

Borland C++ 3.1 编程指南

[美] Ted Faison 著

蒋维杜 吴志美
张新宇 李景淑 译
李 莉 审校

清华大学出版社

目 录

引言.....	(1)
0.1 为什么使用 OOP 方法?	(1)
0.2 本书结构	(2)
0.3 本书的描述	(2)
0.4 软硬件需求	(3)

第一部分 用 Borland C++ 进行面向对象的程序设计

第 1 章 基础知识	(3)
1.1 Borland C++ 项目文件的结构	(3)
1.1.1 头文件	(3)
1.1.2 一个完整的样本程序	(4)
1.2 变量	(8)
1.2.1 作用域	(8)
1.2.2 类型.....	(10)
1.2.3 存储类.....	(10)
1.2.4 const 限定符	(10)
1.2.5 Volatile 限定符	(12)
1.2.6 语句.....	(13)
1.2.7 函数.....	(21)
1.2.8 指针和引用.....	(26)
1.2.9 指针、引用与 const 连用	(28)
1.2.10 提高部分	(29)
第 2 章 对象和类	(41)
2.1 定义类.....	(41)
2.1.1 类标识符.....	(42)
2.1.2 类体.....	(42)
2.2 使用类.....	(43)
2.2.1 封装.....	(43)
2.2.2 类存取控制.....	(44)
2.2.3 类私有成员.....	(45)
2.2.4 类公有成员.....	(46)
2.2.5 类保护成员.....	(46)

2.2.6	类对象的存储类	(47)
2.2.7	类作用域	(47)
2.2.8	空类	(48)
2.2.9	类嵌套	(48)
2.2.10	类的实例化	(50)
2.2.11	不完全的类声明	(50)
2.3	使用数据成员	(51)
2.3.1	静态数据成员	(51)
2.3.2	private static 数据成员	(53)
2.3.3	类对象用作数据成员	(54)
2.3.4	指针数据成员	(56)
2.3.5	指向类数据成员的指针	(56)
2.3.6	指向对象数据成员的指针	(57)
2.4	使用成员函数	(58)
2.4.1	简单成员函数	(59)
2.4.2	静态成员函数	(59)
2.4.3	Const 成员函数	(60)
2.4.4	volatile 成员函数	(61)
2.4.5	内联成员函数	(61)
2.4.6	带有 const this 的成员函数	(62)
2.4.7	带有 volatile this 的成员函数	(63)
2.4.8	特殊类函数	(64)
2.4.9	构造函数	(65)
2.4.10	析构函数	(70)
2.4.11	友元关键字	(72)
2.4.12	友元的性质	(73)
2.5	提高部分	(74)
2.5.1	成员函数指针	(74)
2.5.2	数组和类	(76)
2.5.3	成员函数调用的分析	(81)
2.5.4	类模板	(82)
2.5.5	函数模板	(89)
第3章	继承性	(92)
3.1	可复用性	(92)
3.2	继承性	(92)
3.3	继承的作用	(93)
3.4	C++继承性的局限性	(93)
3.5	关于继承的不同观察角度	(94)

3.6	单一继承	(94)
3.6.1	何时继承	(94)
3.6.2	不能被继承的成分	(95)
3.6.3	基类的存取限定符	(95)
3.6.4	可被继承的类	(96)
3.6.5	传递给基类的参数	(97)
3.6.6	构造函数的调用顺序	(98)
3.6.7	析构函数的调用顺序	(99)
3.6.8	种子类	(99)
3.6.9	派生类的类型转换	(101)
3.6.10	作用域的分辨	(102)
3.6.11	性质扩展	(105)
3.6.12	性质约束	(107)
3.6.13	使用单一继承的例子	(108)
3.6.14	函数闭包	(110)
3.7	多重继承	(114)
3.7.1	声明多基类继承的类	(115)
3.7.2	调用基类构造函数	(116)
3.7.3	使用虚基类	(116)
3.7.4	混合使用虚基类和非虚基类	(118)
3.7.5	调用析构函数	(118)
3.7.6	使用类型转换	(118)
3.7.7	保持基类函数的正确性	(120)
3.7.8	多继承中作用域分辨的应用	(121)
3.7.9	跟踪内存	(122)
3.8	提高部分	(123)
3.8.1	运行时刻的考虑	(123)
3.8.2	进入对象内部	(123)
3.8.3	被继承的 Debugger 类	(126)
第4章	重载	(130)
4.1	重载的原因	(130)
4.2	函数重载	(131)
4.2.1	非成员重载函数	(131)
4.2.2	重载成员函数	(132)
4.2.3	类等级中的重载函数	(133)
4.2.4	重载不是覆盖	(134)
4.2.5	作用域分辨	(134)
4.2.6	参数匹配	(135)

4.2.7	重载构造函数	(136)
4.2.8	一些特殊情况	(137)
4.2.9	通过重载定义用户转换规则	(138)
4.2.10	重载静态成员函数	(141)
4.3	操作符重载	(141)
4.3.1	操作符用作函数调用	(143)
4.3.2	重载操作符用作成员函数	(143)
4.3.3	操作符成员函数的几个注意点	(145)
4.3.4	重载操作符用作友元函数	(145)
4.3.5	赋值操作符	(147)
4.3.6	函数调用操作符()	(149)
4.3.7	下标操作符	(151)
4.3.8	操作符重载限制	(152)
4.3.9	操作符的作用域分辨	(152)
4.4	提高部分	(153)
4.4.1	名字分裂的规则	(153)
4.4.2	重载 new 和 delete	(156)
4.4.3	前缀和后缀操作符	(158)
第5章	多态性	(161)
5.1	先期和迟后联编	(161)
5.2	C++是一种混合语言	(162)
5.3	虚函数	(162)
5.3.1	函数覆盖	(163)
5.3.2	空的虚函数	(164)
5.3.3	改善了的类用户接口	(165)
5.3.4	抽象类	(166)
5.3.5	虚函数的局限性	(169)
5.3.6	虚友元	(169)
5.3.7	虚操作符	(170)
5.3.8	虚构造函数	(172)
5.3.9	虚析构函数	(172)
5.4	多态性的例子	(172)
5.5	作用域分辨使多态性失效	(175)
5.6	虚函数和非虚函数连用	(176)
5.7	vptr 和 vtab 结构的内存布局	(177)
5.8	虚函数可以不被覆盖	(178)
5.9	确定是否使用虚函数	(179)
5.10	私有虚函数	(180)

5.11	提高部分	(182)
5.11.1	多态性机制	(182)
5.11.2	单一继承中的多态性	(182)
5.11.3	多重继承中的多态性	(186)
5.11.4	嵌入式虚函数	(189)
5.11.5	基类中调用多态函数	(192)
5.11.6	虚函数和分类等级	(194)
5.11.7	构造函数中调用虚函数	(196)
第 6 章	流	(198)
6.1	Stdio 方法的缺点	(198)
6.2	C++ 流	(199)
6.3	广义的流	(199)
6.4	内部类型的标准 I/O 流	(200)
6.4.1	char 和 char * 类型的 I/O 操作	(202)
6.4.2	int 和 long 类型的 I/O 操作	(203)
6.4.3	float 和 double 类型的 I/O 操作	(204)
6.4.4	用户类的 I/O 操作	(204)
6.5	操作函数	(206)
6.5.1	使用数制(基)操作函数	(208)
6.5.2	设置和清除格式标志	(209)
6.5.3	改变域宽及填充	(209)
6.5.4	使用格式操作函数	(210)
6.6	文件 I/O 的流实现	(211)
6.6.1	文本文件输入	(212)
6.6.2	流的错误检测	(213)
6.6.3	文本文件输出	(214)
6.6.4	二进制文件输入	(216)
6.6.5	二进制文件输出	(217)
6.7	内存格式化	(218)
6.8	将打印机看作流	(220)
6.9	提高部分	(221)
6.10	内部流类型	(221)
6.11	streambuf 等级	(221)
6.11.1	类 streambuf	(222)
6.11.2	从类 streambuf 中派生类	(229)
6.11.3	类 strstreambuf	(232)
6.11.4	类 filebuf	(237)
6.12	ios 等级	(242)

6.12.1	类 ios	(243)
6.12.2	类 istream	(254)
6.12.3	类 ostream	(260)
6.12.4	类 iostream	(263)
6.12.5	类 istream.withassign	(266)
6.12.6	类 ostream.withassign	(268)
6.12.7	类 iostream.withassign	(269)
6.12.8	类 fstreambase	(270)
6.12.9	类 strstreambase	(273)
6.12.10	类 ifstream	(275)
6.12.11	类 ofstream	(277)
6.12.12	类 fstream	(280)
6.12.13	类 istrstream	(282)
6.12.14	类 ostrstream	(285)
6.12.15	类 strstream	(289)
6.12.16	流中二进制文件操作和文本文件操作	(291)
6.12.17	用户定义的操作函数	(296)
6.13	流代码的大小	(303)

第7章 包容类库 (304)

7.1	类的等级结构的优点	(304)
7.2	类层次结构的目标	(306)
7.3	包容类	(306)
7.3.1	类的分类	(306)
7.3.2	程序运行时对类的识别	(308)
7.4	基于 Object 类的包容类	(308)
7.5	类 AbstractArray	(309)
7.6	Array 类	(313)
7.6.1	Array 类的使用	(314)
7.6.2	数组中某一存储位置的再次使用	(315)
7.7	Association(关联)类	(316)
7.7.1	关联所用到的对象的定义	(318)
7.7.2	Association 类的使用	(319)
7.7.3	从 Association 派生类	(321)
7.8	类 Bag	(322)
7.9	类 BaseDate	(326)
7.10	类 BaseTime	(329)
7.11	类 Btree	(331)
7.11.1	树的基本概念	(331)

7.11.2	二叉树	(332)
7.11.3	B 树	(333)
7.12	类 Collection	(345)
7.13	类 Container	(347)
7.14	类 Date	(352)
7.15	类 Deque	(358)
7.16	类 Dictionary	(360)
7.16.1	一个 Dictionary 的例子	(361)
7.16.2	用外部循环量遍历 Dictionary 包容	(363)
7.17	类 DoubleList	(364)
7.18	类 Error	(369)
7.19	类 HashTable	(371)
7.20	类 List	(378)
7.21	类 Object	(382)
7.22	类 PriorityQueue	(385)
7.22.1	类 PriorityQueue 的使用	(387)
7.22.2	把优先队列转换成 GIFO 队列	(390)
7.23	类 Queue	(390)
7.24	类 Set	(394)
7.24.1	处理字符串的类 Set	(394)
7.24.2	一个更接近数学意义上的集合的类 Set	(396)
7.25	类 Sortable	(398)
7.26	类 SortedArray	(400)
7.27	类 Stack	(402)
7.28	类 String	(404)
7.28.1	类 String 的使用	(407)
7.28.2	从类 String 派生出新的类	(408)
7.29	类 Time	(410)
7.29.1	类 Time 的使用	(411)
7.29.2	由类 Time 派生出新类	(413)
7.30	类 Timer	(416)
7.31	类 TShouldDelete	(419)
7.32	循环量	(420)
7.33	构造类库	(422)
7.34	基于模板的包容类	(422)
7.35	FDS 和 ADT 包容	(423)
7.36	FDS 包容	(423)
7.36.1	FDS 存储范例	(423)
7.36.2	FDS 包容	(424)

7.36.3	FDS 向量包容	(424)
7.36.4	简单直接向量.....	(426)
7.36.5	直接计数向量.....	(426)
7.36.6	直接排序向量.....	(427)
7.36.7	间接简单向量.....	(429)
7.36.8	间接计数向量.....	(430)
7.36.9	间接排序向量.....	(431)
7.37	FDS 表包容	(433)
7.37.1	直接简单表.....	(435)
7.37.2	直接排序表.....	(435)
7.37.3	间接表.....	(438)
7.37.4	间接排序表.....	(439)
7.38	ADT 包容	(439)
7.38.1	ADT 数组	(440)
7.38.2	ADT 排序数组	(444)
7.38.3	ADT 栈	(445)
7.38.4	ADT 队列和双端队列	(449)
7.38.5	ADT 包和集合	(452)
7.38.6	异质 ADT 包容	(456)

第二部分 开发 Windows 和 DOS 应用程序

第 8 章	用于 Windows 程序设计的类	(459)
8.1	运行检测程序	(459)
8.2	最终为 PC 机提供标准 GUI	(460)
8.3	围绕对象设计的 Windows	(460)
8.4	C++ 中对 Windows 资源的管理	(461)
8.5	设备描述表	(461)
8.6	绘图闭包	(471)
8.7	对话框	(476)
8.8	位图	(485)
8.9	裁剪板	(488)
8.10	真正的出发点.....	(494)
第 9 章	完整的窗口程序.....	(495)
9.1	文件对话框	(495)
9.1.1	文件打开用的对话框	(501)
9.1.2	文件存储对话框	(504)
9.2	使用打印机	(510)

9.2.1	使用 DLL	(510)
9.2.2	Printer Driver DLL	(512)
9.2.3	Printer 类	(514)
9.3	文本编辑器	(520)
9.3.1	主菜单	(520)
9.3.2	实现	(521)
9.3.3	About 对话框	(526)
第 10 章	对象窗口类库	(528)
10.1	忘记 ANSI C 风格的 Windows 程序	(528)
10.2	Borland“所见即所得”	(528)
10.3	OWL 应用程序的解析	(529)
10.3.1	定制主窗口	(533)
10.3.2	对话框处理	(535)
10.3.3	增加菜单	(537)
10.4	无模式对话框	(538)
10.4.1	带色斑的对话框	(541)
10.4.2	全部数字式	(543)
10.4.3	对话框数据的读写	(544)
10.4.4	子控制框的数据合法性检查	(550)
10.5	自画式控制	(558)
10.5.1	BWCC 的自画式控制	(558)
10.5.2	更多的数字	(558)
10.5.3	非 BWCC 的自画式控制	(561)
10.5.4	自画式圆按钮	(567)
10.6	另外一个文件打开对话框	(573)
10.7	持续的 OWL 对象	(582)
第 11 章	OWL 应用程序	(589)
11.1	OWL 应用程序的概貌	(589)
11.2	状态行	(589)
11.2.1	子窗口方式	(590)
11.2.2	GDI 方式	(597)
11.3	弹出式菜单	(606)
11.3.1	菜单的处理	(607)
11.3.2	一个完整的应用程序:WPOPUP	(608)
11.4	浮动式工具调色板	(614)
11.4.1	一般的画图工具	(615)
11.4.2	专用的画图工具	(616)

11.4.3	系统管理员	(617)
11.4.4	一个完整的应用程序:TOOL	(619)
11.5	扩展 Borland 资源命名规定	(633)
11.6	编辑窗口	(634)
11.6.1	用弹出式窗口编辑	(634)
11.6.2	装入键盘加速键	(638)
11.6.3	用文件编辑	(638)
11.6.4	利用现成的 Borland 资源	(640)
11.6.5	一个完整的应用程序:WFEDIT	(641)
11.7	MDI 应用程序	(642)
11.7.1	OWL 的类 MDI	(643)
11.7.2	一个完整的应用程序:MDIDEMO	(644)
11.7.3	处理菜单命令的子 MDI	(647)
11.7.4	子窗口菜单	(648)
第 12 章	Turbo Vision 类	(650)
12.1	以字符方式工作的 GUI	(650)
12.1.1	Turbo Vision	(650)
12.1.2	Turbo Vision 的新标记和新概念	(651)
12.1.3	TCollection 类库	(652)
12.1.4	访问 TCollection	(652)
12.1.5	持久对象	(653)
12.2	一个 Turbo Vision 应用程序的分析	(653)
12.2.1	初始化用户应用程序	(653)
12.2.2	定制应用程序	(656)
12.2.3	理解彩色调色板	(657)
12.2.4	定制桌面	(662)
12.2.5	定制状态行	(665)
12.2.6	定制应用程序的颜色	(669)
12.2.7	定制菜单栏	(673)
12.2.8	定制窗口	(678)
12.3	对话框	(683)
12.3.1	对话控制中的一般问题	(687)
12.3.2	内部对话框的定制	(688)
12.3.3	视图到视图消息	(692)
12.3.4	广播消息	(693)
12.4	持久对象	(700)
12.4.1	避免流管理程序问题	(701)
12.4.2	注册持久类	(701)

12.4.3	辅助流管理程序	(702)
12.4.4	确定读写内容	(702)
12.4.5	初始化持久对话框	(703)
12.4.6	利用资源文件和持久对象	(704)
12.4.7	组装	(705)
12.4.8	使用没有流的持久对象	(712)
12.4.9	防守式的程序设计	(713)
第 13 章	Turbo Vision 应用程序	(714)
13.1	Turbo Vision 与 Microsoft Windows	(714)
13.2	弹出式窗口	(715)
13.2.1	链接一个弹出式窗口到热键上	(715)
13.2.2	处理热键事件	(715)
13.2.3	编写一个完整的应用程序	(716)
13.3	文件浏览器	(719)
13.3.1	增加滚动条	(719)
13.3.2	开发窗口类层次体系	(720)
13.3.3	编写一个完整的应用程序	(720)
13.3.4	检验窗口合法性	(726)
13.3.5	检验文件大小	(727)
13.3.6	控制窗口属性	(728)
13.4	弹出式菜单	(728)
13.4.1	链接一个弹出式菜单到一热键上	(728)
13.4.2	处理热键事件	(730)
13.4.3	编写一个完整的应用程序	(731)
13.5	带提示的状态行	(736)
13.5.1	定义帮助描述表	(736)
13.5.2	由帮助描述表获得提示	(736)
13.5.3	对菜单使用提示	(737)
13.5.4	对用户类使用提示	(738)
13.5.5	编写一个完整的应用程序	(738)
13.6	视区相关的状态行	(743)
13.6.1	理解帮助描述表范围	(745)
13.6.2	编写一个完整的应用程序	(745)
13.7	上下文相关的帮助	(749)
13.7.1	编制一个帮助文本文件	(749)
13.7.2	编译帮助文件	(750)
13.7.3	从应用程序中激活帮助系统	(751)
13.7.4	编写一个完整的应用程序	(752)

13.8 查看特性.....	(757)
13.8.1 使用遗忘的右鼠标按钮.....	(758)
13.8.2 产生特征检阅命令.....	(760)
13.8.3 编写一个完整的应用程序.....	(760)
13.9 用于编辑的窗口.....	(768)
13.9.1 开发 Turbo Vision 裁剪板类	(768)
13.9.2 使用 TEditWindow 的内部命令	(769)
13.9.3 为 TEditWindow 的提示和出错处理提供支持	(771)
13.9.4 编写一个完整的应用程序.....	(772)
13.10 后记	(781)
参考文献.....	(782)

第一部分

用 Borland C++ 进行 面向对象的程序设计



第 1 章 基础知识

本章探讨了 ANSI C 扩展或派生出的 C++ 语言的主要特征,但没有讲述对象或其它面向对象的构造。笔者花大量笔墨描述了 C++ 语言的一些很少被人了解的特征,并避免了对细小问题的冗长讨论。想了解更基本的方面,可以阅读《Borland C++ 程序员指南》,随标准 Borland 文档提供。

C++ 语言属于混合型面向对象程序设计语言,这是相对于纯面向对象的语言或传统语言而言的,因为它是建立在传统过程语言基础之上的。其他如 Eiffel 或 Smalltalk 等语言,是纯面向对象的语言,但这并没有令它们更具优势。评定一种语言是否最优要看怎样使用它以及它能解决什么任务。

C++ 源代码在执行前必须先编译。因此程序设计过程必然包括编辑、编译、连接、执行这样一个开发周期。尽管这一周期的反复进行是一个缓慢的过程,产生的代码却快速高效。C++ 语言体现了表达能力、运行速度、代码大小的最佳搭配。更有用的是,Borland C++ 的集成开发环境(IDE)自动地将开发周期各个阶段连接起来,使整个周期显著加快。

因此 C++ 语言是作为 C 语言的提高和扩展而设计的,它具有 ANSI C 的传统特征。然而,正如任何软件的新版本一样,即便一些基本项也会有变化。这些特征在本章讲述 C 语言特征时都重点指出了。C 语言程序员不会发现太多的改变,但这并不鼓励他们跳过这章到第 2 章去,因为描述的变化都是很重要的。

1.1 Borland C++ 项目文件的结构

在结构上,Borland C++ 项目文件与 ANSI C 很类似。在头文件中建立结构(和类)以及常量的声明。因为 C++ 头文件中可能包括 ANSI C 不能编译的构造,所以用 *.hpp 来标识,以区别于通常的 ANSI C 头文件 *.h。正如 ANSI C 一样,C++ 实际源代码包含在一个或多个源文件中,然后被编译、连接起来。

1.1.1 头文件

类标识符在使用前必须进入作用域,这通常是用类声明来实现的。使用类的公有成员时,类标识符不仅要在活动作用域中,而且必须被完全声明。因为每个类都有自己的声明部分,而且各种模块都可以使用它,所以常把声明置于头文件中,用到该类的每个模块再包含相应的头文件。

1.1.1.1 多次包含的难题

假设一典型的程序是由大量类和头文件组成的,常常会遇到这样一个难题:在同一文件中一个头文件可能被包含多次。例如:

```
//file HEADER.HPP
class EssentialClass { /* ... */ };
//file DESKTOP.HPP
```



```

#include "HEADER.HPP"
class DeskTop { /* ... */ };

//file DRAWER.C
#include "HEADER.HPP"
#include "DESKTOP.HPP" // file HEADER.HPP included
// twice !

class Drawer { /* ... */ };

```

为避免这种常见错误,C++头文件用一些预处理指令产生下面的简单而有效的分离构造:

```

// file HEADER.HPP
# ifndef HEADER.HPP
# define HEADER.HPP
class Essentialclass { /* ... */ };
...
# endif

```

所有的头文件都采用相同的机制。通常在#define语句中使用头文件的文件名或其省略形式。这样试图再次包含同一文件的错误就可以被预处理器发现并处理。

1.1.1.2 预编译头文件

处理C++源文件时,常因调入大量的头文件而花费大量时间,这使C++用户大为抱怨。有时编译器装入、扫描头文件的时间甚至比编译源文件的时间还长。经常同一头文件因被不同模块使用而一次次地装入内存。这似乎是一种时间上的浪费。

Borland公司的工作人员发现了这一缺点,并用预编译的方法解决了它。当Borland C++扫描源文件包含的头文件时,在磁盘上保留一个“预编译”版本。下次需包含同一头文件时,编译器读预编译过的版本,这大大加快了整个编译过程。当包含的头文件很大或数量很多时,速度的加快是显著的。最常用的头文件有windows.h,dos.h,iostream.h和graphics.h等头文件。

使用预编译的头文件,必须通过Options|Compiler|Source菜单选中Precompiled Header项。当选中该选项时,编译器扫描源文件中包含的头文件,在磁盘上tdef.sym文件中存入其映象。编译器一次只能处理一个预编译的头文件,不过可以不用缺省的文件名以产生不同的预编译头文件。由于这类文件可能会变得很大,这么做必须保证有足够的磁盘空间。如果在制造预处理文件时系统缺少空间,可能会出现一些奇怪的错误信息或结果。

1.1.2 一个完整的样本程序

有关C或C++语言的书大多是从常见的“hello,world!”程序开始的。该程序包括一个小的main()函数,在屏幕上打印一行信息,然后终止。因为本书不是介绍性质的,所以直接从一个更复杂的,包括绝大多数基本构造的程序开始。这使读者马上接触到语言的过程部分,而不必等到读过书的四分之三之后。

实例1.1 程序接收用户键入的一行字符,计算其中有多少空格,然后打印结果。

实例1.1 一个完整的程序

```

// Count the number of spaces in a line of text

```