The-Qt4-Book

Jasmin B., Mark S.[①] | 德山书生[②]

版本: 0.01

---

① 作者全名：Jasmin Blanchette, Mark Summerfield。本文代码来自这个网站。原中文翻译：闫锋欣，曾泉人，张志强；原中文审校：周莉娜，赵延兵。

② 编者：德山书生，湖南常德人氏。我负责整理排版工作。本项目 Github 网站在这里。有意见请反馈。编者邮箱：a358003542@gmail.com。

前言

为什么会是 Qt？为什么像我这样的程序员会选择 Qt？这个问题的答案显而易见：Qt 单一源程序的兼容性、丰富的特性、C++ 方面的性能、源代码的可用性、它的文档、高质量的技术支持，以及在奇趣科技公司那些精美的营销材料中所涉及的其他优势等。这些答案看起来确实都不错，但是遗漏了最为重要的一点：Qt 的成功缘于程序员们对它的喜欢。

那么，是什么让程序员喜欢某种技术而放弃另外一种呢？就我而言，我认为软件工程师们喜欢某种技术，是因为他们觉得这种技术是合适的，但是这也会让他们讨厌所有那些他们觉得不合适的其他技术。除此之外，我们还能解释下面的这些情况吗？例如，一些最出众的程序员需要在帮助之下才能编写出一个录像机程序，或者又比如，似乎大多数工程师在操作本公司的电话系统时总会遇到麻烦。我虽然善于记住随机数字和指令的序列，但是如果将其比作用于控制我的应答系统所需要的条件来说，则可能一条也不具备。在奇趣科技公司，我们的电话系统要求在拨打其他人的分机号码前，一定要按住”*” 键 2 秒后才允许开始拨号。如果忘记了这样做而是直接拨打分机号码，那么就不得不再重新拨一遍全部的号码。为什么是”*” 键而不是”#” 键、”1” 键或者”5” 键？或者为什么不是 20 个电话键盘中的其他任何一个呢？又为什么是 2 秒，而不是 1 秒、3 秒或者 1.5 秒呢？问题到底出在哪里？我发现电话很气人，所以我尽可能不去使用它。没有人喜欢总是去做一些不得不做的随机事情，特别是当这些随机事情显然只出现在同样随机的情况下的时候，真希望自己从来都没有听到过它。

编程很像我们正在使用的电话系统，并且要比它还糟糕。而这正是 Qt 所要解决的问题。Qt 与众不同。一方面，Qt 很有意义；另一方面，Qt 颇具趣味性。Qt 可以让您把精力集中在您的任务上。当 Qt 的首席体系结构设计师面对一个问题的时候，他们不是寻求一个好的、快速的或者最简便的解决方案，而是在寻求一个恰当的解决方案，然后将其记录在案。应当承认，他们犯下了一些错误，并且还要承认的是，他们的一些设计决策没有通过时间的检验，但是他们确实做出了很多正确的设计，并且那些错误的设计应当而且也是能够进行改正的。看一看最初设计用于构建 Windows 95 和 UNIX Motif 之间的桥梁系统，到后来演变为跨越 Windows Vista、Mac OS X

和 GNU/Linux 以及那些诸如移动电话等小型设备在内的统一的现代桌面系统，这些事实就足以证明这一点。

早在 Qt 大受欢迎并且被广泛使用很久以前，正是 Qt 的开发人员为寻求恰当的解决方案所做出的贡献才使 Qt 变得与众不同。其贡献之大，至今仍然影响着每一个对 Qt 进行开发和维护的人。对我们而言，研发 Qt 是一种使命和殊荣。能够使您的职业生涯和开源生活变得更为轻松和更加有趣，这让我们倍感自豪。

人们乐于使用 Qt 的诸多原因之一是它的在线帮助文档，但是该帮助文档的主要目的是集中介绍个别的类，而很少讲述应当如何构建现实世界中那些复杂的应用程序。这本好书填补了这一缺憾，它展示了 Qt 所提供的东西，如何使用“Qt 的方式”进行 Qt 编程，以及如何充分地利用 Qt。本书将指导 C++、Java 或者 C# 程序员进行 Qt 编程，并且提供了丰富详实的资料来使他们成长为老练的 Qt 程序员。这本书包含了很多很好的例子、建议和说明——并且，该书也是我们对那些新加入公司的程序员们进行培训的入门教材。

如今，已有大量的商业或者免费的 Qt 应用程序可以购买或者下载，其中的一些专门用于特殊的高端市场，其他一些则面向大众市场。看到如此多的应用程序都是基于 Qt 构建而成的，这使我们充满了自豪感，并且还激励我们要让 Qt 变得更好。相信在这本书的帮助下，将会前所未有地出现更多的、质量更高的 Qt 应用程序。

Matthias Ettrich

德国，柏林

2007 年 11 月

序言

Qt 使用“一次编写，随处编译”的方式为开发跨平台的图形用户界面应用程序提供了一个完整的 C++ 应用程序开发框架。Qt 允许程序开发人员使用应用程序的单一源码树来构建可以运行在不同平台下的应用程序的不同版本；这些平台包括从 Windows 98 到 Vista、MacOS X、Linux、Solaris、HP-UX 以及其他很多基于 X11 的 Unix。许多 Qt 库和工具也都是 Qt/Embedded Linux 的组成部分。Qt/Embedded Linux 是一个可以在嵌入式 Linux 上提供窗口系统的产品。

本书的目标就是教您如何使用 Qt4 来编写图形用户界面程序。本书从”Hello Qt”开始，然后很快地转移到更高级的话题中，如自定义窗口部件的创建和拖放功能的提

供等。通过本书的互联网站点，您可以下载到一些作为本书文字补充材料的示例程序。附录 A 说明了如何下载和安装这些软件，其中包括一个用于 Windows 的 C++ 免费编译器。

本书分为四部分。第一部分涵盖了在使用 Qt 编写图形用户界面应用程序时所必需的全部基本概念和练习。仅掌握这一部分中所蕴含的知识就足以写出实用的图形用户界面应用程序。第二部分进一步深入介绍了 Qt 的一些重要主题，第三部分则提供了更为专业和高级的材料。您可以按任意顺序阅读第二部分和第三部分中的章节，但这是建立在您对第一部分中的内容非常熟悉的基础之上的。第四部分包括数个附录，附录 B 说明了如何构建 Qt 应用程序，附录 C 则介绍了 Qt Jambi，它是 Java 版的 Qt。

本书的第一版建立在 Qt 3 版本的基础上，尽管已通过全书修订来反映那些很好的 Qt4 编程技术，但本书还是根据 Qt4 的模型，视图结构、新的插件框架、使用 Qt/Embedded Linux 进行嵌入式编程等内容而引入了一些新的章节和一个新的附录。作为第二版，本书充分利用了 Qt 4.2 和 Qt 4.3 中引入的新特性对其进行了彻底更新，并包含“自定义外观”和“应用程序脚本”两个新的章以及两个新的附录。原有的“图形”一章已经拆分为“二维”和“三维”两章，在它们中间，涵盖了新的图形视图类和 QPainter 的 OpenGL 后端实现。此外，在数据库、XML 和嵌入式编程等几章中，还添加了许多新内容。

与本书的前两版一样，这一版的重点放在如何进行 Qt 编程的说明和许多真实例子的提供上，而不是对丰富的 Qt 在线文档的简单拼凑和总结。因为本书纯粹讲授的是 Qt 4 编程中的原理和实践知识，因而读者能够轻松学会将要出现在 Qt 4.4、Qt 4.5 以及 Qt 4.x 等后续版本中的 15 个 Qt 新模块。如果您正在使用的 Qt 版本恰好是这些后续版本中的一个，那么当然要阅读一下参考文档中的”What’s New in Qt 4.x” 一章，以便可以对那些可用的新特性有一个总体把握。

在写作本书的时候，是假定您已经具备了 C++、Java 或者 C# 的基本知识。本书中的例子代码使用的是 C++ 中的一个子集，从而避免了很多在 Qt 编程中极少使用的 C++ 特性。在某些不可避免而必须使用 C++ 高级结构的地方，会在使用时对其做出必要的解释。如果您对 Java 或者 C# 已经非常熟悉但是对 C++ 还知之不多甚至一无所知，那么建议您先阅读附录 D。附录 D 提供了对 C++ 较为充分的介绍，从而能够让您具有使用本书所必备的 C++ 知识。对于 C++ 中的面向对象编程更为全面的介绍，建议您阅读由 P. J. Deitel 和 H. M. Deitel 编著的”C++ How to Program”(Prentice Hall , 2007)，以及由 Stanley B. Lippman , Josée Lajoie 和 Barhara E. Moo 编著的”C++ Primer”(Addison-Wesley , 2005) 这两本书。

Qt 简史

Qt 框架首度为公众可用是在 1995 年 5 月。它最初由 Haavard Nord（奇趣科技公司的 CEO）和 Eirik Chambe-Eng（公司总裁）开发而成。Haavard 和 Eirik 在位于挪威特隆赫姆的挪威科技学院相识，在那里，他们都获得了计算机科学的硕士学位。

Haavard 对 C++ 图形用户界面开发的兴趣始于 1988 年，当时一家瑞典公司委托他开发一套 C++ 图形用户界面框架。几年后，在 1990 年的夏天，Haavard 和 Eirik 因为一个超声波图像方面的 C++ 数据库应用程序而在一起工作。这个系统需要一个能够在 UNIX、Macintosh 和 Windows 上都能运行的图形用户界面。在那个夏天中的某天，Haavard 和 Eirik 一起出去散步，享受阳光，当他们坐在公园的一条长椅上时，Haavard 说：“我们需要一个面向对象的显示系统。”由此引发的讨论，为他们即将创建的面向对象的、跨平台的图形用户界面框架奠定了智力基础。

1991 年，Haavard 和 Eirik 开始一起合作设计、编写最终成为 Qt 的那些类。在随后的一年中，Eirik 提出了“信号和槽”的设想——一个简单并且有效的强大的图形用户界面编程规范，而现在，它已经可以被多个工具包实现。Haavard 实践了这一想法，并且据此创建了一个手写代码的实现系统。到 1993 年，Haavard 和 Eirik 已经开发出了 Qt 的第一套图形内核程序，并且能够利用它实现他们自己的一些窗口部件。同年末，为了创建“世界上最好的 C++ 图形用户界面框架”，Haavard 提议一起进军商业领域。

1994 年成为两位年轻程序员不幸的一年，他们没有客户，没有资金，只有一个未完成的产品，但是他们希望能够闯进一个稳定的市场。幸运的是，他们的妻子都有工作并且愿意为他们的丈夫提供支持。在这两年里，Haavard 和 Eirik 认为，他们需要继续开发产品并且从中赚得收益。

之所以选择字母“Q”作为类的前缀，是因为该字母在 Haavard 的 Emacs 字体中看起来非常漂亮。随后添加的字母”t” 代表“工具包”(toolkit)，这是从”Xt”——一个 X 工具包的命名方式中获得的灵感。公司于 1994 年 3 月 4 日成立，最初的名字是”Quasar Technologies”，随后更名为”Troll Tech”，而公司今天的名字则是”Trolltech”。

1995 年 4 月，通过 Haavard 就读过的大学的一位教授的联系，挪威的 Metis 公司与他们签订了一份基于 Qt 进行软件开发的合同。大约在同一时间，公司雇佣了 Arnt Gulbrandsen，在公司工作的 6 年时间里，他设计并实现了一套独具特色的文

档系统，并且对 Qt 的代码也做出了不少贡献。

1995 年 5 月 20 日，Qt 0.90 被上传到 sunsite.unc.edu。6 天后，在 comp.os.linux.announce 上发布。这是 Qt 的第一个公开发行版本。Qt 既可以用于 Windows 上的程序开发，又可以用于 UNIX 上的程序开发，而且在这两种平台上，都提供了相同的应用程序编程接口。从第一天起，Qt 就提供了两个版本的软件许可协议：一个是进行商业开发所需的商业许可协议版，另一个则是适用于开源开发的自由软件许可协议版。Metis 的合同确保了公司的发展，然而，在随后长达 10 个月的时间内，再没有任何人购买 Qt 的商业许可协议。

1996 年 3 月，欧洲航天局 (European Space Agency) 购买了 10 份 Qt 的商业许可协议，它成了第二位 Qt 客户。凭着坚定的信念，Eirik 和 Haavard 又雇佣了另外一名开发人员。Qt 0.97 在同年 5 月底正式发布，随后在 1996 年 9 月 24 日，Qt1.0 正式面世。到了这一年的年底，Qt 的版本已经发展到了 1.1，共有来自 8 个不同国家的客户购买了 18 份 Qt 的商业许可协议。也就是在这一年，在 Matthias Ettrich 的带领下，创立了 KDE 项目。

Qt 1.2 于 1997 年 4 月发布。Matthias Ettrich 利用 Qt 建立 KDE 的决定，使 Qt 成为 Linux 环境下开发 C++ 图形用户界面的事实标准。Qt 1.3 于 1997 年 9 月发布。

Matthias 在 1998 年加入公司，并且在当年 9 月，发布了 Qt 1 系列的最后一个版本——V 1.40。1999 年 6 月，Qt 2.0 发布，该版本拥有一个新的开源许可协议——Q 公共许可协议 (QPL，Q Public License)，它与开源的定义一致。1999 年 8 月，Qt 赢得了 LinuxWorld 的最佳库/工具奖。大约在这个时候，Trolltech Pty Ltd（澳大利亚）成立了。

2000 年，公司发布了 Qt/Embedded Linux，它用于 Linux 嵌入式设备。Qt/Em-bedded Linux 提供了自己的窗口系统，并且可以作为 X11 的轻量级替代产品。现在，Qt/X11 和 Qt/Embedded Linux 除了提供商业许可协议之外，还提供了广为使用的 GNU 通用公共许可协议（GPL: General Public License）。2000 年底，成立了 Trolltech Inc.（美国），并发布了 Qtopia 的第一版，它是一个用于移动电话和掌上电脑 (PDA) 的环境平台。Qt/Embedded Linux 在 2001 年和 2002 年两次获得了 LinuxWorld 的”Best Embedded Linux Solution” 奖，Qtopia Phone 也在 2004 年获得了同样的荣誉。

2001 年，Qt 3.0 发布。现在，Qt 已经可用于 Windows、Mac OS X、UNIX 和 Linux（桌面和嵌入式）平台。Qt 3 提供了 42 个新类和超过 500 000 行的代

码。Qt 3 是自 Qt 2 以来前进历程中最为重要的一步，它主要在诸多方面进行了众多改良，包括本地化和统一字符编码标准的支持、全新的文本查看和编辑窗口部件，以及一个类似于 Perl 正则表达式的类等。2002 年，Qt 3 赢得了 Software Development Times 的”Jolt Productivity Award”①。

2005 年夏，Qt 4.0 发布，它大约有 500 个类和 9000 多个函数，Qt 4 比以往的任何一个版本都要全面和丰富，并且它已经裂变成多个函数库，从而使开发人员可以根据自己的需要只连接所需要的 Qt 部分。相对于以前的所有 Qt 版本，Qt 4 的进步是巨大的，它不仅彻底地对高效易用的模板容器、高级的模型/视图功能、快速而灵活的二维绘图框架和强大的统一字符编码标准的文本查看和编辑类进行了大量改进，就更不必说对那些贯穿整个 Qt 类中的成千上万个小的改良了。现如今，Qt 4 具有如此广泛的特性，以至于 Qt 已经超越了作为图形用户界面工具包的界限，逐渐成长为一个成熟的应用程序开发框架。Qt 4 也是第一个能够在其所有可支持的平台上既可用于商业开发又可用于开源开发的 Qt 版本。

同样在 2005 年，公司在北京开设了一家办事处，以便为中国及其销售区域内的用户提供服务和培训，并且为 Qt/Embedded Linux 和 Qtopia 提供技术支持。

通过获取一些非官方的语言绑定件 (language bmdings)，非 C++ 程序员也已早就开始使用 Qt，特别是用于 Python 程序员的 PyQt 语言绑定件。2007 年，公司发布了用于 C# 程序员的非官方语言绑定件 Qyoto。同一年，Qt Jambi 投放市场，它是一个官方支持的 Java 版 Qt 应用程序编程接口。附录 C 提供了对 Qt Jambi 的介绍。

自奇趣科技公司诞生以来，Qt 的声望经久不衰，而且至今依旧持续高涨。取得这样的成绩不但说明了 Qt 的质量，而且也说明了人们都喜欢使用它。在过去的 10 年中，Qt 已经从一个只被少数专业人士所熟悉的“秘密”产品，发展了到如今遍及全世界拥有数以千计的客户和数以万计的开源开发人员的产品。

---

① Jolt 大奖素有“软件业界的奥斯卡”之美誉，共设通用类图书、技术类图书、语言和开发环境、框架库和组件、开发者网站等十余个分类，每个分类设有一个“震撼奖”(Jolt Award) 和三个“生产力奖”(Productivity Award)。一项技术产品只有在获得了 Jolt 奖之后才能真正成为行业的主流，一本技术书籍只有在获得了 Jolt 奖之后才能真正奠定其作为经典的地位。虽然 Jolt 奖项并不起决定作用，但它代表了某种技术趋势与潮流——译者注。

编者的话

感谢作者，感谢原中文翻译者，感谢原中文审校者。

书名修改不是想标新立异，实在是 github 和本地文档编译方便，不支持空格。

感谢汉王 OCR 技术支持，感谢 XeLaTeX。

目 录

I Qt 基础

1 Qt 入门

这一章介绍了如何把基本的 C++ 知识与 Qt 所提供的功能组合起来创建一些简单的图形用户界面（Graphical User Interface，GUI）应用程序。在这一章中，还引入了 Qt 中的两个重要概念：一个是“信号和槽”，另外一个是“布局”。第 2 章还将对它们做进一步的阐述，而第 3 章将着手创建一个具有真正意义的应用程序。

如果你已经熟知 Java 或 C#，但对 C++ 的编程经验还有些欠缺的话，那么在开始阅读本书之前，可能需要先阅读附录 D，它对 C++ 做了简要介绍。

1.1 Hello Qt

我们先从一个非常简单的 Qt 程序开始。首先一行一行地研究这个程序，然后将会看到如何编译并运行它。

```
1  #include <QApplication>
2  #include <QLabel>
3
4  int main(int argc, char *argv[])
5  {
6      QApplication app(argc, argv);
7      QLabel *label = new QLabel("Hello Qt!");
8      label->show();
9      return app.exec();
10 }
```

第 1 行和第 2 行包含了类 QApplication 和 QLabel 的定义。对于每个 Qt 类，都有一个与该类同名（且大写）的头文件，在这个头文件中包括了对该类的定义。

第 6 行创建了一个 QApplication 对象，用来管理整个应用程序所用到的资源。这个 QApplication 构造函数需要两个参数，分别是 argc 和 argv，因为 Qt 支持它自己的一些命令行参数。

第 7 行创建了一个显示“Hello Qt!”的 QLabel 窗口部件 (widget)。在 Qt 和 UNIX 的术语 (tenninology) 中，窗口部件就是用户界面中的一个可视化元素。该词起源于“window gadget”（窗口配件）这两个词，它相当于 Windows 系统术语中的“控件”(control) 和“容器”(container)。按钮、菜单、滚动条和框架都是窗口部件。窗口部件也可以包含其他窗口部件，例如，应用程序的窗口通常就是一个包含了一个 QMenuBar、一些 QToolBar、一个 QStatusBar 以及一些其他窗口部件的窗口部件。绝大多数应用程序都会使用一个 QMainWindow 或者一个 QDialog 来作为它的窗口，但 Qt 是如此灵活，以至于任意窗口部件都可以用作窗口。在本例中，就是用窗口部件 QLabel 作为应用程序的窗口的。

第 8 行使 QLabel 标签 (label) 可见。在创建窗口部件的时候，标签通常都是隐藏的，这就允许我们可以先对其进行设置然后再显示它们，从而避免了窗口部件的闪烁现象。

第 9 行将应用程序的控制权传递给 Qt。此时，程序会进入事件循环状态，这是一种等待模式，程序会等候用户的动作，例如鼠标单击和按键等操作。用户的动作会让可以产生响应的程序生成一些事件（event，也称为“消息”），这里的响应通常就是执行一个或者多个函数。例如，当用户单击窗口部件时，就会产生一个“鼠标按下”事件和一个“鼠标松开”事件。在这方面，图形用户界面应用程序和常规的批处理程序完全不同，后者通常可以在没有人为干预的情况下自行处理输入、生成结果和终止。

为简单起见，我们没有过多关注在 main() 函数末尾处对 QLabel 对象的 delete 操作调用。在如此短小的程序内，这样一点内存泄漏 (memory leak) 问题无关大局，因为在程序结束时，这部分内存是可以由操作系统重新回收的。

图 1.1: Linux 上的 Hello 程序

现在是在机器上测试这个程序的时候了，看起来它应该会如图 1.1 所示。首先需要安装 Qt 4.3.2（或是其后的其他 Qt 4 新发行版），附录 A 对这一安装过程进行

了说明。从现在开始，假定你已经正确地安装了 Qt 4 的一个副本，并且假定已经在 PATH 环境变量中对 Qt 的 bin 目录进行了设置。（在 Windows 操作系统中，这些操作会由 Qt 的安装程序自动完成。）还需要将该程序的源代码保存到 hello.cpp 文件，并把它放进一个名为 hello 的目录中。

现在在命令提示符下，进入 hello 目录，输入如下命令，生成一个与平台无关的项目文件 hello.pro：

```
qmake -project
```

然后，输入如下命令，从这个项目文件生成一个与平台相关的 makefile 文件：

```
qmake hello.pro
```

键入 make 命令就可以构建该程序。（在附录 B 中，会给出 qmake 工具更为详细的说明。）要运行该程序，在 Windows 下可以输入`hello`，在 UNIX 下可以输入`./hello`，在 Mac OS X 下可以输入`open hello.app`。要结束该程序，可直接单击窗口标题栏上的关闭按钮。

如果使用的是 Windows 系统，并且已经安装了 Qt 的开源版和 MinGW 编译器，那么将会看到一个指向 MS-DOS 提示符窗口的快捷键，其中已经正确地创建了使用 Qt 时所需的全部环境变量。如果启动了这个窗口，那么就可以在里面像上面所讲述的那样使用 qmake 命令和 make 命令编译 Qt 应用程序。而由此产生的可执行文件将会保存在应用程序所在目录的 debug 或 release 文件夹中。

如果使用的是 Microsoft Visual C++ 和商业版的 Qt，则需要用 nmake 命令代替 make 命令。除了这一方法外，还可以通过 hello.pro 文件创建一个 Visual Studio 的工程文件，此时需要输入命令：

```
qmake -tp vc hello.pro
```

然后就可以在 Visual Studio 中编译这个程序了。如果使用的是 Mac OS X 系统中的 Xcode，那么可以使用如下命令来生成一个 Xcode 工程文件：

```
qmake -spec macx-xcode hello.pro
```

在开始进入下一个例子之前，我们一起来做一件有意思的事情：将代码行

```
QLabel *label = new QLabel("Hello Qt!");
```

替换为

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

然后重新编译该程序。运行程序时，看起来应当是图 1.2 的样子。正如该例子所显示的那样，通过使用一些简单的 HTML 样式格式，就可以轻松地把 Qt 应用程序的用户接口变得更为丰富多彩。

图 1.2: 具有简单 HTML 样式的标签

1.2 建立连接

第二个例子要说明的是如何响应用户的动作。这个应用程序由一个按钮构成，用户可以单击这个按钮退出程序。除了应用程序的主窗口部件使用的是 QPushButton 而不是 QLabel 之外，这个应用程序的源代码和 Hello 程序的源代码非常相似。同时，我们还会将用户的一个动作（单击按钮）与一段代码连接起来。

这个应用程序的源代码位于本书的例子文件中，文件名是 qt4-book/chap01/quit/quit.cpp。程序的运行效果如图 1.3 所示。以下是该文件所包含的内容：

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()),
                     &app, SLOT(quit()));
    button->show();
    return app.exec();
}
```

图 1.3: Quit 应用程序

Qt 的窗口部件通过发射信号 (signal) 来表明一个用户动作已经发生了或者是一个状态已经改变了[①]。例如，当用户单击 QPushButton 时；该按钮就会发射一个 clicked() 信号。信号可以与函数（在这里称为槽，slot）相连接，以便在发射信号时，槽可以得到自动执行。在这个例子中，我们把这个按钮的 clicked() 信号与 QApplication 对象的 quit() 槽连接起来。宏 SIGNAL() 和 SLOT() 是 Qt 语法中的一部分。

现在来构建这个应用程序。假设已经创建了一个包含 quit.cpp 文件的 quit 目录。在 quit 目录中，首先运行 qmake 命令生成它的工程文件，然后再次运行该命令来生成一个 makefile 文件，这两项操作的命令如下：

```
qmake -project
qmake quit.pro
```

① Qt 的信号和 UNIX 的信号并不相关，本书中所讨论的信号仅指 Qt 信号。

现在，就可以编译并运行这个应用程序了。如果单击 Quit 按钮，或者按下了空格键（这样也会按下 Quit 按钮），那么将会退出应用程序。

1.3 窗口部件的布局

这一节将创建一个简单的例子程序，以说明如何用布局 (layout) 来管理窗口中窗口部件的几何形状，还要说明如何利用信号和槽来同步窗口部件。这个应用程序的运行效果如图 1.4 所示，它可以用来询问用户的年龄，而用户可以通过操纵微调框 (spin box) 或者滑块 (slider) 来完成年龄的输入。

图 1.4: Age 应用程序

这个应用程序由三个窗口部件组成：一个 QSpinBox，一个 QSlider 和一个 QWidget。QWidget 是这个应用程序的主窗口。QSpinBox 和 QSlider 会显示在 QWidget 中，它们都是 QWidget 窗口部件的子对象。换言之，QWidget 窗口部件是 QSpinBox 和 QSlider 的父对象。QWidget 窗口部件自己则没有父对象，因为程序是把它当作顶层窗口的。QWidget 的构造函数以及它的所有子类都会带一个参数 QWidget *，以用来说明谁是它们的父窗口部件。

以下是本应用程序的源代码：

```
1 #include <QApplication>
2 #include <QHBoxLayout>
3 #include <QSlider>
4 #include <QSpinBox>
5
```

```
6  int main(int argc, char *argv[])
7  {
8      QApplication app(argc, argv);
9
10     QWidget *window = new QWidget;
11     window->setWindowTitle("Enter Your Age");
12
13     QSpinBox *spinBox = new QSpinBox;
14     QSlider *slider = new QSlider(Qt::Horizontal);
15     spinBox->setRange(0, 130);
16     slider->setRange(0, 130);
17
18     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
19                      slider, SLOT(setValue(int)));
20     QObject::connect(slider, SIGNAL(valueChanged(int)),
21                      spinBox, SLOT(setValue(int)));
22     spinBox->setValue(35);
23
24     QHBoxLayout *layout = new QHBoxLayout;
25     layout->addWidget(spinBox);
26     layout->addWidget(slider);
27     window->setLayout(layout);
28
29     window->show();
30
31     return app.exec();
32 }
```

第 10 行和第 11 行创建了 QWidget 对象，并把它作为应用程序的主窗口。我们通过调用 setWindowTitle() 函数来设置显示在窗口标题栏上的文字。

第 13 行和第 14 行分别创建了一个 QSpinBox 和一个 QSlider，并分别在第 15 行和第 16 行设置了它们的有效范围。我们可以放心地假定用户的最大年龄不会超

过 130 岁。本应把这个窗口传递给 QSpinBox 和 QSlider 的构造函数，以说明这两个窗口部件的父对象都是这个窗口，但在这里没有这个必要，因为布局系统将会自行得出这一结果并自动把该窗口设置为微调框和滑块的父对象，下面将会很快看到这一点。

从第 18 行到第 21 行，调用了两次 QObject::connect()，这是为了确保能够让微调框和滑块同步，以便它们两个总是可以显示相同的数值。一旦有一个窗口部件的值发生了改变，那么就会发射它的 valueChanged(int) 信号，而另一个窗口部件就会用这个新值调用它的 setValue(int) 槽。

第 22 行将微调框的值设置为 35。当发生这种情况时，QSpinBox 就会发射 valueChanged(int) 信号，其中，int 参数的值是 35。这个参数会被传递给 QSlider 的 setValue(int) 槽，它会把这个滑块的值设置为 35。于是，滑块就会发射 valueChanged(int) 信号，因为它的值发生了变化，这样就触发了微调框的 setValue(int) 槽。但在这一点上，setValue(int) 并不会再发射任何信号，因为微调框的值已经是 35 了。这样就可以避免无限循环的发生。图 1.5 对这种情况进行了图示概述。

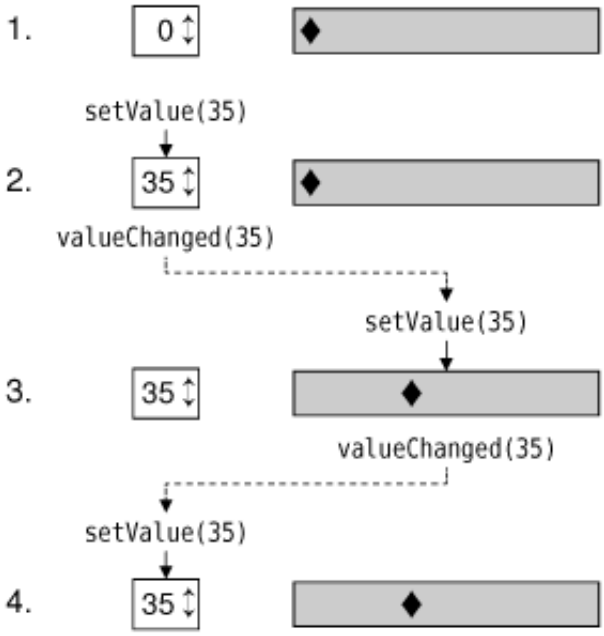


图 1.5: 改变一个窗口部件的值会使两个窗口部件都发生变化

窗口部件的风格

到目前为止，我们看到的这些屏幕截图都来自于 Linux，但是 Qt 应用程序在每一个所支持的平台上都可以看起来像本地程序一样（见图 1.6）。Qt 是通过所模拟平台的视觉外观来实现这一点的，而不是对某个特殊平台的封装或者一个工具包中的窗口部件集。

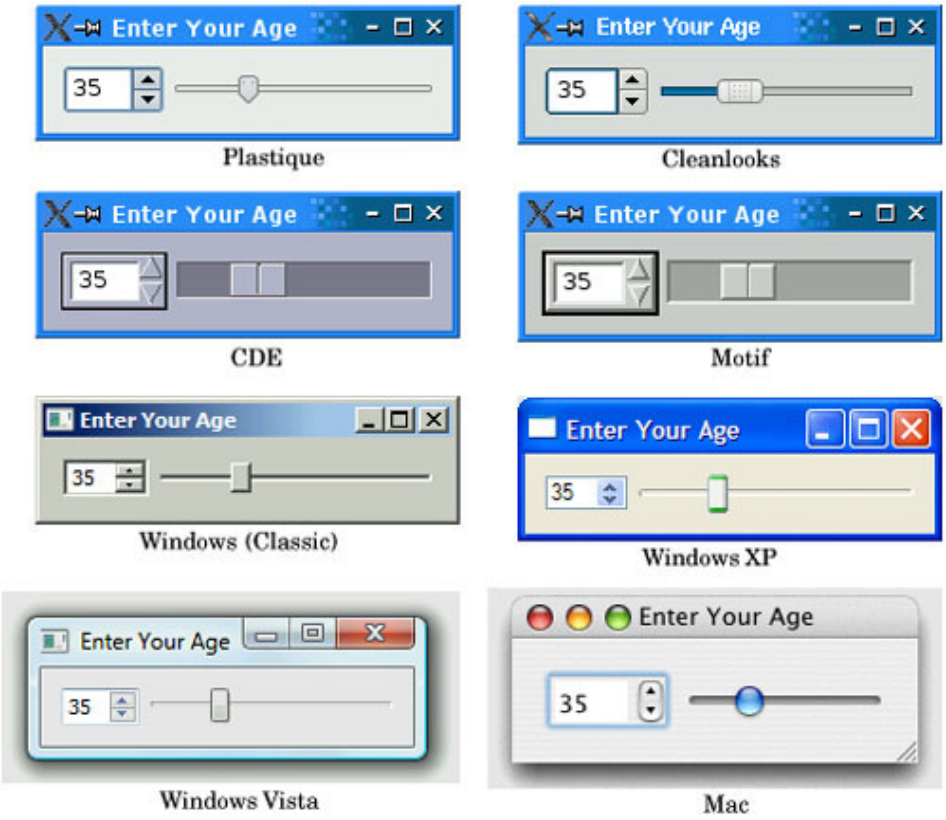


图 1.6: 一些预定义风格

运行于 KDE 下的 Qt/X11 应用程序的默认风格是 Plastique，而运行于 GNOME 下的应用程序的默认风格是 Cleanlooks。这些风格使用了渐变和抗锯齿效果，以用来提供一种时尚的外观。运行 Qt 应用程序的用户可以通过使用命令行参数 -style 覆盖原有的默认风格。例如，在 X11 下，要想使用 Motif 风格来运行 Age 应用程序，只需简单输入以下命令即可：

```
./age -style motif
```

与其他风格不同，Windows XP、Windows Vista 和 Mac 的风格只能在它们的本地平台上有效，因为它们需要依赖平台的主题引擎。

还有另外一种风格 QtDotNet，它来自于 Qt Solutions 模块。创建自定义风格也是可能的，这会在第 19 章中加以阐述。

在源程序的第 24 行到第 27 行，使用了一个布局管理器对微调框和滑块进行布局处理。布局管理器 (layout manager) 就是一个能够对其所负责窗口部件的尺寸大小和位置进行设置的对象。Qt 有三个主要的布局管理器类：

- QHBoxLayout。在水平方向上排列窗口部件，从左到右（在某些文化中则是从右向左）。
- QVBoxLayout。在竖直方向上排列窗口部件，从上到下。
- QGridLayout。把各个窗口部件排列在一个网格中。

第 27 行的 QWidget::setlayout() 函数调用会在窗口上安装该布局管理器（见图 1.7）。从软件的底层实现来说，QSpinBox 和 QSlider 会自动“重定义父对象”，它们会成为这个安装了布局的窗口部件的子对象。也正是基于这个原因，当创建一个需要放进某个布局中的窗口部件时，就没有必要为其显式地指定父对象了。

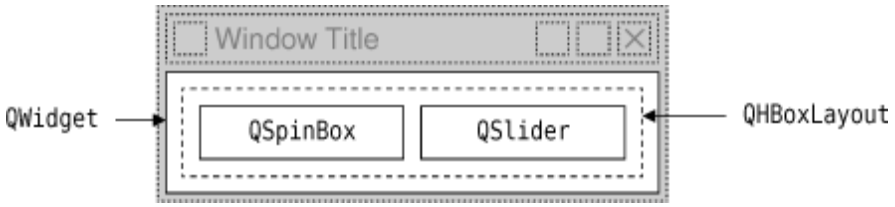


图 1.7: Age 应用程序中的窗口部件和布局

尽管没有明确地设置任何一个窗口部件的位置或大小，但 QSpinBox 和 QSlider 还是能够非常好看地一个挨着一个显示出来。这是因为 QHBoxLayout 可根据所负责的子对象的需要为它们分配所需的位置和大小。布局管理器使我们从应用程序的各种屏幕位置关系指定的繁杂纷扰中解脱出来，并且它还可以确保窗口尺寸大小发生改变时的平稳性。

Qt 中构建用户接口的方法很容易理解并且非常灵活。Qt 程序员最常使用的方式是先声明所需的窗口部件，然后再设置它们所应具备的属性。程序员把这些窗体部件添加到布局中，布局会自动设置它们的位置和大小。利用 Qt 的信号－槽机制，并通过窗口部件之间的连接就可以管理用户的交互行为。

1.4 使用参考文档

由于 Qt 的参考文档涉及了 Qt 中的每一个类和函数，所以对任何一名 Qt 开发人员来说，它都是一个基本工具。本书讲述了 Qt 的许多类和函数，但是也并不能完全覆盖到 Qt 中所有的类和函数，同时也无法对书中所涉及的每个类和函数都提供全部的细节。如果想尽可能多地从 Qt 获益，那么就应当尽快地达到对 Qt 参考文档了如指掌的程度。

在 Qt 的 doc/html 目录下可以找到 HTML 格式的参考文档，并且可以使用任何一种 Web 浏览器来阅读它。也可以使用 Qt 的帮助浏览器 Qt Assistant，它具有强大的查询和索引功能，使用时能够比 Web 浏览器更加快速和容易。

要运行 Qt Assistant，在 Windows 下，可单击“开始”菜单中的”Qt by Trolltech v4.x.y | Assistant”(见图 1.8)；在 UNIX 下，可在命令行终端中输入 assistant 命令；在 Mac OS X Finder 中，只需双击 assistant 即可，在主页的”API Reference” 小节中的链接提供了浏览 Qt 类的几种不同方式，”All Classes” 页面列表会列出 Qt API 中的每一个类，而”Main Classes” 页面列表只会列出 Qt 中那些最为常用的类。作为练习，你或许可以去试着查询一下这一章中所使用过的那些类和函数。

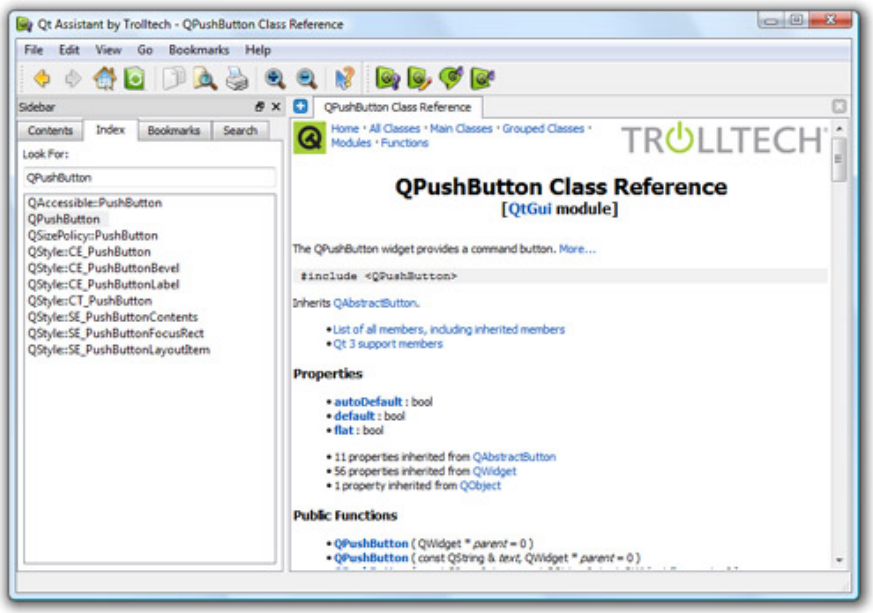


图 1.8: Windows Vista 下 Qt Assistant 中的 Qt 参考文档

需要注意的是，通过继承而得到的函数的文档会显示在它的基类中，例如，QPushButton 就没有它自己的 show() 函数，因为它是从 QWidget 那里继承的函

数。图 1.9 给出了到目前为止我们所见过的各个类之间的关系。

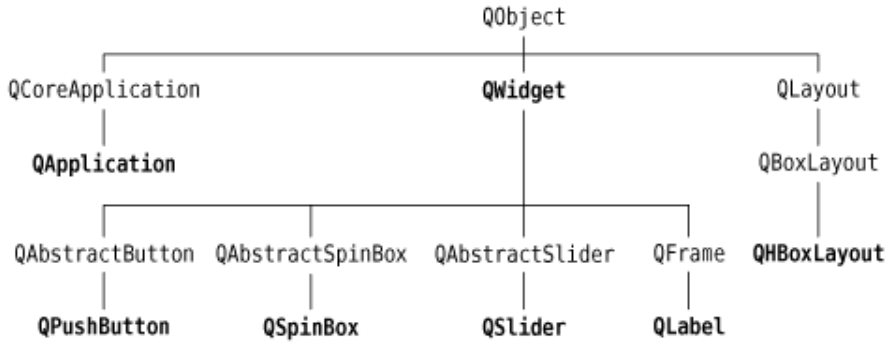


图 1.9: 目前为止我们所见过的那些 Qt 类的继承树

可以从http://doc.trolltech.com中获取 Qt 的当前版和一些早期版本的在线参考文档。这个网站也选摘了 Qt 季刊 (Qt Quarterly) 中的一些文章。Qt 季刊是 Qt 程序员的时事通讯，会发送给所有获得 Qt 商业许可协议的人员。

本章介绍了一些重要概念：信号－槽连接和布局，也逐步展示了 Qt 的兼容性和 Qt 完全面向对象的构建方法和窗口部件的使用。如果你浏览了一遍 Qt 的参考文档，那么将会发现一种如何学习使用新窗口部件的统一方法，并且也将发现 Qt 对函数、参数、枚举等变量选名的严谨性，以及在使用 Qt 编程时令人叹服的愉悦感和舒适性。

本书第一部分的随后几章，都建立在本章的基础之上，它们演示了如何创建一个完整的 GUI 应用程序——拥有菜单、工具栏、文档窗口、状态条和对话框，还有与之相应的用于阅读、处理和输出文件的底层功能函数。

2 创建对话框

这一章讲解如何使用 Qt 创建对话框。对话框为用户提供了许多选项和多种选择，允许用户把选项设置为他们喜欢的变量值并从中做出选择。之所以把它们称为对话框，或者简称为“对话”，是因为它们为用户和应用程序之间提供了一种可以相互“交谈”的交互方式。

绝大多数图形用户界面应用程序都带有一个由菜单栏、工具栏构成的主窗口以及几十个对主窗口进行补充的对话框。当然，也可以创建对话框应用程序，它可以通过执行合适的动作来直接响应用户的选择（例如，一个计算器应用程序）。

本章将首先完全用手写代码的方式创建第一个对话框，以便能够说明是如何完成这项工程的。然后将使用 Qt 的可视化界面设计工具 Qt 设计师 (Qt Designer)。使用 Qt 设计师比手写代码要快得多，并且可以使不同的设计测试工作以及稍后对设计的修改工作变得异常轻松。

2.1 子类化 QDialog

第一个例子是完全使用 C++ 编写的一个 Find(查找) 对话框，它的运行效果如图 2.1 所示，这将实现一个拥有自主权的对话框。通过这一过程，就可以让对话框拥有自己的信号和槽，成为一个独立的、完备的控件。

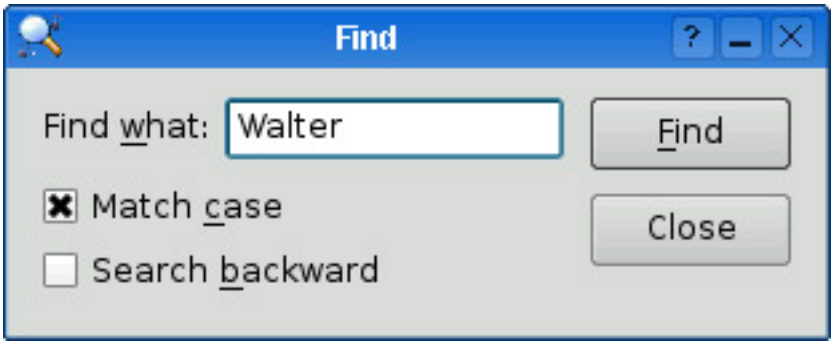


图 2.1：Find 对话框

源代码分别保存在 finddialog.h 和 finddialog.cpp 文件中。首先从 finddialog.h 文件说起：

```
1  #ifndef FINDDIALOG_H
2  #define FINDDIALOG_H
3
4  #include <QDialog>
5
6  class QCheckBox;
7  class QLabel;
8  class QLineEdit;
9  class QPushButton;
```

第 1、2 行（以及后面的第 27 行，见下页）能够防止对这个头文件的多重包含。第 4 行包含了 QDialog 的定义，它是 Qt 中对话框的基类。QDialog 从 QWidget 类中派生出来。第 6 行到第 9 行前置声明了一些用于这个对话框实现中的 Qt 类。前置声明 (forward declaration) 会告诉 C++ 编译程序类的存在，而不用提供类定义中的所有细节 (通常放在它自己的头文件中)。关于这一点，将会再简单地多讲一些。

接下来定义 FindDialog，并让它成为 QDialog 的子类：

```
11 class FindDialog : public QDialog
12 {
```

```
13      Q_OBJECT
14
15  public:
16      FindDialog(QWidget *parent = 0);
```

对于所有定义了信号和槽的类，在类定义开始处的 Q_OBJECT 宏都是必需的。

FindDialog 的构造函数就是一个典型的 Qt 窗口部件类的定义方式。parent 参数指定了它的父窗口部件。该参数的默认值是一个空指针，意味着该对话框没有父对象。

```
18  signals:
19      void findNext(const QString &str, Qt::CaseSensitivity cs);
20      void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

signals 部分声明了当用户单击 Find 按钮时对话框所发射的两个信号。如果向前查询 (search backward) 选项生效，对话框就发射 findPrevious() 信号，否则它就发射 findNext() 信号。

signals 关键字实际上是一个宏。C++ 预处理器会在编译程序找到它之前把它转换成标准 C++ 代码。Qt::CaseSensitivity 是一个枚举类型，它有 Qt::CaseSensitive 和 Qt::CaseInsensitive 两个取值。

```
22  private slots:
23      void findClicked();
24      void enableFindButton(const QString &text);
25
26  private:
27      QLabel *label;
28      QLineEdit *lineEdit;
29      QCheckBox *caseCheckBox;
30      QCheckBox *backwardCheckBox;
```

```
31      QPushButton *findButton;
32      QPushButton *closeButton;
33  };
34
35  #endif
```

在这个类的 private 段声明了两个槽。为了实现这两个槽，几乎需要访问这个对话框的所有子窗口部件，所以也保留了指向它们的指针。关键字 slots 就像 signals 一样也是一个宏，也可以扩展成 C++ 编译程序可以处理的一种结构形式。

对于这些私有变量，我们使用了它们的类前置声明。这是可行的，因为它们都是指针，而且没有必要在头文件中就去访问它们，因而编译程序就无须这些类的完整定义。我们没有包含与这几个类相关的头文件（<QCheckBox>、<QLabel>，等等），而是使用了一些前置声明。这可以使编译过程更快一些。

现在看一下 finddialog.cpp，其中包含了对 FindDialog 类的实现：

```
1  #include <QtGui>
2
3  #include "finddialog.h"
```

首先，需要包含 <QtGui>，该头文件包含了 Qt Gui 类的定义。Qt 由数个模块构成，每个模块都有自己的类库。最为重要的模块有 QtCore、QtGui、QtNetwork、QtOpenGL、QtScript、QtSql、QtSvg 和 QtXml。其中，在 <QtGui> 头文件中为构成 QtCore 和 QtGui 组成部分的所有类进行了定义。在程序中包含这个头文件，就能够使我们省去在每个类中分别包含的麻烦。

在 filedialog.h 文件中，本可以仅简单地添加一个 <QtGui> 包含即可，而不用包含 <QDialog> 和使用 QCheckBox、QLabel、QLineEdit 和 QPushButton 的前置声明。然而，在一个头文件中再包含一个那么大的头文件着实不是一种好的编程风格，尤其对于比较大的工程项目更是如此。

```
5  FindDialog::FindDialog(QWidget *parent)
6      : QDialog(parent)
7  {
8      label = new QLabel(tr("Find &what:"));
9      lineEdit = new QLineEdit;
10     label->setBuddy(lineEdit);
11
12     caseCheckBox = new QCheckBox(tr("Match &case"));
13     backwardCheckBox = new QCheckBox(tr("Search &backward"));
14
15     findButton = new QPushButton(tr("&Find"));
16     findButton->setDefault(true);
17     findButton->setEnabled(false);
18
19     closeButton = new QPushButton(tr("Close"));
```

在第 6 行，把 parent 参数传递给了基类的构造函数。然后，创建了子窗口部件。在字符串周围的 tr() 函数调用是把它们翻译成其他语言的标记。在每个 QObject 对象以及包含有 Q_OBJECT 宏的子类中都有这个函数的声明。尽管也许并没有将你的应用程序立刻翻译成其他语言的打算，但是在每一个用户可见的字符串周围使用 tr() 函数还是一个很不错的习惯。在第 18 章中将对翻译 Qt 应用程序进行详细讲述。

在这些字符串中，使用了表示“与”操作的符号”&” 来表示快捷键。例如，第 15 行创建了一个 Find 按钮，用户可在那些支持快捷键的平台下通过按下 Alt+F 快捷键来激活它。符号”&” 可以用来控制焦点：在第 8 行创建了一个带有快捷键 (Alt+W) 的标签，而在第 10 行设置了行编辑器作为标签的伙伴。所谓“伙伴” (buddy) 就是一个窗口部件，它可以在按下标签的快捷键时接收焦点 (focus)。所以当用户按下 Alt+W（该标签的快捷键）时，焦点就会移动到这个行编辑器（该标签的伙伴）上。

在第 16 行，通过调用 setDefault(true) 让 Find 按钮成为对话框的默认按钮。默认按钮 (default button) 就是当用户按下 Enter 键时能够按下对应的按钮。在第 17 行，禁用了 Find 按钮。当禁用一个窗口部件时，它通常会显示为灰色，并且不能和用户发生交互操作。

```
21      connect(lineEdit, SIGNAL(textChanged(const QString &)),
22              this, SLOT(enableFindButton(const QString &)));
23      connect(findButton, SIGNAL(clicked()),
24              this, SLOT(findClicked()));
25      connect(closeButton, SIGNAL(clicked()),
26              this, SLOT(close()));
```

只要行编辑器中的文本发生变化，就会调用私有槽 enableFindButton(const QString &)。当用户单击 Find 按钮时，会调用 findClicked() 私有槽。而当用户单击 Close 时，对话框会关闭。close() 槽是从 QWidget 中继承而来的，并且它的默认行为就是把窗口部件从用户的视野中、隐藏起来（而无须将其删除）。稍后将会看到 enableFindButton() 槽和 findClicked() 槽的代码。

由于 QObject 是 FindDialog 的父对象之一，所以可以省略 connect() 函数前面的 QObject:: 前缀。

```
28      QHBoxLayout *topLeftLayout = new QHBoxLayout;
29      topLeftLayout->addWidget(label);
30      topLeftLayout->addWidget(lineEdit);
31
32      QVBoxLayout *leftLayout = new QVBoxLayout;
33      leftLayout->addLayout(topLeftLayout);
34      leftLayout->addWidget(caseCheckBox);
35      leftLayout->addWidget(backwardCheckBox);
36
37      QVBoxLayout *rightLayout = new QVBoxLayout;
38      rightLayout->addWidget(findButton);
39      rightLayout->addWidget(closeButton);
40      rightLayout->addStretch();
41
42      QHBoxLayout *mainLayout = new QHBoxLayout;
43      mainLayout->addLayout(leftLayout);
```

```
44      mainLayout->addLayout(rightLayout);
45      setLayout(mainLayout);
```

接下来，使用布局管理器摆放这些子窗口部件。布局中既可以包含多个窗口部件，也可以包含其他子布局。通过 QHBoxLayout、QVBoxLayout 和 QGridLayout 这三个布局的不同嵌套组合，就可能构建出相当复杂的对话框。

如图 2.2 所示，对于 Find 对话框，使用了两个 QHBoxLayout 布局和两个 QVBoxLayout 布局。外面的布局是主布局，通过第 45 行代码将其安装在 FindDialog 中，并且由其负责对话框的整个区域。其他三个布局则作为子布局。图 2.2 中右下角的小“弹簧”是一个分隔符（或者称为“伸展器”）。用它来占据 Find 按钮和 Close 按钮所余下的空白区域，这样可以确保这些按钮完全占用它们所在布局的上部空间。

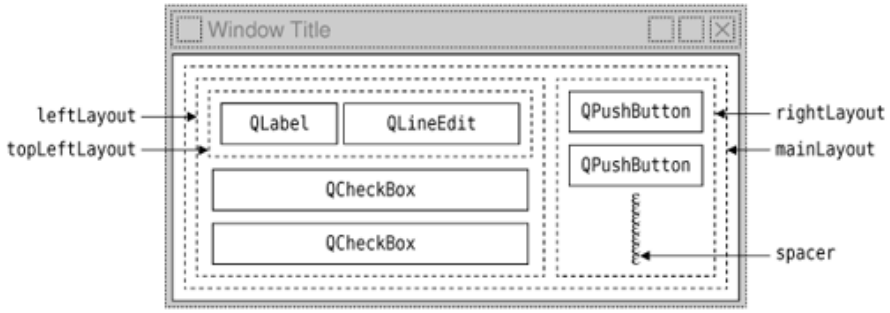


图 2.2: Find 对话框的布局

布局管理器类的一个精妙之处在于它们不是窗口部件。相反，它们派生自 QLayout，因而也就是进一步派生自 QObject，在图 2.2 中，窗口部件用实线轮廓来表示，布局用点线来表示，这样就能够很好地区分窗口部件和布局。在一个运行的应用程序中，布局是不可见的。

当将子布局对象添加到父布局对象中时（第 33、43 和 44 行），子布局对象就会自动重定义自己的父对象。也就是说，当将主布局装到对话框中去时（第 45 行），它就会成为对话框的子对象了，于是它的所有子窗口部件都会重定义自己的父对象，从而变成对话框中的子对象。图 2.3 给出了父子层次关系的最终结果。

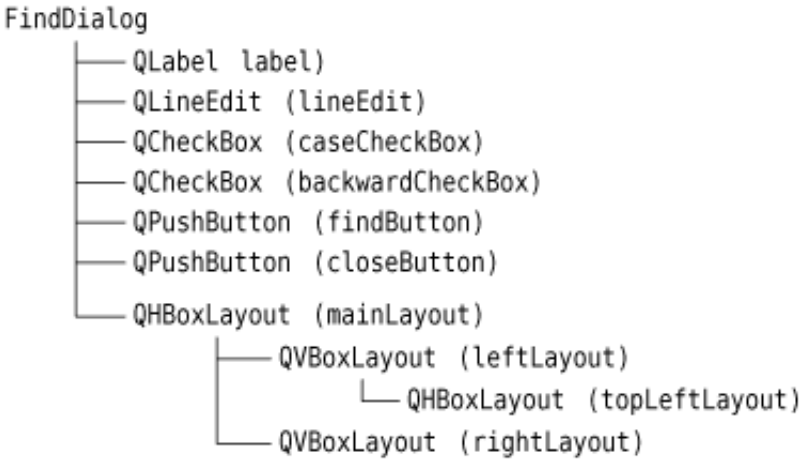


图 2.3: Find 对话框中的父子关系

```
47      setWindowTitle(tr("Find"));
48      setFixedHeight(sizeHint().height());
49  }
```

最后，设置了显示在对话框标题栏上的标题的内容，并让窗口具有一个固定的高度，这是因为在对话框的垂直方向上再没有其他窗口部件可以去占用所多出的空间了。QWidget::sizeHint() 函数可以返回一个窗口部件所“理想”的尺寸大小。

这样，就完成了对 FindDialog 对话框构造函数的分析。由于在创建这个对话框中的窗口部件和布局时使用的是 new，所以需要写一个能够调用 delete 的析构函数，以便可以删除所创建的每一个窗口部件和布局。但是这样做并不是必需的，因为 Qt 会在删除父对象的时候自动删除其所属的所有子对象，也就会删除 FindDialog 中作为其子孙的所有子窗口部件和子布局。

现在来看一下这个对话框中所用到的槽：

```
51  void FindDialog::findClicked()
52  {
53      QString text = lineEdit->text();
54      Qt::CaseSensitivity cs =
55              caseCheckBox->isChecked() ? Qt::CaseSensitive
```

```
56                                      : Qt::CaseInsensitive;
57     if (backwardCheckBox->isChecked()) {
58         emit findPrevious(text, cs);
59     } else {
60         emit findNext(text, cs);
61     }
62 }
63
64 void FindDialog::enableFindButton(const QString &text)
65 {
66     findButton->setEnabled(!text.isEmpty());
67 }
```

当用户单击 Find 按钮时，就会调用 findClicked() 槽。而该槽将会发射 findPrevious() 或 findNext() 信号，这取决于 Search backward 选项的取值。emit 是 Qt 中的关键字，像其他 Qt 扩展一样，它也会被 C++ 预处理器转换成标准的 C++ 代码。

只要用户改变了行编辑器中的文本，就会调用 enableFindButton() 槽。如果在行编辑器中有文本，该槽就会启用 Find 按钮，否则它就会禁用 Find 按钮。

利用这两个槽就完成了这个对话框的功能。现在，可以创建一个 main.cpp 文件来测试一下这个 FindDialog 窗口部件。

```
1 #include <QApplication>
2
3 #include "finddialog.h"
4
5 int main(int argc, char *argv[])
6 {
7     QApplication app(argc, argv);
8     FindDialog *dialog = new FindDialog;
9     dialog->show();
```

```
10      return app.exec();
11  }
```

为了编译这个程序，还像以前一样运行 qmake。由于 FindDialog 类的定义包含 Q_OBJECT 宏，因而由 qmake 生成的 makefile 将会自动包含一些运行 moc 的规则，moc 就是指 Qt 的元对象编译器，即 meta-object compiler。（会在下一节介绍 Qt 的元对象系统。）

为了使 moc 能够正常运行，必须把类定义从实现文件中分离出来并放到头文件中。由 moc 生成的代码会包含这个头文件，并且会添加一些特殊的 C++ 代码。

必须对使用了 Q_OBJECT 宏的类运行 moc。因为 qmake 会自动在 makefile 中添加这些必要的规则，所以这并不成问题。但是如果忘记了使用 qmake 重新生成 makefile 文件，并且也没有重新运行 moc，那么连接程序就会报错，指出你声明了一些函数但是却没有实现它们。这些信息可能是相当不明确的。GCC 会生成像这样的出错信息：

```
finddialog.o: In function ‘FindDialog::tr(char const*, char const*)’:
/usr/lib/qt/src/corelib/global/qglobal.h:1430: undefined reference to
‘FindDialog::staticMetaObject’
```

Visual C++ 输出的出错信息可能是这样的：

```
finddialog.obj : error LNK2001: unresolved external symbol
”public:~virtual int __thiscall MyClass::qt_metacall(enum QMetaObject
::Call,int,void * *)”
```

如果曾经遇到过这种情况，那么请重新运行 qmake 以生成新的 makefile 文件，然后再重新构建该应用程序。

现在来运行该程序。如果在你的系统上能够显示快捷键，那么可以检验一下快捷键 Alt+w、Alt+C、Alt+B 和 Alt+F 是不是触发了正确的行为。可以通过敲击键盘上的 Tab 键来遍历这些窗口部件。默认的 Tab 键顺序就是创建窗口部件时的顺序。要改变这个键顺序，可以使用 QWidget::setTabOrder() 函数。

提供一种合理的 Tab 键顺序和键盘快捷键可以确保不愿（或者不能）使用鼠标的用户能够充分享受应用程序所提供的全部功能。完全通过键盘控制应用程序也深受快速输入人员的赞赏。

在第 3 章，将在一个真实的应用程序中使用 Find 对话框，并且将会把 findPrevious() 信号和 findNext() 信号与一些槽连接到一起。

2.2 深入介绍信号和槽

信号和槽机制是 Qt 编程的基础。它可以让应用程序编程人员把这些互不了解的对象绑定在一起。前面，已经把一些信号和槽连接在了一起，也声明了自己的信号和槽，还实现了自己的槽，并且还发射了自己的信号。让我们再花一点时间，来进一步深入地了解这个机制。

槽和普通的 C++ 成员函数几乎是一样的——可以是虚函数；可以被重载；可以是公有的、保护的或者私有的，并且也可以被其他 C++ 成员函数直接调用；还有，它们的参数可以是任意类型。唯一的不同是：槽还可以和信号连接在一起，在这种情况下，每当发射这个信号的时候，就会自动调用这个槽。

connect() 语句看起来会是如下的样子：

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

这里的 *sender* 和 *receiver* 是指向 QObject 的指针，*signal* 和 *slot* 是不带参数的函数名。实际上，SIGNAL() 宏和 SLOT() 宏会把它们的参数转换成相应的字符串。

到目前为止，在已经看到的实例中，我们已经把不同的信号和不同的槽连接在了一起。但这里还需要考虑一些其他的可能性。

- **一个信号可以连接多个槽**：

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
```

```
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

在发射这个信号的时候，会以不确定的顺序一个接一个地调用这些槽。

- **多个信号可以连接同一个槽**：

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

无论发射的是哪一个信号，都会调用这个槽。

- **一个信号可以与另外一个信号相连接**：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

当发射个信号时，也会发射第二个信号。除此之外，信号与信号之间的连接和信号与槽之间的连接是难以区分的。

- **连接可以被移除**：

```
disconnect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
```

这种情况较少用到，因为当删除对象时，Qt 会自动移除和这个对象相关的所有连接。

要把信号成功连接到槽（或者连接到另外一个信号），它们的参数必须具有相同的顺序和相同的类型：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

例外地，如果信号的参数比它所连接的槽的参数多，那么多余的参数将会被简单地忽略掉：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

如果参数类型不匹配，或者如果信号或槽不存在，则当应用程序使用调试模式构建后，Qt 会在运行时发出警告。与之相类似的是，如果在信号和槽的名字中包含了参数名，Qt 也会发出警告。

到现在为止，我们仅仅在窗口部件之间使用了信号和槽。但是这种机制本身是在 QObject 中实现的，并不只局限于图形用户界面编程中。这种机制可以用于任何 QObject 的子类中：

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }
public slots:
    void setSalary(int newSalary);
signals:
    void salaryChanged(int newSalary);
private:
    int mySalary;
};
void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

注意一下 setSalary() 槽是如何工作的。只有在 newSalary != mySalary 的时候，才发射 salaryChanged() 信号。这样可以确保循环连接不会导致无限循环。

Qt 的元对象系统

Qt 的主要成就之一就是使用了一种机制对 C++ 进行了扩展，并且使用这种机制创建了独立的软件组件。这些组件可以绑定在一起，但任何一个组件对于它所要连接的组件的情况事先都一无所知。

这种机制称为元对象系统 (meta-object system)，它提供了关键的两项技术：信号－槽以及内省 (introspection)。内省功能对于实现信号和槽是必需的，并且允许应用程序的开发人员在运行时获得有关 QObject 子类的“元信息”(meta-information)，包括一个含有对象的类名以及它所支持的信号和槽的列表。这一机制也支持属性（广泛用于 Qt 设计师中）和文本翻译（用于国际化），并且它也为 QtScript 模块奠定了基础。从 Qt 4.2 开始，可以动态添加属性，这一特性将会在第 19 章和第 22 章中付诸实施。

标准 C++ 没有对 Qt 的元对象系统所需要的动态元信息提供支持。Qt 通过提供一个独立的 moc 工具解决了这个问题，moc 解析 Q_OBJECT 类的定义并且通过 C++ 函数来提供可供使用的信息。由于 moc 使用纯 C++ 来实现它的所有功能，所以 Qt 的元对象系统可以在任意 C++ 编译器上工作。

这一机制是这样工作的：

- Q_OBJECT 宏声明了在每一个 QObject 子类中必须实现的一些内省函数：metaObject()、tr()、qt_metacall()，以及其他一些函数。
- Qt 的 moc 工具生成了用于由 Q_OBJECT 声明的所有函数和所有信号的实现。
- 像 connect() 和 disconnect() 这样的 QObject 的成员函数使用这些内省函数来完成它们的工作。

由于所有这些工作都是由 qmake、moc 和 QObject 自动处理的，所以很少需要再去考虑这些事情。但是如果你对此充满好奇心的话，那么也可以阅读一下有

关 QMetaObject 类的文档和由 moc 生成的 C++ 源代码文件，可以从中看出这些实现工作是如何进行的。

2.3 快速设计对话框

Qt 的设计初衷就是为了能够直观并且友好地进行手工编码，并且对于程序员来说，纯粹通过编写 C++ 源代码来开发整个 Qt 应用程序并不稀奇。尽管如此，许多程序员还是喜欢使用可视化的方法来设计窗体，因为他们发现使用可视化方式会比手工编码显得更自然、更快速，并且也希望能够通过可视化方法，对那些手工编码所设计的窗体，进行更快速、更容易的测试和修改。

Qt 设计师 (Qt Designer) 为程序员们提供了可供使用的新选择，它提供一种可视化的设计能力。Qt 设计师可用于开发应用程序中的所有或部分窗体。使用 Qt 设计师所创建的窗体最终仍旧是 C++ 代码，因此，可把 Qt 设计师看作是一个传统的工具集，并且不会对编译器强加其他特殊要求。

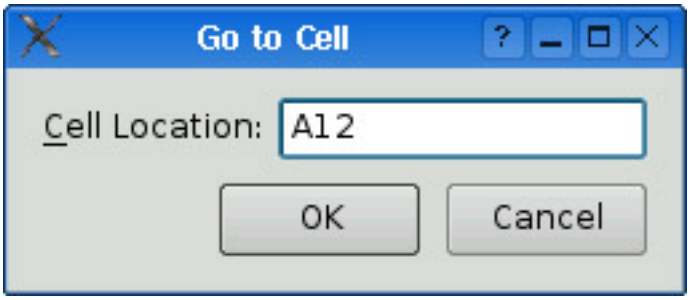


图 2.4: Go to Cell 对话框

在这一节，将使用 Qt 设计师来创建如图 2.4 所示的 Go to Cell 对话框。并且无论是使用手工编码还是使用 Qt 设计师，在创建对话框时总是要包含以下这几个相同的基本步骤:

1. 创建并初始化子窗口部件。
2. 把子窗口部件放到布局中。
3. 设置 Tab 键顺序。
4. 建立信号－槽之间的连接。

5. 实现对话框中的自定义槽。

要启动 Qt 设计师，在 Windows 下，可单击“开始”菜单中的 Qt by Trolltech v4.x.y→Designer；在 UNIX 下，在命令行中输入”designer”；在 Mac OS X Finder 中，直接双击 designer。当 Qt 设计师开始运行后，它会弹出一个多种模板的列表。单击 Widget 模板，然后再单击 Create。（”Dialog with Buttons Bottom” 模板看起来可能更具诱惑力，但是对于这个例子来说，为了能够看到是如何完成 OK 和 Cancel 按钮的，所以需要采用手工方式来创建它们。）现在应该会看到一个名为”Untitled” 的窗口。

默认情况下，Qt 设计师的用户界面由多个顶级窗口构成。如果你更喜欢像图 2.5 所示的那种多文档 (MDI) 界面风格，即只有一个顶级窗口和多个子窗口构成的界面，则可以单击 Edit→Preferences，然后将用户界面模式设置为 Docked Window（停靠窗口）即可。

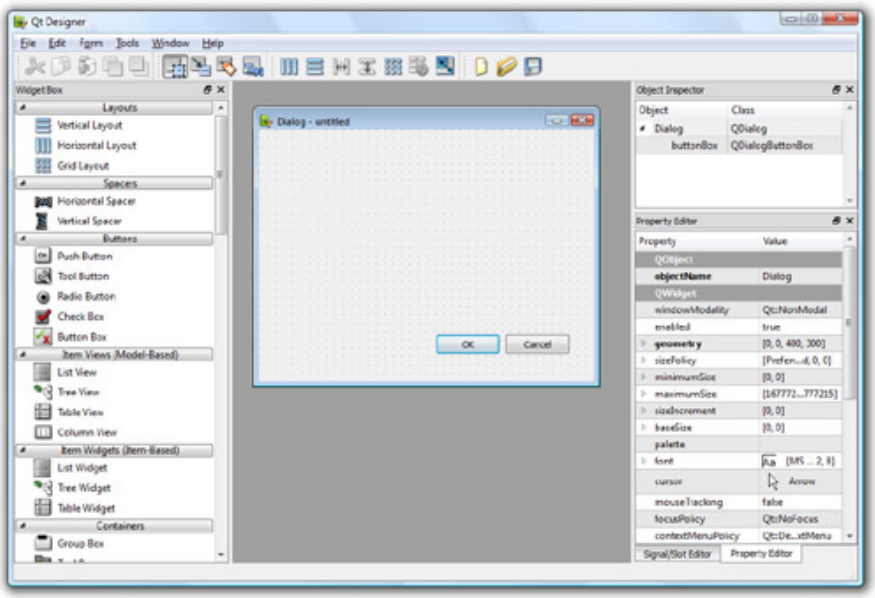


图 2.5: Windows Vista 中显示为停靠窗口模式的 Qt 设计师

第一步是创建子窗口部件并且把它们放置到窗体中。创建一个标签、一个行编辑器、一个水平分隔符和两个按钮。对于这里的每一项，可先从 Qt 设计师的窗口部件工具箱中拖拽其名字或者图标并将其放到窗体中的大概位罩。在 Qt 设计师中，分隔符会显示为一个蓝色的弹簧，但在最终结果的窗体中它是不可见的。

现在，向上拖动窗体的底部使它变短一些，这样将会产生一个类似于图 2.6 的窗体。不要在窗体上为确定这些项的位置而花费太多的时间，会在稍后使用 Qt 的布局管

理器，它可以把这些项摆放得恰到好处。

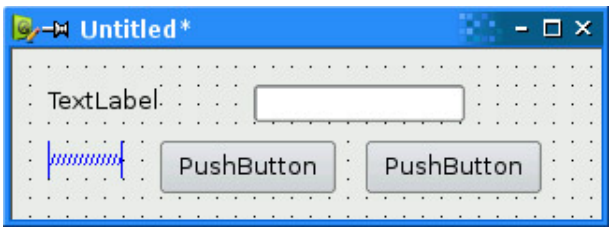


图 2.6: 带一些窗口部件的窗体

使用 Qt 设计师的属性编辑器可以设置每一个窗口部件中的属性：

1. 单击文本标签。确保此时 objectName 的属性是”label”，那么就可以将它的 text 属性设置成”&Cell Location:”。
2. 单击行编辑器。确保 objectName 属性是”lineEdit”。
3. 单击第一个按钮。将它的 objectName 属性设置成”okButton”，将它的 enabled 属性设置成”false”，将它的 text 属性设置成”OK”，并且把它的 default 属性设置成”true”。
4. 单击第二个按钮。将它的 objectName 属性设置成”cancelButton”，并且将它的 text 属性设置成”Cancel”。
5. 单击这个窗体中空白的地方，选中窗体本身。将 objectName 属性设置成”GoToCellDialog”，并且将它的 windowTitle 属性设置成”Go to Cell”。

现在，除了文本标签，所有的窗口部件看起来都很不错，文本标签仍显示为”&Cell Location:”。单击 Edit→Edit Buddies 进入一种允许设置窗口部件伙伴 (buddy) 的特殊模式。然后，单击这个标签并把红色箭头拖到行编辑器上，释放鼠标按键。现在标签看起来应该显示为”Cell Location:”，如图 2.7 所示，同时，它还会把行编辑器看成是自己的伙伴。单击 Edit→Edit Widgets 离开伙伴设置模式。

图 2.7: 带属性设置的窗体

下一步是在窗体中摆放这些窗口部件，步骤如下：

1. 单击”Cell Location:” 标签并且当单击与之相邻的行编辑器时按下 Shift 键，这样就可以同时选择它们。单击 Form→Lay Out Horizontally。
2. 单击分隔符，然后在单击窗体的 OK 按钮和 Cancel 按钮时一直按下 Shift 键。单击 Form→Lay Out Horizontally。
3. 单击窗体中的空白，取消对所有已选中项的选择，然后单击 Form→Lay Out Vertically。
4. 单击 Form→Adjust Size，重新把窗体的大小定义为最佳形式。

在窗体上出现的红线就是已经创建的布局，如图 2.8 所示。但是在窗体运行的时候，它们是绝不会出现的。

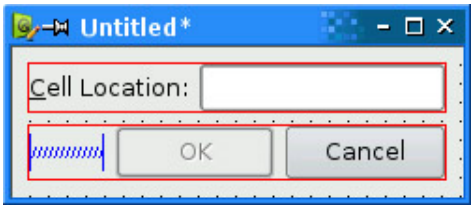


图 2.8: 带布局的窗体

现在，单击 Edit→Edit Tab Order。在每一个可以接受焦点的窗口部件上，都会出现一个带蓝色矩形的数字，如图 2.9 所示。按照你所希望的接受焦点的顺序单击每一个窗口部件，然后单击 Edit→Edit Widget，离开 Tab 键顺序设置模式。

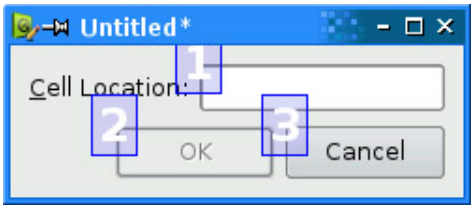


图 2.9: 设置窗体的 Tab 键顺序

要预览这个对话框，可单击 Form→Preview 菜单选项。通过重复按下 Tab 键来检查对话框 Tab 键的顺序。使用窗体标题栏上的 Close 按钮，可以关闭对话框。

把对话框保存到 gotocell 目录下，另存为 gotocelldialog.ui，然后使用一个纯文本编辑器在同一目录下创建一个 main.cpp 文件，内容如下：

```
#include <QApplication>
#include <QDialog>

#include "ui_gotocelldialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Ui::GoToCellDialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();

    return app.exec();
}
```

现在运行 qmake，生成一个.pro 文件和一个 makefile 文件（命令分别是: qmake-project; qmake gotocell.pro）。qmake 工具非常智能，它可以自动检测到用户界面文件 gotocelldialog.ui 并且可以生成适当的 makefile 规则来调用 Qt 的用户界面编译器 (user interface compiler , uic)。uic 工具会将 gotocelldialog.ui 文件转换成 C++ 并且将转换结果存储在 ui_gotocelldialog.h 文件中。

所生成的 ui_tocelldialog.h 文件中包含了类 Ui::GoToCellDialog 的定义，该类是一个与 gotocelldialog.ui 文件等价的 C++ 文件。这个类声明了一些成员变量，它们存储着窗体中的子窗口部件和子布局，以及用于初始化窗体的 setupUi() 函数。生成的类看起来如下所示:

```
class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
```

```
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...
    void setupUi(QWidget *widget) {
    ...
    }
};
```

生成的类没有任何基类。当在 main.cpp 文件中使用该窗体时，可以创建一个 QDialog 对象，然后把它传递给 setupUi() 函数。

如果现在运行该程序，对话框也可以工作，但它并没有正确地实现所想要的那些功能：

- OK 按钮总是失效的。
- Cancel 按钮什么也做不了。
- 行编辑器可以接受任何文本，而不是只能接受有效的单元格位置坐标。

通过写一些代码，就可以让对话框具有适当的功能。最为简捷的做法是创建一个新类，让该类同时从 QDialog 和 Ui::GoToCellDialog 中继承出来，并且由它来实现那些缺失的功能（从而也证明了这句话：通过简单地增加另外一个间接层就可以解决软件的任何问题）。命名惯例是：将该类与 uic 所生成的类具有相同的名字，只是没有 Ui:: 前缀而已。

使用文本编辑器，创建一个名为 gotocelldialog.h 的文件，其中所包含的代码如下所示：

```
1 #ifndef GOTOCELLDIALOG_H
2 #define GOTOCELLDIALOG_H
3
4 #include <QDialog>
```

```
5
6  #include "ui_gotocelldialog.h"
7
8  class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
9  {
10     Q_OBJECT
11
12 public:
13     GoToCellDialog(QWidget *parent = 0);
14
15 private slots:
16     void on_lineEdit_textChanged();
17 };
18
19 #endif
```

在这里，使用了 public 继承，这是因为我们想在该对话框的外面访问该对话框的窗口部件。包含在 gotocelldialog.cpp 文件中的实现代码如下所示：

```
1  #include <QtGui>
2
3  #include "gotocelldialog.h"
4
5  GoToCellDialog::GoToCellDialog(QWidget *parent)
6      : QDialog(parent)
7  {
8      setupUi(this);
9
10     QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
11     lineEdit->setValidator(new QRegExpValidator(regExp, this));
12
13     connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
```

```
14      connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
15  }
16
17  void GoToCellDialog::on_lineEdit_textChanged()
18  {
19      okButton->setEnabled(lineEdit->hasAcceptableInput());
20  }
```

在构造函数中，调用 setupUi() 函数来初始化窗体。正是由于使用了多重继承关系，可以直接访问 Ui::GoToCellDialog 中的成员。创建了用户接口后，setupUi() 函数还会自动将那些符合 on_objectName_signalName() 命名惯例的任意槽与相应的 objectName 的 signalName() 信号连接到一起。在这个例子中，这就意味着 setupUi 函数将建立如下所示的信号－槽连接关系：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));
```

同样还是在构造函数中，设置了一个检验器来限制输入的范围。Qt 提供了三个内置检验器类：QIntValidator、QDoubleValidator 和 QRegExpValidator。在这里使用检验器类 QRegExpValidator，让它带一个正则表达式 "[A-Za-z][1-9][0-9]{0,2}"，它的意思是：允许一个大写或者小写的字母，后面跟着一个范围为 1 ～ 9 的数字，后面再跟 0 个、1 个或 2 个 0 ～ 9 的数字。（对于正则表达式的介绍，请查看参考文档中的 QRegExp 类。）

通过把 this 传递给 QRegExpValidator 的构造函数，使它成为 GoToCellDialog 对象的一个子对象。这样，以后就不用担心有关删除 QRegExpValidator 的事情了：当删除它的父对象时，它也会被自动删除。

Qt 的父－子对象机制是在 QOject 中实现的。当利用一个父对象创建一个子对象（一个窗口部件，一个检验器，或是任意的其他类型）时，父对象会把这个子对象添加到自己的子对象列表中。当删除这个父对象时，它会遍历子对象列表并且删除每一个子对象。然后，这些子对象再去删除它们自己所包含的每个子对象。如此反复递归调用，直至清空所有子对象为止。这种父－子对象机制可在很大程度上简化内存管理工作，降低内存泄漏的风险。需要明确删除的对象是那些使用 new 创建的并且没有父

对象的对象。并且，如果在删除一个父对象之前先删除了它的子对象，Qt 会自动地从它的父对象的子对象列表中将其移除。

对于窗口部件，父对象还有另外一层含义：子窗口部件会显示在它的父对象所在的区域中。当删除这个父窗口部件时，不仅子对象会从内存中消失，而且它也会在屏幕上消失。

在构造函数的最后部分，我们将 OK 按钮连接到 QDialog 的 accept() 槽，将 Cancel 按钮连接到 reject() 槽。这两个槽都可以关闭对话框，但 accept() 槽可以将对话框返回的结果变量设置为 QDialog::Accepted（其值等于 1），而 reject() 槽会把对话框的值设置为 QDialog::Rejected（其值等于 0）。当使用这个对话框的时候，可以利用这个结果变量判断用户是否单击了 OK 按钮，从而执行相应的动作。

根据行编辑器中是否包含了有效的单元格位置坐标，on_lineEdit_textChanged() 槽可以启用或者禁用 OK 按钮。QLineEdit::hasAcceptableInput() 会使用在构造函数中设置的检验器来判断行编辑器中内容的有效性。

这样，就完成了这个对话框。现在可以通过重写 main.cpp 文件来使用这个对话框：

```
#include <QApplication>

#include "gotocelldialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
    return app.exec();
}
```

使用 qmake -project 命令重新生成 gotocell.pro 文件（因为已经在工程中添加了源文件），使用 qmake gotocell.pro 命令更新 makefile 文件，然后再次构建并运行应用程序。在行编辑器中输入"A12"，这时可以注意到 OK 按钮已经变为启用了。尝试输入一些随机文本来看看检验器是如何完成它的工作的。单击 Cancel 按钮

关闭对话框。

这个对话框工作得很好，但对于 Mac OS X 用户，这些按钮却显得不够圆润。在前面采用了单独添加每个按钮的方法，这是为了可以让我们看出是如何完成这些步骤的，但我们本来的确是应当使用 QDialogButtonBox 的，它是一个可以容纳给定的按钮的窗口部件，可以让那些按钮以正确方式呈现在应用程序所运行的平台上，如图 2.10 所示。

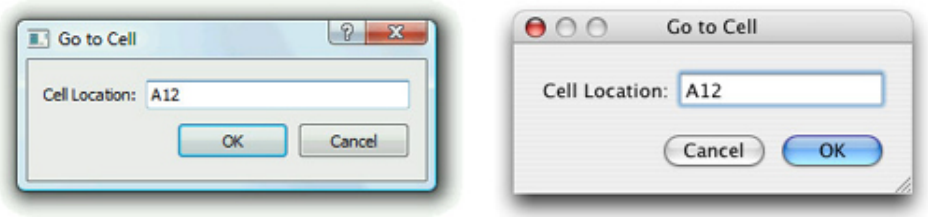


图 2.10: Windows Vista 和 Mac OS X 上的 Go to Cell 对话框

要使用 QDialogButtonBox 来制作这个对话框，必须同时修改设计过程和上述代码。在 Qt 设计师中，一共需要 4 步：

1. 单击窗体（不是任何窗口部件或者布局），然后单击 Form→Break Layout。
2. 单击并删除 OK 按钮、Cancel 按钮、水平分隔符以及（现在为空的）水平布局。
3. 在窗体上拖放一个“按钮盒”(Button Box)，放在标签和行编辑器单元的下方。
4. 单击窗体，然后单击 Form →Lay Out Vertically。

如果只打算修改设计，比如修改对话框的布局和窗口部件的属性等，那么只要重新构建应用程序即可。但在这里是移除了一些窗口部件并且添加了一个新的窗口部件，所以在这种情况下，通常还必须对代码进行修改。

我们所必须做的修改都在 gotocelldialog.cpp 文件中。这里给出的是其构造函数的新版本：

```
5  GoToCellDialog::GoToCellDialog(QWidget *parent)
6      : QDialog(parent)
7  {
8      setupUi(this);
9      buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);
10
11     QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
12     lineEdit->setValidator(new QRegExpValidator(regExp, this));
13
14     connect(buttonBox, SIGNAL(accepted()), this, SLOT(accept()));
15     connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
16 }
```

在前一版本中，一开始在 Qt 设计师中禁用了 OK 按钮。但是在使用 QDialogButtonBox 之后就不能那样做了，因而可以在代码中调用 setupUi() 之后，再立即禁用 OK 按钮，这样也可以达到同样的效果。类 QDialogButtonBox 有一组标准按钮的枚举值，并且可以利用这一点来访问这些特殊的按钮，本例就是访问 OK 按钮。

非常方便的是 Qt 设计师对于 QDialogButtonBox 的默认名称就是 buttonBox。双方的连接会从按钮盒而不是从按钮自己创建出来。在单击一个带 AcceptRole 的按钮时，就会发射 accepted() 信号，这一点与单击一个带 RejectRole 的按钮会发射 rejected() 信号的情况相似。默认情况下，标准的 QDialogButtonBox::Ok 按钮具有 AcceptRole 属性，而标准的 QDialogButtonBox::Cancel 按钮具有 RejectRole 属性。

还需要在`on_lineEdit_textChanged` 槽中做一处修改：

```
18 void GoToCellDialog::on_lineEdit_textChanged()
19 {
20     buttonBox->button(QDialogButtonBox::Ok)->setEnabled(
21             lineEdit->hasAcceptableInput());
22 }
```

与之前的唯一不同之处在于不是对存储为成员变量的特殊按钮进行引用，而是直

接去访问按钮盒中的 OK 按钮。

使用 Qt 设计师的一个好处就在于它为程序员在修改自己设计的窗体时提供了很大的自由，并且不必再强迫自己去修改源代码。当完全通过手写 C++ 代码开发窗体时，对窗体设计的修改将会相当耗时。利用 Qt 设计师，由于 uic 会自动为那些发生了改变的窗体重新生成源代码，所以就不会再浪费时间了。对话框的用户交互界面会被保存为.ui 文件（一种基于 XML 的文件格式），而通过对 uic 所生成的类进行子类化，就可以实现自定义的函数功能。

2.4 改变形状的对话框

我们已经看到了如何创建对话框，无论何时使用它们，这些对话框永远只会显示出一些相同的窗口部件。在某些情况下，人们非常希望能够提供一些可以改变形状的对话框。最常见的可改变形状的对话框有两种：扩展对话框 (extension dialog) 和多页对话框 (multi-page dialog)。在 Qt 中，不论是纯粹使用代码还是使用 Qt 设计师，都可以实现这两种对话框。

扩展对话框通常只显示简单的外观，但是它还有一个切换按钮 (toggle button)，可以让用户在对话框的简单外观和扩展外观之间来回切换。扩展对话框通常用于试图同时满足普通用户和高级用户需要的应用程序中，这种应用程序通常会隐藏那些高级选项，除非用户明确要求看到它们。在这一节中，将使用 Qt 设计师来创建如图 2.11 所示的扩展对话框。

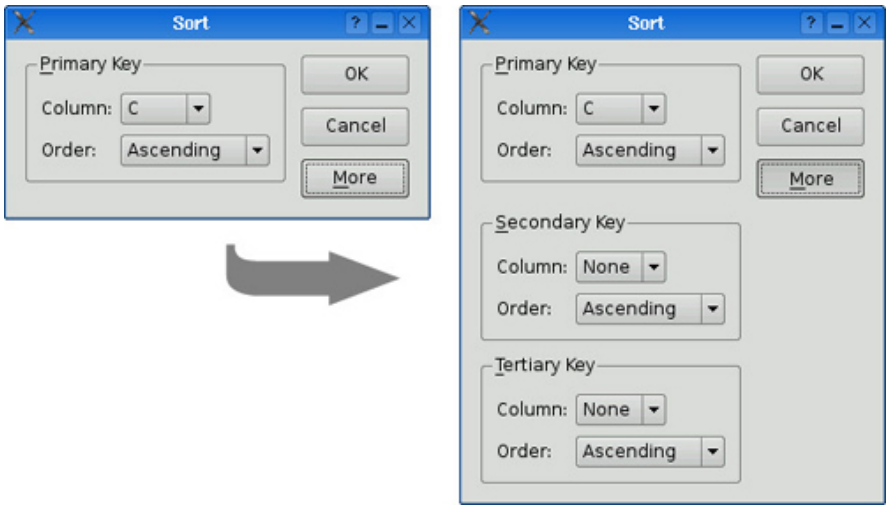


图 2.11: 具有简单外观和扩展外观的 Sort 对话框

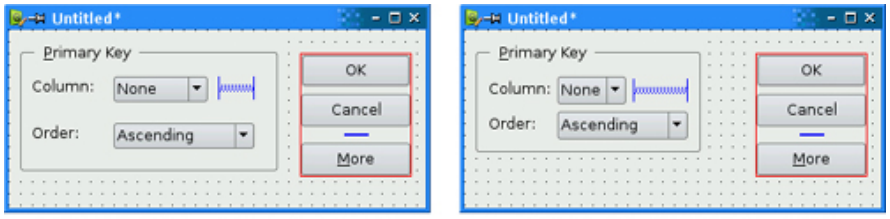
这个对话框是一个用于电子制表软件应用程序的排序对话框（Sort 对话框），在这个对话框中，用户可以选择一列或多列进行排序。在这个简单外观中，允许用户输入一个单一的排序键，而在扩展外观下，还额外提供了两个排序键。More 按钮允许用户在简单外观和扩展外观之间切换。

我们将在 Qt 设计师中创建这个对话框的扩展外观，并且在运行时根据需要隐藏排序的第二键和第三键。这个窗口部件看起来有些复杂，但在 Qt 设计师中可以轻而易举地完成它。简单的诀窍是首先完成主键部分，然后再复制并且粘贴两次就可以获得第二键和第三键所需的内容。

1. 单击 File→New Form，并选择”Dialog without Buttons” 模板。
2. 创建 OK 按钮并把它拖放到窗体的右上角。将它的 objectName 修改为”okButton”，并将它的 default 属性设置为”true”。
3. 创建 Cancel 按钮并把它拖放到 OK 按钮的下方。将它的 objectName 修改为”cancelButton”。
4. 创建一个垂直分隔符并把它拖放到 Cancel 按钮的下方，然后再创建一个 More 按钮。并将它放在垂直分隔符的下方。将 More 按钮的 objectName 修改为”moreButton”，text 属性设置成”&More”，checkable 属性设置为”true”。
5. 单击 OK 按钮，按下 Shift 键后再单击 Cancel 按钮、垂直分隔符和 More 按钮，然后单击 Form→Lay Out Vertically。
6. 创建一个群组框、两个标签、两个组合框以及一个水平分隔符，然后把它们放到窗体上的任意位置。
7. 拖动群组框的右下角使它变大一些。然后，把其他窗口部件移到群组框中，并且按照如图 2.12(a) 所示的那样把它们放置到适当位置。
8. 拖动第二个组合框的右边缘，使它的宽度大约为第一个组合框的两倍。
9. 将群组框的 title 属性设置为”&Primary Key”，第一个标签的 text 属性设置为”Column:”，第二个标签的 text 属性设置为”Order:”。
10. 右键单击第一个组合框。从 Qt 设计师弹出的上下文菜单的组合框编辑器中选择 EditItems。用文本”None” 创建一个项。

11. 右键单击第二个组合框并且同样选择 Edit Items。创建一个”Ascending” 项和一个”Descending” 项。

12. 单击群组框，然后单击 Form→Lay Out in a Grid。再次单击群组框，并且单击 Form→Adjust Size。此时将会产生一个如图 2.12(b) 所示的布局。



(a) Group box without layout　　(b) Group box with layout

图 2.12: 将群组框的子对象摆放到网格中

如果没能生成你所希望的那种布局效果，或者是不小心做错了，那么总是可以随时先通过单击 Edit→Undo 或 Form→Break Layout，然后再重新放置这些要摆放的窗口部件，最后再试着对它们重新布局，直到满意为止。

现在来添加其他两个群组框：Secondary Key 和 Tertiary Key：

1. 让对话框窗口足够高，以便可以容纳另外两个部分。

2. 按下 Ctrl 键（在 Mac 中按下 Alt 键），然后单击并拖动 Primary Key 群组框，这样就可以在原群组框（以及它所包含的所有组件）的上方复制出一个新的群组框。仍旧按下 Ctrl 键（或 Alt 键），把复制的这个群组框拖动到原群组框的下方。重复以上步骤，就可以生成第三个群组框，然后把它拖动到第二个群组框的下方。

3. 将它们的 title 属性分别修改为”&Secondary Key” 和”&Tertiary Key”。

4. 创建一个垂直分隔符，并且把它放到 Primary Key 群组框和 Secondary Key 群组框的中间。

5. 把这些窗口部件按图 2.13(a) 所示的那样排列成网格状。

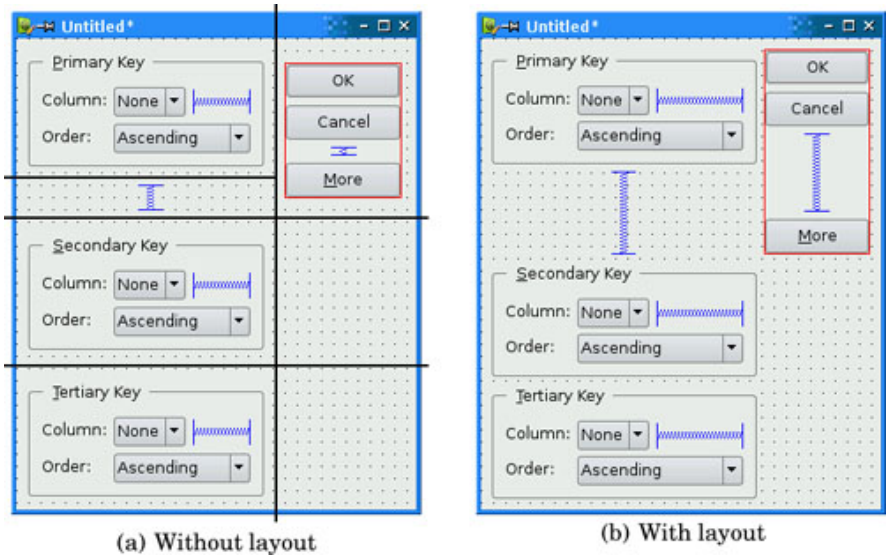


图 2.13: 把窗体的各个子对象摆放到网格中

6. 单击窗体，取消对任意选中窗口部件的选择，然后单击 Form→Lay Out in a Grid。现在，向上和向左拖动窗体的右下角，以便让窗体变得尽可能地小。现在，窗体应该和图 2.13(b) 中显示的一样了。

7. 把两个垂直分隔符的 sizeHint 属性设置为 [20,0]。

最终的网格布局是 4 行 2 列，一共有 8 个单元格。Primary Key 群组框、最左边的垂直分隔符、Secondary Key 群组框和 Tertiary Key 群组框各占一个单独的单元格。包含 OK、Cancel 和 More 按钮的垂直布局占用了两个单元格。最后，会在对话框的右下角剩下两个空白单元格。如果你做出来的对话框不是这样，那么请撤销布局，重新放置窗口部件的位置，然后再重新试试。

把这个窗体重命名为”SortDialog”，并且把它的标题修改为”Sort”。根据图 2.14 修改各个子窗口部件的名称。

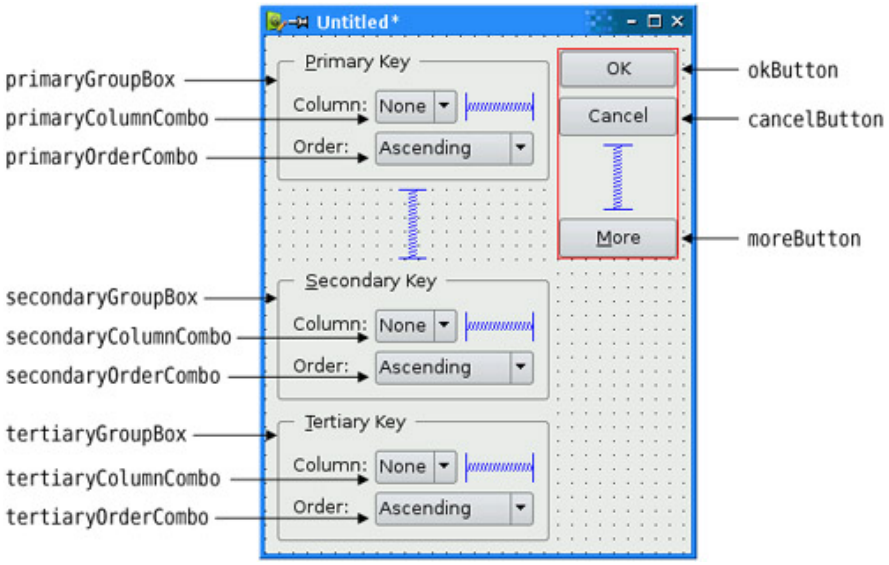


图 2.14: 重新命名窗体中的各个窗口部件

单击 Edit→Edit Tab Order，从窗体的最上面到最下面依次单击每个组合框，然后单击窗体右侧的 OK、Cancel 和 More 按钮。单击 Edit→Edit Widgets 离开 Tab 键顺序设置模式。

现在，窗体已经设计完成，可以开始着手设置一些信号－槽的连接来实现窗体的功能了。Qt 设计师允许我们在构成同一窗体的不同部分内的窗口部件之间建立连接。我们需要建立两个连接。

单击 Edit→Edit Signals/Slots，进入 Qt 设计师的设置连接模式。窗体中各个窗口部件之间的连接用蓝色箭头表示，如图 2.15 所示，并且它们也会同时在 Qt 设计师的 signal/slot 编辑器窗口中显示出来。要在两个窗口部件之间建立连接，可以单击作为发射器的窗口部件并且拖动所产生的红色箭头线到作为接收器的窗口部件上，然后松开鼠标按键。这时会弹出一个对话框，可以从中选择建立连接的信号和槽。

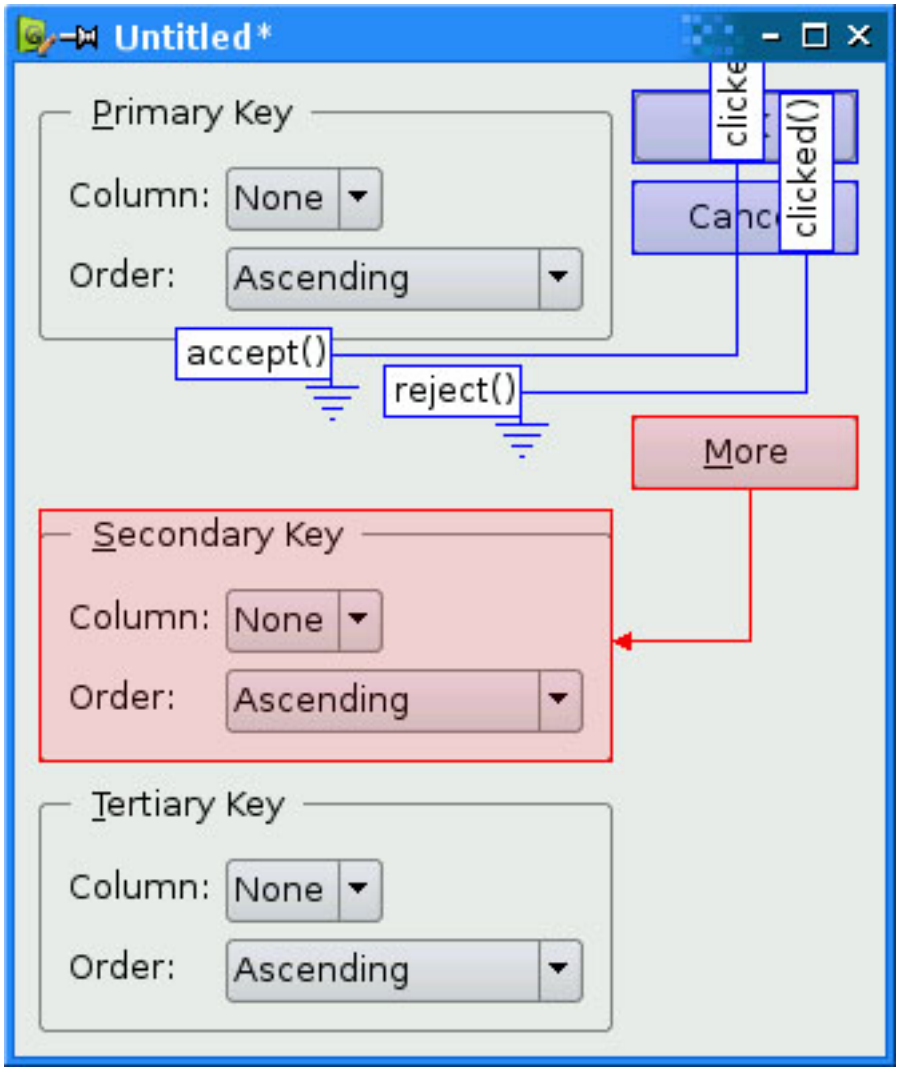


图 2.15: 连接窗体的各个部件

要建立的第一个连接位于 okButton 按钮和窗体的 accept() 槽之间。把从 okButton 按钮开始的红色箭头线拖动到窗体的空白区域，然后松开按键，这样会弹出如图 2.16 所示的设置连接对话框 (Configure Connection dialog)。从该对话框中选择 clicked() 作为信号，选择 accept() 作为槽，然后单击 OK 按钮。

对于第二个连接，把从 cancelButton 按钮开始的红色箭头线拖动到窗体的空白区域，然后在设置连接对话框中连接按钮的 clicked() 信号和窗体的 reject() 槽。

要建立的第三个连接位于 moreButton 按钮和 secondayGroupBox 群组框之间。在这两个窗口部件之间拖动红色箭头线，然后选择 toggled(bool) 作为信号，选择 setVisible(bool) 作为槽。默认情况下，setVisible(bool) 槽不会显示在 Qt 设计师的槽列表中，但如果选中了”Show all signals and slots” 选项，就可以看到这个

槽了。

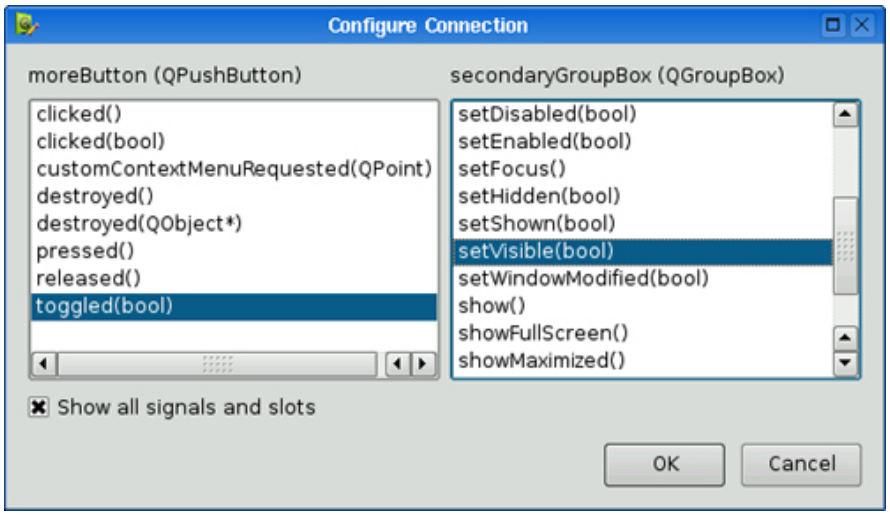


图 2.16: Qt 设计师的连接编辑器

第四个也是最后一个要建立的连接是 moreButton 按钮的 toggled(bool) 信号和 tertiaryGroupBox 群组框的 setVisible(bool) 槽之间的连接。这些连接一旦完成，就可以单击 Edit→Edit Widgets 而离开创建连接模式。

将这个对话框保存在 sort 目录中，文件名为 sortdialog.ui。要给这个窗体添加代码，同样将使用在前一节的 Go to Cell 对话框设计中已经用过的多重继承的方法。

首先，用如下内容创建一个 sortdialog.h 文件：

```
1  #ifndef SORTDIALOG_H
2  #define SORTDIALOG_H
3
4  #include <QDialog>
5
6  #include "ui_sortdialog.h"
7
8  class SortDialog : public QDialog, public Ui::SortDialog
9  {
10     Q_OBJECT
```

```
11
12 public:
13     SortDialog(QWidget *parent = 0);
14
15     void setColumnRange(QChar first, QChar last);
16 };
17
18 #endif
```

然后再创建 sortdialog.cpp 文件:

```
1  #include <QtGui>
2
3  #include "sortdialog.h"
4
5  SortDialog::SortDialog(QWidget *parent)
6      : QDialog(parent)
7  {
8      setupUi(this);
9
10     secondaryGroupBox->hide();
11     tertiaryGroupBox->hide();
12     layout()->setSizeConstraint(QLayout::SetFixedSize);
13
14     setColumnRange('A', 'Z');
15 }
16
17 void SortDialog::setColumnRange(QChar first, QChar last)
18 {
19     primaryColumnCombo->clear();
20     secondaryColumnCombo->clear();
21     tertiaryColumnCombo->clear();
```

```
22
23      secondaryColumnCombo->addItem(tr("None"));
24      tertiaryColumnCombo->addItem(tr("None"));
25
26      primaryColumnCombo->setMinimumSize(
27              secondaryColumnCombo->sizeHint());
28
29      QChar ch = first;
30      while (ch <= last) {
31          primaryColumnCombo->addItem(QString(ch));
32          secondaryColumnCombo->addItem(QString(ch));
33          tertiaryColumnCombo->addItem(QString(ch));
34          ch = ch.unicode() + 1;
35      }
36  }
```

构造函数隐藏了对话框的第二键和第三键这两个部分。它也把有关窗体布局的 sizeConstraint 属性设置为 QLayout::SetFixedSize，这样会使用户不能再重新修改这个对话框窗体的大小。这样一来，布局就会负责对话框重新定义大小的职责，并且也会在显示或者隐藏子窗口部件的时候自动重新定义这个对话框的大小，从而可以确保对话框总是能以最佳的尺寸显示出来。

setColumnRange() 槽根据电子制表软件中选择的列初始化了这些组合框的内容。在（可选的）第二键和第三键的组合框选项中插入了一个"None" 选项。

第 26 行和第 27 行给出了布局中的一个特殊习惯用语。QWidget::sizeHint() 函数可以返回布局系统试图认同的“理想”大小。这也解释了为什么不同的窗口部件或者具有不同内容的类似窗口部件通常会被布局系统分配给不同的尺寸大小。对于这些组合框，这里指的是第二键组合框和第三键组合框，由于它们包含了一个"None" 选项，所以它们要比只包含了一个单字符项目的主键组合框显得宽一些。为了避免这种不一致性，需要把主键组合框的最小大小设置成第二键组合框的理想大小。

这里是一个用于测试效果的 main() 函数，它首先设置了列的范围为从"C" 到"F"，然后再显示这个对话框：

```
#include <QApplication>

#include "sortdialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

这样就完成了这个扩展对话框。就像这个实例所显示的那样，设计一个扩展对话框并不比设计一个简单对话框难：所需要的就是一个切换按钮、一些信号－槽连接以及一个不可以改变尺寸大小的布局。在实际的应用程序中，控制扩展对话框的按钮通常会在只显示了基本对话框时显示为 Advanced»>，而在显示了扩展对话框时才显示为 Advanced«<。这在 Qt 中实现起来非常容易，只需在单击这个按钮时调用 QPushButton 中的 setText() 函数即可完成这一功能。

在 Qt 中，无论是使用手工编码的方式还是使用 Qt 设计师，都可以轻松地创建另一种常用的可以改变形状的对话框：多页对话框。可以通过多种不同的方式创建这种对话框：

- QTabWidget 的用法就像它自己的名字一样。它提供了一个可以控制内置 QStackedWidge 的 Tab 栏。
- QListWidget 和 QStackedWidget 可以一起使用，将 QListWidget::currentRowChanged() 信号与 QStackedWidget::setCurrentIndex() 槽连接，然后再利用 QListWidget 的当前项就可以确定应该显示 QStackedWidget 中的哪一页。
- 与上述 QListWidget 的用法相似，也可以将 QTreeWidget 和 QStackedWidget 一起使用。

第 6 章将讲解 QStackedWidget 类。

2.5 动态对话框

动态对话框 (dynamic dialog) 就是在程序运行时使用的从 Qt 设计师的.ui 文件创建而来的那些对话框。动态对话框不需要通过 uic 把.ui 文件转换成 C++ 代码，相反，它是在程序运行的时候使用 QUiLoader 类载入该文件的，就像下面这种方式：

```
QUiLoader uiLoader;
QFile file(”sortdialog.ui”);
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
    ...
}
```

可以使用 QObject::findChild<T>() 来访问这个窗体中的各个子窗口部件：

```
QComboBox *primaryColumnCombo =
    sortDialog->findChild<QComboBox *>(”primaryColumnCombo”);
if (primaryColumnCombo) {
    ...
}
```

这里的 findChild<T>() 函数是一个模板成员函数，它可以返回与给定的名字和类型相匹配的子对象。由于受编译器的制约，还不能在 MSVC 6 中使用该函数。如果需要使用 MSVC 6 编译器，那么可以通过调用全局函数 qFindChild<T>() 来代替该函数，这个全局函数同样也可以完全相同的方式工作。

QUiLoader 类放在一个独立的库中。为了在 Qt 应用程序中使用 QUiLoader，必须在这个应用程序的.pro 文件中加入这一行内容：

```
CONFIG += uitools
```

动态对话框使不重新编译应用程序而可以改变窗体布局的做法成为可能。动态对话框也同样可用于创建小型终端应用程序，这些程序只有一个内置的前端窗体，并且只是在需要的时候才会去创建所有的其他窗体。

2.6 内置的窗口部件类和对话框类

Qt 提供了一整套内置的窗口部件和常用对话框，这可以满足绝大多数情况。在这一节，几乎给出了它们所有的屏幕截图。会在稍后提供那些少量的特殊窗口部件：第 3 章会讲到用于主窗口的那些窗口部件，如 QMenuBar、QToolBar 和 QStatusBar 等；第 6 章会讲到与布局相关的那些窗口部件，如 QSplitter 和 QScrollArea 等。在本书提供的实例中将会用到绝大多数内置窗口部件和对话框。所有这些窗口部件都会使用 Plastique 风格显示在从图 2.17 到图 2.26 的屏幕截图中。

如图 2.17 所示，Qt 提供了 4 种类型的按钮：QPushButton、QToolButton、QCheckBox 和 QRadioButton。最常使用的就是 QPushButton 和 QToolButton，当单击时，它们就能够发起一个动作，但它们也可以具有像切换按钮（按钮单击一次被按下，再单击一次会还原）一样的行为。复选框 QCheckBox 可用于打开/关闭单独的那些选项，而单选按钮 QRadioButton 通常用于需要互斥条件的地方。

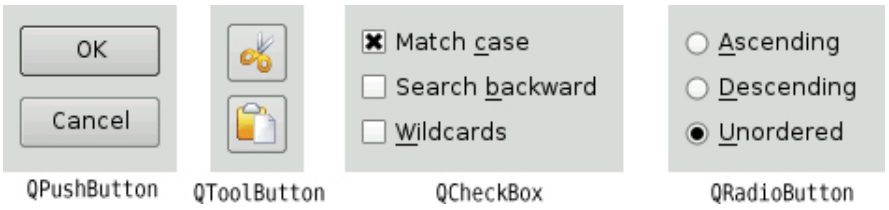


图 2.17: Qt 的按钮窗口部件

Qt 的容器窗口部件是一种可以包含其他窗口部件的窗口部件。图 2.18 和图 2.19 给出了这些容器窗口部件。QFrame 也可用于它自身，这只是为了绘制一些直线，它也可以用作许多其他窗口部件的基类，如 QToolBox 和 QLabel 等。

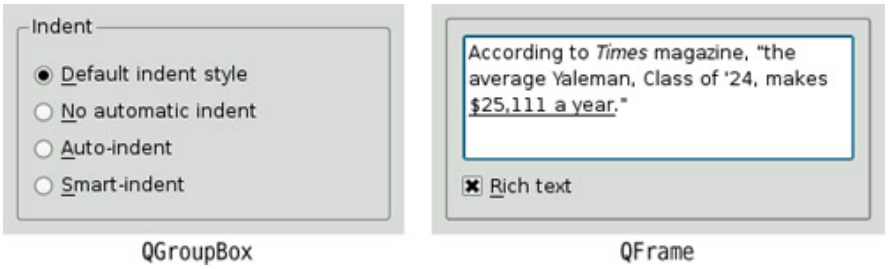


图 2.18: Qt 的单页容器窗口部件

QTabWidget 和 QToolBox 是多页窗口部件。在多页窗口部件中，每一页都是一个子窗口部件，并从 0 开始编号这些页。对于一个 QTabWidget，它的每个 Tab

标签的形状和位置都可以进行设置。如图 2.20 所示，为处理较大的数据量，这些项视图已经进行了优化，并且会经常使用它们的滚动条 (scroll bar)。滚动条机制是在 QAbstractScrollArea 中实现的，它是所有项视图和其他类型的可滚动窗口部件的基类。

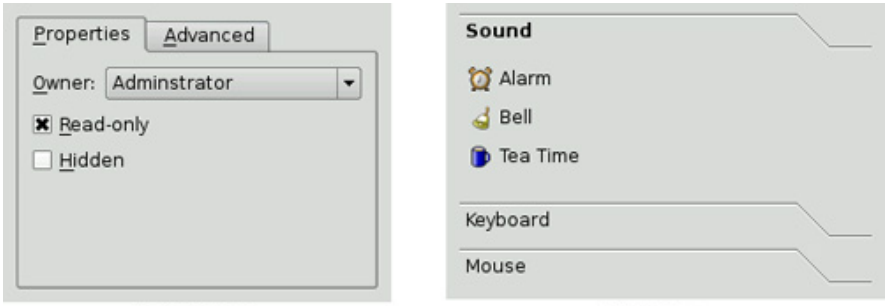


图 2.19: Qt 的多页容器窗口部件

Qt 库含有一个富文本引擎 (rich text engine)，它可用于格式化文本的显示和编辑。该引擎支持字体规范、文本对齐、列表、表格、图片和超文本链接等。可以通过编程的方式一个元素一个元素地生成富文本文档，或者也可以通过所提供的 HTML 格式的文本来生成富文本文档。至于该引擎所支持的 HTML 标记和 CSS 属性的详细说明，请参见文档。

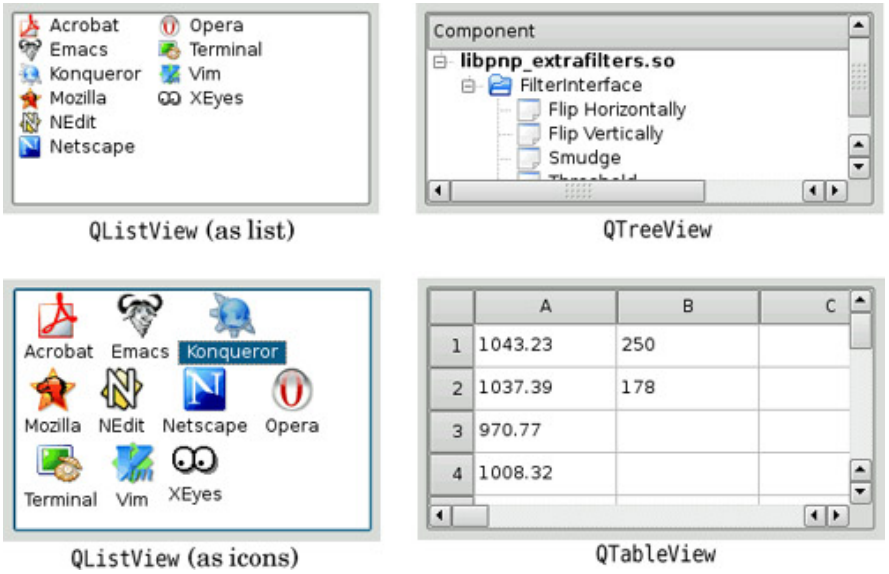


图 2.20: Qt 的项视图窗口部件

如图 2.21 所示，Qt 提供了一些纯粹用于显示信息的窗口部件。QLabel 是这些窗口部件中最重要的一个，并且它也可以用来显示普通文本、HTML 和图片。

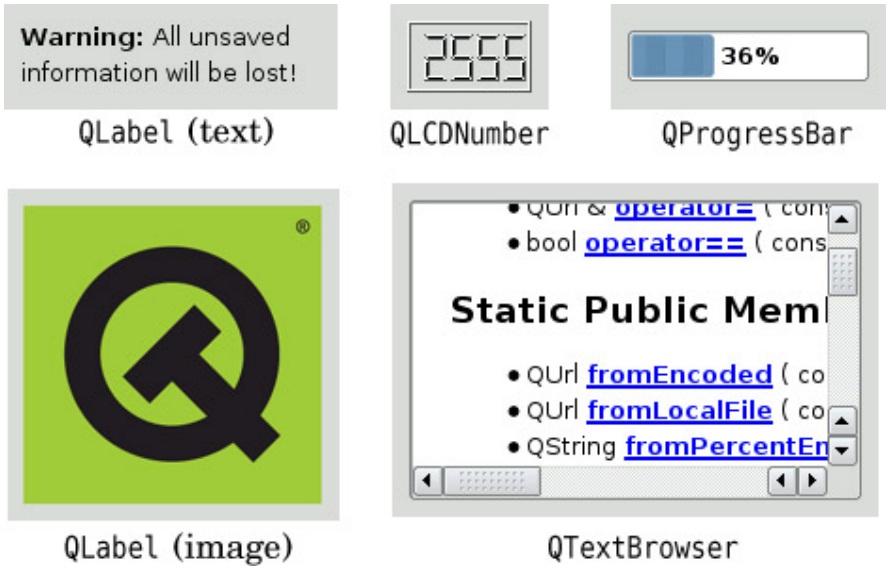


图 2.21: Qt 的显示窗口部件

QTextBrowser 是一个只读型 QTextEdit 子类，它可以显示带格式的文本。对于大型格式化文本文档的处理优先使用这个类而不是 QLabel，因为它与 QLabel 不同，它会在必要时自动提供滚动条，同时还提供了键盘和鼠标导航的广泛支持。Qt 4.3 助手就是使用 QTextBrowser 来为用户呈现文档的。

Qt 提供了数个用于数据输入的窗口部件，如图 2.22 所示。QLineEdit 可以使用一个输入掩码、一个检验器或者同时使用两者对它的输入进行限定。QTextEdit 是 QAbstractScrollArea 的子类，具有处理大量文本的能力。一个 QTextEdit 可设置用于编辑普通文本或者富文本。在编辑富文本的时候，它可以显示 Qt 富文本引擎所支持的所有元素。QLineEdit 和 QTextEdit 两者都对剪贴板提供完美支持。

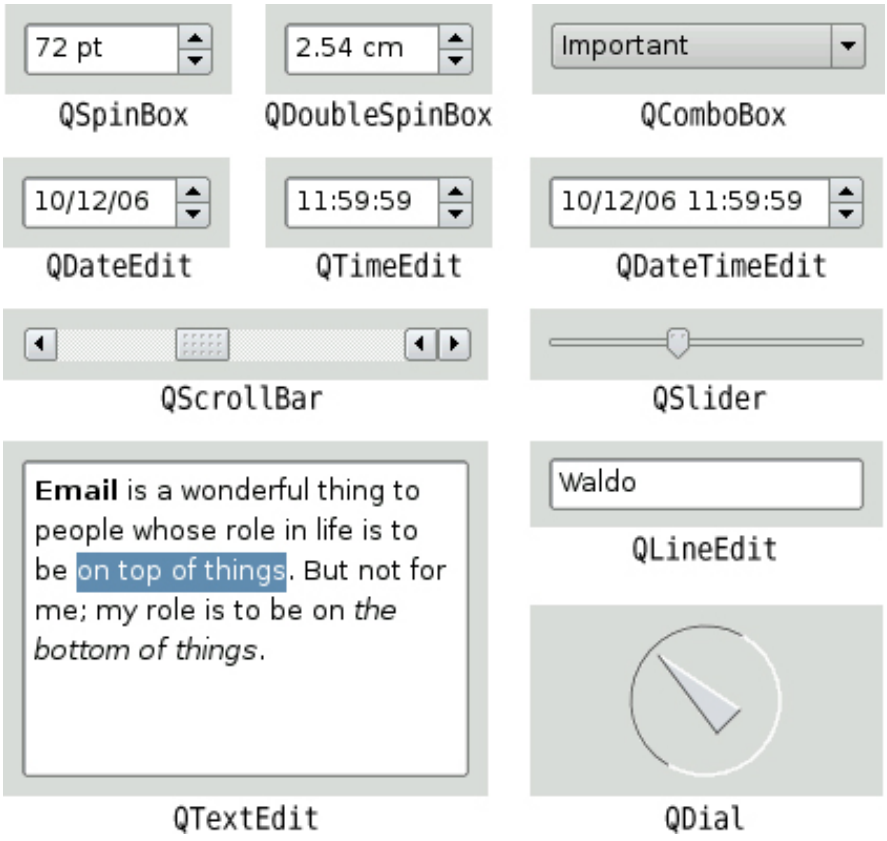


图 2.22: Qt 的输入窗口部件

如图 2.23 所示，Qt 提供了一个通用消息框和一个可以记住它所显示的消息内容的错误对话框。可以使用 QProgressDialog 或者使用图 2.21 中显示的 QProgressBar 来对那些非常耗时的操作进度进行指示。当用户只需要输入一行文本或者一个数字的时候，使用 QInputDialog 会显得非常方便。

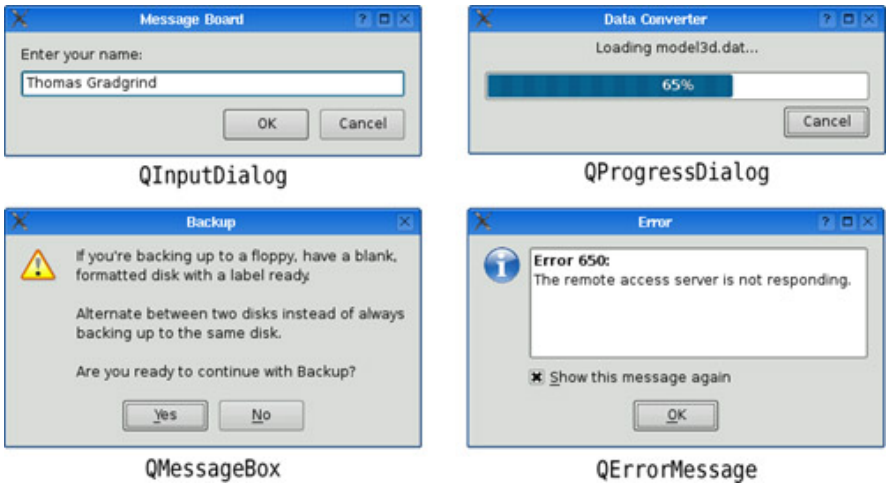
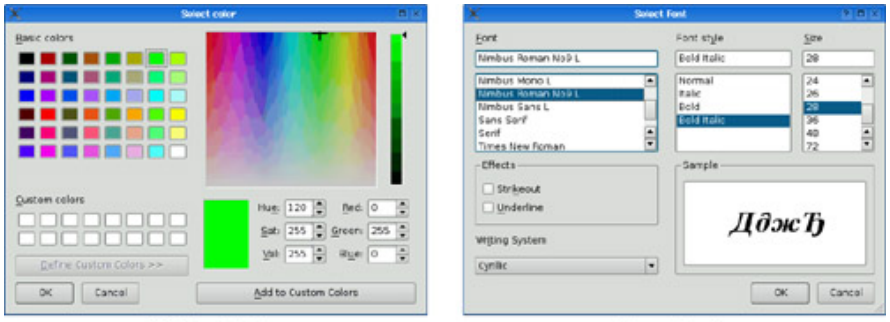


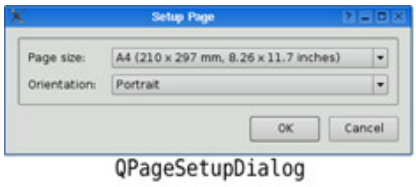
图 2.23: Qt 的反馈对话框

Qt 提供了一套标准的通用对话框，这样可以让用户很容易地选择颜色、字体、文件或者文档打印。图 2.24 和图 2.25 显示了这些对话框。

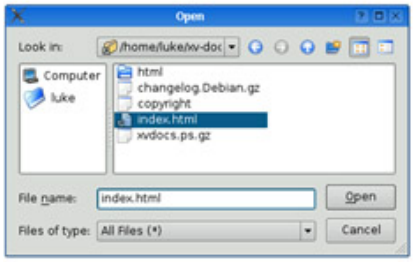


QColorDialog　　QFontDialog

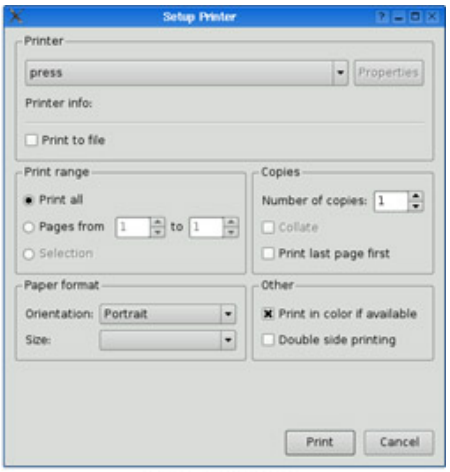
图 2.24: Qt 的颜色对话框和字体对话框



QPageSetupDialog



QFileDialog



QPrintDialog

图 2.25: Qt 的文件对话框和打印对话框

在 Windows 和 Mac OS X 上，Qt 有可能会使用本地系统的对话框，而不是它自己的通用对话框。颜色的选取也可以使用 Qt Solutions 的某个颜色选择窗口部件来完成，而字体也可以使用内置的 QFontComboBox 来选择。

最后，QWizard 为生成向导（wizard，在 Mac OS X 上也称为助手）提供了一个基本框架。对于那些用户可能会难于理解的复杂或者不常见的工作，向导会非常有用。图 2.26 给出了使用向导的一个例子。

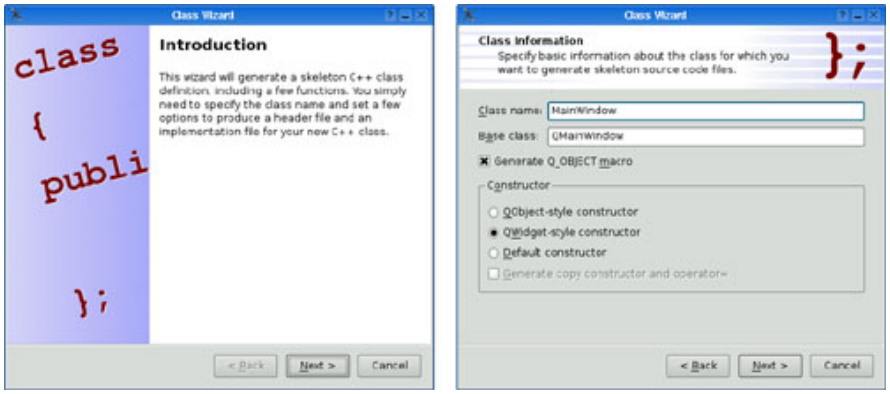


图 2.26: Qt 的 QWizard 对话框

内置窗口部件和常用对话框为用户提供了很多可以直接使用的功能。通过设置窗口部件的属性，或者是通过把信号和槽连接起来并在槽里实现自定义的行为，通常就可以满足许多更为复杂的需求。

如果 Qt 所提供的窗口部件或者常用对话框没有一个合适，那么可以从 Qt Solutions，或者从商业或非商业的第三方软件中找到一个可用的。Qt Solutions 提供了许多额外的窗口部件，包括各种颜色选择器、一个手轮控制器、许多饼状图菜单以及属性浏览器等，还有一个复制对话框。

在某些情况下，你可能希望手动创建一个自定义窗口部件。Qt 使这种工作变得很简单，并且自定义窗口部件也可以像 Qt 的内置窗口部件一样获得与平台无关的所有相同绘制功能。自定义窗口部件甚至可以集成到 Qt 设计师中，这样就可以像使用 Qt 的内置窗口部件一样来使用它们。第 5 章将讲述如何创建自定义窗口部件。

3 创建主窗口

这一章讲解如何使用 Qt 创建主窗口。在本章的最后部分，你将能够创建一个应用程序的完整用户界面，包括菜单、工具栏、状态栏以及应用程序所需的足够多的对话框。

应用程序的主窗口提供了用于构建应用程序用户界面的框架。如图 3.1 所示的 Spreadsheet（电子制表）应用程序的主窗口将构成本章的基础。这个 Spreadsheet 应用程序使用了在第 2 章中创建的三个对话框：Find、Go to Cell 和 Sort。

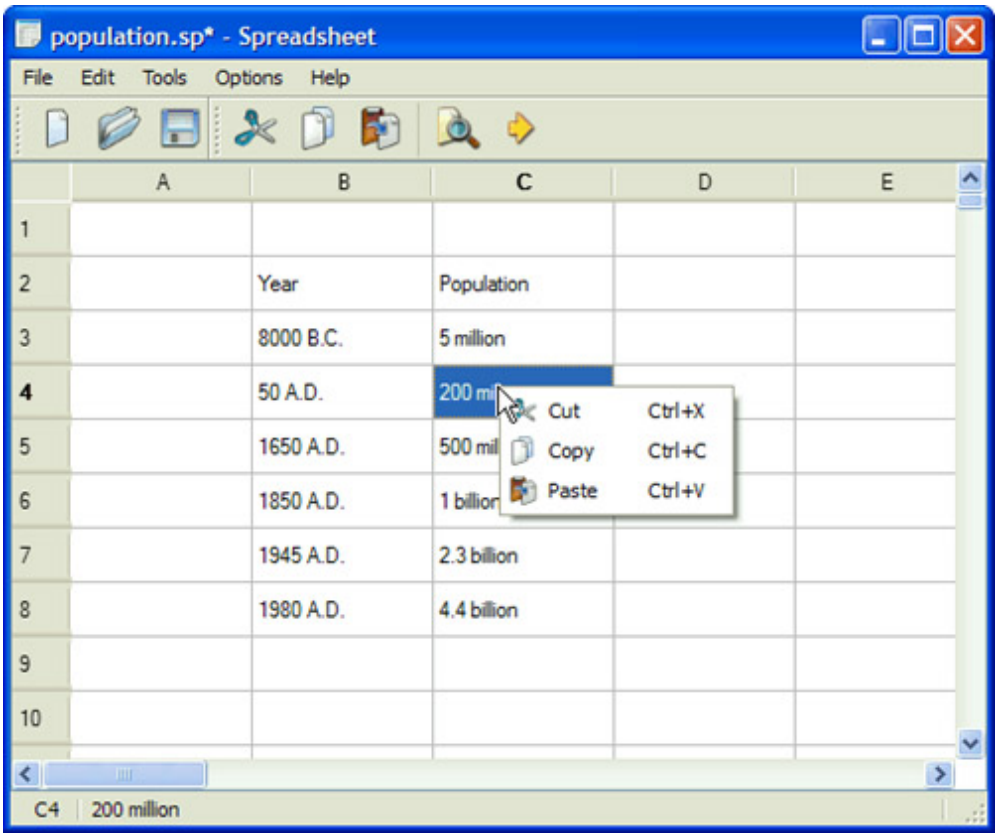


图 3.1: Spreadsheet 应用程序

在绝大多数图形用户界面应用程序的后台，都有一套提供底层功能的代码——例

如，用于读写文件或者用于处理用户界面中的数据的代码。在第 4 章，将会再次把 Spreadsheet 应用程序当作实例，看看如何实现这些功能。

3.1 子类化 QMainWindow

通过子类化 QMainWindow，可以创建一个应用程序的主窗口。由于 QDialog 和 QMainWindow 都派生自 QWidget，所以在第 2 章中看到的许多创建对话框的技术，对于创建主窗口也同样适用。

可以使用 Qt 设计师创建应用程序的主窗口，但是在这一章，将使用代码来完成所有的功能，以便可以说明它们是如何完成的。如果你更喜欢可视化的方式，可以参考 Qt 设计师在线手册中的”Creating a Main Window Application” 一章。

Spreadsheet 应用程序主窗口的源代码分别放在 mainwindow.h 和 mainwindow.cpp 中。先从头文件开始分析：

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

```

```
18 protected:
19     void closeEvent(QCloseEvent *event);
```

我们将 MainWindow 类定义为 QMainWindow 类的子类。由于类 MainWindow 提供了自己的信号和槽所以它包含了 Q_OBJECT 宏。

closeEvent() 函数是 QWidget 类中的一个虚函数，当用户关闭窗口时，这个函数会被自动调用。类 MainWindow 中重新实现了它，这样就可以向用户询问一个标准问题”Do you want to save your changes?”，并且可以把用户的一些偏好设置保存到磁盘中。

```
21 private slots:
22     void newFile();
23     void open();
24     bool save();
25     bool saveAs();
26     void find();
27     void goToCell();
28     void sort();
29     void about();
```

像 File→New 和 Help→About 这样的菜单项，在 MainWindow 中会被实现为私有槽。除了 save() 槽和 saveAs() 槽返回一个 bool 值以外，绝大多数的槽都把 void 作为它们的返回值。当槽作为一个信号的响应函数而被执行时，就会忽略这个返回值；但是当把槽作为函数来调用时，其返回值对我们的作用就和调用任何一个普通的 C++ 函数时的作用是相同的。

```
30     void openRecentFile();
31     void updateStatusBar();
32     void spreadsheetModified();
33
34 private:
```

```
35      void createActions();
36      void createMenus();
37      void createContextMenu();
38      void createToolBars();
39      void createStatusBar();
40      void readSettings();
41      void writeSettings();
42      bool okToContinue();
43      bool loadFile(const QString &fileName);
44      bool saveFile(const QString &fileName);
45      void setCurrentFile(const QString &fileName);
46      void updateRecentFileActions();
47      QString strippedName(const QString &fullFileName);
```

为了能够对用户界面提供支持，主窗口需要更多的私有槽以及数个私有函数。

```
49      Spreadsheet *spreadsheet;
50      FindDialog *findDialog;
51      QLabel *locationLabel;
52      QLabel *formulaLabel;
53      QStringList recentFiles;
54      QString curFile;
55
56      enum { MaxRecentFiles = 5 };
57      QAction *recentFileActions[MaxRecentFiles];
58      QAction *separatorAction;
59
60      QMenu *fileMenu;
61      QMenu *editMenu;
62      QMenu *selectSubMenu;
63      QMenu *toolsMenu;
```

```
64      QMenu *optionsMenu;
65      QMenu *helpMenu;
66      QToolBar *fileToolBar;
67      QToolBar *editToolBar;
68      QAction *newAction;
69      QAction *openAction;
70      QAction *saveAction;
71      QAction *saveAsAction;
72      QAction *exitAction;
73      QAction *cutAction;
74      QAction *copyAction;
75      QAction *pasteAction;
76      QAction *deleteAction;
77      QAction *selectRowAction;
78      QAction *selectColumnAction;
79      QAction *selectAllAction;
80      QAction *findAction;
81      QAction *goToCellAction;
82      QAction *recalculateAction;
83      QAction *sortAction;
84      QAction *showGridAction;
85      QAction *autoRecalcAction;
86      QAction *aboutAction;
87      QAction *aboutQtAction;
88  };
89
90  #endif
```

除了它自己的私有槽和私有函数以外，MainWindow 类还有很多私有变量。当用到这些私有槽和私有函数时，将再对它们进行解释。

现在来看看实现文件：

```
#include <QtGui>

#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

我们包含了 <QtGui> 头文件，其中包含了在子类中所要用到的所有 Qt 类的定义。我们也包含了一些自定义头文件，特别是来自第 2 章的 finddialog.h、gotocelldialog.h 和 sortdialog.h 三个头文件。

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);

    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();

    readSettings();

    findDialog = 0;

    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

在这个构造函数中，先从创建一个 Spreadsheet 窗口部件并且把它设置为这个主窗口的中央窗口部件开始。中央窗口部件会占用主窗口的中央区域部分（如图 3.2

所示）。Spreadsheet 类是 QTableWidget 类的一个子类，并且也具有一些电子制表软件的功能，如对电子制表软件公式的支持等。将会在第 4 章中实现这一功能。

私有函数 createActions()、createMenus()、createContextMenu()、createToolBars() 和 createStatusBar() 创建主窗口中的其余部分，readSettings() 则读取这个应用程序存储的一些设置。

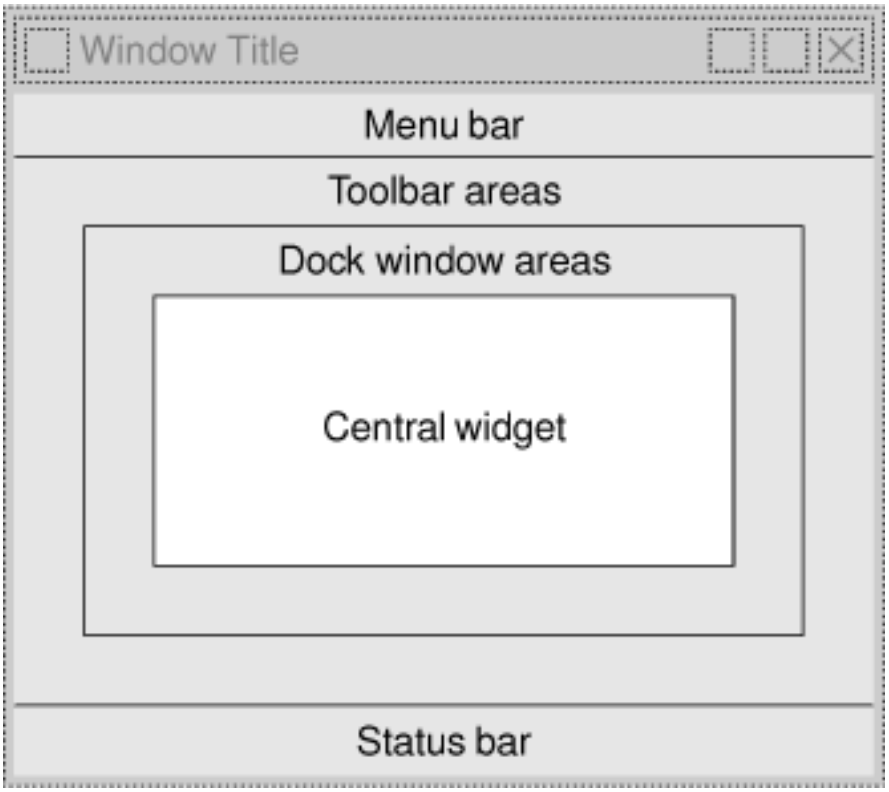


图 3.2: QMainWindow 中的区域分配

我们把 findDialog 指针初始化为空 (null) 指针。在第一次调用 MainWindow::find() 函数时，将会创建该 FindDialog 对象。

在构造函数的最后部分，把窗口的图标设置为 icon.png，它是一个 PNG 格式的文件。Qt 支持很多图像格式，包括 BMP、GIF、JPEG、PNG、PNM、SVG、TIFF、XBM 和 XPM。调用 QWidget::setWindowIcon() 函数可以设置显示在窗口左上角的图标。遗憾的是，还没有一种与平台无关的可在桌面上显示应用程序图标的设置方法。与平台相关的桌面图标设置方法在这个网页中进行了阐述。

图形用户界面 (GUI) 应用程序通常会使用很多图片。为应用程序提供图片的方法有多种，如下是一些最常用的方法：

- 把图片保存到文件中，并且在运行时载入它们。
- 把 XPM 文件包含在源代码中。（这一方法之所以可行，是因为 XPM 文件也是有效的 C++ 文件。）
- 使用 Qt 的资源机制 (resource mechanism)。

这里，使用了 Qt 的资源机制法，因为它比运行时载入文件的方法更方便，并且该方法适用于所支持的任意文件格式。我们将选中的图片存放在源代码树中名为 images 的子目录下。

为了利用 Qt 的资源系统 (resource system)，必须创建一个资源文件 (resource file)，并且在识别该资源文件的.pro 文件中添加一行代码。在这个例子中，已经将资源文件命名为 spreadsheet.qrc，因此只需在.pro 文件中添加如下一行代码：

```
RESOURCES = spreadsheet.qrc
```

资源文件自身使用了一种简单的 XML 文件格式。这里给出的是从已经使用的资源文件中摘录的部分内容：

```
<RCC>
<qresource>
    <file>images/icon.png</file>
    ...
    <file>images/gotocell.png</file>
</qresource>
</RCC>
```

所有资源文件都会被编译到应用程序的可执行文件中，因此并不会弄丢它们。当引用这些资源时，需要使用带路径前缀:/（冒号斜线）的形式，这就是为什么会将图标表示成:/images/icon.png 的形式。资源可以是任意类型的文件（并非只是一些图像），并且可以在 Qt 需要文件名的大多数地方使用它们。第 12 章将对此做进一步说明。

3.2 创建菜单和工具栏

绝大多数现代图形用户界面应用程序都会提供一些菜单、上下文菜单和工具栏。菜单可以让用户浏览应用程序并且可以学会如何处理一些新的事情，上下文菜单和工具栏则提供了对那些经常使用的功能进行快速访问的方法。图 3.3 展示的是 Spreadsheet 应用程序的菜单。

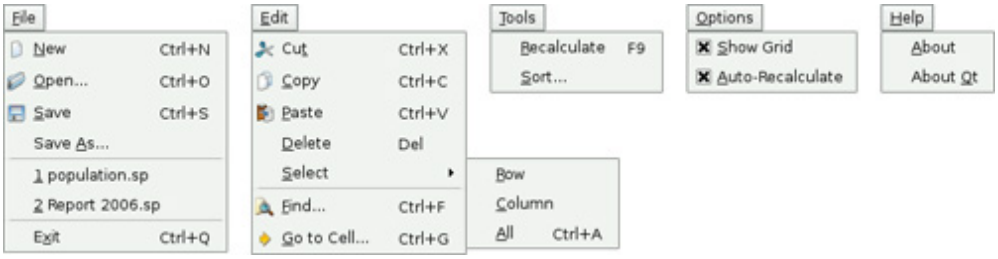


图 3.3: Spreadsheet 应用程序中的菜单

Qt 通过“动作”的概念简化了有关菜单和工具栏的编程。一个动作 (action) 就是一个可以添加到任意数量的菜单和工具栏上的项。在 Qt 中，创建菜单和工具栏包括以下这些步骤：

- 创建并且设置动作。
- 创建菜单并且把动作添加到菜单上。
- 创建工具栏并且把动作添加到工具栏上。

在这个 Spreadsheet 应用程序中，动作是在 createActions() 函数中创建的：

```
163 void MainWindow::createActions()
164 {
165     newAction = new QAction(tr("&New"), this);
166     newAction->setIcon(QIcon(":/images/new.png"));
167     newAction->setShortcut(QKeySequence::New);
168     newAction->setStatusTip(tr("Create a new spreadsheet file"));
```

```
169     connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

动作 New 有一个加速键 (New)、一个父对象（主窗口）、一个图标、一个快捷键和一个状态提示。大多数窗口系统都有用于特定动作的标准化的键盘快捷键。例如，在 Windows、KDE 和 GNOME 中，这个 New 动作就有一个快捷键 Ctrl+N，而在 Mac OS X 中则是 Command+N。通过使用适当的 QKeySequence::StandardKey 枚举值，就可以确保 Qt 能够为应用程序在其运行的平台上提供正确的快捷键。

把这个动作的 triggered() 信号连接到主窗口的私有槽 newFile()——将会在下一节实现它。这个连接可以确保在用户选择 File→New 菜单项、选择工具栏上的 New 按钮或者按下 Ctrl+N 时，都可以调用 newFile() 槽。

由于菜单中的 Open、Save 和 Save As 动作与 New 动作非常相似，所以将会直接跳到 File 菜单中的"recently opened files"（最近打开的文件）的部分。

```
188     for (int i = 0; i < MaxRecentFiles; ++i) {
189         recentFileActions[i] = new QAction(this);
190         recentFileActions[i]->setVisible(false);
191         connect(recentFileActions[i], SIGNAL(triggered()),
192                 this, SLOT(openRecentFile()));
193     }
```

我们为 recentFileActions 数组添加动作。每个动作都是隐式的，并且会被连接到 openRencentFile() 槽。稍后，将会看到如何读这些最新文件中的动作变得可见并且可用。

```
195     exitAction = new QAction(tr("E&xit"), this);
196     exitAction->setShortcut(tr("Ctrl+Q"));
197     exitAction->setStatusTip(tr("Exit the application"));
198     connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

这个 Exit 动作与目前为止所看到的那些动作稍微有些不同。由于没有用于终止应用程序的标准化键序列，所以需要在这里明确指定键序列。另外一个不同之处是我们

连接的是窗口的 close() 槽，而它是由 Qt 提供的。

现在，可以跳到 Select All 动作中：

```
241     selectAllAction = new QAction(tr("&All"), this);
242     selectAllAction->setShortcut(QKeySequence::SelectAll);
243     selectAllAction->setStatusTip(tr("Select all the cells in the "
244                                      "spreadsheet"));
245     connect(selectAllAction, SIGNAL(triggered()),
246             spreadsheet, SLOT(selectAll()));
```

由于槽 selectAll() 是由 QTabelWidget 的父类之一的 QAbstractItemView 提供的，所以就没有必要再去亲自实现它。

现在，不妨进一步跳到 Options 菜单中的 Show Grid 动作中去：

```
273     showGridAction = new QAction(tr("&Show Grid"), this);
274     showGridAction->setCheckable(true);
275     showGridAction->setChecked(spreadsheet->showGrid());
276     showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
277                                     "grid"));
278     connect(showGridAction, SIGNAL(toggled(bool)),
279             spreadsheet, SLOT(setShowGrid(bool)));
```

Show Grid 是一个复选 (checkable) 动作。复选动作在菜单中显示时会带一个复选标记，并且在工具栏中它可以实现成切换 (toggle) 按钮。当启用这个动作时，Spreadsheet 组件就会显示一个网格。我们用 Spreadsheet 组件的默认值来初始化这个动作，这样它们就可以从一开始就同步起来。然后，把 Show Grid 动作的 toggled(bool) 信号和 Spreadsheet 组件的 setShowGrid(bool) 槽连接起来，这个槽继承自 QTableWidget。一旦把这个动作添加到菜单或者工具栏中，用户就可以对网格的显示与否进行切换了。

Show Grid 动作和 Auto-Recalculate 动作是相互独立的两个复选动作。通过 QActionGroup 类的支持，Qt 也可以支持相互排斥的动作。

```
298     aboutQtAction = new QAction(tr("About &Qt"), this);
299     aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
300     connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
301 }
```

对于 About Qt 动作，通过访问 qApp 全局变量，我们可以使用 QApplication 对象的 aboutQt() 槽。这个动作会弹出一个如图 3.4 所示的对话框。

图 3.4: About Qt 对话框

现在已经创建了这些动作，还可以继续构建一个包含这些动作的菜单系统：

```
303 void MainWindow::createMenus()
304 {
305     fileMenu = menuBar()->addMenu(tr("&File"));
306     fileMenu->addAction(newAction);
307     fileMenu->addAction(openAction);
308     fileMenu->addAction(saveAction);
309     fileMenu->addAction(saveAsAction);
310     separatorAction = fileMenu->addSeparator();
311     for (int i = 0; i < MaxRecentFiles; ++i)
312         fileMenu->addAction(recentFileActions[i]);
```

```
313     fileMenu->addSeparator();
314     fileMenu->addAction(exitAction);
```

在 Qt 中，菜单都是 QMenu 的实例。addMenu() 函数可以用给定的文本创建一个 QMenu 窗口部件，并且会把它添加到菜单栏中。QMainWindow::menuBar() 函数返回一个指向 QMenuBar 的指针。菜单栏会在第一次调用 menuBar() 函数的时候就创建出来。

从创建 File 菜单开始，然后再把 New、Open、Save、Save As 动作添加进去。插入一个间隔器 (separator)，可以从视觉上把关系密切的这些项放在一起。使用一个 for 循环从 recentFileActions 数组中添加一些动作（最初是隐藏起来的），然后在最后添加一个 exitAction 动作。

我们已经让一个指针指向了这些间隔器中的某一个。这样就可以允许隐藏（如果没有最近文件的话）或者显示那个间隔器，因为不希望出现在两个间隔器之间什么都没有的情况。

```
316     editMenu = menuBar()->addMenu(tr("&Edit"));
317     editMenu->addAction(cutAction);
318     editMenu->addAction(copyAction);
319     editMenu->addAction(pasteAction);
320     editMenu->addAction(deleteAction);
321
322     selectSubMenu = editMenu->addMenu(tr("&Select"));
323     selectSubMenu->addAction(selectRowAction);
324     selectSubMenu->addAction(selectColumnAction);
325     selectSubMenu->addAction(selectAllAction);
326
327     editMenu->addSeparator();
328     editMenu->addAction(findAction);
329     editMenu->addAction(goToCellAction);
```

现在来创建 Edit 菜单，就像在 File 菜单中所做的那样使用 QMenu::addMenu() 函数添加各个动作，并且在希望出现子菜单的地方使用 QMenu::addMenu() 函数

添加子菜单。一个子菜单与它所属的菜单一样，也是一个 QMenu。

```
331     toolsMenu = menuBar()->addMenu(tr("&Tools"));
332     toolsMenu->addAction(recalculateAction);
333     toolsMenu->addAction(sortAction);
334
335     optionsMenu = menuBar()->addMenu(tr("&Options"));
336     optionsMenu->addAction(showGridAction);
337     optionsMenu->addAction(autoRecalcAction);
338
339     menuBar()->addSeparator();
340
341     helpMenu = menuBar()->addMenu(tr("&Help"));
342     helpMenu->addAction(aboutAction);
343     helpMenu->addAction(aboutQtAction);
344 }
```

通过类似的方式创建 Tools、Options 和 Help 菜单。在 Options 菜单和 Help 菜单之间插入一个间隔器。对于 Motif 和 CDE 风格，这个间隔器会把 Help 菜单放到菜单栏的最右端；对于其他的风格，则将会忽略这个间隔器。图 3.5 是这两种情况的示意图。

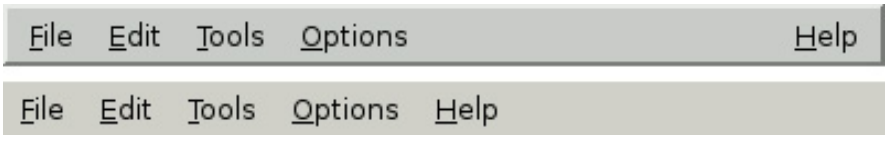


图 3.5: Motif 和 Windows 风格下的菜单栏

```
346 void MainWindow::createContextMenu()
347 {
348     spreadsheet->addAction(cutAction);
349     spreadsheet->addAction(copyAction);
350     spreadsheet->addAction(pasteAction);
```

```
351     spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
352 }
```

任何 Qt 窗口部件都可以有一个与之相关联的 QActions 列表。要为该应用程序提供一个上下文菜单，可以将所需要的动作添加到 Spreadsheet 窗口部件中，并且将那个窗口部件的上下文菜单策略 (context menu policy) 设置为一个显示这些动作的上下文菜单。当用户在一个窗口部件上单击鼠标右键，或者是在键盘上按下一个与平台相关的按键时，就可以激活这些上下文菜单。Spreadsheet 中的上下文菜单如图 3.6 所示。

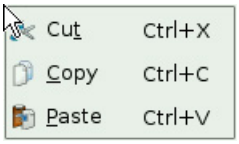


图 3.6: Spreadsheet 应用程序中的上下文菜单

一种更为高级的提供上下文菜单方法是重新实现 QWidget::contextMenuEvent() 函数，创建一个 QMenu 窗口部件，在其中添加所期望的那些动作，并且再对该窗口部件调用 exec() 函数。

```
354 void MainWindow::createToolBars()
355 {
356     fileToolBar = addToolBar(tr("&File"));
357     fileToolBar->addAction(newAction);
358     fileToolBar->addAction(openAction);
359     fileToolBar->addAction(saveAction);
360 
361     editToolBar = addToolBar(tr("&Edit"));
362     editToolBar->addAction(cutAction);
363     editToolBar->addAction(copyAction);
364     editToolBar->addAction(pasteAction);
365     editToolBar->addSeparator();
366     editToolBar->addAction(findAction);
```

```
367     editToolBar->addAction(goToCellAction);
368 }
```

创建工具栏与创建菜单的过程很相似，我们据此创建一个 File 工具栏和一个 Edit 工具栏。就像菜单一样，工具栏也可以有多个间隔器，如图 3.7 所示。

图 3.7: Spreadsheet 应用程序的工具栏

3.3 设置状态栏

随着菜单和工具栏的完成，已经为设置 Spreadsheet 应用程序的状态栏做好了准备。在程序的普通模式下，状态栏包括两个状态指示器：当前单元格的位置和当前单元格中的公式。状态栏也用于显示状态提示和其他一些临时消息。图 3.8 给出了各种情况下的状态栏。

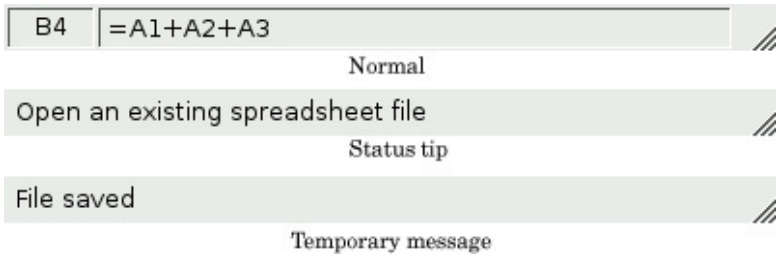


图 3.8: Spreadsheet 应用程序的状态栏

MainWindow 的构造函数会调用 createStatusBar() 来设置状态栏。

```
370 void MainWindow::createStatusBar()
371 {
372     locationLabel = new QLabel(" W999 ");
373     locationLabel->setAlignment(Qt::AlignHCenter);
374     locationLabel->setMinimumSize(locationLabel->sizeHint());
```

```
375
376     formulaLabel = new QLabel;
377     formulaLabel->setIndent(3);
378
379     statusBar()->addWidget(locationLabel);
380     statusBar()->addWidget(formulaLabel, 1);
381
382     connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
383             this, SLOT(updateStatusBar()));
384     connect(spreadsheet, SIGNAL(modified()),
385             this, SLOT(spreadsheetModified()));
386
387     updateStatusBar();
388 }
```

QMainWindow::statusBar() 函数返回一个指向状态栏的指针。[①]状态栏指示器是一些简单的 QLabel，可以在任何需要的时候改变它们的文本。已经在 formulaLabel 中添加了一个缩进格式，以便让那些在它里面显示的文本能够与它的左侧边有一个小的偏移量。当把这些 QLabel 添加到状态栏的时候，它们会自动被重定义父对象，以便让它们成为状态栏的子对象。

图 3.8 所示的两个标签都有不同的空间需求。单元格定位指示器只需要非常小的空间，并且在重新定义窗口大小时，任何多余的空间都会分配给位于右侧的单元格公式指示器。这是通过在公式标签的 QStatusBar::addWidget() 调用中指定一个伸展因子 1 而实现的。位置指示器的默认伸展因子为 0，这也就意味着它不喜欢被伸展。

当 QStatusBar 摆放这些指示器窗口部件时，它会尽量考虑由 QWidget::sizeHint() 提供的每一个窗口部件的理想大小，然后再对那些可伸展的任意窗口部件进行伸展以填满全部可用空间。一个窗口部件的理想大小取决于这个窗口部件的内容以及改变内容时的变化大小。为了避免对定位指示器连续不断地重定义大小，设置它的最小尺寸大小为它所能包含的最大字符数 (”W999”) 和一些空格的总大小。还把它的对齐方式设置为 Qt::AlignHCenter，以便可以在水平方向上居中对齐

① 在第一次调用 statusBar() 函数的时候会创建状态栏。

它的文本。

在函数结尾的附近，把 Spreadsheet 的两个信号和 MainWindow 的两个槽，updateStatusBar() 和 spreadsheetModified()，连接了起来。

```
151 void MainWindow::updateStatusBar()
152 {
153     locationLabel->setText(spreadsheet->currentLocation());
154     formulaLabel->setText(spreadsheet->currentFormula());
155 }
```

updateStatusBar() 槽可以更新单元格定位指示器和单元格公式指示器。只要用户把单元格光标移动到一个新的单元格，这个槽就会得到调用。该槽也可以作为一个普通函数而在 createStatusBar() 的最后用于初始化这些指示器。因为 Spreadsheet 不会在一开始的时候就发射 currentCellChanged() 小心，所以还必须这样做。

```
157 void MainWindow::spreadsheetModified()
158 {
159     setWindowModified(true);
160     updateStatusBar();
161 }
```

spreadsheetModified() 槽把 windowModified 属性设置为 true，用以更新标题栏。这个函数也会更新位置和公式指示器，以便可以让它们反映事件的当前状态。

3.4 实现 File 菜单

在这一节中，将实现那些能够让 File 菜单项正常工作并且能够对最近打开文件进行管理的槽函数和私有函数。

```
38 void MainWindow::newFile()
39 {
40     if (okToContinue()) {
41         spreadsheet->clear();
42         setCurrentFile("");
43     }
44 }
```

当用户点击 File→New 菜单项或者单击工具栏上的 New 按钮时，就会调用 newFile() 槽。如果存在还没有被保存的信息，okToContinue() 私有函数就会弹出如图 3.9 所示的对话框："Do you want to save your changes?"。如果用户选择 Yes 或者 No（保存文档应该选择 Yes），这个函数会返回 true；如果用户选择 Cancel，它就返回 false。Spreadsheet::clear() 函数会清空电子制表软件中的全部单元格和公式。setCurrentFile() 私有函数会更新窗口的标题，以说明正在编辑的是一个没有标题的文档，它还会设置 curFile 私有变量并且更新最近打开文件的列表。

```
416 bool MainWindow::okToContinue()
417 {
418     if (isWindowModified()) {
419         int r = QMessageBox::warning(this, tr("Spreadsheet"),
420                         tr("The document has been modified.\n"
421                            "Do you want to save your changes?"),
422                         QMessageBox::Yes | QMessageBox::No
423                         | QMessageBox::Cancel);
424         if (r == QMessageBox::Yes) {
425             return save();
426         } else if (r == QMessageBox::Cancel) {
427             return false;
428         }
429     }
430     return true;
431 }
```

在 okToContinue() 函数中，会检测 windowModified 属性的状态。如果该属性的值是 true，就显示一个如图 3.9 所示的消息框。这个消息框包含一个 Yes 按钮、一个 No 按钮和一个 Cancel 按钮。

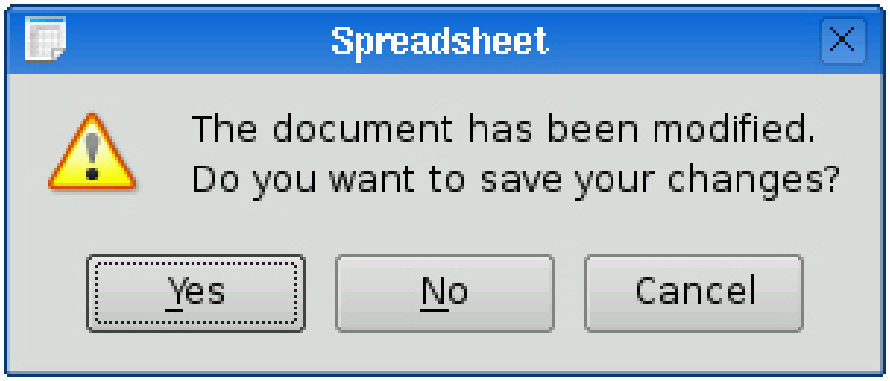


图 3.9: ”Do you want to save your changes?” 消息框

QMessageBox 提供了许多标准按钮，并且会自动尝试着让其中的一个成为默认的确认按钮（在用户按下 Enter 键时会得到激活），一个成为默认的退出按钮（在用户按下 Esc 时会得到激活）。选择一些特殊的按钮作为默认的确认按钮和退出按钮也是有可能的，用户还可以自定义按钮中将要显示的文本内容。

当首次看到 warning() 函数调用时，可能会觉得它有点复杂，但这种常用语法实际上是相当简单的：

```
QMessageBox::warning(parent, title, message, buttons);
```

除了 warning() 之外，QMessageBox 还提供了 information()、question() 和 critical() 函数，它们每一个都有自己特定的图标，这些图标如图 3.10 所示。

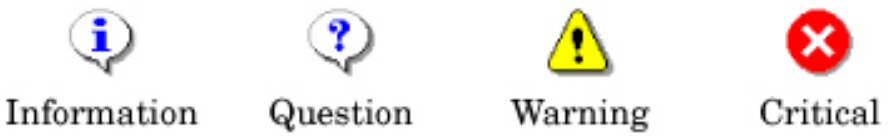


图 3.10: Windows 风格下的消息框图标

```
46 void MainWindow::open()
47 {
48     if (okToContinue()) {
49         QString fileName = QFileDialog::getOpenFileName(this,
50                                     tr("Open Spreadsheet"), ".",
51                                     tr("Spreadsheet files (*.sp)"));
52         if (!fileName.isEmpty())
53             loadFile(fileName);
54     }
55 }
```

open() 槽对 File→Open 做出响应。就像 newFile() 一样，它首先调用 okToContinue() 函数来处理任何没有被保存的变化。然后它使用方便的 QFileDialog::getOpenFileName() 静态函数从用户那里获得一个新的文件名。这个函数会弹出一个文件对话框，让用户选择一个文件，并且返回这个文件名——或者，如果用户单击了 Cancel 按钮，则返回一个空字符串。

传递给 QFileDialog::getOpenFileName() 函数的第一个参数是它的父窗口部件。用于对话框和其他窗口部件的这种父子对象关系意义并不相同。对话框通常都拥有自主权，但是如果它有父对象，那么在默认情况下，它就会居中放到父对象上。一个子对话框也会共用它的父对象的任务栏。

第二个参数是这个对话框应当使用的标题。第三个参数告诉它应当从哪一级目录开始，在这个例子中就是当前目录。

第四个参数指定了文件过滤器。文件过滤器 (filter) 由一个描述文本和一个通配符组成。如果除了要支持 Spreadsheet 本地文件格式以外，还需要支持采用逗号分隔的数据文件和 Lotus 1-2-3 文件，就应当使用如下的文件过滤器：

```
tr("Spreadsheet files (*.sp)\n"
"Comma-separated values files (*.csv)\n"
"Lotus 1-2-3 files (*.wk1 *.wks)")
```

loadFile() 私有函数是在 open() 中得到调用的，它用来载入文件。我们让它成为一个独立的函数，是因为会在载入最近打开的文件中使用同样的功能。

```
433 bool MainWindow::loadFile(const QString &fileName)
434 {
435     if (!spreadsheet->readFile(fileName)) {
436         statusBar()->showMessage(tr("Loading canceled"), 2000);
437         return false;
438     }
439 
440     setCurrentFile(fileName);
441     statusBar()->showMessage(tr("File loaded"), 2000);
442     return true;
443 }
```

我们使用 Spreadsheet::readFile() 函数从磁盘中读取文件。如果载入成功，会调用 setCurrentFile() 函数来更新这个窗口的标题；否则，Spreadsheet::readFile() 将会通过一个消息框把遇到的问题通知给用户。在通常情况下，让底层组件来报告错误消息是一个不错的习惯，这是因为它们可以提供准确的错误细节信息。

在上述两种情况下，都会在状态栏中显示一个消息 2 秒（2000 毫秒），这样可以通知用户应用程序正在做什么。

```
57 bool MainWindow::save()
58 {
59     if (curFile.isEmpty()) {
60         return saveAs();
61     } else {
62         return saveFile(curFile);
63     }
64 }
```

```
445 bool MainWindow::saveFile(const QString &fileName)
446 {
447     if (!spreadsheet->writeFile(fileName)) {
448         statusBar()->showMessage(tr("Saving canceled"), 2000);
449         return false;
450     }
451 
452     setCurrentFile(fileName);
453     statusBar()->showMessage(tr("File saved"), 2000);
454     return true;
455 }
```

save() 槽对 File→Save 做出响应。如果因为这个文件是之前打开的文件或者它是一个已经保存过的文件，这样已经有了一个名字，那么 save() 函数就会用这个名字调用 saveFile() 函数；否则，它只是简单地调用 saveAs() 函数。

```
66 bool MainWindow::saveAs()
67 {
68     QString fileName = QFileDialog::getSaveFileName(this,
69                                 tr("Save Spreadsheet"), ".",
70                                 tr("Spreadsheet files (*.sp)"));
71     if (fileName.isEmpty())
72         return false;
73 
74     return saveFile(fileName);
75 }
```

saveAs() 槽对 File→Save As 做出响应。调用 QFileDialog::getSaveFileName() 函数来从用户那里得到一个文件名。如果用户单击了 Cancel，则返回 false，这将会使这个结果向上传递给它的调用者 [save() 或者 okToContinue()]。

如果给定的文件已经存在，getSaveFileName() 函数将会要求用户确认是否需要覆盖该文件。但通过给 getSaveFileName() 函数传递一个 QFileDia-

log::DontConfirmOverwrite 附加参数，则可以改变这一行为。

```
28 void MainWindow::closeEvent(QCloseEvent *event)
29 {
30     if (okToContinue()) {
31         writeSettings();
32         event->accept();
33     } else {
34         event->ignore();
35     }
36 }
```

当用户单击 File→Exit 或者单击窗口标题栏中的关闭按钮时，将会调用 QWidget::close() 槽。该槽会给窗口部件发射一个“close”事件。通过重新实现 QWidget::closeEvent() 函数，就可以中途截取对这个主窗口的关闭操作，并且可以确定到底是不是真的要关闭这个窗口。

如果存在未保存的更改并且用户选择了 Cancel，就会“忽略”这个关闭事件并且让这个窗口不受该操作的影响。一般情况下，我们会接受这个事件，这会让 Qt 隐藏该窗口。也可以调用私有函数 writeSettings() 来保存这个应用程序的当前设置。

当最后一个窗口关闭后，这个应用程序就结束了。如果需要，通过把 QApplication 的 quitOnLastWindowClosed 属性设置为 false，可以禁用这种行为。在这种情况下，该应用程序将会持续保持运行，直到调用 QApplication::quit() 函数，程序才会结束。

```
457 void MainWindow::setCurrentFile(const QString &fileName)
458 {
459     curFile = fileName;
460     setWindowModified(false);
461 
462     QString shownName = tr("Untitled");
463     if (!curFile.isEmpty()) {
```

```
464         shownName = strippedName(curFile);
465         recentFiles.removeAll(curFile);
466         recentFiles.prepend(curFile);
467         updateRecentFileActions();
468     }
469 
470     setWindowTitle(tr("%1[*] - %2").arg(shownName)
471                                    .arg(tr("Spreadsheet")));
472 }
```

```
497 QString MainWindow::strippedName(const QString &fullFileName)
498 {
499     return QFileInfo(fullFileName).fileName();
500 }
```

在 setCurrentFile() 中，对保存正在编辑的文件名的 curFile 私有变量进行了设置。在把这个文件名显示在标题栏中之前，需要使用 strippedName() 函数移除文件名中的路径字符，这样可以使文件名看起来更友好一些。

每个 QWidget 都有一个 windowModified 属性，如果该窗口的文档存在没有保存的变化，则应当把它设置为 true，否则应当将其设置为 false。在 Mac OS X 下，未保存的文档是通过窗口标题栏上关闭按钮中的一个点来表示的；在其他平台下，则是通过文件名字后跟一个星号来表示的。Qt 会自动处理这一行为，只要始终让 windowModified 属性保持为当前最新状态，并且当需要显示星号的时候，把"[*]"标记放在窗口的标题栏上即可。

传递给 setWindowTitle() 函数的文本是：

```
470     setWindowTitle(tr("%1[*] - %2").arg(shownName)
471                                    .arg(tr("Spreadsheet")));
```

QString::arg() 函数将会使用自己的参数替换最小数字的"%n" 参数，并且会用它的参数返回结果"%n" 字符和最终的结果字符串。在本例中，arg() 被用于两个"%n" 参教中。第一个 arg() 调用会替换参数"%1"，第二个 arg() 调用则会替换

参数”%2”。如果文件名是 budget.sp 并且没有载入翻译文件，那么结果字符串将是”budget.sp[*]-Spreadsheet”。这本应更简单地写作如下代码：

```
setWindowTitle(shownName + tr("[*] - Spreadsheet"));
```

但使用 arg() 函数可以为翻译人员提供更多的灵活性。

如果存在文件名，就需要更新应用程序的最近打开文件列表 recentFiles。可以调用 removeAll() 从列表中移除任何已经出现过的文件名，从而避免该文件名的重复。然后，可以调用 prepend() 把这个文件名作为文件列表的第一项添加进去。在更新了文件列表之后，可以调用私有函数 updateRencentFileActions() 更新 File 菜单中的那些条目。

```
474 void MainWindow::updateRecentFileActions()
475 {
476     QMutableStringListIterator i(recentFiles);
477     while (i.hasNext()) {
478         if (!QFile::exists(i.next()))
479             i.remove();
480     }
481 
482     for (int j = 0; j < MaxRecentFiles; ++j) {
483         if (j < recentFiles.count()) {
484             QString text = tr("&%1 %2")
485                            .arg(j + 1)
486                            .arg(strippedName(recentFiles[j]));
487             recentFileActions[j]->setText(text);
488             recentFileActions[j]->setData(recentFiles[j]);
489             recentFileActions[j]->setVisible(true);
490         } else {
491             recentFileActions[j]->setVisible(false);
492         }
```

```
493     }
494     separatorAction->setVisible(!recentFiles.isEmpty());
495 }
```

使用一个 Java 风格的迭代器，可以移除任何不再存在的文件。一些文件或许已经在前面的会话中使用过，但在此之前还没被删除掉。recentFiles 变量的类型是 QStringList（QString 型列表）。第 11 章会详细说明一些像 QStringList 一样的容器类，其中将会说明它们与 C++ 标准模板库 (Standard Template Library，STL) 之间的关系，也会说明 Qt 的 Java 风格迭代器类的用法。

然后，再遍历一次文件列表，这一次使用数组风格的索引形式。对于每一个文件，创建一个由一个与操作符、一位数字 (j+1)、一个空格和该文件名（不带路径）组成的字符串。我们要为使用这种文本设置相应的动作。例如，如果第一个文件是`C:\My Documents\tab04.sp`，那么第一个动作的文本将会是”&1 tab04.sp”。图 3.11 给出了 recentFileActions 数组和菜单的最终结果之间的对应关系。

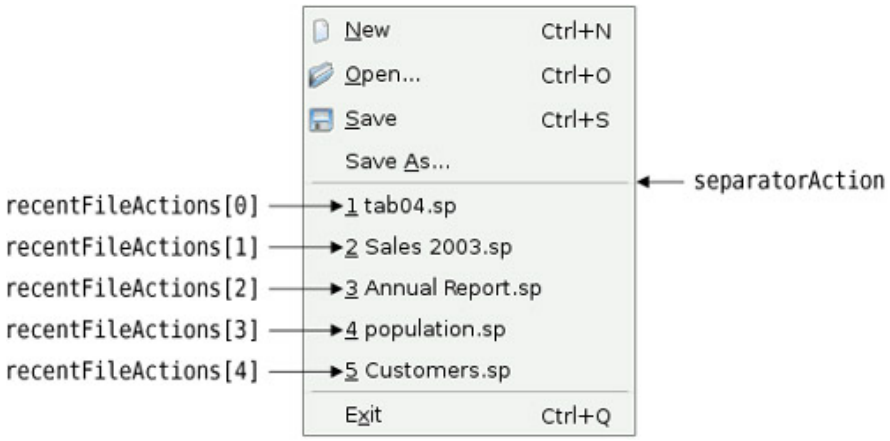


图 3.11: 带最近打开文件列表的 File 菜单

每一个动作都可以带一个与之相关的 QVariant 型 data 项。QVariant 类型可以保存许多 C++ 和 Qt 型变量，第 11 章将说明这一点。这里，将文件的全名保存在动作的 data 项中，以便随后可以方便地找到它。还要将这个动作设置为可见。

如果有比最新文件更多的文件动作，那么只需隐藏那些多余的动作即可。最后，如果至少还存在一个最近打开的文件，那么就应该把间隔器设置为可见。

```
142 void MainWindow::openRecentFile()
143 {
144     if (okToContinue()) {
145         QAction *action = qobject_cast<QAction *>(sender());
146         if (action)
147             loadFile(action->data().toString());
148     }
149 }
```

当用户选择了一个最近打开的文件，就会调用 openRecentFile() 槽。只要有任何未保存的变化，就会调用 okToContinue() 函数，并且假定用户没有取消，还可以使用 QObject::sender() 查出是哪个特有动作调用了这个槽。

qobject_cast<T>() 函数可在 Qt 的 moc（meta-object compiler，元对象编译器）所生成的元信息基础上执行动态类型强制转换 (dynamic cast)。它返回一个指向所需 QObject 子类的指针，或者是在该对象不能被转换成所需的那种类型时返回 0。与标准 C++ 的 dynamic_cast<T>() 不同，Qt 的 qobject_cast<T>() 可正确地跨越动态库边界。在例子中，使用 qobject_cast::<T>() 把一个 QObject 指针转换成 QAction 指针。如果这个转换是成功的（应当是这样的），就可以利用从动作的 data 项中所提取的文件全名来调用 loadFile() 函数。

顺便值得一提的是，由于知道这个发射器是一个 QAction，如果使用 `static_cast<T>()` 或者传统的 C 风格的数据类型强制转换代替原有的数据转换方式，这个程序应当仍然是可以运行的。请参见附录 D 中“类型转换”一节对不同 C++ 数据类型强制转换的概述。

3.5 使用对话框

这一节将说明如何在 Qt 中使用对话框——如何创建、初始化以及运行它们，并且对用户交互中的选择做出响应。本节将会使用在第 2 章中创建的 Find、Go to Cell 和 Sort 对话框，也会创建一个简单的 About 对话框。

我们从如图 3.12 所示的 Find 对话框开始。由于希望用户能够在 Spreadsheet 窗口和 Find 对话框之间进行切换，所以 Find 对话框必须是非模态 (modeless) 的。非模态窗口就是运行在应用程序中对于任何其他窗口都独立的窗口。

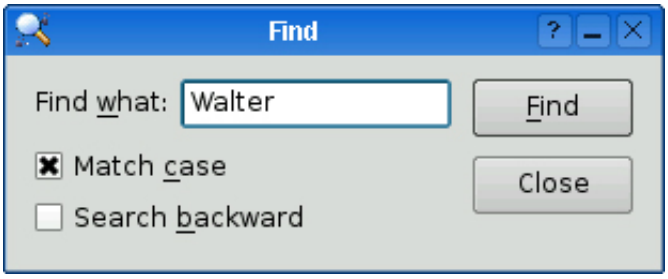


图 3.12: Spreadsheet 应用程序的 Find 对话框

创建非模态对话框时，通常会把它的信号连接到能够对用户的交互做出响应的那些槽上。

```
77 void MainWindow::find()
78 {
79     if (!findDialog) {
80         findDialog = new FindDialog(this);
81         connect(findDialog, SIGNAL(findNext(const QString &,
82                                            Qt::CaseSensitivity)),
83                 spreadsheet, SLOT(findNext(const QString &,
84                                           Qt::CaseSensitivity)));
85         connect(findDialog, SIGNAL(findPrevious(const QString &,
86                                                Qt::CaseSensitivity)),
87                 spreadsheet, SLOT(findPrevious(const QString &,
88                                               Qt::CaseSensitivity)));
89     }
90
91     findDialog->show();
92     findDialog->raise();
93     findDialog->activateWindow();
94 }
```

Find 对话框是一个可以让用户在电子制表软件中搜索文本的窗口。当用户单击 Edit→Find 时，就会调用 find() 槽来弹出 Find 对话框。这时，就可能出现下列几种情形：

- 这是用户第一次调用 Find 对话框。
- 以前曾经调用过 Find 对话框，但用户关闭了它。
- 以前曾经调用过 Find 对话框，并且现在它还是可见的。

如果 Find 对话框还不曾存在过，就可以创建它并且把它的 findNext() 信号和 findPrevious() 信号与 Spreadsheet 中相对应的那些槽连接起来。本应该在 MainWindow 的构造函数中创建这个对话框，但是推迟对话框的创建过程将可以使程序的启动更加快速。还有，如果从来没有使用到这个对话框，那么它就决不会被创建，这样可以既节省时间又节省内存。

然后，调用 show()、raise() 和 activateWindow() 来确保窗口位于其他窗口之上并且是可见的和激活的。只调用 show() 就足以让一个隐藏窗口变为可见的、位于最上方并且是激活的，但是也有可能是在 Find 对话框窗口已经是可见的时候又再次调用了它，在这种情况下，show() 调用可能什么也不做，那么就必须调用 raise() 和 activateWindow() 让窗口成为顶层窗口和激活状态。还有另外一种方法，本可以写成：

```
if (findDialog->isHidden()) {
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->activateWindow();
}
```

但这样的程序就好像明明在穿越单行道却又同时去看这条单行道的两个方向一样，显得多余。

现在看一下如图 3.13 所示的 Go to Cell 对话框。我们希望用户可以弹出、使用和关闭它，但是却不希望让这个窗口能够与应用程序中的其他窗口相互切换。也就是说，Go to Cell 对话框窗口必须是模态 (modal) 的。模态窗口就是一个在得到调用可以弹出并可以阻塞应用程序的窗口，从而会从调用发生开始起妨碍其他的任意处理或者交互操作，直到关闭该窗口为止。前面使用的文件对话框和消息框就是模态的。

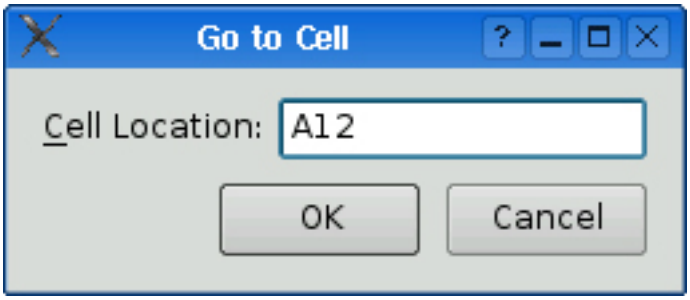


图 3.13: Spreadsheet 应用程序的 Go to Cell 对话框

如果对话框是通过 show() 调用的，那么它就是非模态对话框 [除非此后又调用了 setModal()，才会让它变为模态对话框]。但是，如果它是通过 exec() 调用的，那么该对话框就会是模态对话框。

```
96  void MainWindow::goToCell()
97  {
98      GoToCellDialog dialog(this);
99      if (dialog.exec()) {
100         QString str = dialog.lineEdit->text().toUpper();
101         spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
102                                     str[0].unicode() - 'A');
103     }
104 }
```

如果对话框被接受，函数 QDialog::exec() 可返回一个 true 值 (QDialog::Accepted)，否则就会返回一个 false 值 (QDialog::Rejected)。可以回想一下，当初在第 2 章利用 Qt 设计师创建 Go to Cell 对话框时，就曾经把 OK 连接到 accept()，把 Cancel 连接到 reject()。如果用户选择 OK，就把当前单元格的值设置成行编辑器中的值。

QTable::setCurrentCell() 函数需要两个参数：一个行索引和一个列索引。在 Spreadsheet 应用程序中，单元格 A1 就是单元格 (0,0)，单元格 B27 就是单元格 (26,1)。为了从函数 QLineEdit::text() 返回的 QString 中获得行索引，可以使用 QString::mid() 来提取行号（这个函数将返回一个从字符串的开始直到末尾位置的子字符串），然后使用 QString::toInt() 把它转换成一个整数值，并且把该值再减去 1。对于列号，则可以用这个字符串中第一个字符的大写数值减去字符'A'

的数值而得到。我们知道，该字符串将具有正确的格式，因为为对话框创建了一个 QRegExpValidator 检验器，只有满足一个字符后面再跟至多三个数字格式的字符串才能让 OK 按钮起作用。

goToCell() 函数与目前看到的所有代码都有些不同，因为它在堆栈中创建了一个作为变量的窗口部件（一个 GoToCellDialog）。虽然多使用了一行代码，但是换来了使用 new 和 delete 的简便：

```
void MainWindow::goToCell()
{
    GoToCellDialog *dialog = new GoToCellDialog(this);
    if (dialog->exec()) {
        QString str = dialog->lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                str[0].unicode() - 'A');
}
    delete dialog;
}
```

由于在使用完一个对话框（或者菜单）后，通常就不再需要它了，所以在堆栈中创建对话框（和上下文菜单）是一种常见的编程模式。并且对话框会在作用域结束后自动销毁掉。

现在转到 Sort 对话框上。Sort 对话框是一个模态对话框，它允许用户在当前的选定区域中使用给定的列进行排序。图 3.14 给出了一个排序的实例，用列 B 作为排序的主键，列 A 作为排序的第二键（两个都采用升序）。

	A	B	C
1	George	Washington	1789-1797
2	John	Adams	1797-1801
3	Thomas	Jefferson	1801-1809
4	James	Madison	1809-1817
5	James	Monroe	1817-1825
6	John Quincy	Adams	1825-1829
7	Andrew	Jackson	1829-1837
8			

(a) Before sort

	A	B	C
1	John	Adams	1797-1801
2	John Quincy	Adams	1825-1829
3	Andrew	Jackson	1829-1837
4	Thomas	Jefferson	1801-1809
5	James	Madison	1809-1817
6	James	Monroe	1817-1825
7	George	Washington	1789-1797
8			

(b) After sort

图 3.14: 对电子制表软件的选定区域进行排序

```
106 void MainWindow::sort()
107 {
108     SortDialog dialog(this);
109     QTableWidgetSelectionRange range = spreadsheet->selectedRange();
110     dialog.setColumnRange('A' + range.leftColumn(),
111                           'A' + range.rightColumn());
112 
113     if (dialog.exec()) {
114         SpreadsheetCompare compare;
115         compare.keys[0] =
116               dialog.primaryColumnCombo->currentIndex();
117         compare.keys[1] =
118               dialog.secondaryColumnCombo->currentIndex() - 1;
119         compare.keys[2] =
120               dialog.tertiaryColumnCombo->currentIndex() - 1;
121         compare.ascending[0] =
122               (dialog.primaryOrderCombo->currentIndex() == 0);
123         compare.ascending[1] =
124               (dialog.secondaryOrderCombo->currentIndex() == 0);
125         compare.ascending[2] =
126               (dialog.tertiaryOrderCombo->currentIndex() == 0);
127         spreadsheet->sort(compare);
128     }
129 }
```

sort() 函数中的代码使用了一种和 goToCell() 函数中用到的类似模式：

- 在堆栈中创建对话框并且对其进行初始化。
- 使用 exec() 弹出对话框。
- 如果用户单击 OK，就从对话框的各个窗口部件中提取并且使用这些用户输入的值。

setColumnRange() 调用将那些可用于排序的列变量设置为选定的列。例如，

使用图 3.14 中的选择，range.leftColumn() 将返回值 0，即 $'A' + 0 = 'A'$，并且 range. rightColumn() 将返回值 2，即 $'A' + 2 = 'C'$。

compare 对象储存了主键、第二键和第三键以及它们的排序顺序。（将会在下一章中看到 SpreadsheetCompare 类的定义。）这个对象会由 Spreadsheet::sort() 使用，用于两行的比较。keys 数组存储了这些键的列号。例如，如果选择区域是从 C2 扩展到 E5，那么列 C 的位置就是 0。ascending 数组中按 bool 格式存储了和每一个键相关的顺序。QComboBox::currentIndex() 返回当前选定项的索引值，该值是一个从 0 开始的数。对于第二键和第三键，考虑到”None” 项，我们从当前项减去 1。

sort() 函数会完成这项工作，但是它显得稍有不足。它认为 Sort 对话框是按照一种特定的方式来实现的，也就是像上面那样来处理组合框和”None” 项。这就意味着，如果重新设计了 Sort 对话框，也许就需要重新编写这些代码。如果对话框只会从一个地方调用，那么这样的方式应该是足够了，但是如果对话框可能会在几个地方调用到，那么这种处理方式就等于打开了维护工作的梦魇之门。

一种更为稳健的方法是让 SortDialog 类具有自适应性，这可以通过让它自己创建一个 SpreadsheetCompare 对象，然后使这个对象只能被它的调用者使用来做到这一点。这样就可以有效地简化 MainWindow::sort() 函数：

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());
    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

这种方法可产生松散的耦合组件，并且当从多个地方调用该对话框对，它几乎总可以做出正确的选择。

一种更为极端的方式是在初始化 SortDialog 对象的时候就为其传递一个指向 Spreadsheet 对象的指针，并且允许对话框直接对 Spreadsheet 进行操作。这样做

会使 SortDialog 少一些通用性，因为它仅能适用于一种类型的窗口部件，但是通过去除 SortDialog::setColumnRange() 函数，它的确是进一步简化了程序代码。于是，现在的 MainWindow::sort() 函数将变成如下所示的样子：

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

相比而言，第一种方法中调用者需要知道与这个对话框相关的暗示信息，而第二种方法中的对话框需要知道由调用者所提供的与数据结构相关的暗示信息。在对话框需要作用于现场变化的地方，这种方法显得更为有用些。但是就像第一种方法中调用者的代码功能不足一样，如果数据结构发生了变化，则第三种方法也会失效。

一些开发者只会选用一种对话框处理方法并对其持之以恒。这有一个好处，就是能够精通和简练处理方法，因为所有的对话框都使用的是同一种处理模式，但是这也会失去对调用对话框时没有用到的那些其他有益处理方法。理想情况下，应根据每一个对话框的自身来选择应当使用的对话框处理方法。

我们将用 About 对话框来圆满结束这一节。可以创建一个像 Find 或 Go to Cell 对话框那样的自定义对话框来显示应用程序的有关信息，但是因为绝大多数 About 对话框都具有较为固定的格式，所以 Qt 提供了一种更为简单的解决方案。

```
131 void MainWindow::about()
132 {
133     QMessageBox::about(this, tr("About Spreadsheet"),
134             tr("<h2>Spreadsheet 1.1</h2>"
135                "<p>Copyright &copy; 2008 Software Inc."
136                "<p>Spreadsheet is a small application that "
137                "demonstrates QAction, QMainWindow, QMenuBar, "
```

```
138                 "QStatusBar, QTableWidget, QToolBar, and many other "
139                 "Qt classes."));
140 }
```

通过调用一个方便的静态函数 QMessageBox::about()，就可以获得 About 对话框。这个函数和 QMessageBox::warning() 的形式非常相似，只是它使用了父窗口的图标，而不是标准的“警告”图标。这个对话框的最终结果显示在图 3.15 中。

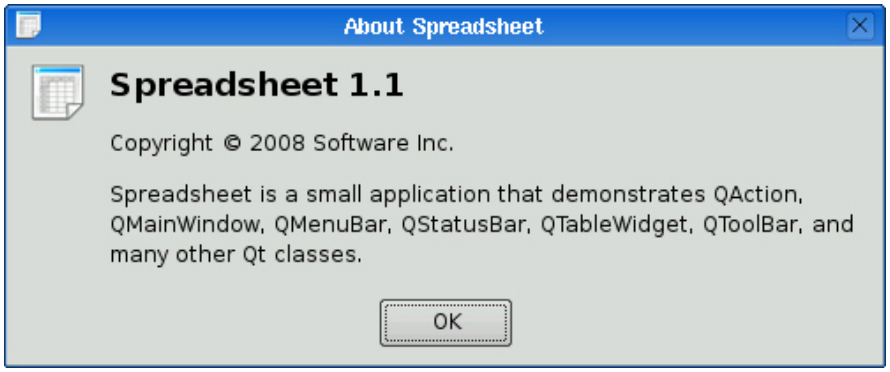


图 3.15: Spreadsheet 的 About 对话框

到现在为止，已经使用了由 QMessageBox 和 QFileDialog 提供的多个方便的静态函数。这些函数可以创建一个对话框，初始化它，并且可以对它调用 exec()。当然，也可以像创建其他任意窗口部件一样创建 QMessageBox 或者 QFileDialog 窗口部件，然后再明确地对它调用 exec()，或者甚至是 show()，尽管这样的处理方式会显得有些不大方便。

3.6 存储设置

在 MainWindow 的构造函数中，调用了 readSettings() 来载入应用程序存储的那些设置。与之相似的是，在 closeEvent() 中，调用 writeSettings() 来保存这些设置。这两个函数是最后两个需要实现的 MainWindow 成员函数。

```
406 void MainWindow::writeSettings()
407 {
408     QSettings settings("Software Inc.", "Spreadsheet");
409 
410     settings.setValue("geometry", saveGeometry());
411     settings.setValue("recentFiles", recentFiles);
412     settings.setValue("showGrid", showGridAction->isChecked());
413     settings.setValue("autoRecalc", autoRecalcAction->isChecked());
414 }
```

writeSettings() 函数保存了主窗口的几何形状（位置和尺寸大小）、最近打开文件列表以及 ShowGrid 和 Auto-Recalculate 选项的设置值。

默认情况下，QSettings 会存储应用程序中与特定平台相关的一些设置信息。在 Windows 系统中，它使用的是系统注册表；在 UNIX 系统中，它会把设置信息存储在文本文件中；在 Mac OS X 中，它会使用 Core Foundation Preferences 的应用程序编程接口。

构造函数的参数说明了组织的名字和应用程序的名字。采用与平台相关的方式，可以利用这一信息查找这些设置所在的位置。

QSettings 把设置信息存储为键值对 (key-value pair) 的形式。键 (key) 与文件系统的路径很相似。可以使用路径形式的语法（例如，findDialog/matchCase）来指定子键 (subkey) 的值，或者也可以使用 beginGroup() 和 endGroup() 的形式：

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

值 (value) 可以是一个 int、bool、double、QString、QStringList 或者是 QVariant 所支持的其他任意类型，包括那些已经注册过的自定义类型。

```
390 void MainWindow::readSettings()
391 {
392     QSettings settings("Software Inc.", "Spreadsheet");
393 
394     restoreGeometry(settings.value("geometry").toByteArray());
395 
396     recentFiles = settings.value("recentFiles").toStringList();
397     updateRecentFileActions();
398 
399     bool showGrid = settings.value("showGrid", true).toBool();
400     showGridAction->setChecked(showGrid);
401 
402     bool autoRecalc = settings.value("autoRecalc", true).toBool();
403     autoRecalcAction->setChecked(autoRecalc);
404 }
```

readSettings() 函数可以载入之前使用 writeSettings() 函数所保存的那些设置。value() 函数中的第二个参数可以在没有可用设置的情况下指定所需的默认值。在应用程序第一次运行时，使用的就是这些默认值。由于没有为形状或者最近打开文件列表指定第二个参数，所以在第一次运行时，窗口会使用任意但是却合理的大小和位置，而最近文件列表会是一个空表。

在 readSettings() 和 writeSettings() 中使用与 QSettings 相关的全部代码为 MainWindow 所选择的布置方案，都只是许多可用方案中的一种而已。可以在应用程序执行期间的任何时候和程序代码中的任何地方，随时随地创建一个 QSettings 对象，用它查询或者修改一些设置。

现在已经完成了对 Spreadsheet 的 MainWindow 的实现。在后续的几节中，将会讨论如何修改 Spreadsheet 应用程序来让它可以处理多文档以及如何实现一个程序启动画面 (splash screen)。将会在下一章中完成它的功能，包括公式和排序的处理。

3.7 多文档

现在，已经为编写 Spreadsheet 应用程序的 main() 函数的代码做好了准备：

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

这个 main() 函数和以前曾经写过的那些函数稍微有点不同：以变量的形式在堆栈里创建 MainWindow 实例，而不是使用 new 来创建它。当函数结束时，MainWindow 实例会自动销毁。

就像上面 main() 函数所显示的那样，Spreadsheet 应用程序只提供了一个单一主窗口，并且在同一时间只能处理一个文档。如果想让它在同一时间具有处理多个文档的能力，就需要同时启动多个 Spreadsheet 应用程序实例。但是这对于用户来讲是很不方便的，用户需要的是一个可以处理多个文档的单一应用程序实例，就像在一个网页浏览器实例中可以同时提供多个浏览器窗口一样。

下面将修改 Spreadsheet 应用程序，以使它可以处理多个文档。首先，需要对 File 菜单做一些简单改动：

- 利用 File→New 创建一个空文档主窗口，而不是再次使用已经存在的主窗口。
- 利用 File→Close 关闭当前主窗口。
- 利用 File→Exit 关闭所有窗口。

在 File 菜单的最初版本中，并没有 close 选项，这只是因为当时它还和 Exit 一样具有相同的功能。新的 File 菜单如图 3.16 所示。

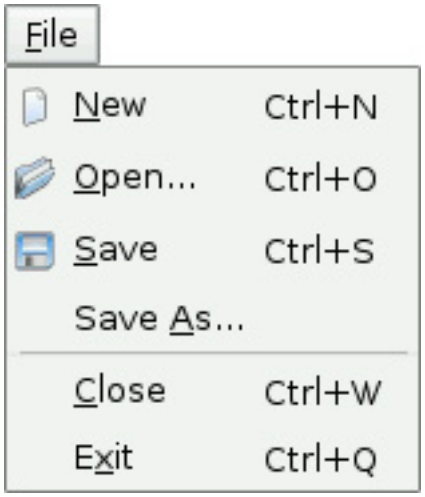


图 3.16: 新的 File 菜单

新的 main() 函数为:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    return app.exec();
}
```

具有多窗口功能后，现在就需要使用菜单中的 new 来创建 MainWindow。考虑到节省内存，可以在工作完成之后使用 delete 操作删除主窗口。

这是新的 MainWindow::newFile() 槽:

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}
```

我们只创建了一个新的 MainWindow 实例。这看起来有些奇怪；因为没有保留指向这个新窗口的任何指针，但实际上这并不是什么问题，因为 Qt 会对所有的窗口进行跟踪。

以下是用于 Close 和 Exit 的动作：void MainWindow::createActions()

```
{
    ...
    closeAction = new QAction(tr("&Close"), this);
    closeAction->setShortcut(QKeySequence::Close);
    closeAction->setStatusTip(tr("Close this window"));
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));
    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
        qApp, SLOT(closeAllWindows()));
    ...
}
```

槽 QApplication::closeAllWindows() 会关闭所有应用程序的窗口，除非其中一个应用程序拒绝了这个关闭事件。这正是在此所需的行为。我们不用再考虑那些有关是否保存的事情，因为就算关闭一个窗口，都会在 MainWindow::closeEvent() 中处理这些情况。

看起来好像已经完成了应用程序对多窗口处理能力的工作。遗憾的是，这里还隐藏着一个潜在的问题：如果用户一直创建并且关闭主窗口，那么这台机器早晚会耗尽它的全部内存。这是因为我们保存了 newFile() 中创建的 MainWindow 窗口部件，但是从没有删除它们。当用户关闭一个主窗口时，默认行为是隐藏它，所以它还会保留在内存中。对于如此多的主窗口，的确会造成一定的问题。

解决办法是在构造函数中对 Qt::WA_DeleteOnClose 的属性进行设置：

```
MainWindow::MainWindow()
{
```

```
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

这样做就会告诉 Qt 在关闭窗口时将其删除。Qt::WA_DeleteOnClose 属性是可以在 QWidget 上进行设置并用来影响这个窗口部件的行为的诸多标记之一。

内存泄漏并不是必须处理的唯一问题。最初的应用程序设计包含了一个隐含的假设，也就是它仅有一个主窗口。对于多窗口，每一个主窗口都有它自己的最近打开文件列表和它自己的一些选项。很明显，最近打开文件列表对于整个应用程序来说应该是全局的。通过把 recentFiles 变量声明为静态变量，可以相当容易地解决这个问题，这样对于整个应用程序来说，会只存在一个该列表的实例。但随后必须确保的是：无论何时调用 updateRecentFileActions() 函数来更新 File 菜单，都必须是在所有的主窗口上调用它。这是实现这一做法的代码：

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

这段代码使用了 Qt 的 foreach 结构体（将在第 11 章中对其进行说明）来遍历这个应用程序的所有窗口，并且对所有类型为 Mainwindow 的窗口部件调用 updateRecentFileActions()。可以使用类似的代码来同步 Show Grid 和 Auto-Recalculate 选项，或者用于确保同一个文件不会被加载两次。

在每一个主窗口中只提供一个文档的应用程序称为单文档界面 (single document interface，SDI) 应用程序。在 Windows 系统下，一种常用的替代方法是多文档界面 (multiple document inteface，MDI)，这种应用程序只有一个单一的主窗口，但可以对主窗口中央区域的多个文档窗口进行管理。Qt 可以在它支持的所有平台上创建 SDI 和 MDI 应用程序。图 3.17 给出了使用这两种方法的 Spreadsheet 应用程序。第 6 章将对 MDI 进行说明。

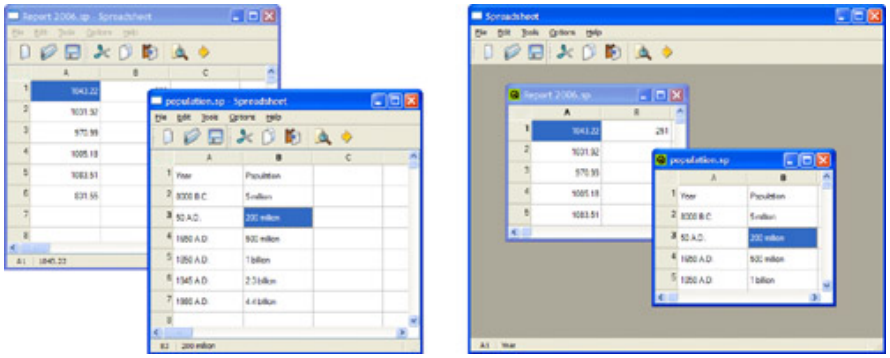


图 3.17: 单文档界面和多文档界面

3.8 程序启动画面

许多应用程序都会在启动的时候显示一个程序启动画面 (splash screen)，图 3.18 给出的就是这样的一个实例。一些程序员使用程序启动画面对缓慢的启动过程进行掩饰，而另外一些人则是用于满足市场部门的要求。使用 QSplashScreen 类，可以非常容易地为 Qt 应用程序添加一个程序启动画面。

图 3.18: 程序启动画面

类 QSplashScreen 会在应用程序的主窗口出现之前显示一个图片。它也可以在这个图片上显示一些消息，用来通知用户有关应用程序初始化的过程。通常，程序启动画面的代码会放在 main() 函数中，位于 QApplication::exec() 调用之前。

下面给出了一个 main() 函数的例子，在应用程序中，它使用 QSplashScreen 显示的程序启动画面表示启动时载入的一些模块和网络连接的建立。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();
    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;
    splash->showMessage(QObject::tr("Setting up the main window..."),
                topRight, Qt::white);
    MainWindow mainWin;
    splash->showMessage(QObject::tr("Loading modules..."),
                topRight, Qt::white);
    loadModules();
    splash->showMessage(QObject::tr("Establishing connections..."),
                topRight, Qt::white);
    establishConnections();
    mainWin.show();
    splash->finish(&mainWin);
    delete splash;
    return app.exec();
}
```

到此为止，我们已经创建了 Spreadsheet 应用程序的用户界面。在下一章中，将会通过实现电子制表软件的核心功能来完成这个应用程序。

4 实现应用程序的功能

前两章说明了如何创建 Spreadsheet 应用程序的用户界面。在这一章中，将通过编写它的底层功能函数来完成这个程序。此外，还将看到如何载人和保存文件，如何内存中存储数据，如何实现剪贴板操作，以及如何向 QTableWidget 中添加对电子制表软件公式的支持等功能。

4.1 中央窗口部件

QMainWindow 的中央区域可以被任意种类的窗口部件所占用。下面给出的是对所有可能情形的概述。

1. **使用一个标准的 Qt 窗口部件**

   像 QTableWidget 或者 QTextEdit 这样的标准窗口部件可以用作中央窗口部件。在这种情况下，这个应用程序的功能，如文件的载入和保存，必须在其他地方实现（例如，在 QMainWindow 的子类中）

2. **使用一个自定义窗口部件**

   特殊的应用程序通常需要在自定义窗口部件中显示数据。例如，一个图标编辑器程序就应当使用一个 IconEditor 窗口部件作为自己的中央窗口部件。第 5 章将会说明如何在 Qt 中编写自定义窗口部件。

3. **使用一个带布局管理器的普通 QWidget**

   有时，应用程序的中央区域会被许多窗口部件所占用。这时可以通过使用一个作为所有这些其他窗口部件父对象的 QWidget，以及通过使用布局管理器管理这些子窗口部件的大小和位置来完成这一特殊情况。

4. **使用切分窗口 (splitter)**

多个窗口部件一起使用的另一种方法是使用 QSplitter。QSplitter 会在水平方向或者竖直方向上排列它的子窗口部件，用户可以利用切分条 (splitter handle) 控制它们的尺寸大小。切分窗口可以包含所有类型的窗口部件，包括其他切分窗口。

5. **使用多文档界面工作空间**

   如果应用程序使用的是多文档界面，那么它的中央区域就会被 QMdiArea 窗口部件所占据，并且每个多文档界面窗口都是它的一个子窗口部件。

布局、切分窗口和多文档界面工作空间都可以与标准的 Qt 窗口部件或者自定义窗口部件组合使用。第 6 章将会进一步深入地讲解这些类。

对于 Spreadsheet 应用程序，会使用一个 QTableWidget 子类作为它的中央窗口部件。类 QTableWidget 已经提供了我们所需要的绝大多数电子制表软件的功能，但是它还不支持剪贴板操作，并且也不能理解诸如”=A1+A2+A3” 这样的电子制表软件公式的意义。我们将会在 Spreadsheet 类中实现这些缺少的功能。

4.2 子类化 QTableWidget

类 Spreadsheet 派生自 QTableWidget，如图 4.1 所示。

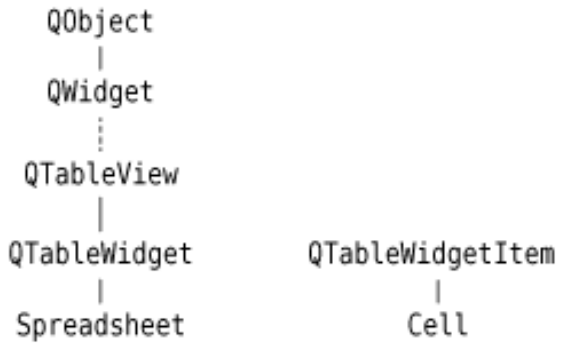


图 4.1: 类 Spreadsheet 和 Cell 的继承树

QTableWidget 是一组格子，可以非常有效地用来表达二维稀疏数组。它可以在规定的维数内显示用户滚动到的任一单元格。当用户在一个空单元格内输入一些文本的时候，QTableWidget 会自动创建一个用来存储这些文本的 QTableWidgetItem。

QTableWidget 派生自 QTableView，它是模型/视图类之一，我们将在第 10 章进一步了解它。对于另外一个表 QicsTable，它有更多的非常规功能，可以从 http://www.ics.com/中获取。

让我们一起来实现 Spreadsheet，首先从它的头文件开始：

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <QTableWidget>

class Cell;
class SpreadsheetCompare;
```

头文件是从 Cell 和 SpreadsheetCompare 类的前置声明开始的。

QTableWidget 单元格的属性，比如它的文本和对齐方式等，都存储在 QTableWidgetItem 中。与 QTableWidget 不同的是，QTableWidgetItem 不是一个窗口部件类，而是一个纯粹的数据类。Cell 类派生自 QTableWidgetItem，会在本章的最后一节对这个 Cell 类进行解释。

```
class Spreadsheet : public QTableWidget
{
    Q_OBJECT

public:
    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; }
    QString currentLocation() const;
    QString currentFormula() const;
    QTableWidgetSelectionRange selectedRange() const;
    void clear();
```

```
21      bool readFile(const QString &fileName);
22      bool writeFile(const QString &fileName);
23      void sort(const SpreadsheetCompare &compare);
```

之所以把 autoRecalculate() 函数实现为内联函数，是因为无论自动重新计算的标识符生效与否，它都必须要有返回值。

在第 3 章中，当实现 MainWindow 时，我们依赖于 Spreadsheet 中的一些公有函数。例如，我们从 MainWindow::newFile() 中调用 clear() 来重置电子制表软件。也使用了一些从 QTableWidget 中继承而来的函数，特别是 setCurrentCell() 和 setShowGrid()。

```
25  public slots:
26      void cut();
27      void copy();
28      void paste();
29      void del();
30      void selectCurrentRow();
31      void selectCurrentColumn();
32      void recalculate();
33      void setAutoRecalculate(bool recalc);
34      void findNext(const QString &str, Qt::CaseSensitivity cs);
35      void findPrevious(const QString &str, Qt::CaseSensitivity cs);
36
37  signals:
38      void modified();
```

Spreadsheet 提供了许多实现 Edit、Tools 和 Options 菜单中的动作的槽：并且它也提供了一个 modified() 信号，用来告知用户，可能已经发生的任何变化。

```
40  private slots:
41      void somethingChanged();
```

还定义了一个由 Spreadsheet 类内部使用的私有槽：

```
43 private:
44     enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };
45 
46     Cell *cell(int row, int column) const;
47     QString text(int row, int column) const;
48     QString formula(int row, int column) const;
49     void setFormula(int row, int column, const QString &formula);
50 
51     bool autoRecalc;
52 };
```

在这个类的私有段中，声明了 3 个常量、4 个函数和 1 个变量。

```
54 class SpreadsheetCompare
55 {
56 public:
57     bool operator()(const QStringList &row1,
58                     const QStringList &row2) const;
59 
60     enum { KeyCount = 3 };
61     int keys[KeyCount];
62     bool ascending[KeyCount];
63 };
64 
65 #endif
```

在这个头文件的最后，给出了 SpreadsheetCompare 类的定义。当查看 Spreadsheet::sort() 时，会解释这个类。

现在来看一下它的实现文件：

```
#include <QtGui>

#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidget(parent)
{
    autoRecalc = true;

    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);

    connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
            this, SLOT(somethingChanged()));

    clear();
}
```

通常情况下，当用户在一个空单元格中输入一些文本的时候，QTableWidget 将会自动创建一个 QTableWidgetItem 来保存这些文本。在电子制表软件中，我们想利用将要创建的 Cell 项来代替 QTableWidgetItem。这可以通过在构造函数中调用 setItemPrototype() 来完成。实际上，QTableWidget 会在每次需要新项的时候把所传递的项以原型的形式克隆出来。

同样是在构造函数中，我们将选择模式设置为 QAbstractItemView::ContiguousSelection，从而可以允许简单矩形选择框方法。我们把表格窗口部件的 itemChanged() 信号连接到私有槽 somethingChanged() 上，这可以确保在用户编辑一个单元格的时候，somethingChanged() 槽可以得到调用。最后，调用 clear() 来重新调整表格的尺寸大小并且设置列标题。

```
39 void Spreadsheet::clear()
40 {
41     setRowCount(0);
42     setColumnCount(0);
43     setRowCount(RowCount);
44     setColumnCount(ColumnCount);
45 
46     for (int i = 0; i < ColumnCount; ++i) {
47         QTableWidgetItem *item = new QTableWidgetItem;
48         item->setText(QString(QChar('A' + i)));
49         setHorizontalHeaderItem(i, item);
50     }
51 
52     setCurrentCell(0, 0);
53 }
```

clear() 函数是从 Spreadsheet 构造函数中得到调用的，用来初始化电子制表软件。它也会在 MainWindow::newFile() 中得到调用。

我们原本使用 QTableWidget::clear() 来清空所有项和任意选择，但是那样做的话，这些标题将会以当前大小的尺寸而被留下。相反的是，我们要把表格向下调整为 0×0。这样就可以完全清空整个表格，包括这些标题。然后，重新调整表的大小为 ColumnCount × RowCount(26 × 999)，并且把 QTableWidgetItem 水平方向上的标题修改为列名”A”，”B”，$\cdots$，”Z”。不需要设置垂直标题的标签，因为这些标签的默认值是”1”，”2”，$\cdots$，”999”。最后，把单元格光标移动到单元格 A1 处。

QTableWidget 由多个子窗口部件构成。在它的顶部有一个水平的 QHeaderView，左侧有一个垂直的 QHeaderView，还有两个 QScrollBar。在它的中间区域被一个名为视口 (viewport) 的特殊窗口部件所占用，QTableWidget 可以在它上面绘制单元格。通过从 QTableView 和 QAbstractScrollArea 中继承的一些函数，可以访问这些不同的子窗口部件（参见图 4.2）。QAbstractScrollArea 提供了一个可以滚动的视口和两个可以打开或关闭的滚动条。第 6 章将讲述 QScrollArea 子类。

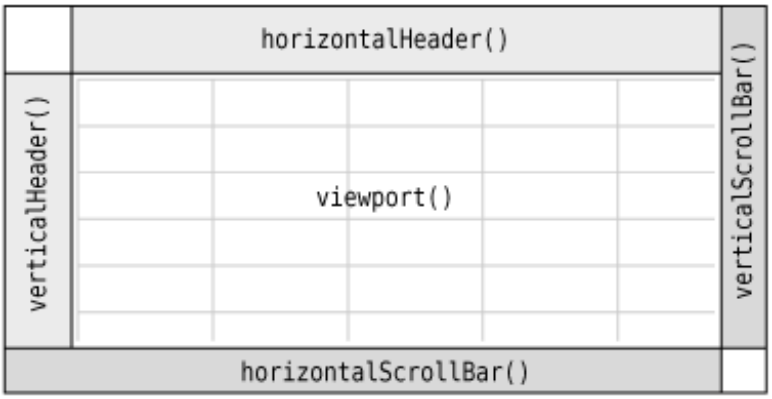


图 4.2: 构成 QTableWidget 的各个窗口部件

```
286 Cell *Spreadsheet::cell(int row, int column) const
287 {
288     return static_cast<Cell *>(item(row, column));
289 }
```

cell() 私有函数可以根据给定的行和列返回一个 Cell 对象。它几乎和 QTableWidget::item() 函数的作用一样，只不过它返回的是一个 Cell 指针，而不是一个 QTableWidgetItem 指针。

```
312 QString Spreadsheet::text(int row, int column) const
313 {
314     Cell *c = cell(row, column);
315     if (c) {
316         return c->text();
317     } else {
318         return "";
319     }
320 }
```

text() 私有函数可以返回给定单元格中的文本。如果 cell() 返回的是一个空指针，则表示该单元格是空的，因而返回一个空字符串。

```
302 QString Spreadsheet::formula(int row, int column) const
303 {
304     Cell *c = cell(row, column);
305     if (c) {
306         return c->formula();
307     } else {
308         return "";
309     }
310 }
```

formula() 函数返回给定单元格中的公式。在很多情况下，公式和文本是相同的。例如，公式"Hello" 等价于字符串"Hello"，所以如果用户在单元格中输入"Hello" 并且按下回车键，那么该单元格就会显示文本"Hello"。但是还有一些例外的情况：

- 如果公式是一个数字，那么它就会被认为是一个数字。例如，公式"1.50" 等价于双精度实数 (double) 的 1.5，它在电子制表软件中会被显示为右对齐的"1.5"。
- 如果公式以单引号开始，那么公式的剩余部分将会被认为是文本。例如，公式"'12345" 等价于字符串"12345"。
- 如果公式以等号开始，那么公式将会被认为是一个算术公式。例如，如果单元格 A1 包含"12" 并且单元格 A2 包含"6"，那么公式"= A1+A2" 就会等于 18。

把公式转换成值的任务是由 Cell 类完成的。这时，要记住的事情是显示在单元格内的文本是公式的结果，而不是公式本身。

```
291 void Spreadsheet::setFormula(int row, int column,
292                                const QString &formula)
293 {
294     Cell *c = cell(row, column);
```

```
295     if (!c) {
296         c = new Cell;
297         setItem(row, column, c);
298     }
299     c->setFormula(formula);
300 }
```

setFormula() 私有函数可以设置用于给定单元格的公式。如果该单元格已经有一个 Cell 对象，那么我们就重新使用它。否则，可以创建一个新的 Cell 对象并且调用 QTableWidget::setItem() 把它插入到表中。最后，调用该单元格自己的 setFormula() 函数，但如果这个单元格已经显示在屏幕上，那么就重新绘制它。我们不需要担心随后对这个 Cell 对象的删除操作。因为 QTableWidget 会得到这个单元格的所有权，并且会在正确的时候自动将其删除。

```
20 QString Spreadsheet::currentLocation() const
21 {
22     return QChar('A' + currentColumn())
23            + QString::number(currentRow() + 1);
24 }
```

currentLocation() 函数返回当前单元格的位置，它是按照电子制表软件的通常格式，也就是一个列字母后跟上行号的形式来表示这个位置的值。MainWindow::updateStatusBar() 使用它把这个单元格的位置显示在状态栏上。

```
26 QString Spreadsheet::currentFormula() const
27 {
28     return formula(currentRow(), currentColumn());
29 }
```

currentFormula() 函数返回当前单元格的公式。它是从 MainWindow::updateStatusBar() 中得到调用的。

```
279 void Spreadsheet::somethingChanged()
280 {
281     if (autoRecalc)
282         recalculate();
283     emit modified();
284 }
```

如果启用了”auto-recalculate”（自动重新计算），那么 somethingChanged-() 私有槽就会重新计算整个电子制表软件。它也会发射 modified() 信号。

把数据存储为项

在 Spreadsheet 应用程序中，每一个非空单元格都被当作一个独立的 QTableWidgetItem 对象而保存在内存中。把数据存储为项 (item) 是一种对 QListWidgetItem 和 QTreeWidgetItem 进行操作的方法，该方法也可用于 QListWidget 和 QTreeWidget。

Qt 的项类可以用作非常规的数据持有者。例如，一个 QTableWidgetItem 已经存储了一些属性，其中包括一个字符串、一种字体、一种颜色和一个图标，以及一个返回到 QTableWidget 的指针。项也可以保存数据（QVariant 型），包括一些已经注册过的自定义类型，以及通过项类的子类化，我们还可以提供其他功能。

许多老一点的工具包在它们的项类中提供一个 void 指针来存储自定义数据。在 Qt 中，更为自然的方法是使用带 QVariant 的 setData()，但如果需要一个 void 指针，那么可以通过子类化一个项类并且添加一个 void 指针成员变量来很简单地实现这一点。

对于更具有挑战性的数据处理需求，比如大数据集、复杂数据项、数据库集成以及多数据视图等，Qt 提供了一套模型/视图 (model/view) 类，利用这些类可以把数据从它们的直观表示中分离出来。这些内容将会在第 10 章中加以讲述。

4.3 载入和保存

现在，我们将使用一种自定义的二进制数格式来实现 Spreadsheet 文件的载入和保存。将使用 QFile 和 QDataStream 来完成这一工作，由它们共同提供与平台无关的二进制数输入/输出接口。

首先从一个 Spreadsheet 文件的输出开始：

```
92  bool Spreadsheet::writeFile(const QString &fileName)
93  {
94      QFile file(fileName);
95      if (!file.open(QIODevice::WriteOnly)) {
96          QMessageBox::warning(this, tr("Spreadsheet"),
97                               tr("Cannot write file %1:\n%2.")
98                               .arg(file.fileName())
99                               .arg(file.errorString()));
100         return false;
101     }
102
103     QDataStream out(&file);
104     out.setVersion(QDataStream::Qt_4_3);
105
106     out << quint32(MagicNumber);
107
108     QApplication::setOverrideCursor(Qt::WaitCursor);
109     for (int row = 0; row < RowCount; ++row) {
110         for (int column = 0; column < ColumnCount; ++column) {
111             QString str = formula(row, column);
112             if (!str.isEmpty())
113                 out << quint16(row) << quint16(column) << str;
114         }
115     }
```

```
116     QApplication::restoreOverrideCursor();
117     return true;
118 }
```

从 MainWindow::saveFile() 中调用的 writeFile() 函数把文件输出到磁盘中。如果输出成功，它会返回 true；如果出现错误，则返回 false。

我们使用给定的文件名创建一个 QFile 对象，并且调用 open() 打开这个用于输出的文件。我们也会创建一个 QDataStream 对象，由它操作这个 QFile 对象并且使用该对象输出数据。

在输出数据之前，我们把这个应用程序的光标修改为标准的等待光标（通常是一个沙漏），并且一旦所有的数据输出完毕，就需要把这个应用程序的光标重新恢复为普通光标。在函数的最后，文件会由 QFile 对象的析构函数自动关闭。

QDataStream 既可以支持 C++ 基本类型，也可以支持多种 Qt 类型。该语法模仿了标准 C++ 的 <iostream> 中的那些类的语法。例如：

```
out << x << y << z;
```

会把变量 x、y 和 z 输出到一个流中，而：

```
in >> x >> y >> z;
```

会从流中读出它们。因为 C++ 的基本类型在不同平台上可能会有不同的大小，所以把这些变量强制转换成 qint8、quint8、qint16、quint16、qint32、quint32、qint64 以及 quint64 中的一个是最安全的做法，这样做可以确保它们能够获得应有的大小（按位计算）。

Spreadsheet 应用程序的文件格式是相当简单的。一个 Spreadsheet 文件以一个 32 位数字作为文件的开始，由它确定文件的格式（MagicNumber，在 spreadsheet.h 中定义为 Ox7F51C883，它是一个任意的随机数）。然后是连续的数据块，每一数据块都包含了用于一个单元格中的行、列和公式。为了节省空间，我们没有输出空白单元格。该文件格式如图 4.3 所示。

图 4.3: Spreadsheet 的文件格式

关于这些数据类型的二进制数确切表示方法则是由 QDataStream 决定的。例如，一个 quint16 按照高字节在后的顺序存储为两个字节，而一个 QString 则被存储为字符串的长度后跟 Unicode 字符的形式。

关于 Qt 数据类型的二进制数确切表示方法，自 Qt1.0 以来已经发生了许多变化。而且在未来的 Qt 发行版中，为了能够与现存的数据类型和将来允许出现的新的 Qt 类型保持一致，这样的表示方法还可能会继续变化下去。默认情况下，QDataStream 会使用最近版本的二进制数格式（在 Qt 4.3 中的版本是第 9 版），但是可以设置它，使它可以读取那些旧的数据版本。如果以后有可能使用新的 Qt 发行版来重新编译这个应用程序，那么为了避免出现任何可能的兼容性问题，需要明确告诉 QDataStream 应该使用的是第 9 版，从而无需再考虑要使用的 Qt 版本。（QDataStream::Qt_4_3 是一个方便的常量，它就等于 9。）

QDataStream 的功能非常齐全。既可以把它用于 QFile 中，也可以把它用于 QBuffer、QProcess、QTcpSocket、QUdpSocket 或者 QSslSocket 中。在读取和输出文本文件时，Qt 也提供了一个 QTextStream 类，可以使用它代替 QDataStream 类。第 12 章将深入地讲解这些类，并且也会讲述处理不同的 QDataStream 版本时所使用的各种方法。

```
55 bool Spreadsheet::readFile(const QString &fileName)
56 {
57     QFile file(fileName);
58     if (!file.open(QIODevice::ReadOnly)) {
59         QMessageBox::warning(this, tr("Spreadsheet"),
60                              tr("Cannot read file %1:\n%2.")
61                              .arg(file.fileName())
62                              .arg(file.errorString()));
63         return false;
64     }
65
```

```
66      QDataStream in(&file);
67      in.setVersion(QDataStream::Qt_4_3);
68
69      quint32 magic;
70      in >> magic;
71      if (magic != MagicNumber) {
72          QMessageBox::warning(this, tr("Spreadsheet"),
73                               tr("The file is not a Spreadsheet file."));
74          return false;
75      }
76
77      clear();
78
79      quint16 row;
80      quint16 column;
81      QString str;
82
83      QApplication::setOverrideCursor(Qt::WaitCursor);
84      while (!in.atEnd()) {
85          in >> row >> column >> str;
86          setFormula(row, column, str);
87      }
88      QApplication::restoreOverrideCursor();
89      return true;
90  }
```

readFile() 函数与 writeFile() 函数非常相似。我们使用 QFile 读取一个文件，但这一次使用的是 QIODevice::ReadOnly 标记，而不是 QIODevice::WriteOnly 标记。然后，把 QDataStream 的版本设置为 9。用于读取文件的格式必须总是与输出文件的格式相同。

如果该文件在开始处具有正确的幻数 (magic number)，那么可以调用 clear() 来清空电子制表软件中的所有单元格，并且读入单元格中的数据。由于该文件中只包

含那些非空单元格的数据，并且也不大可能重置电子制表软件中的每个单元格，所以必须确保在读入数据之前已经清空了所有的单元格。

4.4 实现 Edit 菜单

现在，我们已经为实现响应应用程序 Edit 菜单中的各个槽做好了准备。Spreadsheet 应用程序中的 Edit 菜单如图 4.4 所示。

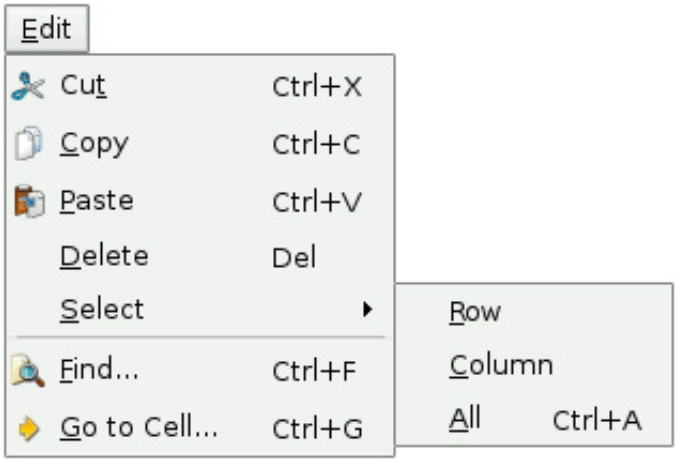


图 4.4: Spreadsheet 应用程序的 Edit 菜单

```
146 void Spreadsheet::cut()
147 {
148     copy();
149     del();
150 }
```

cut() 槽可以对 Edit→Cut 菜单做出响应。由于 Cut 的执行效果与 Copy 之后再加上一个 Delete 的执行效果相同，所以其实现代码很简单。

```
152 void Spreadsheet::copy()
153 {
154     QTableWidgetSelectionRange range = selectedRange();
```

```
155     QString str;
156
157     for (int i = 0; i < range.rowCount(); ++i) {
158         if (i > 0)
159             str += "\n";
160         for (int j = 0; j < range.columnCount(); ++j) {
161             if (j > 0)
162                 str += "\t";
163             str += formula(range.topRow() + i, range.leftColumn() + j);
164         }
165     }
166     QApplication::clipboard()->setText(str);
167 }
```

copy() 槽能够对 Edit→copy 做出响应。它会遍历当前选择（如果没有明确的选择，那么就认为选择的只是当前单元格）。每一个选中单元格的公式都会被添加到一个 QString 中，行与行之间利用换行符\n 分隔，列与列之间则以制表符\t 来分隔。图 4.5 给出了这一实现方法的示意图。

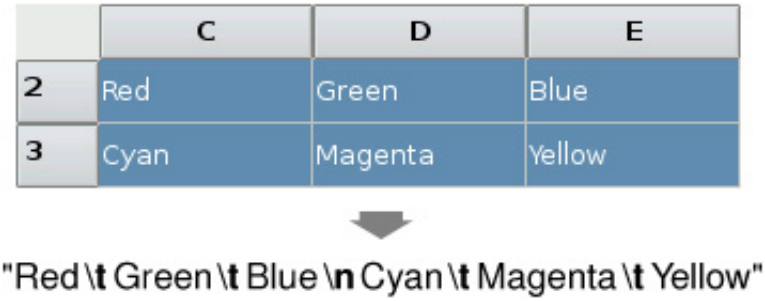


图 4.5: 把选择复制到剪贴板中

在 Qt 中，通过调用 QApplication::clipboard() 静态函数可以使用系统的剪贴板。通过调用 QClipboard::setText()，就既可以在本应用程序中又可以在其他应用程序中使用放在剪贴板上的这些文本。这种使用制表符\t 和换行符\n 作为文件分隔符的形式可以被包括微软 Excel 在内的许多应用程序所支持。

函数 QTableWidget::selectedRanges() 返回一个选择范围列表。我们知道，由于在构造函数中已经将选择模式设置为 QAbstractItemView::ContiguousSelec-

tion，所以选择范围不可能再超过一。为方便起见，我们定义了一个 selectedRange() 函数来返回这个选择范围。

```
31 QTableWidgetSelectionRange Spreadsheet::selectedRange() const
32 {
33     QList<QTableWidgetSelectionRange> ranges = selectedRanges();
34     if (ranges.isEmpty())
35         return QTableWidgetSelectionRange();
36     return ranges.first();
37 }
```

如果只有一个选择，则只需简单地返回第一个（并且也只有这一个）选择即可。没有选择的情况应该永远不会发生，因为 ContiguousSelection 模式至少可以把当前单元格当作是已经选中的选择。但是，为了避免使程序出现缺陷的可能性，还是需要对这种当前没有选中单元格的情况进行单独处理。

```
169 void Spreadsheet::paste()
170 {
171     QTableWidgetSelectionRange range = selectedRange();
172     QString str = QApplication::clipboard()->text();
173     QStringList rows = str.split('\n');
174     int numRows = rows.count();
175     int numColumns = rows.first().count('\t') + 1;
176 
177     if (range.rowCount() * range.columnCount() != 1
178             && (range.rowCount() != numRows
179                 || range.columnCount() != numColumns)) {
180         QMessageBox::information(this, tr("Spreadsheet"),
181                 tr("The information cannot be pasted because the copy "
182                    "and paste areas aren't the same size."));
183         return;
184     }
```

```
185
186     for (int i = 0; i < numRows; ++i) {
187         QStringList columns = rows[i].split('\t');
188         for (int j = 0; j < numColumns; ++j) {
189             int row = range.topRow() + i;
190             int column = range.leftColumn() + j;
191             if (row < RowCount && column < ColumnCount)
192                 setFormula(row, column, columns[j]);
193         }
194     }
195     somethingChanged();
196 }
```

paste() 槽对 Edit→Paste 菜单选项做出响应。我们从剪贴板中取回文本，并且调用静态函数 QString::split() 把这串字符变成一个 QStringList。每行都会变成这个列表中的一个字符串。

接下来，需要求出复制区域的维数。行数就是 QStringList 中字符串的个数，列数就是第一行中制表符\t 字符的个数再加上 1。如果只选中了一个单元格，就把这个单元格作为粘贴区域放在左上角；否则，就把当前选择作为要粘贴的区域。

为了执行粘贴操作，我们遍历所有行并且再次使用 QString::split() 把它们分隔到每一个单元格中，但是这一次要把制表符\t 当作分隔符。图 4.6 给出了这一过程中所使用的步骤。

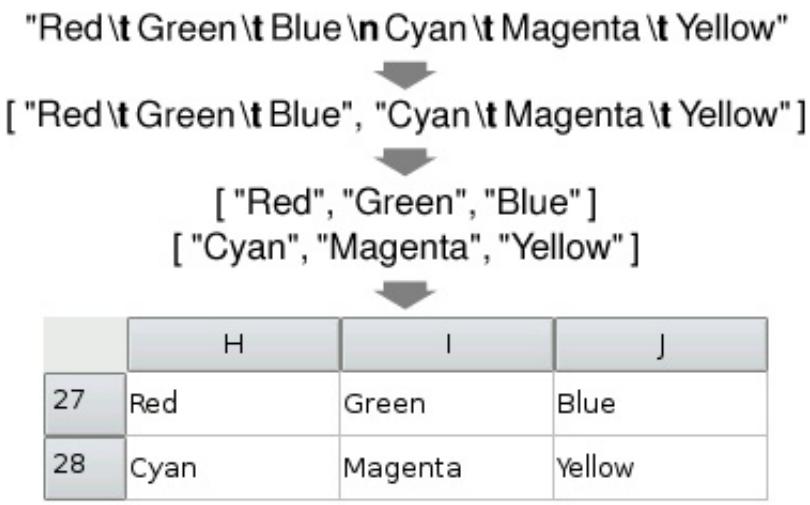


图 4.6: 把剪贴板中的文本粘贴到电子制表软件中

```
198 void Spreadsheet::del()
199 {
200     QList<QTableWidgetItem *> items = selectedItems();
201     if (!items.isEmpty()) {
202         foreach (QTableWidgetItem *item, items)
203             delete item;
204         somethingChanged();
205     }
206 }
```

del() 槽对 Edit→Delete 菜单选项做出响应。如果有选中的项，那么该函数就会删除它们并且调用 somethingChanged() 函数。对选择中的每一个 Cell 对象使用 delete 足以清空所有这些单元格。当删除 QTableWidget 的 QTableWidgetItem 的时候，QTableWidget 就会注意到这一情况的发生，而如果这些项中有可见的任意项，QTableWidget 将会自动对自己进行重绘。如果在一个已经删除过的单元格位置上又调用了 cell()，那么该函数将会返回一个空指针。

```
208 void Spreadsheet::selectCurrentRow()
209 {
```

```
210     selectRow(currentRow());
211 }
212
213 void Spreadsheet::selectCurrentColumn()
214 {
215     selectColumn(currentColumn());
216 }
```

selectCurrentRow() 和 selectCurrentColumn() 对 Edit→Select→Row 和 Edit→Select→Column 菜单选项做出响应。这些实现分别依赖于 QTableWidget 的 selectRow() 和 selectColumn() 函数。我们不必再去实现 Edit→Select→All 菜单选项的功能，因为该功能可以由 QTableWidget 从 QAbstractItemView::selectAll () 的函数中继承过来。

```
236 void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
237 {
238     int row = currentRow();
239     int column = currentColumn() + 1;
240
241     while (row < RowCount) {
242         while (column < ColumnCount) {
243             if (text(row, column).contains(str, cs)) {
244                 clearSelection();
245                 setCurrentCell(row, column);
246                 activateWindow();
247                 return;
248             }
249             ++column;
250         }
251         column = 0;
252         ++row;
253     }
```

```
254     QApplication::beep();
255 }
```

findNext() 槽会遍历单元格一遍，它从当前光标右侧的单元格开始遍历到这一行的最后一列，然后再从下一行的第一个单元格开始继续遍历，如此反复，直到找到所要查找的文本，或者是直到最后一个单元格为止。例如，如果当前的单元格是 C24，那么就会搜索 D24、E24、···、Z24，然后再去搜索 A25、B25、C25、···、Z25，等等，一直遍历到 Z999 为止。如果找到了一个匹配项，那么就清空当前选择，把单元格光标移动到那个匹配的单元格上，并且让包含 Spreadsheet 的窗口变成激活状态。如果没能找到匹配的单元格，那么就让应用程序发出“哔”(beep) 的一声来表明搜索已经结束，匹配没有成功。

```
257 void Spreadsheet::findPrevious(const QString &str,
258                                 Qt::CaseSensitivity cs)
259 {
260     int row = currentRow();
261     int column = currentColumn() - 1;
262 
263     while (row >= 0) {
264         while (column >= 0) {
265             if (text(row, column).contains(str, cs)) {
266                 clearSelection();
267                 setCurrentCell(row, column);
268                 activateWindow();
269                 return;
270             }
271             --column;
272         }
273         column = ColumnCount - 1;
274         --row;
275     }
276     QApplication::beep();
```

```
277 }
```

findPrevious() 槽与 findNext() 槽相似，区别之处是它会向相反的方向遍历并且会在单元格 A1 处停下来。

4.5 实现其他菜单

现在，我们将要实现那些对 Tools 和 Options 菜单做出响应的槽。这些菜单项如图 4.7 所示。

图 4.7: Spreadsheet 应用程序的 Tools 和 Options 菜单

```
218 void Spreadsheet::recalculate()
219 {
220     for (int row = 0; row < RowCount; ++row) {
221         for (int column = 0; column < ColumnCount; ++column) {
222             if (cell(row, column))
223                 cell(row, column)->setDirty();
224         }
225     }
226     viewport()->update();
227 }
```

recalculate() 槽能够对 Tools→Recalculate 菜单选项做出响应。当必要时，它也会被 Spreadsheet 自动调用。

我们遍历每一个单元格，并且对每一个单元格调用 setDirty() 把它们标记为需要重新计算。为了在电子制表软件中显示一个 Cell 对象的值，QTableWidget 会再次对该对象调用 text() 以获得其值，从而使该值重新计算一次。

然后，对这个视口调用 update() 来重新绘制整个电子制表软件。QTableWidget 中的重绘代码就又会对每一个可见单元格调用 text() 来获得它们中要显示的值。因为在每一个单元格上都调用了 setDirty()，所以这些对 text() 的调用将会使用重新计算过的值。该计算可能需要重新计算那些不可见的单元格，这就会造成一个级联计算，直到每一个需要被重新计算的单元格能够在刚才刷新过的视口中重新得到计算，从而使它们也能够显示正确的文本。这一计算是由 Cell 类执行的。

```
229 void Spreadsheet::setAutoRecalculate(bool recalc)
230 {
231     autoRecalc = recalc;
232     if (autoRecalc)
233         recalculate();
234 }
```

setAutoRecalculate() 槽对 Options→Auto-Recalculate 菜单选项做出响应。如果启用了这个特性，则会立即重新计算整个电子制表软件以确保它是最新的，然后，recalculate() 会自动在 somethingChanged() 中得到调用。

因为 QTableWidget 已经提供了一个从 QTableView 中继承而来的 setShowCrid() 槽，所以不需要再对 Options→Show Grid 菜单选项编写任何代码。所有要保留的东西就是 Spreadsheet::sort()，它会在 MainWindow::sort() 中得到调用：

```
120 void Spreadsheet::sort(const SpreadsheetCompare &compare)
121 {
122     QList<QStringList> rows;
123     QTableWidgetSelectionRange range = selectedRange();
124     int i;
125 
126     for (i = 0; i < range.rowCount(); ++i) {
127         QStringList row;
128         for (int j = 0; j < range.columnCount(); ++j)
129             row.append(formula(range.topRow() + i,
```

```
130                                 range.leftColumn() + j));
131         rows.append(row);
132     }
133
134     qStableSort(rows.begin(), rows.end(), compare);
135
136     for (i = 0; i < range.rowCount(); ++i) {
137         for (int j = 0; j < range.columnCount(); ++j)
138             setFormula(range.topRow() + i, range.leftColumn() + j,
139                        rows[i][j]);
140     }
141
142     clearSelection();
143     somethingChanged();
144 }
```

排序操作会对当前的选择进行，并且会根据存储在 compare 对象中的排序键和排序顺序重新排列这些行。我们使用一个 QStringList 来重新表示每一行数据，并且把该选择存储在一个行列表中。我们使用 Qt 的 qStableSort() 算法，并且根据公式而不是根据值来进行简单排序。这一过程如图 4.8 和图 4.9 所示。第 11 章中会讲述 Qt 的标准算法和数据结构。

	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger", "Dijkstra", "1930-05-11"]
1	["Tony", "Hoare", "1934-01-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Donald", "Knuth", "1938-01-10"]

图 4.8: 把选择存储为一个行列表

index	value
0	["Donald", "Knuth", "1938-01-10"]
1	["Edsger", "Dijkstra", "1930-05-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Tony", "Hoare", "1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

图 4.9: 排序后把数据放回表中

qStableSort() 函数可以接受一个开始迭代器、一个终止迭代器和一个比较函数。这个比较函数是一个带两个参数（两个 QStringList）的函数，并且如果第一个参数“小于”第二个参数，它就返回 true，否则返回 false。传递的作为比较函数的这个 compare 对象并不是一个真正的函数，但是它可以用作一个函数，将会很快看到这一点。

在执行完 qStableSort() 之后，我们把数据移回到这个表中，接着清空这一选择，并且调用。somethingChanged() 函数。

在 spreadsheet.h 文件中，Spreadsheet 类的定义如下：

```
54 class SpreadsheetCompare
55 {
56 public:
57     bool operator()(const QStringList &row1,
58                     const QStringList &row2) const;
59
60     enum { KeyCount = 3 };
61     int keys[KeyCount];
62     bool ascending[KeyCount];
63 };
```

SpreadsheetCompare 类有些特殊，因为它实现了一个“()”操作符。这样就允许把这个类像函数一样使用。把这样的类称为函数对象 (function object)，或者称为仿函数 (functor)。为了理解仿函数是如何工作的，首先从一个简单的例子开始：

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
}
```

Square 类提供了一个函数，operator()(int) 函数，它返回其参数的平方值。通过把这个函数命名为 operator()(int)，而不是将其命名为 compute(int) 之类的函数，就可以把一个类型为 Square 的对象当作一个函数。

```
Square square;
int y = square(5);
// y equals 25
```

现在，让我们来看一个包括 SpreadsheetCompare 的实例：

```
QStringList row1, row2;
SpreadsheetCompare compare;
...
if (compare(row1, row2)) {
    // row1 is less than row2
}
```

使用 compare 对象就像使用一个普通的 compare() 函数一样。另外，它的实现可以访问所有存储为成员变量的排序键和排序顺序。

与此方案相同的另一种方法是，把这些排序键和排序顺序存储在全局成员变量中，并且使用一个普通的 compare() 函数。然而，在全局成员变量之间通信是一种并不提倡的做法，并且可能会产生一些莫名其妙的问题。作为像 qStableSort() 这样的模板函数的接口，仿函数是一种更为常用的做法。

这里给出的是对电子制表软件中的两个行进行比较的函数实现：

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
                } else {
```

```
                    return row1[column] > row2[column];
                }
            }
        }
    }
    return false;
}
```

如果第 1 行小于第 2 行，该仿函数就返回 true；否则，就返回 false。qStableSort() 函数会使用这个函数的结果来执行排序操作。

SpreadsheetCompare 对象的 key 与 ascending 数组和 MainWindow::sort() 函数（已经在第 2 章中给出过）一起配合使用。每个键都保存一个列索引，或者 -1（为”None” 时）。

我们按键顺序比较两行中相应的单元格条目。一旦发现有不同之处，就返回一个适当的 true 或者 false 值。如果所有的比较关系都证明两者是相等的，就返回 false。qStableSort() 函数会使用这里给出的顺序来解决这种平局情形。如果一开始的时候 row1 在 row2 之前，并且它们都不“小于”对方，那么，在结果中 row1 还在 row2 前面。这就是 qStableSort() 与它很相似的非稳定版本的 qSort() 函数之间的区别。

现在已经完成了这个 Spreadsheet 类。在下一节中，将分析 Cell 类的代码。这个类用作保存单元格的公式，并且它还重新实现了 QTableWidgetItem::data() 函数，Spreadsheet 可以通过 QTableWidgetItem::text() 间接调用该函数，用它显示单元格公式的计算结果。

4.6 子类化 QTableWidgetItem

Cell 类派生自 QTableWidgetItem 类。这个类被设计用于和 Spreadsheet 一起工作，但是它对类 QTableWidgetItem 没有任何特殊的依赖关系，所以在理论上讲，它也可以用于任意的 QTableWidget 类中。这里给出的是 Cell 类的头文件：

```
#ifndef CELL_H
#define CELL_H

#include <QTableWidgetItem>

class Cell : public QTableWidgetItem
{
public:
    Cell();

    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();

private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;

    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};

#endif
```

通过增加两个私有变量，Cell 类对 QTableWidgetItem 进行了扩展：

- cachedValue 把单元格的值缓存为 QVariant。
- 如果缓存的值不是最新的，那么就把 cacheIsDirty 设置为 true。

之所以使用 QVariant，是因为有些单元格是 double 型值，另外一些单元格则是 QString 型值。

在声明 cachedValue 和 cacheIsDirty 变量时使用了 C++ 的 mutable 关键字，这样就可以在 const 函数中修改这些变量。或者，在每次调用 text() 时，我们可以重新计算这个值，不过这样做是不必要而且效率低下的。

我们注意到，在该类的定义中并没有使用 Q_OBJECT 宏。这是因为，Cell 是一个普通的 C++ 类，它没有任何信号或者槽。实际上，因为 QTableWidgetItem 不是从 QObject 派生而来的，所以就不能让 Cell 拥有信号和槽。为了使 Qt 的项 (item) 类的开销降到最低，它们就不是从 QObject 派生的。如果需要信号和槽，可以在包含项的窗口部件中实现它们，或者在特殊情况下，可以通过对 QObject 进行多重继承的方式来实现它们。

以下是 cell.cpp 文件的开始部分：

```
#include <QtGui>

#include "cell.h"

Cell::Cell()
{
    setDirty();
}
```

在构造函数中，只需要将缓存设置为 dirty。没有必要传递父对象，当用 setItem() 把单元格插入到一个 QTableWidget 中的时候，QTableWidget 将会自动对其拥有所有权。

每个 QTableWidgetItem 都可以保存一些数据，最多可以为每个数据“角色”分配一个 QVariant 变量。最常用的角色是 Qt::EditRole 和 Qt::DisplayRole。编辑角色用在那些需要编辑的数据上，而显示角色用在那些需要显示的数据上。通常情况下，用于两者的数据是一样的，但在 Cell 类中，编辑角色对应于单元格的公式，而显示角色对应于单元格的值（对公式求值后的结果）。

```
10 QTableWidgetItem *Cell::clone() const
11 {
12     return new Cell(*this);
13 }
```

当 QTableWidget 需要创建一个新的单元格时，例如，当用户在一个以前没有使用过的空白单元格中开始输入数据时，它就会调用 clone() 函数。传递给 QTableWidget::setItemPrototype() 中的实例就是需要克隆的项。由于对于 Cell 来讲，成员级的复制已经足以满足需要，所以在 clone() 函数中，只需依靠由 C++ 自动创建的默认复制构造函数就可以创建新的 Cell 实例了。

```
41 void Cell::setFormula(const QString &formula)
42 {
43     setData(Qt::EditRole, formula);
44 }
```

setFormula() 函数用来设置单元格中的公式。它只是一个对编辑角色调用 setData() 的简便函数。也可以从 Spreadsheet::setFormula() 中调用它。

```
46 QString Cell::formula() const
47 {
48     return data(Qt::EditRole).toString();
49 }
```

formula() 函数会从 Spreadsheet::formula () 中得到调用。就像 setFormula() 一样，它也是一个简便函数，这次是重新获得该项的 EditRole 数据。

```
15 void Cell::setData(int role, const QVariant &value)
16 {
17     QTableWidgetItem::setData(role, value);
```

```
18      if (role == Qt::EditRole)
19          setDirty();
20  }
```

如果有一个新的公式，就可以把 cacheIsDirty 设置为 true，以确保在下一次调用 text() 的时候可以重新计算该单元格。

尽管对 Cell 实例中的 Spreadsheet::text() 调用了 text()，但在 Cell 中没有定义 text() 函数。这个 text() 函数是一个由 QTableWidgetItem 提供的简便函数。这相当于调用 data(Qt::DisplayRole).toString()。

```
51  void Cell::setDirty()
52  {
53      cacheIsDirty = true;
54  }
```

调用 setDirty() 函数可以用来对该单元格的值强制进行重新计算。它只是简单地把 cacheIsDirty 设置为 true，也就意味着 cachedValue 不再是最新值了。除非有必要，否则不会执行这个重新计算操作。

```
22  QVariant Cell::data(int role) const
23  {
24      if (role == Qt::DisplayRole) {
25          if (value().isValid()) {
26              return value().toString();
27          } else {
28              return "####";
29          }
30      } else if (role == Qt::TextAlignmentRole) {
31          if (value().type() == QVariant::String) {
32              return int(Qt::AlignLeft | Qt::AlignVCenter);
33          } else {
```

```
34              return int(Qt::AlignRight | Qt::AlignVCenter);
35          }
36      } else {
37          return QTableWidgetItem::data(role);
38      }
39  }
```

data() 函数是从 QTableWidgetItem 中重新实现的。如果使用 Qt::DisplayRole 调用这个函数，那么它返回在电子制表软件中应该显示的文本；如果使用 Qt::EditRole 调用这个函数，那么它返回该单元格中的公式；如果使用 Qt::TextAlignmentRole 调用这个函数，那么它返回一个合适的对齐方式。在使用 DisplayRole 的情况下，它依靠 value() 来计算单元格的值。如果该值是无效的（由于这个公式是错误的），则返回”####”。

在 data() 中使用的这个 Cell::value() 函数可以返回一个 QVariant 值。QVariant 可以存储不同类型的值，比如 double 和 QString，并且提供了把变量转换为其他类型变量的一些函数。例如，对一个保存了 double 值的变量调用 toString()，可以产生一个表示这个 double 值的字符串。使用默认构造函数构造的 QVariant 是一个“无效”变量。

```
56  const QVariant Invalid;
57
58  QVariant Cell::value() const
59  {
60      if (cacheIsDirty) {
61          cacheIsDirty = false;
62
63          QString formulaStr = formula();
64          if (formulaStr.startsWith('\'')) {
65              cachedValue = formulaStr.mid(1);
66          } else if (formulaStr.startsWith('=')) {
67              cachedValue = Invalid;
```

```
68              QString expr = formulaStr.mid(1);
69              expr.replace(" ", "");
70              expr.append(QChar::Null);
71
72              int pos = 0;
73              cachedValue = evalExpression(expr, pos);
74              if (expr[pos] != QChar::Null)
75                  cachedValue = Invalid;
76          } else {
77              bool ok;
78              double d = formulaStr.toDouble(&ok);
79              if (ok) {
80                  cachedValue = d;
81              } else {
82                  cachedValue = formulaStr;
83              }
84          }
85      }
86      return cachedValue;
87  }
```

value() 私有函数返回这个单元格的值。如果 cacheIsDirty 是 true，就需要重新计算这个值。

如果公式是由单引号开始的（例如，"'12345"），那么这个单引号就会占用位置 0，而值就是从位置 1 直到最后位置的一个字符串。

如果公式是由等号开始的，那么会使用从位置 1 开始的字符串，并且将它可能包含的任意空格全部移除。然后，调用 evalExpression() 来计算这个表达式的值。这里的参数 pos 是通过引用 (reference) 方式传递的，由它来说明需要从哪里开始解析字符的位置。在调用 evalExpression() 之后，如果表达式解析成功，那么在位置 pos 处的字符应当是我们添加上的 QChar::Null 字符。如果在表达式结束之前解析失败了，那么可以把 cachedValue 设置为 Invalid。

如果公式不是由单引号或者等号开始的，那么可以使用 toDouble() 试着把

它转换为浮点数。如果转换正常，就把 cachedValue 设置为结果数字；否则，把 cachedValue 设置为字符串公式。例如，公式”1.50” 会导致 toDouble() 把 ok 设置为 true 并且返回 1.5，而公式”World Population” 则会导致 toDouble() 把 ok 设置为 false 并且返回 0.0。

通过给 toDouble() 一个 bool 指针，可以区分字符串转换中表示的是数字 0.0 还是表示的是转换错误（此时，仍旧会返回一个 0.0，但是同时会把这个 bool 设置为 false）。有时候，对于转换失败所返回的 0 值可能正是我们所需要的。在这种情况下，就没有必要再麻烦地传递一个 bool 指针了。考虑到程序的性能和移植性因素，Qt 从来不使用 C++ 异常 (exception) 机制来报告错误。但是，如果你的编译器支持 C++ 异常，那么这也不会妨碍你在自己的 Qt 程序中使用它们。

value() 函数声明为 const 函数。我们不得不把 cachedValue 和 cacheIsValid 声明为 mutable 变量，以便编译器可以让我们在 const 函数中修改它们。当然，如果能够把 value() 声明为一个非 const 函数并且移除 mutable 关键字可能会更吸引人些，但是这将会导致无法编译，因为是从一个 const 函数的 data() 函数中调用 value() 的。

除了要解析这些公式之外，现在已经完成了整个 Spreadsheet 应用程序。这一节的剩余部分将说明 evalExpression() 以及 evalTerm() 和 evalFactor() 这两个帮助函数。这些代码有一些复杂，但是把它们放在这里是为了能够让整个应用程序显得更完善些。由于这些代码和图形用户界面编程无关，所以你可以非常放心地略过这一部分，从第 5 章继续阅读下去。

evalExpression() 函数返回一个电子制表软件表达式的值。表达式可以定义为：一个或者多个通过许多”+” 或者”-” 操作符分隔而成的项。这些项自身可以定义为：由”*” 或者”/” 操作符分隔而成的一个或者多个因子 (factor)。通过把表达式分解成项，再把项分解成因子，就可以确保以正确的顺序来使用这些操作符了。

例如，”2*C5+D6” 就是一个表达式，它由作为第一项的”2*G5” 和作为第二项的”D6” 构成。项”2*C5” 是由作为第一个因子的”2” 和作为第二个因子的”C5” 组成的，而项”D6” 则由一个单一的因子”D6” 组成。一个因子可以是一个数 (”2”)、一个单元格位置 (”C5”)，或者是一个在圆括号内的表达式，在它们的前面可以有负号。

在图 4.10 中，定义了电子制表软件表达式的语法。对于语法（表达式、项和因子）中的每一个符号，都对应一个解析它的成员函数，并且函数的结构严格遵循语法。通过这种方式写出的解析器称为递归渐降解析器 (recursive-descent parser)。

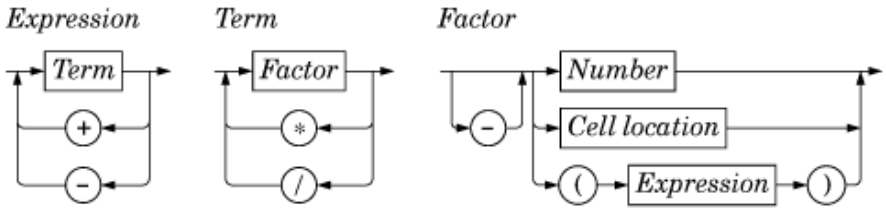


图 4.10: 用于电子制表软件表达式的语法图

让我们先从 evalExpression() 开始，这个函数可以解析一个表达式：

```
89  QVariant Cell::evalExpression(const QString &str, int &pos) const
90  {
91      QVariant result = evalTerm(str, pos);
92      while (str[pos] != QChar::Null) {
93          QChar op = str[pos];
94          if (op != '+' && op != '-')
95              return result;
96          ++pos;
97
98          QVariant term = evalTerm(str, pos);
99          if (result.type() == QVariant::Double
100                 && term.type() == QVariant::Double) {
101             if (op == '+') {
102                 result = result.toDouble() + term.toDouble();
103             } else {
104                 result = result.toDouble() - term.toDouble();
105             }
106         } else {
107             result = Invalid;
108         }
109     }
110     return result;
111 }
```

首先，调用 evalTerm() 得到第一项的值。如果它后面紧跟的字符是”+” 或者”-

”，那么就继续第二次调用 evalTerm()；否则，表达式就只包含一个单一项，并且把它的值作为整个表达式的值而返回。在得到前两项的值之后，根据操作符计算出这一操作的结果。如果两项都求出一个 double 值，就把计算出的结果当作一个 double 值；否则，把结果设置为 Invalid。

像前面那样继续操作，直到再没有更多的项为止。这样做可以正确地进行，因为加法和减法都是左相关 (left-associative) 的；也就是说，”1-2-3” 的意思是”(1-2)-3”，而不是”1-(2-3)”。

```
113 QVariant Cell::evalTerm(const QString &str, int &pos) const
114 {
115     QVariant result = evalFactor(str, pos);
116     while (str[pos] != QChar::Null) {
117         QChar op = str[pos];
118         if (op != '*' && op != '/')
119             return result;
120         ++pos;
121 
122         QVariant factor = evalFactor(str, pos);
123         if (result.type() == QVariant::Double
124                 && factor.type() == QVariant::Double) {
125             if (op == '*') {
126                 result = result.toDouble() * factor.toDouble();
127             } else {
128                 if (factor.toDouble() == 0.0) {
129                     result = Invalid;
130                 } else {
131                     result = result.toDouble() / factor.toDouble();
132                 }
133             }
134         } else {
135             result = Invalid;
136         }
```

```
137     }
138     return result;
139 }
```

除了 evalTerm() 函数是处理乘法和除法这一点不同之外，它和 evalExpression() 都很相似。在 evalTerm() 中唯一的不同就是必须要避免除零，因为在一些处理器中这将是一个错误。尽管测试浮点数值是否相等通常并不明智，因为其中存在取舍问题，但是在这个防止除零的问题上，这样做相等性测试已经足够了。

```
141 QVariant Cell::evalFactor(const QString &str, int &pos) const
142 {
143     QVariant result;
144     bool negative = false;
145 
146     if (str[pos] == '-') {
147         negative = true;
148         ++pos;
149     }
150 
151     if (str[pos] == '(') {
152         ++pos;
153         result = evalExpression(str, pos);
154         if (str[pos] != ')')
155             result = Invalid;
156         ++pos;
157     } else {
158         QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
159         QString token;
160 
161         while (str[pos].isLetterOrNumber() || str[pos] == '.') {
162             token += str[pos];
```

```
163             ++pos;
164         }
165 
166         if (regExp.exactMatch(token)) {
167             int column = token[0].toUpper().unicode() - 'A';
168             int row = token.mid(1).toInt() - 1;
169 
170             Cell *c = static_cast<Cell *>(
171                               tableWidget()->item(row, column));
172             if (c) {
173                 result = c->value();
174             } else {
175                 result = 0.0;
176             }
177         } else {
178             bool ok;
179             result = token.toDouble(&ok);
180             if (!ok)
181                 result = Invalid;
182         }
183     }
184 
185     if (negative) {
186         if (result.type() == QVariant::Double) {
187             result = -result.toDouble();
188         } else {
189             result = Invalid;
190         }
191     }
192     return result;
193 }
```

evalFactor() 函数比 evalExpression() 和 evalTerm() 函数都要复杂一些。

它先从计算因子是否为负开始。然后，判断它是否是从左圆括号开始的。如果是，就先把圆括号内的内容作为表达式并通过调用 evalExpression() 来处理它。当解析到带圆括号的表达式时，evalExpression() 调用 evalTerm()，evalTerm() 调用 evalFactor()，evalFactor() 则会再次调用 evalExpression()。这就是在解析器中出现递归调用的地方。

如果该因子不是一个嵌套表达式，就提取下一个记号，它应当是一个单元格的位置，或者也可能是一个数字。如果这个记号匹配 QRegExp，就把它认为是一个单元格引用并且对给定位置处的单元格调用 value()。该单元格可能在电子制表软件中的任何一个地方，并且它可能会依赖于其他的单元格。这种依赖不是什么问题，它们只会简单地触发更多的 value() 调用和（对于那些”dirty” 单元格）更多的解析处理，直到所有相关的单元格的值都得到计算为止。如果记号不是一个单元格的位置，那么就把它看作是一个数字。

如果单元格 A1 包含公式”=A1” 时会发生什么呢？或者如果单元格 A1 包含公式”=A2” 并且单元格 A2 包含公式”=A1” 时又会发生什么呢？尽管还没有编写任何特定代码来检测这种循环依赖关系，但解析器可以通过返回一个无效的 QVariant 来完美地处理这一情况。之所以可以正常工作，是因为在调用 evalExpression() 之前，我们会在 value() 中把 cacheIsDirty 设置为 false，把 cachedValue 设置为 Invalid。如果 evalExpression() 对同一个单元格循环调用 value()，它就会立即返回 Invalid，并且这样就会使整个表达式等于 Invalid。

现在，我们已经完成了公式解析器。通过扩展因子的语法定义，公式解析器可以非常方便地处理那些在电子制表软件中像”sum()” 和”avg()” 一样的某些预定义函数。而另外一种比较容易的扩展方式是将”+” 操作符实现为字符串连接（就像串联一样），这样就无需再对这个语法进行修改了。

II 附录

A Qt 的获取和安装

本部分工作稍后将会补上，可能还会加入一些最新的资源信息。

B 编译 Qt 应用程序

本部分工作放在稍后面些。

C Qt Jambi 简介

本部分工作将放在最后面最后面。

D 面向 Java 和 C# 程序员的 C++ 简介

这个附录为已经熟知 Java 或者 C# 的开发人员提供一个关于 C++ 的简短介绍。这里假定你已经熟悉了面向对象中的那些概念，如继承和多态，并且认为你也的确是想学习 C++。为了不让本书变成一部厚达 1500 页的涵盖全部 C++ 入门知识的不实用的“大部头”，所以要把这个附录仅仅限定在基本知识的范围内：只给出用来理解本书其他部分所示例子的基本知识和方法，但这些知识也足以使用 Qt 开发跨平台的 C++ 图形用户界面应用程序。

在编写这本书的时候，C++ 是开发跨平台、高性能、面向对象的图形用户界面应用程序的唯一现实选择。而一些别有用心的批评者也可能会指出，Java 或者 C# 具有更好的可用性，而 C++ 则降低了 C 的兼容性。实际上，作为 C++ 发明人的 Bjarne Stroustrup，他在”The Design and Evolution of C++”(Addison-Wesley，1994) 一书中早就指出：“即使有 C++，还可以找出更小、更简洁的语言”。

幸运的是，在使用 Qt 进行编程的时候，我们通常只关注于 C++ 的子类，这非常接近于 Stroustrup 所设想的“乌托邦”式的编程语言，从而能够让我们集中精力去解决手头的问题。此外，通过 Qt 独创性的“信号和槽”机制、对统一字符编码标准的支持以及 foreach 关键字，Qt 也在多个方面扩展了 C++。

在这个附录的第一节中，将会看到如何使用 C++ 的源文件产生一个可执行程序。这将可以引导我们探索 C++ 的一些核心概念，如编译单元、头文件、目标文件、库等，并且也可以让我们逐步熟悉 C++ 的预处理器、编译器和连接器。

然后，会转到对 C++、Java 和 C# 这些主要语言不同点的说明上：如何定义类，如何使用指针和引用，如何重载运算符，如何使用预处理器，等等。尽管 C++ 语法从表面上看与 Java 或者 C# 的语法很相似，但从深层意义上来讲，这些概念却稍微显得不尽相同。同时，作为 Java 和 C# 的创意之源，C++ 语言也与这两种语言有着诸多相同之处，包括相似的数据类型，同样的数学运算符，以及同样的基本控制流语句等。

最后一节专门用于说明标准 C++ 库，该库提供了可用于任意 C++ 程序中的完善功能。这个库是 30 多年来演化的结果，并且因此提供了涵盖程序、面向对象、函数编程风格以及宏和模板等方面的诸多方法。与 Java 和 C# 提供的库相比，标准 C++ 库的范围显得有些窄。比如，标准 C++ 库不支持图形用户界面程序设计、多线程、数据库、国际化、网络、XML 或者统一字符编码标准。要在这些领域进行开发，C++ 程序员则可能要使用各种各样（通常是与平台相关的）的库。

这就是为什么说 Qt 可以节约时间的原因。Qt 首先作为跨平台的图形用户界面工具包（一个让编写可移植图形用户界面应用程序成为可能的类的集合）而起步，但很快发展成为一个成熟的程序开发框架，它对标准 C++ 库进行了部分扩展和部分替换。尽管本书使用的是 Qt，但是如果能够知道标准 C++ 库到底提供了哪些功能也是很有用的，因为你有可能需要去处理一些使用了那些功能的代码。

D.1 C++ 入门

一个 C++ 程序由一个或者多个编译单元 (compilation unit) 构成。每个编译单元都是一个独立的源代码文件，通常是一个带.cpp 扩展名（其他常用的扩展名还有.cc 和.cxx）的文件，编译器每次可以处理一个这样的文件。对于每一个编译单元，编译器都会产生一个目标文件，它的扩展名是.obj（在 Windows 中）或者.o（在 UNIX 和 Mac OS X 中）。这个目标文件是一个二进制文件，其中包含了系统架构方面的机器代码，而程序则要运行在此基础之上。

一旦所有的.cpp 文件都已编译完成，那么我们就可以使用一个称为连接器的特殊程序，把这些目标文件连接在一起，生成一个可执行程序。连接器会连接这些目标文件，并且会解析函数和编译单元中引用到的其他符号的内存地址。

在构建一个程序时，必须确保其中的某个编译单元包含一个 main() 函数，它是程序入口的标志。这个函数不属于任何类，它是一个全局函数 (global function)。图 D.1 给出了这一过程的原理图。

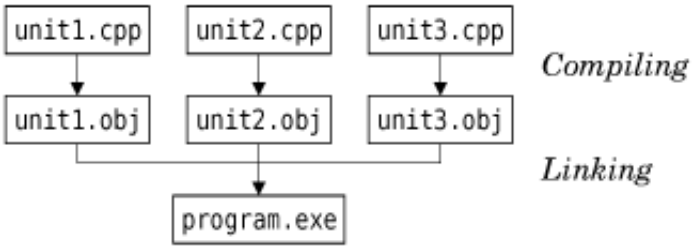


图 D.1: C++ 的编译过程（在 Windows 中）

不像 Java 的每一个源文件都必须严格包含一个类那样，C++ 可以按我们想要的形式组织各个编译单元。我们可以在同一个.cpp 文件中实现多个类，或者也可以把一个类的实现分散到多个.cpp 文件中，并且还可以把这些源文件命名为我们所喜欢的任意名字。当在某一个特殊的.cpp 文件中进行修改时，只需要重新编译那个文件，然后再重新连接这个应用程序就可以生成一个新的可执行程序。

在进一步深入学习之前，让我们快速看一个 C++ 小程序的源代码，该程序可以计算一个整数的平方。这个程序由两个编译单元构成：main.cpp 和 square.cpp。

这是 square.cpp 文件中的内容：

```
double square(double n)
{
    return n * n;
}
```

这个文件只简单包含了一个称为 square() 的全局函数，它可以返回所带参数的平方值。

这里是 main.cpp 文件中的内容：

```
#include <cstdlib>
#include <iostream>

double square(double);

int main(int argc, char *argv[]){
    if (argc != 2) {
        std::cerr << "Usage: square <number>" << std::endl;
        return 1;
    }
    double n = std::strtod(argv[1], 0);
    std::cout << "The square of " << argv[1] << " is "
                << square(n) << std::endl;
```

```
14      return 0;
15  }
```

源文件 main.cpp 包含了 main() 函数的定义。在 C++ 中，这个函数的参数是一个 int 和一个 char * 数组（一个字符串数组）。可以从 argv[0] 中获取程序的名字，命令行参数则分别放在 argv[1]、argv[2]、···、argv[argc-1] 中。把参数命名为 argc（argument count，参数个数）和 argv（argument values，参数值）是一种习惯性的做法。如果这个程序不能使用命令行参数，那么可以把 main() 定义成不带参数的形式。

这个 main() 函数使用标准 C++ 库中的 strtod()（即“string 转换到 double”）、cout（c++ 的标准输出流），和 cerr（c++ 的标准错误信息输出流），把命令行参数转换成 double，并且以文本的形式打印到终端控制台。字符串、数字和行尾标记符 (endl) 都是使用 « 操作符的输出流，该操作符也用于移位操作 (bit-shifting) 中。为了可以使用这一标准功能，我们需要在第 1 行和第 2 行中加入`#inlcude` 指示符。

标准 C++ 库中的所有函数和大多数的其他对象都在 std 命名空间中。一种访问命名空间中的某一项的方法是用命名空间的名字和:: 操作符作为该项名字的前缀。在 C++ 中，:: 操作符可以作为复杂名字的分隔符。命名空间可以使巨大的多人合作项目变得更容易些，因为命名空间可以避免命名冲突问题。在本附录的后面，还会做进一步的讨论。

位于第 3 行的代码是一个函数原型 (function prototype)。它告诉编译器：存在这样一个带有给定参数和返回值的函数。而实际的函数则可以位于同一个编译单元中，也可以放在其他编译单元中。没有这个函数原型，编译器将不会让我们在第 12 行调用函数。在函数原型中，这些参数的名字是可有可无的。

编译这个程序的过程因平台的不同而略有不同。例如，要在 Solaris 上使用 Sun C++ 编译器编译这个程序，应当输入以下命令：

```
CC -c main.cpp
CC -c square.cpp
CC main.o square.o -o square
```

最前面的两行会调用编译器生成这些.cpp 文件对应的.o 文件。第三行则调用连接

器，并且据此生成一个称为 square 的可执行程序，于是我们就可以像下面那样来运行该程序：

```
./square 64
```

这个程序运行后会在终端上输出下列信息：

```
The square of 64 is 4096
```

要编译这个程序，你可能希望从 C++ 专家那里得到帮助。但如果无法从别人那里获得帮助，那么可以继续阅读本附录中剩余的部分而不必编译任何东西，并且可以按照第 1 章中给出的用法说明来编译你的第一个 C++/Qt 应用程序。Qt 提供了一些工具，它们可以让你在所有平台上编译应用程序都变得轻松、简单。

再重新回到我们的程序中：在真实的应用程序中，我们通常会把 square() 函数的函数原型放在一个单独的文件中，然后在需要调用这个函数的所有编译单元中都包含那个文件。这样的文件就是所谓的头文件 (header file)，并且通常带一个.h 的扩展名（常见的还有.hh、.hpp、.hxx 等）。如果使用这种头文件的形式重写我们的程序，则需要创建一个名为 square.h 的文件，它所包含的内容如下所示：

```
1 #ifndef SQUARE_H
2 #define SQUARE_H
3
4 double square(double);
5
6 #endif
```

这个头文件被预处理命令（#ifndef、#define 和 #endif）分成三部分。这三个命令可以确保这个头文件只作用一次，即使这个头文件在同样的编译单元中被包含了多次都是如此（当在一个头文件中又包含了其他的头文件时，就会发生这种多次包含的情况。根据惯例，一般使用这个文件的名字作为预处理器的符号（在我们的例子中，就是 SQUARE_H），在本附录的稍后部分，还会回到预处理器这一主题上。

此时，新的 main.cpp 文件看起来像这样：

```
1  #include <cstdlib>
2  #include <iostream>
3
4  #include "square.h"
5
6  int main(int argc, char *argv[])
7  {
8      if (argc != 2) {
9          std::cerr << "Usage: square <number>" << std::endl;
10         return 1;
11     }
12     double n = std::strtod(argv[1], 0);
13     std::cout << "The square of " << argv[1] << " is "
14         << square(n) << std::endl;
15     return 0;
16 }
```

第 3 行中的#include 命令扩展了 square.h 文件的内容。C++ 预处理器会在编译开始之前获取所有这些以"#" 开始的指示符。以前，预处理器是一个单独的程序，需要在运行编译器之前由程序员手动调用它。而现在的编译器则会隐式地调用预处理器。

第 1 行和第 2 行中的 #include 命令扩展了 cstdlib 和 iostream 头文件的内容，它们都是标准 C++ 库的一部分。标准的头文件没有.h 后缀。包围文件名的尖括号说明这些头文件都位于系统的标准位置，而双引号则告诉编译器要到当前目录中查找头文件。这些包含命令通常都会放在.cpp 文件内容的最前面。

不像.cpp 文件，这些头文件自身都不是编译单元，并且也不会产生任何目标文件。头文件或许只包含一些让不同的编译单元能够互相联系的声明而已。因此，把 square() 函数的实现代码放在一个头文件中就显得有些不合适了。如果在我们的例子中那样做了，也不会产生任何不良影响，因为只包含了 square.h 一次，但是如果在多个.cpp 文件中都包含了 square.h 文件，那么就会得到 square() 函数的多重实现（每个.cpp 文件都包含头文件一次）。于是，连接器就会抱怨 square() 出现了多重（同样的）定义，并且会拒绝生成可执行程序。相反，如果我们声明了一个函数但是

却再没有实现它，那么连接器也会报错，输出”unresolved symbol”（不可解析的符号）的错误信息。

到目前为止，我们可能会认为：一个可执行程序只是由一些目标文件构成的。但在实际情况中，可执行程序通常都会连接许多库，而这些库则可以实现许多现成的功能。库主要有两种类型：

- 静态库 (static library) 可以直接放进可执行程序，就好像它们也是一些目标文件一样。这可以确保不会弄丢这些库，但却会让可执行程序变得很大。
- 动态库（dynamic library，也称共享库或 DLL）位于用户机器上的标准位置，并且会在应用程序启动的时候自动加载它们。

对于以上的 square 程序，连接的是标准的 C++ 库，通常在大多数平台上都是采用这种动态库的形式实现的。Qt 自身就是一个库的集合，既可编译为静态库又可编译为动态库（默认是动态库）

D.2 主要语言之间的差异

现在：我们将会采用一种更有条理的方式来看看 C++ 与 Java 和 C# 之间的不同之处。语言之间的这些许多不同之处都是因为 C++ 的可编译本性和对性能的追求而产生的。因此，C++ 不会在运行时检测数组是否越界，并且也没有垃圾信息收集器回收那些分配出去但是却不再使用的动态内存。

为简便起见，对于 C++ 与 Java 和 C# 中结构几乎一致的地方就不再提及了。另外，还有一些 C++ 的主题在这里也不再涉及，因为在使用 Qt 编程时并不需要它们。这其中就包括了模板类和模板函数的定义、共用体类型的定义，以及异常的使用等主题。如果要对所有这些主题都想了解，可以参考像 Bjarne Stroustrup 编著的”The C++ Programming Language”(Addison-Wesley, 2000) 和 Mark Allen Weiss 编著的”C++ for Java Programmers”(Prentice Hall, 2003) 等这样的一些书籍。

D.2.1 基本数据类型

由 C++ 语言提供的这些基本数据类型 (primitive data type) 与 Java 或者 C# 中的数据类型很相似。下表列出了 C++ 的基本类型以及它们在 Qt 4 所支持的平台上的定义。

表 4-1: C++ 基本类型

C++ 类型	说明
bool	布尔值
char	8 位整型值
short	16 位整型值
int	32 位整型值
long	32 位或 64 位整型值
long long[a]	64 位整型值
float	32 位浮点值 (IEEE 754)
double	64 位浮点值 (IEEE 754)
O	奇数页 (odd)

[a] 微软将 long long 类型称为 _int64，在 Qt 程序中，可以使用 qlonglong 类型来代替 long long 类型，qlonglong 可以运行于所有 Qt 支持的平台上。

默认情况下，short、int、long 和 long long 数据类型都是符号型数据。也就是说，它们既可以保存负数值，又可以保存正数值。如果只需要存储非负型整数值，只需把 unsigned 关键字放在该类型前面即可。因此，一个 short 变量可以保存介于 -32 768 到 +32 767 之间的任意值，而一个 unsigned short 变量则只能保存介于 0 到 65 535 之间的任意值。如果在操作符中有这个 unsigned 操作符，那么右移操作符 » 就会具备 unsigned（“用多个 0 填充”）的语法含义。

bool 类型可以接受的值是 true 和 false。此外，数字类型也可以用于需要 bool 值的地方，其使用规则是：0 表示 false，而其他任意的非零值表示 true。

char 类型用于存储 ASCⅡ 字符和 8 位整型值（字节）。当用作整型值时，它可以是 signed 或者 unsigned 类型，这取决于所在的平台。类型 signed char 和 unsigned char 可用于在那些能够区分 char 正负的地方代替 char。Qt 提供了一个 QChar 类型，它可用于存储 16 位的 Unicode 字符。

内置类型的实例不会被默认初始化。当创建一个 int 变量时，它的值应当可以明确地说是 0，但是也很有可能是 -209 486 515。幸运的是，绝大多数的编译器都会在我们试图读取未初始化变量时给予警告，并且我们也可以使用一些像 Rational PurifyPlus 和 Valgrind 这样的工具来检查运行时对未初始化内存的访问和其他与内存相关的问题。

在内存中，数值类型（long 除外）在 Qt 所支持的不同平台上都具有相同的大小，但它们的表示方法会根据系统存储字节顺序的不同而略有不同。对于高字节在后的系统架构（如 PowerPC 和 SPARC），32 位变量值 0x12345678 会存储为 4 个字节：0x12 0x34 0x56 0x78。然而，对于高字节在前的系统架构（比如 Intel x86 体系），这些字节的存储顺序就会颠倒过来。这样就会在一些需要把内存区域中的数据复制到磁盘或者是在网络上发送二进制数据的程序中产生差异。Qt 的 QDataStream 类（参见第 12 章的说明）则可用于存储与平台无关的二进制数据。

D.2.2 类定义

在 C++ 中，类定义 (class definition) 与 Java 和 C# 中的类定义相似，但是也有一些不同之处需要注意。我们将使用一系列的例子来研究这些不同点。先从一个表示 (x, y) 坐标对的类开始：

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D{
public:
    Point2D() {
        xVal = 0;
        yVal = 0;
    }
    Point2D(double x, double y) {
        xVal = x;
        yVal = y;
    }
    void setX(double x) { xVal = x;}
    void setY(double y) { yVal = y;}
    double x() const{ return xVal;}
    double y() const{ return yVal;}

```

```
19 private:
20     double xVal;
21     double yVal;
22 };
23
24 #endif
```

上面的类定义将会放在一个头文件中，通常把这个文件命名为 point2d.h。这个例子说明了 C++ 的以下几个特性：

- 类定义可以划分为 public、是和 private 三段，且以一个分号结束。如果没有定义段，那么就默认是 private 段。（为了保持与 C 的兼容性，C++ 提供了一个 struct 关键字，除了在没有指定段时它的默认段是 public 这一点不同外，其他都与类相同。）
- 类有两个构造函数（一个没有参数，一个则有两个参数）。我们没有声明构造函数，那么 C++ 将会自动提供一个不带参数的构造函数，并且这个构造函数的函数体为空。
- 用来获取值的函数 x() 和 y() 声明为 const。这就意味着它们不会（而且也不能）修改成员变量或者调用非 const 成员函数 [比如 setX() 和 setY()]。

上述的这些函数都实现为内联函数 (inline)，是类定义的一部分。还有另外一种方式是只把函数原型放在头文件中，而把实现这些函数的代码放在.cpp 文件中。使用这种方式时，头文件看起来应当是这样的：

```
1 #ifndef POINT2D_H
2 #define POINT2D_H
3
4 class Point2D{
5 public:
6     Point2D();
```

```
    Point2D(double x, double y);
    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;

private:
    double xVal;
    double yVal;
};

#endif
```

于是就可以在 point2d.cpp 文件中实现这些函数：

```
#include "point2d.h"

Point2D::Point2D(){
    xVal = 0.0;
    yVal = 0.0;
}

Point2D::Point2D(double x, double y){
    xVal = x;
    yVal = y;
}

void Point2D::setX(double x){
    xVal = x;
}

```

```
17 void Point2D::setY(double y){
18     yVal = y;
19 }
20
21 double Point2D::x() const{
22     return xVal;
23 }
24
25 double Point2D::y() const{
26     return yVal;
27 }
```

文件是从包含 point2d.h 开始的，因为编译器需要在它分析类的成员函数的实现之前要先知道类的定义。然后，我们再实现这些函数，并且在函数的名字前加上以”::”操作符和类名一起构成的前缀。

我们在前面看到了如何把一个函数实现成内联函数的方法，而现在则看到了如何在.cpp 文件中实现它。从语法上来讲，这两种方法是等效的，但是当我们调用一个声明为内联函数的函数时，绝大多数的编译器都只是简单地扩展其函数体，而不会生成实际的函数调用。这通常可以产生更为快速的代码，但是可能会增加应用程序的大小。基于这样的原因，只有非常简短的函数才应该实现为内联函数，比较长的函数都总是应当在.cpp 文件中加以实现。此外，如果我们忘记了某个函数的实现并试图去调用这样的函数，那么连接程序将会报错：”unresolved symbol”（不可解析的符号）。

现在尝试一下这个类：

```
1 #include ”point2d.h”
2
3 int main(){
4     Point2D alpha;
5     Point2D beta(0.666,0.875);
6
```

```
    alpha.setX(beta.y());
    beta.setY(alpha.x());

    return 0;
}
```

在 C++ 中，任意类型的变量都可以直接声明而不必一定要使用 new。第一个变量会使用默认的 Point2D 构造函数（这个构造函数没有参数）进行初始化。第二个变量则使用第二个构造函数进行初始化。对一个对象的成员进行访问需要使用”.”（点）操作符。

以这种方式声明变量的行为就像在 Java/C# 中声明一些基本类型一样，比如 int 和 double。例如，当使用赋值操作符时，会复制变量的内容，而不是复制对象的引用 (reference)。如果要在以后修改一个变量值，那么从它那里赋值而来的其他任何变量都仍旧会保持不变。

作为一种面向对象的语言，C++ 支持继承 (inheritance) 和多态 (polymorphism)。为了说明它们是如何工作的，我们将分析一个例子，该例以 Shape 为抽象基类，以 Circle 为子类。先从基类开始：

```
#ifndef SHAPE_H
#define SHAPE_H

#include ”point2d.h”

class Shape{
public:
    Shape(Point2D center) { myCenter = center; }
    virtual void draw() = 0;
protected:
    Point2D myCenter;
};

```

```
#endif
```

这个定义放在头文件 shape.h 中。由于在这个类的定义中引用了类 Point2D，所以需要包含头文件 point2d.h。

类 Shape 没有基类。这一点不像 Java 和 C#，C++ 没有为所有的类提供一个可以从中继承出来的一般类 Object。Qt 则为所有类型的对象提供了一个简单基类 QObject。

draw() 函数的声明有两个有趣的特点：它含有 virtual 关键字，并且以”=0” 为结尾，关键字 virtual 表明这个函数可能会在子类中重新得到实现。就像在 C# 中一样，C++ 的成员函数在默认情况下也是不能重新实现的。这个奇特的”=0” 的语句表明这个函数是一个纯虚函数 (pure virtual function)——一个没有默认实现代码并且必须在子类中实现的函数。要把 Java 和 C# 中的“接口”的概念对应到类中，就只能用 C++ 的纯虚函数来表示了。

以下是子类 Circle 的定义：

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include ”shape.h”

class Circle : public Shape{
public:
    Circle(Point2D center, double radius = 0.5)
    : Shape(center) {
        myRadius = radius;
    }

    void draw() {
        // do something here
    }
```

```
16
17 private:
18     double myRadius;
19 };
20
21 #endif
```

类 Circle 通过公有 (public) 方式继承了 Shape，也就是说，Shape 中的所有公有成员在 Circle 中仍旧是公有的。C++ 也支持保护 (protected) 继承和私有 (private) 继承，利用它们可以限制对基类的 public 成员和 protected 成员的访问。

这个构造函数带有两个参数。第二个参数是可选的，并且如果没有给定参数值就会取 0.5。构造函数在函数名和函数体之间使用一种特殊的语法把 center 参数传递给基类的构造函数。在函数体中，我们对成员变量 myRadius 进行了初始化。在基类构造函数初始化时，我们也本应在同一行初始化该变量：

```
Circle(Point2D center, double radius = 0.5)
    : Shape(center), myRadius(radius) { }
```

另一方面，C++ 不允许在类定义中初始化成员变量，因此，下面的代码就是错误的：

```
// 将无法编译通过
private:
    double myRadius = 0.5;
};
```

draw() 函数与 Shape 中声明的虚函数 draw() 具有相同的名字。它是该函数的一个重新实现，并且在 Circle 实例上通过 Shape 引用或者指针调用 draw() 时，就会以多态的形式调用该函数。C++ 不像 C# 那样有 override 关键字。而且 C++ 也没有能够指向基类的 super 或者 base 关键字。如果需要调用一个函数的基本实现，则可以在这个函数的名字前加上一个由基类的名字和”::” 操作符构成的前缀。例如：

```
class LabeledCircle : public Circle
{
public:
    void draw() {
        Circle::draw();
        drawLabel();
    }
...
};
```

C++ 支持多重继承，也就是说，一个类可以同时从多个类中派生出来。语法形式如下所示：

```
class DerivedClass : public BaseClass1, public BaseClass2, ...,
                public BaseClassN{
    ...
};
```

默认情况下，类中声明的函数和变量都与这个类的实例相关。我们也可以声明静态 (static) 成员函数和静态成员变量，可以在没有实例的情况下使用它们。例如：

```
#ifndef TRUCK_H
#define TRUCK_H

class Truck{
public:
    Truck() { ++counter; }
    ~Truck() { --counter; }
    static int instanceCount() { return counter; }

private:
    static int counter;
```

```
};

#endif
```

通过这里的静态成员变量 counter，我们可以在任何时候知道还存在多少个 Truck 实例。Truck 的构造函数会增加它的值。通过前缀”~” 识别的析构函数 (destructor) 可以减少它的值。在 C++ 中，在静态分配的变量超出作用域或者是在删除一个使用 new 分配的变量时会自动调用这个析构函数。除了我们还可以在某个特定时刻调用析构函数这一点之外，这都与 Java 中的 finalize() 方法相似。

一个静态成员变量在一个类中只有单一的存在实体：这样的变量就是“类变量” (class variable) 而不是“实例变量”(instance variable)。每一个静态成员变量都必须定义在.cpp 文件（但是不能再次重复 static 关键字）中。例如：

```
#include ”truck.h”

int Truck::counter = 0;
```

不这样做将会在连接时产生一个”unresolved symbol”（不可解析的符号）的错误信息。只要把类名作为前缀，就可以在该类外面访问这个 instanceCount() 静态函数。例如：

```
#include <iostream>
#include ”truck.h”

int main(){
    Truck truck1;
    Truck truck2;
    std::cout << Truck::instanceCount() << ” equals 2” << std::endl;
    return 0;
}
```

D.2.3 指针

在 C++ 中，指针 (pointer) 就是一个可以存储对象的内存地址的变量（而不是直接存储这个对象）。Java 和 C# 都有类似的概念——“引用”(reference)，但是在语法上却并不相同。我们从研究一个精心设计的例子开始，利用它来说明指针的用法：

```
1  #include "point2d.h"
2
3  int main(){
4      Point2D alpha;
5      Point2D beta;
6      Point2D *ptr;
7      ptr = &alpha;
8      ptr->setX(1.0);
9      ptr->setY(2.5);
10     ptr = &beta;
11     ptr->setX(4.0);
12     ptr->setY(4.5);
13     ptr = 0;
14     return 0;
15 }
```

这个例子依赖于前一小节中给出的 Point2D 类。第 4 行和第 5 行定义了两个 Point2D 对象。根据 Point2D 的默认构造函数，这两个对象将被初始化为 (0, 0)。

第 6 行定义了一个指向 Point2D 对象的指针。指针的语法是在变量名的前面再加上一个星号。由于没有初始化这个指针，所以它包含的是一个随机的内存地址值。通过在第 7 行给这个指针分配 alpha 对象的地址就可以解决这个初始化问题。这里的一元运算符”&” 可以返回一个对象的内存地址值。地址值通常是一个 32 位或者是一个 64 位的整型值，可以用来确定一个对象在内存中的偏移量。

在第 8 行和第 9 行，我们通过 ptr 指针访问 alpha 对象。因为 ptr 是指针而不是对象，所以必须使用”->”（箭头）操作符代替”.”（点）操作符。

在第 10 行，我们把 beta 的地址也赋给这个指针。于是从此时开始，通过这个指

针执行的任何操作都将会影响到 beta 对象。

第 13 行把这个指针设置为空 (null) 指针。C++ 没有一个可以用于表示不指向对象指针的关键字。所以，我们改换用值 0（或者是符号常量 NULL，它可以扩展为 0）来代替。试图使用一个空指针会造成系统的崩溃，其提示的错误信息有“段错误”(Segmentation fault)、“常规保护错误”(General protection fault) 或者是“总线错误”(Bus error) 等。使用程序调试器，可以找出是那一行代码造成了系统的崩溃。

在这个函数的最后，alpha 对象保存了坐标对 (1.0, 2.5)，而 beta 保存了 (4.0, 4.5)。

指针通常用于存储使用 new 动态分配的对象。在 C++ 术语中，我们把这样的对象称为是分配在“堆”(heap) 上，而局部变量（在一个函数中定义的变量）则存储在“栈”(stack) 里。

这里给出了一段用来说明使用 new 进行动态内存分配的代码片段：

```
#include "point2d.h"


int main(){
    Point2D *point = new Point2D;
    point->setX(1.0);
    point->setY(2.5);
    delete point;
    return 0;
}
```

new 操作符返回一个新近分配对象的内存地址。我们把这个地址存储在一个指针变量中，并且通过这指针访问该对象。当处理完这个对象后，就可以使用 delete 操作符释放它的内存。不像 Java 和 C#，C++ 没有垃圾信息收集器，当不再需要那些动态分配的对象时，就必须明确使用 **delete** 来释放它们。利用第 2 章中讲述的 Qt 父－子对象机制，可以大大简化 C++ 程序中的内存管理工作。

如果忘记调用 delete，则内存就会一直保留到该程序结束时为止。这在上面的例子中不是什么大问题，因为我们只是分配了一个对象，但是如果在一个总是需要不断分配新对象的程序中，就可能造成程序总是在不断分配内存，那么就可能将机器的内存耗尽。对象一旦删除，则指向该对象的指针变量仍旧会保存这个对象的地址值。这

样的指针就称为“悬摆指针”(dangling pointer)，最好不要再使用这样的指针访问该对象。Qt 提供了一种“智能”(smart) 指针 QPointer<T>，如果删除了它所指向的 QObject 对象，那么它就会自动把自己设置成 0。

在上面的例子中，我们调用了默认的构造函数并且调用 setX() 和 setY() 来初始化该对象。我们本应当使用带两个参数的构造函数来代替默认的构造函数：

```
Point2D *point = new Point2D(1.0, 2.5);
```

这个例子并不需要使用 new 和 delete。我们最好也像下面那样在栈上分配该对象：

```
Point2D point;
point.setX(1.0);
point.setY(2.5);
```

像这样分配的对象会在出现它们的程序块的末尾自动得到释放。

如果不打算通过该指针来修改这个对象，则可以把指针声明为 const 型指针。例如：

```
const Point2D *ptr = new Point2D(1.0, 2.5);
double x = ptr->x();
double y = ptr->y();

// WON'T COMPILE
ptr->setX(4.0);
*ptr = Point2D(4.0, 4.5);
```

这个常量指针 ptr 只能用于调用常量成员函数，比如 x() 和 y()。当不打算使用指针修改它们时，把指针声明为 const 是一种不错的习惯。而且，如果该对象自身就是常量，那么我们就没有什么选择了，只能使用常量指针来存储它的地址值。const 的用法可以为编译器提供一定的信息，这可以提早发现一些 bug，并且也可以获得良好

的性能。C# 有 const 关键字，与 C++ 的 const 关键字相似。而在 Java 中，最为接近的等价概念就是 final 了，但是它只能保护变量不被赋值，避免在它上面调用“非常量”的成员函数。

指针既可以用在内置类型上，也可以用在类上。需要说明的是，一元运算符”*” 可以返回与这个指针相关的对象的值。例如：

```
int i = 10;
int j = 20;

int *p = &i;
int *q = &j;

std::cout << *p << ” equals 10” << std::endl;
std::cout << *q << ” equals 20” << std::endl;

*p = 40;

std::cout << i << ” equals 40” << std::endl;

p = q;
*p = 100;

std::cout << i << ” equals 40” << std::endl;
std::cout << j << ” equals 100” << std::endl;
```

箭头运算符”->” 可用于通过指针来访问对象的成员，这纯粹是一种语法糖 (syntactic sugar) 而已。除了 ptr->member 的形式之外，我们还可以使用 (*ptr).member 的形式。这里的圆括号是必需的，因为”.” 运算符具有比”*” 运算符更高的运算优先级。

指针在 C 和 C++ 中名声不良，正是因为这一点，Java 经常借鼓吹自己没有指针而大做文章。实际上，C++ 指针在概念上与 Java 和 C# 中的引用非常相似，只是我们还可以使用指针来遍历整个内存而已——关于这一点，在这一节的后面还会讲到。

此外，在 Qt 中还包含了“写时复制”(copy on write) 的容器类，它具有与 C++ 一样的可在栈上实例化任意类的能力，这就意味着通常可以尽量避免指针的使用。

D.2.4 引用

除了指针，C++ 也支持“引用”的概念。像指针一样，一个 C++ 的引用存储的也是一个对象的地址值。两者的主要不同点在于：

- 声明引用时使用的是”&” 而不是”*”。
- 引用必须是初始化过的，并且不能在后面再次重新赋值。
- 可以直接访问与引用相关联的对象，且没有像”*” 或者”->” 这样的特殊语法。
- 引用不能为空 (null)。

在声明参数时，经常会用到引用。对于大多数类型来讲，C++ 会使用按值调用 (call-by-value) 的方式来作为它的默认参数传递机制。也就是说，当给一个函数传递参数的时候，该函数会接收到这个对象的一个新的副本。这里给出了一个函数的定义，它就是通过按值调用的方式来接收它的参数值的：

```
#include <cstdlib>
double manhattanDistance(Point2D a, Point2D b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

于是就可以按照如下的方式来调用该函数：

```
Point2D broadway(12.5, 40.0);
Point2D harlem(77.5, 50.0);
double distance = manhattanDistance(broadway, harlem);
```

C 程序员通过把参数声明为指针而不是值的方式，能够避免不必要的复制操作：

```
double manhattanDistance(const Point2D *ap, const Point2D *bp)
{
    return std::abs(bp->x() - ap->x()) + std::abs(bp->y() - ap->y());
}
```

于是，在调用该函数的时候传递的必须是地址而不是值：

```
double distance = manhattanDistance(&broadway, &harlem);
```

C++ 引入引用的概念而使得语法变得更为简单，并且还可以使调用者避免出现传递空指针的现象。如果使用的是引用而不是指针，那么该函数看起来就会像下面这样：

```
double manhattanDistance(const Point2D &a, const Point2D &b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

引用的声明与指针的声明有点相似，只是用的是”&” 而不是”*” 罢了。但是当我们实际使用引用的时候，无需记住它是一个内存地址，而只需把它看作是一个普通的变量就行了。另外，调用一个带有引用作参数的函数时；并不需要给予太多的考虑（不带”&” 运算符）。

总而言之，通过在参数列表中把 Point2D 替换为 const Point2D & ，就可以降低函数调用的开销：不用再复制 256 位（4 个 double 值的大小），需要复制的只是 64 位或者 128 位——这取决于目标平台指针的大小。

在前一个例子中使用 const 引用，就可以让函数避免修改与这些引用相关联的对象。但是当我们需要这种特殊效果时，则可以传递一个非常量引用或者一个指针。例如：

```
void transpose(Point2D &point)
{
```

```
    double oldX = point.x();
    point.setX(point.y());
    point.setY(oldX);
}
```

在某些情况下，我们有一个引用并且需要调用一个带指针的函数，或者是相反的情形。要把引用转换成指针，可以使用一元运算符”&”：

```
Point2D point;
Point2D &ref = point;
Point2D *ptr = &ref;
```

要把指针转换成引用，可以使用一元运算符”*”：

```
Point2D point;
Point2D *ptr = &point;
Point2D &ref = *ptr;
```

引用和指针在内存中的表达方式一样，并且在使用时会经常互换它们，这需要根据具体问题来具体确定到底应该使用哪一种形式。一方面，引用具有更为简便的语法；另一方面，指针可以在任何时刻指向其他的任意对象，它们都可以保存一个空值，并且它们更为明晰的语法通常可以使事情变得似繁而简。基于这些原因，指针会略占上风，而用于声明函数的参数时引用则是最佳的选择，它们还可以与 const 一起配合使用。