULL 表示分配失败，就像 malloc() 那样，因此程序员不得不总检查其返回值。标准 C++ 修订了 new 的语义，plain new 在失败后抛出标准异常 std::bad\_alloc 而不是返回 NULL。然而，现在很多使用 C++ 的程序员还以为 new 是以前的老样子，于是通过检查其返回值是否为 NULL 来判断分配成功与否，显然这种检查是徒劳的。

plain new 的使用方法见示例 16-10。

示例 16-10

```
Char * GetMemory(unsigned long size)  
{  
    char *p = new char[size];  
    return p;  
}  
void main(void)  
{  
    try {  
        char *p = GetMemory(1000000);    // 可能抛出 std::bad_alloc 异常
```

```

//...
delete []p;
}

catch(const std::bad_alloc& ex) {
    cout << ex.what() << endl;
}
}

```

### 16.9.2 nothrow new/delete

**【提示 16-3】** 顾名思义，nothrow new 就是不抛出异常的运算符 new 的形式，nothrow new 在失败时返回 NULL。所以使用它时就不需要设置异常处理器，而是像过去一样检查返回值是否为 NULL 即可。

nothrow new 在<NEW>中是这样定义的：

```

void * operator new(std::size_t, const std::nothrow_t&) throw();
void operator delete(void *) throw();

```

nothrow new 带一个参数，其类型为 nothrow\_t（定义在头文件<NEW>中）：

```

struct nothrow_t{ };
const nothrow_t nothrow;

```

nothrow 是 C++ 语言实现定义的一个 nothrow\_t 的全局 const 对象，作为 new 的标志性哑元而已。nothrow new 的使用方法见示例 16-11。

示例 16-11

```

Void func(unsigned long length)
{
    unsigned char *p = new(nothrow) unsigned char[length];
    if(p == NULL) cout << "allocate failed!" << endl;
    //...
    delete []p;
}

```

### 16.9.3 placement new/delete

placement 意即“放置”，这种 new 形式允许在一块已经分配成功的内存上重新构造对象或者对象数组。你还记得吗，在本书 15.2.7 节剖析 new 如何处理构造函数中抛出的异常，那里就使用了 placement new。

显然，placement new 不用担心内存分配失败，因为它根本就不会分配内存，它所做的唯一一件事情就是调用对象的构造函数。它们在<NEW>中如下定义：

```

void *_operator new(size_t, void *);
void _operator delete(void *, void *);

```

placement new 的语法如下：

```
type-name *q = new(p) type-name;
```

其中, `p` 就是已经分配成功的内存区的首地址, 它被转换为目标类型的指针 `q`, 因此 `p` 和 `q` 相等。placement new 的使用方法见示例 16-12。

示例 16-12

```
#include <new>
#include <iostream>
void main(void)
{
    using namespace std;
    char *p = new(nothrow) char[4];    // nothrow new
    if(p == NULL) {
        cout << "allocate failed!" << endl;
        exit(-1);
    }
    //...
    long *q = new(p) long(1000);      // placement new
    //...
    delete []p;    // 释放内存
}
```

甚至可以在一块静态分配的内存上使用 placement new, 比如函数堆栈, 这就相当于显式地调用了一次构造函数。不过建议你不要这样做, 不仅是因为这可能会给你造成错觉: 以为要使用 delete 来释放它, 或者别人误以为“调用了 new 却没有调用 delete”, 更重要的是这会导致未定义行为。

**【提示 16-4】** placement new 的主要用途就是: 反复使用一块较大的动态分配成功的内存来构造不同类型的对象或者它们的数组。比如, 可以先申请一个足够大的字符数组, 然后当需要时在它上面构造不同类型对象或其数组。见示例 16-13 (假设 ADT 表示任意复合数据类型)。

示例 16-13

```
#include <new>
#include <iostream>
void main(void)
{
    using namespace std;
    char *p = new(nothrow) char[100];    // nothrow new
    if(p == NULL) {
        cout << "allocate failed!" << endl;
        exit(-1);
    }
    //...
    long *q1 = new(p) long(100);        // placement new: 不必担心失败
}
```



```

//...
int *q2 = new(p) int[100 / sizeof(int)];    // placement new 数组
//...
ADT *q3 = new(p) C[100 / sizeof(ADT)];
//...
delete []p;    // 释放内存
}

```

**【提示 16-5】:**

使用 placement new 构造起来的对象或其数组,要显式地调用它们的析构函数来销毁(析构函数并不释放对象的内存),千万不要使用 delete。这是因为, placement new 构造起来的对象或其数组的大小并不一定等于原来分配的内存大小,因此使用 delete 会造成内存泄漏,或者在之后释放内存时出现运行时错误。见示例 16-14(假设 ADT 表示任意复合数据类型)。

示例 16-14

```

#include <new>
#include <iostream>
void main(void)
{
    using namespace std;
    char *p = new(nothrow) char[sizeof(ADT) + 2];    // nothrow new
    if(p == NULL) {
        cout << "allocate failed!" << endl;
        exit(-1);
    }
    .
    ADT *q = new(p) ADT;                            // placement new: 不必担心失败
    //...
    // delete q;                                     // 错误!不能在此处调用 delete q;
    q->ADT::~~ADT();                                  // 显示调用析构函数
    delete []p;                                       // 再释放内存
}

```

我们总结一下 new/delete 的各种用法,见表 16-1。

表 16-1 new/delete 的各种用法

new/delete 类型	plain	nothrow	placement
对 象	new type-name; delete p;	new(nothrow) type-name; delete p;	new(p) type-name; delete p;
对 象 数 组	new type-name[x]; delete []p;	new(nothrow) type-name[x]; delete []p;	new(p) type-name[x]; delete []p;

new/delete 的 3 种形式实际上是预先定义好的 new/delete 运算符重载。当然,你

也可以为自定义类型重载 new/delete 运算符, 请参考《Effective C++》条款 10。如果你想更多地了解 new/delete 运算符背后的秘密, 请参考《Inside The C++ Object Model》第 6 章。

## 16.10 new/delete 的使用要点

运算符 new 使用起来要比函数 malloc()简单得多, 例如:

```
int *p1 = (int *)malloc(sizeof(int) * length);
int *p2 = new int[length];
```

这是因为 new 内置了 sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言, new 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数, 那么 new 的语句也可以有多种形式。例如:

```
class Obj {
public:
    Obj();                // 默认构造函数
    Obj(int x);           // 带一个参数的构造函数
    ...
}
void Test(void)
{
    Obj *a = new Obj;
    Obj *b = new Obj(1);  // 初值为 1
    ...
    delete a;
    delete b;
}
```

如果用 new 创建对象数组, 那么只能使用对象的默认构造函数。例如:

```
Obj *objects = new Obj[100];    // 创建 100 个动态对象
```

不能写成:

```
Obj *objects = new Obj[100](1); // 创建 100 个动态对象的同时赋初值 1
```

在用 delete 释放对象数组时, 留意不要丢了符号 “[]”。例如:

```
delete []objects;    // 正确的用法
delete objects;      // 错误的用法
```

后者相当于 delete objects[0], 漏掉了另外 99 个对象。

**【提示 16-6】:** (1) 不论是何种类型, new/delete 和 new[]/delete[]总是应该正确配对使用。因此, 没有任何一种数据类型, 对于其动态创建的数组来说 delete p 和 delete []p 是等价的;

(2) 多次 delete 一个不等于 NULL 的指针会导致运行时错误, 但是多次 delete 一个 NULL 指针没有任何危险, 因为 delete 运算符会首先检查这种情况, 如果指针为 NULL, 则直接返回。

## 16.11 内存耗尽怎么办

如果在申请动态内存时找不到足够大的连续字节内存块, malloc() 和 new 会使用不同的方式宣告内存申请失败。通常有如下几种方式处理“内存耗尽”问题。

(1) 判断指针是否为 NULL, 如果是则立刻用 return 语句终止本函数。例如:

```
void Func(void)
{
    A *a = new(nothrow) A;
    if(a == NULL) return;
    ...
}
```

(2) 判断指针是否为 NULL, 如果是则立刻用 exit(1) 终止整个程序的运行。例如:

```
void Func(void)
{
    A *a = new(nothrow) A;
    if(a == NULL) exit(1);
    ...
}
```

(3) 为 new 和 malloc() 预设异常处理函数。例如, Visual C++ 可以用 \_set\_new\_handler 函数为 new 设置用户自定义异常处理函数, 也可以让 malloc() 享用与 new 相同的异常处理函数 (详细内容请参考 C++ 使用手册)。

(4) 捕获 new 抛出的异常, 并尝试从中恢复。

上述 (1) 和 (2) 两种方式使用最普遍。如果一个函数内有多处需要动态申请内存, 那么方式 (1) 就显得力不从心 (释放内存很麻烦), 应该用方式 (2) 来处理。不过在 C++ 中我们提倡使用方式 (4)。

很多人不忍心用 exit(1), 问: “不编写出错处理程序, 让操作系统自己解决行不行?”

不行。如果发生“内存耗尽”这样的事情, 一般说来应用程序已经无药可救。如果不用 exit(1) 把坏程序杀死, 它可能会害死操作系统。道理如同不把该死的歹徒击毙, 歹徒在老死之前会犯下更多的罪恶。

有一个很重要的现象要告诉大家:

对于 32 位以上的应用程序而言, 一般情况下使用 malloc() 和 new 几乎不可能导致“内存耗尽”。我在 Windows 98 下用 Visual C++ 编写了测试程序, 见示例 16-15。

这个程序会无休止地运行下去，根本不会终止。因为 32 位操作系统支持“虚存”，内存用完了，自动用硬盘空间顶替。我只听到硬盘嘎吱嘎吱地响，Window 98 已经累得对键盘、鼠标毫无反应了。

示例 16-15

```
void main(void)
{
    float *p = NULL;
    while(true) {
        p = new(nothrow) float[1000000]; // 或者 malloc(sizeof(float)*1000000);
        cout << "eat memory" << endl;
        if(p==NULL) exit(1);
    }
}
```

可以得出这么一个结论：

对于 32 位以上的应用程序，“内存耗尽”错误处理程序几乎毫无用处。这下可把 UNIX 和 Windows 程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。

我不想误导读者，必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

对于那些大型科学运算，比如求解数学难题、气象预报、环境监测等领域，以及那些需要使用大规模递归算法来求解的难题，还有 64 层汉诺塔问题，很容易出现“内存不足”的问题，所以不可掉以轻心。

## 16.12 用对象模拟指针

可以这样说，C++/C 程序中绝大多数 Bug 都与内存使用有关，也就是与指针使用不当有关。既然使用指针有这么大的危险性，何不用对象来模拟指针呢？是的，当然可以！但是这是一项很艰巨的工作，不仅要保持指针的使用方式不变，还要考虑把指针变为对象以后带来的各种副作用，毕竟是在使用含有指针成员的对象而不是单纯的指针。

用对象模拟指针的程序如示例 16-16。

示例 16-16

```
class ADT {
public:
    void mf(){ cout << "ADT::mf()" << endl; }
    ...
};
class ADTPtr {
```

```

public:
    ADTPtr() : m_p(NULL){}                // 构造函数
    ADTPtr(ADT *p);                        // 构造函数
    ADTPtr(const ADTPtr& ptr);              // 构造函数
    ADTPtr& operator=(const ADTPtr& ptr);    // 赋值函数
    ~ADTPtr();                             // 析构函数

    // operator overloads
    ADT& operator*(){ return *m_p; }        // 重载 "*"
    ADT* operator->(){ return m_p; }         // 重载 "->"
    ADTPtr& operator++(){ m_p++; return *this; } // 重载前置版 "++"
    ADTPtr operator++(int){
        ADT temp = *this;
        ++(*this);                          // 调用前置版
        return temp;
    }
    ADTPtr& operator--(){ m_p--; return *this; } // 重载前置版 "--"
    ADTPtr operator--(int){
        ADT temp = *this;
        --(*this);                          // 调用前置版
        return temp;
    }
    bool operator==(const ADTPtr& right){ return (m_p == right.m_p); }
    bool operator!=(const ADTPtr& right){ return (!(*this == right)); }
    operator ADT*()const{ return m_p; }

private:
    ADT * m_p;
};

```

既然用对象来模拟指针，那就要用对象的观点来看待指针。例如，如果向一个函数传递一个简单的指针，那么就是简单地拷贝该指针的值，而不会重新创建它指向的对象，但是如果以值来传递一个 ADTPtr 对象，那就会调用 ADTPtr 的拷贝构造函数；同样在两个指针之间赋值时执行的是简单的值的拷贝，而把一个 ADTPtr 对象赋值给另一个 ADTPtr 对象，则会调用 ADTPtr 的拷贝赋值函数（operator=）；而如果 delete 一个指针，就会调用指针所指向对象的析构函数，ADTPtr 对象在生命期结束时要调用 ADTPtr 的析构函数……那么这些函数该如何实现呢？

一般来说，含有指针成员的对象初始化和拷贝有两种方式：接管和深拷贝。前者指的是新对象的指针成员接管源指针指向的数据对象，即与源指针具有相同的值，在 C++ 中接管就是指按 bit 拷贝语义；后者指的是新对象的指针成员指向新创建的数据对象，并用源指针指向的对象来初始化该数据对象，在 C++ 中这就是按成员拷贝语义。这两者的一般用法如下：

- ✧ 普通的 ADT/UDT 中含有指针成员作为数据成员之一，例如 char\*，此时不应该看作是专门对指针的封装（专门定义的 string 类型除外）。应采用按成员拷贝方式来处理指针成员指向的对象。
- ✧ 通过封装 ADT/UDT 的指针来改变或者扩展它们的接口，这样的新类型就被

当做是原来的 ADT/UDT，俗称二次封装、适配器或者句柄类。此时有些情况下采用接管方式，有些情况下又采用深拷贝方式，而有时候还同时使用两种方式。

- ◇ 用于其他目的的指针成员，例如仅作为临时信息记录的对象等，此时应采用接管方式。

显然，模拟指针应该采用接管而非拷贝方式，因为它要与普通指针的行为保持一致。因此，示例 16-16 中的拷贝构造函数、拷贝赋值函数和析构函数可按示例 16-17 来实现。

示例 16-17

```
ADTPtr::ADTPtr(ADT *p) : m_p(p){}
ADTPtr::ADTPtr(const ADTPtr& ptr) : m_p(ptr.m_p){}
ADTPtr& ADTPtr::operator=(const ADTPtr& ptr)
{
    if(this != &ptr)m_p = ptr.m_p;
    return *this;
}
~ADTPtr::ADTPtr(){}

```

有些软件库使用这种技术来封装指针，基本上分为如下几种情况。

- ◇ 采用拷贝方式。这样的指针对象既负责创建数据对象，又负责删除数据对象，STL 容器对象采用的就是这种方式。显然，采用这种方式的指针对象责任最清晰。
- ◇ 采用完全接管方式，指针对象不负责创建数据对象，但是负责删除数据对象，即不仅接管了源指针指向的对象，而且接管了它的所有权。auto\_ptr<> 类就是采用这种方式实现的。
- ◇ 采用接管方式，是既不负责创建数据对象也不负责删除数据对象，这就是我们的模拟指针。STL 中的迭代器 (iterator) 采用的就是这种方式，它们在行为上与底层的指针变量没有什么太大区别，只是在使用方式上统一了起来。
- ◇ 完全接管方式和深拷贝方式结合。一般情况是：拷贝构造和拷贝赋值采用深拷贝方式，而指针构造和指针赋值采用接管方式。这种方式最容易产生运行时内存访问冲突和内存泄漏问题，因此建议不要使用。

举一个使用 ADTPtr 的例子，如示例 16-18 所示。

示例 16-18

```
ADTPtr CallMf(ADTPtr ptr)
{
    ptr->mf();
    return ptr;
}

```

```

void main(void)
{
    ADTPtr p = new ADT;
    CallMf(p);
    ...
    delete p;
}

```

## 16.13 泛型指针 auto\_ptr

上一节中的模拟指针是专门针对一种 ADT/UDT 类型的，不具有通用性。要想达到通用，就得使用模板技术（即泛型技术）。比如 STL 容器的迭代器，虽然它们本身并不是类模板，但是把它们与特定的泛型容器绑定在一起，也就成了泛型指针。

下面我们来看一看 auto\_ptr<T>类是如何定义的（简化版本，见示例 16-19）。

示例 16-19

```

template<typename T>
class auto_ptr {
public:
    explicit autp_ptr(T *p = 0) : m_ptr(p){}
    auto_ptr(const auto_ptr<T>& copy) : m_ptr(copy.release()){}
    auto_ptr<T>& operator=(const auto_ptr<T>& assign) {
        if (this != &assign){
            delete m_ptr;
            m_ptr = assign.release();    // 释放并移交拥有权
        }
        return *this;
    };
    ~auto_ptr(){ delete m_ptr; }        // 负责释放存储
    T& operator*(){ return *m_p; }     // 重载 "*"
    T* operator->(){ return m_p; }     // 重载 "->"
    T* release() const {
        T *temp = m_ptr;
        (const_cast<auto_ptr<T> *>(this))->m_ptr = 0;
        return temp;
    }
private:
    T * m_ptr;
};

```

使用 auto\_ptr<T>这样的灵巧指针有一个好处：当函数即将退出或有异常抛出的时候，不再需要我们显式地用 delete 来删除每一个动态创建起来的对象，C++ 保证会在堆栈清退的过程中自动调用每一个局部对象的析构函数，而析构函数会调用 delete

来完成这个任务。从而有效地避免了因忘记删除动态对象而造成的内存泄露问题。见示例 16-20。

示例 16-20

```
void func()
{
    auto_ptr<BigClass> pb(new BigClass);
    //...
    if(x == 0)throw "x equals to 0";
    //...
} //这里会自动调用~auto_ptr<>
```

## 16.14 带有引用计数的智能指针

实际上, 关于引用计数和智能指针这两项技术已经有许多 C++ 书籍详细介绍了, 如在 Scott Meyers 的《More Effective C++》和 Andrei Alexandrescu 的《Modern C++ Design》以及 David Vandevoorde 的《C++ Templates》中都有精彩的论述, 所以这里就不再班门弄斧了。总结起来其实就是一句话: 带有引用计数功能的智能指针兼有普通指针共享实值对象和 auto\_ptr 自动释放实值对象的双重功能, 并自动管理实值对象的生命周期和有效引用的计数, 不会造成丢失引用、内存泄漏及多次释放等问题。作者实现了一个依赖于非侵入式引用计数器的智能指针, 并且一直在实际开发中使用, 现在贡献出来供大家参考 (见示例 16-21, 经过了简化)。其对象模型如图 16-1 所示。

示例 16-21

```
// 非侵入式引用计数器管理对象接口
template<typename _Ty>
class INonintrusiveRefCountManager {
public:
    virtual size_t      addRef() = 0;           // return new refcount
    virtual size_t      release() = 0;         // return new refcount
    virtual _Ty*         realObject() = 0;
    virtual const _Ty*   realObject() const = 0;
};

template<typename _Ty>
class DefaultNonintrusiveRefCountMgr : public INonintrusiveRefCountManager<_Ty> {
public:
    typedef _Ty   DataObjectType;
    explicit DefaultNonintrusiveRefCountMgr(DataObjectType *rep = NULL)
        : m_pData(rep), m_refCount(1) { }    /* 初始引用计数为 1 */
    virtual ~DefaultNonintrusiveRefCountMgr() { }
    virtual size_t addRef() {                  // 引用计数+1
```



```

        ::InterlockedIncrement((long*)&m_refCount);           // WIN32 API
        return m_refCount;
    }
    virtual size_t release() {                                // 引用计数-1
        ::InterlockedDecrement((long*)&m_refCount);           // WIN32 API
        if (m_refCount == 0) {
            this->_Destroy();    // 如果引用计数为 0, 则删除实值对象
            delete this; return 0;    // must return 0 quickly!
        }
        return m_refCount;
    }
    virtual DataObjectType* realObject() { return m_pData; }
    virtual const DataObjectType* realObject() const { return m_pData; }
protected:
    void _Destroy() {
        delete m_pData; m_pData = NULL;                        // 销毁实值对象
    }
    DataObjectType    *m_pData;
    size_t            m_refCount;
private:
    // 禁止拷贝和赋值
    DefaultNonintrusiveRefCountMgr(const DefaultNonintrusiveRefCountMgr<_Ty>&);
    void operator=(const DefaultNonintrusiveRefCountMgr<_Ty>&);
};

// 非侵入式智能指针模板
template<typename _Ty, /*实值对象类型*/
        typename RefcountManager = DefaultNonintrusiveRefCountMgr<_Ty>>
class SmartPtr {
public:
    typedef typename RefcountManager::DataObjectType DataObjectType;
    SmartPtr() : m_rep(NULL) {}
    explicit SmartPtr(DataObjectType *pointee) {                // 接管
        if (pointee != NULL)
            m_rep = new RefcountManager(pointee);
        else
            m_rep = NULL;                                       // m_rep 可能为 NULL
    }
    SmartPtr(const SmartPtr<_Ty, RefcountManager>& other) { // 拷贝智能指针
        if ((m_rep = other.m_rep) != NULL)
            m_rep->addRef();                                     // 只增引用计数
    }
    ~SmartPtr() {
        if (m_rep != NULL)
            m_rep->release();                                     // 释放拥有权
    }
};
SmartPtr<_Ty, RefcountManager>& \

```

```
operator=(const SmartPtr<_Ty, RefcountManager>& other) { // 智能指针赋值
if (this != &other && m_rep != other.m_rep) {
    if (m_rep != NULL)
        m_rep->release(); // 释放原指针拥有权
    if ((m_rep = other.m_rep) != NULL)
        m_rep->addRef(); // 只增引用计数
    }
return (*this);
}

SmartPtr<_Ty, RefcountManager>& operator=(DataObjectType* pointee) { // 接管
    RefcountManager *pTemp = NULL;
    if (pointee != NULL)
        pTemp = new RefcountManager(pointee);
    if (m_rep != NULL)
        m_rep->release();
    m_rep = pTemp; return (*this);
}

bool operator!() const
{ return ((m_rep == NULL || m_rep->realObject() == NULL) ? true : false); }

operator void*() const
{ return (m_rep == NULL ? NULL : m_rep->realObject()); }

DataObjectType* operator->() {
    assert(m_rep != NULL && m_rep->realObject() != NULL);
    return (m_rep->realObject());
}

const DataObjectType* operator->() const {
    assert(m_rep != NULL && m_rep->realObject() != NULL);
    return (m_rep->realObject());
}

DataObjectType& operator*() {
    assert(m_rep != NULL && m_rep->realObject() != NULL);
    return (*(m_rep->realObject()));
}

const DataObjectType& operator*() const {
    assert(m_rep != NULL && m_rep->realObject() != NULL);
    return (*(m_rep->realObject()));
}

// 获得实值对象的指针
DataObjectType* _ugly_method()
{ return (m_rep == NULL ? NULL : m_rep->realObject()); }
const DataObjectType* _ugly_method() const
{ return (m_rep == NULL ? NULL : m_rep->realObject()); }

private:
    RefcountManager    *m_rep;
};
```

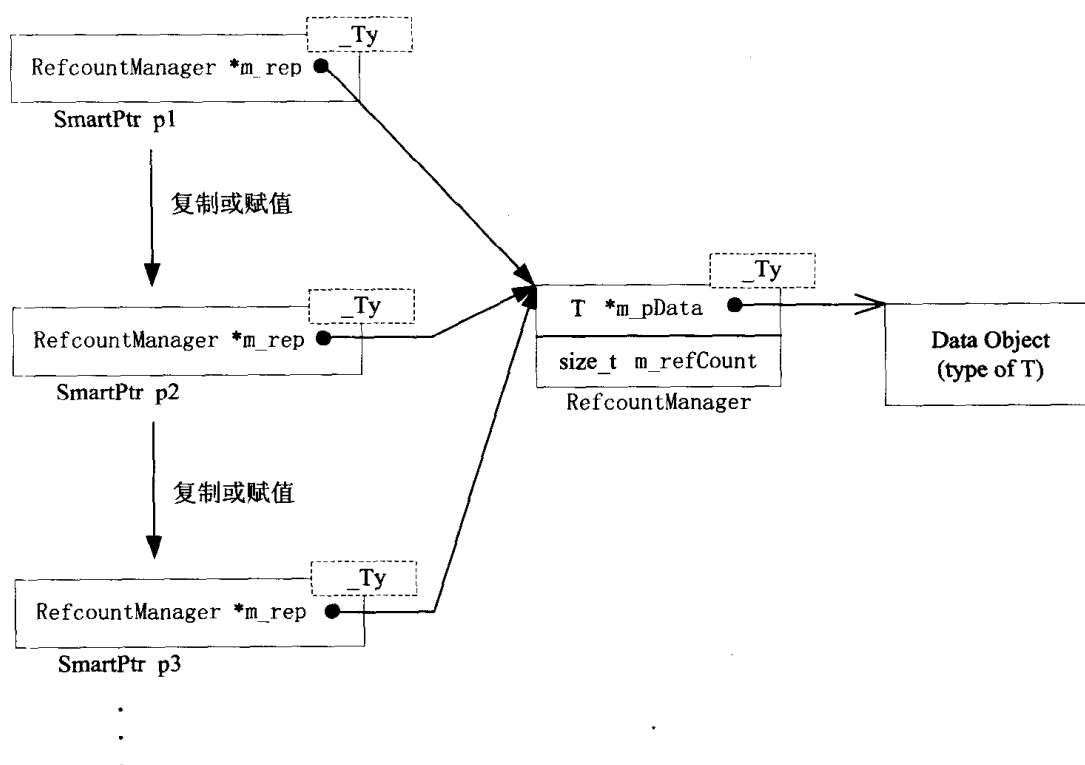


图 16-1 智能指针对象模型

而 `CComPtr<>` 则是一个依赖于侵入式引用计数器的智能指针（参见 ATL 头文件 `<ATLBASE.H>`），它用于接管 COM 对象的接口指针，要求被接管的 COM 接口（Interface）必须派生自类似于接口 `IUnknown` 类，即至少提供 `AddRef()`、`Release()` 和 `QueryInterface()` 三个成员函数。

我们来比较一下普通指针、`auto_ptr` 及 `SmartPtr` 的特点，如表 16-2 所示。

表 16-2 指针比较

特点 \ 指针类别	普通指针	<code>auto_ptr</code>	引用计数智能指针
是否共享所指对象及拥有权	是	否	是
是否自动释放所指对象	否	是	是
是否具有拥有权管理能力	否	是	有
拥有权是否会转移	共享	是	共享
是否适合作为容器元素	是	否	是
是否可作为类成员	是	是	是
是否会失效	是	是	否
所指对象是否应该动态创建	不一定	如果特化了对象销毁方法， 则不一定必须动态创建	可由分配和释放的 Policy 模板 参数类来决定

## 16.15 智能指针作为容器元素

对 C++ 语言本身来说，它并不在乎用户把什么类型的对象作为 STL 容器的元素，因为模板类型参数在理论上可以为任何类型。比如说 STL 容器仅支持“值”语义而不支持“引用（&）”语义，这并非因为模板类型参数不能为引用，而是因为如果容器元素为引用类型，就会出现“引用的引用”、“引用的指针”等 C++ 语言不支持的语法和语义。

直接以普通指针作为容器的元素时，许多人很容易在这上面犯糊涂，尤其是当元素类型为 `char*` 或 `const char*` 时，他们总以为容器中存放的是字符串。我们知道，指针就是一个地址值，因此以指针为元素容器存放的就是一些内存地址，而不是真正数据。这种容器只负责指针元素本身的内存动态分配和释放，而不会负责指针元素指向对象的内存管理任务，因为那是程序员的责任。比如定义一个 `vector<char*>` 的对象，然后插入一些元素，如示例 16-22 所示。

示例 16-22

```
void test()
{
    typedef std::vector<char*> MyStrVector;
    MyStrVector strVect;           // 空容器
    strVect.reserve(10);          // 预留 10 个指针空间
    for (int i = 0; i < 10; i++) {
        char *p = new char[i + 1];
        ::memset(p, 'H', i + 1);
        *(p + i) = '\0';
        strVect.insert(strVect.end(), p); // 新建指针元素，并用 p 来初始化
    }
    ...
    for(int j = 0; j < 10; j++)
        delete [] strVect[j];          // 用户负责释放存储
} // strVect 的析构函数只负责释放指针元素的内存
```

千万不要以为容器的析构函数会为你释放指针元素指向的动态对象。要想让容器自动帮你管理内存，就得把 `char*` 封装成 `string` 类，并为它定义“Big-Three”（参见第 13 章）。指针数组也是一样的道理。`strVect` 的内存映像如图 16-2 所示。

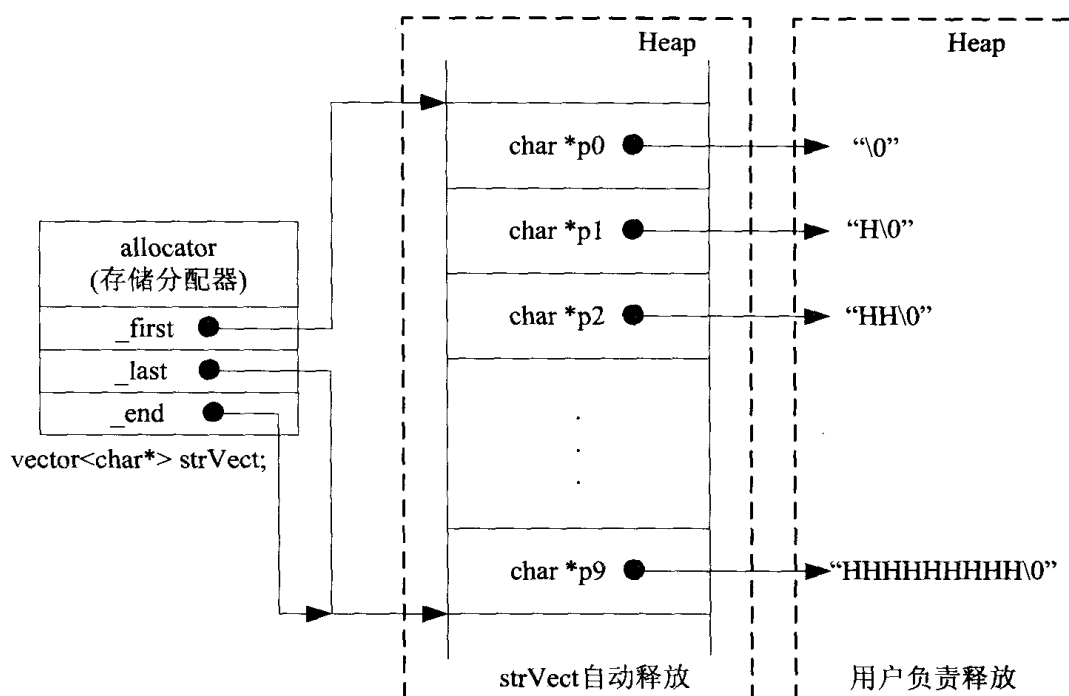


图 16-2 strVect 的内存映像

智能指针是一种模拟原始指针行为的对象，因此理论上也可以作为容器（包括数组）的元素，就像原始指针可以作为容器的元素一样。但是智能指针毕竟是一种特殊的对象，它们在原始指针共享实值对象的基础能力上普遍增加了自动销毁实值对象的能力，而且其实现方式和种类都非常多，如果不分好坏地都将它们作为容器的元素，可能导致容器之间共享元素对象实值，这不仅不符合 STL 容器的概念和“值”语义，而且会存在安全隐患，同时也会存在许多应用上的限制，特别是像 STL 中 `auto_ptr` 这样的智能指针。

Scott Meyers 在《More Effective C++》一书中对智能指针及其相关问题（构造、析构、拷贝、反引用、测试以及类型转换等）作了深入的分析，其中也提到“STL 的 `auto_ptr` 这种在拷贝时会把对实值对象的拥有权转交出去的智能指针不宜作为 STL 容器的元素”，而且在他的《Effective STL》Item 8 中明确指出了这一点。Nicolai M. Josuttis 的《The C++ Standard Library》中有一节专门针对 `auto_ptr` 的阐述，也指出“`auto_ptr` 不满足 STL 标准容器对元素的最基本要求”。但是他们都是从容器的需求、语义以及应用的安全性来阐述，而没有从语言的静态类型安全性和 `auto_ptr` 实现方案的角度深入地分析其原因，因此有些读者看了之后可能仍然不明就理：它是如何不满足容器需求的？它是如何违反 C++ 的静态类型安全性从而避免误用的？

我们知道，可以作为 STL 容器元素的数据类型一般来说需要满足下列条件。

(1) 可默认构造的 (Default Constructible)，也即具有 `public` 的 `default constructor`，不论是用户显式定义的还是编译器默认的。但是用户定义的带参数的 `constructor`（包括 `copy constructor`）会抑制编译器合成 `default constructor`。实际上并非任何情况下任意一种容器都强制要求其元素类型满足这一条件，特别是关联式容器，因为只有序列式容器的某些成员函数才可能明确地或隐含地使用元素类型的 `default constructor`，

如果你不调用这样的成员函数，编译器就不需要元素类型的默认构造函数。

(2) 可拷贝构造 (Copy Constructible) 和拷贝赋值 (Copy Assignable) 的，即具有 public 的 copy constructor 和 copy assignment operator，不论是编译器默认的还是用户显式定义的。其他版本的 operator=() 重载并不会抑制编译器合成 copy assignment operator，如果没有显式定义它的话。这个条件可归结为：元素必须是可拷贝的 (Copyable)，但实际上拷贝赋值的要求也不是强制的，原因和 default constructor 类似。

(3) 具有 public 的 destructor，不论是编译器默认的还是用户显式定义的。

(4) 对于关联式容器，要求其元素必须是可比较的 (Comparable)。

auto\_ptr 满足上述条件吗？至少满足前三条，因此至少可以作为序列式容器的元素。如果为 auto\_ptr 定义了比较运算符的话，应该还可以把它作为关联式容器的元素。

但是 auto\_ptr 的特点是接管和转移拥有权，而不是像原始指针那样可以共享实值对象，即：auto\_ptr 在初始化时接管实值对象和拥有权，而在拷贝时（拷贝构造和拷贝赋值）会交出实值对象及其拥有权。因此，auto\_ptr 对象和它的拷贝不会共享实值对象，任何两个 auto\_ptr 也不应该共享同一个实值对象。这就是说，auto\_ptr 对象和它的拷贝并不相同。然而根据 STL 容器“值”语义的要求，可拷贝构造意味着一个对象必须和它的拷贝相同（标准中的正式定义比这稍复杂一些）。同样，可赋值意味着把一个对象赋值给另一个同类型对象将产生两个相同的对象。显然，auto\_ptr 不能满足这一要求，它与上面的结论矛盾！

那么问题究竟出在哪里呢？

在揭开 auto\_ptr 的神秘面纱之前需要了解 copy constructor 和 copy assignment operator 的几种合法形式。任何一个类都允许两种形式的 copy constructor (C 代表任何一个类)：

```
C(const C& copy);  
C(C& copy);
```

同样，copy assignment operator 也允许类似的两种形式（返回值类型实际需要可改变）：

```
C& operator=(const C& copy);  
C& operator=(C& copy);
```

实际上，由于 copy assignment operator 为普通的运算符重载成员函数，因此还可以定义下列形式的赋值函数：

```
C& operator=(C copy);
```

这两个函数具体是什么形式，取决于用户的定义或者该类的成员对象及其基类具有什么样的 copy constructor 和 copy assignment operator。比如，如果基类的 copy constructor 为第一种形式，那么编译器自动为派生类合成的 copy constructor 也为第一种形式；相反为第二种形式。Copy assignment operator 亦类似。

这两种形式的区别就在于参数有无修饰符 const：如果有 const 修饰，则该函数

体不能修改实参对象（即拷贝源），也不能调用其 non-const 成员函数；如果没有 const 修饰，则该函数可以修改实参对象，也可以调用其 non-const 成员函数。

如果要防止用户把一些不适宜的对象放入容器中，就要求对象的设计和实现者使用一些语言支持但不常用的特征。也就是说，要能够在编译阶段就阻止这种具有潜在危险性的行为。常用的方法就是迫使其违反 C++ 静态类型安全规则。我们就来看看 auto\_ptr 到底是如何通过迫使其违反 C++ 静态类型安全规则而在编译时阻止将其作为容器元素的。

其实 auto\_ptr 的拥有权管理非常简单。根据上面的阐述，可以使用两种方案来实现 auto\_ptr。下面是拷贝构造函数和拷贝赋值函数采用 non-const 参数的一个实现版本：

---

```

template<class T>
class auto_ptr {
private:
    T      *m_ptr;                // 原始指针
public:
    explicit auto_ptr(T *p = 0) throw()    // explicit constructor
        : m_ptr(p) {}                // *p 必须是运行时创建的对象
    auto_ptr(auto_ptr& other) throw()      // 非常规 copy constructor
        : m_ptr(other.release()) {}      // 转让拥有权，修改了实参对象
    auto_ptr& operator=(auto_ptr& other) throw() { // 非常规 assignment
        if (&other != this) {
            delete m_ptr;                // 释放实值对象
            m_ptr = other.release();      // 交出拥有权，修改了实参对象
        }
        return (*this);
    }
    // 从析构函数看，m_ptr 必须指向动态创建的对象
    ~auto_ptr() { delete m_ptr; }        // destructor, "delete 0;" 没有任何问题！
    T& operator*() const throw() { return *m_ptr; }
    T* operator->() const throw() { return m_ptr; }
    T* get() const throw() { return m_ptr; }
    T* release() throw() {
        T *temp = m_ptr;
        m_ptr = 0;                    // 必要！修改成员，释放拥有权
        return temp;
    }
    void reset(T *p = 0) throw() {
        if (p != m_ptr) {
            delete m_ptr;
            m_ptr = p;
        }
    }
}

```

---

```
bool owns() const { return (m_ptr != 0); }
...
};
```

// 这里省略了一些无关紧要的东西

如你所见，该 `auto_ptr` 实现版本的 `copy constructor` 和 `copy assignment operator` 参数类型都是 `non-const` 的，因为这两个函数都会修改实参对象的数据成员，即调用其 `release` 方法（`non-const` 方法）释放其对实值对象的拥有权，并把实值对象的指针置为 0。如果参数类型为 `const` 的，那么这种修改就不可能直接进行了。所以，一旦用一个 `auto_ptr` 对象去构造另一个 `auto_ptr` 对象，或者把一个 `auto_ptr` 对象赋值给另一个 `auto_ptr` 对象，你就不能再使用原来的那个 `auto_ptr` 对象了，因为反引用 `NULL` 指针会导致运行时异常，除非你让它重新接管一个新的实值对象。

这个版本的 `auto_ptr` 就不能作为任何容器的元素，如果你这样做了，在编译阶段就会检查出错误，即违反了 C++ 的静态类型安全规则。如示例 16-23 所示。

示例 16-23

```
std::list< std::auto_ptr<int> > la;    // auto_ptr 列表
std::auto_ptr<int> p1(new int(1));
std::auto_ptr<int> p2(new int(2));
std::auto_ptr<int> p3(new int(3));
la.push_back(p1);    // compiling-error!
la.push_back(p2);    // compiling-error!
la.push_back(p3);    // compiling-error!

set<auto_ptr<int> > sa; // auto_ptr 集合：假设为 auto_ptr 定义了 operator<
sa.insert(p1);        // compiling-error!
sa.insert(p2);        // compiling-error!
sa.insert(p3);        // compiling-error!
```

STL 容器管理元素的方法是动态拷贝元素，并负责管理这些动态分配的资源，即值的深拷贝语义（`deep copy`），具体由一个可定制的 `memory allocator` 来负责，不过这不是我们讨论的重点。`std::list<T>::push_back` 方法的实际动作如下：

```
template<typename T>
void list<T>::push_back(const T& x)
{
    T *p = operator new(sizeof(T));    // 分配内存空间
    new (p) T(x);                      // placement new, 调用 T 的 copy constructor
    .....                             // 将 p 交给容器管理，调整容器大小
}
```

由于 `auto_ptr` 的 `copy constructor` 被显式地定义为接受 `non-const&`，因此上述函数实现就需要将一个 `const T& x` 转换为 `non-const&`，这显然是违反静态类型安全规则的。STL 容器不能使用强制类型转换来帮你达到此目的，否则它本身就不是类型安全的了。



其他追加元素的方法比如 insert 的某些版本，也是一样的道理。

上述 auto\_ptr 通过采用非常规 copy constructor 和 copy assignment operator 使“企图将 auto\_ptr 对象作为 STL 容器元素”的行为在编译阶段就被检测出来，从而避免了潜在的危险。

然而如果 auto\_ptr 采用常规 copy constructor 和 copy assignment operator 形式，编译器就无能为力了，因为它们不违反 C++ 静态类型安全规则。STL 的 P.J.Plauer 实现版本（被 MS VC++ 采用）的 auto\_ptr 就是一个例子！

P.J.Plauer 版本的 auto\_ptr 确实可以作为容器的元素，这并不是因为它没有修改拷贝源的拥有权，而是它的 release 函数虽然是 const member function，却在修改拥有权时使用了 const\_cast<>() 强制类型转换。因为在一个 const member function 里面，编译器把当前对象看成是一个 const 对象（即 this 的类型为 const auto\_ptr<T> \* const），调用 copy constructor 时通过强制类型转换就可以修改实参对象的拥有权属性，尽管它是 const & 传递。

下面是拷贝构造函数和拷贝赋值函数采用 const 参数的一个实现版本：

---

```

template<class T>
class auto_ptr {
private:
    T      *m_ptr;                // 原始指针
public:
    explicit auto_ptr(T *p = 0) throw()    // explicit constructor
        : m_ptr(p) {}                // *p 必须是运行时创建的对象
    auto_ptr(const auto_ptr& other) throw() // 常规 copy constructor
        : m_ptr(other.release()) {}    // 转让拥有权，修改了实参对象
    auto_ptr& operator=(const auto_ptr& other) throw() { // 常规 assignment
        if (&other != this) {
            delete m_ptr;                // 释放实值对象
            m_ptr = other.release();     // 交出拥有权，修改了实参对象
        }
        return (*this);
    }
    // 从析构函数看，m_ptr 必须指向动态创建的对象
    ~auto_ptr() { delete m_ptr; }       // destructor, “delete 0;” 没有任何问题！
    T& operator*() const throw() { return *m_ptr; }
    T* operator->() const throw() { return m_ptr; }
    T* get() const throw() { return m_ptr; }
    T* release() const throw() {
        T *temp = m_ptr;
        ((auto_ptr<T>*)this)->m_ptr = 0; // 必要！修改成员，释放拥有权
        return temp;
    }
    void reset(T *p = 0) throw() {
        if (p != m_ptr) {
            delete m_ptr;

```

```
        m_ptr = p;
    }
}
bool owns() const { return (m_ptr != 0); }
...    // 这里省略了一些无关紧要的东西
};
```

一旦如此实现, `auto_ptr` 容器就可以顺利通过编译并可能正确执行, 如示例 16-24 所示。

示例 16-24

```
int main()
{
    typedef std::list<std::auto_ptr<int>> IntPtrList;
    IntPtrList la; // or: std::vector<std::auto_ptr<int>> va;
    std::auto_ptr<int> p1(new int(1));
    std::auto_ptr<int> p2(new int(2));
    std::auto_ptr<int> p3(new int(3));
    std::auto_ptr<int> p4(new int(4));
    std::auto_ptr<int> p5(new int(5));
    la.push_back(p1);           // ok! 转交所有权
    la.push_back(p2);           // ok! 转交所有权
    la.push_back(p3);           // ok! 转交所有权
    la.push_back(p4);           // ok! 转交所有权
    la.push_back(p5);           // ok! 转交所有权
    // 不能再使用 p1、p2、p3、p4、p5
    for (IntPtrList::const_iterator first = la.begin(),
         last = la.end(); first != last; ++first)
        std::cerr << **first << '\t';
    // 不能再使用 p1、p2、p3、p4、p5
    return 0;
} // la 析构的时候会自动调用每一个 auto_ptr 元素的析构函数,
// 从而保证释放动态分配的内存
```

输出:

1 2 3 4 5

但是把 `auto_ptr` 作为容器元素毕竟是一个危险的动作, 而且这样的容器在使用时会受到很大的限制。如果上面的程序接着使用 `la`, 比如创建它的拷贝元素, 调整它的大小, 甚至对它排序, 那么 `la` 就可能遭到破坏, 它的所有元素会变成无效指针, 或者里面夹杂了无效指针, 甚至有可能丢失一些元素, 而你却没有意识到。如示例 16-25 所示, 假设为 `auto_ptr` 定义了泛型比较运算符。

示例 16-25

```
int main()
{
    typedef std::vector<std::auto_ptr<int>> IntPtrVector;
    IntPtrVector va;
    std::auto_ptr<int> p1(new int(1));
    std::auto_ptr<int> p2(new int(2));
    std::auto_ptr<int> p3(new int(3));
    std::auto_ptr<int> p4(new int(4));
    std::auto_ptr<int> p5(new int(5));
    va.push_back(p1);
    va.push_back(p2);
    va.push_back(p3);
    va.push_back(p4);
    va.push_back(p5);

    // (注意: 以下操作并非放在一起进行, 仅是示范)
    IntPtrVector vb = va;           // va 丧失对所有实值对象的拥有权,
                                    // 元素成为 NULL 指针
    vb.resize(10);                  // 新增的元素都为 NULL 指针
    std::sort(vb.begin(), vb.end()); // 可能会使其中某些元素成为 NULL 指针
    std::auto_ptr<int> t = vb.front(); // 改变了容器元素
    std::auto_ptr<int> r = vb[3];    // 改变了容器元素

    std::list<std::auto_ptr<int>> la;
    std::copy(vb.begin(), vb.end(), std::back_inserter(la));
                                    // copy 改变了拷贝源

    return 0;
}
```

Scott Meyers 在《Effective STL》Item 8 中详细地分析了对 `auto_ptr` 容器进行排序时可能会导致的问题。但是在 MS VC++ 环境下经测试, 并没有出现书中所描述的悲惨结局, 而是结果正确。主要原因在于 C++ 标准并没有要求 `std::sort` 等泛型算法的实现必须采用某一种方法, 而是只规定了它们的接口、功能和应该达到的性能要求 (容器也是如此)。因此, 不同的 STL 实现可能采取不同的方法, 比如有的 `sort` 实现采用快速排序法, 而有的采用插入式排序法等。不同的排序方法在遭遇 `auto_ptr` 这样的容器时可能就会产生不同的结果。

P.J.Plauger 版本在这方面的防范能力确实不如 SGI 版本做得好! 不过没关系, STL 的源代码都是公开的, 你可以比较不同的实现甚至修改它们, 使之更安全、更适合你的应用。

应该说, 从应用的方便性和安全角度出发, 容器应该要求其元素对象的拷贝与原对象相同或者等价, 但 `auto_ptr` 显然不满足这一条。`auto_ptr` 作为容器元素其危险性主要表现在如下几个方面。

(1) 将 `auto_ptr` 对象插入容器中之后企图继续使用它, 比如通过它调用实值对象的成员函数, 然而此时它指向的实值对象已经交给容器中的某一个元素对象了。

(2) 就像 `auto_ptr` 和它的拷贝并不相同一样, `auto_ptr` 容器和它的拷贝也不一样, 如果对某些成员函数的返回结果使用不当的话, 可能无意中会产生不期望的行为。会使其应用受到很大限制, 必须小心应付。

(3) 某些算法将无法用于这样的容器。比如 `sort` 等会修改区间的算法, 因为它们实现调用元素对象的 `copy constructor` 或 `copy assignment operator`, 可能会释放掉某些元素对实值对象的拥有权。就连本来不会修改源区间的算法如 `copy`, 如果应用于 `auto_ptr` 容器, 也可能修改源区间。还有的算法比如 `find` 等要求元素对象提供比较能力, 如果 `auto_ptr` 不是可比的, 那也不能用于容器。

(4) 不可移植。目前有些 STL 实现比如 SGI 版本可以在编译阶段阻止这种行为, 但是某些 STL 实现仍然允许这样做。

鉴于此, 无论你使用的 STL 平台是否允许 `auto_ptr` 容器, 你都不应该这样做。

然而许多其他的功能强大的智能指针, 比如使用了引用计数的智能指针 (参见 16.14 节), 作为容器的元素时不会存在上述问题, 但是 `auto_ptr` 不是这样的智能指针。关于智能指针的更详细阐述还可参考 Andrei Alexandrescu 的《Modern C++ Design》一书。

可见, 智能指针“可以”还是“不可以”作为容器的元素并非绝对的, 不仅与 STL 的实现有关, 而且与 STL 容器的需求和安全性及容器的语义有关。

既然 `auto_ptr` 在拷贝或赋值时会使原来的 `auto_ptr` 失效, 那么我们只要防止其拷贝和赋值行为的发生就可以了。比如在传递 `auto_ptr` 对象时使用 `const &` 或 `const *` 传递而不是值传递。如示例 16-26 所示。

示例 16-26

```
void func(const auto_ptr<int>& pInt)
{
    cout << *pInt << endl;
}
int main()
{
    auto_ptr<int> a(new int(100));
    func(a);
}
```

但即使这样, 如果遇到像 P.J.Plauger 那样实现的 `auto_ptr`, 还是不能保证在函数内部不会出现对它的拷贝或者赋值。

不要用静态创建的对象来初始化 `auto_ptr`, 如示例 16-27 所示。

示例 16-27

```
int main()
```

```
{  
    int x(100);  
    auto_ptr<int> a(&x);  
} // 这里调用 delete 删除本地对象，错误！
```

由于 `auto_ptr` 是对象化的智能指针，具有自动释放资源的能力，因此它真正有价值的用途是在发生异常时避免资源泄漏。比如，如果不使用 `auto_ptr`，则下列代码在发生异常的情况下不得不多次手工释放资源（见示例 16-28）。

示例 16-28

```
class A { ... };  
void func()  
{  
    ...  
    A *pA = new A;  
    try {  
        ... // using *pA  
    }  
    catch(...) {  
        delete pA; // 发生异常时要显式释放  
        throw;  
    }  
    delete pA; // 函数退出时还要显式释放  
}
```

现在有了 `auto_ptr`，我们就可以这么做（见示例 16-29）。

示例 16-29

```
class A { ... };  
void func()  
{  
    ...  
    auto_ptr<A> pA(new A);  
    ... // using *pA  
}
```

这是因为 C++ 有一个保证：本地对象在函数退出时总是会被销毁，而不论函数以何种方式退出。也就是说，不管是在发生异常的情况下函数退出，还是函数的正常退出，堆栈都要展开，每一个本地对象的析构函数都会被依次调用。

如果想防止无意中修改 `auto_ptr` 对实值对象的拥有权，可以使用 `const auto_ptr`，这样的 `auto_ptr` 只能使用引用或指针传递，不能使用值传递，也不能赋值和拷贝构造。如示例 16-30 所示。

示例 16-30

```
class A{ ... };
    void func()
    {
        ...
        const auto_ptr<A> p1(new A);
        ...                                     // using *pA
        auto_ptr<A> p2(p1);                     // error!
        auto_ptr<A> p3;
        p3 = p1;                                // error!
    }
```

关于 `auto_ptr` 的运用技巧可参考《The C++ Standard Library》（Nicolai M. Josuttis, 1999）中的相关章节。

至于带有引用计数的智能指针，就没有那么多悬念了。如果你已经明白 `SmartPtr` 的原理和所有实现技术，那么将 `SmartPtr` 作为容器的元素，根本不需要什么再多的技术原理。在这里再举一个例子（见示例 16-31）。

示例 16-31

```
typedef SmartPtr<Shape>  ShapeSmartPtr;
typedef std::list<ShapeSmartPtr>  ShapeList;

bool operator<(const Point& left, const Point& right)
{ return ((left.m_x < right.m_x) && (left.m_y < right.m_y)); }
bool operator==(const Point& left, const Point& right)
{ return ((left.m_x == right.m_x) && (left.m_y == right.m_y)); }
bool operator<(const ShapeSmartPtr left, const ShapeSmartPtr right)
{ return (left->GetOrigin() < right->GetOrigin()); }
bool operator==(const ShapeSmartPtr left, const ShapeSmartPtr right)
{ return (left->GetOrigin() == right->GetOrigin()); }

ShapeList  shapes;
ShapeSmartPtr  p(new Shape(Point(1, 1)));
ShapeSmartPtr  q(new Circle(Point(2, 2), 5));
ShapeSmartPtr  r(new Rectangle(Point(3, 3), Point(4, 4)));
shapes.push_back(p);
shapes.push_back(q);
shapes.push_back(r);

std::sort(shapes.begin(), shapes.end());           // 智能指针排序
for (ShapeList::const_iterator first = shapes.begin(); first != shapes.end(); ++first)
    (*first)->Draw();
```

## ❧ 一些心得体会 ❧

---

作者认识不少技术不错的 C++/C 程序员，很少有人能拍拍胸脯说通晓指针与内存管理。作者最初学习 C 语言时特别怕指针，导致在开发第一个应用软件（约 1 万行 C 代码）时不敢使用一个指针，全用数组来代替，实在蠢得过分。躲避指针不是办法，后来作者改写了这个软件，该用指针的地方就用指针，结果代码量减少到原来的一半。

作者的经验教训是：

（1）越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。

（2）必须养成“使用调试器逐步跟踪程序”的习惯，只有这样才能发现问题的根源。

---





## 第 17 章 学习和使用 STL

一般地，当我们掌握一门语言的语法之后，更多的时间都是在学习和使用标准库。函数库和类库虽然不是语言本身的组成部分，但是它们对应用软件开发的价值实在是太大了。试想一下，如果不使用 C 函数库和 C++ 标准库，让我们自己编写那些将要用到的库函数、容器类及算法，这种难度和工作量是无法想象的！

C++ 标准库不仅仅包含 STL，还包含标准 C 函数库、I/O 流、串、异常、数值计算、国际化与本地化等的支持，其中大都用模板技术进行了泛型实现。

许多程序员对 C++ 的泛型编程和 STL 比较陌生，甚至没有意识到它们的存在，本章将把 C++ 程序员引领到这个“魔幻”般的世界中，并期待您能有所收获。

### 17.1 STL 简介

STL (Standard Template Library) 是 C++ 标准库的最主要和最重要的组成部分。C++ 的库在标准化之前，仅包含一些 I/O 组件、string 类以及 C 函数库等极少的内容，而且还不是模板化的。模板的概念被提出和实现以后，开发一些实用的泛型容器和泛型算法成为可能。C++ 标准化委员会在 1993 年接受了 Alexander Stepanov 的建议，给标准 C++ 增加了 STL。正如 Bjarne Stroustrup 所说的那样：“自从 1991 年以来，对 C++ 最重要的改变不是语言本身的改变，而是增加了标准库。”

为什么标准化的模板库如此重要呢？因为：

- (1) 它可用来创建动态增长和减小的数据结构；
- (2) 它是类型无关的，因此具有很高的可复用性；
- (3) 它在编译时而不是运行时进行数据类型检查，保证了类型安全；
- (4) 它是平台无关的，因此保证了最大的可移植性；

(5) 它还可用于基本数据类型，包括指针和引用，这对那些必须使用类层次结构来构造应用程序的语言来说是没法比的。

STL 是一个标准规范，它只是为容器、迭代器和泛型算法等组件定义了一整套统一的上层访问接口及各种组件之间搭配运用的一般规则，而没有定义组件底层的具体实现方法，因此不同的软件供应商都可以提供自己的 STL 实现版本，如 Microsoft、Borland、HP、SGI 等的实现版本，虽然这些版本之间可能存在某些差异（比如底层存储结构和效率差异），但是概念和接口都是一致的。基于这样的认识，我们可以把 STL 看做是一个概念模型库或者是一个组件架构。STL 内含的标准组件分类思想，无论对程序员学习和运用 STL 还是对研究和扩展 STL 都是极有意义和帮助的。

STL 主要包括下面这些组件：I/O 流、string 类、容器类（Container）、迭代器（Iterator）、存储分配器（Allocator）、适配器（Adapter）、函数对象（Functor）、泛型算法（Algorithm）、数值运算、国际化和本地化支持，以及标准异常类等。其中最主要的组件就是容器、存储分配器、迭代器、泛型算法、函数对象和适配器，俗称“六大组件”。容器类相当于数据结构，算法用于操作容器中的数据，而迭代器则是它们之间的一个桥梁。这些组件之间的相互关系如图 17-1 所示。

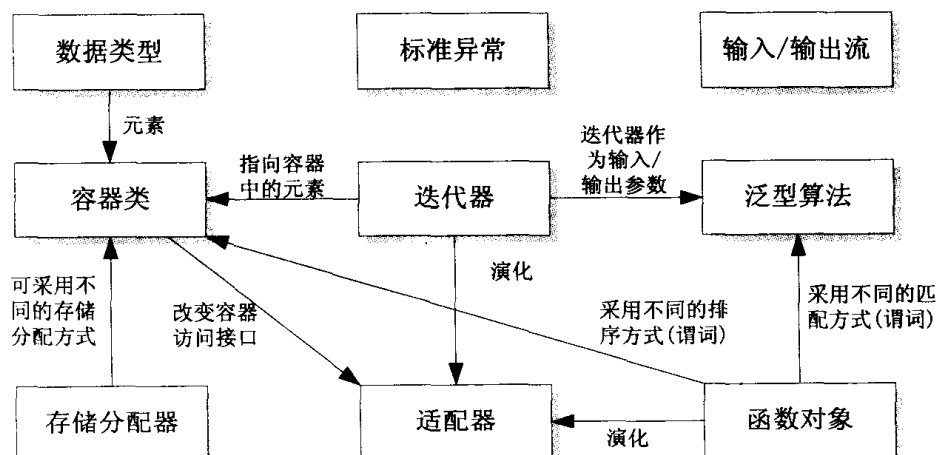


图 17-1 STL 各组件之间的关系

学习 STL，必须首先掌握其设计思想：总体架构思想及每一类组件的设计思想。本章仅阐述一些主要的和常用的组件原理，通晓了这些基本组件，其他组件的学习和运用都不在话下。如果读者想对图 17-1 有一个清晰的认识，有必要钻研一下 STL 的源码。

## 17.2 STL 头文件的分布

C++标准规定，STL 的头文件都不使用扩展名，但是某些实现版本可能没有遵循这个规定，或者在标准 STL 头文件上又增加了一层包装。过去的 C 程序库头文件在并入 C++标准库时也都去掉了.h 扩展名，同时增加了前缀“c”。

STL 源代码头文件（实际上实现都是内联的）一般都存放在开发环境的 include 目录下。STL 组件都被纳入了名字空间 std::，所以在使用其中的组件之前需使用 using 声明或 using 指令，或者也可以在每一处都直接使用完全限定名 std::。但是过去的 C 程序库（包括不带扩展名的和带扩展名的），其中的内容仍然位于全局名字空间中（即没有名字空间）。

### 17.2.1 容器类

容器类定义在表 17-1 所示的头文件中。关联式容器 multimap 和 multiset 也都分别定义在<map>和<set>中，hash\_multimap 和 hash\_multiset 定义在<hash\_map>和<hash\_set>中。

表 17-1 STL 容器的头文件

头 文 件	内 容
<vector>	元素类型为 T 的向量，包括了特化 vector<bool>
<list>	元素类型为 T 的双向链表
<deque>	元素类型为 T 的双端队列
<queue>	元素类型为 T 的普通队列，包括了 priority_queue
<stack>	元素类型为 T 的堆栈
<map>	元素类型为 T 的映射
<set>	元素类型为 T 的集合
<bitset>	布尔值的集合（实际不是真正意义上的集合）
<hash_map>	元素类型为 T 的 hash 映射
<hash_set>	元素类型为 T 的 hash 集合

### 17.2.2 泛型算法

只要是由一系列元素构成的结构原则上都可以应用泛型算法，像 C++/C 数组、字符串、I/O 流等特殊的容器也可以使用某些泛型算法——它们定义在头文件 <algorithm> 和 <utility> 中。

### 17.2.3 迭代器

迭代器就是用来遍历元素序列或元素集合的“通用指针”，但是每一种容器都定义了适合自己使用的迭代器，那些具有特殊功能的迭代器，如输入/输出迭代器、插入迭代器、反向迭代器等都是迭代器适配器，定义在头文件 <iterator> 中。

### 17.2.4 数学运算库

STL 有几个专门为数学运算设计的类和算法，定义在如表 17-2 所示的头文件中。

表 17-2 STL 数学运算库

头 文 件	内 容
<complex>	复数及其相关操作
<valarray>	数值向量及其相关操作
<numerics>	通用数学运算
<limits>	常用数值类型的极限值和精度等

### 17.2.5 通用工具

表 17-3 中的头文件定义了 STL 容器和泛型算法中用到的辅助组件，有标准的函数对象、pair<>、auto\_ptr<> 类等。

表 17-3 STL 通用工具

头 文 件	内 容
<utility>	运算符重载和 pair 定义
<functional>	标准的函数对象及其便捷函数定义
<memory>	存储分配器和 auto_ptr 类

## 17.2.6 其他头文件

除了上面这些头文件外，还有一些经常使用的组件，它们分别定义在下面这些头文件中（从名称上就可以知道它们是做什么用的）：<typeinfo>、<stdexcept>、<strstream>、<string>、<istream>、<ostream>、<iostream>、<new>、<iomanip>、<fstream>，等等。

## 17.3 容器设计原理

### 17.3.1 内存映像

生活中的容器是无处不在的，比如抽屉、水缸、菜篮子等，所不同的是应用领域和实现方式。容器就是能够容纳其他对象作为其元素的对象，例如数学中的集合就是一种容器概念。在计算机上实现类似的数学模型时要考虑各种因素，比如：容器元素对象存储在哪里，采用什么存储结构，容器对象如何指示其每一个元素对象的位置（即通过容器对象定位其每一个元素对象），如何进行元素的插入、删除和修改等操作，如何进行容器的遍历，等等。

由于容器在概念上是一种可以动态增大和减小的模型，所以其元素对象在实现上不可能直接保存在容器对象里面，而应该保存在自由内存（Free Memory）或堆（Heap）上。这里要区分两个概念：“容器对象”和“容器元素对象”。容器本身就是一个 C++ 类的对象，其大小在运行时不可能改变，因此容器必须有办法指示其每一个元素对象在内存中的位置（它们的个数和位置可能经常变动），以使用户能够通过容器对象找到其中的元素对象。

我们先介绍一个典型的容器 `std::vector<T>` 的内存映像，来展示 C++ 中 STL 是如何实现容器的，如图 17-2 所示。

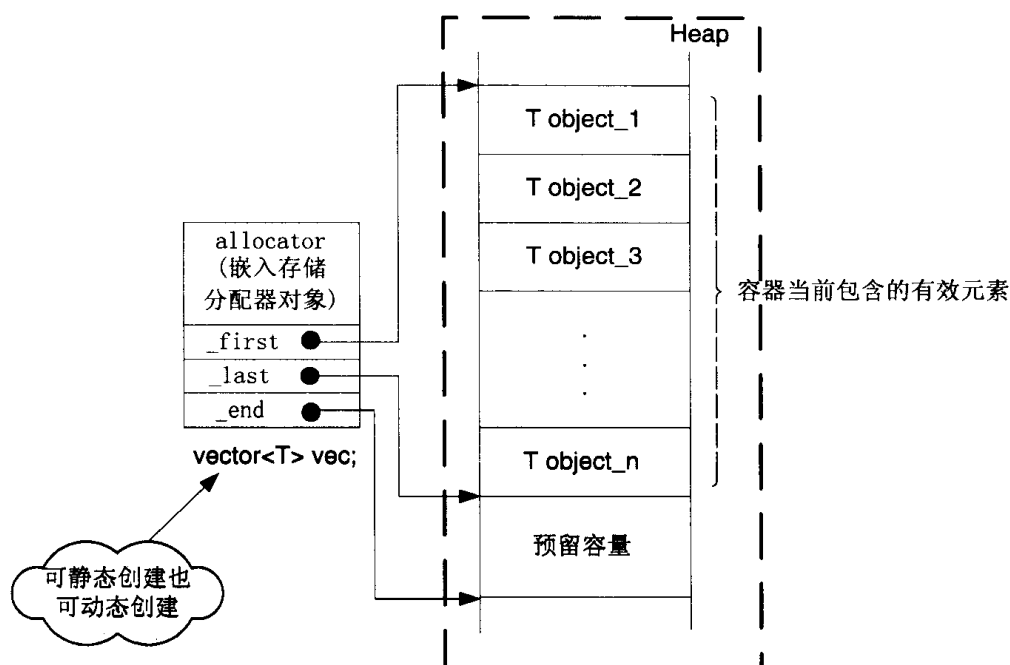


图 17-2 vector 容器对象及其元素的内存映像

不同容器的存储模型一般互不相同，但是容器对象和其元素对象之间的关系是类似的。STL 容器的实现方式可以归纳为如图 17-3 所示的层次模型。

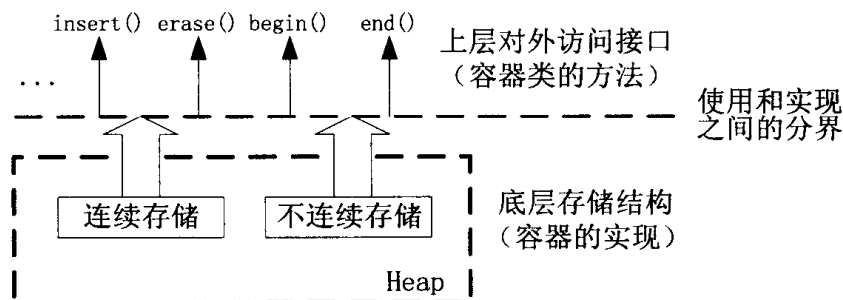


图 17-3 STL 容器设计基本模型

图 17-3 这个模型是不是很简单呢？在 C++ 中，我们使用信息隐藏和封装技术把数据隐藏在类内部不许外部直接操作，同时提供访问器（如 `get_xxx` 成员函数）和修改器（如 `set_xxx` 成员函数）。STL 容器的设计原理如出一辙，只是它们在实现时考虑的问题更多、更复杂而已。容器不仅把元素对象隐藏了起来，而且把元素对象的内存申请和释放操作也全部隐藏了起来（通过存储分配器），这就使程序员彻底摆脱了直接操纵底层内存指针的麻烦，也避免了危险。经过严格测试的容器类几乎不可能存在动态内存管理上的问题，因此可以放心使用。

### 17.3.2 存储方式和访问方式

向量（vector）和链表（linked list）是两种最基本的动态结构，也是 STL 中两种最基本的容器，分别对应动态数组和链接表结构，同时它们分别代表了内存中同类

型批量数据存放的两种基本方式：连续存储和随机存储（不连续存储）。

不同的存储方式决定了元素的不同访问方式，即随机访问和顺序访问。所谓随机访问就是指可以直接通过开销恒定的算术运算来得到任一元素的内存地址的访问方法；而顺序访问则是指必须从第一个元素开始遍历，直到找到所需的元素对象为止，而无法直接得到任一中间元素对象的地址。C++/C 的内置数组和 `vector` 都是既可以随机访问又可以顺序访问的容器，而 `list` 则只能顺序访问。我们看一下 STL `std::list` 的内存映像，如图 17-4 所示。

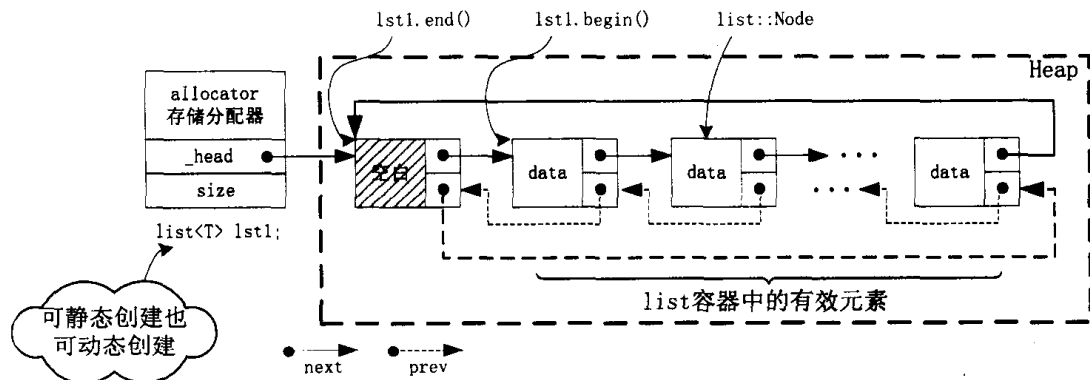


图 17-4 `list` 容器对象及其元素的内存映像

可以这样说，只要底层存储机制采取连续存储方式的容器，就可以随机访问其中任一元素对象，否则只能顺序访问；而任何容器都可以顺序访问，即遍历。

**注意：**`stack`、`queue` 及 `priority_queue` 在概念和接口上都不支持随机访问和遍历，这是由它们的语义决定的，而不是由底层存储方式决定的，因此没有迭代器（所以它们才被叫做容器适配器而不是归为容器类）。

从这两种基本的存储方式可以演变出各种不同的存储方式，比如分层连续存储、树（Tree）、邻接表、图等，甚至可以把二者组合起来。从底层存储结构上来说，这些灵活的存储结构同样决定了元素的访问方式。就拿“树”来说，它在本质上就是一种特殊的链表结构，因此只能顺序访问，即从某个节点开始搜索直至到达所要访问的元素对象，或者采用深度优先、广度优先或者前序、中序、后序等方法遍历整棵树，但是不可能直接定位到树上的任一个结点对象。

在 STL 中，“树”被看做一种基本的存储方式，它比链表高效而又不失灵活性，实现起来要比“图”简单些，主要有二叉搜索树（`binary-search`）、平衡二叉树（`balanced binary search`）、红黑树（`red-black`）等。由于红黑树（平衡二叉搜索树的一种）在元素定位上的优异性能（ $O(\log_2 N)$ ），STL 通常使用它来实现关联式容器。

## 17.3.3 顺序容器和关联式容器的比较

在 STL 中，容器被划分为两类：顺序容器和关联式容器（联合容器）。这里的“顺序”和“关联”指的是上层接口表现出来的访问方式，并非底层存储方式。为什么这样划分呢？因为对 STL 的用户来说，他们并不需要知道容器的底层实现机制，只要知道如何通过上层接口访问容器元素就可以了，否则就违背了泛型容器设计的初

表。

STL 主要采用向量、链表、二叉树及它们的组合作为底层存储结构来实现容器。尽管一种容器可以采用多种存储方式来实现，但是出于效率的考虑，STL 必须为不同的容器类型选择最恰当的存储方式，而且不能改变容器的数学模型定义。顺序容器主要采用向量和链表及其组合作为基本存储结构，如堆栈和各种队列；而关联式容器采用平衡二叉搜索树作为底层存储结构。表 17-4 详细地列出了各种容器的存储方式和访问方式。

表 17-4 顺序容器和关联式容器的存储方式和访问方式

容器类名	可选存储结构	STL 采用的存储结构	底层访问方式	上层访问方式	是否排序
vector	—	—	随机/顺序	随机/顺序	否
list	—	—	顺序	顺序	否
queue	list/deque	deque	—	—	否
deque	vector/list	多层 vector	随机/顺序	随机/顺序	否
stack	vector/list/deque	deque	—	—	否
set	vector/list/tree	tree	顺序	随机（直接）/顺序	是
multiset					
map	vector/list/tree	tree	顺序	随机（直接）/顺序	是
multimap					

链表（list）作为顺序容器的典型代表，其特点就是“天然有序（Sequential）”，即各元素之间具有天然的相对位置关系，这就决定了理论上它可以存储任意多个元素，并且不需要对所有元素按值的大小进行排序。

注意：你要区分“有序”和“排序”两个概念。这对理解关联式容器的实现非常重要。

集合（set）是关联式容器的典型代表，需要从两个层次来理解：概念模型和实现方式。集合在概念上是一种无序的容器，这不仅指其中的各个元素之间没有相对位置关系（即不存在谁在前谁在后的概念），而且元素不需要按值的大小排序。但是如果要在计算机上实现这样的数学模型，不但要使上层访问接口正确反映集合的概念，还要考虑实现的效率。因此关联式容器在实现上必须是有序的和排序的，只不过要对用户透明（不让用户知道）。平衡二叉搜索树能够满足这些要求。

**【提示 17-1】：** 由于关联式容器在概念上是无序的，所以它只能通过元素的值来定位其中的元素对象，这就是关联式容器具有 find()函数的原因，也是调用这种容器的 insert()函数和 erase()函数时可以不指定插入位置和删除位置而仅指定元素值或者索引值的原因（关联式容器会自动地为新插入的元素安排一个合适的位置）。

**【提示 17-2】：** 由于顺序容器本来就有“序”，所以它是通过元素对象在容器中的位置来标识一个元素的，而不是通过元素的值（因为它可以存储值相等的多个

元素对象, 而且它们的位置不一定相邻)。这也就是调用顺序容器的 `insert()` 函数和 `erase()` 函数时必须指定插入位置和删除位置而不能仅指定元素值的原因。当然, 关联式容器也能存储值相等的元素, 比如 `multimap` 和 `multiset` 等, 但是它们在容器中的位置肯定是相邻的。

**【提示 17-3】:**

显然, 顺序容器的实现不会像关联式容器那样在增加和删除元素时对其元素进行自动排序, 它的概念决定了自动排序对查找 (定位) 其中任一元素的平均效率没有任何贡献 (否则反而会影响插入和删除的效率), 永远是  $O(N)$ , 因为你只能顺序访问它。这就是顺序容器没有提供 `find()` 函数的原因 (`string` 除外)。当然, 对一个具体的实现来说, 是否提供 `find()` 函数来查找第一个符合条件的元素, 以及是否提供 `sort()` 函数来对顺序容器进行排序, 那就是 STL 实现版本的问题了。另外, 这并不妨碍我们将 `sort()`、`find()` 和 `binary_search()` 等泛型算法应用于顺序容器。

映射 (`map`) 是一种独特的关联式容器, 它是一个二维的索引表, 其每一个元素是一个 <关键字, 值> 的二元组, 即 `pair<Key, Value>`。这就是说, `set` 是 `Key` 与自己关联, 而 `map` 则是 `Key` 与 `Value` 关联。

**【提示 17-4】:**

显然, `map` 应该提供通过 `Key` 值来定位 `Value` 对象的接口, 这就是 `map` 具有 `operator[]` 运算符函数的原因。

**【提示 17-5】:**

关联式容器的 `find()` 和 `operator[]` 函数反映了它们在应用层面的“随机访问”方式。但从底层实现上看, 它们采用的二叉树存储结构决定了访问方式仍然是顺序访问。真正的随机访问是时间恒定的, 例如 `vector` 和 `deque` 的 `operator[]`, 而关联式容器的这种“模拟随机”访问的效率则是  $O(\log_2 N)$ , 随着  $N$  的增大而按对数级增大。在创建关联式容器对象的时候, 可以指定二叉搜索树结点的排序方式 (递增、递减)。这就是关联式容器具有“谓词”而顺序容器没有“谓词”的原因 (优先级队列除外)。

**【提示 17-6】:**

如果元素查找是经常做的操作, 插入和删除反而是很少做的操作, 那么就不适合使用顺序容器, 而应该使用关联式容器; 反之就是用顺序容器。

然而, 对于某一种特定的容器来说, 由于不同的 STL 实现可能采用不同的底层存储方式, 因此从实现层面讲其访问方式可能不同, 但是从应用层面讲其访问方式是固定不变的。比如 `map`, 理论上你可以使用 `vector`、`list`、`tree` 中的任何一种来实现它, 因此在实现上有可能采用随机访问 (`vector` 的直接地址计算), 也可能采用顺序访问, 但是在应用层面或者说逻辑上应该总是“随机”访问; `set` 也是类似的道理。图 17-5 是 STL `std::set` 的一个简单的底层存储示意图。

知道了关联式容器在实现上必须维持元素的有序性 (`sorted`) 这一点, 也就不能再企图修改已经插入到其中的元素对象, 否则就会破坏这种有序性, 从而影响性能和正确性。为此, `set` 把 `iterator` 强制声明为 `const_iterator` 来达到修改的目的, `map` 则把 `pair` 中的 `Key` 强制声明为 `const Key`, `map` 仅允许修改 `pair` 中的 `Value`。这也就是说, 你无法直接为关联式容器的元素对象赋予新值或者调用它们的 `non-const` 方法。



相反，顺序容器则允许修改元素对象的值，或者直接赋值，或者调用 non-const 方法。参见 STL 源码。

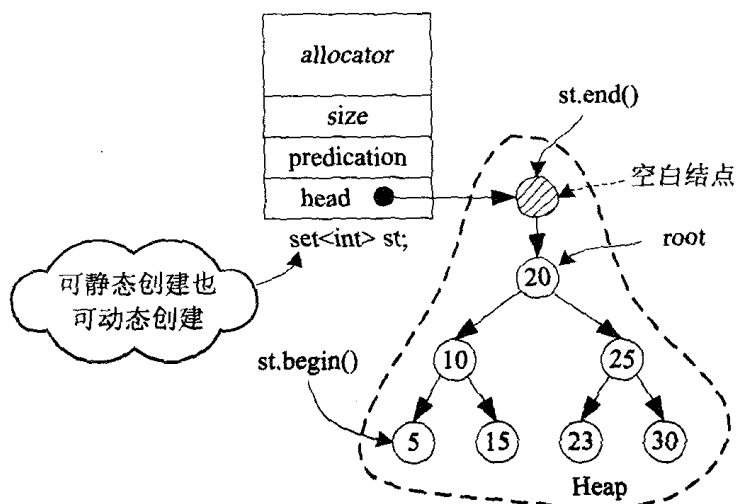


图 17-5 set 容器对象及其元素的内存映像

**【提示 17-7】:** Hash table 也是很重要的数据结构，但是目前的 STL 没有定义它。下一次修订 STL 的时候可能会考虑将 hash table 及由它演化出来的 hash\_set/hash\_map/hash\_multiset/hash\_multimap 作为关联式容器加进来。

### 17.3.4 如何遍历容器

你是否已经从图 17-2 和图 17-4 看出了 STL 容器的有效元素的范围表示法了呢？其中的迭代器 last 要么指向最后一个有效元素的末尾，要么指向一个空白节点，反正不是指向最后一个有效元素。这就是 STL 容器范围表示的“前闭后开”区间法，即 [first, last)。遍历容器较好的方法见示例 17-1（把 Container 替换为具体的泛型容器类，theContainerObj 替换为该容器类的对象即可）。

示例 17-1

```
for (Container::iterator first = theContainerObj.begin(),
     last = theContainerObj.end();
     first != last;
     ++first)    // 或者使用反向迭代器 rbegin()和 rend()
{
    cout << first->... << endl;
    cout << (*first)... << endl;
    ...
}

template<typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{

```

```
while(first != last && *first != value) ++first;
return first;
}
```

观察联合容器的存储结构（图 17-5）可以发现，它与 list 有些共同之处，例如使用一个空白节点来表示 end()，而且 begin() 指向最左端节点，end() 指向开头的空白节点而不是指向最右端节点。这就是说，对联合容器的遍历采用的是二叉树的“中序遍历”方法。因此，如果对图 17-5 的 set 进行遍历的话，依次访问的节点为：5、10、15、20、23、25、30，即从小到大（默认的按升序排列）。

## 17.3.5 存储空间重分配问题

存储空间重分配问题起源于容器元素对象的动态创建和连续存储特性，因此只有连续存储的容器才可能需要运行时的存储空间重分配，典型的就是 vector，其他的连续存储容器也会部分地需要存储空间重分配，比如 deque。

对于一个已经存在的 vector，其元素被保存在地址连续的存储空间上。当向该 vector 中插入一个（一些）新元素时，必须保证新的容器仍然满足元素连续存储的条件，这就要求扩张原有容器的存储空间。但是，当前的 vector 容器可能并没有那么多空余容量。怎么办呢？唯一的办法就是给整个 vector 的所有元素（包括新元素）重新分配存储空间，并把所有元素拷贝到新的地址处（依次调用它们的拷贝构造函数），然后释放原来的存储空间，最后将原来的容器对象调整为指向新的内存位置。

另一种情况是：vector 当前持有的预留容量足以容纳待插入的新元素，此时虽然不需要进行存储空间重分配，但是如果你不是在 vector 的尾部进行插入操作的话，同样需要移动 vector 的部分元素和调整指针。

由此可见，无论是存储空间重分配还是元素移动都是代价比较昂贵的操作，而且每当你插入新的元素时，这个过程都可能会进行一遍，并且还存在着分配失败的危险。那么是不是我们就别再用 vector 了呢？不是的。这实际上是内存需求与效率之间的矛盾，该如何解决这个问题呢？

**【建议 17-1】：** 如果你在创建一个容器时能够预先估计出它可能存放的最大元素数目，那么你就可以给它预先分配足够数量的存储空间，从而可以避免频繁的存储空间重分配操作。某些 STL 容器带有这种功能的构造函数及 reserve() 方法。

**【建议 17-2】：** 尽量在容器的尾部执行插入操作，因为这里的插入操作效率最高。对于顺序容器来说，它们本身并不要求元素排序，因此你完全没有必要刻意地在开头或中间插入元素；对于关联式容器，它们在实现时就是有序的，即调用 insert() 的时候会自动进行重新排序（伴随着树的平衡、旋转等操作），因此你也没有必要刻意地在开头或中间进行插入操作。

鉴于存储空间重分配的巨大开销，vector 在删除一个元素的时候并不会释放其存储空间，而仅是调用一下它的析构函数，目的就是保留这块空间以避免将来插入新元素时可能进行的存储空间重分配。string 类也是如此。

对于采用随机存储（非连续存储）方式的容器（如 list 和关联式容器），显然不需要进行存储空间重分配，也不需要预留空间，因此在删除元素对象时也不需要保留其空间。

### 17.3.6 什么样的对象才能作为 STL 容器的元素

这是在你使用 STL 容器前必须要搞清楚的一个很重要的问题！

使用动态内存的一般原则是：谁申请的内存就应由谁来释放。这可以避免大量由内存使用导致的问题。该原则与指针调用和引用调用并不矛盾。

为此，STL 容器采用拷贝方式来接收待插入的元素对象——在插入的时候容器自动新建等量的元素对象，并用待插入对象依次初始化它们（调用拷贝构造函数）；在删除元素时，容器负责释放其内存资源（对采用随机存储策略的容器）或者仅仅调用元素的析构函数（对采用连续存储策略的容器）。

容器只负责其元素对象本身一级的存储分配和释放，而不负责元素对象包含的额外内存的管理问题，这需要用户自己负起这个责任，典型的就是用指针作为容器元素（参见示例 16-22 和图 16-2）。

对象类型一般需要符合下述要求，才能够作为 STL 容器的元素。

（1）可默认构造的。但在不是在任何情况下都需要满足这一条，比如关联式容器，对于顺序容器，除非在初始化的时候需要插入默认构造的若干个对象，或者调用容器的 `resize()`、`assign()`、`insert()` 等函数的某些版本，否则也不需要满足这一条。

（2）可拷贝构造的。

（3）可拷贝赋值的（但也不是在任何情况下都需要）。

这几条条对基本数据类型及不含指针成员和引用成员的类型都是适用的。

（4）或者，具有 `public` 的、采用拷贝的方式显式定义的拷贝构造函数、拷贝赋值函数和析构函数。这一条适用于含有指针成员或引用成员的对象，但模拟指针（例如迭代器）应该归入前面几条中。

#### 【提示 17-8】

很多人对基本数据类型如 `int`、`long`、`char` 等具有默认构造函数和拷贝构造函数的很不理解。这其实是标准 C++ 为了使之与 ADT/UDT 一致而对它们改造的结果，同时也是为了适应 STL 的需要，因为当初始创建一个非空容器对象的时候，默认调用元素类型的默认构造函数或拷贝构造函数来初始化其中的元素对象（可参考 STL 源码），如果基本数据类型没有这些函数，STL 容器就不能适用于基本数据类型了，程序员们会暴跳如雷的。

#### 【提示 17-9】

根据上述条件，显然引用不能作为 STL 容器的元素类型：第一，引用在创建时必须初始化为一个具体的对象，而 STL 容器不能满足这一要求；第二，引用没有构造函数和析构函数，更没有赋值语义。这就是说，STL 容器只支持对象语义，而不支持引用语义。例如，下面的定义都是错误的。

```
std::list<double&> ld(10);
```

或：

```
typedef Object& ObjectRef;  
std::set<ObjectRef> objSet;
```

要想让容器支持引用语义，就得把引用封装为对象，就像把指针封装为对象那样。

采用接管方式创建和释放不对称的指针对象都不适合作为 STL 容器的元素，包括普通指针和 `auto_ptr`，除非你十分清楚这样做的实际意义，并且不会有错误和危险出现。参见本书第 16.15 节的例子。

**【提示 17-10】** 对于关联式容器，元素类型的要求可能更苛刻一些。关联式容器在实现上默认按 “<” 对其元素进行排序，从而确定它们在二叉树上的位置次序，同时在定位元素对象时也要调用元素类型的 `operator<` 运算符。这就要求元素类型必须至少定义 “<” 运算符重载函数。

## 17.4 迭代器

### 17.4.1 迭代器的本质

程序员刚刚接触 STL 时，总是不理解为什么有迭代器 (Iterator) 这种东西。我们知道循环结构其实就是一种迭代操作，不同的是它有两种循环控制方式：标志控制和计数器控制 (参见本书第 4 章)。迭代器可以把这些标志控制的循环和计数器控制的循环统一为一种控制方法：迭代器控制，每一次迭代操作中对迭代器的修改就等价于修改标志或计数器。

在 STL 中，容器的迭代器被作为容器元素对象或者 I/O 流中的对象的位置指示器，因此可以把它理解为面向对象的指针——一种泛型指针或通用指针，它不依赖于元素的真实类型。该如何理解 “通用” 二字的含义呢？

**【提示 17-11】** 迭代器的 “通用” 是一种概念上的通用，所有的泛型容器和泛型算法都使用 “迭代器” 来指示元素对象，所有的迭代器都具有相同或相似的访问接口，但是每一种容器都有自己的迭代器类型，毕竟每一种容器的底层存储方式不尽相同，所以迭代器的实现方式就会不同。千万不要以为存在一种 “通用的迭代器” ——它可以应用于任何类型的容器。

简而言之，迭代器是为了降低容器和泛型算法之间的耦合性而设计的，泛型算法的参数不是容器，而是迭代器。迭代器的概念如图 17-6 所示。

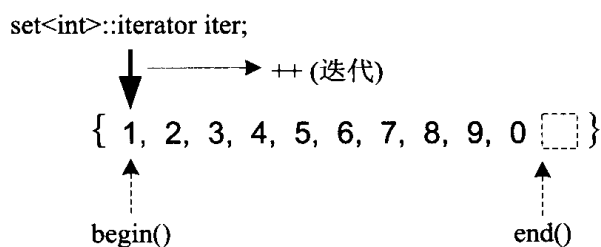


图 17-6 迭代器的工作逻辑

可见，容器迭代器的作用类似于数据库中的游标（cursor），它屏蔽了底层存储空间的不连续性，在上层使容器元素维持一种“逻辑连续”的假象。切不可把迭代器与 `void*` 和“基类指针”这样的通用指针混淆。

**【提示 17-12】：** 指针代表真正的内存地址，即对象在内存中的存储位置；而迭代器则代表元素在容器中的相对位置（当遍历容器的时候，关联式容器的元素也就具有了“相对位置”）。

STL 把迭代器划分为 5 个类别（Category），这 5 类迭代器分别具有不同的能力，表现为支持不同的运算符。它们都是类模板，因此具有通用性，见表 17-5。

表 17-5 标准迭代器

迭代器种类	提供的操作	特征说明
trivial 迭代器	<code>X x;</code> <code>X();</code> <code>*x;</code> <code>*x = t;</code> <code>x -&gt; m</code>	只是一个概念，用以区分所有迭代器的定义
输入迭代器 (Input Iterator)	<code>*i;</code> <code>(void)i++;</code> <code>++i;</code> <code>*i++</code> 还包含 trivial 迭代器中的所有操作	提供只读操作，即可读取它所指向的元素的值，但不可改变该元素的值； 如果 <code>i == j</code> ，并不意味着 <code>++i == ++j</code> ；
输出迭代器 (Output Iterator)	<code>X x;</code> <code>X();</code> <code>X(x);</code> <code>X y(x);</code> <code>X y = x;</code> <code>*x = t;</code> <code>++x;</code> <code>(void) x++;</code> <code>*x++ = t;</code>	只提供写操作，即可改变它所指向的元素的值，但不可读取该元素的值

(续表)

迭代器种类	提供的操作	特征说明
前进迭代器 (Forward Iterator)	<code>++i;</code> <code>i++;</code>	只能向前访问下一个元素, 不能反向访问前一个元素
双向迭代器 (Bidirectional Iterator)	<code>++i;</code> <code>i++;</code> <code>i--;</code> <code>--i;</code> 还包含前进迭代器中的所有操作	它是对前进迭代器的扩充, 提供双向访问
随机访问迭代器 (Random Access Iterator)	<code>i += n;</code> <code>i + n</code> 或 <code>n + i;</code> <code>i -= n;</code> <code>i - n;</code> <code>i - j;</code> <code>i[n];</code> <code>i[n] = t;</code> 还包含双向迭代器中的所有操作	不像前进迭代器或双向迭代器只能访问下一个或上一个元素, 它能访问前面或后面第 $n$ 个元素, 亦即随机访问任何一个元素

这 5 种迭代器的能力大小关系如下。

输入迭代器      输出迭代器      <      前进迭代器 <      双向迭代器 <      随机访问迭代器

因此, 这些迭代器之间必然存在下列关系:

- ◇ 随机访问迭代器是一种双向迭代器;
- ◇ 双向迭代器是一种前进迭代器;
- ◇ 前进迭代器是一种输入迭代器;

并且这种关系显然是可传递的。但是由于迭代器的类别是其相关类型之一, 因此切不可为它们之间存在 `public` 继承关系。比如原始指针可直接作为随机访问迭代器, 但它并非从其他迭代器派生而来。在 STL 中, 它们的关系通过下面的标记类层次反映出来 (摘自 P.J. Plauger 版<UTILITY>):

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag : public input_iterator_tag{};
struct bidirectional_iterator_tag : public forward_iterator_tag{};
struct random_access_iterator_tag : public bidirectional_iterator_tag{};
```

对于完全连续存储的容器, 例如 `vector`, 没有必要重新定义迭代器类型, 其元素的指针就可以直接充当迭代器, 这是很显然的。`vector<T>::iterator` 和 `const_iterator` 一般定义如下:

```
typedef T* iterator;
```

```
typedef const T* const_iterator;
```

而采用不连续存储或其他存储方式的容器，例如 `list`、`deque`、`set`、`map` 等，则需定义自己的迭代器类（`class`），一般情况下它们是对元素指针的封装，即模拟指针（参见 16.12 节或者 STL 源代码）。

但是要注意，一些特殊容器如 `vector<bool>` 和 `bitset<N>` 等，由于它们的存储单位为 `bit` 而不是 `Byte`，因此无法直接使用元素指针或其封装来定义它们的迭代器。如果想保持上层接口与普通容器一致，就必须做特殊处理。

对于一些常用的容器，你必须清楚地知道它们自己的迭代器所属的迭代器类别：

- ◇ `vector` 的迭代器为随机访问迭代器，因为它就是原始指针。
- ◇ `list` 的迭代器是双向迭代器，因为 `list` 是双向链表。
- ◇ `slist` 的迭代器是前进迭代器。
- ◇ `deque` 的迭代器是随机访问迭代器。
- ◇ `set/map` 的迭代器是双向迭代器。

.....

**【提示 17-13】：**为什么要对迭代器进行分类呢？主要是泛型算法可以根据不同类别的迭代器所具有的不同能力来实现不同性能的版本，使得能力大的迭代器用于这些算法时具有更高的效率。较典型的算法就是 `distance` 和 `advance`。这方面的知识涉及到 `traits` 技术，比较复杂，因此不再赘述。

**【提示 17-14】：**连续存储的容器，其元素的位置指示器有两种：下标和迭代器。下标的类型为 `unsigned int(size_t)`，有效范围为 `0 ~ size_t(-1)`，迭代器的有效范围则从 `begin()` 到 `end()`。这类容器的接口中都会提供相应的两种元素访问方法，典型的就 `vector` 和 `string`，它们都支持 `begin()`、`end()` 和 `operator[]` 等操作。

输入迭代器和输出迭代器是两种较特别的迭代器类别。由于每一种容器的迭代器都是针对自己的存储结构特别设计的，因此既可作为输入迭代器用，也可作为输出迭代器用。但是不能说它们的类别就是输入迭代器或输出迭代器。记住：输入/输出是相对而言的，是一种类别标记而已，一般在泛型算法中才会体现出这种方向性。真正专用于输入功能的迭代器是输入流迭代器，专用于输出功能的迭代器则是插入式迭代器和输出流迭代器。

关于迭代器的使用，我们给出下面几条建议。

**【建议 17-3】：**尽量使用迭代器类型，而不是显式地使用指针。例如使用 `vector<int>::iterator`，而不是 `int *`，虽然它们是等价的。

**【建议 17-4】：**只使用迭代器提供的标准操作，不要使用任何非标准操作，以避免 STL 版本更新的时候出现不兼容问题。

**【建议 17-5】：**当不会改动容器中元素值的时候，请使用 `const` 迭代器（`const_iterator`）。

## 17.4.2 迭代器失效及其危险性

迭代器失效是指当容器底层存储发生变动时，原来指向容器中某个或某些元素的迭代器由于元素的存储位置发生了改变而不再指向它们，从而成为无效的迭代器。使用无效的迭代器就像使用无效的指针（野指针）一样危险。

哪些操作可能引起容器存储的变动呢？主要有：`reserve()`、`resize()`、`push_back()`、`pop_back()`、`insert()`、`erase()`、`clear()`等容器方法和一些泛型算法，如 `sort()`、`copy()`、`replace()`、`remove()`、`unique()`，以及集合操作（并、交、差）算法等。见示例 17-2。

示例 17-2

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    vector<int> ages;                // 未预留空间
    ages.push_back(2);              // 引起内存重分配
    vector<int>::const_iterator p = ages.begin();
    for (int i = 0; i < 10; i++) {
        ages.push_back(5);          // 会引起若干次内存重分配操作
    }
    cout << "The first age : " << *p << endl; // p 已经失效，危险！
}
```

有两个办法可以解决迭代器失效问题：其一，在调用上述操作后重新获取迭代器；其二，在修改容器前为其预留足够的空闲空间以避免存储空间重分配（见示例 16-22）。示例 17-2 可改为示例 17-3。

示例 17-3

```
void main()
{
    //...
    vector<int>::const_iterator p = ages.begin();
    for (int i = 0; i < 10; i++) {
        ages.push_back(5);          // 会引起若干次内存重分配操作
    }
    p = ages.begin();              // 重新获取迭代器
    cout << "The first age : " << *p << endl; // ok
}
```

**【提示 17-15】** 顺序容器 `vector` 和 `string` 都可用 `reserve()` 和 `resize()` 来预留空间或调整它们的大小：`reserve()` 用来保留（扩充）容量，它并不改变容器的有效元素个数；`resize()` 则调整容器大小（size，有效元素的个数），而且有时候也会增大容器的容量。当把这两个函数与 `assign()`、`insert()`、`push_back()`、`replace()`



及泛型算法搭配起来使用的时候，需要小心从事！

首先要搞清楚“容量”和“容器”及“有效元素”的概念。

容量是为了减少那些使用连续空间（线性空间）存储元素的容器在增加元素时重新分配内存的次数的一种机制，即当增加元素且剩余空闲空间不足时，按照一定比例（通常是原来容量的 2 或 1.5 倍）多分配出一些空闲空间以备将来再增加元素时使用，以提高插入操作的性能。一个具有多余容量的 `std::vector<T>` 的典型内存映像如图 17-7 所示。

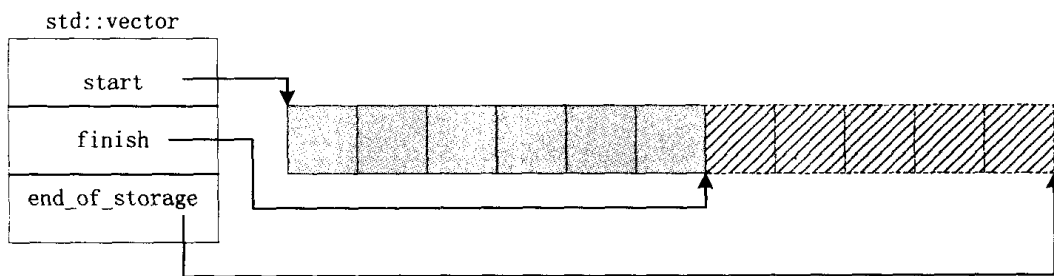


图 17-7 vector 容器和容量示意图

图中迭代器 `start` 和 `finish` 之间的元素就是容器的有效元素（实际上是有效元素对象本身的内存单元），而 `start` 和 `end_of_storage` 之间的空间就是该容器的总容量，容量是包含有效元素空间在内的。`finish` 和 `end_of_storage` 之间的空闲空间就是冗余容量，冗余容量不属于容器。

多余出来的容量（空闲存储空间）是未经初始化的（注意：并不是调用元素类型的默认构造函数来初始化）。我们可以从 `std::vector<T>` 的 `size()` 和 `capacity()` 这两个成员函数的实现上看出容器所辖元素空间和容量的区别：

```
size_type capacity() const { return (start == 0 ? 0 : end_of_storage - start); }  
size_type size() const { return (start == 0 ? 0 : finish - start); }
```

一个容器可以没有任何有效元素，但是却有许多冗余的容量；或者一个容器可以没有任何冗余的容量，全是有效元素；又或者既有一些有效元素，又有一些冗余容量。

那我们先来看一下 `reserve()` 到底做了一些什么吧！`reserve()` 的原型如下：

```
void reserve(size_type n);
```

其中 `n` 就是用户请求保留的总容量的大小（在不重新分配内存的情况下可容纳元素的个数）。`reserve()` 可按如下实现：

(1) 如果 `n` 大于容器现有的容量（即 `capacity()`），则需要在自由内存区为整个容器重新分配一块新的更大的连续空间，其大小为 `n * sizeof(T)`，然后将容器内所有有效元素从旧位置全部拷贝到新位置（调用拷贝构造函数），最后释放旧位置的所有存储空间并调整容器对象的元素位置指示器（就是让那三个指针指向新内存区的相应位置）。也就是说，如果请求容量比原有容量大的话，结果是容器的冗余容量加大（即 `end_of_storage` 指针的相对位置发生了改变），而容器本身的有效元素不会发生

任何变化，即容器的大小并没有改变。

(2) 否则，什么也不做，如图 17-8 所示。

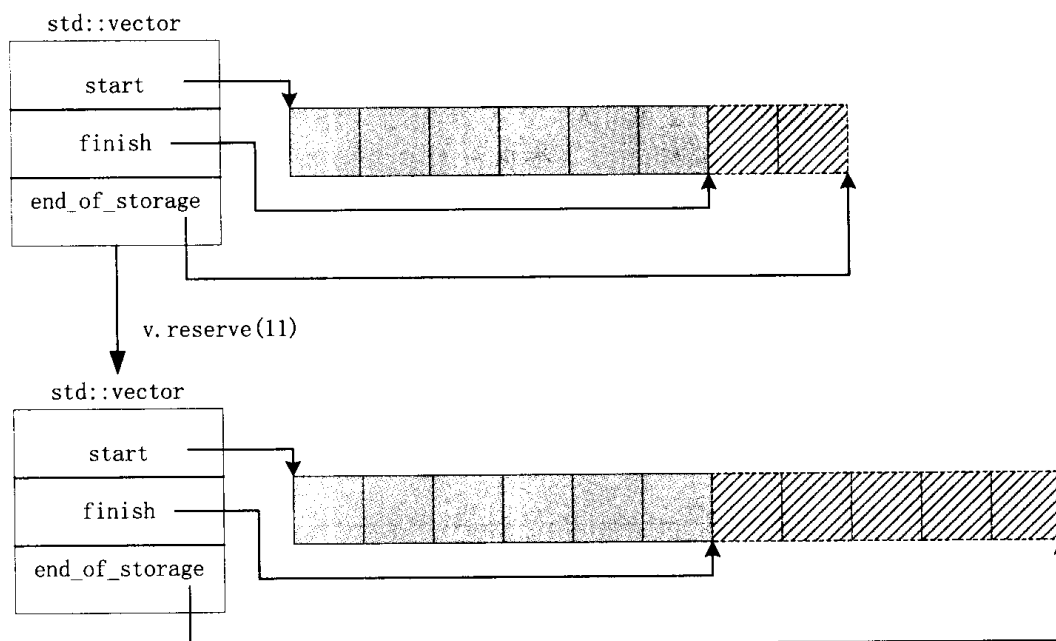


图 17-8 vector::reserve()功能示意

我们可以将容器看做一个区间，实际上泛型算法就是这样看待容器的。

我们知道，除了调用容器的某些方法可以改变容器的大小外，在容器外部没有任何方法可以做到这一点。因此如果想使用迭代器在冗余容量的空间上通过赋值来给容器增加元素的话，那结果一定会令你大失所望！如果不相信，就来看一个赋值的例子吧（见示例 17-4）。

示例 17-4

```
std::list<int> li;
std::vector<int> vi;
for (int c = 0; c < 10; c++)
    li.push_back(c);
vi.reserve(li.size()); // 预留空间，但是并没有改变容器的大小，预留空间未初始化
std::copy(li.begin(), li.end(), vi.begin()); // 拷贝赋值
std::copy(vi.begin(), vi.end(), std::ostream_iterator<int>(std::cerr, "\t"));
```

这段程序显然是错误的。虽然 `vi.reserve()` 为 `vector` 预留了内存，但是改变的只是容器的容量。同时在 `copy()` 算法中对容器元素赋值也不会改变容器的大小（毕竟算法不知道容器的存在，它只知道区间），因此拷贝过后容器的 `size()` 仍然为 0，虽然 `list` 的元素已经被拷贝到了为 `vector` 预留的空间上。结果可想而知：没有输出任何东西！`vector` 在拷贝前后的状态变化可以用图 17-9 来说明。

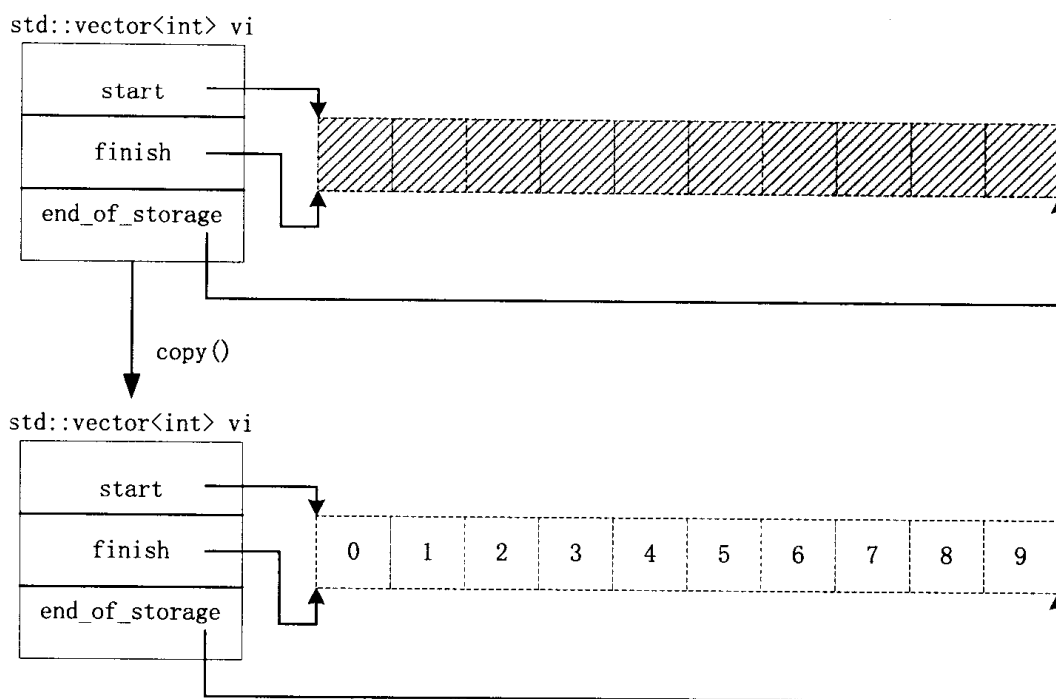


图 17-9 vector::reserve()和 std::copy()算法搭配

因此 reserve() 不能简单地与赋值运算配合, 比如不应该与简单的 copy() 算法合作, 而应该和 push\_back()、insert() 等操作合作, 或者与使用 insert 迭代器的 copy() 算法合作。例如:

```
vi.reserve(li.size()); // 预留空间, 但是并没有改变容器的大小, 预留空间并不初始化
std::copy(li.begin(), li.end(), std::back_inserter(vi));
```

这样的话, copy() 算法通过 insert 迭代器不仅给容器增加了元素, 而且也间接地改变了容器的大小, 结果当然如我们所愿了!

而 resize() 顾名思义, 它调整容器的大小 (size), 有时也会扩大容器的容量。换句话说, 不管容器当前包含多少个有效元素, 也不管容器的冗余容量是多少, 它都将容器的有效元素个数调整为用户指定的个数。resize() 的原型如下:

```
void resize(size_type n, const T& c = T());
```

其中 n 就是最后要保持的元素个数, 如果需要新增元素的话, c 则是新增元素的默认初始值。下面是 resize() 的实现策略。

(1) 如果 n 大于容器当前的大小 (即 size()), 则在容器的末尾插入 (追加) n - size() 个初值为 c 的元素, 如果不指定初值, 则用元素类型的默认构造函数来初始化每一个新元素 (这可能引起内存重分配以及容器容量的扩张)。

(2) 如果 n 小于容器当前的大小, 则从容器的末尾删除 size() - n 个元素, 但不释放元素本身的内存空间, 因此容量不变。

(3) 否则, 什么也不做。

因此, resize() 和赋值操作及 insert()、push\_back() 等都可以合作。例如上面的例子可以按如下编写, 见示例 17-5。

示例 17-5

```
std::list<int> li;
std::vector<int> vi;
for (int c = 0; c < 10; c++)
    li.push_back(c);
vi.resize(li.size());           // 调整容器大小
std::copy(li.begin(), li.end(), vi.begin()); // 拷贝赋值
std::copy(vi.begin(), vi.end(), std::ostream_iterator<int>(std::cerr, "\t"));
```

这一次，就可以输出正确的结果了，如图 17-10 所示。

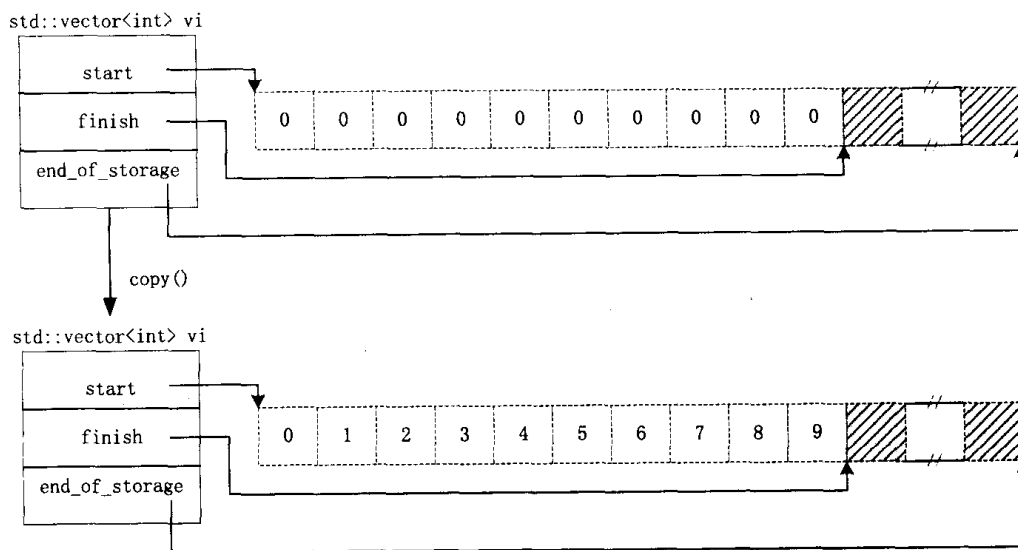


图 17-10 vector::resize()和 std::copy()算法搭配

相反地，使用 resize()缩减容器大小也是类似的。

显然，使用 reserve()和 resize()都不能缩减容器的容量。那么如何才能压缩容器的多余容量从而节省存储空间呢？一个解决办法就是使用容器的拷贝构造函数和 swap()函数，因为拷贝构造函数可以根据已有容器的大小决定一次性分配多少元素空间，就不会产生冗余容量，如示例 17-6 所示。读者不妨思考一下，看是否还有其他更好的解决方法。

示例 17-6

```
std::vector<int> vi;
for (int c = 0; c < 10; c++)
    vi.push_back(c);
std::vector<int>(vi).swap(vi); // 构造一个临时对象，然后与之交换元素
```

明白了容器与容量的关系后，我们来实现一个固定容量的循环队列，代码如下（见示例 17-7）。

示例 17-7

```
template<typename T /*元素类型*/, unsigned int N /*容量*/>
class CyclicQueue {
public:
    typedef T                value_type;
    typedef size_t           size_type;
    typedef T&               reference;
    typedef const T&         const_reference;

    CyclicQueue() : m_popPos(0), m_count(0) {
        assert(N > 0);
        m_beginPtr = (T*)(::operator new(sizeof(T) * N)); // 分配原始空间
    }
    ~CyclicQueue() {
        _Clear(); // this->_Clear();
        ::operator delete((void*)m_beginPtr);
    }
    CyclicQueue(const CyclicQueue<T, N>& copy) : m_popPos(0), m_count(0) {
        assert(N > 0);
        m_beginPtr = (T*)(operator new(sizeof(T) * N)); // 分配原始空间
        size_t copyPos = copy.m_popPos;
        for (size_type idx = 0; idx < copy.m_count; ++idx) {
            _Copy(idx, copy.m_beginPtr[copyPos]); // this->_Copy();
            ++copyPos; copyPos %= N; ++m_count;
        }
    }
    CyclicQueue& operator=(const CyclicQueue<T, N>& other) {
        CyclicQueue<T, N> temp(other); // 调用拷贝构造函数
        swap(temp); // this->swap();
        return (*this);
    }
    bool is_empty() const { return (m_count == 0); }
    bool is_full() const { return (m_count == N); }
    value_type front() {
        assert(m_count != 0);
        return (m_beginPtr[m_popPos]);
    }
    value_type front() const {
        assert(m_count != 0);
        return (m_beginPtr[m_popPos]);
    }
    value_type back() {
        assert(m_count != 0);
```

```

        size_type pushPos = (m_popPos + m_count) % N;
        if (pushPos == 0)
            return (*(m_beginPtr + N - 1));
        return (m_beginPtr[pushPos - 1]);
    }

    value_type back() const {
        assert(m_count != 0);
        size_type pushPos = (m_popPos + m_count) % N;
        if (pushPos == 0)
            return (*(m_beginPtr + N - 1));
        return (m_beginPtr[pushPos - 1]);
    }

    bool push(const_reference data = T()) {
        if (m_count < N) {
            size_type pushPos = (m_popPos + m_count) % N;
            _Copy(pushPos, data);
            ++m_count;
            return true;
        }
        return false;
    }

    bool pop(reference data) {
        if (m_count > 0) {
            data = m_beginPtr[m_popPos];
            _Destroy(m_popPos);
            --m_count; ++m_popPos; m_popPos %= N;
            return true;
        }
        return false;
    }

    size_type size() const { return m_count; }
    size_type capacity() const { return N; }
    void clear() { _Clear(); }
    void swap(CyclicQueue<T, N>& other) {
        std::swap(m_beginPtr, other.m_beginPtr);
        std::swap(m_popPos, other.m_popPos);
        std::swap(m_count, other.m_count);
    }

private:
    void _Clear() {
        for (; m_count > 0; --m_count) {
            _Destroy(m_popPos);
            ++m_popPos; m_popPos %= N;
        }
    }

```

// 不满!

// this->\_Copy();

// 不空!

// operator=

// this->\_Destroy();

// 新的 pop 位置

// this->\_Clear();

// this->\_Destroy();

```

        m_popPos = 0;
    }
    void _Destroy(size_type idx) {
        assert(idx < N);
        T *pTemp = (m_beginPtr + idx);
        pTemp->~T(); // 调用析构函数销毁元素对象
    }
    void _Copy(size_type idx, const_reference data) {
        assert(idx < N);
        T *pTemp = (m_beginPtr + idx);
        new ((void*)pTemp) T(data); // 调用 placement new 和拷贝构造函数拷贝对象
    }

    value_type      *m_beginPtr; // 队列存储空间起始位置
    size_type        m_popPos;    // 下次 pop 位置
    size_type        m_count;     // 有效元素个数
};

```

读者可自己动手画出这个循环队列的内存映像，并与示例 12-3 的循环缓冲区模型对照。

从循环队列的实现可以看出：对于冗余容量，不能用元素的默认构造函数来初始化（从而也就不需要元素类型提供默认构造函数）；对于 push 操作，实际上是使用拷贝构造函数来初始化新的元素对象（而不是调用拷贝赋值函数，因为新的 push 位置上还不是一个有效的元素对象）；而对于 pop 操作，则必须显式地调用元素的析构函数来销毁它（从而使它的地盘变为空白）。读者可以参照 `std::vector` 的实现来印证这一点。

读者可试着为示例 12-3 的 `RingBuffer` 类编写拷贝构造函数和赋值函数。

#### 【提示 17-16】:

尽量不要在遍历容器的过程中对容器进行插入元素、删除元素等修改操作，这和不要在 for 循环中修改计数器是一个道理，特别是连续存储的容器中。因为这些操作会使一些迭代器失效，特别是当前迭代器，这在效果上等价于修改了循环计数器。更进一步的原因是下一次迭代操作，即 `++iterator` 会使用本次迭代操作的迭代器，而当前迭代器可能已经失效。虽然有些容器如 `list`，修改操作只会使当前迭代器失效，即并不会引起存储空间重分配，所以可以在遍历的过程中正确地删除当前元素（这里面有一个技巧），但是也最好不要这样做，否则可能存在重大隐患。参见 `list` 等的 `remove()`、`remove_if()` 成员函数的实现。

#### 【提示 17-17】:

修改容器和修改容器中的元素对象的值是两码事儿。前者有可能引起容器底层存储的变动，因此可能使迭代器失效，而后者则不会。顺序容器允许直接修改其中元素对象的值；而关联式容器则不允许修改其元素的 Key 值，如 `set` 的元素对象及 `map` 的 Key 都不能直接修改，因为它们使用 Key 值来排序。

## 17.5 存储分配器

STL 容器元素的存储空间是动态分配和释放的，不同的硬件平台和操作系统对内存的管理方法和使用方式各不相同，这些问题对一个内存对象的创建和销毁应该是完全透明的。但如果把内存的分配和释放等操作交给用户来完成的话，用户又会回到直接和内存及指针打交道的悲惨境地。更有甚者，对象不一定是保存在内存中的，某些应用可能要保存在数据库中或磁盘文件中，显然对象持久化和重构等底层细节对容器的使用都应该是透明的。为此，STL 为容器类定义了一个专门负责存储管理的类——`allocator`，但它仅针对内存管理。

`allocator` 类是一个模板，作为容器类模板的一个 `policy` 参数，它不仅与将要为之分配空间的数据对象的类型无关，并且为动态内存的分配和释放提供了面向对象的接口。它是对 `new` 运算符的更高层次的抽象，即隐藏了底层的内存模式（段内存、共享内存、分布式内存等），封装了动态内存分配和释放操作，隐藏了指针本身的大小、存储空间重分配模型及内存页大小等细节，提供了更好的可移植性。因此，`allocator` 类为 `string` 类、容器类及用户自定义类型、应用程序框架和类库等提供了独立于硬件和操作系统的接口。

存储分配器是可以替换的，如果实现了一个更好的存储分配器或者应用于特殊环境的存储分配器，那么就可以用它来替换 STL 容器的默认存储分配器，但是必须遵守 STL 的组件定义规则。关于 `allocator` 类的详细信息请参考 STL 头文件 `<memory>` 或相关资料。这里我们给出一个应用于 COM 环境下 STL 容器的 `allocator` 实现（见示例 17-8）。

示例 17-8

```
template<typename _Ty>
class STLCOMAllocator {
public:
    typedef size_t          size_type;
    typedef ptrdiff_t       difference_type;
    typedef _Ty             value_type;
    typedef _Ty*            pointer;
    typedef const _Ty*       const_pointer;
    typedef _Ty&             reference;
    typedef const _Ty&       const_reference;

    pointer address(reference ref) const { return (&ref); }
    const_pointer address(const_reference ref) const { return (&ref); }
    _Ty* allocate(size_type n, const void* /* no use */)
    { return (pointer)(::CoTaskMemAlloc(n * sizeof(_Ty))); } // aligned!
    void deallocate(void *p, size_type /* no use */)
    { ::CoTaskMemFree(p); }
```



```

void construct(pointer p, const _Ty& v) {
    if (p != NULL) new ((void*)p) _Ty(v);    // placement new & copy constructor
}
void destroy(pointer p) {
    if (p != NULL) p->~_Ty();                // destructor
}
size_type max_size() const {
    size_type sz = (size_type)(-1) / sizeof(_Ty);
    return (0 < sz ? sz : 1);
}
};
template<typename _Ty, typename _U> inline
bool __stdcall operator ==(const STLCOMAllocator<_Ty>&,
    const STLCOMAllocator<_U>&)
{ return (true); }
template<typename _Ty, typename _U> inline
bool __stdcall operator !=(const STLCOMAllocator<_Ty>&,
    const STLCOMAllocator<_U>&)
{ return (false); }

```

## 17.6 适配器

基本的容器就那么几种，而可用的数学模型却有很多种，如何来实现它们呢？

我们可以在基本容器的基础上通过改变它们的接口来实现特殊的容器，这就是容器适配器 (Container Adapter) 的概念 (有时候俗称“二次封装”，专业一点是“用……来实现……”，实际上就是一种扩展手段)。

适配器 (Adapter) 往往是利用一种已有的比较通用的数据结构 (通过组合而非继承) 来实现更加具体的、更加贴近实际应用的数据结构。因此容器适配器窄化了 (narrow) 或者说强化了基础容器的接口，而基础容器的接口是比较通用的。例如 stack 就是比较特殊的容器，它可使用 deque 或 list 等来实现，但是去掉了那些不符合 stack 特点的操作，stack 的接口受到了比 deque 和 list 的接口更强的约束。此外，stack 使用 deque 或 list 的 back()、push\_back() 和 pop\_back() 操作分别实现其 top()、push() 和 pop() 操作，见示例 17-9。

示例 17-9

```

template<typename T, typename S = deque<T>>
class stack {
    ...
    T& top(){ return (c.back()); }
    const T& top() const{ return (c.back()); }
    void push(const value_type& v){ c.push_back(v); }
    void pop(){ c.pop_back(); }
}

```

```
...
private:
    S c;
};
```

**【提示 17-18】** “窄化 (narrow)” 这个词常用在类型转换 (cast) 中，专指类型安全的向下转型。这里的意思是去掉基础结构中不适用于适配器的接口，也就是适配器在概念上不应该支持的接口，而保留或增加适配器特有的接口和属性。

其他的容器适配器有 `queue` 和 `priority_queue`，使用的方法都是类似的。

除了容器适配器外，STL 还有迭代器适配器 (Iterator Adapter) 和函数对象适配器 (Functor Adapter)，并且提供了大量的足以满足普通应用的适配器组件。

STL 定义的迭代器适配器并不都是像容器适配器那样去改变另一个迭代器的接口，而是具有特殊的设计和特殊的用途。

顾名思义，插入式迭代器 (Insert Iterator) 以一个容器为操作对象，并向其中插入元素来完成批量输出功能，包括 `back_insert_iterator`、`front_insert_iterator` 和 `insert_iterator`。对一个 `back_insert_iterator` 执行赋值操作 (`operator=`) 就相当于对其绑定的容器执行 `push_back()` 操作，对一个 `front_insert_iterator` 执行赋值操作就相当于对其绑定的容器执行 `push_front()` 操作，而一个 `insert_iterator` 执行赋值操作则相当于对其绑定的容器执行 `insert()` 操作。同时对于输出迭代器，反引用 (`*`)、前进 (`++`)、后退 (`--`)、取元素地址 (`->`) 等操作都是无意义的，但是为了与普通迭代器具有一致的行为模式，还是保留了反引用和前进的能力，并关闭它们的功能，仅简单地返回自己。我们列出 `back_insert_iterator` 的源代码 (见示例 17-10)，其他两个可参考头文件 `<ITERATOR>`。

示例 17-10

```
template<class C>          /* 摘自 P.J. Plauger 版<ITERATOR> */
class back_insert_iterator : public iterator<output_iterator_tag, void, void> {
public:
    typedef C container_type;
    typedef C::value_type value_type;
    explicit back_insert_iterator(C& x) : container(x) {}    // 指定一个容器对象
    back_insert_iterator<C>& operator=(const value_type& val) {
        container.push_back(val);    // 以 push_back()取代赋值
        return *this;
    }
    // 关闭下列运算符的功能，仅简单返回自己
    back_insert_iterator<C>& operator*() {
        return (*this);    // 使得调用语句*iter = X;能够调用到当前 iterator
                           // 的 operator=运算符，而该运算符则调用容器的
                           // push 或 insert 操作
    }
};
```

```

        back_insert_iterator<C>& operator++(){ return (*this); }
        back_insert_iterator<C>& operator++(int){ return (*this); }
protected:
        C& container;      // 内部维护对容器的引用
};

```

插入式迭代器的用法可参考示例 17-12 和示例 17-14。

输出流迭代器（`ostream_iterator`）则通过绑定一个 `ostream` 对象来完成批量输出功能，即内部维护一个 `ostream` 对象，并将赋值操作（`operator=`）转换为对该 `ostream` 对象的运算符 `operator<<` 的调用。同样，它关闭了反引用和前进功能，并且禁止后退和取地址操作。源代码请参考头文件 `<ITERATOR>`。无论是把容器的内容全部输出到磁盘文件还是终端，都可以使用该迭代器。用法见示例 17-11。

示例 17-11

```

#include <list>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
void main()
{
    list<int> li;
    for (int k = 0; k < 10; k++) {
        li.push_back(k);
    }
    copy(li.begin(), li.end(), ostream_iterator<int>(cout, " "));
}

```

类似地，输入流迭代器（`istream_iterator`）绑定了一个 `istream` 对象来完成批量输入功能，并将前进操作（`++`）（注意：不是 `operator=`）转换为对 `istream` 对象的运算符 `operator>>` 的调用。该迭代器既可用于终端输入也可用于磁盘文件输入。其源代码不再列出，请参考 `<ITERATOR>`。其用法见示例 17-12。

示例 17-12

```

#include <list>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
void main()
{
    list<int> li;
    istream_iterator<int> eos, isiter(cin);
    copy(isiter, eos, back_inserter(li));
}

```

```
        copy(li.begin(), li.end(), ostream_iterator<int>(cout, " "));  
    }
```

反向迭代器 (Reverse Iterator) 用于将一个指定的迭代器的迭代行为反转 (前进变后退, 后退变前进), 可用于修饰除前进迭代器外的其他类型迭代器。容器具有的 `rbegin()` 和 `rend()` 方法返回的就是这种类型的迭代器。

函数对象及其适配器又是另一番情景, 而且比较复杂, 有兴趣的读者请阅读其他书籍, 或者参考头文件 `<FUNCTIONAL>`。

在了解了各种适配器的设计原理后, 就可以开发满足自己特殊需求的适配器, 只需遵守 STL 的游戏规则即可。

## 17.7 泛型算法

STL 定义了一套丰富的泛型算法, 可施行于容器或其他序列上, 它们不依赖具体容器的类型和元素的数据类型。标准 C++ 算法的通用性是通过模板、迭代器、运算符重载及函数对象来实现的, 迭代器就像算法和容器的中间人 (见图 17-1), 而重载的运算符就是算法的武器, 函数对象则用来定制 (改变) 算法默认的行为。算法从迭代器得到一个数据对象, 而迭代器则指示数据对象在容器中的位置。迭代器对象用来查找容器中下一个元素对象的位置并把它告诉算法, 算法就能逐个地访问容器中所有元素对象。

作为算法, 它不必关心所操作的数据对象在容器中的什么位置, 也不必知道容器的类型甚至数据对象的类型, 它所要做的工作就是 “改变迭代器, 并按照用户指定的方式 (即函数对象或谓词, 可以没有), 逐个地对迭代器指向的对象进行定制的操作”。

STL 提供的泛型算法主要有如下几种:

- ◇ 查找算法, 如 `find()`、`search()`、`binary_search()`、`find_if()` 等。
- ◇ 排序算法, 如 `sort()`、`merge()` 等。
- ◇ 数学计算, 如 `accumulate()`、`inner_product()`、`partial_sum()` 等。
- ◇ 集合运算, 如 `set_union()`、`set_intersection()`、`includes()` 等。
- ◇ 容器管理, 如 `copy()`、`replace()`、`transform()`、`remove()`、`for_each()` 等。
- ◇ 统计运算, 如 `max()`、`min()`、`count()`、`max_element()` 等。
- ◇ 堆管理, 如 `make_heap()`、`push_heap()`、`pop_heap()`、`sort_heap()`。
- ◇ 比较运算, 如 `equal()` 等。

这些泛型算法都是定义于名字空间 `std::` 内的函数模板, 虽然某些功能在具体的容器里可能有对应的实现 (作为成员函数), 例如 `find()` 和 `remove()` 等, 但是泛型算法是通用的, 不依赖于具体的容器。它们有些作用于一个容器 (如 `min_element`), 而有些作用于两个容器 (如 `find_first_of`), 有的甚至同时作用于 3 个容器 (如 `merge`)。

值得一说的是算法的参数。泛型算法一般接受下列参数类型的一种或几种:

- ◇ 迭代器，标示容器或区间的范围，以值传递。
- ◇ 谓词，返回 bool 值的函数对象，指定算法的操作方式，例如 find\_if() 的第三个参数。
- ◇ 函数对象，用户指定要做的操作，例如 for\_each() 的第三个参数。
- ◇ 容器元素，用户指定的基准对象，例如 find() 的第三个参数。

**【提示 17-19】:** 注意，泛型算法并不接受容器对象作为参数，因此它不了解与传入的迭代器关联容器的具体情况，当然它也不能对容器做任何操作（即无法调用容器的方法），除非使用迭代器适配器。因此，你必须保证传入的每一对迭代器是有效的，并且从 first 开始经过有限步迭代可以到达 last，即指定的区间是可达的，否则结果不可预料。

算法 copy() 的定义见示例 17-13。

示例 17-13

---

```
Template<typename InputIterator, typename OutputIterator> inline
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator oi)
{
    for(; first != last; ++first, ++oi) // 调用 InputIterator::operator!=
        *oi = *first;                // 调用 OutputIterator::operator=
    return oi;
}
```

---

给算法传递迭代器对象时没有必要使用引用传递，因为迭代器一般就是指针或是模拟指针，本身开销很小。

**【提示 17-20】:** 在使用带有输出迭代器参数的泛型算法（如 merge()、copy() 等）时，一定要给输出容器（目标容器）分配足够的空闲空间，否则操作结果不可预料。简单的迭代器只是标记了容器中元素的位置，而不具备在算法执行过程中对目标容器进行动态扩充的能力。另一种方法是使用插入迭代器（insert\_iterator、back\_insert\_iterator、front\_insert\_iterator）作为输出迭代器（见示例 17-14）。

示例 17-14

---

```
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
void main()
{
    list<int> li; vector<int> vi;
    for (int c = 0; c < 10; c++) li.push_back(c);
    vi.resize(li.size());
}
```

---

```
// 拷贝链表元素到 vi 中，目标容器开始位置为 vi.begin()
copy(li.begin(), li.end(), vi.begin()); // 调用 int::operator=()
}
// 或者：
void main()
{
    list<int> li; vector<int> vi;
    for (int c = 0; c < 10; c++) li.push_back(c);
    // 拷贝链表元素到 vi 中，使用插入迭代器
    copy(li.begin(), li.end(), back_inserter(vi)); //调用 vi.push_back()
}
```

**【提示 17-21】:**

很多泛型算法总是假定容器的元素类型定义了 `operator=()`、`operator==()`、`operator!=()`、`operator<()` 或 `operator>()` 等函数，因此你有义务为你的容器元素类型定义它们，否则泛型算法将采用元素类型的默认语义或者报错。

**【建议 17-6】:**

在应用编程时要选用最合适的算法。有时候几个算法都可以解决某个问题，此时要找出效率最高的那个算法，而不是随便挑一个。例如，`find()` 算法的复杂度为  $O(n)$ ，而 `binary_search()` 算法的复杂度为  $O(\log_2 N)$ ，当容器中的元素有序时，当然应选用 `binary_search()`。

**【建议 17-7】:**

STL 提供了最常用的算法，尽管它们很有用，但不可能解决所有的应用问题。建议读者在编写自己的算法时，尽量做到与 STL 框架无缝结合，这样会提高算法的可扩展性。

示例 17-15 是基于 STL 框架实现的“折半”查找算法，供读者参考。

示例 17-15

```
template<typename RandomAccessIterator, typename T>
RandomAccessIterator binary_search(RandomAccessIterator first,
                                   RandomAccessIterator last,
                                   const T& value)
{
    RandomAccessIterator mid, not_found = last;
    while (first != last) {
        mid = first + (last - first) / 2;    // 注意：不是 (first + last) / 2
        if (!(value < *mid) && !(*mid < value))
            return mid;                    // 调用 T::operator<()
        if (value < *mid)
            last = mid;                    // 调用 T::operator<()
        else
            first = mid + 1;
    }
    return not_found;
}
```

**【提示 17-22】:**

泛型算法在声明和实现中需要满足最低级别（最低能力）的迭代器的需求，所以你给它传入比声明的迭代器级别高的迭代器也不成问题。如果当传入的迭代器类别确实影响到算法的效率时，泛型算法会针对不同级别的迭代器来实现不同的版本。算法内部对迭代器类别（Category）的识别是通过 `Iterator Traits` 技术从传入的迭代器对象中萃取出来的，所以从算法的接口上看不到它是如何标识迭代器类别的。示例 17-13 所示的 `copy()` 算法要求传入的迭代器类别至少是输入迭代器，因此你可以将它应用于前进迭代器、双向迭代器，甚至随机访问迭代器。

某些算法的行为可能无法和你预想的结果相符。这是因为作为泛型算法，它不了解其所作用容器的细节（存储结构、访问方法等），因此它只能对所有可应用它的容器做最普通的处理。最典型的就是 `remove()`、`unique()`、`merge()` 等算法。比如 `remove()` 和 `unique()`，它们并不是把指定的元素或重复的元素从指定的区间内删除（即 `erase`，它们也无法删除），而是简单地用位于后面的元素把它覆盖掉，结果就使容器后面的部分元素仍然留在那里（本来是应该删除的）。这样的算法显然不能完全满足某些容器的需要，比如 `list`。因此 `list` 特别为自己定义了一些与对应泛型算法功能相似的方法，但是行为和结果却不同，见示例 17-16。

示例 17-16

---

```
// 泛型算法 remove          摘自 P.J. Plauger 版<ALGORITHM>
template<class _FI, class _Ty> inline
_FI remove(_FI _F, _FI _L, const _Ty& _V)
{
    _F = find(_F, _L, _V);
    if (_F == _L) return (_F);
    else{ _FI _Fb = _F;
        return (remove_copy(++_F, _L, _Fb, _V)); }
}

template<class _II, class _OI, class _Ty> inline
_OI remove_copy(_II _F, _II _L, _OI _X, const _Ty& _V)
{
    for (; _F != _L; ++_F)
        if (!(*_F == _V)) *_X++ = *_F;    // 向前移动元素
    return _X;
}

// list<_Ty>::remove()      摘自 P.J. Plauger 版<LIST>
void remove(const _Ty& _V)
{
    iterator _L = end();
    for (iterator _F = begin(); _F != _L; ) {
        if (*_F == _V) erase(_F++);    // 直接删除
    }
}
```

```

        else ++_F;
    }
}

```

**【提示 17-23】** 那么当容器的方法和泛型算法都可以完成一项工作时，该选择谁呢？当然是选择容器本身的方法了，不仅是因为泛型算法的结果可能不尽如人意，还因为其效率较差。

## 17.8 一些特殊的容器

容器是一个很宽泛的概念，理论上讲只要具有对象聚集特点的数据结构都可以看成容器，而且并非必须是可动态变化的结构。

### 17.8.1 string 类

如果你切实理解了字符串和字符数组的关系的话，理解 `string` 类就不是难事。`std::string` 是类模板 `std::basic_string` 的一个元素类型为 `char` 的实例化，而 `basic_string` 则是对元素指针的封装。由于 `basic_string` 的实现对于字符串操作进行了优化，它几乎不能用来表示除 `char` 以外的对象串。但是使用 `string` 类避免了和 ‘\0’ 纠缠不休，也就是说，`string` 并不关心它所代表的字符串有无 ‘\0’ 结束标志，相反 ‘\0’ 也可以是它的一个合法元素。只要你不是企图通过其中的字符指针访问整个字符串，那么中间的 ‘\0’ 不会成为你使用 `string` 的障碍。

`std::string` 类的构造函数可以接受 C 字符指针，但是它并不负责检查该指针的有效性，你必须保证该指针指向一个合法的字符串而不是 `NULL`。这一点需要引起注意。

`std::string` 定义了两个成员函数 `c_str()` 和 `data()` 返回 C 风格的字符指针，但是没有定义可以向 C 字符串自动转换的 `operator`，即：

```
operator Element*() const { return _Ptr; }
```

原因就是 ‘\0’ 问题。如果提供了这样一个自动转型运算符的话，就意味着我们可以在标准 C 的字符串处理库函数上直接使用 `string` 对象了。例如：

```
strcpy(string(10), "Hello!");
```

C 字符串库函数关心 ‘\0’，但是 `string` 类并不关心，也许它的串中根本就没有 ‘\0’，于是当你使用自动转换返回的字符指针时就可能导致不可预料的后果。

`std::string` 是一个容器，因此它具有容器的一般特征，比如内存动态分配，支持动态插入、删除操作，存储空间重分配，迭代器失效等，因此它类似于一个 `vector<char>`。重要的是 `string` 类提供了和 C 字符串配合的方法，比如对 `+=`、`=` 的重载，头文件 `<string>` 中也提供了 `string` 对象和 C 字符串之间的各种比较运算符，比如 `==`、`+`、`<=`、`>=`、`<`、`>`、`!=` 的各种重载版本，方便了字符串的操作。

`std::string` 类保存了串的长度信息，可以直接返回，而不是像库函数 `strlen()` 那样



需要遍历整个串空间，因此提高了效率。

### 17.8.2 bitset 并非 set

`std::bitset` 也是一个比较重要而且特殊的类型，但是它并非 `std::set` 的特化，甚至它连容器都不是，而是一个以数据值为参数的类模板。从 `bitset` 的定义和接口来看，不如叫它 `bit array` 更合适。其一般定义如示例 17-17 所示。

示例 17-17

```
template<size_t Length>
class bitset {
public:
    class reference{ ... };    // 可以自动转换为 bool 变量
    // constructors...
    // accessors...
    // mutators...
    // convertors...
    // statistics...
    // test...
    // &=, |=, ^=, <<=, >>=, >>, <<, ==, !=, ~ overloading...
    // <<(output), >>(input), &, |, ^ friend overloading...
private:
    enum {
        WORD_BITS = sizeof(unsigned long) * 8, // WORD 的 bit 数
        N_WORD = ( Length == 0 ? 0 : (Length - 1) / WORD_BITS )
                // Length 个 bit 折合多少个 WORD
    };
    unsigned long array[N_WORD + 1];    // 内置数组（不能定义长度为 0 的数组）
};
```

`bitset` 常用来设计由许多标志位组合而成的变量，如掩码及其运算等。`bitset` 的底层存储结构如图 17-11 所示。

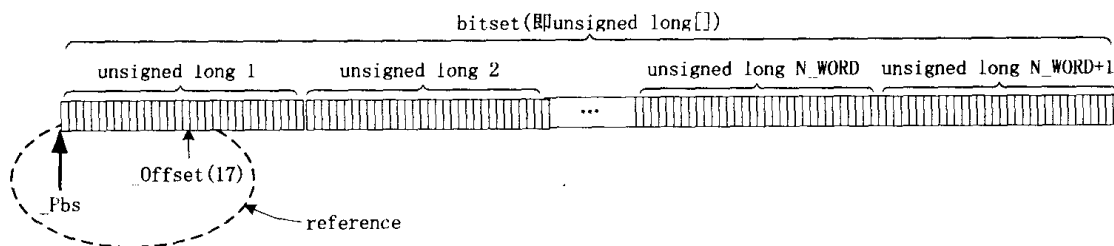


图 17-11 `bitset<Length>` 的存储结构

`bitset` 并非容器，因此长度一旦确定就不能改变，亦不支持插入、删除操作，没有迭代器，只有 `reference`。但是 `bitset` 支持随机访问，并且提供了丰富的位操作方法，无论是针对单个 `bit` 还是整个 `bitset` 的所有 `bit`。`bitset` 支持的运算见表 17-6。

表 17-6 bitset 支持的运算

方 法	解 释
bitset()	默认构造函数, 创建指定长度的 bitset, 所有 bit 初始化为 0
bitset(unsigned long n)	把一个 unsigned long 型数转换为 bitset (用一个 unsigned long 初始化 bitset 对象)
bitset(const string& s)	把由 0/1 序列组成的字符串转换为 bitset
at(size_t p)	访问偏移为 p 的 bit 元素的值 (const 版和 non-const 版)
operator[](size_t p)	访问偏移为 p 的 bit 元素的值 (const 版和 non-const 版)
operator&=	当前 bitset 对象与另一个 bitset 对象按位与, 结果为 this 对象
operator =	当前 bitset 对象与另一个 bitset 对象按位或, 结果为 this 对象
operator^=	当前 bitset 对象与另一个 bitset 对象按位异或, 结果为 this 对象
operator>>(size_t n)	当前 bitset 的拷贝对象右移 n 位, 返回新的 bitset, 当前 bitset 不变
operator<<(size_t n)	当前 bitset 的拷贝对象左移 n 位, 返回新的 bitset, 当前 bitset 不变
operator>>=	当前 bitset 右移 n 位, 结果为 this 对象
operator<<=	当前 bitset 左移 n 位, 结果为 this 对象
set()	把当前 bitset 的全部 bit 都设置为 1
set(size_t p, bool)	把指定位置的 bit 设置为指定的 bool 值
reset()	把当前 bitset 的所有 bit 都设置为 0
reset(size_t p)	把指定位置的 bit 设置为 0
operator~	把当前 bitset 的拷贝的每一个 bit 取反, 返回新的 bitset 对象, 当前 bitset 不变
flip()	把当前 bitset 整个按位翻转
flip(size_t p)	把位置在 p 处的 bit 翻转
to_ulong()	把当前 bitset 的拷贝对象转换为 unsigned long 后返回该数
to_string()	把当前 bitset 的拷贝对象转换为 string 对象(0/1)后返回该 string 对象的值
count()	返回当前 bitset 中值为 1 的 bit 的个数
size()	返回当前 bitset 中包含的 bit 总个数
operator==	判断当前 bitset 和给定的 bitset 是否对应位完全一致
operator!=	如果当前 bitset 和给定的 bitset 至少有一个对应位不等, 就返回 true
test(size_t p)	如果位置为 p 的 bit 为 1, 则返回 true
any()	如果当前 bitset 中至少有一个 bit 为 1, 则返回 true
none()	如果当前 bitset 中没有有一个 bit 的值为 1 (即全 0), 则返回 true
friend member	
operator&	把两个给定的 bitset 按位与, 返回新的 bitset 对象
operator	把两个给定的 bitset 按位或, 返回新的 bitset 对象

(续表)

方 法	解 释
<code>operator ^</code>	把两个给定的 bitset 按位异或, 返回新的 bitset 对象
<code>operator &lt;&lt;</code>	把给定的 bitset 转换为 0/1 串输出到输出流中
<code>operator &gt;&gt;</code>	把输入流中输入的 0/1 串转换为 bitset

### 17.8.3 节省存储空间的 `vector<bool>`

`vector<bool>` 通过把用户指定的 `bool` 值位数折合成由若干个无符号整数 (WORD) 组成的 `vector` 来节约存储空间, 其换算方法不同于 `bitset`, 因为数组不能为空, `vector` 则可以为空:

$$Nword = (L + \text{sizeof}(\text{unsigned int}) * 8 - 1) / (\text{sizeof}(\text{unsigned int}) * 8)$$

其中, `L` 表示当前 `vector` 的长度, 即包含的 `bool` 值的个数。要注意的是, `vector<bool>` 中的元素并非 C++ 的 `bool` 变量 (byte), 而是一个 bit。由于这种独特的存储机制, `vector<bool>` 的迭代器必须重新实现。`vector<bool>` 的存储结构如图 17-12 所示。

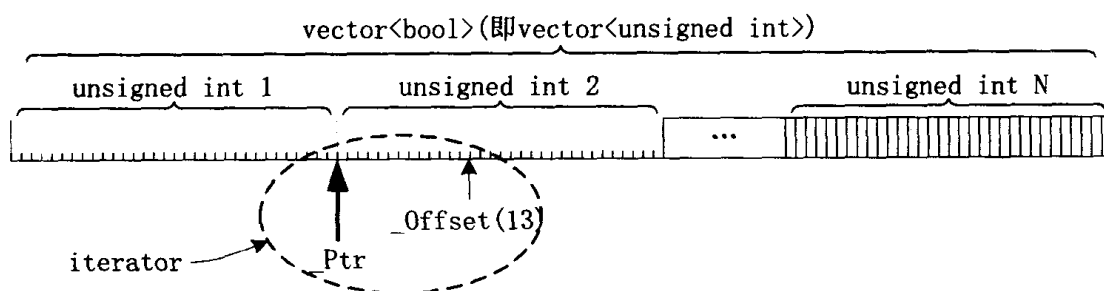


图 17-12 `vector<bool>` 的存储结构

`vector<bool>` 的 `reference` 也必须重新定义, 而不能是简单的 `T&`, 其接口基本上和 `bitset::reference` 一样。对每一个 `bool` 元素的访问就是通过图 17-12 所示的迭代器和一个掩码 (Mask) 进行运算完成的, 并返回 `vector<bool>::reference` 对象。你可以把一个 `bool` 变量指派给 `vector<bool>` 的任一个元素, 同样也可以把任一个元素转换成 `bool` 变量。具体情况请阅读 STL 源代码。

虽然 `vector<bool>` 在底层存储机制上与 `bitset` 类似, 但是它是一个真正的容器, 因此可以动态插入、删除元素, 会重新分配内存, 存在迭代器失效等问题, 但是不支持 `bitset` 那样丰富的位操作, 仅支持简单的“位取反 (flip)”操作。尽管 `vector<bool>` 的底层存储方式比较特殊, 但是其上层访问接口和基本模板 `vector<T>` 基本一致, 见示例 17-18。

示例 17-18

```
#include <vector>
#include <iostream>
```

```
using namespace std;
void main()
{
    typedef vector<bool> BoolVector; // vector<bool>的具体名字与 STL 实现有关
    BoolVector bvect;
    bvect.reserve(100);
    for (int i = 0; i < 100; i += 2) {
        bvect.insert(bvect.end(), true);
        bvect.insert(bvect.end(), false);
    }
    for (BoolVector::iterator first = bvect.begin(),
        last = bvect.end(); first != last; ++first) {
        cout << *first << endl;
    }
    cout << bvect.size() << endl;
    bvect.front().flip();
    bvect.back().flip();
    bvect.flip();
    bvect.clear();
}
```

`vector<bool>`的迭代器就像 `vector<T>`一样是随机访问迭代器,因此你可以随机访问其中任何一个 `bool` 值。在具体应用中,你可以把 `vector<bool>`的元素看成是一个个的 C++ `bool` 变量,而不必关心其底层存储方式。

## 17.8.4 空容器

由于容器的独特存储机制,使得容器可以是空的,而不像数组那样不能为空。这是因为,虽然容器不包含任何元素,但是容器对象本身还是存在于内存中的,只不过它的那些指针数据成员都为 0 而已;而如果数组为空,那也就意味着数组对象本身就不存在,所以不存在空数组。

对空容器进行某些操作可能会产生错误。如对一个空容器调用 `erase()`或 `pop()`这类方法就会出错。如果你无法确定容器是否为空,可以首先使用 `empty()`来检查,见示例 17-19。

示例 17-19

```
#include <list>
using namespace std;
void main()
{
    list<int> li; // 空容器
    li.pop_back(); // runtime error!
    if (!li.empty()) li.erase(li.begin());
}
```

从容器中删除一个本不属于它的元素和从空容器中删除元素一样危险（它们是一回事儿）！因此，从容器中删除元素前一定要检查该元素是否属于该容器。不过这里又需要区别对待顺序容器和关联式容器了：

- ✧ 在顺序容器中，一个元素是否属于该容器要看它的内存空间是否属于该容器所有，而不是看元素对象的值。因此只能通过指定位置（迭代器）来删除元素，此时它不用检查指定的元素是否属于自己。如果你指定的对象确实不属于该容器所有，那么删除动作会抛出异常。
- ✧ 关联式容器是通过值来定位元素对象的，因此通过指定值来删除元素的时候，容器会检查该值是否与自己的某一个元素对象匹配，如果找到则删除与该值相等的所有元素对象，反之简单返回。当然你也可以通过指定位置来删除元素，但是其结果与从顺序容器中删除元素是一样的。

见示例 17-20。

示例 17-20

```
#include <list>           // 顺序容器
#include <set>             // 联合容器
#include <iostream>
using namespace std;
void main()
{
    int *p = new int(100);
    list<int> temp1;
    temp1.insert(temp1.end(), 10);
    list<int> li;           // 空容器
    li.insert(li.end(), 10); // 创建新对象
    li.insert(li.end(), 10); // 创建新对象
    li.erase(temp1.begin()); // runtime error! 虽然 li 中也有值为 10 的 int 元素
                             // 但是 temp1.begin()指向的元素对象并不属于 li 所有
    li.erase(li.begin());   // ok! li.begin()指向的对象为 li 所有
    li.erase(*p);           // compiling error! 没有定义这样的方法，对象*p 不
                             // 属于 li 所有。不能以元素的值来判断其是否属于一
                             // 个顺序容器，因为顺序容器可以同时存储值完全相同
                             // 的多个对象；否则容器将不知道该删除哪一个元素了

    set<int> temp2;
    temp2.insert(10);
    set<int> si;           // 空容器
    si.insert(10);         // si 包含一个值为 10 的元素
    si.insert(si.end(), 20);
    si.insert(si.end(), 20); // 忽略！
    si.erase(temp2.begin()); // runtime error! temp2.begin()指向的对象不属于 si 所有
    si.erase(*p);           // OK! 从 si 中删除值等于*p 的元素
    si.erase(si.begin());   // OK! si.begin()指向的对象为 si 所有
    delete p;
}
```

}

除了上面这些特殊容器外, 还有一些专用于数学运算的容器、算法和类型, 如 `valarray`、`complex` 等, 分别定义在头文件 `<valarray>` 和 `<complex>` 中, 头文件 `<numeric>` 中定义了用于向量计算的算法。关于它们的用法请参考其他书籍或者 STL 源代码。

## 17.9 STL 容器特征总结

STL 中顺序容器类和关联式容器类的主要特征如下。

### (1) vector

- ◇ 内部数据结构: 连续存储, 例如数组。
- ◇ 随机访问每个元素, 所需要的时间为常量。
- ◇ 在末尾增加或删除元素所需时间与元素数目无关, 在中间或开头增加或删除元素所需时间随元素数目呈线性变化。
- ◇ 可动态增加或减少元素, 内存管理自动完成, 但程序员可以使用 `reserve()` 成员函数来管理内存。
- ◇ `vector` 的迭代器在内存重新分配时将失效(它所指向的元素在该操作的前后不再相同)。当把超过 `capacity() - size()` 个元素插入 `vector` 中时, 内存会重新分配, 所有的迭代器都将失效; 否则, 指向当前元素以后的任何元素的迭代器都将失效。当删除元素时, 指向被删除元素以后的任何元素的迭代器都将失效。

**【建议 17-8】:** 使用 `vector` 时, 用 `reserve()` 成员函数预先分配需要的内存空间, 它既可以保护迭代器使之不会失效, 又可以提高运行效率。

### (2) deque

- ◇ 内部数据结构: 连续存储或分段连续存储, 具体依赖于实现。
- ◇ 随机访问每个元素, 所需要的时间为常量。
- ◇ 在开头和末尾增加元素所需时间与元素数目无关, 在中间增加或删除元素所需时间随元素数目呈线性变化。
- ◇ 可动态增加或减少元素, 内存管理自动完成, 不提供用于内存管理的成员函数。
- ◇ 增加任何元素都将使 `deque` 的迭代器失效。在 `deque` 的中间删除元素将使迭代器失效。在 `deque` 的头或尾删除元素时, 只有指向该元素的迭代器失效。

### (3) list

- ◇ 内部数据结构: 双向环状链表。
- ◇ 不能随机访问一个元素。
- ◇ 可双向遍历。
- ◇ 在开头、末尾和中间任何地方增加或删除元素所需时间都为常量。
- ◇ 可动态增加或减少元素, 内存管理自动完成。

- ✧ 增加任何元素都不会使迭代器失效。删除元素时，除了指向当前被删除元素的迭代器外，其他迭代器都不会失效。

#### (4) `slist`

- ✧ 内部数据结构：单向链表。
- ✧ 不可双向遍历，只能从前向后遍历。
- ✧ 其他特性同 `list` 相似。

**【建议 17-9】：** 尽量不要使用 `slist` 的 `insert`, `erase`, `previous` 等操作。因为这些操作需要向前遍历，但是 `slist` 不能直接向前遍历，所以它会从头开始向后搜索，所需时间与位于当前元素之前的元素个数成正比。`slist` 专门提供了 `insert_after`, `erase_after` 等函数进行优化。但若经常需要向前遍历，建议选用 `list`。

#### (5) `stack`

- ✧ 适配器，它可以将任意类型的序列容器转换为一个堆栈，一般使用 `deque` 作为支持的序列容器。
- ✧ 元素只能后进先出（LIFO）。
- ✧ 不支持遍历操作。

#### (6) `queue`

- ✧ 适配器，它可以将任意类型的序列容器转换为一个队列，一般使用 `deque` 作为支持的序列容器。
- ✧ 元素只能先进先出（FIFO）。
- ✧ 不支持遍历操作。

#### (7) `priority_queue`

- ✧ 适配器，它可以将任意类型的序列容器转换为一个优先级队列，一般使用 `vector` 作为底层存储结构。
- ✧ 只能访问第一个元素，不支持遍历操作。
- ✧ 第一个元素始终是优先级最高的元素。

**【建议 17-10】：** 当需要 `stack`、`queue` 或 `priority_queue` 这样的数据结构时，直接使用对应的容器类，不要使用 `deque` 去做它们类似的工作。

#### (8) `set`

- ✧ 键和值相等。
- ✧ 键唯一。
- ✧ 元素默认按升序排列。
- ✧ 如果迭代器指向的元素被删除，则该迭代器失效。其他任何增加、删除元素的操作都不会使迭代器失效。

#### (9) `multiset`

- ✧ 键可以不唯一。
- ✧ 其他特点与 `set` 相同。

## （10）hash\_set

- ✧ 与 set 相比较，它里面的元素不一定是经过排序的，而是按照所用的 hash 函数分派的，它能提供更快搜索速度（当然跟 hash 函数有关）。
- ✧ 其他特点与 set 相同。

## （11）hash\_multiset

- ✧ 键可以不唯一。
- ✧ 其他特点与 hash\_set 相同。

## （12）map

- ✧ 键唯一。
- ✧ 元素默认按键的升序排列。
- ✧ 如果迭代器所指向的元素被删除，则该迭代器失效。其他任何增加、删除元素的操作都不会使迭代器失效。

## （13）multimap

- ✧ 键可以不唯一。
- ✧ 其他特点与 map 相同。

## （14）hash\_map

- ✧ 与 map 相比较，它里面的元素不一定是按键值排序的，而是按照所用的 hash 函数分派的，它能提供更快搜索速度（当然也跟 hash 函数有关）。
- ✧ 其他特点与 map 相同。

## （15）hash\_multimap

- ✧ 键可以不唯一。
- ✧ 其他特点与 hash\_map 相同。

**【建议 17-11】：** 当元素的有序比搜索速度更重要时，应选用 set、multiset、map 或 multimap。否则，选用 hash\_set、hash\_multiset、hash\_map 或 hash\_multimap。

**【建议 17-12】：** 往容器中插入元素时，若元素在容器中的顺序无关紧要，请尽量加在最后面。若经常需要在序列容器的开头或中间增加或删除元素时，应选用 list。

**【建议 17-13】：** 对关联式容器而言，尽量不要使用 C 风格的字符串（即字符指针）作为键值。如果非用不可，应显式地定义字符串比较运算符，即 operator<、operator==、operator<=等。

**【建议 17-14】：** 当容器作为参数被传递时，请采用引用传递方式。否则将调用容器的拷贝构造函数，其开销是难以想象的。

## 17.10 STL 使用心得

STL 组件与平台无关，与应用无关，与数据类型无关，几乎在任何应用程序开发中都可以使用；它不仅大大减少编程工作量，提高编程效率，而且也减少了



编程出错的机会；它不仅可以提高代码的可读性、清晰性，还可提高应用程序的健壮性、性能和可移植性。因此我们要尽可能地使用它们。

不过，千万不要“为了使用 STL 而使用 STL”。比如，如果你仅需要对一两个字节的数据进行简单的移位操作，那就没有必要“劳驾” `bitset`；如果可以使用静态数组，也就不需要“劳驾” `vector`；如果你仅是比较两个 C 字符串，直接调用 C 库函数 `strcmp()` 好了，没有必要把它们转换成 `string` 对象，然后再调用重载的运算符或者 `string::compare()` 来比较。

容易使你迷失的是 STL 中几乎每一个部分都充斥着 `Template`、迭代器和重载的运算符，如果你对它们一无所知，在 STL 的海洋里就会寸步难行。

虽然 STL 有许多优点，可是学习和掌握 STL 并不是一件轻松的事儿。你首先必须了解隐藏在 STL 组件背后的设计原理和技术，然后才能运用自如，出了问题也容易定位。如果你仅是停留在“使用”这个层次上，那么当出现问题而问题又并非位于表面时，你可能就会“找不着北”，甚至开始埋怨 STL 一点也不好用，其实问题往往出在自己这里。

侯捷认为学习和使用 STL 分 3 个层次（或 3 个境界、3 个阶段）：运用、研究、扩展。扩展的前提是必须对 STL 有很深的造诣，无论从宏观还是微观上都对它了如指掌；扩展的要求是必须遵守 STL 的“游戏规则”，才能使新开发的组件较好地融入整个 STL 体系，并能够与已有组件很好地搭配运用。作者也仅是处于第二阶段而已，何况本章对许多复杂的组件没有涉及，比如函数对象、数学库及其算法、国际化和本地化支持、I/O 流库及玄妙的 `traits` 技术等，要学的东西真是太多了！下列图书无论是从宏观还是微观上对泛型编程及 STL 都有极为精辟的论述，很值得阅读，特此推荐：

- (1) 侯捷著，《泛型思维》；
- (2) 侯捷著，《STL 源码剖析》；
- (3) Meyers 著，《Effective STL》；
- (4) Vandevoorde 著，《C++ Templates》。



# 附录 A C++/C 试题

假设本试题均为 Windows NT 下的 32 位 C++ 程序。本试题仅用于考查 C++/C 程序员的基本编程技能，不涉及数据结构、算法，以及深奥的语法。考试成绩能反映出考生的编程质量，以及对 C++/C 的理解程度，但不能反映考生的智力和软件开发能力。笔试时间为 90 分钟。请考生认真答题，切勿轻视。

## 一、请计算 sizeof 表达式和 strlen 表达式的值（10 分）

```
char s1[] = "";  
char s2[] = "Hello World!";  
char *p = s2;  
char *q = NULL;  
void *r = malloc(100);
```

请计算：

```
sizeof(s1) =  
sizeof(s2) =  
sizeof(p) =  
sizeof(q) =  
sizeof(r) =
```

```
char s1[10] = {'m', 'o', 'b', 'i', 'l'};  
char s2[20] = {'A', 'N', 'S', 'I', '\0', 'C', '+', '+'};  
char s3[6] = {'T', 'S', 'O', 'C', '+', '+'};
```

请计算：

```
strlen(s1) =  
strlen(s2) =  
strlen(s3) =  
s2[8] = ?
```

```
void Func(char str[100])
```

```
{
```

```
    请计算 sizeof(str) =
```

```
}
```

## 二、请填写 bool、float、指针变量与“零值”比较的 if 语句。（10 分）

提示：这里“零值”可以是 0、0.0、FALSE 或者“空指针”。例如 int 变量 n 与“零值”比较的 if 语句为：  
if(n == 0);      if(n != 0);  
依次类推。

请写出 bool flag 与“零值”比较的 if 语句：

请写出 float x 与“零值”比较的 if 语句：

请写出 char \*p 与“零值”比较的 if 语句：

## 三、简答题（20 分）

- （1）头文件中的 ifndef/define/endif 是干什么用的？
- （2）#include <filename.h> 和 #include "filename.h" 有什么区别？

# 高质量程序设计指南

——C++/C语言 (第 3 版)

2. const 有什么用途? (请至少说明两种)
3. 在 C++ 程序中调用被 C 编译器编译后的函数, 为什么要加 extern "C" 声明?
4. 请简述以下两个 for 循环的优缺点。

<pre>// 第一个 for (i=0; i&lt;N; i++) {     if (condition)         DoSomething();     else         DoOtherthing(); }</pre>	<pre>// 第二个 if (condition) {     for (i=0; i&lt;N; i++)         DoSomething(); } else {     for (i=0; i&lt;N; i++)         DoOtherthing(); }</pre>
优点:	优点:
缺点:	缺点:

## 四、有关内存的思考题 (20 分)

<pre>void GetMemory(char *p) {     p = (char *)malloc(100); }  void Test(void) {     char *str = NULL;     GetMemory(str);     strcpy(str, "hello world");     printf(str); }</pre> <p>请问运行 Test 函数会有什么样的结果? 为什么?</p>	<pre>char *GetMemory(void) {     char p[] = "hello world";     return p; }  void Test(void) {     char *str = NULL;     str = GetMemory();     printf(str); }</pre> <p>请问运行 Test 函数会有什么样的结果? 为什么?</p>
---	---

<pre> void GetMemory(char **p, int num) {     *p = (char *)malloc(num); }  void Test(void) {     char *str = NULL;     GetMemory(&amp;str, 100);     strcpy(str, "hello");     printf(str); } </pre> <p>请问运行 Test 函数会有什么样的结果？为什么？</p>	<pre> void Test(void) {     char *str = (char *) malloc(100);     strcpy(str, "hello");     free(str);     if(str != NULL)     {         strcpy(str, "world");         printf(str);     } } </pre> <p>请问运行 Test 函数会有什么样的结果？为什么？</p>
---	---

### 五、类型转换（5 分）

```

double d = 100.25;
int x = d;
int *pInt = (int *) &d;

```

请问以下两个输出语句的结果是否相同？为什么？

```

cout << x << endl;
cout << *pInt << endl;

```

### 六、编写 strcpy 函数（10 分）

已知 strcpy 函数的原型是：

```
char *strcpy(char *strDest, const char *strSrc);
```

其中，strDest 是目的字符串，strSrc 是源字符串。

（1）不调用 C++/C 的字符串库函数，请编写函数 strcpy：

（2）strcpy 能把 strSrc 的内容拷贝到 strDest，为什么还要 char \* 类型的返回值？

### 七、编写类 String 的普通和拷贝构造函数、析构函数和赋值函数（25 分）

已知类 String 的原型为：

```

class String {
public:
    String(const char *str = "");           // 普通构造函数
    String(const String &other);           // 拷贝构造函数

```

```
        ~String(void);                // 析构函数
        String& operate =(const String &other); // 赋值函数
    private:
        char      *m_data;            // 用于保存字符串
};
```

请编写 String 的上述 4 个基本函数。

# 附录 B C++/C 试题答案与评分标准

## 一、计算 sizeof 表达式和 strlen 表达式的值。(每小题 1 分, 共 10 分)

<pre>char s1[] = ""; char s2[] = "Hello World"; char *p = s2; char *q = NULL; void *r = malloc(100);  请计算  sizeof(s1) = 1 sizeof(s2) = 13 sizeof(p) = 4 sizeof(q) = 4 sizeof(r) = 4</pre>	<pre>char s1[10] = {'m', 'o', 'b', 'i', 'l'}; char s2[20] = {'A', 'N', 'S', 'T', '\0', 'C', '+', '+'}; char s3[6] = {'T', 'S', 'O', 'C', '+', '+'};  请计算  strlen(s1) = 5 strlen(s2) = 4 strlen(s3) = 不确定  s2[8] = '\0' 或者 0 或者 0x00 都对</pre>
	<pre>void Func(char str[100]) {     请计算 sizeof(str) = 4 }</pre>

## 二、填写 bool、float、指针变量与“零值”比较的 if 语句。(10 分)

请写出 bool flag 与“零值”比较的 if 语句 (3 分)	
标准风格: <pre>if(flag) if(!flag)</pre>	如下写法均属不良风格, 不得分 <pre>if(flag == TRUE) if(flag != TRUE) if(flag == 1)</pre>
请写出 float x 与“零值”比较的 if 语句 (4 分)	
标准风格: // 精度 EPSILON 根据应用要求而定 <pre>const float EPSILON = 1e-6; if((x &gt;= -EPSILON) &amp;&amp; (x &lt;= EPSILON))</pre>	不可将浮点变量用“==”或“!=”与数字比较, 应该设法转化成“>=”或“<=”此类形式。如下是错误的写法, 不得分。 <pre>if(x == 0.0) if(x != 0.0)</pre>
请写出 char *p 与“零值”比较的 if 语句 (3 分)	
标准风格: <pre>if(p == NULL) if(p != NULL)</pre>	如下写法均属不良风格, 不得分 <pre>if(p == 0)    if(p != 0) if(p)         if(!p)</pre>

## 三、简答题。（每题5分，共20分）

1. (1) 头文件中的 `ifndef/define/endif` 是干什么用的？

(2) `#include <filename.h>` 和 `#include "filename.h"` 有什么区别？

答：

(1) 防止该头文件被重复引用。

(2) 对于 `#include <filename.h>`，编译器从开发环境设置的路径开始搜索 `filename.h`；对于 `#include "filename.h"`，编译器从用户的工作路径开始搜索 `filename.h`。

2. `const` 有什么用途？（请至少说明两种）

答：

(1) 可以定义 `const` 常量。

(2) `const` 可以修饰函数的参数、返回值及函数的定义体。被 `const` 修饰的东西都受到强制保护，可以预防意外地修改，能提高程序的健壮性。

3. 在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 `extern "C"`？

答：

C++ 语言支持函数重载，C 语言不支持函数重载。函数被 C++ 编译器编译和被 C 编译器编译后生成的内部名字是不同的。假设某个函数的原型为：`void foo(int x, int y)`；该函数被 C 编译器编译后的内部名字为 `_foo`，而 C++ 编译器则会产生像 `_foo_int_int` 之类的名字。C++ 提供了 C 连接交换指定符号 `extern "C"` 来解决名字匹配问题（即二进制兼容性问题）。

4. 请简述以下两个 for 循环的优缺点。

<pre>// 第一个 for (i=0; i&lt;N; i++) {     if (condition)         DoSomething();     else         DoOtherthing(); }</pre>	<pre>// 第二个 if (condition) {     for (i=0; i&lt;N; i++)         DoSomething(); } else {     for (i=0; i&lt;N; i++)         DoOtherthing(); }</pre>
优点：程序简洁。 缺点：多执行了 $N-1$ 次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。	优点：循环的效率 high。 缺点：程序不简洁。



## 四、有关内存的思考题。(每小题 5 分, 共 20 分)

```
void GetMemory(char *p)
{
    p = (char *)malloc(100);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(str);
    strcpy(str, "hello world");
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果?

答: 程序崩溃。

因为 GetMemory 并不能传递动态内存, Test 函数中的 str 一直都是 NULL。strcpy(str, "hello world");将使程序崩溃。

```
char *GetMemory(void)
{
    char p[] = "hello world";
    return p;
}
void Test(void)
{
    char *str = NULL;
    str = GetMemory();
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果?

答: 可能是乱码。

因为 GetMemory 返回的是指向“栈内存”的指针, 该指针的地址不是 NULL, 但其原有的内容已经被清除, 新内容不可知。

```
void GetMemory (char **p, int num)
{
    *p = (char *)malloc(num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果?

答:

- (1) 能够输出“hello”。
- (2) 内存泄漏。

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str, "hello");
    free(str);
    if(str != NULL)
    {
        strcpy(str, "world");
        printf(str);
    }
}
```

请问运行 Test 函数会有什么样的结果?

答: 篡改动态内存区的内容, 后果难以预料, 非常危险。因为 free(str);之后, str 成为野指针, if(str != NULL)语句不起作用。

## 五、类型转换（5分）

```
double d = 100.25;
int x = d;
int *pInt = (int *) &d;
```

请问以下两个输出语句的结果是否相同？为什么？

```
cout << x << endl ;
cout << *pInt << endl ;
```

答：两个输出结果不相同。第一个结果为 100，x 取 d 的整数部分；第二个结果不是 100，\*pInt 等于 d 的前 4 字节的数值，而不是 d 的整数部分。

## 六、编写 strcpy 函数（10分）

已知 strcpy 函数的原型为：

```
char *strcpy(char *strDest, const char *strSrc);
```

其中，strDest 是目的字符串，strSrc 是源字符串。

（1）不调用 C++/C 的字符串库函数，请编写函数 strcpy：

```
char *strcpy(char *strDest, const char *strSrc);
{
    assert((strDest!=NULL) && (strSrc !=NULL));    // 2 分
    char *address = strDest;                        // 2 分
    while( (*strDest++ = * strSrc++) != '\0' )      // 2 分
        NULL ;
    return address ;                                // 2 分
}
```

（2）strcpy 能把 strSrc 的内容拷贝到 strDest，为什么还要 char \* 类型的返回值？

答：为了实现链式表达式。 // 2 分

例如：int length = strlen( strcpy( strDest, "hello world" ) );

## 七、编写类 String 的构造函数、析构函数和赋值函数（25分）

```
// String 的普通构造函数
String::String(const char *str)                // 6 分
{
    if(str==NULL)
    {
        m_data = new char[1];
        *m_data = '\0';
    }
    else
```

```
{
    int length = strlen(str);
    m_data = new char[length+1];
    strcpy(m_data, str);
}
}
// String 的析构函数
String::~String(void)                                // 3 分
{
    delete [] m_data;
}
// 拷贝构造函数
String::String(const String &other)                  // 3 分
{
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, other.m_data);
}
// 赋值函数
String & String::operator =(const String &other)    // 13 分
{
    // (1) 检查自赋值                                // 3 分
    if(this != &other)
    {
        // (2) 分配新的内存资源，并拷贝内容        // 4 分
        char *temp = new char[strlen(other.m_data) + 1];
        strcpy(temp, other.m_data);
        // (3) 其次释放原有的内存资源                // 3 分
        delete []m_data;
        m_data = temp;
    }
    // (4) 返回本对象的引用                            // 3 分
    return *this;
}
```



## 附录C 大学十年

写此文我很为难，一是担心读者误以为我轻浮得现在就开始写自传，二是担心朋友们误以为我得了绝症而早早留下遗作。

不论是落俗套还是不落俗套地评价，我在大学十年里都是出类拔萃的好学生。并且一直以来我对朋友们和一些低年级的学生们都有很大的正面影响。这十年是一个从幼稚到成熟的过程，交织着聪明与愚蠢、勤奋与懒散、狂热与怯懦、成功与失败。做对了的事可树立为榜样，做错了的事可挂作警钟。我写下经历与感受，期望以此引导和勉励无数比我年轻的学生们。我资历尚浅，既没有哲学家的深邃，也没有诗人的风华，不足以堂皇地育人，只能讲一些故事以表心愿。

我出生在1973年的春节，属牛，是“牛头”。父母为我起了很好听的名字叫“林锐”。这暗示着上天对我别有用心，将降大任于我，可是这时候上帝去了一趟厕所。天堂与人间的时差如此之大，就在上帝大小便的几分钟内，我混混沌沌地度过了童年和少年，天才因此成为凡人。

我小时候生长在浙江黄岩的偏僻山区。父母都是中学教师，由于山区师资缺乏，父母经常要从一个山头调到另一个山头教学。我换读过的小学的数目比我的年龄还大，没有伙伴，也没有家的概念。我就像活在货郎担里的小鸡，缩成一团，在高兴或恐惧时至多“啾”“啾”地叫几声。我在读小学与初中的8年里，既不聪明活泼，也不调皮捣蛋，确切地说像块木头，简直是我名字的反义词。在学习上我没有受过一次表扬，也没有任何值得留念的人或事。唉，无论我现在多么努力都已无法追回失去的8年金色年华，好心痛！

我草草地并且稀里糊涂地在13岁时从初中毕业，无处可去。这下我发慌了，开始渴望学习。我灰溜溜地离开山区，可怜巴巴地到一个比较好的乡下中学重读初三。我勤快得早晨4:30就起来读英语，脑袋似乎也被吓开窍了，“数理化”学得很好，并且生平第一次在物理考试中得了满分。当我“再一次”从初中毕业时，我以全校第一的成绩考入了黄岩中学读高中。

黄岩中学分农村班与城市班，我当然是农民阶级。“阶级划分与出身成分”对我是相当有促进作用的。我连任了几年的卫生委员，星期六和星期天同学们习惯性地把活留给我，我这小官当得有滋有味。物理学得极好，有一种直觉帮我快速准确地解题，常常是老师刚把题目写完我就报出答案来。上物理课时我没法讲废话，因为我一开口就是标准答案。

可惜我的文科成绩极差。那时期盛传“学好数理化，走遍天下都不怕”，我们年少不懂事，糟蹋了学文科的好时光。我写作文的最高目标就是不跑题，考试前我总是反复祈祷：我没干过坏事，保佑我作文不跑题吧！历史考试时填写“任课老师某年某月某日在我家乡英勇就义”，和同学比谁的成绩更接近零分。更让我沮丧的是，

这些行径都不是我发明的，我顶多是个跟屁虫而已，一点回忆的自豪感都没有。

我现在认为文科教育的实质是素质教育，如果素质不高，男孩再聪明也难以成大器，当然也难以吸引好女孩。

高考时我语文得了 54 分（是班里的中上水平），总分只比重点线高十几分。我不敢报考好地方，只好选择内地。选来选去觉得西安与成都两个城市都还不错，我拿把尺子在地图上一量，发现我家乡离西安的直线距离较短，于是就选择了西安。老师们只听说过西安交通大学比较有名气，但谁也不了解。我以为在西安交通大学是学习开火车、开轮船的，尽管我也很渴望能开车开船，但考虑到自己的身材矮小，就只能忍痛割爱了。我觉得西安电子科技大学的名字很好听，符合我做科学家的梦想，于是就报考了西安电子科技大学（以下简称西电）技术物理系。

上帝精神抖擞地从厕所回来，发现我已经上大学。也许他原先想把我安排在清华或者北大的，但事已至此，干脆也就撒手不管了。他这一偷懒反而是好事，我在读大学的十年中自由发展，成了卓尔不群的学生。

刚进西电，首先吸引我的是麻雀和馍。那麻雀滚圆滚圆的，简直是会飞的肉弹。它们不怕人，成堆聚集吵闹，常让我误以为是没有管教的一群小鸡。那馍又白又大，既不放盐也不放糖，既不像馒头也不像包子。馍凉了后贼硬，据说有同学被楼上扔下的半块馍砸中脑袋，当场长出一个“肉包子”。最好笑的是人们把“馍夹肉”叫成“肉夹馍”，那东西实在好吃。

西电原是全国闻名的军校，作风严谨，校园并不华丽，生活有些单调。尽管我来自山清水秀的地方，可也喜欢西电的粗犷与憨厚。有一天我看到一个新生写得很肉麻的赞美西电的海报，有一句是“我踏上了东去的列车”，我不禁笑掉牙。这一笑意味着“大个子欺负小个子”历史的结束，“小个子欺负大个子”新纪元的开始。

上大学的第一个学期刚好碰上美国攻打伊拉克（“沙漠风暴”行动）。那时全国都在谈电子战，我们全校都是研究电子的，而且以军事应用为主。在那种气氛里，同学们都有很强的使命感，并且被鼓动得信心十足。

一日，系主任视察早读，偏偏有同学迟到。系主任喝问：“你为什么迟到了？”

“因为我来迟了。”同学毫不含糊地回答，昂然入座。那时候的学生充满了理由。

我在班里年龄小个子也小，上课时就像猩猩堆里的猴子那么显眼。由于我们是物理系学生，第一学期的《普通物理》课程就显得非常重要。系副主任给我们上课，他长得像叶利钦，口若悬河，板书极快。像在高中上物理课那样，我常在“叶利钦”刚写完题目时就报出答案。开头几次，“叶利钦”满脸狐疑地扫视我们，好像是要抓住拔掉他自行车气门芯的那个捣蛋鬼。后来他在第一排发现了我，我俩乐得裂了嘴。课间休息时，“叶利钦”常坐在我旁边，乘他给同学们答疑时，我就用笔拨弄他硕大无比的手指，在他指甲上涂点什么。

在第一学年，我就像乱丛中的野花那样脱颖而出，倍受老师和同学们的关爱。就在我光荣到感觉屁股都能绽放光彩的时候，发现了令我胆战心惊的学习缺陷——不会做实验。一进实验室，我就束手无策，浑身发抖。我相信大一的学生们都有虚荣心，为了维护“最聪明”这个荣耀，我完全可以掩盖、躲避甚至偷偷地弥补实验能力的不足。

我做了一件了不起的事：为了对抗虚荣的引诱，我夸大其辞地把“缺陷”告诉每一个我认识的人，让我没有机会欺骗自己。

聪明的人并不见得都有智慧，他可能缺乏“真实”这种品质。虽然我是在硕士毕业的时候才立下誓言——“做真实、正直、优秀的科技人员”，但我在 18 岁的时候就已经做到了“真实”，我必定一生保持。

第一年暑假回家，得到一个惊喜：家里竟然有了电路实验室！

因为我常在信中鼓噪自己实验能力何等之差，“长此以往，下场将极为悲惨”。父母经不起这种“恐吓”，当英语教师的父亲将半年的工资连同“私有财产”全部捐出，每周到很远的商品交易市场购买电子元件及器材，在家里建立了实验室。父亲很威严，我从小就怕他，但那个暑假我一点也不怕他。我们一起做实验，都从零学起，话不投机就用电烙铁“交流”，完全是同事关系。后来，我的兴趣转向了计算机，家里的实验室就由父亲独掌，继续发扬光大。现在父亲修理电器的水平在家乡远近闻名，学生们都忘了他是英语教师。

母亲是数学教师，年轻时略有姿色，智力远胜过我父亲。当她与他在山头的学校里相遇时，他一顿热情洋溢的饭菜就让她“缴械投降”了。我小时候家里很贫穷，家就像一条飘荡的小船，父亲划桨，母亲掌舵。当我 6 岁上学时，母亲就说：“儿子啊，你将来只能靠笔吃饭而不能靠锄头吃饭。”小时候，母亲怕我变狡诈而不允许我学下棋。尽管我在大学里已经相当出色，母亲来信总不忘叮嘱“德智体全面发展”。她常用独特的方式检查我：

第一，看我是否变胖。如果我胖了，表明我懒了。因为勤奋的人没有理由变胖。

第二，看我说话是否还快。如果我说话慢条斯理，表明我变笨了。因为脑子灵活的人没有理由说话不快。

我读博士研究生时，母亲的眉头才舒展开。她经常在本校师生中发表自由言论：“儿子的智力与性格完全是遗传我的，他爸毫无半点功劳。”

本科第三学期的主要课程是电路分析。电路题目常常很滑稽，当你满头大汗地解完方程式时，答案往往是零。我归纳了不少公式用于简化计算，所以照样能在老师画完电路图时报出答案。学习是如此轻松以至于我有太多的课余时间。

在课余我常做两件有意思的事：

一件事是我为学习较差的十几名同学办了补习班，给他们讲课，改作业，出考题。我就像老母鸡那样看护着一群小鸡，使班长、学习委员等班干部都形同虚设。我这样做既提高了自己的表达能力，又帮助了同学。这事不是老师叫我干的，是我自己的主意。

另一件事是我经常在宿舍里焊接电子线路，技艺渐精。我曾花了两天时间，把磁带盒做成能发声、发光的精美礼物，乐颠颠地送给一个女孩子。可惜不久后我迷上了计算机，从此再也没亲手做过好玩的东西。

上大学以前我根本没见过计算机。在第四学期时我遇到了十年来最敬爱的老师周维真，从而对编程产生了强烈的兴趣。他教我们 Fortran 语言，Fortran 语言本身对我没有影响，影响我的是周老师高尚的师德及他在教学和科研中的敬业精神。我从他那里学到的是怎样做人，怎样做事。

很多计算机系老师改作业时喜欢打“√”或打“×”了事。周老师不仅把作业里的错误都找出来,而且逐一评注“好在哪里”和“差在哪里”。为了不让周老师过于劳累,全系同学有一个约定:上课时不准吵闹,否则别来;作业必须清楚,否则别交;提的问题必须有质量,否则闭嘴。

Fortran 语言期末考试,我的卷面成绩是 97 分,有个女同学考了 99 分。我当时为课代表,想不到被一个女生超过,甚为沮丧。可是报到系里的成绩单上,我的成绩是 99 分,那个女同学是 97 分。我以为周老师搞错了,跑去问他。周老师笑笑说:“你平时的学习表现,该得满分。不能因为考试中的一个失误而打击你的积极性,所以给你加 2 分作为鼓励。而她一上机就束手无策,要让她知道考试成绩高并不表明已经学好了,扣她 2 分以示警告。你本来就是第一名。”这时又跑来一个“查”成绩的同学,他得了 59 分,哀求周老师让他及格。周老师说:“你的试卷我看了好几遍,的确确是 59 分。而你平时的学习表现也不会超过 59 分。这一分不能加,否则我会害你一生。”

在我这一级(90 级),周老师至少为技术物理系教出两名软件高手——我和马佩军。我和马佩军读到硕士时已在软件方面雄霸西电,计算机系学生毫无“翻身”之望。由于马佩军不好名利,风头让我一人独得。我离开西电数年后,余威尚在。可惜我和周老师相处不到一年,他便调到北京信息工程学院。然而师恩的厚薄不在于时间长短,好的老师会让人想念、感激一辈子。

在上大学的前三个学期,学习如同表演,有趣而且轻松。自从第四学期学习了计算机课程,我就有了新的追求,我多么渴望拥有一台计算机,可以天天编程。如果挨一个巴掌能换取一分钟上机时间的话,我愿意每天挨 1440 个巴掌。如果非得加上一个期限不可,我希望是一万年。

我本科的专业是半导体物理,一二年级由系里负责教学,三四年级由微电子研究所负责教学。在第四学期末,我央求系里把我推荐到微电子所参加科研,贾新章教授“收留”了我。我踏进微电子所的那一脚,让我从纯粹学习转向了科研,从“高分低能”转向了“低分高能”。

我终于有了一台 286 电脑,那个暑假我就睡在实验室里,时时刻刻守着它,深夜里我一个人冲着它发笑,一会儿盖上布,一会儿掀开布,一会儿摸摸它的“脸”,一会儿理理它的“辫子”。我很快地完成了任务:设计一个“立方运算”的模拟电路,并且学会了 C 语言。

西电有一个好传统,每年冬季举办一次全校性的“星火杯”学生科技作品竞赛。每届都有六七百件作品展示,低年级的学生看后无不热血沸腾,跃跃欲试。我很希望能独立开发一套软件,参加本届“星火杯”竞赛。贾新章老师是研究集成电路可靠性的,见我如此热切,就让我开发“集成电路可靠性分析软件”。

我开始一边研究数值算法,一边设计软件。从炎热的 8 月到发冷的 11 月,几乎天天通宵编程,程序很快增长到一万多行。在离竞赛还剩一个月左右的时候,出现了大量的问题。不仅程序老是出错误,而且发现原先的算法并不有效。此时已经没人能够“救”我。贾老师不会编程,不知道问题究竟出在程序上还是出在算法上(实质上两者都有问题)。而那些懂软件开发的年青教师,实在看不明白我的上万行程序



是如何组织的。他们只能悲伤地看着我挣扎。由于我经常逃课，好学生变成了坏学生。系里对我意见极大，贾老师十分为难。不少老师和同学劝我赶紧“改邪归正”，放弃项目，不值得因小失大。

当时我有个无法动摇的信念：如果放弃一次，那么碰到下一个挫折时我就会继续放弃；如果坚持而成功，那么碰到下一个挑战时我会激励自己再取得成功。

在压力面前，我依然坚持。每当略有进展时，心里就一阵狂喜，但很快又会碰到新的困难，有时一坐就是 20 小时。每天在喜悦的巅峰与苦恼的深渊之间反复折腾。在竞赛前两天，我终于成功地完成了软件研发，结果获得软件与论文两个二等奖。这个荣誉本身不值得一提，并且我付出很重的代价——对物理专业失去兴趣而彻底抛弃了它。但那时我才 19 岁，在极限状态下磨练了意志，使我日后充满激情。

在本科四年级，我认识了微电子所的郝跃老师。他是数学博士，是微电子所最有才华、最潇洒、最有领导风范的青年学者。我常去向他请教数学问题，他讲得意气风发，我听得如痴如醉。我俩一个月的“交流量”很多硕士花一年时间也得不到。有一天，郝老师说：“你做我的学生吧。”我就毫不迟疑地从贾老师门下“跳槽”到郝老师门下。郝老师后来是我的硕士导师，他高兴时喜欢说：“好，很好，非常好！”我看着他升教授，当博导，荣任副校长，师生两人分别在各自的阶层中名声显赫。

在本科三四年级，我的专业课程没有一门及格过。但由于微电子所的老师们已经认可了我，就把我的卷面成绩作为及格线，“水涨船高，水落船低”，我对同学们的帮助莫大于此！如果要我考研究生，我绝对考不上。系主任安毓英教授觉得我将来会有前途，于是不顾别人反对，一锤定音让我免试读硕士研究生。

读硕士研究生时，我的科研条件相当好。导师十分开明，任我自由发展。我最喜欢做的事是设计图形用户界面和开发数据可视化软件。图形软件的最大魅力是即便它毫无用处，你也可以自我欣赏。总有人担心“花很多精力、物力让界面那么漂亮，图形那么逼真是否值得？”这种问题不能强求别人与你一致。我当时赞美女孩子的最高境界就是把她和我的图形软件相提并论。

我喜欢设计用户界面是因为自己有相当好的美感。在读本科时我模仿过六七个流行软件的界面。那时大家编程都用 Turbo C 2.0，我伪造了一个“Turbo C 2.6”。有个北大的博士生来实验室参观，看了“Turbo C 2.6”后非常羡慕地对我导师说：“郝老师，你们的工具比我们的先进多了”。

我常常向同学演示、卖弄自己开发的软件。觉得还不过瘾，就写了一篇名为“用户界面设计美学”的短文。凡是路过我实验室的同学都被我逮住，被迫听完我得意之极的朗读，茫然者与痛苦者居多。不久我的朗读便所向披靡，闻声者逃之夭夭。这篇文章我 6 年后照搬到博士学位论文中，可见当初写时的确有所“超前”。

我的研究工作基本上以集成电路的数值计算为主，数值计算产生的一堆数据常把我搞得晕头转向。我发现用图形来表征、解释数据可以让自己不再迷糊下去，那感觉就像刚睡醒时冲凉水一样。我硕士学位论文中的软件就是用图形来仿真集成电路生产过程中“缺陷”对成品率的影响。我并不是在看了学术论文后才开始研究可视化技术的，我是在做了工作后才发现那些好玩的技术叫做“可视化”。由于我肚子里头的确有货，在硕士一年级，我没有使用“剪刀”与“糨糊”（这是很多人写书的

法宝), 只花了三个月时间就写完了第一本著作《微机科学可视化系统设计》。

我在读硕士期间的工作强度与本科时的强度相当, 但工作方式有很大不同。我有了明确的目标: 一是开发自主知识产权的软件产品; 二是培养做领导的才能。这个目标可以通过团队工作, 参加全国性大学生科技竞赛而实现。

我在西电成立了“可视创意软件小组”, 马佩军、戴玉宏、马晓宇是我的主要技术伙伴, 帮手很多。有几个漂亮的女生负责宣传 (有一个长得像孟庭苇)。办公室里贴满了标语, 如“创造性的事业要靠激情来推动”, “生于忧患, 死于安乐”, “让春天消失”。还有大幅的“作战图”, 倒计时牌。每个伙伴写了一张“军令状”放在机器上, 我迄今还记录着那些纯真、活泼、充满激情的文字。那是多么艰苦却又幸福的日子, 夜里放震耳欲聋的音乐、咬尖辣椒提神, 有伙伴累得蹲在厕所里睡着了。

在 1994 年和 1995 年的冬季, 我们的软件作品分别获中国大学生应用科技发明大奖赛二等奖和全国大学生“挑战杯”学术科技作品竞赛二等奖。在西北地区, 我们是“老大”。我成了西电学生的榜样, 仰慕我的学生有一大批, 刚到浙大读博士时, 收到一个西电计算机系学生的信, 他说: “你走了, 我呆在西电也没有意思, 我准备考浙大的硕士生, 你到哪里我就跟到哪里。”

在硕士毕业前, 我在鉴定表上这样写道: “我热爱科技事业, 如同热爱生命一样。近 5 年的科研工作带给我最充实的生活, 也寄托着我美好的向往。可我同时也感到了痛苦, 因为 5 年来我耳闻目睹科研中太多的弄虚作假。我发誓做一名真实、正直、优秀的科技人员, 以正身自勉。”

我在西电度过了幸福的 6 年半, 最让我牵肠挂肚的是“吃”、“友情”和“爱情”。

当我第一次吃红红的和青青的辣椒时, “感动”得满脸是泪, 那滋味让我觉得前 17 年白活了。我在读硕士时已经能自力更生, 我开发的软件不仅竞赛获奖争了名气, 而且还挣了钱 (卖了二十多份软件, 平均每份挣 500 元)。写书得稿费 7000 元, 那时我简直就是富翁。这些钱的小部分用来给女孩子买礼物, 大部分用于和哥儿们吃吃喝喝。

我相信自己已经尝遍了西安的小吃, 并且发现了一个“秘密”: 最好吃的东西都在地摊上, 最香的东西一定是辣的, 最辣的东西一定是香的。曾经沧海难为水, 我在浙大的三年里很少再吃辣椒, 因为怕它玷污我心目中的辣椒。

在我小时候, 我爸很讨厌土豆 (在困难时期他吃了太多的土豆), 他竟然因此不让我吃土豆。我哪敢跟他论理, 于是忍啊忍, 一直忍到我上大学“远走高飞”。如果说辣椒是我新交的女朋友, 那么土豆就是我天生的命。我在西电经常用电炉 (从来都没被抓住过) 做“以土豆为核心”的菜, 天长日久, 朋友们干脆叫我“土豆”。

我吃饱土豆和辣椒后不免深思而感叹, 人要是认认真真地吃, 真的花不了多少钱, 那些贪官究竟是怎么吃掉巨款的? 我将来怎么吃得掉自己挣来的钱?

我在读中学与本科时, 满头白发, 脑袋可以当白炽灯泡用。当我硕士毕业再照镜子时, 吓了一跳, 白头发不见了! 我不知道究竟是哪种食物起的作用 (估计是辣椒)。那些早生白发的小伙子们, 你们就到西安上学吧。

马佩军是我在西电最早的朋友。刚入学时我们分到一个宿舍, 他像国民党兵盘问良家妇女那样上下打量我, 问: “喜欢干啥?”

我怯生生地回答：“打乒乓球。”

他再问：“什么风格。”

我答：“快球。”

他突然像阎锡山那样怪笑，拍拍我的肩膀说：“好！我喜欢，以后你就是我的朋友。我是陕西人，农民，会开拖拉机和卡车。这里是我家，以后你有啥事，就跟我讲一声。”

马佩军和我打乒乓球时口中念念有词：“哼！你对我狠，我对你更狠；你对我好，我对你更好。”他好几次说要把世上最好吃的板栗送给我一袋，这一袋板栗到现在我都没拿到。马佩军夜里极能侃，吹他家乡的人跑得快，常把野兔追断气。有时他吹得太离谱，常令我们 6 个舍友群起而攻之。为了把我们一举歼灭，他白天到图书馆查“资料”，夜里再挑起事端。我们双方就像印度与巴基斯坦，常干两个秃子争一把梳子的事。

马佩军上大学之前也没有见过计算机，但他对计算机技术有极强的领悟力。我们第一次上机时，他把我拉到打印机旁边说：“帮我防着管机房的，我要修理这台打印机”。还没等我反应过来，他就开始“肢解”打印机。我无比深刻地体会到：歹徒在作案时都不害怕，最提心吊胆的就是那个放风的。他在 5 分钟内修好了打印机，令我佩服得五体投地，甘愿下次再跟他干“坏事”。

我们读本科和硕士时主要用 DOS 操作系统，那时病毒泛滥。马佩军杀病毒不用软件，用手杀。看他杀病毒简直是一种享受：噼里啪啦地敲一阵子汇编命令，然后机器就好了。求救电话太多，他经常无怨无悔地带着那双铁手游荡于西电的各个角落，却不知道编写个杀病毒软件来赚钱。

我一直认为马佩军是西电编程第一高手，他编程的时候根本不是人，是指针。之所以我的名气大，一是因为他不好名利，二是我把他的程序写上了我的名字（并且卖了不少钱）。

马佩军的女朋友是我介绍的，我一眼就看出她将嫁给他。后来俩人果真结婚了，只是他嫁给了她，现在他还有了一只“小马驹”。硕士毕业后，马佩军留在西电读博士。前年我再见到他时，他说我害死他了，快乐得要宰了我。马佩军在西电已经呆了十年，秉承了西电所有的优点与缺点——“很土气但结实耐用”。我在西电时很土气，离开西电后变得“半土不洋”。马佩军简直就是西电人自己的“兵马俑”，每次看到他或者想起他时，我就明白自己的“根”还在西电。我喜欢陕西人源于马佩军。

宋任儒是我们的班长，也是班里最早的党员，满口仁义之道，比唐僧还让人受不了。在二年级时，我迷上一个比我大一岁有了男朋友的女同学，多日沮丧。他看在眼里痛在心里，跑去把那女同学教育了一通。苏联解体的时候他十分沉痛，在思想教育课上，他向我们作了深刻的检讨，好像是他没有管教好戈尔巴乔夫。最后他为我们点燃了希望：在不久的将来，“苏联”将重新成为苏联，共产主义旗帜将继续在全世界飘扬。

在本科三四年级，他对跳舞十分入迷，连上厕所都滑翔而去。我那时常把自己关在实验室里搞科研，极少有空与他玩乐，等到本科毕业时，猛地发现他已经风度翩翩了。

宋任儒在读本科时学习既不好也不差，我们从来没有合作研究过什么。我喜欢他是因为他很有情趣，不落俗套，并且刚正不阿。也许，我俩本来就有相似的秉性，只是表现形式不同而已。

本科毕业时，他分到威海工作，走之前我为他饯行。可在硕士开学时，我的房门被人一脚踢开，他对我喊了一声“林子啊，我又回来了”。我就像祥林嫂见到了被狼叼走的孩子那样惊喜。

宋任儒读硕士时被发配到临潼 771 所，他在那里过上了乐不思蜀的日子。有一天，他带来两个看上去很文静的女孩子（一个读硕士，一个读本科）来串门。就在我洗水果的几分钟里，三个人已玩得乐翻了天，两个女孩满屋子追他，一会儿把他按到桌子上打，一会按到床上打。我惊诧至极而又羡慕至极，恨不得挨打的人是我。想不到上学竟然会有这等欢乐，看来我读硕士的日子白过了。

后来，那个大一点的女孩子嫁给了他。当他带她去见公婆时，公公长叹一声：“把儿子交给你，我就放心了。”而婆婆已乐得合不拢嘴，竟然无法叹气。

现在，宋任儒已从复旦大学获得博士学位，比我更早地成家立业。他和她既是夫妻，又像兄妹，还像伙伴。他叫她“聪聪”，她叫他“笨笨”。

“聪聪”问“笨笨”：“老公啊，人活着为了什么？”

“笨笨”答：“就是让咱们每天快快乐乐。”

我亲眼看到的幸福莫过于此。

我在大二时曾为系里学习最差的十几名同学办了补习班，谢伟在这个补习班里倒数名列前茅。在他睡懒觉时，我像催命鬼那样喊他捅他。他无比吃力地抬起沉重的眼皮，就像软弱无力的举重运动员，还没有挺起来就趴下了。

他开始呻吟：“这一次就饶了我吧，下一次我一定，一定会去的，求你了。”

我不肯。

“那么让我再睡 5 分钟，”他不死心。

我仍不同意。

“那么你就从 1 数到 10，要慢一点，”他讨价还价。

当我数到 9 时，他就接着数 9.1, 9.2, 9.3 ……

一开始他觉得我很好玩，后来他就离不开我了。并不是因为我学习好，而是那时候我天真并且充满活力。在三四年级我忙于科研时，他照顾我的生活，叫我“少爷”，既做管家又做兄长。我们不仅共用饭菜票，并且共用仅有的一个碗，总是他买饭菜和洗碗。

我们那一级的学生大多崇拜巨人公司的创始人史玉柱，我问谢伟：“我是不是和史玉柱一样能吃苦？”

他说：“如果考虑年龄因素，你已经比他更能吃苦。你将来一定能做大事，我就把希望寄托于你了。”

在我们都还不成熟的时候，我成了他心中的灯塔，只要灯不灭，希望就在。现在他为了娶一个日本姑娘，披荆斩棘追到日本，有了新的希望。

二十几年来，我就为一个男人哭过，那时他本科毕业离校。

我读硕士研究生时，由于受我的影响，本系三四年级的学生蜂拥至微电子所参

加科研。夜里看十二层高的科技大楼，灯火通明、热闹非凡的那一层就是微电子所。那时，我在微电子所学生中的地位极高，手下兵将极多。

我写第一本书时，有好几个人帮我输入稿子，使我没时间慢腾腾地打草稿。我就像金庸写小说，有如神来之笔，想到哪里就写到哪里，写了一段他们马上输入一段，一气呵成。那本科技书写得很滑稽，同学们看得笑出眼泪，编辑看了拍案叫绝，只改了几个字就出版了。

那时候我的心情是如此之好，为一男同学乱蓬蓬的头发写了一篇散文，并送他一把梳子。又把一女同学的实验报告写成评书。我的文笔大概就是这样练出来的。

这一群学生中，戴玉宏、史江一和马晓华是我最好的伙伴（我们都属牛）。

戴玉宏其貌甚帅，眉中有一根白毛闪闪发光，因此号称“白眉鹰王”。“白眉鹰王”武功了得，是我软件产品的核心开发人员，我们合作最深最久。后来我开公司，他就从广州辞职到杭州为我助威，令我感动不已，可见读大学时期我们有多铁！戴玉宏有一次打饱嗝，整整打了两天两夜，我差点心疼死。

我尚未发迹前曾与戴玉宏在校园里卖花，无人问津，就请电子工程系的鲁洁救助。鲁洁温柔貌美，她一言一笑犹如春风吹拂苏堤的杨柳，令人心里一荡再荡。顷刻间就有男生围观，有人看花，有人看“贵妃”，鲁洁一走，我和戴玉宏可怜得就像两根蜡烛。鲁洁读大学时调皮捣蛋，到四年级时还不太会编程。她的本科毕业设计是仿真“雷达跟踪飞行物”，程序基本上全是我编写的。我已记不起用了什么公式，只知道每次计算后都弹出一个对话框“报告长官，击中目标”。鲁洁毕业后到深圳的一家软件公司工作，几年一过，她成了行家。再与她交谈时，我只有听的份，像鸡啄米一样点头。

史江一和马晓华都是陕西人，和马佩军一样厚道热情。史江一性格稳重，属于“你办事我放心”的那类人。我对微电子专业一窍不通，全靠他帮我混过实验这一关。后来我开公司失败，陷入经济危机，就把希望工程的一个小孩托给了他。

马晓华是我最不放心的人。他常常为别人做事情，但热情过头就忘了自己的事情。有几个不道德的学生就利用他的这个缺点，经常使唤他，并且借他的钱不还。马晓华喜欢为那些人“卖命”并且挨训，他总是在受虐待够了的时候再跟我们嘀咕，我们实在气不过，只好对着他的屁股追加一顿拳脚，并给他一个绰号“受虐狂”。

我们这一群小伙子同时喜欢上一个女孩子，她叫姜姗，是她班里的四大美人之一。我们不仅没有争风吃醋，而且心甘情愿地让她坐遍每个人的脖子。姜姗小姐 5 岁时，她爸姜晓鸿成了我的同事，我们经常一起去钓鱼，亲得像一家人。姜姗喜欢大喊大叫，声音高过帕瓦罗蒂，我们教唆她喊她爸“姜球球”。

我常带姜姗到小吃摊去吃女孩子不敢吃的东西，并哄她：“世上最好吃的东西是鸡屁股。”

她无师自通地加上一句：“世上最好听的屁是鸡放的屁。”

我常想着将来生个儿子并把他培养成天才，但如果能有姜姗这样的女儿，不要儿子也罢。

在本科三年级我第一次参加“星火杯”竞赛并获得软件二等奖后，马上成为低年级学生眼中的明星。我义务当上了一年级学生的上机辅导员。一天晚上我巡视机

房，一女生请求帮助。

我见屏幕上空白一片，根本没有一程序，十分疑惑地问：“什么问题？”

“没有问题。”她把书往我手上一塞说：“这些作业你帮我做。”然后就自个儿跟她的同学玩乐，把我撇在一边，似乎我辛辛苦苦地学习就是为了给她做作业。

我定神对她细看，发觉她简直就是《射雕英雄传》里的黄蓉再世，顿时心就“突突突”直跳。当天晚上我没睡着，接下来几天的课也不知所云。在选修课《操作系统》考试时，我给家里写了一封超短信：请快寄钱来，我谈恋爱了。我交了白卷直奔她去。

我的初恋只有两个月，却让我思念了8年。她离我而去时没有任何理由，而我却失魂落魄。在我本科毕业前的18个月里，白天我狂热工作强作笑容，夜深人静时心痛如刀割而无法抑制。没有人为我“疗伤”，我是硬挺过去的，这一段经历使我日后心理承受能力极强。后来我开公司的失败虽然对信心有所打击，但与失恋相比根本无痛可言。

我们分手后并未成为陌生人，就像两只刺猬，离得远了就有点留恋，离得近了，就刺着对方。认识她时我虽然已略显才气，但并不具备成熟男人的魅力，很多事情我并不知道怎么去把握。有时“喜欢”并不能成为“爱”，感情也许是永远研究不透的学问。

我读硕士研究生时有了一群生机勃勃的朋友，感情的伤痛被淹没了。朋友堆里夹着一位女生，她文雅而富有气质。平日里无拘无束，大伙戏称她是我的秘书。我的言行举止和穿着都经过她的调教，俩人出双入对，十分亲密，不知不觉有了感情。别人已经把我们当成恋人，我和她牵着小姜姗散步时，简直就像一家人。

可是我当时沉迷于事业，认为自己不久将干出一番惊天动地的事。鉴于史玉柱在创业时就离过婚，所以我认为感情是事业的累赘，两者不可兼得。

更糟糕的是，我和第一个女朋友藕断丝连，偏偏她俩是同班同学。我知道脚踏两只船没有好下场，可我的的确确同时喜欢着两个人，并梦想她俩能合二为一。我情愿被人指责，也不愿掩饰真实的感觉。有时她俩一同走过，我站在路上丢了魂似地看着俩人的背影，任凭看热闹的人指指点点。

我和第二个女朋友已经有了很深的感情，她毕业后我曾坐火车千里送鲜花给她，让她感动过。而我固执的性格和对初恋的思念终于让她心碎。尽管我们已经几年没见，我依然看得见她留在我心里的那颗眼泪。

我在西电六年半的学习和生活也许是一生中最珍贵的，叫我怎能不爱西电。

两年前我回西电，惊奇地发现校园里房前屋后长满了待收割的小麦！这所大学是从事电子科技的，种小麦干啥呀？

老同学告诉我，种小麦是为了应付“211”工程（为21世纪选拔100所重点大学）的检查团，因为“211”工程有较高的绿化指标。偏偏检查赶在冬天，那时的西北极难长草。西电本来就人多地少，地上一长草马上就会被谈恋爱的学生给折磨死。一到冬天，整个校园就光秃秃一片。小麦在年轻的时候还真和青草长得一个模样，用小麦绿化校园可谓千古绝笔。

浙江大学依山而傍西湖，是一所美丽而高贵的大学。1997年春天，我就像干儿

子那样挤进她的怀抱，并渴望得到关爱。我到了向往已久的计算机辅助设计与图形学（CAD&CG）国家重点实验室读博士学位。导师是石教英教授，石老师虽然年过六旬，但精力充沛，红光满面，施拉普纳不及他半分精神。

我幸福地幻想着大干一番自己喜爱的事业，并计划在 35 岁左右成为实验室主任。开学的第一天，我兴冲冲地奔向实验室。进门不到 5 分钟，就因不懂规矩被看门的年轻女子训了几次。为了不再冒犯规矩，我就老老实实在地抓起一份计算机报纸并且站着阅读，心想这下不得罪谁了吧！

突然一个气得脸色铁青的男人（机房管理员之一）对我断喝：“你在干什么！你怎么可以不经允许就翻看别人的报纸！”似乎我是他一生中见到的最无耻的人。

我就像一个情窦初开的少年飘飘然地去拥抱梦中情人，不料迎来两个耳光，此下场比《猫和老鼠》中的猫还惨。如果这两个年轻人有幸看到我这篇文章，应该好好悔过自新，她与他的工作态度打击过数十个学生的积极性。我本是因为向往 CAD&CG 实验室而来的，得到的却是极坏的第一印象。（我博士毕业后，这两人也离开了实验室，我替后来的学生们谢天谢地）

CAD&CG 实验室在理论研究方面很有名气，但我的兴趣是开发实用的软件，“嫁错人”了。我颇费周折地考入 CAD&CG 实验室，却尚未热身就全身而退，决心自立门户。至今我都没有用实验室的计算机编过一程序。

刚读博士时我穷困潦倒，只有一床，一盆，一壶，一碗。我那些穷朋友们像挤牙膏一样挤一些钱资助我。我买了一台计算机，在宿舍里开发软件产品“可视化软件开发工具 VA 4.0”。1997 年 8 月，我去北京参加首届中国大学生电脑大赛软件展示，路费也是借的。同学为我壮胆时说：“如果不能获奖，就回到实验室干活吧。”

我说一定会拿第一名，不然去干啥。

在软件展示时，我们发现很多好的作品是国家的科研项目，根本不是学生个人的作品，这就违背了竞赛的宗旨。如果允许这样做的话，学校可以运几条生产线过来。我写了一份抗议书，找了十几个人签名（很多人敢恨而不敢签）。但抗议能顶屁用，我参加过的科技竞赛、听过见过的科研鉴定多了，哪一次我没看到虚假？我写抗议书是因为眼里容不得有沙子。

这次竞赛选出十个“软件明星”，只有我的软件和清华大学某位博士生的项目值得一看。他的项目水平很高，但那不是他个人的作品（评委甚至认识他的导师，知道项目的来龙去脉）。综合诸多因素，我的作品被评为第一，他的项目被评为第二。组委会来拍电视，可是找不到浙江大学的展板。因为浙江大学没有任何准备，我是一个人来的，我的作品夹在杭电的作品之中，没名没姓。我只好从塑料袋上剪下“浙江大学”四个字，贴在展板上撑撑门面。

自新中国成立以来，清华大学就一直自视“龙头老大”，在其他大学头上“作威作福”，我这次好歹也争了一口气。可是颁奖时，组委会竟按地方顺序从北京念起，我沦落到第七，差点让我咽气。

我曾在上海的一辆公共汽车上与一位北京来的旅客聊天，此公极健谈。似乎他到上海旅游的目的就是为了发掘北京的优越性。见我挂着浙江大学的红色校徽，且对清华、北大并不神往，不禁十分迷惑，就问：“浙江大学在浦东还是浦西？我要去

看看。”

1997年11月,在穷得快挨饿的时候,我获得了中国大学生跨世纪发展基金特等奖(全国共20名,奖金1万元),到人民大会堂领奖。给我们出钱的是一个靠资本运作发财的集团。在宴会前,该集团领导人和我们座谈,他什么不好吹,偏偏吹自己是个高科技企业:“我们主要从事生物工程,几年前就掌握了克隆技术,英国的‘克隆羊’简直是小菜一碟……我们在东北有个农场,新品种的小麦长得比人还高,麦粒跟葡萄一样大,你们不久都会喝到用这种小麦酿的啤酒……我如果去美国炒个总统,那就跟玩儿似的。”

我们几个获奖的博士生吃饱喝足、拿了钱后,关起门来把那个老板臭骂一通,扬长而去。

刚拿了“跨世纪发展基金”,又马上获得“浙江省青少年英才奖”,浙江大学也给我发奖学金。比起那些一个月只有300元工资的博士生们,我简直是“暴富”。还了朋友们给我的“救济款”后,仍然是个“富翁”。我老是觉得手里的钱是“抢劫”来的,心里不踏实。于是找浙江大学校团委“诉苦”,请校团委把我的“不义之财”捐给浙大的贫困学生。校团委的老师热情而坦诚,说愿意等我成为真正的富翁时再接受捐款,现在不能“杀鸡取卵”。但为了能让我表达心意,建议我资助“希望工程”的中学生,让我选了5个初一的学生,每个学生500元。我轻浮地以为自己真的帮助了5个中学生,直到1998年暑假我见到了其中的一个中学生,才发现自己做的好事只不过杯水车薪而已。我是到了自己贫困失意时才真正去帮助那些孩子的。

在1997年,我在学生时期的荣誉已经登峰造极,觉得自己的翅膀已经硬了,不想再混下去。我总以为自己是第二个史玉柱,应该开个软件公司来振兴民族软件产业。我曾到东软集团(沈阳)参加“民族软件产业青年论坛”,大不咧咧地作了一次演讲(现在发现演讲的内容没有一项是可以操作的)。杭州有一个记者来采访我,我谈了一天的理想,记者还是没听明白,干脆自己写新闻报道,并且含蓄地做了一个广告:万事俱备,只待投资。

由于我能说会道,频频上电视,引来近10个投资者。我选择了一位年龄比我大一倍、非常精明的商人作合伙人,成立了“杭州临境软件开发有限公司”。彼时,我可谓光芒四射,名片上印着“以振兴民族软件产业为己任,做真实、正直、优秀的科技人员。”浙江大学有关部门想开除我,被我“晓之以理、动之以情”安抚住。

我当时想开发一套名为Soft3D的图形系统,此系统下至开发工具,上至应用软件,无所不包。公司名字起为“临境”有两个含义:一是表示身临其境,这是我对图形技术的追求;二是表示快到了与SGI公司称兄道弟的境界,这是我对事业的追求。“临境”这个名字我在读本科时就已经想好了,1997年底公司成立的那一天,我有一种“多年的媳妇熬成婆”的悲壮感觉。

我从实验室挖来一位聪明绝顶的硕士生做技术伙伴。他叫周昆,年龄很小(1978年出生),研究能力极强。如果按照浙江大学计算机系博士生毕业的论文要求,他入学读硕士研究生的那一天就可以博士研究生毕业。周昆的头明显比我的大,估计其脑容量至少是我的1.5倍。我曾经以师兄的身份为他洗过一双袜子,他因此觉得我是个好人。我俩一拍即合,常常为Soft3D的设计方案自我倾倒。一想到Microsoft公司



的二维 Windows 系统即将被 Soft3D 打击得狼狈不堪时，我们就乐不可支，冲劲十足。

我已经把“振兴民族软件产业”列入日程，并且提前担忧将来钱挣得太多用不完该怎么办。1998 年 5 月份，我们做了一套既不是科研又不全像商品的软件。软件产品宣传了几个月，并没有出现订单如潮、应接不暇的局面（事实上压根就没有反应）。我意识到没有找对市场，但仍觉得产品中的一些技术很有价值，将它改装成其他软件也许能开创“东方不亮西方亮”的新局面。

于是我向只有一面之缘尚在北大方正工作的周鸿祎求助。他是真正的软件高手，当我小心翼翼地展示约 10 万行 C++ 代码的软件时，他竟在十几分钟内就指出多处重大的设计错误，使我目瞪口呆地意识到整个软件系统的价值为零。那种心痛啊，就像眼睁睁看着孩子被狼吃掉一样。

到 1998 年 10 月，我用光了 30 万元资金。周鸿祎再一次从北京飞到杭州，三下五除二替我把只活了一年的公司关闭掉。他放心不下，觉得我“恶病需用猛药治”，于是意犹未尽地把我捉到北大方正插在他管辖的部门，让我学习怎样做事情。

北京寒冷的冬天可以营造一种凄凉的气氛，冲去一切可以自我原谅的借口。我并不是太爱虚荣的人，知道这次失败是我的毛病积累到一定水准忍不住喷发出来的结果。我绝不能以年纪尚轻不太懂市场与管理为由轻率地敷衍过去。

从北大方正“劳改”了两个月回来，我心服口服地承认失败了。我把察觉到的数十个毛病列出来，日后一个一个克服掉。现在我能比较清醒地分析我和投资方所犯的主要错误，以祭我那幼年夭折的软件公司。

我的主要错误：

（1）年轻气盛，在不具备条件的情况下，想一下子做成石破天惊的事。我的设计方案技术难度很大（有一些是热门的研究课题），只有 30 万元资金的小公司根本没有财力与技术力量去做这种事。

（2）我以技术为中心而没有以市场为中心去做产品，以为自己喜欢的软件别人也一定喜欢。我涉足的是在国内尚不成气候的市场，我无法估计这市场有多大，人们到底要什么。伙伴们跟着我瞎忙乎一整年，结果做出一个洋洋洒洒没人要的软件。

（3）我做到了“真实、正直”，但并没有达到优秀的程度。我曾得到很多炫目的荣誉，但学生时代的荣誉只是一种鼓励，并不是对我才能和事业的认可。正因为我不够优秀，学识浅薄，加上没有更高水平的人指点我，才会把事情搞砸了。

投资方的主要错误：

（1）投资者是一个精明的商人，他把我的设计方案交给美国的一个软件公司分析，结论是否定的。但他觉得我这个人很有利用价值，希望可以做成其他事情，即使 Soft3D 软件做不成功，只要挣到钱就行。这种心态使得正确的可行性分析变得毫无价值。

（2）由于我不懂商业，又像所有单纯的学生那样容易相信别人。他让我向他借钱买下本来就属于我的 30% 技术股份，写下了不公正的合同（借条）。他名为投资方，实质上双方各出了一半的资金（他出 51%，我出 49%）。他在明知 Soft3D 软件不能成功的情况下，却为了占我的便宜而丧失了应有的精明，最终导致双方都损失。

关闭公司时，他搬走了所有东西。我明明投入了技术，又亏了 15 万元，却一无

所得。几个月后当我意识到不公平而找他协商时，他说：“只能怨你自己愚蠢，读到博士，连张合同都看不懂。”此事充分地显示了我的无知与愚蠢。自己的奋斗没有必要后悔太多，学到的远比失去的多，我相信下一次会做得更好。

公司关闭后，我就面壁反省，补习基础，准备为几年之后“东山再起”养精蓄锐。

1999年1月，有一位民营企业家G先生问我一个问题：“我给一个年轻人投资了100万元，建立一家从事环保信息应用开发的软件公司。他曾许诺一年内创利润上千万元，可是才过去5个月，他就把100万元用完了，什么也没挣到。我实在不明白是怎么回事，请你帮我分析分析。”

这位G先生年龄是我的2.5倍，曾在西北当过几十年的技术兵，性格豪爽。他投资的那个年轻人叫Y（以下称Y经理），自称有英国的管理学文凭，能对公司的市场、技术、管理一把抓。G先生喜欢说“钱没问题”，于是想也不想就投了100万元，并且给Y经理40%的股份。

G先生请Y经理到家里座谈。我那时突然狡猾起来，自称是G先生的远房亲戚，在浙大读半导体物理，特羡慕那些做软件的同龄人，渴望听听Y经理的高见。Y经理果然信口开河，滔滔不绝，连绵不断，如黄河泛滥，一发而不可收拾。我激动地想去参观他的公司和产品，并表示要抛弃物理专业，立马转向软件专业。

Y经理得意地笑：“对于IT行业你就不懂了，我们经营的是一种理念而不是产品，这是国外最先进的思想。你可以来参观我的公司，但你看不到具体的东西，只能用心去领会。”

这话比《围城》里那个曹元朗的诗还臭。我搞软件只有8年功夫，说我不懂IT行业并不过分。可我读了10年大学都没听到过如此“先进”的思想。如果这是英国管理学教育的成果，我认为自己已经发现了这个曾经是“日不落帝国”的衰败的真正原因，有必要找英国首相切磋一番。

我对G先生说：“Y经理根本不懂技术，为人极其浮夸。应马上关闭公司，以绝后患。那100万元你也亏得起，就买个教训吧。”

G先生说：“钱我没问题，那100万元就当我在澳门赌博输掉了。”

1999年5月，G先生又来找我请教另一个问题。

他说：“小林啊，你上次说得很有道理，我接受了教训。”

我说：“那是好事，不论年龄大小，知错就改才是好孩子嘛。”

他叹了一口气：“最近几个月，Y经理又花了我100万元。”

我当时差点被噎死，气势汹汹地训G先生：“我早跟你讲过，Y经理不是好东西，叫你关闭公司你不听，你老说钱没问题，亏你200万元活该。”

老先生像犯了错误的小孩子：“Y经理每一次向我要钱时，都拍拍胸脯保证下个月就有利润，所以我一而再、再而三地掏钱给他，希望能救活软件公司。现在该怎么办？”

一个有20名职员的公司，程序员只有三四个，连“十羊九牧”都不如。200万元的财务报表中，有100多万元用于吃喝玩乐和行贿。这种公司完全无药可救。台湾作家李敖曾说过：“当你没法扶一个人上马时，也许应该拉他下马”。从5月份

到 8 月份，我行侠仗义，替 G 先生清理软件公司，根除 Y 经理这些败类。

可是难哪，因为 G 先生投资的公司根本不把 G 先生放在眼里，又岂能让我插手。就在我想方设法卡住 Y 经理的脖子时，Y 经理总能从 G 先生那里挖出钱。G 先生就像被吸血鬼附身，却仍存幻想：“如果吸血鬼能治好我的病，就让它再吸些血吧。”

Y 经理又和一个来自深圳的骗子 H 想了主意，教唆 G 先生再投资 100 万元新建一个“指纹”公司，说利润将比开发环保信息更加可观（估计要用亿来度量）。就在他们准备签合同之际，我偶尔路过，发现异常，便强行阻止。

G 先生是个好人，但太顽固。好几次我气极想撒手不管，但又不忍心好人被坏人欺负。我曾请求 G 先生：“我求您别再说钱没有问题，您的私人财产会被人骗光。请让我把这漏洞堵住吧，好让我安心地回学校做完博士学位论文。”

到 8 月份，G 先生的两个儿子和我，伙同一帮五大三粗的朋友，强行把那个软件公司搬回 G 先生的工厂中，辞退所有员工。现在那个软件公司被别人接管，仍然半死不活，好在每月亏损不过几万元，G 先生承受得起，我就不再去碰 G 先生的伤疤。

我以前从未玩过与人勾心斗角的游戏，此三个月的经历让我疲惫不堪。那个软件公司的员工曾透露，Y 经理的英国文凭大约是在上海或杭州某个大专培训班里混来的。《围城》里方鸿渐买美国克莱顿大学博士文凭尚知羞耻，而 Y 经理却趾高气扬。害得我平白无故为英国教育界担心。

G 先生是正人君子，不防小人，实在不算是现代精明的商人。我和他成了忘年交。G 先生第一次见到我时问我工资几何，我答曰：“300 元，够买几本书。”G 先生甚为着急：“这样的条件怎么能生活？你就搬到我家来住吧，我家条件好，你可以安心地学习，将来可为国家多作贡献。”后来他几次相邀，我就看在国家的份上入住他家，一直住到博士毕业。自从读中学以来，我第一次享受食来张口，不用洗衣服的奢侈。唯一的麻烦是我得向很多朋友解释：“我不是被别人养起来了，是为了国家的利益，不得已才这么做的。G 先生是男的不是女的，并且没有待出嫁的女儿。”

我在读博士学位的三年半里，经历有点奇特。我遗憾的是“真才实学”没有长进多少，并且没有了在西电那样的纯真友情。略为欣慰的是我做了几件有意义的事情。我很想讲一讲自己参加希望工程的经历与感受。

1998 年暑假，浙江省云和县梅源中学的老师们带着希望班几名优秀学生来到浙江大学，其中有受我资助的何晓丽同学。我才知道初中学生一个学期的学杂费就要 600 元。何晓丽哭诉下学期不能再上学，其他的学生处境相似。我以前资助的 2000 元是 5 个人 3 年平均分配的，根本不起作用。

那时候，公司倒闭使我债务累累，并且自信心遭受十年来最大的打击。我在入不敷出、心事重重的情况下，没有推卸责任，而是“变本加厉”地去尽这个义务。我在西电的好朋友史江一替我“接管”了一个中学生。有一个小姐追求我，我乘机给了她一个活生生的“见面礼”。1999 年 7 月份，我把饭卡送给了一个大学生，自己成了无产阶级。从 1997 年 11 月起到我博士研究生毕业期间，我直接或间接地为 7 个贫困学生捐助了约 1 万元。我有了几点感受：

(1) 对人的帮助莫过于给予希望。

(2) 人在任何时候都能够帮助比自己更困难的人，哪怕自己处于困境。

(3) 帮助是要负责任的，一定要设法做成有意义的结果。不负责任的帮助就是“施舍”。“施舍”缺乏诚意，不配称为“帮助”。

不少人曾对我说：“你是做大事的人，不要在小事上浪费精力，更不要为了别人而贻误了自己。”

很多人总以为自己将来是伟大人物而不愿做小事，从而到死也没做成什么有价值的事。也有很多人希望自己成功后再去帮助别人，无论他最终成功还是失败，一辈子也没有帮助过人。还有很多人略有权势或略有名气后，便觉得自己吃喝玩乐、放屁、上厕所都是重要的事，在他们最能够帮助人的时候却以“太忙”或“没空”为理由而拒绝。

我也在忙碌、在奋斗，也渴望成为伟大人物，但我希望让有意义的小事充实一生。

我还要讲另一件我常干的小事。

很多受过高等教育的人保留了随地扔垃圾的习惯，这恶习就像脚气那样虽然不置人于死地，但能遗臭万年。即便像浙江大学这等风雅的地方，你都经常可以看见草坪、校门口的废纸、果皮和塑料袋等，垃圾就如同天使脸上的一坨狗屎那样醒目，人们竟然无动于衷。我记不清自己多少次当众、当道捡垃圾，可是几年来我都没有在大学里发现第二个做这种事的人。

我很想对所有的教授、博士、硕士、学士们讲句话：“救人并不只是医生干的事，保护环境也不只是清洁工干的事。只要你多花几秒钟，弯几次腰，就能让环境更加清洁，让心灵更加清洁。我们不必个个道貌岸然，但至少应该做到‘读书明理’。”

我这样喋喋不休地讲“希望工程”和“捡垃圾”，并不是在沽名钓誉，也不是在布道，只是希望我这些“金玉良言”能触动更多的自以为是高素质的人们。

在浙大的三年多时间里，我没有对感情“播种”，所以也没有收获，但有一次“艳遇”。

在关闭公司的那天晚上，人去楼空，我像严监生断气前那样盯着尚未熄灭的灯。这时某大学的一位四年级女生来找我。一年前她曾作为实习记者采访过我，谈得很投机。我知道她是聪明好学的学生，曾大言不惭地教导过我几次。我开公司的一年里几乎没与她来往过，想不到当我成了光杆司令时她还能“兔死狐悲”地来看望我，着实让我感动。

我不无自嘲地对她说：“你不用安慰我了，这次失败我还能挺得住。”

她说：“我不是来安慰你的。我一直盼望你的公司倒闭，等了整整一年。在你去北京之前，我有话跟你说。”

我心下一凉，搞不清什么地方得罪她了，让她如此记恨我。大概是我得意之日教导她时言语过重，伤了她的自尊心。好在我是知错就改的人，当下惭愧地向她道歉。

她不理睬我，说：“你开公司时光环重重，我根本无法靠近你。即便那时我成了你的好朋友，你也不会把我放在眼里。我暗恋你一年了，一直都没跟你讲。我早知道你会失败的，失败时你就剩下一个人，你才会知道我是真心爱你的，而不是冲着

其他来的。你是个优秀的理工科学生，我是个优秀的文科学生，门当户对，珠联璧合。请你不要觉得女的追男的很荒唐，我是认真的，请你给我一次机会。”

我虽然评不上情场高手，好歹也在爱河里游过泳，呛过水。想不到仓促之下，被一女子说得脸红耳赤，无法掩盖窘相。

我一直认为男人应该勤劳一辈子，好让柔弱的女子舒舒服服地在大树下乘凉。而学习、工作出色的女子只能做朋友，不能做夫人。

她从小习诗弄文，读大学时蜚声校园。我见到她第一面时就把她归类为事业上的朋友，所以才会正儿八经地与她交谈并教导她。我在西电的两个女朋友就属于读书不太好但比较有魅力的女生，我从来也没有指导过她们学习。如果我喜欢一个女孩子并希望她成为我的女朋友，我早就去追求她了，岂能轮到她追我。

她见我彷徨不安，便滔滔不绝地列举爱我的“证据”。我开公司一年来发生的事她了如指掌，就像在我的房间里放了窃听器，在我的朋友中安插了间谍。她甚至趁着实习机会跑到团中央去查阅我的老底，有些“光辉记录”我过去的伙伴都未必知道。她思念我时，写了很多诗，流了很多泪……

我早知道有些人不编程、不做实验就能写出论文，难道男女之间不接触也能滋生感情？

第一回合我就被她挑翻在地，我莫名其妙地成了“负心郎”，无地自容地把她送走。我以为这是文科女生的风格，就当做一件趣事不放在心上。

我从北大方正“劳改”回来不久后，她提着一篮鲜花来找我，并对我说了她的梦想：在寒冷的冬天，大地铺满积雪，四野人鸟绝迹。我孤独地深居在冷冰冰的小木屋里。在一个狂风呼啸的黄昏，她一手拎着亲手做的饭菜（我想应该有土豆和辣椒），一手拎着一捆木柴，敲开了那扇紧闭的门和心房。终于木屋四壁生辉……

我曾对第一个朋友最好的赞美是：“黄蓉很像你。”

我曾对第二个朋友最深情的话是：“将来咱们老了，我回黄岩当物理老师，你当语文老师。”

相比之下，我的确不及她浪漫。此后她再找过我几次，当我意识到她动真格的时候，她已不能自拔。爱情是很怪的东西，并不是两个好人在一起就能碰出火花。与其让她长痛，还不如让她狠痛一次。

我对她说：“我们真的不能在一起。”

她问为什么？

我说：“不为什么，我没有心跳的感觉。”

她说十年之后再找我。

我知道她会奋发图强，因为她会一直想着“为什么”，期望让自己有个满意的答案。这条路 8 年前我已经走过了。后来她读硕士时我曾再见过她，她在文学上已经有了长足的进步。

她说将会送给我她的第一本著作，书中开头的几个故事是关于我和她。

我说看了她的书后一定会写一篇读后感给她。

她仍然提醒我不要忘记十年后的相约。

我在浙大有一个值得怀念的人，她是管宿舍楼的大妈。在 1999 年 1 月至 5 月，

我在博士生宿舍静心修炼内功，大妈就像我的“护法”。晚上九点钟时，她就会烧些东西给我吃。我和大妈非亲非故，同学们都不明白大妈为什么待我好。我想那是因为我没把自己当成“博士”来看，而是当成“人”来看。

5月份后，我看在国家的份上搬到千万富翁G先生家里去住，大妈也调到“熊猫馆”当掌门人。我一般隔几个月去看望大妈一次，中秋节我就和她在一起。朋友们知道我和大妈有这层关系，就纷纷托大妈物色女朋友。

大妈果然称职，她就像特务那样审视大楼里的女生。可大妈毕竟是大妈，她采用的“标准”是几十年前的版本，无法与现今的兼容。她盯住了不该盯的，却漏掉了不该漏的，至今都未“推销”成功一个。

这件事让我又明白了关于软件的一个道理：光有完善的数据库还不够，还应该提供很好的搜索引擎。

我相信生活、科学、艺术中的很多道理是相通的，于是就不嫌人笑，写下了十年来的故事，交最后一次作业。

大学十年给我留下了很多美好的回忆，现在可以打上漂亮的句号了。尽管我即将告别大学，但我会终生学习。也许我成不了天才，但还有机会成为天才的爸爸。

我想大声呼喊出那种可以用双手把握未来的自豪。

我要对年轻的朋友们说两句肺腑之言：

主动去创造环境，否则你无法设计人生。

生活和工作要充满激情，否则你无法体会到淋漓尽致的欢乐与痛苦。

如果我碰到上帝，只会对他说一句话：“你看厕所去吧。”



2000年3月于浙江大学

## 附录 D 《大学十年》后记

2000年7月份，我从浙江大学博士毕业后到上海贝尔有限公司工作。从2000年8月至2001年底，大约一年半时间，我在网络应用事业部从事软件工程和CMM/CMMI的研究与推广工作。从2002年初至今，我调到公司总部从事企业研发管理的研究。

我在上海贝尔没有发财，没有当官，那么工作近两年来我都在做些什么呢？

我在心平气和、踏踏实实地做学问。

我读本科的专业是半导体物理，硕士专业是集成电路，博士专业是计算机图形学。十年之内我换了三个专业，哪一个专业都没有学精通。我觉得自己在软件工程方面有些悟性，可是没有当成专业来系统地学习。所以博士毕业时，除了有点虚名外，我的确没有什么过人的才能。

如今大学里的博士、教授中“水货不少”，我不幸是其中之一。像我这样的好苗子沦落为“水货”，是中国研究生教育走向腐败、堕落的恶果。

在公司里，很多员工恭谨地叫我“林博士”，甚至还有年轻人特意来看看我这个“好榜样”。虽然我爱慕虚荣，可是良心未泯，彼时我哪有博士的真才实学，多么羞耻啊！

刚到公司时，我有两类工作可选择：一是开发产品，二是研究并解决企业存在的软件工程问题。

我在开发产品方面比绝大部分应届毕业生和员工们有经验，当项目经理可谓熟能生巧。软件工程则是我的研究兴趣。前者可能会有更高的经济收入，后者能提高自己的学问。

当时我的想法很简单：趁着自己还年轻，赶紧好好做学问，弥补读博士期间浪费的三年半青春，让自己有真才实学。

在企业里做学问与高校里很不一样。如今学校里有不少学者越来越浮躁，他们的聪明才智大多建立在“纸上谈兵”上，做学问变成了造文章。而企业特别讲究“务实”，所有的工作围绕一个目标：努力让企业活下去并且活得更好。

我在公司里没有像在学校里那么“兴风作浪”，日子过得很简单，不停地调研、写规范和培训，就像少林寺藏经阁里的修炼者。上海贝尔提供了一流的软件工程研究与实践环境（不是现成的，是我争取来的，这就是我信奉的“创造环境”）。在这种环境下，只要人不笨，认真工作，谁都能成为软件工程专家。

在博士毕业前，我写了一本薄薄的书叫《软件工程思想》。这本书与《大学十年》一样在网上流传，我曾经自鸣得意。最近我要出版一些研究成果，便重新阅读了那本《软件工程思想》。发现此书真的是彻底的“纸上谈兵”，不仅对企业毫无用处，并且会误导读者。

我真是悲喜交加，悲的是和我同类的一大批“假博士”长期干些“自欺欺人”的研究工作，浪费生命并且浪费国家资源；喜的是我终于跳出了虚假学术的火坑。我不敢说现在的水平有多高，但至少能够拿出一些对企业有价值的东西来。

我的目标是创作出可以与 Rational 公司 RUP 相媲美的软件过程规范，并且开发出物美价廉的适合于中国 IT 企业的项目管理软件。

基于大量实践的基础上，我和合作者研制了一套通用的“IT 企业软件过程改进解决方案”（Software Process Improvement Solution, SPIS），请访问网站 <http://www.chinaspis.com> 获取更多的信息。

SPIS 的主要组成部分有：

- ◇ 基于 CMMI 3 级的软件过程改进方法与规范，称为精简并行过程（Simplified Parallel Process, SPP）。
- ◇ 一系列培训教材，包括软件工程、项目管理、高质量程序设计等。
- ◇ 基于 Web 的项目管理工具，包括项目规划、项目监控、质量管理、配置管理、需求管理等功能，命名为 Future。

以我们研究小组的实力本来是可以顺利完成 SPIS 的。但遗憾的是，SPIS 不是公司产品发展战略范畴之内的东西，不能作为正式项目开发，我的组员们都被分配了其他的任务。我只能在公司之外组织一些朋友共同开发 SPIS。虽然困难很大，但是工作很有意义，很鼓舞人。

我依然相信“创造性的事业要靠激情来推动”。希望在今后的岁月里，我能结识不少志同道合的朋友，共同把事业做好。



2002 年 4 月于上海贝尔有限公司



# 附录 E 术语与缩写解释

英文术语与缩写	解 释
ADT/ UDT	Abstract Data Type/User Defined Types 抽象数据类型/用户定义类型
Adapter	适配器
COM	Component Object Model, 组件对象模型
Container	容器
CORBA	Common Object Request Broker Architecture 公共对象请求代理体系结构
Dynamic Binding	动态绑定或运行时绑定
Encapsulation	封装
Generic Programming	泛型编程
GUI	Graphic User Interface, 图形用户界面
Implementation	实现
inline	内联
Interface	接口
Iterator	迭代器 (迭代子)
Instantiation	实例化
Name-Mangling	名字修饰或重命名
OO	Object Oriented, 面向对象
OOAD	Object Oriented Analysis and Design, 面向对象分析和设计
Overload	重载
Override	覆盖或重写
POD	Plain Old Data type, 即具有平凡构造函数、平凡赋值函数和平凡析 构函数的数据类型
Polymorphism	多态
RTTI	Run-time Type Identification, 运行时类型识别或运行时类型信息
Signature	指函数的原型或经过名字修饰后的内部名称
Specialization	特化
STL	Standard Template Library
Template	模板
trivial	平凡的
UML	Uniform Modeling Language, 统一建模语言
vptr	虚函数表指针
vtable	虚函数表



# 参 考 文 献

- [Cline] Marshall P. Cline and Greg A. Lomow. C++ FAQs, Addison-Wesley, 1995
- [Cusumano] Michael A. Cusumano and Richard W. Selby. Microsoft Secrets (程化 等译). 北京大学出版社, 1996
- [Deitel] H.M.Deitel and P.J. Deitel. C++程序设计教程(薛万鹏等译). 机械工业出版社, 2000
- [Don] Don Box. Essential COM. 2001
- [潘爱民] COM 原理与应用. 清华大学出版社, 2001
- [Eckel] Bruce Eckel. Thinking in C++(C++ 编程思想. 刘宗田等译). 机械工业出版社, 2000
- [James] Geoffery James. The Tao of Programming (编程之道. 郭海, 郭涛译). 清华大学出版社, 1999
- [Maguire93] Steve Maguire. Writing Clean Code (编程精粹. 姜静波等译). 电子工业出版社, 1993
- [Maguire94] Steve Maguire. Debugging the Development Process(微软研发致胜策略. 苏斐然译). 机械工业出版社, 2000
- [Meyers-1] Scott Meyers. More Effective C++. Addison-Wesley, 1992
- [Meyers-2] Scott Meyers. Effective STL. Addison-Wesley, 2001
- [Murry] Robert B. Murry. C++ Strategies and Tactics. Addison-Wesley, 1993
- [Summit] Steve Summit. C Programming FAQs. Addison-Wesley, 1996
- [Lippman-1] Stanley B.Lippman. Inside The C++ Object Model (侯捷译). 华中科技大学出版社, 2001
- [Lippman-2] Stanley B.Lippman. Essential C++ (侯捷译). 华中科技大学出版社, 2001
- [Stroustrup] Bjarne Stroustrup. The Design and Evolution of C++ (裘宗燕译). 机械工业出版社, 2001
- [Josuttis] Nicolai M.Josuttis. The C++ Standard Library. 1999
- [Alexandrescu] Andrei Alexandrescu. Modern C++ Design. 2001
- [Vandevoorde] David Vandevoorde & Nicolai M. Josuttis. C++ Templates. 2004
- [Hughes] Cameron & Tracey Hughes. Mastering the Standard C++ Classes (健莲科技译). 人民邮电出版社, 2000

[ G e n e r a l    I n  
f o r m a t i o n ]

S S 号 = 1 1 8 6 3 5 9  
5

书名 = 高质量程序设计指  
南    C + + / C 语言    第  
三版

作者 = 林锐

页数 = 3 9 7

I S B N = 9 7 8 - 7 -

1 2 1 - 0 4 1 1 4 - 3

出版日期 = 2 0 0 7 年 5

月第 1 版 | 出版社 = 电子

工业出版社

下载地址 =

报文 =

封面

书名

版权

前言

目录

第 1 章 高质量软件开发之道

1 . 1 软件质量基本概念

1 . 1 . 1 如何理解软件的质量

1 . 1 . 2 提高软件质量的基本方法

1 . 1 . 3 “零缺陷”理念

1 . 2 细说软件质

量属性

1 . 2 . 1

正确性

1 . 2 . 2 健

壮性

1 . 2 . 3 可

靠性

1 . 2 . 4 性

能

1 . 2 . 5

易用性

1 . 2 . 6 清

晰性

1 . 2 . 7 安

全性

1 . 2 . 8 可

扩展性

1 . 2 . 9 兼

容性

1 . 2 . 1 0

可移植性

1 . 3 人们关注的  
不仅仅是质量

1 . 3 . 1 质  
量、生产率和成本之间的  
关系

1 . 3 . 2 软  
件过程改进的基本概念

1 . 4 高质量软件  
开发的基本方法

1 . 4 . 1 建  
立软件过程规范

用	1 . 4 . 2	复
而治之	1 . 4 . 3	分
化与折中	1 . 4 . 4	优
术评审	1 . 4 . 5	技
试	1 . 4 . 6	测
量保证	1 . 4 . 7	质
错	1 . 4 . 8	改

## 1 . 5 关于软件开发的一些常识和思考



1 . 5 . 1 有  
最好的编程语言吗

1 . 5 . 2 编  
程是一门艺术吗

1 . 5 . 3 编  
程时应该多使用技巧吗

1 . 5 . 4 换  
更快的计算机还是换更快的  
算法

1 . 5 . 5 错  
误是否应该分等级

1 . 5 . 6 一  
些错误的观念

1 . 6 小结  
第 2 章 编程语言发展简  
史

	2 . 1	编程语
言大事记		
	2 . 2	A d a
的故事		
	2 . 3	C / C
+ + 发展简史		
	2 . 4	B o r
l a n d 与 M i c r o s		
o f t 之争		
	2 . 5	J a
v a 阵营与 M i c r o s		
o f t 的较量		
	2 . 6	小结
第 3 章	程序的基本概念	
	3 . 1	程序设
计语言		

3 . 2 语言实  
现

3 . 3 程序  
库

3 . 4 开发  
环境

3 . 5 程序的  
工作原理

3 . 6 良好  
的编程习惯

第 4 章 C + + / C 程序  
设计入门

4 . 1 C +  
+ / C 程序的基本概念

4 . 1 . 1 启  
动函数main ( )

命令行参数	4 . 1 . 2	命
部名称	4 . 1 . 3	内
接规范	4 . 1 . 4	连
量及其初始化	4 . 1 . 5	变
C R u n t i m e L i b r a r y	4 . 1 . 6	
译时和运行时的不同	4 . 1 . 7	编
译单元和独立编译技术	4 . 1 . 8	编
	4 . 2	基本数

## 据类型和内存映像

4 . 3 类型转  
换

4 . 3 . 1  
隐式转换

4 . 3 . 2 强  
制转换

4 . 4 标识符  
4 . 5 转义

## 序列

4 . 6 运算符  
4 . 7 表达式  
4 . 8 基本控

## 制结构

4 . 9 选择（  
判断）结构

布尔变量与零值比较	4 . 9 . 1	布
型变量与零值比较	4 . 9 . 2	整
点变量与零值比较	4 . 9 . 3	浮
针变量与零值比较	4 . 9 . 4	指
i f 语句的补充说明	4 . 9 . 5	对
w i t c h 结构	4 . 9 . 6	s
循环 ( 重复 ) 结构	4 . 1 0	
f o r 语句的循环控制变	4 . 1 0 . 1	

量

4 . 1 0 . 2

循环语句的效率

4 . 1 1      结构化

程序设计原理

4 . 1 2      g o t

o / c o n t i n u e /

b r e a k 语句

4 . 1 3      示例

第 5 章    C + + / C 常量

5 . 1      认识常量

5 . 1 . 1

字面常量

5 . 1 . 2      符

号常量

5 . 1 . 3      契

## 约性常量

5 . 1 . 4 枚

## 举常量

5 . 2 正确定义符

## 号常量

5 . 3 c o n s

t 与 # d e f i n e 的比  
较

5 . 4 类中的常量

5 . 5 实际应用中

## 如何定义常量

## 第 6 章 C + + / C 函数 设计基础

6 . 1 认识函数

6 . 2 函数原型和

## 定义



式	6 . 3	函数调用方
栈	6 . 4	认识函数堆
范	6 . 5	函数调用规
范	6 . 6	函数连接规
则	6 . 7	参数传递规
则	6 . 8	返回值的规
实现的规则	6 . 9	函数内部实
6 . 1 0	存储类型	
及作用域规则		

6 . 1 0 . 1

存储类型

6 . 1 0 . 2

作用域规则

6 . 1 0 . 3

连接类型

6 . 1 1 递归函数

6 . 1 2 使用断言

6 . 1 3 使用 c o

n s t 提高函数的健壮性

6 . 1 3 . 1

用 c o n s t 修饰函数的  
参数

6 . 1 3 . 2

用 c o n s t 修饰函数的  
返回值

## 第 7 章 C + + / C 指针、数组和字符串

### 7 . 1 指针

#### 7 . 1 . 1 指针的本质

#### 7 . 1 . 2 指针的类型及其支持的运算

#### 7 . 1 . 3 指针传递

### 7 . 2 数组

#### 7 . 2 . 1 数组的本质

#### 7 . 2 . 2 二维数组

#### 7 . 2 . 3 数组传递

7 . 2 . 4 动态创建、初始化和删除数组的方法

7 . 3 字符数组、字符指针和字符串

7 . 3 . 1 字符数组、字符串和 ' \ 0 ' 的关系

7 . 3 . 2 字符指针的误区

7 . 3 . 3 字符串拷贝和比较

7 . 4 函数指针

7 . 5 引用和指针的比较

第 8 章 C + + / C 高级

## 数据类型

8 . 1 结构 ( S t  
r u c t )

8 . 1 . 1 关  
键字 s t r u c t 与 c l  
a s s 的困惑

8 . 1 . 2 使  
用 s t r u c t

8 . 1 . 3 位  
域

8 . 1 . 4 成  
员对齐

8 . 2 联合  
( U n i o n )

8 . 3 枚举 ( E n u m )

8 . 4 文件  
第 9 章 C + + / C 编译  
预处理

9 . 1 文件包含  
9 . 1 . 1  
内部包含卫哨和外部包含  
卫哨

9 . 1 . 2 头  
文件包含的合理顺序

9 . 2 宏定义  
9 . 3 条件编译  
9 . 3 . 1  
# i f 、 # e l i f 和 #  
e l s e

9 . 3 . 2 #  
i f d e f 和 # i f n d

e f

9 . 4 # e r r o

r

9 . 5 # p r a g

m a

9 . 6 # 和 # # 运

算符

9 . 7 预定义符号

常量

第 1 0 章 C + + / C 文  
件结构和程序版式

1 0 . 1 程序文件

的目录结构

1 0 . 2 文件的结

构

1 0 . 2 . 1

## 头文件的用途和结构

1 0 . 2 . 2

## 版权和版本信息

1 0 . 2 . 3

## 源文件结构

1 0 . 3 代码的版  
式

1 0 . 3 . 1

## 适当的空行

1 0 . 3 . 2

## 代码行及行内空格

1 0 . 3 . 3

## 长行拆分

1 0 . 3 . 4

## 对齐与缩进

1 0 . 3 . 5



修饰符的位置

1 0 . 3 . 6

注释风格

1 0 3 . 7

A D T / U D T 版式

第 1 1 章 C + + / C 应  
用程序命名规则

1 1 . 1 共性规则

1 1 . 2 简单的W

i n d o w s 应用程序命  
名

第 1 2 章 C + + 面向对  
象程序设计方法概述

1 2 . 1 漫谈面向

对象

1 2 . 2 对象的概

念

1 2 . 3    信息隐藏  
与类的封装

1 2 . 4    类的继承  
特性

1 2 . 5    类的组合  
特性

1 2 . 6    动态特性  
1 2 . 6 . 1

虚函数

1 2 . 6 . 2  
抽象基类

1 2 . 6 . 3  
动态绑定

1 2 . 6 . 4  
运行时多态

1 2 . 6 . 5

多态数组

1 2 . 7 C + + 对

象模型

1 2 . 7 . 1

对象的内存映像

1 2 . 7 . 2

隐含成员

1 2 . 7 . 3

C + + 编译器如何处理  
成员函数

1 2 . 7 . 4

C + + 编译器如何处理  
静态成员

1 2 . 8 小结

第 1 3 章 对象的初始化

## 、拷贝和析构

1 3 . 1 构造函数  
与析构函数的起源

1 3 . 2 为什么需  
要构造函数和析构函数

1 3 . 3 构造函数的  
成员初始化列表

1 3 . 4 对象的构  
造和析构次序

1 3 . 5 构造函数  
和析构函数的调用时机

1 3 . 6 构造函数  
和赋值函数的重载

1 3 . 7 示例：类  
S t r i n g 的构造函数  
和析构函数

1 3 . 8 何时应该  
定义拷贝构造函数和拷贝  
赋值函数

1 3 . 9 示例：类  
S t r i n g 的拷贝构造  
函数和拷贝赋值函数

1 3 . 1 0 用偷懒  
的办法处理拷贝构造函数  
和拷贝赋值函数

1 3 . 1 1 如何  
实现派生类的基本函数  
第 1 4 章 C + + 函数  
的高级特性

1 4 . 1 函数重  
载的概念

1 4 . 1 . 1

## 重载的起源

1 4 . 1 . 2

## 重载是如何实现的

1 4 . 1 . 3

## 当心隐式类型转换导致 重载函数产生二义性

## 1 4 . 2 成员函数 的重载、覆盖与隐藏

1 4 . 2 . 1

## 重载与覆盖

1 4 . 2 . 2

## 令人迷惑的隐藏规则

1 4 . 2 . 3

## 摆脱隐藏

## 1 4 . 3 参数的默 认值

1 4 . 4 运算符重载

1 4 . 4 . 1  
基本概念

1 4 . 4 . 2  
运算符重载的特殊性

1 4 . 4 . 3  
不能重载的运算符

1 4 . 4 . 4  
重载 + + 和 - -

1 4 . 5 函数内联

1 4 . 5 . 1  
用函数内联取代宏

1 4 . 5 . 2  
内联函数的编程风格

1 4 . 5 . 3

慎用内联

1 4 . 6      类型转换

函数

1 4 . 7      c o n s

t 成员函数

第 1 5 章      C + + 异常  
处理和 R T T I

1 5 . 1      为什么

要使用异常处理

1 5 . 2      C + +

异常处理

1 5 . 2 . 1

异常处理的原理

1 5 . 2 . 2

异常类型和异常对象



1 5 . 2 . 3

异常处理的语法结构

1 5 . 2 . 4

异常的类型匹配规则

1 5 . 2 . 5

异常说明及其冲突

1 5 . 2 . 6

当异常抛出时局部对象  
如何释放

1 5 . 2 . 7

对象构造和析构期间的  
异常

1 5 . 2 . 8

如何使用好异常处理技术

1 5 . 2 . 9

C + + 的标准异常

1 5 . 3 虚函数面  
临的难题

1 5 . 4 R T T I  
及其构成

1 5 . 4 . 1  
起源

1 5 . 4 . 2  
t y p e i d 运算符

1 5 . 4 . 3  
d y n a m i c \_ c a s  
t < > 运算符

1 5 . 4 . 4  
R T T I 的魅力与代价

第 1 6 章 内存管理

1 6 . 1 内存分  
配方式

1 6 . 2 常见的内存错误及其对策

1 6 . 3 指针参数是如何传递内存的

1 6 . 4 f r e e 和 d e l e t e 把指针怎么啦

1 6 . 5 动态内存会被自动释放吗

1 6 . 6 杜绝“野指针”

1 6 . 7 有了 m a l l o c / f r e e 为什么还要 n e w / d e l e t e

1 6 . 8 m a l l

o c / f r e e 的使用要点

1 6 . 9 n e w 有  
3 种使用方式

1 6 . 9 . 1  
p l a i n n e w /  
d e l e t e

1 6 . 9 . 2  
n o t h r o w n e w  
/ d e l e t e

1 6 . 9 . 3  
p l a c e m e n t n  
e w / d e l e t e

1 6 . 1 0 n  
e w / d e l e t e 的使  
用要点

1 6 . 1 1

内存耗尽怎么办

1 6 . 1 2 用

对象模拟指针

1 6 . 1 3 泛

型指针 `auto_ptr`

1 6 . 1 4 带

有引用计数的智能指针

1 6 . 1 5 智能指

针作为容器元素

第 1 7 章 学习和使用 S

T L

1 7 . 1 S

T L 简介

1 7 . 2 S

T L 头文件的分布

	1 7 . 2 . 1	
容器类		
	1 7 . 2 . 2	
泛型算法		
	1 7 . 2 . 3	
迭代器		
	1 7 . 2 . 4	
数学运算库		
	1 7 . 2 . 5	
通用工具		
	1 7 . 2 . 6	其他
头文件		
1 7 . 3	容器设计原理	
1 7 . 3 . 1	内存	
映像		
1 7 . 3 . 2	存储	

## 方式和访问方式

1 7 . 3 . 3 顺序

## 容器和关联式容器的比较

1 7 . 3 . 4 如何

## 遍历容器

1 7 . 3 . 5 存储

## 空间重分配问题

1 7 . 3 . 6 什么

## 样的对象才能作为 S T L 容器的元素

1 7 . 4 迭代器

1 7 . 4 . 1 迭代

## 器的本质

1 7 . 4 . 2 迭

## 代器失效及其危险性

1 7 . 5 存储分配器

1 7 . 6 适配器

1 7 . 7 泛型算法

1 7 . 8 一些特殊  
的容器

1 7 . 8 . 1

s t r i n g 类

1 7 . 8 . 2

b i t s e t 并非 s e t

1 7 . 8 . 3

节省存储空间的 v e c t  
o r < b o o l >

1 7 . 8 . 4

空容器

1 7 . 9 S T L 容

器特征总结

1 7 . 1 0 S T L



使用心得

附录 A C + + / C 试题

附录 B C + + / C 试

题答案与评分标准

附录 C 大学十年

附录 D 《大学十年》

后记

附录 E 术语与缩写解释

参考文献