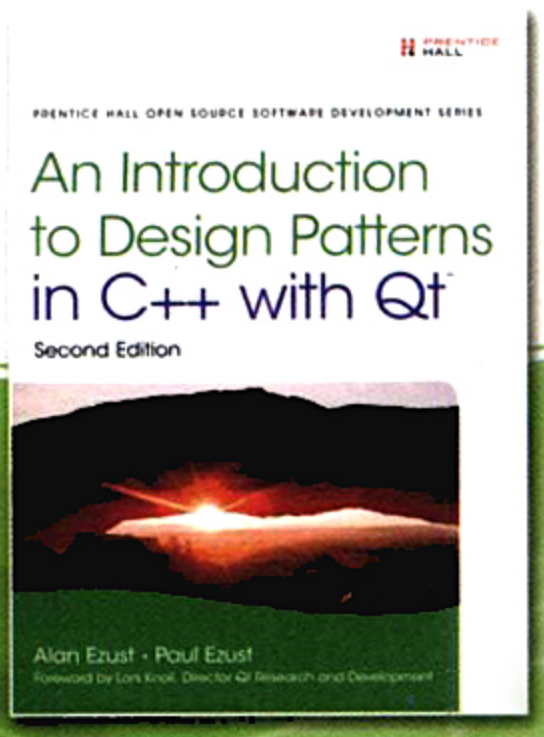
PEARSON

C++ Qt 设计模式(第二版)

An Introduction to Design Patterns in C++ with Qt

Second Edition



[美] Alan Ezust
Paul Ezust 著

闫锋欣 张学敏 张君施 等译

张德保 审

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

C++ Qt 设计模式

（第二版）

An Introduction to Design Patterns in C++ with Qt

Second Edition

Alan Ezust
Paul Ezust
著

闫锋欣　张学敏　张君施　等译

张德保　审

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是美国萨福克大学已使用十余年的经典教程，利用跨平台开源软件开发框架 Qt 阐释了 C++和设计模式中的主要思想。全书共分四个部分：第一部分介绍 C++、UML、Qt、模型-视图、SQL、XML、设计模式等基础知识，目的是为零基础的 C++初学者铺垫一条学习面向对象编程的快捷之路；第二部分讲解内存访问、继承等重要的 C++特性，是前一部分的延伸和拓展；第三部分使用 Phonon 编写了一个多媒体播放器，展示了主要技术理念的应用方法；附录部分给出了 C++保留关键字、Debian 和 Qt 程序开发环境的配置等内容。每节的练习题和各章后面的复习题，既可作为课堂上的讨论题，也可进一步启发读者对于关键知识点的思考。

本书可作为软件开发人员学习 Qt 开发技术的参考书，也可作为从事 Qt 软件开发的研究人员和科技工作者的工具书。

版权贸易合同登记号 图字：01-2012-1498

图书在版编目(CIP)数据

C++ Qt 设计模式：第 2 版/(美)艾朱斯特(Ezust, A.)，(美)艾朱斯特(Ezust, P.)著；闫锋欣等译.
北京：电子工业出版社，2012.7
书名原文：An Introduction to Design Patterns in C++ with Qt
ISBN 978-7-121-16890-1

I. ①C… II. ①艾… ②艾… ③闫… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 080514 号

策划编辑：许菊芳
责任编辑：许菊芳
印　　刷：
装　　订：北京京师印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×1092　1/16　印张：31.75　字数：813 千字
印　　次：2012 年 7 月第 1 次印刷
定　　价：78.00 元

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

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

前　言

C++在 1989 年被标准化之前，就已经被广泛使用了很多年，这使得 C++比当今流行的其他编程语言要相对成熟一些。C++是一种能够用于创建快速、高效且任务关键的系统的重要语言。C++同时也是最为灵活的编程语言之一，能够给开发人员提供多种编程风格，其中囊括了从高级的 GUI 代码到低级的设备驱动程序。

20 世纪 90 年代的最初几年，C++是最流行和使用最为广泛的面向对象(Object-Oriented，OO)编程语言。许多计算机科学(Computer Science，CS)专业的学生都是借助 C++来学习面向对象编程的。这是因为 C++允许 C 程序员相对容易地转换到面向对象编程(Object-Oriented Programming，OOP)，而在此之前，许多 CS 教授则一直在讲授 C 语言编程。

从 1996 年左右开始，Java 超越 C++成为学生们开启学习的一种面向对象语言。Java 之所以会如此流行，是有许多原因的。

- 它比 C++语言更简单。
- 有内建的垃圾回收机制，因此程序员无须关注令人恼火的内存释放工作。
- 开发工具箱中包含了一个标准的 GUI 类集合。
- 内建的 String 类支持 Unicode。
- 支持多线程。
- 创建和“插入”Java 档案(Java Archive，JAR)要比重新编译和链接库容易得多。
- 许多 Web 服务器提供 Java API，能够容易地集成。
- Java 程序是平台独立的(Wintel、Solaris、MacOS、Linux、*nix，等等)。

通过将 C++和 Qt 结合使用，也可以获得上述许多优点。

- Qt 提供了一组更容易理解的 GUI 类，而且与 Java 的 Swing 类相比，其运行更快，看起来更舒服，使用起来也更加灵活。
- 信号和槽的使用要比 Java 中的(Action/Event/Key)Listener 接口更容易。
- Qt 拥有插件体系结构，这使得可以将代码加载到一个应用程序中而无须重新编译或者链接。
- Qt 提供 foreach 机制，可以对集合进行迭代，其读写操作更为简单。

尽管 Qt 没有提供垃圾回收机制，但它提供的各种替代方法可用来避免在代码中直接进行堆对象的删除操作。

- 容器(参见 6.8 节)支持引用计数和写时复制。
- 父对象和子对象(参见 8.2 节)。
- QPointer、QSharedPointer 和 QWeakReference(参见 19.11 节)。
- 对象子类化(参见 2.14 节)。
- 栈对象(参见 20.3 节)。

目　录

第一部分　设计模式与 Qt

第二部分 C++语言规范

第三部分 编 程 作 业

第一部分　设计模式与 Qt

第1章　C++简介

本章介绍C++编程语言。将给出一些基本的概念，如关键字、常量、标志符、声明、基本类型以及类型转换。还将给出C++的历史、演变过程以及它与C语言的关系。也会介绍几个标准库和Qt类。

1.1　C++概述

C++最初是在C中添加了一系列的预处理器宏，作为C的扩展而编写的，它被称为“带类的C”[①]。经过多年的演变和优化，C++在C的基础上添加了许多高级特性，比如强类型化、数据抽象、引用、运算符重载、函数重载以及对面向对象编程的大量支持。

C++保留了使C语言流行和成功的主要特性：速度、效率以及广泛的表达能力，这种表达能力使得程序员能够在从最低层(例如直接的操作系统调用和位操作)到最高层(例如操作包含大而复杂的对象的容器)的多个层次上进行编程。

C++设计之初的基本原则是：添加到C++中的任何功能，都不应导致不使用此功能的C语言代码的运行时开销[②]。C++中存在许多高级特性，它们使程序员能够编写出可阅读的、可复用的、面向对象的程序，而使用这些特性不应导致编译器做额外的大量工作。不过，为了维持程序的功能和代码的可维护性，付出一些小代价(稍长的编译时间)还是值得的。有些特性存在运行时开销，但是被C++编译器编译的C程序，应该与使用C编译器编译时运行得一样快。

1.2　C++简史

C++由Bjarne Stroustrup在AT&T公司Bell实验室工作时所设计，最终由Bell实验室打包并负责其市场化工作。1981年，AT&T公司内部开始出现最初的C++版本，其后C++根据用户的反馈逐步演化发展。

1986年初，Stroustrup撰写的图书 *The C++ Programming Language* 第一版发行。随着1989年C++ 2.0的发布，C++迅速成为一种严谨、实用的编程语言。同年，人们开始致力于制定C++的国际标准。1997年，美国国家标准化学会(American National Standards Institute，ANSI)的一个委员会完成并在内部公布了一个C++语言的草案标准，名称为 *Draft Standard The C++ Language*, X3J16/97-14882, Information Technology Council (NSITC), Washington, DC.。

1998年6月，参加过历时9年的ANSI/ISO(International Standards Organization，国际标准化组织)工作的来自20个主要国家的代表一致接受了该草案标准。Stroustrup撰写的 *The C++ Programming Language* 第三版于1997年出版，该书被公认为是权威的C++参考手册。

对标准的后续完善是由ISO及国际电工委员会(International Electrotechnical Commission,

---

① 参见 http://www.research.att.com/~bs/bs_faq.html#invention。

② 遗憾的是，异常处理打破了这一原则，如果启用会导致一些开销。这就是为什么许多库都不使用异常的原因。

IEC)负责的，IEC 是一家对电工学各个领域进行标准制订定规范评估的国际机构。2005 年发布的 Technical Report 1(也称为“TR1”)，对 C++语言和标准库进行了大量扩充。2010 年，负责 C++的国际标准工作组被命名为 ISO/IEC JTC1/SC22/WG21。C++草案标准 2010 版[①]可从网络自由获得。C++0x 是“C++的下一个版本”的非官方名称，有望于 2011 年定型[②]。

1.3　第一个 C++例子

在整本书中，都将通过代码例子来解释并演示重要的编程概念以及面向对象编程(Object Oriented Program, OOP)的思想。每一个例子的目标都是用最小的例子来扼要而有效地讲解概念和技术。示例 1.1 展示了 C++语法中的一些基本要素。

示例 1.1　src/early-examples/example0/fac.cpp

```
/* Computes and prints n! for a given n.
   Uses several basic elements of C++. */

#include <iostream>                                                    1
int main() {                                                           2
    using namespace std;                                               3
    // Declarations of variables
    int factArg = 0 ;                                                  4
    int fact(1) ;                                                      5
    do {                                                               6
        cout << "Factorial of: ";                                      7
        cin >> factArg;                                                8
        if ( factArg < 0 ) {
            cout << "No negative values, please!" << endl;
        }                                                              9
    } while (factArg < 0) ;                                            10
    int i = 2;
    while ( i <= factArg ) {                                           11
        fact = fact * i;
        i = i + 1;
    }                                                                  12
    cout << "The Factorial of " << factArg << " is: " << fact << endl;
    return 0;                                                          13
}                                                                      14
```

1 标准 C++库。在较老的 C++版本中，可能会发现<iostream.h>。但是那个版本已经过时，不提倡使用。

2 main 函数的开始，它返回一个 int 值。

3 允许使用符号 cin, cout 和 endl，而不必在每个符号前面加上“std::”。

4 C 语言风格的初始化语法。

5 C++语言风格的初始化语法。

6 do…while 循环的开始。

---

① 参见 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf。

② 国际标准化组织 ISO/IEC 于 2011 年 8 月发布了 C++11（先前被称作 C++0x）编程语言标准。——译者注

7 写到标准输出设备。
8 从标准输入设备读入并且转换成 int 类型。
9 if 语句块的结尾。
10 如果为 false，跳出 do 循环。
11 while 循环的开始。
12 while 语句块的结尾。
13 当 main 函数返回 0 时，通常表示没有错误。
14 main 语句块的结尾。

在大多数平台上，都可以使用随处可见的 GNU C 编译器 gcc 编译并运行这个程序。编译 C++程序的命令是 g++，这是一个调用 gcc 的程序，它将.c 文件和.h 文件当成 C++源文件，并会自动连接到 C++库。

为了尽量多地获得编译过程中的调试信息，需使用命令行选项-Wall。

```
src/early-examples/example0> g++ -Wall fac.cpp
src/early-examples/example0> g++ -Wall -o execFile fac.cpp
```

选项-Wall 为那些通常可能会造成问题的结构启用警告，即使它们符合标准。

在第二种方法中，可选的开关参数-o *execFile* 被用来指定所产生的可执行文件的名称。如果省略这个选项，则编译器会产生一个名称为 a.out 的可执行文件[①]。无论哪一种情况，如果在同一个目录下已经有一个与目标可执行文件重名的文件(例如，是在重新编译)，则编译器会不加提示地自动覆盖旧文件。

以上选项只不过是众多常用编译器选项中的两个而已。在*nix 系统中，通过输入命令

```
man g++
```

或者

```
info g++
```

来查看手册页(manual page)，它是关于命令选项及其使用方法的汇总。

在大多数系统中，这个命令使用户能够浏览 g++的在线文档，一次一屏。关于 gcc 文档的更完整信息，可以查看 GNU 的在线文档中心[②]。

编译成功后，可以输入可执行文件的名称来运行程序。下面是*nix 平台上的一个运行示例。

```
src/early-examples/example0> ./a.out
Factorial of: -3
No negative values, please!
Factorial of: 5
The Factorial of 5 is: 120
src/early-examples/example0>
```

这个简短的程序包含了大多数 C++程序中最常见的几个语言要素。

1.3.1 注释

与 Java 一样，C++也有单行注释。在一行代码中，位于//之后到行尾的任何文本，都被看成是注释。C++中也可以使用 C 语言风格的多行注释，即位于/*和*/之间的文本。

① 在 Windows 系统中，利用 mingw 命令可以产生一个名称为 a.exe 的文件。
② 参见 http://www.gnu.org/software/gcc/onlinedocs/。

#include 指令

为了复用库中的函数、类型或者标志符，需使用预处理器指令#include[①]。与 C 语言一样，C++中所有的预处理器指令都以字符#开头，且在编译器准备代码之前进行处理。本例中包含的<iostream>头文件，用来引入标准库的输入/输出定义。

使用命名空间

标准库(参见附录 B)中的符号都包含在命名空间 std 中。

命名空间(参见 20.4 节)是由类、函数和对象组成的一个集合，其中的元素都可以通过命名前缀定位。using 声明的作用是告诉编译器，要将命名空间 std 中的所有符号都加入全局命名空间中。

1.3.2　声明并初始化变量

C++中存在三种变量声明形式：

```
type-expr  variableName;
type-expr  variableName = init-expr;
type-expr  variableName (init-expr);
```

在第一种形式中，变量没有被初始化。第三种形式是第二种形式的变体。

1.3.3　选择

C++为选择结构和控制结构语法提供了几种常规的语法分类，见 19.2.2 节的讨论。

1.3.4　迭代

示例 1.1 中使用了三种迭代结构中的两种，19.2.3 节中将详细探讨这三种迭代结构。

1.4　标准输入与输出

示例 1.1 中，指令

```
#include <iostream>
```

使用户可以使用如下预定义的全局输入对象(istream)和输出对象(ostream)。

1. cin，控制台输入，默认为键盘。
2. cout，控制台输出，默认为控制台屏幕。
3. cerr，控制台错误，另一种输出到控制台屏幕的输出流，它频繁刷新，通常用来显示错误信息。

示例 1.1 中使用了全局 ostream 对象 cout，还调用了它的成员函数 operator<<()。这个函数重载了<<运算符，它用于将数据插入到输出流中，所以称其为插入运算符[②]。这种输出语句的语法也相当有趣，它没有使用笨重的函数记法

---

① 见 C.2 节中的讨论。

② 5.1 节将讨论重载的函数和运算符。这个特别的运算符已经在 C 语言中命名并定义，它就是左移运算符。例如，n << 2 会将 int 值 n 左移两位，并将空出来的两位用 0 填充，在效果上就是将 n 乘以 4。

```
cout.operator<<("Factorial of: ");
```

而是用更灵活、可读性更好的中缀语法来调用同一个函数

```
cout << "Factorial of: ";
```

这个运算符可以(在多个值上)被“链”起来，并且被预定义成能在许多内置类型上使用，下一条输出语句中就是这样。

```
cout << "The cost is $" << 23.45 << " for " << 6 << " items." << '\n';
```

示例 1.2 中可以看到 operator>>()用于 istream 对象 cin 的输入，就如同将<<用于 ostream 对象 cout 的输出一样。由于这个运算符的作用是从输入流中提取数据，因此称其为提取运算符[①]。

示例 1.2 src/iostream/io.cpp

```
#include <string>
#include <iostream>

int main() {
    using namespace std;
    const int THISYEAR = 2011;
    string yourName;
    int birthYear;

    cout << "What is your name? "  << flush;
    cin >> yourName;

    cout << "What year were you born? " ;
    cin >> birthYear;

    cout << "Your name is " << yourName
            << " and you are approximately "
            << (THISYEAR - birthYear)
            << " years old. " << endl;
}
```

符号 flush 和 endl 是来自 std 命名空间的操作算子(manipulator)[②]。

示例 1.2 中使用了 string 类[③]，它也来自于 C++标准库。1.8 节中将讨论这种类型，并演示它的一些函数的用法。

注意：Windows/MSVC 用户

默认情况下，MSVC 产生的应用不包含控制台支持。这意味着除非告诉 MSVC 要建立一个控制台应用，否则就不能在程序中使用标准的输入/输出流和错误流。可以参见 1.6 节中的段落“CONFIG += console(MSVC 用户)”。

① 这一次重载的是右移运算符。n >> 2 会将 int 值右移两位，实际上就是将 n 除以 4，而如何填充空出来的两位，与 n 是有符号值还是无符号值相关。

② 操作算子是一个函数引用，它能够插入到输入或者输出流中，以改变流的状态。1.9 节将更深入地探讨操作算子。

③ 第 2 章中将详细讲解类。现在，只需将类看成带有内置函数的一种数据类型。

1.4.1　练习：标准输入与输出

1. 本书附带资源的 dist 目录中提供了一个名称为 `src.tar.gz` 的 tarball(压缩文件)，它包含书中全部的示例代码。一种极其有用的练习方法是实践一下每一个代码示例，构建(build)并运行它，以查看设想中的执行过程，然后对它进行各种“破坏”，看会发生什么。有时，可以从这种练习中了解到编译器是如何响应特定的语法错误的。如果希望了解某一种语言，编译器就是最好的朋友。在其他情况下，可以看到特定的逻辑错误是如何影响程序的运行时行为的。无论是何种情况，这种经历都能够强化对 C++ 编程的理解。
2. 利用示例 1.2 完成下面的任务。

   - 首先，编译并运行它，查看它的正常执行过程。
   - 如果输入的出生年份是一个非数字值，会发生什么?
   - 如果输入类似“Curious George”的姓名，会发生什么?
   - 如果删除下面这一行，会发生什么?

     ```
     using namespace std;
     ```

   - 将语句

     ```
     cin >> yourName;
     ```

     替换成

     ```
     getline(cin, yourName);
     ```

     然后再次输入姓名“Curious George”，会发生什么?

   - 请解释 `cin >>`与 `getline()` 的不同表现。1.8 节将探讨这种不同。
   - 为程序添加更多的问题，要求得到各种不同的数字和字符串答案，并测试结果。

1.5　函数介绍

每一种现代的编程语言都具有使程序员能够定义函数的途径。函数使程序能够被分解成各种可管理的组件，而不是一个庞大而复杂的整体。这样就可以单个地或者小规模地开发并测试小的组件，从而能够更容易地构建并维护软件。函数也为针对特定任务编写可复用的代码提供了途径。例如，示例 1.1 在 `main()` 函数中计算某个数的阶乘，示例 1.3 展示了如何抽取用于计算阶乘的代码并将它转换成一个可复用的函数。

示例 1.3　src/early-examples/fac2.cpp

```
#include <iostream>

long factorial(long n) {
    long ans = 1;
    for (long i = 2; i <= n; ++i) {
        ans = ans * i;
        if (ans < 0) {
            return -1;
        }
    }
```

```
    return ans;
}

int main() {
    using namespace std;
    cout << "Please enter n: " << flush;
    long n;
    cin >> n;

    if (n >= 0) {
        long nfact = factorial(n);
        if (nfact < 0) {
            cerr << "overflow error: "
                 << n << " is too big." << endl;
        }
        else {
            cout << "factorial(" << n << ") = "
                 << nfact << endl;
        }
    }
    else {
        cerr << "Undefined:  "
             << "factorial of a negative number: " << n << endl;
    }
    return 0;
}
```


1
2

1 long int。

2 从 stdin 读取，尝试将它转换成 long 类型。

除了构造函数、析构函数[①](参见第 2 章)以及转换运算符(参见 19.9.1 节)之外，每一个函数都必须具有

- 返回类型(可能为 void)
- 名称
- 一个代表函数参数类型的有序的、以逗号分隔的列表(可以为空)
- 函数体(包含在一对大括号里的零条或者多条语句)

前三项表示函数的接口，最后一项是函数的具体实现。

示例 1.3 中，函数定义出现在调用它的语句的上面。但是，并不总是可以将函数定义置于调用它的每一条语句之前，有时也不希望这样做。C 和 C++语言允许在定义函数之前就调用它，只要这个函数在调用之前已经被声明了即可。

用于声明函数(即向编译器描述如何调用它)的机制是函数原型(function prototype)。函数原型中包含如下信息。

- 函数的返回类型
- 函数的名称
- 函数的参数表

① 构造函数一定不能有返回类型，且其函数体可以为空。析构函数一定不能有返回类型，但必须有空的参数表，且其函数体可以为空。

换句话说，函数原型包括除函数体之外的其他函数元素。以下是几个函数原型。

```
int toCelsius(int fahrenheitValue);
QString toString();
double grossPay(double hourlyWage, double hoursWorked);
```

记住，在函数首次被使用之前，必须声明或者定义它，这样编译器就能够正确地设置对它的调用。20.1 节中将更详细地探讨函数的声明与定义。除了 main() 函数外，省略函数的返回类型是一个错误(即使其为 void)，main() 函数的返回类型被隐式地默认为 int。

尽管在函数原型中参数名称可有可无，但在编程实践中最好是将其加上。对程序的文档化而言，它们构成了一个有效的部分。

一个简单的例子就足以说明为什么函数原型中应当使用参数名称。假设需要一个函数来用年、月、日的值设置 Date 变量的值。如果函数原型为 setDate(int, int, int)，则当调用这个函数时，使用 Date 变量的程序员就无法单独从函数原型中立即知晓这三个值的顺序。因为世界上使用三种日期顺序的现象很常见，所以不存在“显而易见的”答案来减少对更多信息的需求。2.2 节中将看到，成员函数的定义与声明通常位于两个不同的文件中(且可能无法访问它的定义文件)，这样程序员就难以推算出应该如何调用它。为参数设置一个好的名称，就可以解决这个问题，且函数本身的文档化工作已经部分完成了[①]。

函数重载

C++中允许重载（overloading）函数的名称。这使得程序可以利用具有不同参数的相同函数名称实现不同的任务。

函数的签名(signature)由名称和参数表组成。C++中，函数的返回类型不属于签名。

如果两个或者多个函数在某个给定的作用域内的名称相同但它们的签名不同，就称此函数名称被重载了。如果位于同一作用域中的两个函数具有相同的签名而返回类型不同，则会导致错误。重载要求编译器通过分析实参表(argument list)来判断应该执行被重载函数的哪一个版本，以响应函数调用。由于这种判断是完全以实参表为基础的，所以容易看出为什么编译器不能允许在同一个作用域内同时存在签名相同而返回类型不同的两个函数。5.1 节将探讨这种判断过程。同时，示例 1.4 提供了需要认真分析的几个函数调用。

示例 1.4　src/functions/overload-not.cpp

```
#include <iostream>
using namespace std;

void foo(int n) {
  cout << n << " is a nice number." << endl;
}


int main() {
   cout << "before call: " << 5 << endl;
   foo(5);
   cout << "before call: " << 6.7 << endl;
```

① 15.2 节中分析了一个示例，从中可以看出省略某些参数名称的好处。

```
    foo(6.7);
    cout << "before call: " << true << endl;
    foo(true);
}
```

这里只有一个函数，但用三种不同的数字类型调用了它。自动类型转换允许对这个函数的三次调用。

```
src/functions> g++ overload-not.cpp
src/functions> ./a.out
before call: 5
5 is a nice number.
before call: 6.7
6 is a nice number.
before call: 1
1 is a nice number.
src/functions>
```

输出反映了几种数字类型的“残酷”现实。首先，当将浮点数转换成 int 值时，会丢弃它的小数部分(即小数点及其右侧的全部数字)。尽管与 6 相比，6.7 更接近于 7，但那是没有出现四舍五入的情况。其次，布尔值 true 被显示成 1(false 被显示成 0)。如果希望看到单词 true(或者 false)，则需要添加代码以输出合适的字符串，如示例 1.5 所示。

这个示例中使用了重载函数。

示例 1.5 src/functions/overload.cpp

```
#include <iostream>
using namespace std;

void foo(int n) {
  cout << n << " is a nice int." << endl;
}

void foo(double x) {
  cout << x << " is a nice double." << endl;
}

void foo(bool b) {
   cout << "Always be " << (b?"true":"false") << " to your bool." << endl;
}

int main() {
  foo(5);
  foo(6.7);
  foo(true);
}
```

利用函数的三个重载版本，当使用同一个 main() 函数时无须进行类型转换。应注意 foo() 函数第三个版本中条件运算符的使用[①]。

① 三元条件运算符 testExpr ? valueIfTrue : valueIfFalse，为在一个表达式中插入一个简单的选择结果提供了一种简易的方法。如果 testExpr 具有非零值(例如，true)，则返回的是紧跟在问号右边的那一个值；如果 testExpr 具有零值(例如，false)，则返回的是冒号右边的那一个值。

```
src/functions> g++ overload.cpp
src/functions> ./a.out
5 is a nice int.
6.7 is a nice double.
Always be true to your bool.
src/functions>
```


第 5 章中将更详细地探讨 C++函数的许多有趣而有用的特性。

1.5.1　练习：函数简介

1. 找到位于 src 目录下示例 1.3 的代码，然后完成下列任务。

   - 建立该程序并用各种输入值测试它，包括非数字值。
   - 确定能够获得有效输出的最大输入值。
   - 修改程序，使其能为更大的输入值产生有效的输出结果。
   - 修改程序，使其不会产生无效的输出结果。
   - 分析在程序中包含处理 n 的语句

     ```
     if (cin >> n)  {  ...  }
     ```

     的作用。特别地，尝试在提示符之后输入非数字数据。这是使用转换运算符的一种情况，19.9.1 节中将更详细地探讨。

   - 修改程序，使其不断接收用户输入的值，直到输入 9999 为止。

2. 编写如下两个函数的定义

   ```
   double toCelsius(double fahrenheitTemp);
   double toFahrenheit(double celsiusTemp);
   ```

   然后编写程序，让用户根据一种温度值获得另一种温度值(摄氏温度和华氏温度)。

1.6　qmake，工程文件及 Makefile

C++应用通常由许多个源文件、头文件和外部库组成。在工程开发的常规过程中，会添加、修改或者删除源文件和库。为了建立能够反映工程当前状态的可执行程序，这样的更改要求编译全部受影响的文件，并且要求正确地链接(link)了结果目标文件。这种更改-重建的过程，通常都会发生许多次。

为了跟踪工程的全部部分，要求有一种机制来精确地指定涉及的输入文件、建立程序时所需的工具、中间目标文件和它们的依赖关系，以及最终的可执行目标文件。

用于处理工程建立任务的最广泛使用的工具是 make[①]，它会从 Makefile 中读取工程规范的细节，也会读取提供给编译器的指令。Makefile 与 shell 脚本类似，但它(至少)包含如下信息。

- 用于建立某种文件类型的规则(例如，为了从.cpp 文件得到.o 文件，必须对.cpp 文件执行 gcc -c 指令)。
- 源文件列表和头文件列表，它们包含工程所需的全部源文件和头文件的名称。

---

① 根据开发环境的不同，make 有各种不同的变体，比如 mingw32-make，gmake 或者 cmake 等。在 MS Dev Studio 中，其名称为 nmake。

- 目标文件指定了必须建立哪些可执行文件(或者库)。
- 依赖关系列出了当某些文件发生改变时，哪些目标文件必须重新建立。

默认情况下，make 命令会从当前的工作目录下加载名称为 Makefile 的文件，并执行指定的链编步骤(编译并链接)。

使用 make 的直接好处是，它只会重新编译那些发生了变化的文件，或者那些由于任何变化而受到影响的文件，而不会盲目地重新编译每一个源文件。图 1.1 中给出了建立 Qt 应用时涉及的那些步骤。

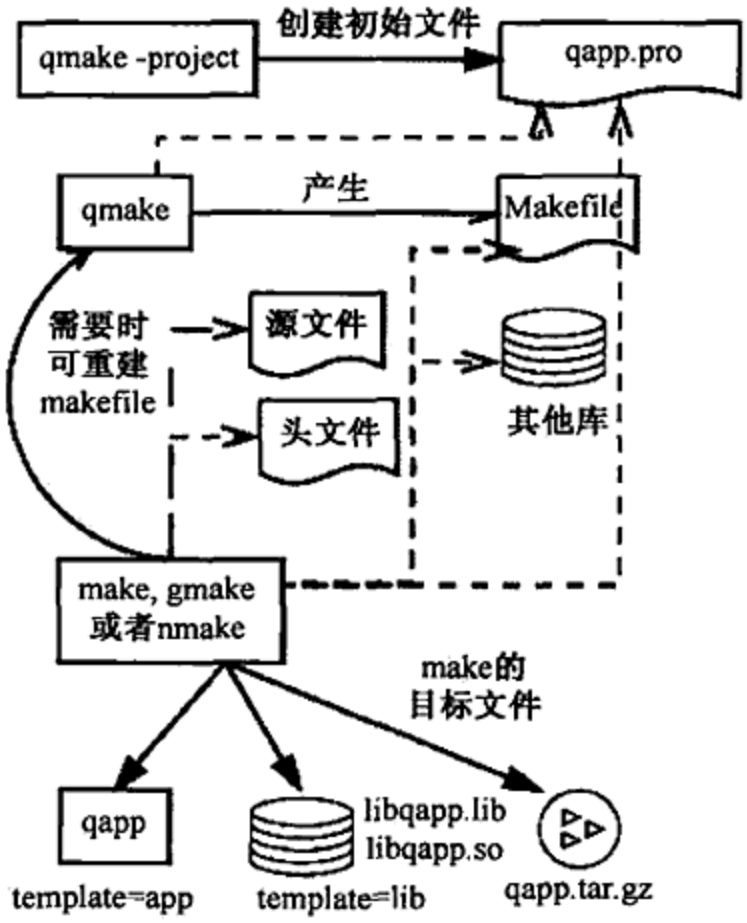


图 1.1 Qt 中 make 工具(qmake)建立程序的步骤

对于 Qt 程序，程序员不必编写 Makefile 文件。Qt 提供了一个 qmake 工具，它会产生 Makefile 文件。尽管如此，还是有必要运行 make 并理解它的输出。大多数 IDE 都是通过单击某个按钮的形式来运行 make 的(或者类似的方式)，并会显示或者过滤它的输出。

为了创建 Makefile，qmake 要求一个工程文件。工程文件描述了这个工程，其中列出了其他的所有文件以及全部的选项和文件位置，它们都是建立工程所需要的。工程文件比 Makefile 更为简单，用户很容易就能够创建它。尽管程序员能够创建工程文件，但也可以用 qmake -project 命令产生一个简单的工程文件。当执行这个命令时，qmake 会将当前工作目录下的全部源文件(*.cpp)作为 SOURCES 列出来，而将该目录下的全部头文件(*.h)作为 HEADERS 列出来。所得到的工程文件的名称，就是选项开关-o 后面给出的那个名称。如果没有提供这个选项和它的参数，则 qmake 会用当前工作目录的名称来命名工程文件以及最终的可执行文件。

创建完工程文件后，qmake 命令会根据工程文件创建 Makefile。然后，make 命令会尝试根据 Makefile 中的指令建立可执行文件[①]。可执行文件的名称由 TARGET 变量指定，其默认名称为工程的名称。

① 如果需要在 Mac OS X 系统中获得 Makefile，则应在工程文件中添加一行 CONFIG -= app_bundle，且输入的命令是 qmake -spec macx-g++，而不仅仅是 qmake。

下面的脚本展示了如何用 qmake 来建立示例 1.1 中讨论、编译并运行过的同一个小程序。此过程中执行完每一步之后所创建的文件，都用斜体显示出来。

```
src/early-examples/example0> ls
fac.cpp
src/early-examples/example0> qmake -project
src/early-examples/example0> ls
example0.pro fac.cpp
src/early-examples/example0> cat example0.pro

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += fac.cpp
src/early-examples/example0> qmake
src/early-examples/example0> ls
example0.pro fac.cpp Makefile
src/early-examples/example0> make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB
  -DQT_CORE_LIB -DQT_SHARED -I/usr/share/qt4/mkspecs/linux-g++ -I.
  -I/usr/include/qt4/QtCore -I/usr/include/qt4/QtCore
  -I/usr/include/qt4/QtGui -I/usr/include/qt4/QtGui -I/usr/include/qt4
  -I. -I. -I. -o fac.o fac.cpp
g++ -o example0 fac.o  -L/usr/lib -lQtGui -lQtCore -lpthread
src/early-examples/example0> ls
example0 example0.pro fac.cpp fac.o Makefile
src/early-examples/example0>
```

当运行 make 时，可以看到传递给编译器的参数。如果遇到错误，同样也能看到它。现在就可以运行这个应用了。

```
src/early-examples/example0> ./example0
Factorial of: 10
The Factorial of 10 is: 3628800
src/early-examples/example0> ./example0
Factorial of: -3
No negative values, please!
Factorial of: 0
The Factorial of 0 is: 1
src/early-examples/example0>
```

这里不是直接从命令行运行编译器指令，而是应使用 qmake 和 make，它们极大地简化了建立程序的过程，尤其是当工程涉及多个源文件、头文件和库时。

C.1 节中更为详细地探讨了 make 命令和 Makefile。

可以将工程文件理解成关于工程的一个地图，它包含建立应用或者库所需要的全部文件及其位置信息。与其他源代码文件一样，工程文件中的内容都能够被人和机器阅读并理解。在链编过程中(尤其是链接程序时)，如果遇到 not found 或者 undefined 消息，则应首先查看 .pro 文件。更多细节，推荐阅读 Qt 的 qmake 指南[①]。

① 参见 http://doc.qt.nokia.com/latest/qmake-manual.html。

如果要在工程中添加更多的源代码文件、头文件或者库模块，则必须编辑.pro 文件，将新项目添加到相应的 SOURCES, HEADERS 或者 LIBS 列表中。工程文件中采用的文档化标准，与 C++源代码中使用的文档化标准相同(但注释以#开头)。

示例 1.6 中给出的一些常见 qmake 变量设置，可以将它们用在工程中。将这个文件复制到一个方便的位置，并在工程文件中添加如下的一行

```
include (/path/to/wherever you put/common.pri)
```

就可以避免输入一些代码，节约时间。

示例 1.6 src/common.pri

```
# required if you want to see qDebug () messages

CONFIG += debug


# place auto-generated files in "invisible" subdirectories
win32 {
    MOC_DIR = _moc
    UI_DIR = _ui
    OBJECTS_DIR = _obj
} else {
    UI_DIR = .ui
    MOC_DIR = .moc
    OBJECTS_DIR = .obj
}

# rules below apply to TEMPLATE=app projects only:
app {
    # place executable in same folder:
    DESTDIR=$$OUT_PWD
    # don't place executables in an app bundle on mac os
    # this also permits console apps to work on the mac
    mac {
        CONFIG -= app_bundle
    }

    # Open a console for stdin, stdout, and stderr Windows:
    win32 {
        CONFIG += console
    }
}
```

注意：CONFIG += console(Windows 用户)

只有针对 Windows 平台才需要一行 CONFIG += console，其作用是告诉编译器要建立一个“控制台”应用，这种应用能够通过标准的输入/输出流与用户交互。如果使用的是 MS Dev studio，则这个设置与菜单选项 Project Properties->Configuration Properties->Linker->System->Subsystem->Console 等价。如果没有这个选项，就不会看到发送给 qDebug(), stdout 或者 stderr 的消息。

注意：CONFIG -= app_bundle（Mac OS X 用户）

对于 Mac 上的控制台程序，如果在工程文件中添加一行 CONFIG -= app_bundle，就可以阻止创建一个 app bundle，它会将可执行文件放入一个子目录下。这样做还允许用户与标准 I/O 的交互。然后，就可以使用命令行

```
qmake -spec macx-g++
```

创建一个 Makefile。这样，命令 make 就只会为应用创建一个可执行文件。

1.6.1　#include：查找头文件

使用#include 包含库头文件的三种方式是

```
#include <headerFile>
#include "headerFile"
#include "path/to/headerFile"
```

尖括号（<>）表示预处理器必须在所包含路径中列出的目录下（依次）查找这个头文件；用引号包含的文件名表明预处理器应当首先在包含文件所在目录下查找这个头文件；用引号包含的路径表明预处理器应当首先在该目录路径下查找这个头文件。路径信息既可以是绝对路径，也可以是相对路径（相对于包含文件的路径）。如果在指定的位置没有找到头文件，则会在包含路径中列出的目录下搜索它。

如果在包含路径下的多个目录中存在同一个文件名称的多个版本，则只要找到了一个，搜索就会停止。如果在搜索路径的全部目录中都没有找到文件，则编译器就会报告错误。

当安装编译器时，会告诉它到哪里去寻找 C++标准库中的头文件。对于其他的库，可以通过添加编译开关-I/path/to/headerfile，通知编译器扩展搜索路径。

如果使用 IDE，则存在一个 Project->Settings->Preprocessor 或者 Project->Options->Libraries 配置菜单，它可以指定更多的包含目录，这些配置会在链编程序时作为-I 开关传递给编译器。

下面很快就会看到，通过 qmake 可以向工程文件添加更多的 INCLUDEPATH += *dirName* 行。这些目录最终会在所产生的 Makefile 中成为 INCPATH 宏，然后会在链编程序时传递给编译器和预处理器。

qmake -r

有些工程文件是 SUBDIRS（子目录），这意味着它们会递归地沿着文件树同时运行 qmake 和 make。如果有来自于以前的 Qt 版本的旧 Makefile，并且希望立即强制重新生成全部的 Makefile，则可以调用 qmake -r，以沿着文件树向下递归地产生全部的 Makefile。

提示

通常而言，好的做法是在包含了 Qt 头文件之后再包含非 Qt 头文件。由于 Qt（为编译器和预处理器）定义了许多符号，这使得避免名称冲突变得更容易，也更容易找到文件。

关于预处理器及其用法的更多信息，请参见 C.2 节。

1.7 获得在线帮助

除了 Qt 在线文档[①]之外(它包含 API 文档的链接，还包含相关 Qt 产品的文章和文档的链接)，还有许多在线资源可以使用。以下是一些 Qt 的在线文档。

1. Qt 兴趣邮件列表[②]提供了一个开发者社区以及一些可搜索的存档文件。搜索要使用的某个类或者不能理解的某个错误消息，经常可以得到有用的结果。
2. Qt 开发者网络[③]包含一些关于 Qt 的技术性文章和教程。
3. QtCentre[④]是一个基于 Web 的在线社区，它关注用 Qt 进行 C++编程。它使全球的 Qt 用户都能彼此沟通并互相帮助。这个社区中有大量的论坛、工程和 FAQ。这个站点一直由专业 Qt 开发人员维护，他们经常参与论坛讨论、发表声明并给予有用的建议。
4. ics.com[⑤]是诺基亚的一个培训和咨询伙伴，它向需要 Qt 专业知识的人提供现场咨询和顾问服务。这有点做广告的意味，因为本书的作者之一是它的员工。
5. 如果遇到看似不好对付的错误，且错误消息不是很长的话，则可以在 Google 中尝试寻找解决之道。

1.8 字符串

在 C++中处理字符串数据有三种途径。

1. `const char*`，或者 C 语言风格的字符串，主要用于与 C 语言库的接口，或者极少的其他情况。它经常是应当避免的运行时错误的来源。
2. 来自于 C++标准库的 `string`，任何地方都可以使用这种类型。
3. `QString`，优于 STL 中的字符串类型，因为它有丰富的 API 且更容易使用。它的实现支持延迟写时复制(lazy copy-on-write)和隐式共享(implicit sharing)，后者将在 11.5 节中解释，所以函数能够接收 `QString` 类型的实参并返回 `QString` 类型的值，而不必每一次都为字符串分配内存并进行复制。此外，`QString` 还内置支持 Unicode 标准[⑥]，以方便程序的国际化。

示例 1.7 演示了 STL 字符串的基本用法。

示例 1.7 src/generic/stlstringdemo.cpp

```
#include <string>
#include <iostream>

int main() {
    using namespace std;
```

---

① Qt 在线文档的网址为 http://doc.qt.nokia.com/。
② Qt 邮件列表的网址为 http://lists.qt.nokia.com/。
③ Qt 开发者网络的网址为 http://developer.qt.nokia.com/wiki。
④ Qt 中心的网址为 http://www.qt.centre.org/。
⑤ 参见 http://www.ics.com。
⑥ Unicode 标准的网址为 http://www.unicode.org/standard/standard.html。

```
    string s1("This "), s2("is a "), s3("string.");
    s1 += s2;                                                          1
    string s4 = s1 + s3;
    cout << s4 << endl;
    string s5("The length of that string is: ");
    cout << s5 << s4.length() << " characters." << endl;
    cout << "Enter a sentence: " << endl;
    getline(cin, s2);                                                  2
    cout << "Here is your sentence: \n" << s2 << endl;
    cout << "The length of your sentence is: " << s2.length() << endl;
    return 0;
}
```


1 拼接。

2 s2 获得整个行的内容。

以下是编译并运行的结果。

```
src/generic> g++ -Wall stlstringdemo.cpp
src/generic> ./a.out
This is a string.
The length of that string is 17
Enter a sentence:
20 years hard labor
Here is your sentence:
20 years hard labor
The length of your sentence is: 20
src/generic>
```

注意观察语句

```
getline(cin, s2)
```

是如何从标准输入流提取字符串的。示例 1.8 中用 Qt 代替 STL 重写了同一个程序，它的输出与示例 1.7 相同。

示例 1.8　src/qstring/qstringdemo.cpp

```
#include <QString>
#include <QTextStream>

QTextStream cout(stdout);                                              1
QTextStream cin(stdin);

int main() {
    QString s1("This "), s2("is a "), s3("string.");
    s1 += s2;  // concatenation
    QString s4 = s1 + s3;
    cout << s4 << endl;
    cout << "The length of that string is " << s4.length() << endl;
    cout << "Enter a sentence with whitespaces: " << endl;
    s2 = cin.readLine();                                               2
    cout << "Here is your sentence: \n" << s2 << endl;
    cout << "The length of your sentence is: " << s2.length() << endl;
    return 0;
}
```

1 定义 QTextStream，它看起来类似 C++的标准输入/输出流。

2 不是 iostream，而是 QTextStream::readLine()。

注意，这一次使用了语句

```
s2 = cin.readLine()
```

来从标准输入流提取 QString。

1.9 流

流是用来读取或者写入的对象。标准库中定义了<iostream>，Qt 中为同样的功能定义了<QTextStream>。

前面已经看到 iostream 定义了三种全局流：

- cin——控制台输入(键盘)。
- cout——控制台输出(屏幕)。
- cerr——控制台错误(屏幕)。

<iostream>中还定义了操作算子，比如 flush 和 endl。操作算子被函数隐式调用，这些函数能够以各种方式改变流的状态。操作算子可以被添加到如下流中。

- 输出流，改变格式化输出数据的方式。
- 输入流，改变输入数据的解释方式。

示例 1.9 演示了控制台输出流上操作算子的用法。

示例 1.9 src/stdstreams/streamdemo.cpp

```
#include <iostream>

int main() {
    using namespace std;
    int num1(1234), num2(2345) ;
    cout << oct << num2 << '\t'
            << hex << num2 << '\t'
             << dec << num2
             << endl;
cout << (num1 < num2) << endl;
cout << boolalpha
             << (num1 < num2)
             << endl;
double dub(1357);
cout << dub << '\t'
        << showpos << dub << '\t'
        << showpoint << dub
        << endl;
dub = 1234.5678;
cout << dub << '\t'
        << fixed << dub << '\t'
        << scientific << dub << '\n'
        << noshowpos << dub
```

```
                << endl;
    }
```

输出如下所示。

```
4451    929     2345
1
true
1357    +1357   +1357.00
+1234.57        +1234.567800    +1.234568e+03
1.234568e+03
```

很容易用与 iostream 对应的同一个名称定义 QTextStream。由于控制台输入和输出主要用在调试过程，所以 Qt 提供了一个全局函数 qDebug()，用它可方便地将消息发送到控制台(不管控制台是什么)，且具有灵活的接口，见示例 1.10 的演示。

示例 1.10　src/qtstreams/qtstreamdemo.cpp

```
#include <QTextStream>
#include <QDebug>

QTextStream cin(stdin);
QTextStream cout(stdout);
QTextStream cerr(stderr);

int main() {
   int num1(1234), num2(2345) ;
    cout << oct << num2 << '\t'
            << hex << num2 << '\t'
             << dec << num2
             << endl;
    double dub(1357);
    cout << dub << '\t'
            << forcesign << dub << '\t'
            << forcepoint << dub
            << endl;
    dub = 1234.5678;
    cout << dub << '\t'
            << fixed << dub << '\t'
            << scientific << dub << '\n'
            << noforcesign << dub
            << endl;

    qDebug() << "Here is a debug message with " << dub << "in it." ;
    qDebug("Here is one with the number %d in it.", num1 );
}
```

输出如下所示。

```
4451    929     2345
1357    +1357   +1357.00
+1234.57        +1234.567800    +1.234568e+03
1.234568e+03
Here is a debug message with  1234.57 in it.

Here is one with the number 1234 in it.
```

符号 stdin, stdout 和 stderr 来自于 C 语言标准库。注意，QTextStream 还提供了一些操作算子，它们的拼写方式与示例 1.9 中 iostream 里的操作算子相同。

1.10 文件流

流被用来读取/写入文件、连接网络和处理字符串，它的一个有用的特性是易于从混合数据类型中得到字符串。示例 1.11 用字符和数字创建了一些字符串，并将它们写入文件中。

示例 1.11 src/stl/streams/streams.cpp

```
[ . . . . ]

#include <iostream>
#include <sstream>
#include <fstream>

int main() {
    using namespace std;
    ostringstream strbuf;

    int lucky = 7;
    float pi=3.14;
    double e=2.71;

cout << "An in-memory stream" << endl;
strbuf << "luckynumber: " << lucky << endl
       << "pi: " << pi << endl
       << "e: " << e << endl;

string strval = strbuf.str();                                    1
cout << strval;

ofstream outf;                                                   2
outf.open("mydata");                                             3
outf << strval ;
outf.close();
```

1 将字符串流转换成字符串。

2 输出文件流。

3 创建(或覆盖)磁盘文件，用于输出。

写入字符串之后，有多种途径来读取它。可以用简单的输入运算符从文件中读取，且由于记录之间存在空白，插入运算符可以如示例 1.12 所示。

示例 1.12 src/stl/streams/streams.cpp

```
[ . . . . ]

    cout << "Read data from the file - watch for errors." << endl;
    string newstr;
    ifstream inf;                                                1
    inf.open("mydata");
    if(inf) {   /*Make sure the file exists before attempting to read.*/
```

```
    int lucky2;
    inf >> newstr >> lucky2;
    if (lucky != lucky2)
      cerr << "ERROR! wrong " << newstr << lucky2  << endl;
    else
      cout << newstr << " OK" << endl;

    float pi2;
    inf >> newstr >> pi2;
    if (pi2 != pi)
      cerr << "ERROR! Wrong " << newstr << pi2 << endl;
    else
      cout << newstr << " OK" << endl;
double e2;
inf >> newstr >> e2;
if (e2 != e)
  cerr << "ERROR: Wrong " << newstr << e2 <<  endl;
else
  cout << newstr << " OK" << endl;
inf.close();
}
```

1 输入文件流。

可以逐行读取文件并将每一行当作一个字符串看待，如示例 1.13 所示。

示例 1.13　src/stl/streams/streams.cpp

```
[ . . . . ]

    cout << "Read from file line-by-line" << endl;
    inf.open("mydata");
    if(inf) {
      while (not inf.eof()) {
        getline(inf, newstr);
        cout << newstr << endl;
      }
      inf.close();
    }
    return 0;
}
```

示例 1.14 用 Qt 文件、字符串和流实现了同样的功能。这个示例中还使用了另外两个 Qt 类型：QString 和 QFile，前者具备功能强大而灵活的字符串表示能力，后者提供处理文件的接口。

示例 1.14　src/qtstreams/files/qdemo.cpp

```
#include <QTextStream>
#include <QString>
#include <QFile> .

QTextStream cout(stdout);
QTextStream cerr(stderr);
```

```
int main() {
  QString str, newstr;
  QTextStream strbuf(&str);

  int lucky = 7;
  float pi = 3.14;
  double e = 2.71;

  cout << "An in-memory stream" << endl;
  strbuf << "luckynumber: " << lucky << endl
         << "pi: " << pi << endl
         << "e: " << e << endl;

  cout << str;

  QFile data("mydata");
data.open(QIODevice::WriteOnly);
QTextStream out(&data);
out << str ;
data.close();

cout << "Read data from the file - watch for errors." << endl;
if(data.open(QIODevice::ReadOnly)) {
  QTextStream in(&data);
  int lucky2;
  in >> newstr >> lucky2;
  if (lucky != lucky2)
    cerr << "ERROR! wrong " << newstr << lucky2  << endl;
  else
    cout << newstr << " OK" << endl;

  float pi2;
  in >> newstr >> pi2;
  if (pi2 != pi)
    cerr << "ERROR! Wrong " << newstr << pi2 << endl;
  else
    cout << newstr << " OK" << endl;

  double e2;
  in >> newstr >> e2;
  if (e2 != e)
    cerr << "ERROR: Wrong " << newstr << e2 <<  endl;
  else
    cout << newstr << " OK" << endl;
  data.close();
}

cout << "Read from file line-by-line" << endl;
if(data.open(QIODevice::ReadOnly)) {
  QTextStream in(&data);
  while (not in.atEnd()) {
    newstr = in.readLine();
```

1

2

3

4

5

6

```
        cout << newstr << endl;
      }
      data.close();
    }
    return 0;
}
```

1 strbuf 用 str 的地址初始化。
2 创建(或覆盖)磁盘文件，用于输出。
3 输出文件流。
4 在尝试读之前应确保文件存在。
5 输入文件流。
6 输入文件流。

1.15.1 节中讨论了用来初始化 strbuf 的取址运算符。

1.10.1　练习：文件流

1. 运行示例 1.12 中的程序并完成如下工作。
   - 修改这个程序，在读取或者写入之前从用户处以 STL 字符串的形式取得文件名 fileName。需要使用函数 fileName.c_str()将字符串转换成 open()函数可接收的形式。
   - 修改这个程序，以确保用户指定的文件在被打开用于输出之前不存在(或者，如果存在的话可以覆盖它的内容)。
   - 如果在“watch for errors”部分用错误的变量类型(例如，用 int 而不是 float 或者 double)读取各个数字，会发生什么？请解释。
   - 如果在“watch for errors”部分只读取数字型变量且不使用 newstr 变量，会发生什么？请解释。
2. 对示例 1.14 中的程序完成与上一题相同的工作。

1.11　用于用户输入/输出的 Qt 对话框

示例 1.15 中给出了前面的第一个 C++例子的重写版本，它用标准的 Qt 对话框代替了标准的输入/输出，还用 QString 替换了标准库中的字符串。尽管这个示例中的代码中包含有一些还没有讲解过的内容，但依然将其在此列出，以便能够看到进行输入/输出的另一种途径——使用 Qt 图形用户界面(GUI)的便捷函数。

示例 1.15　src/early-examples/example1/fac1.cpp

```
#include <QtGui>

int main (int argc, char* argv[]) {                    1
    QApplication app(argc, argv);                      2
    QTextStream cout(stdout);                          3

    // Declarations of variables
```

```
int answer = 0;                                                    4

do {
    // local variables to the loop:
    int factArg = 0;
    int fact(1);
    factArg = QInputDialog::getInt(0, "Factorial Calculator",
        "Factorial of:", 1);                                       5
    cout << "User entered: " << factArg << endl;
    int i=2;
    while (i <= factArg) {
        fact = fact * i;
        ++i;
    }
    QString response = QString("The factorial of %1 is %2.\n%3")
        .arg(factArg).arg(fact)                                    6
        .arg("Do you want to compute another factorial?");         7
    answer = QMessageBox::question(0, "Play again?", response,
        QMessageBox::Yes | QMessageBox::No);                       8
  } while (answer == QMessageBox::Yes);
  return EXIT_SUCCESS;
}
```

1 main 函数的开始，它返回一个 int 值。
2 每一个 Qt GUI 应用的开始部分。
3 创建用于标准输出的 QTextStream。
4 必须在 do 循环之外定义，因为它用在 do 语句块之外的条件中。
5 弹出一个对话框，等待用户输入一个整数并返回它。
6 每一个%n 都会用一个 arg()值替换。
7 长语句可以跨越多行，只要它是在标记(token)边界折行的。
8 两个值的“位或”运算。

这个程序中使用了下面列出的 Qt 类型(类)。

- QApplication——需要存在于 Qt GUI 应用中的一个简单对象。
- QInputDialog——向用户询问问题。
- QMessageBox——向用户回送响应消息。
- QString——Unicode字符串类。这个示例中使用了功能强大的QString函数arg()，它能够在字符串中格式化参数值(%1, %2 等)。
- QTextStream——向文本文件输入/输出流。这个示例中定义了一个名称为 cout 的变量，其放置位置与 C++标准库中 iostream cout 的位置相同(stdout)。如果是从对话框和其他的窗件(widget)获得用户输入，则不需要 cin。

示例 1.15 中的代码包含将在下面的各个小节中讨论的内容。

- 命令行参数(argc 和 argv)——1.13.1 节。
- 类与作用域解析运算符(::)——2.2 节。
- 静态成员函数——2.9 节。

- 指针——1.15 节。
- 可选实参——5.2 节。

当运行这个应用时，首先会看到如图 1.2 所示的一个输入对话框。

位于对话框里面的输入窗件被称为 spin box，它被当作 QSpinBox 实现。它显示当前的值，且如果用户单击了显示区域右端的上箭头或者下箭头按钮的话，就会显示另一个可接受的值。用户也可以按键盘上的上/下箭头键来改变值。这个窗件的实现是灵活的，且容易定制它(例如，指定可接受的最大值和最小值)。用户输入一个数字并单击 OK 按钮之后，对话框会被一个弹出的 QMessageBox 代替，它显示计算结果，如图 1.3 所示。

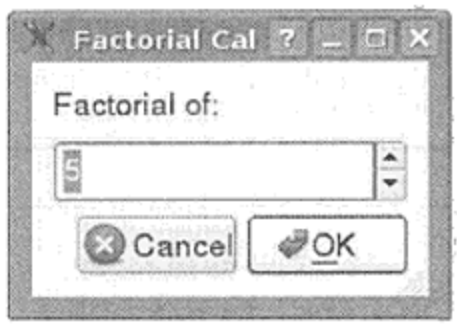


图 1.2　QInputDialog 用于输入一个整数

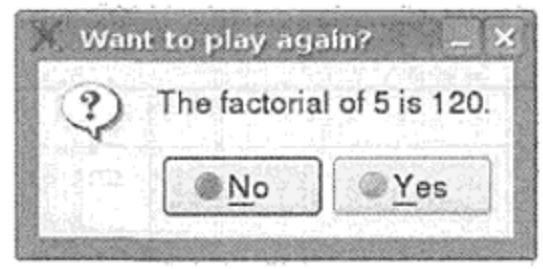


图 1.3　QMessageBox 问题

关于工程文件的更多说明

使用 Qt 类的任何应用都需要一个工程文件。前面说过，工程文件描述工程，其中列出了建立这个工程所需要的所有文件以及全部的选项和文件位置。由于这是一个简单的应用，所以工程文件也相当简单，如示例 1.16 所示。

示例 1.16　src/early-examples/example1/example1.pro

```
TEMPLATE = app
include (../../common.pri)
SOURCES += fac1.cpp
```

第一行，TEMPLATE＝app，表明 qmake 应以一个适合建立这个应用的模板化的 Makefile 开始。如果这个工程文件用于建立库，则应当使用语句 TEMPLATE = lib，以表明应使用 Makefile 库模板。第三种可能是源代码文件分布在多个子目录下，而每一个子目录下的文件都具有自己的工程文件。这种情况下，位于父目录的工程文件中应包含语句 TEMPLATE = subdirs，这会导致在父目录和每一个子目录下都生成 Makefile。

第二行包含可选的共同工程设置。最后，源代码文件在 SOURCES 中列出。

1.11.1　练习：用于用户输入/输出的 Qt 对话框

这一节的练习请参考示例 1.15。可以在 Qt 指南 API 文档(Qt Reference API Documentation)中找到大多数答案。

1. 应该如何修改程序，以便当用户单击 Cancel 按钮时不执行计算且退出循环?
2. 此时，程序不会检查用户是否输入了一个负数。应该如何修改程序，以确保程序绝对不会接收负数?

1.12 标志符，类型和常量

标志符是 C++程序中用于参数、变量、常量、类以及类型的名称。

标志符由字母、数字和下画线组成，但不能以数字开头。标志符不能是 C++的保留字。C++中的保留字清单在附录 A 中给出。C++标准中并没有指定标志符的长度限制，但有些 C++版本在区分不同的标志符时只会检查前 31 个字符。

常量(literal)是一个出现在程序中某个地方的值。由于每一个值都有类型，所以每一个常量也都有类型。每一种原始数据类型都可以具有常量，同样也可以有字符串常量。表 1.1 中给出了常量及其类型的一些示例。

表 1.1 常量示例

常量	含义
5	int 型常量
5u	u 或者 U 指定无符号 int 型常量
5L	整数之后的 l 或者 L 指定 long int 型常量
05	八进制 int 型常量
0x5	十六进制 int 型常量
true	bool 型常量
5.0F	f 或者 F 指定单精度浮点型常量
5.0	double(双精度)浮点型常量
5.0L	浮点数之后的 l 或者 L 指定 long double 型常量
'5'	char 型常量(ASCII 码 53)
"50"	包含字符'5', '0'和'\0'的 const char* 型常量
"any" "body"	"anybody"
'\a'	警告
'\\'	反斜线
'\b'	退格
'\r'	回车键
'\"'	单引号
'\" '	双引号
'\f'	进纸(下一页)
'\t'	制表符
'\n'	换行符
"\n"	换行后接一个空终止符(const char*)
'\0'	空字符
'\v'	水平制表符
"a string with newline\n"	另一个 const char*

C++中的字符串常量有些特殊，其形成历史可追溯到 C 语言。示例 1.17 中展示了有些字符应该如何在双引号字符串分隔符中进行转义。

示例 1.17 src/early-examples/literals/qliterals.cpp

```
#include <QTextStream>
#include <QString>

int main() {
    const char* charstr = "this is one very long string "
```

```
                " so I will continue it on the next line";
    QTextStream cout(stdout);
    QString str = charstr;
    cout << str << endl;
    cout << "\nA\tb\\c\'d\"" << endl;
    return 0;
}
```


1 C 风格的字符串可以被转换成 QString。

链编并运行这个程序可得到如下输出。

```
src/early-examples/literals> qmake -project
src/early-examples/literals> qmake
src/early-examples/literals> make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB
  -DQT_CORE_LIB -DQT_SHARED -I/usr/share/qt4/mkspecs/linux-g++ -I.
  -I/usr/include/qt4/QtCore -I/usr/include/qt4/QtCore
  -I/usr/include/qt4/QtGui -I/usr/include/qt4/QtGui
  -I/usr/include/qt4 -I. -I. -I. -I. -o qliterals.o qliterals.cpp
g++ -o literals qliterals.o  -L/usr/lib -lQtGui -lQtCore -lpthread
src/early-examples/literals> ./literals
```

其运行结果应与下面的类似。

```
this is one very long string so I will continue it on the next line

A       b\c'd"
```

注意，当处理字符串常量时，这个程序展示了如何控制输出行的长度。字符串可以在任何空白符处截断，并能用这种语法自动拼接起来。

1.12.1　练习：标志符，类型和常量

修改示例 1.17，用一条输出语句输出如下两种结果。

1. GNU stands for "GNU's Not UNIX".
2.
```
        Title 1         "Cat Clothing"
        Title 2         "Dog Dancing"
```

1.13　C++简单类型

C 和 C++中支持的简单类型在表 1.2 中列出。C 和 C++中还提供了一个特殊的符号 void，用于类型信息缺失的情况。

表 1.2　简单类型的层次

字节/字符类型	整型类型	浮点类型
bool	short int	float
char	unsigned short	double
signed char	int	long double
unsigned char	unsigned int	
wchar_t	anyType*	
	long int	
	unsigned long	

利用下面的关键字，C++中的简单类型可以变成另一种简单类型。

- short
- long
- signed
- unsigned[①]

C++编译器允许省略类型名称 short int, long int 以及 unsigned int 中的“int”字样。也可以省略大多数类型中的“signed”字样，因为这是默认设置。

C++对象的大小

C++中对象的大小是以 char 类型的大小来度量的。这种尺度下，char 的大小为 1。由于某种类型的值的范围与编译器平台的底层结构有关，所以 C++的 ANSI/ISO 标准没有指定表 1.2 中任何类型的大小。它只保证某种类型(例如，int)必须不能比表中出现在其上面的类型(例如，short)小。

有一个特殊的运算符 sizeof()，它返回存储某个表达式所要求的内存单元数(以 char 类型的大小为单位)。在大多数系统中，一个 char 类型值需用一个 8 位的字节保存。与大多数函数不同的是，sizeof()运算符的实参可以是值表达式，也可以是类型表达式。示例 1.18 中给出了 sizeof()的用法以及在 32 位 x86 系统中它返回的一些结果。

示例 1.18 src/early-examples/size/qsize.cpp

```
#include <QString>
#include <QTextStream>
#include <QChar>
#include <QDate>

int main() {
    QTextStream cout(stdout);
    char array1[34] = "This is a dreaded C array of char";
    char array2[] = "if not for main, we could avoid it entirely.";
    char* charp = array1;                                          1
    QString qstring = "This is a unicode QString. Much preferred." ;
    Q_ASSERT (sizeof(i) == sizeof(int));
    cout << "  c type sizes: \n";
    cout << "sizeof(char) = " << sizeof(char) << '\n';
    cout << "sizeof(wchar_t) = " << sizeof(wchar_t) << '\n';
    cout << "sizeof(int) = " << sizeof(int) << '\n';
    cout << "sizeof(long) = " << sizeof(long) << '\n';
    cout << "sizeof(float) = " << sizeof(float) << '\n';
    cout << "sizeof(double) = " << sizeof(double) << '\n';
    cout << "sizeof(double*) = " << sizeof(double*) << '\n';
    cout << "sizeof(array1) = " << sizeof(array1) << '\n';
    cout << "sizeof(array2) = " << sizeof(array2) << '\n';
    cout << "sizeof(char*) = " << sizeof(charp) << endl;
```

① 关于有符号整数类型与无符号整数类型的详细讨论，请参见 19.5 节。

```
    cout << "  qt type sizes: \n";
    cout << "sizeof(QString) = " << sizeof(QString) << endl;
    cout << "sizeof(qint32) = " << sizeof (qint32) << "\n";          2
    cout << "sizeof(qint64) = " << sizeof(qint64) << '\n';           3
    cout << "sizeof(QChar) = " << sizeof (QChar) << endl;            4
    cout << "sizeof(QDate) = " << sizeof(QDate) << endl;
    cout << "qstring.length() = " << qstring.length() << endl;       5
    return 0;
}
```

1 指向数组第一个元素的指针。

2 保证在所有平台上都是 32 位。

3 保证在所有平台上都是 64 位。

4 是 char 的两倍大小。

5 对于字节数，应确保考虑了 QChar 的大小。

输出如下所示。

```
(example run on 32-bit system)
sizeof(char) = 1
sizeof(wchar_t) = 4
sizeof(int) = 4
sizeof(long) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(double*) = 4
sizeof(array1) = 34
sizeof(array2) = 45
sizeof(char*) = 4
  qt type sizes:
sizeof(QString) = 4
sizeof(qint32) = 4
sizeof(qint64) = 8
sizeof(QChar) = 2
sizeof(QDate) = 4
qstring.length() = 42
```

注意，不管指针的类型如何，它们都具有相同的大小。

从输出可以看出 sizeof(qstring)只有 4 字节，但它是一个使用动态内存的复杂类，因此必须调用 length()来获得字符串中 QChar 的数量。由于 QChar 的大小为 char 的两倍，所以在计算内存中 QString 中实际大小时，必须将长度乘以 2。在运行时，QString 能够与具有同一个值的另一个字符串共享内存，所以在复制之后，动态内存不会排他性地“属于”某一个 QString 对象。

整型类型(bool, char, int)值的范围在标准头文件 limits.h 中定义。在*nix 系统的常规安装中，这个文件位于/usr/include 的一个子目录下。

注意：基本类型化变量的初始化

如果变量为基本类型，则必须初始化它。启动程序时，未初始化的基本类型变量具有未定义的值。当调试时，它们可能具有初始值 0，而在其他环境下运行时，它们可能具有垃圾值。

1.13.1 main与命令行参数

main()是当程序启动时调用的函数。如果希望程序接收命令行参数，则必须用main的完全参数表定义它。

C和C++中允许main()函数中的参数具有一定的灵活性，所以可以用多种方式定义它，例如

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

这些形式都是有效的，它们都定义了两个参数：int型参数包含了命令行参数的个数，而C风格的字符串数组包含了实际的参数。这些参数从父进程[①]传递给程序，包含了用于重构命令行参数所需的足够信息。示例1.19是一个简单的main程序，它输出命令行参数。

示例1.19 src/clargs/clargs-iostream/clargs.cpp

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

int main (int argc, char* argv[]) {
    using namespace std;
    cout << "argc = " << argc << endl;
    for (int i = 0; i < argc; ++i) {
        cout << "argv# " << i << " is " << argv[i] << endl;
    }
    int num = atoi(argv[argc - 1]);
    cout << num * 2 << endl;
    return 0;
}
```

1 用于atoi()函数。

argv是argument vector(参数数组)的缩写，它是包含全部命令行字符串的一个数组(参见21.4节)。换句话说，由于每一个命令行字符串本身就是一个char数组，所以argv就是char数组的一个数组，而argc是argv中char数组的个数，argc是argument count(参数个数)的缩写。

main()需要向父进程返回一个int值。在*nix系统中，进程的返回值被称为它的“退出状态”(exit status)。父进程可以使用退出状态来决定下一步做什么[②]。如果一切正常，则返回值应为0；如果执行过程中出现错误，则应返回一个非0的错误码。返回结果的方式有多种。

- 用语句

  ```
  return 0;
  ```

  返回，正如本书中大多数示例中那样。
- 如果main()函数没有返回语句，则当遇到它的结束大括号(})时，默认会返回0。示例1.20中就是这种情况。

---

① 即调用main()的进程(例如，命令行shell，窗口管理器，等等)。

② 17.1节中会探讨控制各种不同进程的程序。

- 为了强化可移植性，C++程序应严格遵循 C++标准，在其中包含语句

```
#include <cstdlib>
```

然后以语句

```
return EXIT_SUCCESS;
```

终止 main() 函数，正如 1.11 节中那样。

注意

不要将这里对于 0 的解释与布尔值 false 相混淆，false 也等于 0。

如果用命令行参数运行这个程序，则可以看到类似下面的输出。

```
clargs> ./clargs spam eggs "space wars" 123
argc = 5
argv# 0 is ./clargs
argv# 1 is spam
argv# 2 is eggs
argv# 3 is space wars
argv# 4 is 123
246
```

第一个参数是可执行文件的名称，其他参数都是从命令行中以字符串的形式取出来的，字符串之间以空格或者跳表键分开。如果要将一个包含空格的字符串当作一个参数，则必须将其放入引号中。

最后一个参数看起来似乎是数字 123，实际上它是一个字符串。如果用这个“数字”进行计算，则需要用合适的函数将字符串"123"转换成数字 123。

在 Qt 中处理命令行参数

示例 1.20 是示例 1.19 的重写，它用 Qt 类型设置并读取命令行参数，从而避免了数组的使用。它们的输出相同。

示例 1.20　src/clargs/qt/clargs.cpp

```
#include <QTextStream>
#include <QCoreApplication>
#include <QStringList>

int main (int argc, char* argv[]) {
    QCoreApplication app(argc, argv);
    QTextStream cout(stdout);
    QStringList arglst = app.arguments();
    cout << "argc = " << argc << endl;
    for (int i=0; i<arglst.size(); ++i) {
        cout << QString("argv#%1 is %2").arg(i).arg(arglst[i]) << endl;
    }
    int num = arglst[argc - 1].toInt();
    cout << num * 2 << endl;
}
```

采用 Qt 类型的大多数应用，都应尽可能早地在 main() 函数中定义 QCoreApplication 类

型或者 QApplication 类型的一个对象[①]。8.3 节中给出了这样做的理由，并分析了这两种类型的区别。

QCoreApplication app 是用参数 count 和 vector 初始化的。app 会将 argv 中的 char 数组自动转换成 QString，并将这些字符串保存在 QStringList 中(参见 4.2.1 节)。然后，就可以对 app 调用 arguments() 函数，访问并处理这些命令行参数。这样使用高级数据结构，就无须用到 char 数组，从而减少了内存出错的风险[②]。

应注意 QString 函数 toInt() 的用法。

1.13.2 算术运算

每一种编程语言都支持基本的算术运算和算术表达式操作。对于每一种原始数字类型，C++都提供了 4 种基本的算术运算符：

- 加(+)
- 减(–)
- 乘(*)
- 除(/)

这些运算符号用于构成标准的中缀语法中的表达式，也就是数学课中所学的那种语法。

C++中提供了几种快捷运算符，它将每一个基本运算符与赋值运算符(=)结合在一起。例如，可以将

```
x += y;
```

写成

```
x = x + y;
```

C++还为整型类型提供了一元增 1 运算符(++)和一元减 1 运算符(--)。如果将这种运算符置于变量的左边(前缀)，则它的运算会在计算表达式的其余部分之前进行；如果将这种运算符置于变量的右边(后缀)，则它的运算会在计算表达式的其余部分之后进行。与编译后缀运算符时相比，编译前缀运算符时当今的编译器通常会产生更短的机器码，所以如果在计算表达式的值之前或者之后再对变量进行运算不会影响结果的话，推荐采用前缀运算符而不是后缀运算符。示例 1.21 至示例 1.25 演示了各种 C++算术运算符的用法。

示例 1.21 src/arithmetic/arithmetic.cpp

```
[ . . . . ]

#include <QTextStream>

int main() {
    QTextStream cout(stdout);
    double x(1.23), y(4.56), z(7.89) ;
    int i(2), j(5), k(7);
    x += y ;
    z *= x ;
```

---

① 只使用诸如 QString，QStringList 和 QTextStream 类型的应用，并不需要 QCoreApplication。

② 可先阅读 1.15 节，然后阅读第 21 章。针对这个主题，Bjarne Stroustrup 也给出了一些有益的建议。

```
    cout << "x = " << x << "\tz = " << z
         << "\nx - z = " << x - z << endl ;
```

整除是作为一种特殊情况处理的。将一个 int 类型的数与另一个 int 类型的数相除，其结果是一个 int 类型的商。运算符 / 用于获取整除的商；运算符%(称为求余运算符)用于获取余数。示例 1.22 演示了这两个整数算术运算符的用法。

示例 1.22　src/arithmetic/arithmetic.cpp

```
[ . . . . ]

    cout << "k / i = " << k / i
         << "\tk % j = " << k % j << endl ;
    cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;
    cout << "++k / i = " << ++k / i << endl;
    cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;
    cout << "i * j-- = " << i * j-- << endl;
    cout << "i = " << i << "\tj = " << j << "\tk = " << k << endl;
```

混合表达式(任何有效的表达式)所得到的结果类型，通常是参数类型中数值范围最大的那一个[①]。从示例 1.23 中可看出，将 int 值与 double 值相除的结果是一个 double 值。

示例 1.23　src/arithmetic/arithmetic.cpp

```
[ . . . . ]

    cout << "z / j = " << z / j << endl ;
```

关于各种类型间的转换，将在第 19 章中详细探讨。

C++中还提供了全套的布尔运算符，用于比较各种数字表达式。这些运算符都返回一个布尔值，其结果可以是 false 或者 true。这些运算符如下。

- 小于(<)
- 小于或者等于(<=)
- 等于(==)
- 不等于(!=)
- 大于(>)
- 大于或者等于(>=)

利用一元非运算符(!)，可以将布尔表达式的结果取反。利用“与”运算符和“或”运算符，可以将两个或者多个布尔表达式组合成一个表达式。这两个运算符如下。

- 与(&&)
- 或(||)

示例 1.24　src/arithmetic/arithmetic.cpp

```
[ . . . . ]

    /*   if () ... else   approach */
    if (x * j <= z)
```

① 各种数值类型的表示范围请参见表 1.2。

```
        cout << x * j << " <= " << z << endl ;
    else
        cout << x * j << " > " << z << endl;
```

除了二元布尔运算符之外，示例 1.25 中还用到了条件表达式

(boolExpr) ? expr1 : expr2

如果 *boolExpr* 为 `true`，则返回 *expr1*，否则返回 *expr2*。

示例 1.25　src/arithmetic/arithmetic.cpp

```
[ . . . . ]

    /* conditional operator approach */
    cout << x * k
            <<( (x * k < y * j) ? " < " : " >= ")
            << y * j << endl;
    return 0;
}
```

示例 1.26 是这个程序的输出结果。

示例 1.26　src/arithmetic/arithmetic.cpp

```
[ . . . . ]
```

输出如下所示。

```
x = 5.79        z = 45.6831
x - z = -39.8931
k / i = 3       k % j = 2
i = 2   j = 5   k = 7
++k / i = 4
i = 2   j = 5   k = 8
i * j-- = 10
i = 2   j = 4   k = 8
z / j = 11.4208
23.16 <= 45.6831
46.32 >= 18.24
```

1.13.3　练习：C++简单类型

1. 编写一个小程序，要求用户输入摄氏温度值，然后计算对应的华氏温度值。应使用 QInputDialog 来获得用户输入的值，用 QMessageBox 显示结果。然后，以表格形式向控制台输出 0～100℃对应的华氏温度值，以 5℃递增。
2. 如果程序中包含语句#include <cstdlib>，则可以使用 rand() 函数，它会产生 0～RAND_MAX 范围内均匀分布的一个伪随机 long int 值序列，其产生方法是根据前一个值来计算序列中的下一个值。函数调用

   ```
   srand(unsigned int seed)
   ```

   将 rand() 产生的值序列的第一个值设置成 seed。编写一个小程序测试这个函数。要求用户从键盘输入 seed 的值，然后产生一个伪随机数序列。
3. 如果希望每次运行程序时能得到不同的结果，可以使用 srand(time(0)) 作为提供给 rand() 函数的 seed 值。因为函数调用 time(0) 返回的是自某个初始点开始的秒数，所以每次运行程序时，seed 值都会不同，这样就可以使程序员编写出其行为模式无法预测的程序。

当执行程序时，多次调用 srand()并无特别的优势。事实上，由于计算机在 1 s 内能够做许多事情，所以重复地调用 srand()甚至会明显减少程序的随机性。

编写一个程序，使用户与计算机模拟掷骰子的游戏。游戏规则如下。

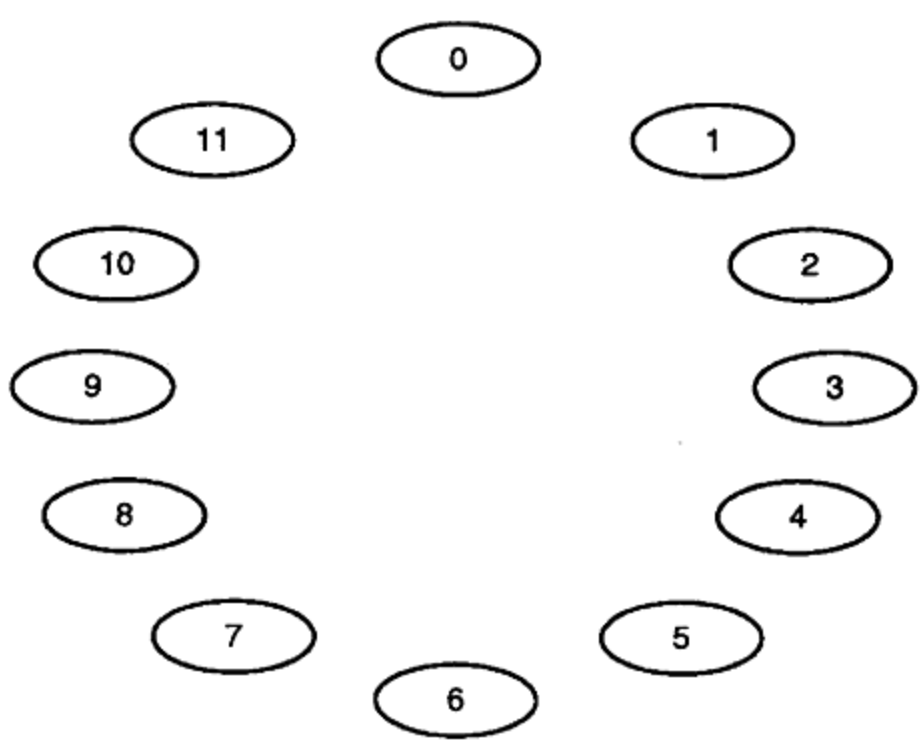
- 游戏不断地掷一对骰子。
- 每枚骰子都有六个面，以 1～6 编号。
- 掷一次的结果就是两个朝上的面的点数之和。
- 第一次掷的结果是玩家的点数。
- 如果点数为 7 或 11，则玩家自动获胜。
- 如果点数为 2，则玩家自动输。
- 否则，玩家需继续掷骰子，直到分出胜负。胜的情况是玩家再次掷出了与第一次相同的点数，输的情况是掷出了点数为 7 或者 11。

4. 编写一个程序，从用户(客户)处接收两个值：总的采购金额和用户支付的金额。
   将这两个值保存在 double 类型的变量中。计算并显示需向用户找回的钱。将找回的钱数以美元纸币 10 元、5 元、1 元和硬币 0.25 元、0.1 元、0.05 元、0.01 元表示(这个输出可能要发送到某台机器，由它自动吐出这样多的钱)。
   例如，如果总采购金额为 73.82 美元，用户支付的金额为 100 美元纸币，则找回的钱应为纸币 10 美元两张、5 美元一张、1 美元一张，硬币 0.1 元一个、0.05 元一个、0.01 元三个，没有 0.25 元的硬币。

提示

将找回的钱数转换成以美分(0.01 元)为单位的数量，以 int 型数值保存。然后，使用整除运算符。

5. 编写一个程序，让用户与计算机玩下面这个游戏。
   这个游戏在一块虚拟的游戏板上玩，游戏板被带编号的圆圈分成不同的场地[①]。例如，有 12 块场地的游戏板可能像下面这样。



① 这个练习不需要图形。此处所说的游戏板，是为了帮助理解游戏的玩法而杜撰的。

- 当程序启动时，要求用户指定游戏板上应该有多少块场地。最少必须有 5 个。
- 然后，计算机会随机地选择两块场地(编号为 0 和 1 的场地除外)。其中的一块场地被称为“目标”(The Goal)，另一块被称为“陷阱”(The Pit)。
- 计算机告知这两个特殊的场地。
- 然后，游戏从 0 号场地开始：玩家依次掷一对骰子，并向前移动 *N* 步到达另一块场地(*N* 是骰子掷出的点数和)。例如，依然以 12 块场地的游戏板为例，如果玩家第一轮时掷出点数 10(6 + 4)，第二轮时掷出点数 7(3 + 4)，则会到达 5 号场地。下一轮时将从这块场地开始。
- 用户和计算机依次轮流掷骰子并在场地间移动。每一次移动都显示在屏幕上(文本方式而非图形方式，例如，显示文字“现在位于 3 号场地”)。
- 如果有人位于“目标”场地(获胜)或者“陷阱”场地(失败)，则游戏结束。
- 计算机给出每一轮游戏的结果，并记录每一位玩家的输赢情况。
- 用户可以选择一个人玩(掷完骰子后必须按回车键)，或者继续与计算机玩(骰子会自动掷出，游戏会持续到有人胜出)。
- 玩完一次后，用户可以选择退出或者再次玩。

提示

在尝试解决这个问题之前先解决问题 2。可能还需要借鉴问题 3 的开头部分以及来自于该问题描述的一些思想。

1.14 const 关键字

把某个实体声明为 const 后，编译器会将其视为只读。正如很快就可以看到的情况一样，关键字 const 在程序中使用得极其频繁。

因为不能进行赋值操作，const 对象必须进行恰当的初始化。例如

```
const int x = 33;
const int v[] = {3, 6, x, 2 * x};    // a const array
```

针对上面的声明，下面的操作都是错误的。

```
++x ;              // error
v[2] = 44;         // error
```

对于整型和其他一些简单类型，无须针对 const 变量分配存储空间，除非要取它的地址。更一般的情况是，如果 const 变量的初始化器(initializer)是一个能够在编译时进行求值的常量表达式，且编译器知道它的每一个使用之处，则没有必要为它分配存储空间。

一种好的编程实践是在代码中使用 const 实体而不是嵌入数字型常量(有时称它们为“幻数”)。如果以后需要改变它的值时，就可以获得这种灵活性。一般而言，将常量“孤立”出来，可提高程序的可维护性。例如，不应当这样编写代码

```
for(i = 0; i < 327; ++i) {
    ...
}
```

而是应当将它写成

```
// const declaration section of your code
const int SIZE = 327;
...
for(i = 0; i < SIZE; ++i) {
    ...
}
```


注意

在某些 C/C++程序中，可能会看到将常量定义成预处理器宏的情况，例如

```
#define STRSIZE 80
[...]
char str[STRSIZE];
```

在编译器看到预处理器宏之前，它会被替换。使用宏而不是常量，意味着编译器所进行的类型检查将无法达到编译器对适当的 const 表达式所能够进行的类型检查的水平。对定义常量值来说，C++程序更倾向于使用 const 表达式而不是宏。预处理器的其他用法请参见 C.2 节。

1.15　指针与内存访问

与许多其他编程语言不同的是，C 和 C++中允许通过指针直接访问内存。这一节将讲解基本的指针操作，介绍指针修饰符以及动态内存的用法。初看起来，指针似乎很复杂。第 21 章将详细讨论指针的使用以及误用情况。

1.15.1　一元运算符&与*

对象(在最通常的意义下)就是一片能够容纳数据的内存区域。变量是一个能够被编译器识别的具有名称的对象。变量的名称可以当作对象自己进行使用。例如，如果有语句

```
int  x  =  5;
```

就可以使用 x 来代表值为 5 的整型对象，也可以通过名称 x 来直接操作此整型对象。例如

```
++x ;  // symbol x now refers to an integer with value 6
```

每一个对象都有一个内存地址(数据开始的位置)。一元运算符&也被称为取址运算符，当将其用于对象时，返回的是该对象的内存地址。例如，&x 返回对象 x 的内存地址。

保存某个对象的内存地址的另一个对象，被称为指针(pointer)，指针会指向这个内存地址处的对象。例如

```
int*  y  =  &x ;
```

在这个示例中，y 指向整数 x，类型名称 int 后面的星号表明 y 是一个 int 类型的指针。

此处的 int 型指针 y 被初始化成 int 变量 x 的地址。指针强大的功能之一是：某种类型的指针可以拥有一个不同(但相关)类型的对象的地址，当然，这要遵守后面将给出的一些规则。

在 C 程序中，经常使用 NULL 这个宏来代表零(0)，它是一个能够被合法地赋予指针的特殊值，通常在初始化时或者删除指针后将它赋给指针变量。0 不是某个对象的地址，存储了 0 的指针被称为空指针(null pointer)。Stroustrup 建议在 C++程序中使用 0 而不是 NULL 宏。

指向简单类型变量的指针与指向大型、复杂对象的指针占用相同的内存空间，其大小通常等于该机器上 sizeof(int)的求值结果。

一元运算符*被称为解引用运算符(dereference operator)，当应用到非空指针时，它返回指针指向的地址处的对象。

警告

符号*可以有两种与指针相关的使用方式:

- 在指针变量定义中用作类型修饰符。
- 作为解引用运算符。

示例 1.27 src/pointers/pointerdemo/pointerdemo.cpp

```
#include <QTextStream>

int main() {
    QTextStream cout(stdout);
    int x = 4;
    int* px = 0 ;                                          1
    px = &x;
    cout << "x = " << x
         << " *px = " << *px                               2
         << " px = " << px
         << " &px = " << &px << endl;
    x = x + 1;
    cout << "x = " << x
         << " *px = " << *px
         << " px = " << px << endl;
    *px = *px + 1;
    cout << "x = " << x
         << " *px = " << *px
         << " px = " << px << endl;
    return 0;
}
```

1 类型修饰符。

2 一元解引用运算符。

输出如下所示。

```
OOP> ./pointerdemo
x = 4 *px = 4 px = 0xbffff514 &px = 0xbffff510
x = 5 *px = 5 px = 0xbffff514
x = 6 *px = 6 px = 0xbffff514
OOP>
```

当在不同时刻或者不同的机器上运行示例 1.27 时，这些内存地址会有所不同。

变量 x 直接访问它的数据，而变量 px 间接访问同样的数据。这也就是为什么经常使用“间接”这个词语来描述通过指针访问数据的过程。变量 x 和 px 之间的关系见图 1.4。

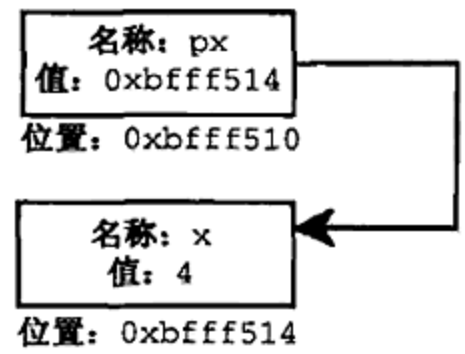


图 1.4 指针演示

由于空白符会被编译器忽略，所以空白符出现的位置既可以有助于理解程序，也可能导致混淆。为了提高包含指针声明的 C++代码的可读性和可维护性，推荐遵循下面的规范。

- 每一个指针都有自己的声明语句。
- 星号应紧挨在类型名称的右边放置。

```
T* ptr;
```

这个声明是无歧义的，ptr 就是一个类型为 T 的指针。

1.15.2　运算符 new 和 delete

C++具有一种在运行时动态分配内存的机制，这意味着即使程序员不事先估计程序的内存需求量，也可以保证它所需要的最大内存量会得到满足。运行时动态内存分配是帮助程序员构建高效而灵活的程序的强大工具。

new 运算符从堆(heap，也称为动态内存)中分配内存空间，并且返回指向新分配的对象的指针。如果由于某种原因无法完成内存分配，就会抛出一个异常[①]。

delete 运算符的作用是释放动态分配的内存并将其返回给堆。delete 操作只能用于由 new 运算符返回的指针或者空指针。不再需要的堆内存应当进行释放，以便再次使用。如果不这样做，则可能导致内存泄漏。

通常而言，调用 new 运算符的代码应当做好文档化工作，或者至少在位置上靠近释放该内存的代码。这样做的目的是使内存管理代码尽可能地简单和可靠。

对空指针、被删除的指针或者未初始化的指针进行解引用操作，会导致运行时错误，通常为段错误(segmentation fault)，在 Windows 中则为通用保护错误(GPF)。程序员有责任确保不会出现这种情况。后面将探讨如何确保不会出现这种错误的技术。

内存管理使程序员获得了强大的能力，但是，“权力越大，责任越大”。

示例 1.28 中的代码片段演示了 new 和 delete 运算符的用法。

示例 1.28　src/pointers/newdelete/ndsyntax.cpp

```
#include <iostream>
using namespace std;

int main() {
 int* ip = 0;                                          1
 delete ip;                                            2
 if(ip) cout << "non-null" << endl;
 else cout << "null" << endl;
 ip = new int;                                         3
 int* jp = new int(13);                                4
 //[...]
 delete ip;                                            5
 delete jp;
}
```

1 空指针。

2 完全没有影响——ip 依然为空。

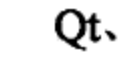
① 本书的配套网站上探讨了这种情况。

3 给 int 变量分配空间。

4 分配并初始化。

5 如果没有这条语句，就会发生内存泄漏。

实际上，程序中的空指针相当有用。尽管对空指针进行解引用操作是一个致命的运行时错误，但检查指针是否为空是完全合法的。例如，函数的一种常见操作是搜索指针容器中的某一项，如果搜索成功，则返回一个指向该项的指针。如果搜索不成功，则依然必须返回一个指针。这时，返回空指针就是一个不错的选择。当然，在解引用这样的指针之前，必须仔细地检查函数的返回值，以确保它非空。解引用指针后，能够对它进行的唯一合法操作是赋值。如果没有合理的其他选择，推荐立即对解引用的指针赋值成 0。



Qt、标准库以及 Boost.org 中都提供了各种类和函数，以帮助管理和清理堆内存。除容器类之外，每一个库都有一个或多个智能指针类。智能指针是一个对象，它用来保存和管理指向堆对象的指针，其行为与常规的指针非常类似，只是它会在合适的时刻自动删除堆对象。这个类在 Qt 中对应的是 QPointer，在标准库中是 std::auto_ptr，在 Boost 中是 shared_ptr。使用其中的任何一个类，都可以使 C++的内存管理变得更轻松且更安全。

1.15.3 练习：指针与内存访问

1. 预测示例 1.29 的输出。

示例 1.29 src/pointers/newdelete1/newdelete1.cpp

```
#include <QTextStream>

int main() {
    QTextStream cout(stdout);
    const char tab = '\t';
    int n = 13;
    int* ip = new int(n + 3);
    double d = 3.14;
    double* dp = new double(d + 2.3);
    char c = 'K';
    char* cp = new char(c + 5);
    cout << *ip << tab << *dp << tab << *cp << endl;
    int* ip2 = ip;
    cout << ip << tab << ip2 << endl;
    *ip2 += 6;
    cout << *ip << endl;
    delete ip;
    cout << *ip2 << endl;
    cout << ip << tab << ip2 << endl;
    return 0;
}
```

编译并运行这些代码。解释输出的结果，尤其是最后两行。

2. 修改示例 1.28，使用 jp 指向的值进行算术运算。将结果赋给 ip 指向的内存位置，然后输出结果。在程序的不同位置输出结果，分析编译器和运行时系统对输出语句的不同位置有何反应。
3. 阅读第 21 章，体验这一章中的代码示例的运行情况。

1.16　引用变量

前面说过，对象(在最通常的意义下)就是一片能够容纳数据的内存区域。左值(lvalue)是一个指向对象的表达式。左值的例子有变量、数组元素、解引用的指针等。本质而言，左值就是具有内存地址并且可以拥有名称的任何元素。注意，临时表达式或者常量表达式都不是左值，比如 i+1 或者 3。

在 C++中，引用(reference)提供了一种给左值赋予一个别名的机制。对于避免费时或者不必要的复制，引用是非常有用的，例如向函数传送一个非常大的对象作为参数。引用必须在声明时进行初始化，且其初始化器也必须是一个左值。

为了创建一个 SomeType 类型对象的引用，必须声明一个 SomeType&类型的变量。例如

```
int n;
int& rn = n;
```

int 后面的符号&表示 rn 是一个 int 类型的引用。引用变量 rn 是实际变量 n 的别名。注意，此处的&被用作声明中的类型修饰符，而不是一个针对左值进行操作的运算符。

在整个生命周期中，引用变量都可以作为初始化该变量的实际左值的别名，但这种关联可以被撤销或者转移。例如

```
int a = 10, b = 20;
int& ra = a;                    // ra is an alias for a
ra = b;                         // this causes a to be assigned the value 20

const int c = 45;               // c is a constant: its value is read-only.
const int& rc = c;              // legal but probably not very useful.
rc = 10;                        // compiler error - const data may not be changed.
```

一定要注意，本节中符号&的使用与前一节中在指针上的使用有所不同，不要将它们混淆。为了避免混淆，需要记住如下两个事实。

1. 取址运算符应用到一个对象上，返回其地址。因此，它只在赋值语句的右侧出现，或者在指针变量初始化表达式中出现。
2. 当与引用相关时，符号&仅仅用在引用的声明中。因此，它只出现在类型名称与所声明的引用名称的中间。

注意

对于引用声明，推荐将符号&紧挨在类型名称的右边放置：Type& ref(initLval);

1.17 const*与*const

假设有一个 ptr 指针，它保存变量 vbl 的地址

```
Type* ptr = &vbl;
```

当使用指针时，会涉及两个对象：指针本身以及它所指向的对象。这意味着对 const 可以采用如下三种保护层次。

1. 如果希望确保 ptr 无法指向任何其他的内存位置(也就是说，让它无法保存另一个内存地址)，则有两种编写语句的方式

```
Type* const ptr = &vbl;
Type* const ptr(&vbl);
```

指针是常量(const)，但地址对象可以变化。

2. 如果希望确保无法通过解引用 ptr 来改变 vbl 的值，则有两种编写语句的方式

```
const Type* ptr = &vbl;
const Type* ptr(&vbl);
```

这时，地址对象是常量，而指针不是。

3. 如果希望二者都受保护(无法改变)，则可以编写语句

```
const Type* const ptr = &vbl;
const Type* const ptr(&vbl);
```

记住它们的一种好办法是：把下面的定义从右向左(从定义的变量开始)读。

```
const char* x = &p;          /* x is a pointer to const char  */
char* const y = &q;          /* y is a const pointer to char  */
const char* const z = &r;    /* z is a const pointer to a const char  */
```

volatile

volatile 是能够修改变量和指针定义的另一个关键字。使用 const 的地方同样能够使用 volatile。可以将 volatile 理解成 const 的反义。用它标记的对象在任何时候都可以被修改，这可以发生在另一个程序或者另一个线程中。它是对编译器的一个提示，访问它时不应当进行优化。

volatile 可以用于变量，但更常见的情形是用于指针。与 const 一样，volatile 可以用于指针或者被编址的内存。为了将常规指针声明成 volatile 内存，应使用形式：

```
volatile char* vcharptr;
```

为了将 volatile 指针声明成常规内存，应使用形式：

```
char* volatile vptrchar;
```

示例 1.30 中演示了这两种保护类型。

示例 1.30 src/constptr/constptr.cpp

```
#include <QTextStream>

int main() {
    QTextStream cout(stdout);
```

```
int m1(11), m2(13);
const int* n1(&m1);
int* const n2(&m2);
// First snapshot
cout << "n1 = " << n1 << '\t' << *n1 << '\n'
     << "n2 = " << n2 << '\t' << *n2 << endl;
n1 = &m2;
//*n1 = 15;                                    1
m1 = 17;                                       2
//n2 = &m1;                                    3
*n2 = 16;                                      4
// Second snapshot
cout << "n1 = " << n1 << '\t' << *n1 << '\n'
     << "n2 = " << n2 << '\t' << *n2 << endl;
return 0;
}
```

1 错误：对只读位置赋值是不允许的。

2 m2 是一个普通的 int 型变量，可以赋值。

3 错误：对只读变量 n2 赋值是不允许的。

4 可以改变目标变量的值。

输出如下所示。

```
src/constptr> ./constptr
n1 = 0xbffff504 11
n2 = 0xbffff500 13
n1 = 0xbffff500 16
n2 = 0xbffff500 16
src/constptr>
```

图 1.5 给出了示例 1.29 中注释点处的两个内存快照，有助于理解程序运行时发生的事情。注意，这个程序会导致内存泄漏。

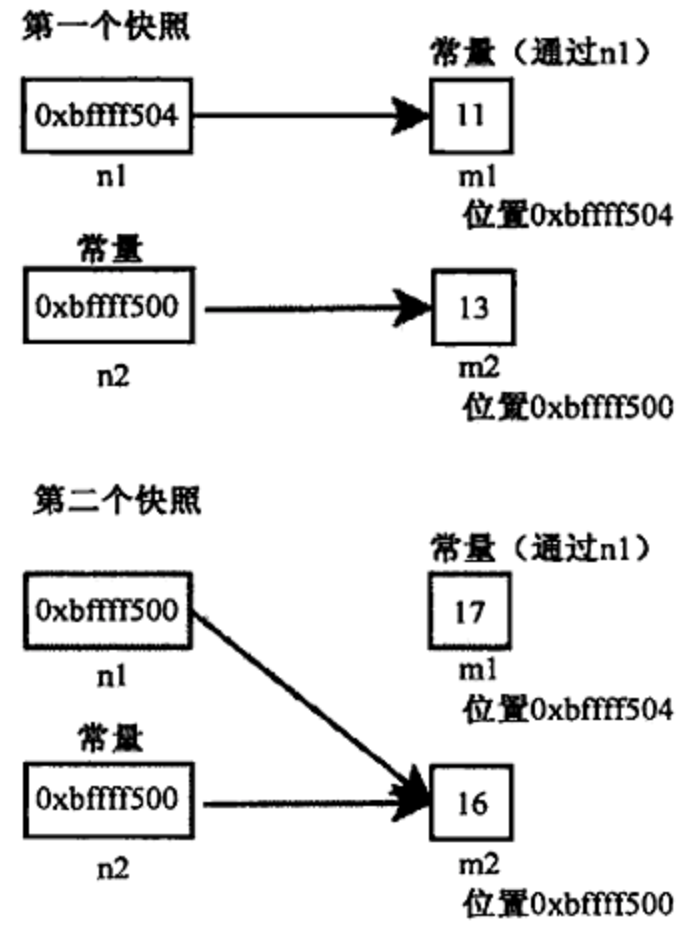


图 1.5　内存快照

即使一个对象通过某个指针访问时是只读的，也可以通过另一个指针访问它而进行改变。这一事实通常体现在函数的设计中。例如

```
char* strcpy(char* dst, const char* src); // strcpy cannot change *src
```

给指向 const 变量的指针赋值一个变量的地址是可行的，但将一个 const 对象的地址赋给一个未限定(非 const)的指针变量是错误的，因为那样就允许对 const 对象的值进行改变。

```
int a = 1;
const int c = 2;
const int* p1 = &c;    // okay
const int* p2 = &a;    // okay
int* p3 = &c;          // error
*p3 = 5;               // error
```

一种好的编程实践是：对于不需要通过函数来进行变化的指针和引用参数，应使用 const 来保护它们。只读引用参数的能力在于：对按引用传递参数提供了高效性，而对按值传递参数提供了安全性(参见 5.5 节)。

1.18 复习题

1. 什么是流？存在哪些类型的流？
2. 给出使用 ostrstream 的一个理由。
3. getline 与>>运算符之间的主要区别是什么？
4. 下列各个表达式是什么类型？

   (1) 3.14
   (2) 'D'
   (3) "d"
   (4) 6
   (5) 6.2f
   (6) “something stringy”
   (7) false

5. 示例 1.31 中，为每一个带编号的项给出类型和值。

示例 1.31 src/types/types.cpp

```
#include <QTextStream>

int main() {
        QTextStream cout(stdout);
        int i = 5;
        int j=6;
        int* p = &i;                                    1
        int& r=i;
        int& rpr=(*p);                                  2
        i = 10;
        p = &j;                                         3
        rpr = 7;                                        4

        r = 8;
        cout << "i=" << i << " j=" << j << endl;        5
        return 0;
}
```

   (1) *p: _____
   (2) *p: _____
   (3) *p: _____
   (4) rpr: _____
   (5) i: _____ j: _____

6. 指针与引用有什么不同？
7. 关键字 const 的作用是什么？为什么要在程序中使用它？如何使用它？
8. 取址运算符是什么？为什么要在程序中使用它？如何使用它？
9. 解引用运算符是什么？为什么要在程序中使用它？如何使用它？

10. 什么是空指针？为什么在程序中应定义它？
11. 什么是内存泄漏？程序中什么情况会导致内存泄漏？
12. 什么情况会导致段错误(或者 Windows 系统中的通用保护错误)？
13. 当处理指针时，可能使用 const 的情形是什么？
14. 什么是函数的签名？
15. 术语“函数重载”的含义是什么？
16. 位于同一作用域中的两个函数具有相同的签名而返回类型不同，为什么这是一个错误？
17. 为什么 main(int argc, char* argv[])有时带有参数？这些参数的作用是什么？

1.18.1　关于指针的更多探讨

1. 参见 21.1 节，了解指针的使用和误用的情况。
2. 参见第 19 章，了解指针如何与类型进行转换。

第2章 类与对象

本章讲解类和对象，并会探讨对象上的成员函数操作。类和对象可以具有不同的结构，本章将给出描述它们的关系的一些方法。将介绍 UML，也会解释 static 成员和 const 成员。还会探讨构造函数、析构函数、复制操作以及友元。

2.1 struct 简介

在 C 语言中，关键字 struct 使程序员可以定义一块结构化的内存，用于存储具有各种数据类型的一个数据集。示例 2.1 给出了一个结构化内存块的定义，它由几个较小的内存块组成。

示例 2.1 src/structdemo/demostruct.h

```
[ . . . . ]
struct Fraction {
    int numer, denom;
    string description;
};
[ . . . . ]
```

struct 中的每一个小型内存块(numer, denom, description)都可以通过名称访问。这些小型的内存块被称为数据成员(data member)，有时也被称为字段(field)。位于 struct 定义外部的代码，被称为客户代码(client code)。示例 2.2 展示了客户代码如何能够将 struct 当作一个实体使用。

示例 2.2 src/structdemo/demostruct.cpp

```
[ . . . . ]
void printFraction(Fraction f) {                                    1
    cout << f.numer << "/" << f.denom << endl;
    cout << "  =? " << f.description << endl;
}
int main() {
    Fraction f1;
    f1.numer = 4;
    f1.denom = 5;
    f1.description = "four fifths";
    Fraction f2 = {2, 3, "two thirds"};                             2

    f1.numer = f1.numer + 2;                                        3
    printFraction(f1);
    printFraction(f2);
    return 0;
}
```

1 如果一个结构的组成元素非常庞大，那么通过值传递这个结构的代价是非常大的。

2 成员初始化。

3 客户代码可以改变各个数据成员的值。

输出如下所示。

```
6/5
=? four fifths
2/3
=? two thirds
```

这个应用中的 printFraction() 函数以各自特有的方法在屏幕上显示了各个数据成员的值。还要注意，客户代码能够产生一个具有错误描述的 Fraction 对象。

2.2 类定义

C++语言有另外一种称为类(class)的数据类型，它类似于 struct。类的定义如下所示。

```
class ClassName {
    public:
        publicMembers
    private:
        privateMembers
};
```

类定义的第一行称为类首部(classHead)。

类的特性包括数据成员、成员函数以及访问限定符(public, private, protected)。成员函数用来初始化、操作或者管理数据成员。第 5 章中更详细地探讨了函数，尤其是 C++中独有的一些函数特性。这里使用函数的方式，是保证它在上下文环境中是清晰的，且对具有其他编程语言经验的人是熟悉的。

在定义了一个类之后，其类名称就可以作为变量、参数和函数返回值的类型来使用。类类型的变量被称为这个类的对象或者实例。

ClassName 类的成员函数指定了 *ClassName* 类型的所有对象的行为。每一个成员函数都能够访问类中的其他成员。非成员函数只能通过调用成员函数间接得操作对象。

对象的数据成员的值集合，被称为这个对象的状态。

2.2.1 头文件

为了定义类(或者任何其他类型)，应将它的定义放在头文件中，这个文件一般与类同名，后缀为 .h。示例 2.3 中给出了一个包含类定义的头文件。

示例 2.3 src/classes/fraction/fraction.h

```
#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <QString>
class Fraction {
public:
    void set(int numerator, int denominator);
```

```
    double toDouble() const;
    QString toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

预处理器会将头文件包含到其他的文件中。为了防止头文件在某个编译过的文件中被错误地包含多次,一般使用#ifndef…#define…#endif 预处理器宏来将头文件包裹起来(参见 C.2 节)。

一般而言,应当将成员函数的定义置于类定义的外面,将其放入一个单独的实现文件中,其扩展名为.cpp。

示例 2.4 是一个包含示例 2.3 中声明的函数定义的实现文件。

示例 2.4 src/classes/fraction/fraction.cpp

```
#include <QString>
#include "fraction.h"


void Fraction::set(int nn, int nd) {
    m_Numerator = nn;
    m_Denominator = nd;
}

double Fraction::toDouble() const {
    return 1.0 * m_Numerator / m_Denominator;
}

QString Fraction::toString() const {
  return QString("%1 / %2").arg(m_Numerator).arg(m_Denominator);
}
```

每一个标志符都有作用域(参见 20.2 节),即一个代码区,标志符在这个代码区内是“可知的”(“可见的”)或者是“可访问的”。在前面的几个示例中,我们看到了具有代码块作用域的标志符,它从声明标志符所在的那一行开始,向下一直到包含这个声明的代码块结束。在这个声明的前面以及这个代码块的后面,标志符都是不可见的。

类成员名称具有类作用域。开始时,类作用域包含整个类定义,不管这个类定义中在哪里声明成员名称。它也扩展到实现文件(.cpp 文件)。位于类定义之外的任何类成员的定义,都要求在名称前面有一个形式为“*ClassName::*”的作用域解析运算符。作用域解析运算符告诉编译器,类的作用域扩展到了类定义之外,包含了位于符号“::”和函数定义的闭括号之间的代码。

例如,尽管成员 Fraction::m_Numerator 和 Fraction::m_Denominator 是在一个单独的文件中声明的,但它们在 Fraction::toString()和 Fraction::toDouble()的定义里面依然是可见的。

实践中经常需要做的是:显示一个对象、将对象保存到文件或者通过网络发送到另一个程序。有许多种途径能够执行这些操作。toString()成员函数通常会返回一个字符串,它包含对象当前状态的一个“快照”。可以利用这个字符串来实现调试、显示、存储、传送或者

转换目的。为了增加灵活性，可以向 toString()函数提供一个或者多个参数，使字符串具有各种格式。许多 Qt 类就是这样做的。例如，根据传递的 Qt::DateFormat 实参的情况，QDate::toString()会返回各种格式的日期。

可以尝试将 toString()成员函数用于具有数据成员的那些类。通常而言，类定义不应当包含那些显示或者传送数据成员值的成员函数，例如，display(), print(), saveToFile(), readFromFile()等[①]。

2.3 成员访问限定符

前面已经看到了类定义代码和类实现代码，前者在头文件中包含类定义和其他声明，后者在与头文件对应的.cpp 文件中包含头文件中没有的那些定义。第三类代码与某个给定的类相关。客户代码是位于类作用域之外的代码，但它使用了这个类的对象或者成员。通常而言，客户代码包含一个具有类定义的头文件。示例 2.5 中，可以看到 fraction.h 包含 Fraction 的类定义代码，这个文件本身是 QString 的一个客户。

示例 2.5 src/classes/fraction/fraction.h

```
#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <QString>


class Fraction {
public:
    void set(int numerator, int denominator);
    double toDouble() const;
    QString toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

访问限定符 public, protected 和 private 用于类定义中，它们指定了在程序中的哪些位置可以访问被修饰的成员。下面给出这三个关键字的非正式的定义，更详细的定义可以参见脚注。

- 只要一个程序包含了某个类的定义文件，就可以在程序的任何位置(通过该类的对象)访问 public 成员[②]。
- protected 成员只能在本类的成员函数或其派生类的成员函数的定义中访问[③]。
- private 成员仅仅允许在本类的成员函数中访问[④]。

---

① 用于输入/输出的面向流的数据序列化，将在 7.4.1 节中探讨。
② public static 成员可以不通过对象而进行访问。2.9 节将探讨这种情况。
③ 将在第 6 章讨论派生类。
④ private 成员也可以通过类的友元函数进行访问，将在 2.6 节对此进行讨论。

默认情况下，类成员是 private 类型的。如果类定义的成员声明前面没有访问限定符，则成员就是 private 类型的。

可访问性与可见性

可访问性与可见性之间存在微妙的差异。具有名称的某个项在其作用域内都是可见的。可访问性适应于类成员。如果要使被命名的类成员具有可访问性，它首先必须是可见的。并不是所有可见的项都是可访问的。可访问性依赖于成员访问限定符 public，private 和 protected。

示例 2.6 使用 Fraction 客户代码以各种方式演示了可访问性和可见性。这个示例还关注了块作用域。在语句块里声明的变量(不是类成员)，仅在其声明语句与块结束括号之间是可见的和可访问的。对于函数来说，包含此函数定义的语句块也包含它的参数表。

示例 2.6 src/classes/fraction/fraction-client.cpp

```
#include <QTextStream>
#include "fraction.h"

int main() {
    const int DASHES = 30;
    QTextStream cout(stdout);

    {                                                          1
        int i;
        for (i = 0; i < DASHES; ++i)
            cout << "=";
        cout << endl;
    }

    cout << "i = " << i << endl;                               2
    Fraction f1, f2;
    f1.set(3, 4);
    f2.set(11,12);                                             3
    f2.m_Numerator = 12;                                       4
    cout << "The first fraction is: " << f1.toString() << endl;
    cout << "\nThe second fraction, expressed as a double is: "
         << f2.toDouble() << endl;
    return 0;
}
```

1 嵌套作用域，内部语句块。

2 错误：i 不再存在，所以在这个作用域内是不可见的。

3 通过成员函数设置。

4 错误：m_Numerator 是可见的，但不是可访问的。

现在，可以清楚地给出 C++中 struct 和类的关系了。Stroustrup 将 struct 定义成其成员默认都为 public 类型的一个类。这样

```
struct T { ...
```

等价于

```
class T {public: ...
```

特别地，C++中的 struct 可以具有成员函数和数据。C++程序员在多数情况下更愿意使用 class 而不是 struct，也许是因为更偏爱于 private 访问。应用中使用 struct 的大多数情形，是需要将数据项分组在一起，但是不需要成员函数。

2.4 封装

封装是面向对象编程中的第一个概念性步骤，它主要包括：

- 将数据和操作数据的函数一起封装在适当命名的类中。
- 提供命名清晰、文档良好的 public 函数，使类的用户能够对这个类的对象做任何需要做的事情。
- 隐藏实现细节。

类中的 public 函数的原型集合被称为这个类的 public 接口。非 public 类成员的集合加上这些成员函数的定义，被称为类的实现。

封装的一个直接优势是它允许程序员使用一致的类成员命名机制。例如，可以有多个不同的类具有一个数据成员，该成员包含某个特定实例的单位成本，或者如前面提到的，具有一个名称为 toString() 的成员函数。由于类成员的名称在类作用域之外是不可见的，所以可以安全地在需要成员的每一个类中使用名称 m_unitCost 和 toString()[①]。

2.5 UML 介绍

现代的、面向对象的应用是以设计良好的类为基础的。在大多数工程中，通用用途的库（比如 Qt）和特定任务的库就提供这样的类。程序员提供其他的类。UML（Unified Modeling Language，统一建模语言）是用于面向对象设计的最常用语言。它使得设计者能够用丰富的框图描述工程。我们使用 UML 框图，是因为“一图胜万言”。例如，UML 类框图能够以简明而直观的方式展示出类中重要的元素或者有关联的元素，以及这些元素之间的关系。其他的 UML 框图可以展示类之间如何协作、用户如何与类对象交互等。本书中将仅使用 UML 的一个很小的子集。

书中大部分的 UML 框图都是使用 UML 设计工具 Umbrello 创建的[②]。为了对 UML 有一个更好的理解，推荐阅读 Umbrello 帮助菜单中附带的 *The Umbrello UML Modeller Handbook*。另外一个有参考意义的文献是[Fowler04]，它以最小的篇幅提供了最多的内容。

图 2.1 所示是一个仅包含一个 Person 类的类框图。注意，声明是 Pascal 风格的 *name*：*type*，而不是我们相对较熟悉的 C++/Java 风格，在 C++/Java 风格中，声明时名称位于类型之后。这

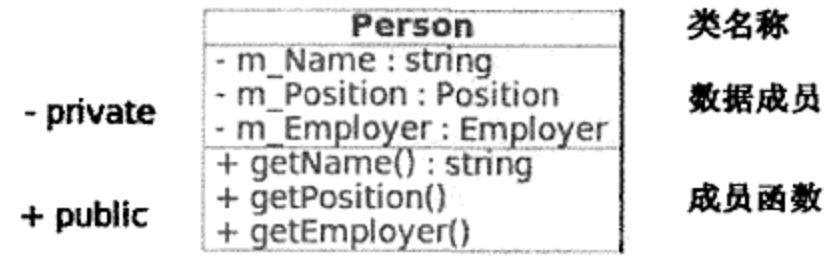


图 2.1 Person 类

① 编译器不会混淆，因为它会将类的名称嵌入所提供的每一个数据成员的完整名称中，

② 参见 http://uml.sourceforge.net。

种风格有助于提高可读性。因为我们总是倾向于从左向右阅读，而这种语法能够帮助我们更快地看到名称。同时也要注意，public 成员前面有一个加号，而 private 成员前有一个减号。

2.5.1 UML 关系

UML 尤其擅长描述类之间的关系。先跳到示例 2.22，它描述了 UML 中的一种重要关系。在这个示例中引入了一个 Point 类和一个 Square 类，前者表示屏幕上的一个几何点，后者代表屏幕上的一个几何形状。Point 类具有两个 int 类型的数据成员，而 Square 类具有两个 Point 数据成员。由于这两个 Point 数据成员是类对象，所以它们被认为是 Square 类的子对象，而 Square 对象是它的 Point 子对象的父对象。当删除了 Square 对象时，其子对象也会被删除。这使得子对象成为了父对象的一部分，它们之间的关系被称为“组合”(composition)。图 2.2 中，实心菱形表明这个类的实例与关系连接线另一端的那个类的实例是“组合”关系(至少部分如此)。

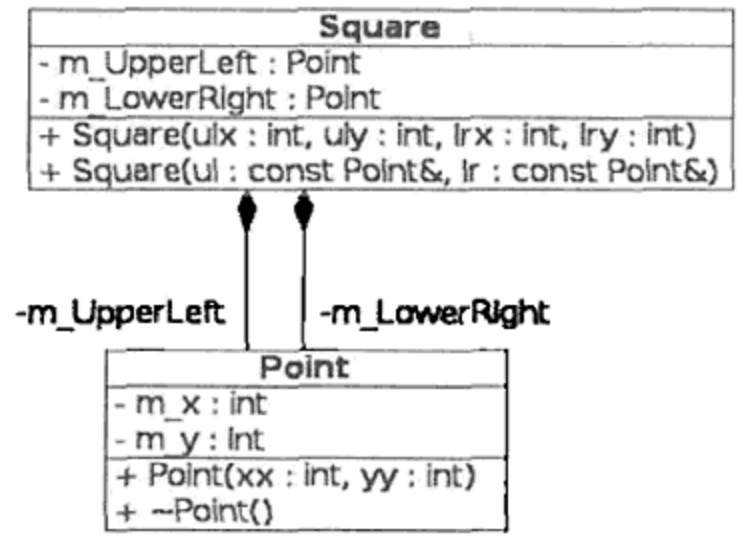


图 2.2 组合

2.6 类的友元

前面讲解了类的访问性规则，现在讲解如何偶尔打破这些规则。友元机制允许非成员函数访问一个类的私有数据。关键字 friend 可以放在类声明或者函数声明之前，友元声明位于类定义之内。以下是几个语法示例。

```
class Timer {
   friend class Clock;
   friend void Time::toString();
   friend ostream& operator <<(ostream& os, const Timer& obj);
   [...]
 private:
   long m_Elapsed;
};
```

友元可以是一个类、另外一个类的成员函数或者任何一个非成员函数。在前面的示例中，Clock 类是一个友元类，因此其所有的成员函数都可以访问 Timer::m_Elapsed。Time::toString()是 Timer 的一个友元函数，编译器会假设此函数是 Time 类的一个有效成员。第三个友元是一个非成员函数，即一个重载的插入运算符，此运算符将其第二个实参插入到输出流中，同时返回这个流的一个引用，这样就可以将这一类的操作连接起来。

破坏封装性可能会危害程序的可维护性，因此应该尽量少地使用友元，不得已使用时也要倍加小心。通常而言，为了达到下面两个目的才会使用友元函数。

1. 为了使用工厂方法，此时需要对某个类强制实施某些创建规则(参见 16.1 节)。
2. 为了使用全局运算符函数，比如 operator<<()和 operator>>()，此时不希望将运算符作为某个类的成员函数，或者没有写入(write-access)类定义的权利。

2.7 构造函数

构造函数是一种控制对象初始化过程的特殊成员函数。构造函数必须与它的类同名。它不返回任何值，也不存在返回类型。

构造函数定义的语法比较特殊，如下所示。

```
ClassName::ClassName( parameterList )
        :initList                                    1
        {
           constructor body
        }
```

1 可选但重要。不要省略它，即使编译器允许这样做也不行。

在参数表的右括号和函数体的左括号之间，可以提供一个可选的成员初始化表。成员初始化表以一个冒号开始，后面是一个以逗号分隔的成员初始化项列表，其形式如下。

```
memberName(initializingExpression)
```

当且仅当类的定义中没有提供任何构造函数时，编译器才会提供一个如下的构造函数。

```
ClassName::ClassName()
    { }
```

能够不带实参进行调用的构造函数，被称为默认构造函数。默认构造函数会对它的类对象进行默认初始化。任何没有在成员初始化表中明确初始化的数据成员，都会被编译器赋予默认初始值。

类可以有多个构造函数，每一个都可以用不同(但有用)的方式初始化。示例 2.7 中包含三个构造函数。

示例 2.7 src/ctor/complex.h

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

class Complex {
 public:
    Complex(double realPart, double imPart);
    Complex(double realPart);
    Complex();
    string toString() const;
 private:
    double m_R, m_I;
};
```

示例 2.8 中包含的一段客户代码展示了它们的实现。

示例 2.8 src/ctor/complex.cpp

```
#include "complex.h"
#include <iostream>
#include <sstream>
using namespace std;
```

```
Complex::Complex(double realPart, double imPart)
    :   m_R(realPart), m_I(imPart)
{
    cout << "complex(" << m_R << "," << m_I << ")" << endl;
}

Complex::Complex(double realPart) :
    m_R(realPart), m_I(0) {
}

Complex::Complex() : m_R(0.0), m_I(0.0) {

}

string Complex::toString() const {
    ostringstream strbuf;
    strbuf << '(' << m_R << ", " << m_I << ')';
    return strbuf.str();
}

int main() {
    Complex C1;
    Complex C2(3.14);
    Complex C3(6.2, 10.23);
    cout << C1.toString() << '\t' << C2.toString()
         << C3.toString() << endl;
}
```

1 成员初始化表。

这个类的默认构造函数将对象 C1 中的两个数据成员进行了默认初始化。这种初始化类似于下面代码中两个 double 类型变量的初始化。

```
double x, y;
cout << x << '\t' << y << endl;
```

问题

你认为这段代码的输出是什么?

如果没有成员初始化表，会发生什么？例如，考虑下面的构造函数定义：

```
Complex(double realPart, double imPart) {
   m_R = realPart;
   m_I = imPart;
}
```

每一个数据成员首先都包含默认初始值，然后被赋予一个值。这样做不会有错误发生，但初始化会浪费处理时间。

再分析一个示例，参考图 2.2，出于演示的目的，假设在 Point 类框图中只存在一个 Point 构造函数。进一步假设，定义的 Square 构造函数没有成员初始化表。

```
Square::Square(const Point& ul, const Point& lr) {
    m_UpperLeft = ul;
    m_LowerRight = lr;
}
```

由于没有为 Point 类明确地定义默认构造函数，而是定义了一个带两个参数的构造函数，所以 Point 类就没有默认构造函数。因此，编译器将会对这个 Square 构造函数报告错误。

2.8 析构函数

析构函数是一种特殊的成员函数，会在对象销毁之前进行自动清理工作。

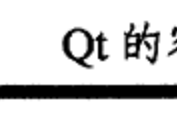
对象何时销毁

- 当一个局部(自动)对象超出其作用域时(例如，当函数调用返回时)。
- 通过 new 运算符创建的对象使用 delete 运算符销毁时。
- 当程序终止时，所有静态存储的对象都会被销毁。

析构函数的名称是以一个波浪号(~)开始的类名称。它没有返回类型，也没有任何参数。因此，类只能有一个析构函数。如果类定义中没有包含析构函数的定义，则编译器会提供一个如下的析构函数。

```
ClassName::~ClassName()
    { }
```

2.9 节中将给出一个较为完整的析构函数例子。



何时需要编写析构函数

通常而言，一个直接管理或者共享某个外部资源(打开文件，打开网络连接，创建进程，等等)的类，需要在合适的时刻释放资源。这样的类通常是一个负责对象清除工作的封装器。Qt 的容器类使得我们可以避免编写直接管理动态内存的代码。

如果某个类满足以下条件，则无须提供析构函数。

- 只拥有非指针的简单类型成员
- 拥有已经适当定义的析构函数的类成员
- 它是满足某些条件的一种 Qt 类[①]

由编译器生成的默认析构函数，会在此类的对象被销毁之前调用该类所有成员的析构函数，调用时按照成员在类定义中出现的次序进行。当销毁指针成员时，默认析构函数不会删除所分配的内存。

① 这种类将在第 8 章深入探讨。

2.9 static 关键字

关键字 static 可以用于局部变量、类成员以及全局变量和全局函数。在不同情况下，static 的含义有所不同。

static 局部变量

关键字 static 可以用在局部变量的声明中，使变量成为静态存储类(参见 20.3 节)。

程序运行过程中，只有首次处理 static 局部变量的声明语句时才会创建并初始化这个变量；程序终止时，会销毁这个变量。当对象模块加载到内存中时，非局部的 static 变量会被创建一次，程序终止时会销毁它。

static 类成员

static 数据成员是一块与类本身相关联的数据，而不是属于某个特定的对象。它不会影响通过 sizeof()运算符获得的类对象的大小。类的每一个对象都维护属于自己的非 static 数据成员，而任何 static 数据成员只有一个实例，并且被该类的所有对象共享。

相对于全局变量，我们更倾向于使用 static 成员(且一般可以代替全局变量)，因为 static 成员不会在全局命名空间中增加不必要的名称。

全局命名空间污染

将名称添加到全局作用域中(例如，通过声明全局变量或者全局函数)，被称为全局命名空间污染(global namespace pollution)，这被认为是一种不好的编程风格，其中之一是它会增加名称冲突和名称混淆的可能性。有许多正当理由可以避免在程序中声明全局变量。有些专家将程序中全局名称的数量当作判断程序质量好坏的一个因素(数量越少，质量越高)。

static 类成员必须在(也只能在)类定义中声明为 static 的。

示例 2.9 src/statics/static.h

```
[ . . . . ]
class Thing {
public:
    Thing(int a, int b);
    ~Thing();
    void display() const ;
    static void showCount();
private:
    int m_First, m_Second;
    static int s_Count;
};
[ . . . . ]
```

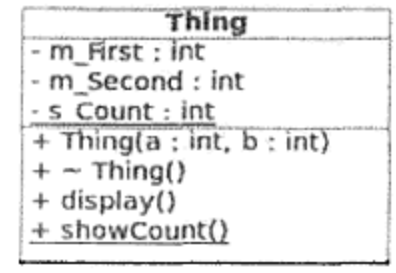


图 2.3 static 类的 UML 类定义

图 2.3 中给出了示例 2.9 中 Thing 类的 UML 类框图。注意，在框图中，static 成员加有下画线。

一个不以任何形式访问该类的非 static 数据成员的类成员函数，也可以(并且应当)声

明成 static 的。示例 2.9 中的 static 数据成员是一个私有计数器，它保存给定时刻存在的 Thing 类对象的个数。而 public static 成员函数用于显示这个 static 计数器的当前值。

每一个 static 数据成员都必须在类定义之外的文件中被初始化(定义)，合适的地方是对应的类实现文件[①]。示例 2.10 中给出了如何初始化并使用 static 成员。

示例 2.10 src/statics/static.cpp

```
#include "static.h"
#include <iostream>

int Thing::s_Count = 0;                                          1

Thing::Thing(int a, int b)
        : m_First(a), m_Second(b) {
    ++s_Count;
}

Thing::~Thing() {
    --s_Count;
}

void Thing::display() const {
    using namespace std;
    cout << m_First << "$$" << m_Second;
}

void Thing::showCount() {                                        2
    using namespace std;
    cout << "Count = " << s_Count << endl;
}
```

1 必须初始化 static 成员！

2 static 函数。

术语 static 并没有在 s_Count 或 showCount() 的定义中出现。对于 s_Count 的定义，关键字 static 的含义将会完全不同：它会将变量的作用域由全局作用域变成文件作用域(参见 20.2 节)。对于函数定义，它完全是多余的。

块作用域内的 static 变量

在一个函数或者一个代码块内定义的 static 变量，会在第一次执行时进行初始化。

```
long nextNumber() {
    int localvar(24);
    static long statNum = 1000;
    cout << statNum + localvar;
    return ++statNum;
}
```

① 这个规则的一种例外情况是 static const int，它可以在类定义中被初始化。

第一次调用 nextNumber() 函数时，会将 localvar 初始化成 24，将 statNum 初始化成 1000，然后在屏幕上输出 1024，并返回 1001。当函数返回时，localvar 被销毁，而 statNum 不会。以后每次调用此函数时，localvar 都会重新创建并被初始化为 24。static 变量 statNum 一直存在，并在多次调用之间拥有上次调用时的值。因此，第二次调用该函数时会输出 1025 并返回 1002。

static 变量初始化

不在代码块或者函数内定义的 static 对象，在其相应的对象模块[①]首次加载时会进行初始化。大部分情况是在程序启动之后、main() 函数运行之前。模块的加载顺序和变量初始化顺序与具体的实现有关，所以绝对不要使初始化工作依赖于另一个文件中的初始值，即使在编译时将这个文件列在前面也不可以。

static 对象只会构建一次，并保持到程序终止。static 数据成员可以看成一个具有类际作用域的 static 对象。

示例 2.11 使用了一个内部语句块来建立一些局部对象，它们会在程序终止之前被销毁。

示例 2.11 src/statics/static-test.cpp

```
#include "static.h"

int main() {
    Thing::showCount();                                    1
    Thing t1(3,4), t2(5,6);
    t1.showCount();                                        2
    {                                                      3
        Thing t3(7,8), t4(9,10);
        Thing::showCount();                                4
    }                                                      5

    Thing::showCount();
    return 0;
}
```

1 这时还不存在任何对象，但所有的 static 类都已经被初始化。

2 通过对象访问。

3 进入代码的内部语句块。

4 通过类作用域解析运算符访问。

5 终止内部语句块。

以下是编译并运行的结果。

```
src/statics> g++ -Wall static.cpp static-test.cpp
src/statics> ./a.out
Count = 0
Count = 2
Count = 4
Count = 2
src/statics>
```

---

① 参见第 7 章。

static 全局变量

当将 static 应用于全局函数或者变量时，它不会按照所期待的方式执行。static 不会改变存储类，而是告知链接器(linker)不要将这个符号输出到程序的其他部分。这会赋予符号文件作用域，将在 20.2 节中讨论。

2.10 类的声明和定义

图 2.4 中显示的这种双向关系，在类中也经常遇到。为了实现这种关系，在某一个类中定义另一个类之前，编译器需要对定义的这个类有所了解。首先想到的是在头文件中包含另一个头文件，如示例 2.12 所示。

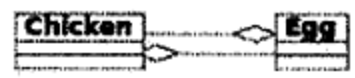


图 2.4 双向关系

示例 2.12 src/circular/badegg/egg.h

```
[ . . . . ]
#include "chicken.h"
class Egg {
 public:
    Chicken* getParent();
};
[ . . . . ]
```

当看到示例 2.13 时，问题就出现了。这个示例中包含了头文件 egg.h。

示例 2.13 src/circular/badegg/chicken.h

```
[ . . . . ]
#include "egg.h"

class Chicken {
 public:
    Egg* layEgg();
};
[ . . . . ]
```

预处理器不允许这样的循环依赖。这个示例中，两个头文件都不需要包含另一个头文件。在每一种情况下，这样做会不必要地在头文件之间建立一种强依赖性。

在条件正确的情况下，C++允许使用前置类声明(forward class declaration)，而不必包含某个特定的头文件。

示例 2.14 src/circular/goodegg/egg.h

```
[ . . . . ]
class Chicken;                                          1
class Egg {
 public:
    Chicken* getParent();                               2
};
[ . . . . ]
```

1 前置类声明。

2 如果为指针，则可以这样声明。

前置类声明使得不必具有某个符号的完整定义就可以引用这个符号。这会向编译器隐性地承诺：当需要时，类的定义将被包含进来。已经声明但没有定义的类，只能用作指针类型或者引用类型，只要它没有在文件中被解引用。

源代码模块 egg.cpp 中定义了 getParent() 函数，见示例 2.15。注意，.cpp 文件中可以包含多个头文件，而不会导致它们之间的循环依赖性。.cpp 文件对这两个头文件具有强依赖性，但头文件之间彼此不存在依赖性。

示例 2.15 src/circular/goodegg/egg.cpp

```
#include "chicken.h"
#include "egg.h"

Chicken* Egg::getParent() {
    return new Chicken();                                    1
}
```

1 要求 Chicken 的定义。

因此，前置类声明使得定义双向关系成为可能，比如示例 2.15 中的这种情形，这样就不会创建循环包含文件了。我们将真正需要的那些依赖关系在源代码模块中设置，而不是将其放在头文件中。

Java 中可以在两个(或者多个)类之间创建强循环双向依赖性。换句话说，每一个类都可以导入并使用(解引用)其他的类。这种循环依赖性使得对两个类的维护变得更困难，因为其中一个的改变就会影响到另一个。与 Java 相比，C++能够更好地保护程序员的一种情况是：在 C++中无法创建这样一种关系。

Java 还提供前置类声明，但很少使用它，因为 Java 程序没有将头文件和实现文件分开。

更多细节，请参见 C.2 节。

2.11 复制构造函数与赋值运算符

C++给类的设计者提供了“几乎等同于上帝”的权力！对象的“生命周期”管理意味着完全地控制对象的诞生、繁殖和消亡的过程。前面已经看到如何通过构造函数管理对象的诞生，也看到了如何通过析构函数管理对象的消亡。本节将探讨对象的繁殖过程：使用复制构造函数和赋值运算符。

复制构造函数(copy constructor)是一种构造函数，其原型类似于

```
ClassName(const ClassName & x);
```

复制构造函数的作用是创建一个对象，该对象是同一个类中已有对象的精确副本。

针对某个类的赋值运算符(assignment operator)重载了符号=，其含义对特定的类都不相同。赋值运算符的一个特殊版本的原型如下

```
ClassName& operator=(const ClassName& x);
```

因为在一个类中 operator=() 可能有几个不同的重载版本，我们将这个特殊的版本称为复制赋值运算符(copy assignment operator)。

示例 2.16 中的 Fraction 版本具有三个 static 计数器，见示例 2.17 中的定义，所以能够计算出每一个成员函数调用的总次数。这有助于更好地理解何时会复制对象。

示例 2.16 src/lifecycle/copyassign/fraction.h

```
[ . . . . ]
class Fraction {
 public:
    Fraction(int n, int d) ;                                    1
    Fraction(const Fraction& other) ;                           2
    Fraction& operator=(const Fraction& other) ;                3
    Fraction multiply(Fraction f2) ;
    static QString report() ;
 private:
    int m_Numer, m_Denom;
    static int s_assigns;
    static int s_copies;
    static int s_ctors;
};
[ . . . . ]
```

1 常规构造函数。

2 复制构造函数。

3 复制赋值构造函数。

示例 2.17 src/lifecycle/copyassign/fraction.cpp

```
[ . . . . ]
int Fraction::s_assigns = 0;                                    1
int Fraction::s_copies = 0;
int Fraction::s_ctors = 0;

Fraction::Fraction(const Fraction& other)
   :  m_Numer(other.m_Numer), m_Denom(other.m_Denom) {
   ++s_copies;
}

Fraction& Fraction::operator=(const Fraction& other) {
    if (this != &other) {                                       2
        m_Numer = other.m_Numer;
        m_Denom = other.m_Denom;
        ++s_assigns;
    }
    return *this;                                               3
}
[ . . . . ]
```

1 static 成员定义。

2 在自赋值中，operator=()应什么也不做。

3 对于链式赋值，例如 a = b = c，operator=()应总是返回*this。

示例 2.18 用这个类来创建、复制并赋值一些对象。

示例 2.18 src/lifecycle/copyassign/copyassign.cpp

```
#include <QTextStream>
#include "fraction.h"
```

```
int main() {
    QTextStream cout(stdout);
    Fraction twothirds(2,3);                                    1
    Fraction threequarters(3,4);
    Fraction acopy(twothirds);                                  2
    Fraction f4 = threequarters;                                3

    cout << "after declarations - " << Fraction::report();
    f4 = twothirds;                                             4
    cout << "\nbefore multiply - " << Fraction::report();
    f4 = twothirds.multiply(threequarters);                     5
    cout << "\nafter multiply - " << Fraction::report() << endl;
    return 0;
}
```

1 使用带两个实参的构造函数。

2 使用复制构造函数。

3 同样使用复制构造函数。

4 赋值。

5 这里创建了几个对象。

以下是程序的输出。

```
copyassign> ./copyassign
after declarations -  [assigns: 0 copies: 2 ctors: 2]
before multiply -  [assigns: 1 copies: 2 ctors: 2]
after multiply -  [assigns: 2 copies: 3 ctors: 3]
copyassign>
```

问题

正如所看到的，调用 multiply 可以创建三个 Fraction 对象。请解释其原因。

编译器提供的版本

这里需要知道的一件非常重要的事情是：如果去掉类定义中的复制构造函数或复制赋值运算符(或者都去掉)，编译器就会提供它们的默认版本。编译器提供的默认版本是 public 类型的，对于类 *T* 的这些默认版本的原型如下。

```
T::T(const T& other);
T& T::operator=(const T& other);
```

这两个默认版本都会精确复制每一个数据成员的值。对于其数据类型都为简单类型或者值类型的类，比如 int, double, QString 等，编译器提供的版本没有问题。但是，如果类具有指针成员或者对象成员[①]，则需要同时编写对这个类特别设计的复制构造函数和复制赋值构造函数。后面将看到[②]，有时有必要阻止某些类对象的复制。这种情况下，需

① 参见 8.1 节。

② 参见第 8 章。

要将复制构造函数和复制赋值构造函数都声明成 private 类型的，并应适当地将它们定义成非复制的，以防止编译器提供 public 版本。

2.12 转换

可以只用一个(不同类型的)实参进行调用的构造函数，称为转换构造函数(conversion constructor)，它定义了将实参类型转换成构造函数的类类型的一种转换(conversion)。

示例 2.19 src/ctor/conversion/fraction.cpp

```
class Fraction {
public:
    Fraction(int n)                                                    1
       : m_Numer(n), m_Denom(1) {}
    Fraction(int n, int d )
       : m_Numer(n), m_Denom(d) {}
    Fraction times(const Fraction& other) {
       return Fraction(m_Numer * other.m_Numer, m_Denom * other.m_Denom);
    }
private:
    int m_Numer, m_Denom;
};
int main() {
    int i;
    Fraction frac(8);                                                  2
    Fraction frac2 = 5;                                                3
    frac = 9;                                                          4
    frac = (Fraction) 7;                                               5
    frac = Fraction(6);                                                6
    frac = static_cast<Fraction>(6);                                   7
    frac = frac2.times(19);                                            8
    return 0;
}
```

1 单一实参构造函数定义了 int 类型的一种转换。

2 转换构造函数调用。

3 复制 int 值(也会调用转换构造函数)。

4 转换后赋值。

5 C 语言风格的类型转换(已被弃用)。

6 显式的临时变量，也是 C++类型转换。

7 推荐的 ANSI 风格的类型转换。

8 隐式调用转换构造函数。

示例 2.19 的 main() 函数中，Fraction 变量 frac 使用一个 int 类型的参数进行了初始化。匹配的构造函数是单参数版本的那一个。它会有效地将整数 8 转换成分数 8/1。

转换构造函数的原型通常如下所示。

```
ClassA::ClassA(const ClassB& bobj);
```

当需要 ClassA 类的一个对象，并且该对象可以由提供的 ClassB 的值通过 ClassA 的转换构造函数创建时，就会自动调用这个构造函数。当然，ClassB 的值需要由初始化器或者通过赋值提供。

例如，如果 frac 如示例 2.19 中定义的那样是一个被正确地初始化的 Fraction，就可以编写语句

```
frac = frac.times(19);
```

因为 19 不是 Fraction 类的一个对象，如 times() 函数定义所要求的那样，所以编译器会检查它是否可以转换成一个 Fraction 对象。而又因为存在一个转换构造函数，所以这确实是可行的。

由于没有定义 Fraction::operator=()，所以编译器使用它提供的默认赋值运算符：

```
Fraction& operator=(const Fraction& fobj);
```

这个赋值运算符按照成员的顺序从 fobj 的数据成员到主对象相应的数据成员进行赋值操作。

因此，这条语句会这样调用三个 Fraction 成员函数：

1. Fraction::operator=()执行赋值。
2. Fraction::times()执行乘法。
3. Fraction::Fraction(19)将 19 从 int 转换成 Fraction。

由 times()函数返回的临时 Fraction 对象，其生存时间只够完成赋值操作，随后会自动销毁。

可以简化 Fraction 的类定义，去除单参数构造函数，为两参数构造函数的第二个参数提供一个默认值。因为只用一个实参就可以调用它，所以它满足转换构造函数的定义。

示例 2.20 src/ctor/conversion/fraction.h

```
class Fraction {
public:
    Fraction(int n, int d = 1)
            : m_Numer(n), m_Denom(d) {}
    Fraction times(const Fraction& other) {
       return Fraction(m_Numer* other.m_Numer, m_Denom* other.m_Denom);
    }

private:
    int m_Numer, m_Denom;
};
```

通常而言，任何可以只使用一个不同类型的实参进行调用的构造函数，都可以看成是转换构造函数，这些构造函数都有前面讨论过的隐式机制。如果这些隐式机制由于某些原因而不合适，有可能需要将其取消。关键字 explicit 将阻止编译器为了隐式转换而自动使用此构造函数[①]。

explicit

可以将关键字 explicit 放于类定义中单实参构造函数的前面，以防止编译器使用自动转换。如果这个实参不与类所构造的参数相似，或者二者之间存在父-子关系，则使用 explicit 关键字就是有用的。QtCreator 生成的类会将这个关键字放于所产生的定制窗件构造函数的前面，建议程序员对 QWidget 派生的类也这样做。

① 19.8.4 节中非正式地探讨了一个示例。

2.13 const 成员函数

当通过 objx 对象 objx.f()调用类成员函数 ClassX::f()时：就称 objx 为主对象(host object)。

当将关键字 const 应用到一个(非 static 的)类成员函数时，其含义是比较特殊的。如果放置在参数表之后，const 就成为函数签名的一部分，从而保证此函数不会改变主对象的状态。

要理解这一点，一个好方法是意识到每一个非 static 成员函数都有一个称为 this 的隐式参数，this 是一个指向主对象的指针。当将一个成员函数声明为 const 时，就相当于告诉编译器只要此函数被调用，this 就是一个指向 const 的指针。

为了解释 const 如何改变调用函数的方式，可以看一下原始的 C++到 C 语言的预处理器是如何处理成员函数的。因为 C 语言不支持重载函数或成员函数，预处理器会将这些函数翻译成具有“重整名称”(mangled name)的 C 语言函数，在这个名称中包含了函数的完整签名，以此将其与其他函数相区分。重整过程也在参数表中添加一个额外的隐式参数：this，即执行主对象的指针。示例 2.21 中给出了 C 语言中经过翻译后用链接器得到的成员函数。

示例 2.21 src/const/constmembers/constmembers.cpp

```
#include <QTextStream>
#include <QString>

class Point {
  public:
    Point(int px, int py)
       : m_X(px), m_Y(py) {}

    void set(int nx, int ny) {                                    1
        m_X = nx;
        m_Y = ny;
    }
     QString toString() const {                                   2
        // m_X = 5;                                               3
         m_Count++;                                               4
         return QString("[%1,%2]").arg(m_X).arg(m_Y);

    }
  private:
    int m_X, m_Y;
    mutable int m_Count;                                          5
};


int main() {
   QTextStream cout(stdout);
   Point p(1,1);
```

```
    const Point q(2,2);
    p.set(4,4);
    cout << p.toString() << endl;
    //q.set(4,4);
    return 0;
}
```


1 C语言版本：_Point_set_int_int(Point* this, int nx, int ny)。
2 C语言版本：_Point_toString_string_const(const Point* this)。
3 错误：this->m_X = 5, *this 为常量。
4 可以这样做，成员的值是可变的。
5 在 const 方法中可以更改可变对象的值。
6 可以对 p 重新赋值。
7 错误：Const 对象不能调用非 const 方法。

在具体的编译器中，为了节省空间，set 和 print 的重置名称会被大幅压缩，但这样对于我们来说也就更加难以理解。注意，可以在 const 成员函数里更改可变成员(mutable member)的值，而常规的数据成员的值不能改变。

可以将 print()签名中的 const 看成是指向主对象的不可见参数 this 的修饰符。这意味着 this 指向的内存无法被 print()函数的行为改变。赋值语句“x = 5;”会导致错误的理由是：它等价于语句“this->x = 5;”，而这种赋值违反了 const 的规则。

假设需要用到一个工程，其中的类没有正确地使用 const。当向成员函数、参数和指针添加 const 时，会发现这种改变将导致编译器错误的不断出现，从而无法建立工程。解决之道是在整个工程中都应正确地添加 const。当最终在所有地方都正确地添加了 const 之后，就可以认为类已经是“const 正确的”(const correct)。

2.14 子对象

对象中可以包含另外的对象，被包含的对象被称为子对象。示例 2.22 中，每一个 Square 对象都具有两个 Point 子对象(subobject)。

示例 2.22 src/subobject/subobject.h

```
[ . . . . ]
class Point {
 public:
    Point(int xx, int yy) : m_x(xx), m_y(yy){}
    ~Point() {
       cout << "point destroyed: ("
            << m_x << "," << m_y << ")" << endl;
    }
 private:
    int m_x, m_y;
};

class Square {
 public:
```

```
    Square(int ulx, int uly, int lrx, int lry)
    : m_UpperLeft(ulx, uly), m_LowerRight (lrx, lry)                    1
    {}

    Square(const Point& ul, const Point& lr) :
    m_UpperLeft(ul), m_LowerRight(lr) {}                                2
private:
    Point m_UpperLeft, m_LowerRight;                                    3
};

[ . . . . ]
```

1 此处的成员初始化是必要的，因为没有默认的构造函数。

2 用隐式生成的 Point 复制构造函数初始化成员。

3 嵌入的子对象。

只要创建了 Square 的实例，就会创建它的每一个子对象，因此全部三个对象会占据一个连续的内存块。当销毁 Square 实例时，也会销毁其全部的子对象。

Square 对象由两个 Point 对象组成。由于 Point 没有默认构造函数[①]，所以必须在 Square 的成员初始化表中正确地初始化每一个 Point 子对象[②]。

示例 2.23 是创建类实例的客户代码。

示例 2.23 src/subobject/subobject.cpp

```
#include "subobject.h"

int main() {
    Square sq1(3,4,5,6);
    Point p1(2,3), p2(8, 9);
    Square sq2(p1, p2);
}
```

尽管没有为 Square 定义析构函数，但只要父对象被销毁了，其每一个 Point 子对象都会被正确地销毁。正如 2.5.1 节中看到的那样，这是组合(composition)的一个例子。细节将在 6.9 节中探讨。

```
point destroyed: (8,9)
point destroyed: (2,3)
point destroyed: (8,9)
point destroyed: (2,3)
point destroyed: (5,6)
point destroyed: (3,4)
```

2.15 练习：类

1. 示例 2.24 到示例 2.26 是一个程序的三个部分，将它们作为一个整体回答下列问题。

① 思考一下，为什么没有默认构造函数?

② 这里为什么不能简单地初始化 m_x 和 m_y?

示例 2.24 src/early-examples/thing/thing.h

```
#ifndef THING_H_
#define THING_H_

class Thing {
 public:
    void set(int num, char c);
    void increment();
    void show();
 private:
    int m_Number;
    char m_Character;
};
#endif
```

示例 2.25 src/early-examples/thing/thing.cpp

```
#include <QTextStream>
#include "thing.h"


void Thing::set(int num, char c) {
  m_Number = num;
  m_Character = c;
}

void Thing::increment() {
  ++m_Number;
  ++m_Character;
}

void Thing::show() {
  QTextStream cout(stdout);
  cout << m_Number << '\t' << m_Character << endl;
}
```

示例 2.26 src/early-examples/thing/thing-demo.cpp

```
#include <QTextStream>
#include "thing.h"


void display(Thing t, int n) {
    int i;
    for (i = 0; i < n; ++i)
        t.show();
}


int main() {
  QTextStream cout(stdout);
  Thing t1, t2;
  t1.set(23, 'H');
  t2.set(1234, 'w');
```

```
    t1.increment();
    //cout << t1.m_Number;
    display(t1, 3);
    //cout << i << endl;
    t2.show();
    return 0;
}
```


a. 将示例 2.26 中的两个注释行的注释符号删除，用如下的命令链编这个程序：

```
qmake -project
qmake
make
```

解释由编译器报告的错误。

b. 将 public 成员函数添加到 Thing 类的定义中，使得数据成员虽然是私有的，但客户代码依然可以输出它们的值。

2. 根据图 2.5 中的 UML 框图，定义增强的 Fraction 类以及所指定的每一个成员函数。以示例 2.4 为基础。编写一些客户代码，检验是否进行了正确的计算。

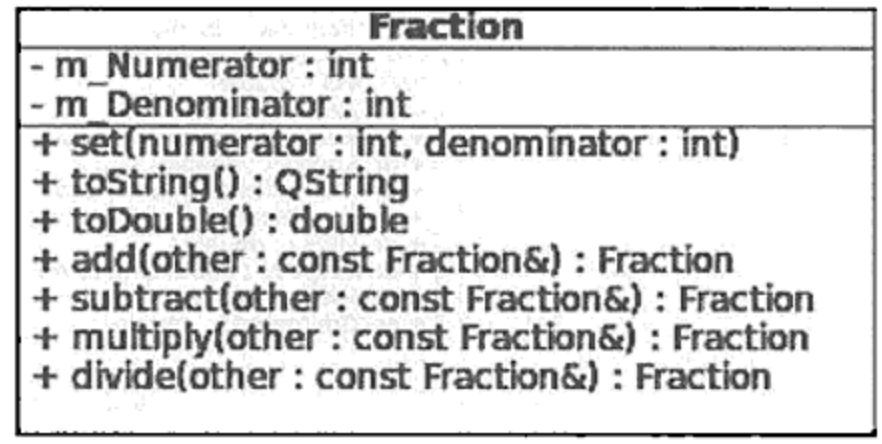


图 2.5 Fraction 类框图

3. 假设要为某公司编写一个应用，将求职者与雇员进行匹配。第一步将是设计出合适的类。图 2.6 是设计的开始部分。在这个框图中，Person 有两个子对象：Employer 和 Position。

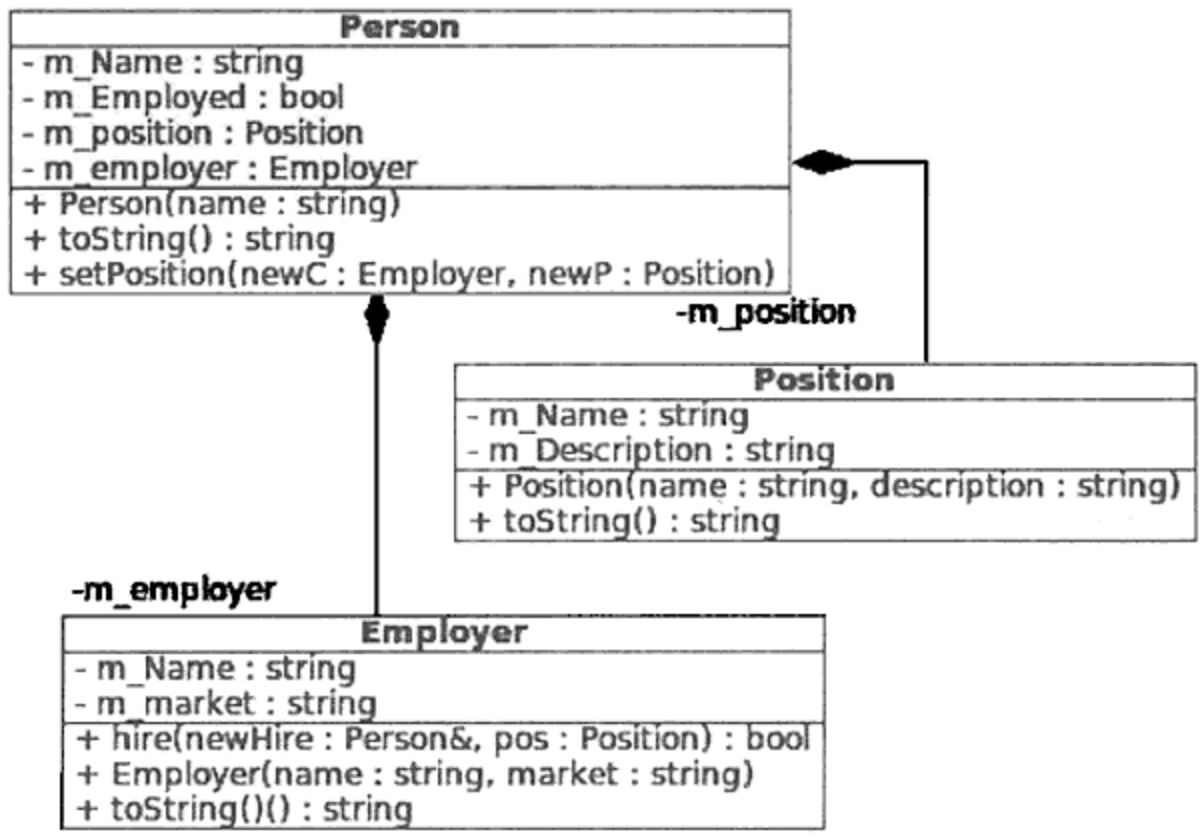


图 2.6 公司图表

为了完成此练习，需要使用前置类声明(参见 2.10 节)。

a. 根据图 2.6 中的描述编写出 Person 类、Position 类和 Employer 类。

b. 创建 Person::getPosition()和 getEmployer()函数，如果某人还没有被公司雇用，则返回一些有趣的信息。

c. 针对 hire(...)函数设置 Person 的状态，使得后面对 getPosition()和 getEmployer()函数的调用能给出正确的结果。
d. 在 main()程序中，至少创建两位雇主：StarFleet Federation 和 Borg。
e. 至少创建两位员工：Jean-Luc Picard 和 Wesley Crusher。
f. 对每一个类都编写 toString()函数，提供每一个对象的 string 表示。
g. 编写创建某些对象的主程序，然后输出每一个公司的员工列表。

4. 分析图 2.6 中的设计。能够看出哪些问题？特别地，应该如何编写 Employer::getEmployees()函数和 Position::getEmployer()函数？给出优化这种设计的建议。
5. 根据图 2.7 定义一个表示当代汽车的类。

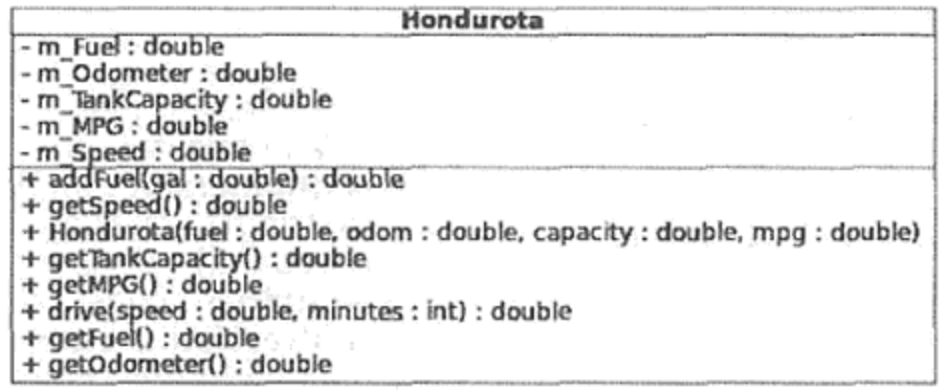


图 2.7 Hondurota 类

以下是这个类的一些特性。

- 除了 4 个命名的数据成员外，构造函数应初始化速度。值 0 看似是合理的，但它不适合于正在移动的汽车。
- drive()函数应相当聪明：
  - 如果没有燃料则禁止汽车行驶。
  - 正确调整里程计和燃料总量。
  - 返回油箱中剩余的燃料量。
  - 函数 addFuel()应该正确地调整燃料数量，并返回油箱中燃料的总量。addFuel(0)应该将油箱加满油。

编写客户代码，测试这个类。

6. 在上一个问题中，实际上没有使用速度成员。drive()函数假设了一个平均速度，并且使用了一个平均燃料消耗率。现在将采用速度成员，以使问题更加接近于现实情况。给类 Hondurota 增加一个成员函数，其原型如下：

```
double highwayDrive(double distance, double speedLimit);
```

返回值是本次行驶耗费的时间。
当在高速公路上行驶时，通常以最高限速或接近最高限速行驶。遗憾的是，各种各样事情的发生使得行驶速度要低很多，有时甚至是缓慢的。
另外一个有趣的因素是速度的变化会对燃料消耗率产生影响。大多数当代的汽车都有一个最优能效的行驶速度(例如，每小时 45 英里，45 mph)。新函数的作用是计算在高速公路上行驶指定距离需要花费的时间和燃料量。

- 编写这个函数，使得在完成指定距离之前，每分钟都更新速度、里程计数以及燃料消耗量。
- 使用 45 mph 作为 m_FuelConsumptionRate 存储的精确燃料消耗率对应的速度。
- 对其他与 45 mph 不同的速度都使用一个修正的燃料消耗率，每英里每小时的燃料消耗率增加 1%。
- 如果油料耗光，则应该立即停车。
- 假设以最高限速行驶，但每分钟都有一个在-5 mph 到+5 mph 之间的随机速度调整。确保汽车不会比最高限速快 40 mph。当然，也不可能以小于 0 mph 的速度行驶。如果随机速度调整产生了不可接受的速度，则需重新产生一个速度。

编写客户代码，测试这个函数。

7. 预测示例 2.28 的输出结果(它是示例 2.27 的客户代码)。

示例 2.27 src/statics/static3.h

```
#ifndef _STATIC_3_H_
#define _STATIC_3_H_

#include <string>
using namespace std;

class Client {
 public:
    Client(string name) : m_Name(name), m_ID(s_SavedID++)
        { }
    static int getSavedID() {
        if(s_SavedID > m_ID) return s_SavedID;
        else return 0;
    }
    string getName() {return m_Name;}
    int getID() {return m_ID; }
 private:
    string m_Name;
    int m_ID;
    static int s_SavedID ;
};

#endif
```

示例 2.28 src/statics/static3.cpp

```
#include "static3.h"
#include <iostream>

int Client::s_SavedID(1000);

int main() {
    Client cust1("George");
    cout << cust1.getID() << endl;
    cout << Client::getName() << endl;
}
```

8. 根据下面的限制和建议，以图 2.8 为基础设计并实现一个 Date 类。

图 2.8 Date 类的 UML 框图

- 每一个 Date 类对象都应该用一个单独的整数进行存储，这个整数等于当前日期与 1000 年 1 月 1 日(或者自己喜欢的任意日期)的天数之差。将这个数据成员称为 m_DaysSinceBaseDate。
- 基准年份应该存储为一个 static int 数据成员(例如，1000 或者 1900)。
- 类拥有一个构造函数和一个 set 函数，它们都以年、月、日为参数。这三个值必须用来计算从基准日期到给定日期的天数之差。这里指定了一个名称为 ymd2dsbd() 的 private 成员函数来进行这项计算。
- toString() 函数以某种标准字符串格式返回存储日期的表示，该字符串要适合于显示结果(例如，yyyy/mm/dd)。这涉及上面描述的 ymd2dsbd() 函数中计算的逆向计算。这里指定了一个名称为 getYMD() 的 private 成员函数来负责这项计算。还建议为 toString() 函数提供一个参数(bool brief)，以进行日期格式的选择。
- 指定了几个 static 实用函数(utility function)，例如 leapyear()。它们之所以是 static 类型的，是因为它们不会影响任何 Date 对象的状态。
- 确认使用了正确的规则来判断某一年是否为闰年。
- 创建一个名称为 date.h 的文件，保存类定义。
- 创建名称为 date.cpp 的文件，其中包含在 date.h 中声明的所有函数的定义。
- 编写客户代码，全面测试这个 Date 类。
- 类应当以合理的方式处理“无效”日期(例如，年份早于基准年份，月份或者日期超出范围等)。
- 下面是 setToToday() 函数的代码，此函数使用了系统时钟来判断当天的日期。为了使用这些代码，需要加入语句#include <time.h>(来自 C 语言标准库)。

```
void Date::setToToday() {
  struct tm *tp = 0;
  time_t now;
  now = time(0);
  tp = localtime(&now);
  set(1 + tp->tm_mon, tp->tm_mday, 1900 + tp->tm_year);
}
```

- getWeekDay() 函数根据存储的日期返回当天的星期名称。在 toString() 的“美化”版本中使用此函数。提示：1900 年 1 月 1 日是星期一。

9. 考虑示例 2.29 中的类。

示例 2.29 src/destructor/demo/thing.h

```
#ifndef THING_H_
#define THING_H_
#include <iostream>
#include <string>
```

```
using namespace std;

class Thing {
 public:
    Thing(int n) : m_Num(n) {

    }
    ~Thing() {
        cout << "destructor called: "
             << m_Num << endl;
    }

 private:
    string m_String;
    int m_Num;
};
#endif
```

示例 2.30 中的客户代码以各种方式创建了几个对象，随后销毁了其中的绝大部分对象。

示例 2.30 src/destructor/demo/destructor-demo.cpp

```
#include "thing.h"

void function(Thing t) {
    Thing lt(106);
    Thing* tp1 = new Thing(107);
    Thing* tp2 = new Thing(108);
    delete tp1;
}

int main() {
    Thing t1(101), t2(102);
    Thing* tp1 = new Thing(103);
    function(t1);
    {                                                    1
        Thing t3(104);
        Thing* tp = new Thing(105);
        }
        delete tp1;
        return 0;
    }
```

1 嵌套的语句块作用域。

以下是程序的输出。

```
destructor called: 107
destructor called: 106
destructor called: 101
destructor called: 104
destructor called: 103
destructor called: 102
destructor called: 101
```

a. 有多少对象创建了但没有被销毁?
b. 为什么 101 会在结果中出现两次?

2.16 复习题

1. 描述 class 与 struct 之间的至少一个不同点。
2. 类作用域与语句块作用域的不同点是什么?
3. 描述应该使用友元函数的两种情形。
4. static 数据成员与非 static 数据成员相比有何不同?
5. static 成员函数与非 static 成员函数相比有何不同?
6. 将一个成员函数声明为 const 有何意义?
7. 试解释如果某个类 T 有一个原型如下的复制构造函数,则会发生什么?为什么会发生?

```
T::T(T other);
```

8. 示例 2.31 中标记出的一些行中包含错误。示例 2.32 中给出的答案是多选的。

示例 2.31 src/quizzes/constquiz.cpp

```
#include <iostream>

class Point {
  public:
  Point(int px, int py)
        : m_X(px), m_Y(py) {}                                    1

    void set(int nx, int ny) {
        m_X = nx;
        m_Y = ny;
    }
    void print() const {
        using namespace std;
        cout << "[" << m_X << "," << m_Y << "]";
        m_printCount ++;                                         2
    }
  private:
    int m_X, m_Y;
    int m_printCount;                                            3
};

int main() {
    Point p(1,1);
    const Point q(2,2);
    p.set(4,4);                                                  4
    p.print();
    q.set(4,4);                                                  5
    q.print();                                                   6
    return 0;
}
```

1 _____
2 _____
3 _____
4 _____
5 _____
6 _____

示例 2.32 src/quizzes/constquiz-questions.txt

1. 以下对错误的描述，哪些是正确的?
   a. 不允许在此处出现。
   b. m_pointCount 没有出现，导致编译错误。
   c. {}内缺少分号。
   d. m_pointCount 没有出现，导致运行时错误。

2. 以下对错误的描述，哪些是正确的?
   a. 没有错误。
   b. 需将 m_printCount 声明成 const。
   c. 需将 m_printCount 声明成 explicit。
   d. 编译器错误：不会改变 m_printCount 的值。
   e. 需将 m_printCount 声明成 volatile。

3. 以下对错误的描述，哪一个是正确的?
   a. 没有错误。
   b. 需将 m_printCount 声明成 volatile。
   c. 需将 m_printCount 声明成 const。
   d. 需将 m_printCount 声明成 mutable。
   e. 需将 m_printCount 声明成 explicit。

4. 以下对错误的描述，哪些是正确的?
   a. 不能调用 const 成员。
   b. 不能调用非 const 成员。
   c. 没有错误。
   d. Set 应为 const 的。
   e. Set 应为 volatile 的。

5. 以下对错误的描述，哪些是正确的?
   a. 不能调用 const 成员。
   b. 不能调用非 const 成员。
   e. Set 应为 volatile 的。
   d. q 应为非 const 的。
   e. Set 应为 volatile 的。

6. 以下对错误的描述，哪一个是正确的?
   a. 没有错误。
   b. 不能调用非 const 成员。
   c. 需将 print 声明成 const。
   d. q 应为 explicit 的。
   e. 不能调用 const 成员。

9. 找出示例 2.33 中的错误，并回答示例 2.34 中的问题。

示例 2.33 src/quizzes/statics-quiz.cpp

```
// wadget.h:

class Wadget {

public:
    Wadget(double a, double b);
    void print();
    static double calculation();
    static int wadgetCount();

private:
    double m_d1, m_d2;
    static int m_wadgetCount;
};
```

```
// wadget.cpp:

Wadget::Wadget(double a, double b)
:  m_d1(a), m_d2(b) {
    m_wadgetCount ++;
}

static int wadgetCount() {
    return m_wadgetCount;
}

double Wadget::calculation() {
    return d1*d2 + m_wadgetCount;
}
[ . . . . ]
```

示例 2.34 src/quizzes/statics-quiz.txt

1. 示例 2.33 中的代码存在许多问题。首先，编译它时不会进行链接，因为缺少 m_wadgetCount 的定义。应该如何修复这个问题?
   a. 在 wadget.h 的类定义里面添加语句
      static int m_wadgetCount = 0;
   b. 在 wadget.h 的类定义之外添加语句
      static int Wadget::m_wadgetCount = 0;
   c. 在 wadget.cpp 文件的顶部添加语句
      int Wadget::m_wadgetCount = 0;
   d. 在 wadget.cpp 文件的顶部添加语句
      static int Wadget::m_wadgetCount = 0;

2. 对于 static int Wadget::wadgetCount()的声明，这里的“static”表示什么?
   a. 函数必须在.cpp 文件中定义。
   b. 函数只能被 static 对象调用。
   c. 函数必须用“Wadget::”作用域解析运算符调用。
   d. 函数名称具有文件作用域。
   e. 函数只能访问 static 成员。

3. 对于 static int Wadget::wadgetCount()的定义，这里的“static”表示什么?

a. 函数只能访问 static 成员。
b. 函数只能被 static 对象调用。
c. 函数必须用“Wadget::”作用域解析运算符调用。
d. 函数名称会被输出到链接器。
e. 函数名称不会被输出到链接器。

4. 对于 Wadget::calculation()的定义，以下正确的是哪些?
a. d1 和 d2 不能从 static 方法访问。
b. 函数定义的前面缺少“static”。
c. 没有错误。
d. a 和 b。
e. 函数名称不会被输出到链接器(发生错误)。

第3章　Qt简介

这一章将会对用于本书其余部分的一些风格指南和命名约定进行介绍。Qt核心模块会用一些例子和使用了Qt流及数据类的练习中进行介绍。

Qt是一个类和工具的模块化系统，它可以使用户更容易地编写出自己的小代码模块。Qt提供了一个近乎完美的STL类/类型替代品，可在比使用C++0x编写的代码更多的编译器上进行构建/运行，并支持一些无须现代编译器的同样特性。这一章将介绍如何开始复用Qt。

在Qt的源文档内，有一个示例和范例(Examples and Demos)集，有时会需要引用它，它们位于带有“examples/”或者“demos/”路径前缀的目录中。如果使用的是Qt SDK或者是用Linux安装的Qt，或许需要运行安装包管理器来安装它们。图3.1给出了来自Qt Creator欢迎画面的Getting Started界面。

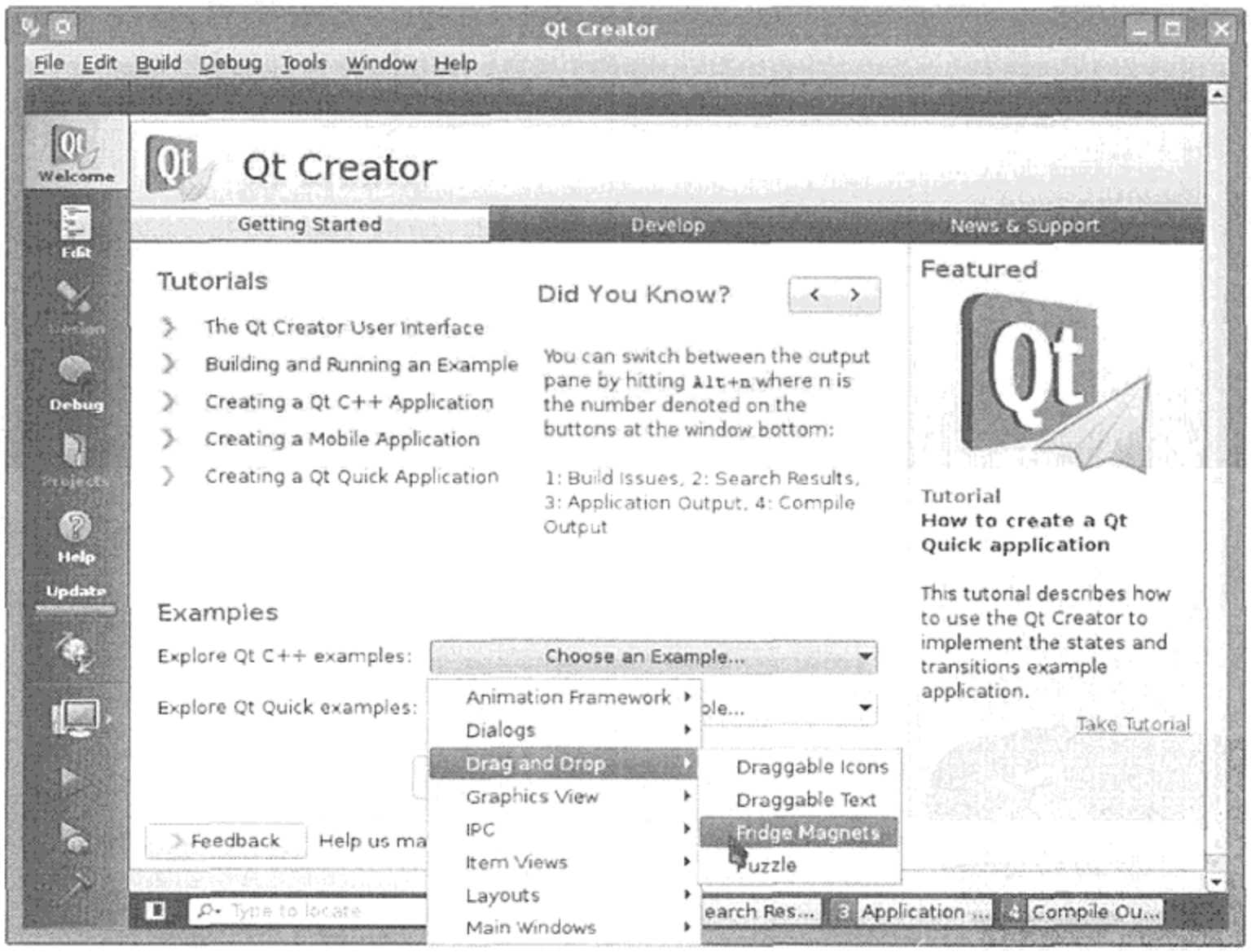


图3.1　Qt Creator的欢迎界面

注意

附录E中给出了一些在不同平台上进行快速设置的提示。

3.1　风格指南与命名约定

C++是一种支持多种不同编程风格的功能强大的编程语言。在大多数 Qt 程序中使用的编码风格并非“纯”C++，取而代之的是一种宏与预处理技巧相结合的风格，以此来获得一种相对于 C++来说更加类似于 Java 和 Python 的高级动态语言。事实上，为了更充分地利用 Qt 的强大功能和使用简单的特性，我们倾向于完全放弃标准库(Standard Library)。

在任意协作严密的编程项目中，可能都有风格指南来提高编写的代码的可读性、可复用性和可靠性。在[qtapistyle]和[kdestyle]中给出了半官方的 Qt 编程风格指南。这里对采用的一些风格总结如下。

- 名称是一些字母和数字构成的序列，第一位不能为数字。
- 名称的第一位也可以使用下画线字符(_)，但是不鼓励使用它，除非是用于类的数据成员。
- 类名称以大写字母开头，例如：`class Customer`。
- 函数名称以小写字母开头。
- 通过合并多个单词并且让每个单词首字母大写，即用“驼峰规则”(CamelCase)的方式构造多单词的名称，例如，`class MetaDataLoader`,`void getStudentInfo()`。
- 常量应当大写并且尽可能在类的作用域内创建成枚举值，全局常量和宏通常应当都是全部大写。
- 每一个类名称都应当是一个名词或者名词短语，例如，`classLargeFurryMammal`。
- 每一个函数名称都应当是一个动词或者动词短语，例如，`processBookOrder()`。
- 用于`if()`语句时，每一个布尔变量都应当近似于一个句子，例如，`bool isQualified`。

对于数据成员，本书采用改进的匈牙利标记法，其中会使用共同的前缀，以便在代码中让数据成员总是清晰可见。

- 数据成员：`m_Color`，`m_Width`——以小写字母 m 开头。
- 静态数据成员：`s_Singleton`，`s_ObjCount`——以小写字母 s 开头。

对于每一个属性，其相应的获取器/设置器(getter/setter)都有约定俗成的命名规则。

- 非布尔型获取器：`color()`或者`getColor()`[①]。
- 布尔型获取器：`isChecked()`或者`isValid()`。
- 设置器：`setColor(const Color& newColor)`。

一致的命名约定往往能够极大地提高程序的可读性和可维护性。

3.1.1　其他编码标准

下面是一些广泛应用的其他编码标准。需要记住的是，与 Qt 编程最为相关的风格依旧是[qtapistyle]。

① 后者属于 Java 风格，而前者属于 Qt 风格。两种约定都有广泛的使用，只要在代码中保持一致即可(本书中并没有保持一致，是因为希望能够展示一些不同的约定)。

- C 和 C++风格文档[①]。
- 编码标准生成器[②]。

3.2 Qt 核心模块

Qt 是一个大库，由数个较小的库或者模块组成，其中最为常见的有如下这些。

- core——包括 QObject, QThread, QFile, Qvariant, 等等。
- gui——所有从 QWidget 派生的类外加一些相关的类。
- xml——用于解析和序列化 XML。
- sql——用于与 SQL 数据库通信。
- phonon——用于播放多媒体文件。
- webkit——用于使用一种嵌入式 Web 浏览器，QtWebkit。

除了 core 和 gui，这些模块都需要在 qmake 的工程文件中启用。例如

```
QT += xml  # to use the xml module
QT -= gui  # to not use QWidgets
QT += sql  # to use SQL module
```

3.2.1 节将会介绍一些核心库中的类。

3.2.1 流和日期

在之前的数个例子中，已经看到 QTextStream 的几个实例，该类在行为上类似于 C++ 标准库中的全局 iostream 对象。当使用它们来与标准输入(键盘)和标准输出(屏幕)交互时，希望将它们命名成类似的名称，cin, cout 和 cerr。为了使用方便，将这些定义和另外一些有用的函数一起放置到了一个命名空间内，这样就可以轻松地用#include 将这些定义包含到任何程序中了。

示例 3.1 src/qstd/qstd.h

```
[ . . . . ]
namespace qstd {

    // declared but not defined:
    extern QTextStream cout;
    extern QTextStream cin;
    extern QTextStream cerr;

    // function declarations:
    bool yes(QString yesNoQuestion);
    bool more(QString prompt);
    int promptInt(int base = 10);
    double promptDouble();
    void promptOutputFile(QFile& outfile);
```

① 参见http://www.chris-lott.org/resources/cstyle/。

② 参见http://www.rosvall.ie/CSG/。

```
    void promptInputFile(QFile& infile);
};
[ . . . . ]
```

示例 3.1 声明了类似于 iostream 的 QTextStream 对象，示例 3.2 包含了这些静态对象所需的定义。

示例 3.2　src/qstd/qstd.cpp

```
[ . . . . ]

QTextStream qstd::cout(stdout, QIODevice::WriteOnly);
QTextStream qstd::cin(stdin, QIODevice::ReadOnly);
QTextStream qstd::cerr(stderr, QIODevice::WriteOnly);

/* Namespace members are like static class members */
bool qstd::yes(QString question) {
    QString ans;
    cout << QString(" %1 [y/n]? ").arg(question);
    cout.flush();
    ans = cin.readLine();
    return (ans.startsWith("Y", Qt::CaseInsensitive));
}
```

QTextStream 能够针对 Unicode 的 QString 和其他 Qt 类型起作用，因此在后面的示例中，将使用 QTextStream 而不再使用 iostream。示例 3.3 使用 QTextStream 对象和刚刚介绍过的 qstd 命名空间中的函数。该例还使用 QDate 类的一些函数成员以几种不同的格式输出日期。

示例 3.3　src/qtio/qtio-demo.cpp

```
[ . . . . ]
#include <qstd.h>

int main() {

    using namespace qstd;
    QDate d1(2002, 4,1), d2(QDate::currentDate());
    int days;
cout << "The first date is: " << d1.toString()
        << "\nToday's date is: "
        << d2.toString("ddd MMMM d, yyyy")<< endl;

if (d1 <  d2)
    cout << d1.toString("MM/dd/yy") << " is earlier than "
            << d2.toString("yyyyMMdd") << endl;

cout << "There are " << d1.daysTo(d2)
        << " days between "
        << d1.toString("MMM dd, yyyy") << " and "
        << d2.toString(Qt::ISODate)  << endl;

cout << "Enter number of days to add to the first date: "
        <<  flush;
```

```
        days = promptInt();
        cout << "The first date was " << d1.toString()
                << "\nThe computed date is "
                <<  d1.addDays(days).toString() << endl;
        cout << "First date displayed in longer format: "
                << d1.toString("dddd, MMMM dd, yyyy")  << endl;
[ . . . . ]
```


从这个例子所在的 src 目录，可以构建并运行这个程序。示例 3.4 中给出的工程文件会使用相对路径找到 qstd 的头文件和实现文件。

示例 3.4　src/qtio/qtio.pro

```
CONFIG += debug console
DEFINES += QT_NOTHREAD_DEBUG

CONFIG -= moc
INCLUDEPATH += . ../qstd
DEPENDPATH += ../qstd

# Input
SOURCES += qtio-demo.cpp qstd.cpp
HEADERS += qstd.h
```

以下是程序的输出结果。

```
The first date is: Mon Apr 1 2002
Today's date is: Wed January 4, 2006
04/01/02 is earlier than 20060104
There are 1374 days between Apr 01, 2002 and 2006-01-04
Enter number of days to add to the first date: : 1234
The first date was Mon Apr 1 2002
The computed date is Wed Aug 17 2005
First date displayed in longer format: Monday, April 01, 2002
```

3.3　Qt Creator，用于 Qt 编程的集成开发环境

Qt Creator 是一个跨平台的集成开发环境(Integrated Development Environment，IDE)，它用于简化基于 Qt 的 C++应用程序开发工作。正如所期望的，Qt Creator 含有一个优良的 C++代码编辑器，可以提供智能代码补全、基于 Qt 助手(Qt Assistant)的上下文关联帮助、错误/警告信息键入以及快速代码工具导航等功能。此外，可使用拖动/放下窗体布局来完成设计的 Qt 设计师也完全集成到了 Qt Creator 中。Qt Creator 拥有一个可视化的调试工具和工程构建/管理工具，还有一些非常容易使用的代码生成和导航特性。在 YouTube 上可以获得一些 Qt Creator 的视频教程[①]。

如果仍希望使用自己喜欢的 IDE，并且碰巧是 xcode, MS dev studio 或者 Eclipse，那么你很幸运——可以下载 Qt 集成开发包，它会让你所喜好的 IDE 提供一个与 Qt Creator 相似的特性。从 qt.nokia.com 通过下载二进制安装包或者源代码包，可以把 Qt Creator 安装到 Windows, Mac 和 Linux 平台上[②]。在 Qt 软件开发包(Qt Software Development Kit，SDK)中包含有 Qt Creator，其中包含了 Qt 和快速开始用 Qt 进行开发工作的所有东西。

---

① 参见http://www.youtube.com/view_play_list?p=22E601663DAF3A14。

② 参见http://qt.nokia.com/downloads。

Qt Creator 用到的项目文件是 qmake 的 .pro 文件。通过创建或编辑 Qt Creator 中已有的项目文件，可以完全避免命令行工具的使用。

默认情况下，Qt Creator 会导入工程并会在一个“影子”(shadow)构建目录下建立工程。这就意味着，中间文件和可执行文件并没有和源文件放在一起，而是在其他的地方。而且，程序默认情况下会从那个目录中运行。

要查看/修改工程的构建目录或者运行目录，可以选择 Projects 模式(见图 3.2)并单击 Build 标签修改 Qt 的版本、构建目录或者其他设置。单击 Run 标签，可修改 Run 目录或者其他与运行相关的设置项。

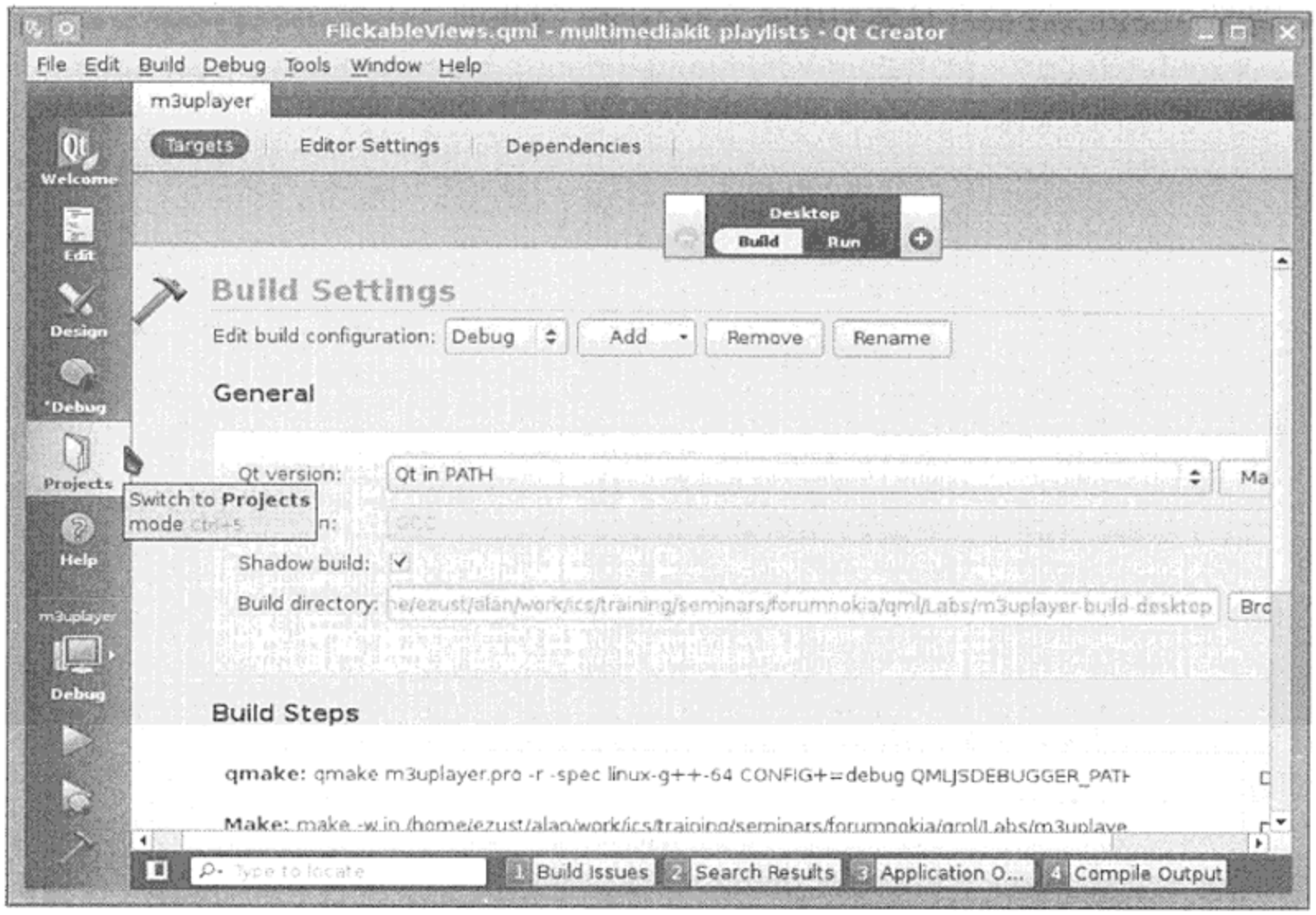


图 3.2　Qt Creator 的 Projects 模式

3.4　练习：Qt 简介

1. 编写一个燃料耗用计算器，可用方向键在英里/加仑和升/100 km 之间转换。
   例如，汽车可以用 1 加仑(美国单位制)燃料行驶 34 英里，那么行驶 100 km 将会耗用 6.92 升的燃料。
   可以使用 1.11 节中演示过的 QInputDialog 和 QMessageBox。
2. 询问用户的生日并计算他的当前年龄。提示：可使用 QInputDialog::getText() 把结果从 QString 转换成 QDate，并将其与 QDate::currentDate() 比较。或者，可以创建一个对话框，显示一个 QDateEdit，从用户那里得到合适的日期值。

3.5 复习题

1. 什么是 QTextStream? 如何复用它?
2. qmake 工程文件中 Qt 变量的作用是什么? 它可能的取值是什么?
3. 为什么程序中应当使用 QTextStream 而不是使用 iostream?

3.5.1 更多探讨

1. 访问 Unicode 编码的 Web 站点，并解释什么是 Unicode 以及为什么说 QString 支持 Unicode 标准非常重要。
2. 浏览 QTextStream 文档，并编写几个测试从文本文件读取数据方式的小程序。
3. 理解 QTextStream 是如何用来支持不同字符集的。

第4章 列 表

只要有可能，就应当使用列表而不是数组。本章将讲解容器，并探讨在列表中进行分组的各种方式，还会讲解如何迭代列表。

4.1 容器简介

在许多情况下都需要处理若干事物的集合。在 C 语言这样的编程语言中，最基本的方法就是使用数组(参见 21.4 节)来存储这种集合。但在 C++中，数组被看成是“邪恶的”。下面是应该避免在 C++中使用数组的一些理由。

- 编译器和运行时系统都不会检查数组下标是否位于正确的范围之内。
- 使用数组的程序员有责任编写额外的范围检查代码。
- 数组的大小可以是固定不变的，或者必须使用堆中的动态内存。
- 对于堆数组，程序员有责任确保在所有可能的情况下当数组销毁时都要正确地释放内存。
- 为此，需要深入理解 C++以及异常，特别是发生异常时的底层处理机制。向数组插入、预分配或追加元素都是费时的操作(在运行时和开发时都是如此)。

C++标准库和 Qt 都为程序员提供了在需要时安全地重新调整数组大小的能力，并会执行范围检查。`std::list` 和 `QList` 是在各自库中最基本的泛型容器。在接口方面它们类似(接口即客户代码使用它们的方式)。二者的不同在于实现(即运行时它们的行为方式)。这两个库还提供了其他几个泛型容器，它们针对特定类型的应用进行了优化。

这里使用术语“泛型容器”，其原因有如下两个。

1. 泛型是指能够接收模板(参见 11.1 节)参数的类或者函数，这样它们就能够用于不同类型的数据。
2. 容器(参见 6.8 节)是指能够包含其他对象的对象。

为了使用泛型容器，客户代码必须包含一个能够回答如下问题的声明：什么的容器？例如

```
QList<double>  doubleList;
QList<Thing>  thingList;
```

`QList` 支持许多操作。对于复用的任何类，推荐阅读 API 文档，以全面了解它的完整功能。对于单一的函数调用，可以有多种方式来添加、删除、交换、查询、清除、移动、定位和计数函数中的项。

4.2 迭代器

在容器中存储元素之后，迟早都要遍历容器对其中的每一个元素进行某种操作。迭代器(iterator)是一个提供对容器中的每一个元素进行间接访问的对象，它专门被设计用在循环之中。

Qt 支持如下几种风格的迭代。

- Qt 风格的 `foreach` 循环。类似于 Perl 和 Python 中的用法。
- Java 1.2 风格的 `Iterator`，它总是指向两个元素之间。
- 标准库风格的 *`ContainerType`*`::iterator`。
- 手写的、使用容器的获取函数的 `while` 循环或者 `for` 循环。
- `QDirIterator`，用于迭代遍历目录结构中的项。

与 Java 风格的迭代器相比，STL 风格的迭代器的行为更与指针类似。迭代器与指针的一个重要差异是：不存在与指针的空值对应的迭代器值。例如，使用指针的函数在搜索集合中的项时，如果搜索失败则可以返回一个空指针。不存在对应的、可被广泛认知的迭代器值能够表示一个失败的、基于迭代器的搜索。

4.2.1 节演示了 C++中 Qt 可用的各种风格的迭代。

迭代器模式

提供通用方法访问集合元素的类、函数或者编程元素，如果没有类型限制，则它们就是迭代器设计模式的实现[①]。C++迭代器、Java 风格的迭代器以及 `foreach` 循环，都是迭代器模式的例子。

Qt 中有许多类都提供了针对各种数据类型的专门化迭代，例如 `QDirIterator`，`QSortFilterProxyModel`，`QTreeWidgetItemIterator` 和 `QDomNodeList`。当引入迭代器时，其定义必须包含足够的信息，以使它能够在容器中从一个项移动到下一个项。

4.2.1 QStringList 与迭代

对于文本处理，对字符串列表进行处理是有用的。`QStringList` 实际上就是一个 `QList<QString>`，这样就能够使用 `QList` 的 public 接口[②]。此外，`QStringList` 还具有一些特别针对字符串的方便函数，例如 `indexOf()`，`join()` 和 `replaceInStrings()`。

利用与 Perl 中类似的 `split()` 函数和 `join()` 函数，在列表与字符串之间进行转换相当简单。示例 4.1 中演示了列表、迭代、`split()` 函数以及 `join()` 函数的用法。

示例 4.1 src/containers/lists/lists-examples.cpp

```
#include <QStringList>
#include <QDebug>

/* Some simple examples using QStringList, split, and join */

int main() {

    QString winter = "December, January, February";
    QString spring = "March, April, May";
```

① 有关设计模式的详细讨论，请参见 7.4 节。

② 事实上，`QStringList` 派生自 `QList<QString>`，所以它继承了 `QList` 的全部 public 接口。第 6 章中探讨了派生和继承。

```
QString summer = "June, July, August";
QString fall = "September, October, November";

QStringList list;
list << winter;                                          1
list += spring;                                          2
list.append(summer);                                     3
list << fall;

qDebug() << "The Spring months are: " << list[1] ;


QString allmonths = list.join(", ");                     4
qDebug() << allmonths;

QStringList list2 = allmonths.split(", ");
/* Split is the opposite of join. Each month will have its own element. */

Q_ASSERT(list2.size() == 12);                            5

foreach (const QString &str, list) {                     6
    qDebug() << QString(" [%1] ").arg(str);
}

for (QStringList::iterator it = list.begin();
     it != list.end(); ++it) {                           7
    QString current = *it;                               8
    qDebug() << "[[" << current << "]]";
}

QListIterator<QString> itr (list2);                      9
while (itr.hasNext()) {                                  10
    QString current = itr.next();
    qDebug() << "{" <<  current << "}";
}
    return 0;
}
```

1 追加操作数 1。
2 追加操作数 2。
3 追加成员函数。
4 从列表到字符串，以逗号为分隔符。
5 如果条件不满足，则 `Q_ASSERT` 会终止程序。
6 `Qt foreach` 循环——类似于 Perl/Python 和 Java 1.5 风格的 `for` 循环。
7 C++ STL 风格的迭代。
8 指针风格的解引用。
9 Java 1.2 风格的迭代器。
10 元素之间的 Java 迭代器指针。

以下是示例 4.1 的输出。

```
src/containers/lists> ./lists
The Spring months are:  "March, April, May"
"December, January, February, March, April, May, June, July, August, September,
October, November"
" [December, January, February] "
" [March, April, May] "
" [June, July, August] "
" [September, October, November] "
[[ "December, January, February" ]]
[[ "March, April, May" ]]
[[ "June, July, August" ]]
[[ "September, October, November" ]]
{ "December" }
{ "January" }
{ "February" }
{ "March" }
{ "April" }
{ "May" }
{ "June" }
{ "July" }
{ "August" }
{ "September" }
{ "October" }
{ "November" }
/src/containers/lists>
```

Qt 试图满足具有各种习惯和编程风格的程序员。例如，QList::Iterator 只不过是 QList::iterator 的一种 typedef 定义(别名)，它们提供了引用 STL 风格的 iterator 类的两种不同途径。QListIterator 类和 QMutableListIterator 类提供 Java 风格的迭代器，指向列表元素之间，并且可以用 previous() 和 next() 访问特定的元素。

4.2.2 QDir，QFileInfo 和 QDirIterator

目录(有时称为文件夹)，是文件的容器。由于目录也可以包含其他的目录，所以它天生就是一种树结构。目录也可以包含符号链接(称为 symlink)，指向另外的文件或者目录。对于处理文件或者目录的大多数操作而言，可以使用符号链接而不是文件名称或者路径名称。

Qt 提供了几种独立于平台的方法来遍历目录树。利用 QDir 类和 QFileInfo 类，可以获得目录的内容列表以及关于每一个项的信息。示例 4.2 中给出了一个递归函数，它使用这两个类来访问目录中所选择的项。它能够判断这个项是文件、目录还是符号链接，并能够根据参数的选择适当地处理这个项。第一个参数确定了要遍历的目录，第二个参数决定函数是否应递归向下地进入该目录下找到的任何子目录中，第三个参数决定函数是否应处理该目录下找到的任何符号链接。这个特定函数的主要作用是找到 MP3 文件并将它的路径添加到列表中。

示例 4.2 src/iteration/whyiterator/recurseadddir.cpp

```
[ . . . . ]
void recurseAddDir(QDir d, bool recursive=true, bool symlinks=false ) {
    d.setSorting( QDir::Name );
    QDir::Filters df = QDir::Files | QDir::NoDotAndDotDot;
```

```
        if (recursive) df |= QDir::Dirs;
        if (not symlinks) df |= QDir::NoSymLinks;
        QStringList qsl = d.entryList(df, QDir::Name | QDir::DirsFirst);
        foreach (const QString &entry, qsl) {
            QFileInfo finfo(d, entry);
            if ( finfo.isDir() ) {
                QDir sd(finfo.absoluteFilePath());
                recurseAddDir(sd);
            } else {
                if (finfo.completeSuffix()=="mp3")
                    addMp3File(finfo.absoluteFilePath());          1
            }
        }
    }
    [ . . . . ]
```

1 非复用的部分。

示例 4.3 中列出的应用复用了 QDirIterator，以更少的行完成了示例 4.2 中同样的任务。

示例 4.3 src/iteration/diriterator/diriterator.cpp

```
[ . . . . ]
int main (int argc, char* argv[]) {
    QCoreApplication app(argc, argv);
    QDir dir = QDir::current();
    if (app.arguments().size() > 1) {
        dir = app.arguments()[1];
    }
    if (!dir.exists()) {
        cerr << dir.path() << " does not exist!" << endl;
        usage();
        return -1;
    }
    QDirIterator qdi(dir.absolutePath(),
            QStringList() << "*.mp3",
            QDir::NoSymLinks | QDir::Files,
            QDirIterator::Subdirectories );
    while (qdi.hasNext()) {
        addMp3File(qdi.next());
    }
}
[ . . . . ]
```

这两个应用有一个重要的差别。示例 4.2 中，对特定于工程的 addMp3File() 函数的调用，发生在 recurseAddDir() 函数的定义之内，recurseAddDir() 函数管理迭代，这严重地限制了函数的复用性。示例 4.3 中，用 QDirIterator 管理迭代。对 addMp3File() 函数的调用发生在 QDirIterator 的客户代码中，即 main() 中，因此对这个类的复用性没有影响。在合适的地方使用迭代器模式，可以使代码更简单、更容易复用。

4.3 关系

既然某种类型的两个对象之间可以进行一对一或者一对多的通信,就可以在 UML 类框图中用各种连接线来描述对象之间的关系。

图 4.1 中，连接两个类之间的线描述了它们之间的一种特殊关系。

根据这个框图所描述的情况，一个 Employer 对象可以有许多 Person 实例。关系的两端可以用说明符指定：Employer 端的“1”和 Person 端的“*”。“*”遵从它在正则表达式中的定义，即表示“0 个或者多个”(参见 14.3 节)。

回顾图 2.6 可知，当从 Employer 的角度看待公司时，就会产生另外一组关系。图 4.2 中给出了三种关系。

图 4.1 一对多关系

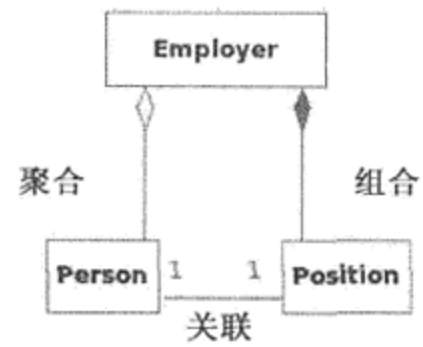


图 4.2 从 Employer 的角度看待公司时得到的关系

1. Employer 与 PositionPosition 存在组合关系(实心菱形)。它表明 Employer“拥有”或者“管理”Position，而如果没有 Employer 的话，就不应当存在 Position。
2. 从 Employer 到它的雇员存在聚合关系(空心菱形)。Employer 对应一组 Person，并让他们在工作时间进行某项工作。在聚合关系中，位于两端的对象的生命周期彼此不相关。
3. 在 Person 与 Position 之间存在一种双向关联关系。关联是一种双向关系，它不必指定对象的实现、对象之间的所属关系或者管理关系。Person 中可能并不存在指向它的 Employer 的实际指针，因为 Employer 可以通过进入 Position 并调用 Position::getEmployer()计算得到。尽管这个框图中显示了 Person 与 Position 之间存在一对一关系，但是一个 Person 可以具有多个 Position，或者一个 Employer 可以对同一个 Position 雇佣多个 Person。如果希望描述一个能够处理这种情况的系统，则它们之间必须是多对多的关联关系。

4.3.1 关系小结

前面已经讲解了三种关系：

- 关联(只用于导航性)
- 聚合(无管理的包含关系)
- 组合(带管理的包含关系)

组合关系是一种强关系，其中还描述了父-子关系和包含(托管)容器的关系。此外，每一种关系都可以具有如下的属性。

- 基数——可以是一对一、一对多或者多对多。线的一端旁边的数字((1, 1..5, *)经常用来指定这个基数。
- 导航性——可以是单向的或者双向的。单向关系在连接两个类的线上可以有箭头而没有菱形。

图 4.3 中给出了一个示例，它使用了包含关系和单向关系。

这个框图中，Book 是一个 Page 指针的(托管)容器，但是 Page 之间有自己的导航关系。也许，读者(或者浏览器)能够从这种直接导航链接中获得好处，可以进入相邻的页面或者进入目录。它们都是单向关系，所以能够加上三个自指向的箭头而框图不会出错。

因为没有对 m_pages 关系给出逆向的关系，所以它描述的是一种单向关系。它也可以是一种双向包含关系。对 Page 而言，如果需要导航到它所包含的 Book 对象，则需要在框图中将这种关系描述得更清楚一些。方法可以是标记关系的另一端，或者在 Page 类中添加一个 m_Book 属性。

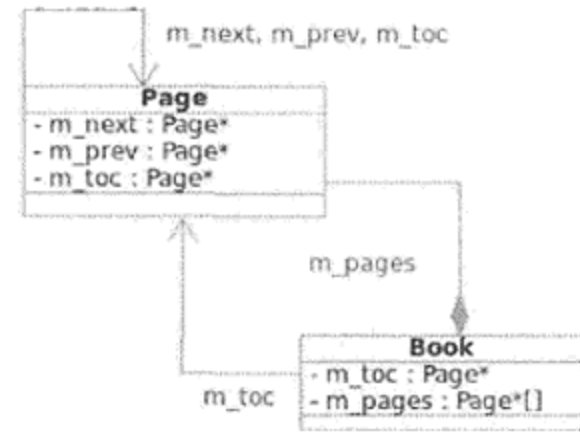


图 4.3　Book 与 Page 的关系

4.4　练习：关系

1. 这些练习中，需根据图 4.2 实现某些关系。图 4.2 中的框图仅仅是个开始。为了完成这项任务，需向类添加一些成员。
   a. 实现 Employer::findJobs()函数，返回全部开放职位(Position)的列表。
   b. 实现调用 Employer::hire()函数的 Person::apply(Position*)函数，并且如果成功的话返回与 hire()函数相同的结果。
   c. 为了增加趣味性，让 Employer::hire()函数有一半的次数随机返回 false。
   d. 对于返回有关雇佣信息的那些 Person 方法，一定要处理 Person 还没有被雇佣的情况，并返回一些有意义的信息。
   e. 在客户代码中创建更多用于测试的 Employer 对象(Galactic Empire 和 Rebel Forces)、Person 对象(Darth Vader, C3PO 和 Data)以及 Position 对象(Tie Fighter Pilot, Protocol Android 和 Captain)。
   f. 在 Person 类中定义一个 QList<Person*> s_unemploymentLine 对象，确保所有还没有工作的人都位于其中。
   g. 编造一些有趣的工作情景，运行这个应用以判断它们是否成功。
2. a. 图 4.4 中描述了一个 Contact 系统的模型[①]。ContactList 可以派生自或者复用任何 Qt 容器，只要该容器支持下面列出的操作。
   - getPhoneList(int category)接收的值将与 Contact 的类别成员比较，以此来达到选择的目的。该函数返回一个 QStringList，其中包含选中的 Contact、姓名以及电话号码，中间使用制表符\t 分隔。

① 模型与视图将在第 13 章探讨。这里，将管理应用信息(不包括信息的获取、显示和传送)的数据结构称为模型。

- getMailingList()选择机制与 getPhoneList 相似，它返回一个包含地址标号数据的 QStringList。

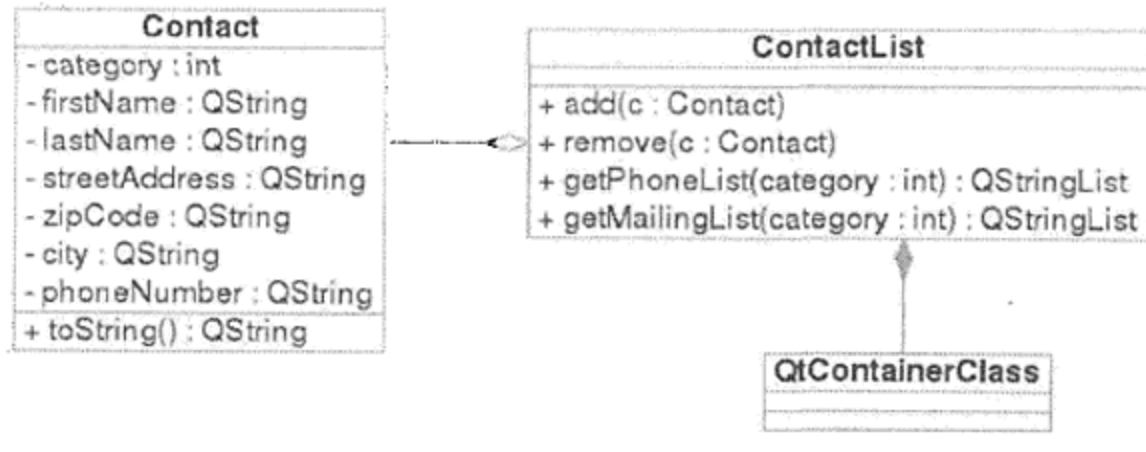


图 4.4 Contact 的 UML 框图

b. 编写一个 ContactFactory 类，产生随机的 Contact 对象。

示例 4.4 中给出了许多提示。

示例 4.4 src/containers/contact/testdriver.cpp

```
[ . . . . ]

void createRandomContacts(ContactList& cl, int n=10) {
    static ContactFactory cf;
    for (int i=0; i<n; ++i) {
        cf >> cl;                                                    1
    }
}
```

1 将联系人信息添加到联系人列表中。

有许多种途径能够产生随机的名称/地址。一种方法是使用 ContactFactory 创建常见的名、姓、街道名、城市名等[①]。在需要产生 Contact 对象时，可以从列表中随机选取一个元素，然后添加随机生成的地址编号、邮编等。1.13.1 节中演示过 random()函数的用法。

c. 编写客户代码，测试这些类。特别地，客户代码应当随机生成某些联系人信息。之后，应测试两种查询方法，即 getPhoneList()和 getMailingList()，确保它们返回正确的子列表。在标准输出终端输出原始列表以及查询结果。通过列举原始 ContactList 中与查询结果对应元素的数量来总结查询的结果。

4.5 复习题

1. 命名三个可能需要使用 QStringList 的应用程序。
2. 如何将 QStringList 转换成 QString？为什么要这样做？
3. 如何将 QString 转换成 QStringList？为什么要这样做？
4. 什么是迭代器模式？给出三个示例。

① 垃圾邮件是联系人信息的一个绝佳来源。只需查看垃圾邮件文件夹，即可从邮件标题或者消息体中获得这些信息。在 Dist 目录下已经提供了一个垃圾邮件联系人信息的初始列表。

5. 画一个 UML 框图，它具有三个或者多个类，并确保框图体现了如下这些不同类型的关系：

   - 聚合和组合
   - 一对一和一对多
   - 单向和双向

   这些类应反映现实世界的情况，且关系应是真实的。用几段话解释框图中每种关系存在的理由。
6. 组合与聚合有何不同？
7. 阅读 QList 的 API 文档，找出向列表添加元素的三种不同方式。
8. 给出 QStringList 中具有而 QList 中没有的三个方法。
9. QList 中为什么具有 iterator 和 Iterator？二者有什么不同？
10. 将 Qt 或者 STL 容器类与数组进行比较，探讨它们的优缺点。
11. 分析 QList 声明的语法。位于尖括号中的是什么？为什么它是必须的？
12. 为什么 Qt 支持这么多不同类型的迭代器？
13. QFile 与 QFileInfo 有何不同？

第5章 函 数

本章将探讨函数重载、函数调用解析、默认/可选实参、临时变量、引用参数、返回值以及 inline 函数。

C++中的函数与其他编程语言中的函数和例程类似。但是，C++函数还支持在某些语言中没有的许多特性，因此这里有必要探讨它们。

5.1 函数重载

正如 1.5 节中指出的，函数的签名由名称和参数表组成。C++中，函数的返回类型不属于函数签名。

我们已经看到，C++中允许重载函数的名称。前面说过，如果在给定的作用域范围内一个函数的名称具有多种含义，那么它就是被重载的。如果两个或者多个函数在某个给定的作用域内的名称相同但它们的签名不同，就称此函数名称被重载了。1.5 节中还说过，如果位于同一作用域中的两个函数具有相同的签名而返回类型不同，则会导致错误。

函数调用解析

如果在某个作用域范围内调用了某个重载函数，则 C++编译器会根据实参来决定应该调用函数的哪一个版本。为此，必须使实参的个数和类型与重载函数的签名完全匹配。为了找出对应的签名，*Type* 和 *Type&*参数必须匹配。

以下是编译器确定应该调用哪一个重载函数的步骤。

1. 如果存在一个完全匹配的函数，则调用此函数。
2. 否则，通过标准的类型提升转换(参见 19.6 节)来进行匹配。
3. 否则，通过转换构造函数或者转换运算符(参见 2.12 节)来进行匹配。
4. 否则，判断是否可以通过省略号(…)进行匹配(参见 5.11 节)。
5. 否则，编译器报错。

示例 5.1 中展示的类有 6 个成员函数，每一个都有不同的签名。需记住的是，每一个成员函数都具有一个额外的隐式参数：this。在参数表之后的关键字 const，能够保护主对象(指向 this 的对象)不随函数的行为而改变，因此也是函数签名的一部分。

示例 5.1 src/functions/function-call.cpp

```
[ . . . . ]

class SignatureDemo {
public:
    SignatureDemo(int val) : m_Val(val) {}
    void demo(int n)
        {cout << ++m_Val << "\tdemo(int)" << endl;}
```

```
    void demo(int n) const
        {cout << m_Val << "\tdemo(int) const" << endl;}
/*  void demo(const int& n)
      {cout << ++m_Val << "\tdemo(int&)" << endl;}  */
    void demo(short s)
        {cout << ++m_Val << "\tdemo(short)" << endl;}
    void demo(float f)
        {cout << ++m_Val << "\tdemo(float)" << endl;}
    void demo(float f) const
        {cout << m_Val << "\tdemo(float) const" << endl;}
    void demo(double d)
        {cout << ++m_Val << "\tdemo(double)" << endl;}
private:
    int m_Val;
};
```

1 对 const 重载。

2 与前一个函数冲突。

示例 5.2 中包含的客户代码测试了 SignatureDemo 中的那些重载函数。

示例 5.2 src/functions/function-call.cpp

```
[ . . . . ]

int main() {
    SignatureDemo sd(5);
    const SignatureDemo csd(17);
    sd.demo(2);
    csd.demo(2);                                                    1
    int i = 3;
    sd.demo(i);
    short s = 5;
    sd.demo(s);
    csd.demo(s);                                                    2
    sd.demo(2.3);                                                   3
    float f(4.5);
    sd.demo(f);
    csd.demo(f);
    //csd.demo(4.5);
    return 0;
}
```

1 调用 const 版本。

2 不能调用非 const short 版本，所以需要进行 int 类型提升，以调用 const int 版本。

3 这里为 double，不是 float。

其运行结果应与下面的类似。

```
6       demo(int)
17      demo(int) const
7       demo(int)
8       demo(short)
17      demo(int) const
```

```
9       demo(double)
10      demo(float)
17      demo(float) const
```


5.1.1 练习：重载函数

1. 体验示例 5.1。首先将第三个成员函数的注释符号删除并编译。
2. 将 main() 结尾处前面的那一行的注释符号删除：

```
//  csd.demo(4.5);
```

会发生什么？对出现的错误消息给出解释。

3. 当链编这个应用时，注意编译器对于未使用的参数给出的警告。将编译器提示的函数首部中未使用的参数名称全部删除(不要删除参数类型)，然后重新链编，观察编译器的输出。如果想告知编译器不是在故意使用某些参数，则这就是一种好的做法。15.2 节中可以看到能够更方便地利用这种技术。
4. 添加其他的函数调用以及其他版本的 demo() 函数，并解释结果。

5.2 可选实参

默认(可选)实参

函数参数可以有默认值，此时它们就是可选的。可选实参的默认值可以是常量表达式或者不涉及局部变量的表达式。

拥有默认实参的参数必须是参数表中最靠右(最后面)的参数。具有默认值的最靠右实参在函数调用时可以忽略，而此时这些参数就会使用默认值进行初始化。

从函数的角度来看，如果调用时缺少一个实参，那么此实参必须对应于参数表中的最后一个参数。如果调用时缺少两个实参，则它们必须正好是参数表中的最后两个参数(以此类推)。

因为可选实参指示符应用在函数的接口上，所以它处于函数声明中；如果函数声明存放在一个单独的头文件中，则可选实参指示符不会出现在函数定义中。带有可选实参的函数可以有多种调用方式。如果所有的函数实参都是可选的，则不用任何实参就可以调用这个函数。声明有 n 个可选实参的函数，可以看成是声明了 $n+1$ 个函数的简化版本，其中每一个函数都对应此函数的一种调用方式。

示例 5.3 中，Date 类的构造函数有三个参数，每一个都是可选的，其默认值都为 0。

示例 5.3 src/functions/date.h

```
[ . . . . ]
class Date {
public:
    Date(int d = 0, int m = 0, int y = 0);
    void display(bool eoln = true) const;
private:
    int m_Day, m_Month, m_Year;
};
[ . . . . ]
```

示例 5.4 中的构造函数定义看似普通，它们无须再指定可选实参。如果函数调用时任何实参的实际值都为 0，则它会被从当前日期中获取的一个有意义的值替换。

示例 5.4　src/functions/date.cpp

```
#include <QDate>
#include "date.h"
#include <iostream>

Date::Date(int d , int m , int y )
: m_Day(d), m_Month(m), m_Year(y) {

    static QDate currentDate = QDate::currentDate();          1

    if (m_Day == 0) m_Day = currentDate.day();
    if (m_Month == 0) m_Month = currentDate.month();
    if (m_Year == 0) m_Year = currentDate.year();
}

void Date::display(bool eoln) const {
    using namespace std;
    cout << m_Year << "/" << m_Month << '/' << m_Day;
    if (eoln)
        cout << endl;
}
```

1 此处使用 Qt 的 `QDate` 类的目的是取得当前日期。

示例 5.5　src/functions/date-test.cpp

```
#include "date.h"
#include <iostream>

int main() {
    using namespace std;
    Date d1;
    Date d2(15);
    Date d3(23, 8);
    Date d4(19, 11, 2003);

    d1.display(false);
    cout << '\t';
    d2.display();
    d3.display(false);
    cout << '\t';
    d4.display();
    return 0;
}
```

示例 5.5 表明，实际上是通过定义默认值重载了函数。函数的不同版本执行相同的代码，只不过是靠后的参数接收了不同的值。

如果在 2011 年 5 月 14 日运行这个程序，则其运行结果如下。

```
src/functions> qmake
src/functions> make
[ compiler linker messages ]
src/functions> ./functions
2011/5/14       2011/5/15
2011/8/23       2003/11/19
src/functions>
```

5.3 运算符重载

C++使用关键字 operator 为运算符赋予新的含义，比如运算符+, -, =, *和&。为运算符增加新的含义，是重载的一种特殊形式。运算符重载提供了一种更紧致的函数调用语法，可以在很大程度上提高代码的可读性(假设运算符都按照常规理解的意义来进行操作)。

可以重载 C++中几乎所有已存在的运算符。例如，假设希望定义一个名称为 Complex 的类来表示复数[①]。

为了指定如何用这些对象进行基本的算术运算，可以重载四则算术运算符。当然，也可以重载插入运算符<<，以提高输出语句的可读性。

示例 5.6 中展示了一个包含成员运算符和非成员运算符的类的定义。

示例 5.6 src/complex/complex.h

```
#include <iostream>
using namespace std;

class Complex {
    // binary nonmember friend function declarations
    friend ostream& operator<<(ostream& out, const Complex& c);
    friend Complex operator-(const Complex& c1, const Complex & c2);
    friend Complex operator*(const Complex& c1, const Complex & c2);
    friend Complex operator/(const Complex& c1, const Complex & c2);

    public:
    Complex(double re = 0.0, double im = 0.0);                    1

    // binary member function operators
    Complex& operator+= (const Complex& c);
    Complex& operator-= (const Complex& c);


    Complex operator+(const Complex & c2);                        2

private:
    double m_Re, m_Im;
};
```

1 默认构造函数和转换构造函数。

2 这里应该像其他的非可变运算符一样，是一个非成员友元函数。

示例 5.6 中声明的运算符都是二元的(接收两个操作数)。对成员函数而言，却只有一个形式参数，因为第一个(左边的)操作数都是隐式的：*this。示例 5.7 中给出了这些成员运算符的定义。

---

① 复数的形式为：$a + bi$，其中 a 和 b 为实数，i 表示–1 的平方根。由于复数中存在 $b = 0$ 的情况，所以实数是复数的一个子集。

复数最初是为了描述类似下面这样的方程的解而引入的：

$x^2 - 6x + 25 = 0$

利用二次式，可以很容易地知道这个方程的根为 $3 + 4i$ 和 $3 - 4i$。

示例 5.7 src/complex/complex.cpp

```
[ . . . . ]

Complex& Complex::operator+=(const Complex& c) {
    m_Re += c.m_Re;
    m_Im += c.m_Im;
    return *this;
}

Complex Complex::operator+(const Complex& c2) {
    return Complex(m_Re + c2.m_Re, m_Im + c2.m_Im);
}

Complex& Complex::operator-=(const Complex& c) {
    m_Re -= c.m_Re;
    m_Im -= c.m_Im;
    return *this;
}
```

示例 5.8 中给出了非成员友元函数的定义，定义它们时就像常规的全局函数那样。

示例 5.8 src/complex/complex.cpp

```
[ . . . . ]


ostream& operator<<(ostream& out, const Complex& c) {
    out << '(' << c.m_Re << ',' << c.m_Im << ')' ;
    return out;
}

Complex operator-(const Complex& c1, const Complex& c2) {
    return Complex(c1.m_Re - c2.m_Re, c1.m_Im - c2.m_Im);
}
```

前面已经讲解了定义 C++代码中四则代数运算的数学规则，它们的细节被封装并隐藏了，这样客户代码就不必处理它们。示例 5.9 中给出的客户代码演示并测试了这个 Complex 类。

示例 5.9 src/complex/complex-test.cpp

```
#include "complex.h"
#include <iostream>

int main() {
    using namespace std;
    Complex c1(3.4, 5.6);
    Complex c2(7.8, 1.2);

    cout << c1 << " + " << c2 << " = " << c1 + c2 << endl;
    cout << c1 << " - " << c2 << " = " << c1 - c2 << endl;
    Complex c3 = c1 * c2;
    cout << c1 << " * " << c2 << " = " << c3 << endl;
    cout << c3 << " / " << c2 << " = " << c3 / c2 << endl;
    cout << c3 << " / " << c1 << " = " << c3 / c1 << endl;

    return 0;
}
```

以下是示例 5.9 的输出。

```
(3.4,5.6) + (7.8,1.2) = (11.2,6.8)
(3.4,5.6) - (7.8,1.2) = (-4.4,4.4)
(3.4,5.6) * (7.8,1.2) = (19.8,47.76)
(19.8,47.76) / (7.8,1.2) = (3.4,5.6)
(19.8,47.76) / (3.4,5.6) = (7.8,1.2)
```


成员运算符与全局运算符的比较

前面已经看到，可以将运算符重载为成员函数或者全局函数。首先应注意，它们的主要差异是调用它们的方式。特别地，成员函数运算符要求有一个用作左操作数的对象，而全局函数允许对任何一个操作数进行某种类型转换。

示例 5.10 中给出了为什么 `Complex::operator+()` 更适合作为非成员函数的理由。

示例 5.10 src/complex/complex-conversions.cpp

```
#include "complex.h"

int main() {
    Complex c1 (4.5, 1.2);
    Complex c2 (3.6, 1.5);

    Complex c3 = c1 + c2;
    Complex c4 = c3 + 1.4;                    1
    Complex c5 = 8.0 - c4;                    2
    Complex c6 = 1.2 + c4;                    3
}
```

1 提升右操作数。

2 提升左操作数。

3 错误：左操作数没有针对成员运算符而提升。

运算符重载存在一些限制。只有内置的运算符才能被重载，因此无法对类似于“$”、“@”、“`”等未拥有运算符定义的符号重新进行定义。此外，尽管可以给内置运算符赋予新的定义，但是它们的结合性和优先级无法改变。

可以重载除下面这些运算符以外的全部一元或者二元内置运算符：

- 三元条件运算符 `testExpr ? valueIfTrue : valueIfFalse`
- 作用域解析运算符 `::`
- 成员选择运算符 `.` 和 `.*`

提示

有一种方法可以帮助记住哪一个运算符可以被重载。如果符号中的某个位置有一个点(.)，则它可能就不能被重载。

注意

重载逗号运算符是允许的，但是建议不要这样做，除非你是 C++行家。

19.1 节中给出了运算符以及它们的特性的一个完整列表。

注意

可以为内置运算符赋予一种新的含义，这样它就可以使用具有不同类型的操作数。但是，无法改变内置运算符的结合性和优先级。

5.3.1 练习：运算符重载

1. 继续开发 Fraction 类，添加重载的加、减、乘、除以及各种比较运算符。在每一种情况下，参数都应声明为 const Fraction&。编写客户代码，测试这些新的运算符。
2. 如果要使其真正有用，则 Fraction 对象应该能够与其他类型的数交互。扩展 Fraction 类的定义，使得上一题中的运算符也能够用于 int 类型和 double 类型。应能够清楚地对表达式进行求值，例如 frac + num。当 frac 为分数而 num 是 int 类型时，应该如何处理表达式 num + frac? 编写客户代码，测试这些新的函数。
3. 为 Complex 类添加算术运算符和比较运算符。编写客户代码，测试这个扩展的类。

5.4 按值传递参数

默认情况下，C++参数是按值传递的。当调用函数时，会生成每一个实参对象的一个临时(局部)副本并放入程序栈中。只有这些临时副本在函数内部进行操作，而调用块中的实参对象不会受这些操作的影响。当函数返回时，这些临时的栈变量就会被销毁。可以通过一种有效的方法来理解值参数：值参数仅仅是用函数调用时指定的对应实参对象进行初始化的局部变量。如示例 5.11 所演示的那样。

示例 5.11 src/functions/summit.cpp

```
#include <iostream>

int sumit(int num) {
    int sum = 0;
    for (; num ; --num)                    1
        sum += num;
    return sum;
}

int main() {
    using namespace std;
    int n = 10;
    cout << n  << endl;
    cout << sumit(n) << endl;
    cout << n << endl;                     2
    return 0;
}
```

1 参数值减少到 0。

2 看一看 sumit()对 n 做了什么?

输出如下所示。

```
10
55
10
```

如果将指针传递给函数，那么这个指针的一个临时副本会被放入栈中。对这个指针的任何改变都不会对调用块中的指针产生影响。例如，临时指针可能会被赋予一个不同的值(参见示例 5.12)。

示例 5.12 src/functions/pointerparam.cpp

```
#include <iostream>
using namespace std;

void messAround(int* ptr) {
    *ptr = 34;                                          1
    ptr = 0;                                            2
}


int main() {
    int n(12);                                                  3
    int* pn(&n);                                                4
    cout << "n = " << n << "\tpn = " << pn << endl;
    messAround(pn);                                             5
    cout << "n = " << n << "\tpn = " << pn << endl;
    return 0;
}
```

1 改变所指向的值。

2 改变由 ptr 保存的地址。更好的做法是不解引用它。

3 初始化一个 int 值。

4 初始化指向 n 的指针。

5 看一看 messAround()改变了什么?

输出如下所示。

```
n = 12  pn = 0xbffff524
n = 34  pn = 0xbffff524
```

输出结果中显示了指针 pn 的十六进制值以及 n 的十进制值，这样对函数动作产生的改变就不存在任何疑问了。

注意

总结如下:

- 当将对象按值传递给函数时，会产生该对象的一个副本。
- 这个副本被函数当成局部变量。
- 函数返回时，会销毁这个副本。

5.5 按引用传递参数

不应该将大型对象或者具有大量复制构造函数的对象进行按值传递，因为副本的创建会消耗大量不必要的机器周期和时间。在 C 语言中，可以通过指针来传递对象，以避免这类对象的复制。但是,使用指针的语法与使用对象的语法有所不同。此外，指针即使是偶尔的误用都可能引起数据的崩溃，从而导致难以发现和修复的运行时错误。在 C++(以及 C99)中，可以按引用传递参数，这种机制提供了与用指针传递参数相同的性能。对于对象而言，可以使用点运算符(.)来间接地访问成员。

引用参数也是参数，只不过它是其他对象的一个别名。为了将一个参数声明为引用参数，只须在类型名称与参数名称之间添加一个和符号(&)。

函数的引用参数使用函数调用时传递的实际实参进行初始化。正如其他所有的引用那样，这个实参必须是一个非 const 的左值。在函数中对非 const 引用参数进行的改变，将导致用来初始化此参数的实参对象的相应变化。这一特性可用于定义函数，通常情况下函数最多返回一个值，而在多个实参对象中的变化能够有效地使函数返回多个值。示例 5.13 中展示了如何针对整型变量使用引用参数。

示例 5.13 src/reference/swap.cpp

```
#include <iostream>
using namespace std;

void swap(int& a, int& b) {
    int temp = a;
    cout << "Inside the swap() function:\n"
         << "address of a: " << &a
         << "\taddress of b: " << &b
         << "\naddress of temp: " << &temp << endl;
    a = b;
    b = temp;
}

int main() {
    int n1 = 25;
    int n2 = 38;
    int n3 = 71;
    int n4 = 82;
    cout << "Initial values:\n"
         << "address of n1: " << &n1
         << "\taddress of n2: " << &n2
         << "\nvalue of n1: " << n1
         << "\t\t\tvalue of n2: " << n2
         << "\naddress of n3: " << &n3
         << "\taddress of n4: " << &n4
         << "\nvalue of n3: " << n3
         << "\t\t\tvalue of n4: " << n4
         << "\nMaking the first call to swap()" << endl;
```

```
    swap(n1,n2);
    cout << "After the first call to swap():\n"
         << "address of n1: " << &n1
         << "\taddress of n2: " << &n2
         << "\nvalue of n1: " << n1
         << "\t\t\tvalue of n2: " << n2
         << "\nMaking the second call to swap()" << endl;
    swap(n3,n4);
    cout << "After the second call to swap():\n"
         << "address of n3: " << &n3
         << "\taddress of n4: " << &n4
         << "\nvalue of n3: " << n3
         << "\tvalue of n4: " << n4 << endl;
    return 0;
}
```

这个程序中包含有许多额外的输出语句，以帮助跟踪重要变量的地址。

```
Initial values:
address of n1: 0xbffff3b4        address of n2: 0xbffff3b0
value of n1: 25                  value of n2: 38
address of n3: 0xbffff3ac        address of n4: 0xbffff3a8
value of n3: 71                  value of n4: 82
```

开始时，程序栈可能与图 5.1 相似。随着程序的执行，输出可能是这样的：

```
Making the first call to swap()
Inside the swap() function:
address of a: 0xbffff3b4         address of b: 0xbffff3b0
address of temp: 0xbffff394
```

当将引用传递给函数时，压入栈的值是地址而不是右值。本质上，按引用传递非常类似于按指针传递。现在，程序栈可能与图 5.2 相似。

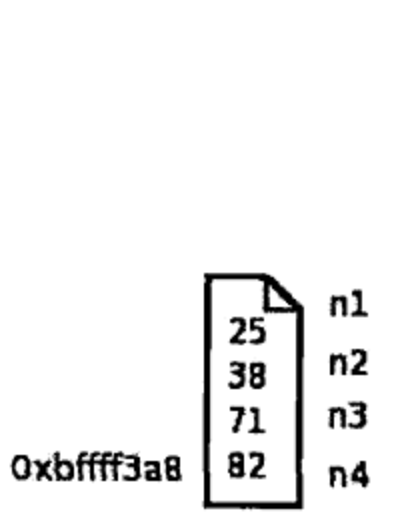


图 5.1 执行第一个 swap() 之前的程序栈

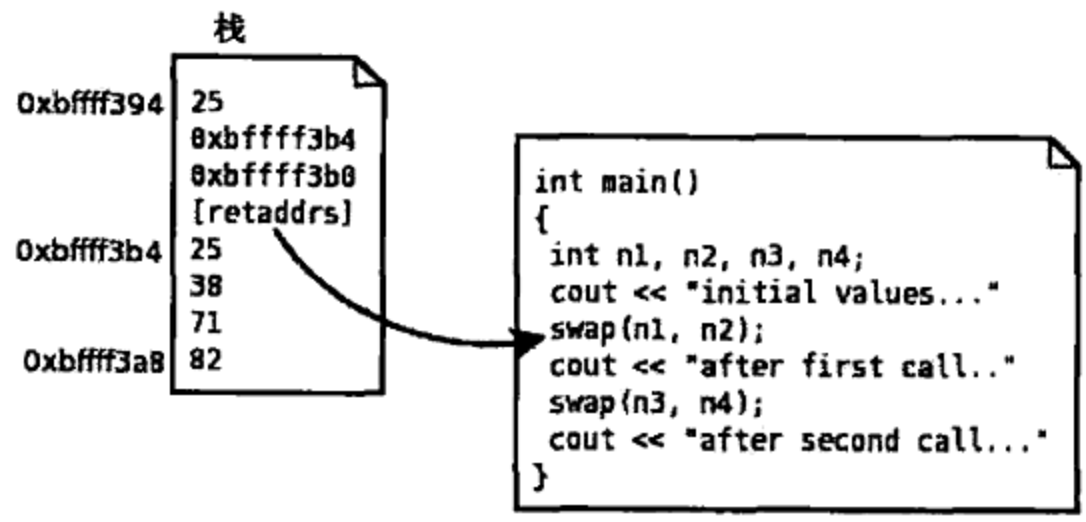


图 5.2 执行第一个 swap() 时的程序栈

```
After the first call to swap():
address of n1: 0xbffff3b4        address of n2: 0xbffff3b0
value of n1: 38                  value of n2: 25
Making the second call to swap()
Inside the swap() function:
```

图 5.3 给出的是上面的行输出之后的栈状态。

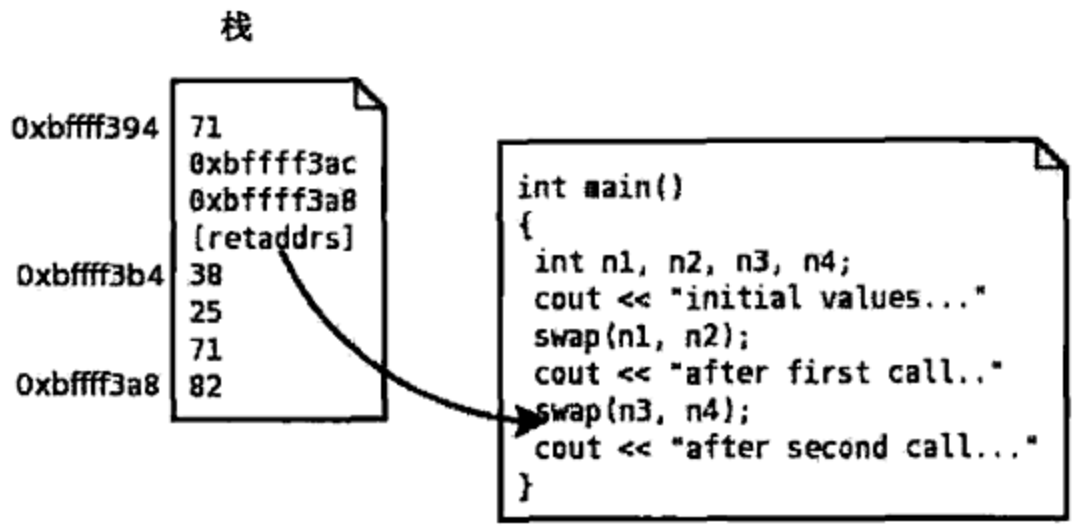
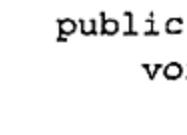


图 5.3　执行第二个 swap()时的程序栈

```
address of a: 0xbffff3ac        address of b: 0xbffff3a8
address of temp: 0xbffff394
After the second call to swap():
address of n3: 0xbffff3ac       address of n4: 0xbffff3a8
value of n3: 82 value of n4: 71
```

第一次调用时，swap()函数实际上只作用于 n1 和 n2，第二次调用时，作用的是 n3 和 n4。

按引用传递的语法提供了按指针传递的一种替代方式。本质上，按引用传递就是用指针实现的，不会复制值。二者的主要区别是：对于指针，必须解引用它；对于引用，可以用访问被引用实体同样的方式访问按引用传递的对象。



按指针传递还是按引用传递

如果可以选择，则通常更倾向于使用引用而不是指针，因为这样可以降低程序偶然发生内存崩溃的概率。只有在管理那些需要对指针进行操作的对象时(创建、销毁或者添加到一个托管容器中)，才会选择使用指针，并且，通常可以将这些例程封装为成员函数。

5.6　const 引用

将一个引用参数声明为 const，就是通知编译器，要确保函数不会试图改变这个对象。对大于指针的对象，const 引用要比值参数高效，因为不需要进行数据复制。示例 5.14 中包含了三个函数，每一个函数都以不同的方式接收参数。

示例 5.14　src/const/reference/constref.cpp

```
#include <QString>

class Person {

public:
    void setNameV( QString newName) {
        newName += " Smith";                                    1
        m_Name = newName;
    }

    void setNameCR( const QString& newName) {
//      newName += " Python";                                   2
```

```
        m_Name = newName;
    }
    void setNameR( QString& newName) {
        newName += " Dobbs";                                    3
        m_Name = newName;
    }
private:
    QString m_Name;
};

#include <QDebug>

int main() {
    Person p;
    QString name("Bob");
    p.setNameCR(name);                                          4
//  p.setNameR("Monty");                                        5
    p.setNameCR("Monty");                                       6
    p.setNameV("Connie");                                       7
    p.setNameR(name);                                           8
    qDebug() << name;
}
```

1 改变一个将被销毁的临时变量。

2 错误：无法转换为 const&。

3 改变原始的 QString。

4 没有创建临时变量。

5 错误：不能转换为 QString&。

6 将一个 char*变量转换为临时变量，并获得由 const 引用传递的值。

7 创建第一个临时 QString，将 char*转换成 QString。当按值传递时，创建第二个临时变量。

8 没有创建临时变量。

5.7 函数返回值

当执行完给它设计的任务时，有些函数会返回一个值。为返回对象临时准备的场所通常为一个寄存器(如果可以容纳的话)，但是有时也可以是栈上分配的一个对象。当执行 return 语句时，会初始化临时的返回对象，且其存在时间只为满足包含该函数调用的任何表达式的使用。对函数而言，这个对象副本通常是局部的，或者是在 return 语句的表达式中构造的一个对象。

5.8 从函数返回引用

有时候将函数设计成返回引用是非常有用的。例如，可以将多个操作“链”起来：

```
cout << thing1 << thing2 << thing3 ... ;
```

返回引用(尤其是*this)通常用来给成员函数提供左值行为。

采用引用参数，就可以通过将对象的别名指定成 const 来保护返回的引用。

示例 5.15 中展示了返回引用的本质。

示例 5.15　src/reference/maxi.cpp

```
#include <iostream>
using namespace std;
int& maxi(int& x, int& y) {

    return (x > y) ? x : y;
}

int main() {
    int a = 10, b = 20;
    maxi(a,b) = 5;                    1
    maxi(a,b) += 6;                   2
    ++maxi(a, b) ;                    3
    cout << a << '\t' << b << endl;
    return 0;
}
```

1 给 b 赋值 5。

2 给 a 增加 6，a 现在的值为 16。

3 给 a 增加 1。

输出如下所示。

```
17    5
```

正如在 main() 函数中看到的那样，函数 maxi() 的引用返回值使表达式 maxi(a,b) 成为了一个可修改的左值。

警告

要小心，不要让函数返回一个指向临时(局部)对象的引用。只需稍加思考即可知晓这种限制的缘由：当函数返回时，所有的局部变量都销毁了。

```
int& max(int i,int j) {
    int retval = i > j ? i : j;

    return retval;
}
```

如果足够幸运，上述代码会导致一个编译器警告，但还不会使编译器认为是一个错误。以下是稍新的 C++版本给出的警告信息：

```
badmax.cpp:4: warning: reference to local variable 'retval' returned
```

示例 5.16 是返回引用的好处的一个实际例子，其中定义了针对矢量的一些常见运算符。

5.9　对 const 重载

const 改变了成员函数的签名，这意味着这种函数可以对 const 进行重载。示例 5.16 是自定义矢量类的一个例子，其成员函数就进行了 const 重载。

示例 5.16 src/const/overload/constoverload.h

```
#ifndef CONSTOVERLOAD_H
#define CONSTOVERLOAD_H

#include <iostream>

class Point3 {                                                    1
 public:
    friend std::ostream& operator<<(std::ostream& out, const Point3& v);
    Point3(double x = 0, double y = 0, double z = 0);
    double& operator[](int index);
    const double& operator[](int index) const;                    2
    Point3 operator+(const Point3& v) const;
    Point3 operator-(const Point3& v) const;
    Point3 operator*(double s) const;                             3
 private:
    static const int cm_Dim = 3;
    double m_Coord[cm_Dim];
};

#endif
```

1 一个(double 类型的)三维点。

2 对 const 重载。

3 标量乘法。

示例 5.17 中给出了这些运算符函数的定义。

示例 5.17 src/const/overload/constoverload.cpp

```
[ . . . . ]
const double& Point3::operator[](int index) const {
    if ((index >= 0) && (index < cm_Dim))
        return m_Coord[index];
    else
        return zero(index);
}

double& Point3::operator[](int index) {
    if ((index >= 0) && (index < cm_Dim))
        return m_Coord[index];
    else
        return zero(index);
}

[ . . . . ]
```

两个函数体相同这一事实值得考虑。如果 index 位于范围之内，则每一个函数都返回 m_Coord[index]。二者有什么不同呢？需重点理解的是，这个运算符的非 const 版本的行为与示例 5.15 中的 maxi() 函数非常类似。

5.9.1 练习：对 const 重载

1. 示例 5.18 中，编译器能够根据对象的 const 属性区别对 operator[]的 const 和非 const 版本的调用。

示例 5.18 src/const/overload/constoverload-client.cpp

```
#include "constoverload.h"
#include <iostream>

int main( ) {
    using namespace std;
    Point3 pt1(1.2, 3.4, 5.6);
    const Point3 pt2(7.8, 9.1, 6.4);
    double d ;
    d = pt2[2];                                    1
    cout << d << endl;
    d = pt1[0];                                    2
    cout << d << endl;
    d = pt1[3];                                    3
    cout << d << endl;
    pt1[2] = 8.7;                                  4
    cout << pt1 << endl;
    //  pt2[2] = 'd';
    cout << pt2 << endl;
    return 0;
}
```

1 _____

2 _____

3 _____

4 _____

每一个注释处分别调用的是哪一个运算符？

2. 为什么最后一个赋值被注释掉了？

5.10 inline 函数

为了避免函数调用带来的开销(例如，创建包含实参副本、引用参数地址以及返回地址的栈帧)，C++允许将函数声明为 inline(内联)的。这样的声明会要求编译器将所有对该函数的调用都以函数完全展开之后的代码来替换。例如：

```
inline int max(int a, int b){
  return a > b ? a : b ;
}

int main(){
   int temp = max(3,5);
    etc....
}
```

编译器会将 max 的代码展开成如下的样子：

```
int main() {
  int temp;
  {
    int a = 3;
    int b = 5;
    temp = a > b ? a : b;
  }
  etc.......
}
```


如果要重复地调用(例如，在一个大型循环中)，则 inline 函数可极大地提高性能。inline 函数的缺点是会使编译代码变得更大，在运行时会占用更多内存。对于需调用许多次的小型函数，将其声明成 inline 对内存的影响是微小的，而潜在的性能提升收益会很大。

inline 函数是否会提升程序的性能，这个问题没有简单的答案。这一方面取决于编译器的优化设置，另一方面也取决于程序的性质。程序是否会给处理器造成很重的负荷？是否会大量使用系统内存？是否花费大量的时间与慢速设备(例如，输入/输出设备)交互？这些问题的答案将决定是否应该采用 inline 函数，在此不再深入地探讨，而是将其作为一个高级课题。可以访问 Marshall Cline 的 FAQ Lite 网站，大体了解此问题的复杂性[①]。

inline 函数与#define 宏类似，但有一个重大差异：对#define 宏的替换过程是由预处理器处理的，预处理器本质上就是一个文本编辑器。对 inline 函数的替换过程是由编译器处理的，它会执行更智能的操作，进行正确的类型检查。下一节中将更详细地探讨这种差异。

inline 函数的一些规则

- inline 函数必须在被调用之前定义(仅仅声明它是不够的)。
- 在一个源代码模块中只能有一次 inline 定义。
- 如果类成员函数的定义出现在类定义之内，则成员函数就是隐含 inline 的。

如果函数太复杂，或者编译器的选项改变了，则编译器可能会忽略 inline 指令。大多数编译器会拒绝包含如下语句的 inline 函数：

- while, for, do...while 语句
- switch 语句
- 超过一定数量的代码行

如果编译器拒绝了 inline 函数，则会将其当成常规函数，并会生成常规的函数调用。

5.10.1 inline 函数与宏扩展的比较

宏扩展是一种通过如下的预处理器指令植入代码的机制：

```
#define MACRO_ID expr
```

这不同于 inline 函数。

宏扩展不对实参进行类型检查。本质上，它是一种编辑操作：在出现 MACRO_ID 的每一个地方都用 expr 替换。在宏中必须注意圆括号的使用，以避免优先级错误。但是，圆括号

① 参见 http://www.parashift.com/c++-faq-lite/inline-functions.html。

无法解决与宏有关的全部问题，见示例 5.19。由宏引起的错误可以导致奇怪的(不明晰的)编译器错误，甚至是更危险的无效结果。示例 5.19 演示了后一种情况。

示例 5.19 src/functions/inlinetst.cpp

```
// Inline functions vs macros

#include <iostream>
#define  BADABS(X)    (((X) < 0)? -(X) : X)
#define  BADSQR(X) (X * X)
#define  BADCUBE(X) (X) * (X) * (X)

using namespace std;

inline double square(double x) {
    return x * x ;
}

inline double cube(double x) {
    return x * x * x;
}

inline int absval(int n) {
    return (n >= 0) ? n : -n;
}

int main() {
    cout << "Comparing inline and #define\n" ;
    double  t = 30.0;
    int i = 8, j = 8, k = 8, n = 8;
    cout << "\nBADSQR(t + 8) = " << BADSQR(t + 8)
            << "\nsquare(t + 8) = " << square(t + 8)
            << "\nBADCUBE(++i) = " << BADCUBE(++i)
            << "\ni = " << i
            << "\ncube(++j) = " << cube(++j)
            << "\nj = " << j
            << "\nBADABS(++k) = " << BADABS(++k)
            << "\nk = " << k
            << "\nabsval(++n) = " << absval(++n)
            << "\nn = " << n << endl;
}
```

以下是输出结果。

```
BADSQR(t + 8) = 278
square(t + 8) = 1444
BADCUBE(++i) = 1100
 i = 11
cube(++j) = 729
j = 9
BADABS(++k) = 10
k = 10
absval(++n) = 9
n = 9
```

BADSQR(t+8)的结果是错误的，因为：

```
    BADSQR(t + 8)
=   (t + 8 * t + 8)          (preprocessor)
=   (30.0 + 8 * 30.0 + 8)    (compiler)
=   (30 + 240 + 8)           (runtime)
=   278
```

不过，更麻烦的是由 BADCUBE 和 BADABS 产生的错误，尽管它们都使用了足够多的圆括号来防止发生在 BADSQR 上的那一种错误。以下是 BADCUBE(++i)的运行情况：

```
     BADCUBE(++i)
=   ((++i) * (++i)) * (++i)    // left associativity
=   ((10) * (10)) * (11)
=   1100
```

一般而言，应该避免使用代码替换宏，大多数严肃的 C++程序员都将其视为危险的。预处理器宏主要用于下面几种情况。

1. 使用#ifndef/#define/#endif 将头文件包裹起来，以避免多次包含某个头文件。
2. 使用#ifdef/#else/#endif 对某些代码部分进行条件编译。
3. __FILE__宏和__LINE__宏用于调试并给出框架信息。

作为一个规则，一般应使用 inline 函数而不是宏来进行代码替换。这个规则的一个例外情况是使用 Qt 宏来向使用某些 Qt 类的程序插入代码。这也就容易理解为什么有些 C++专家怀疑 Qt 中宏的使用。

5.11 带变长实参表的函数

在 C 和 C++中，可以定义其参数表以省略号结尾的函数。省略号使调用者能够指定参数的数量以及类型。这种函数的一个例子来自于<stdio.h>：

```
int printf(char* formatStr, ...)
```

这种灵活机制使得可以进行下面这样的调用：

```
printf("Eschew Obfuscation!\n");
printf("%d days hath %s\n", 30, "September");
```

为了定义使用省略号的函数，需要包含 cstdarg 库，其中的一个宏集合用于访问 std 命名空间中实参表的各项。除省略号之外，参数表中必须至少还有另外一个参数。通常会用一个 va_list 类型的变量 ap(表示“实参指针”)来遍历未命名的实参表。宏

```
va_start(ap, p)
```

中的 p 是参数表中的最后一个命名参数，初始化 ap，使得它指向未命名实参中的第一个。

宏

```
va_arg(ap, typename)
```

返回 ap 指向的那个实参，并使用 typename 来确定(用 sizeof)找到下一个实参应前进多少。宏

```
va_end(ap)
```

必须在已经处理完全部的未命名实参后才能调用。它清除未命名实参的栈，并确保在函数终止后程序依然会运行正常。

示例 5.20 中演示了如何使用这些特性。

示例 5.20　src/ellipsis/ellipsis.cpp

```
#include <cstdarg>
#include <iostream>
using namespace std;

double mean(int n ...) {                                    1
    va_list ap;                                             2
    double sum(0);
    int count(n);
    va_start(ap, n);                                        3
    for (int i = 0; i < count; ++i) {
        sum += va_arg(ap, double);
    }
    va_end(ap);                                             4
    return sum / count;
}

int main() {
    cout << mean(4, 11.3, 22.5, 33.7, 44.9) << endl;
    cout << mean (5, 13.4, 22.5, 123.45, 421.33, 2525.353) << endl;
}
```

1 第一个参数是实参的数量。

2 依次指向每一个未命名实参。

3 现在，ap 执行第一个未命名实参。

4 返回之前清除栈。

5.12　练习：加密

1. 示例 5.16 中声明了 Point3 类的三个运算符但没有实现它们。为这三个运算符添加实现代码，并在客户代码中添加相应的测试代码。
2. 这个练习中，需复用来自于<cstdlib>(参见附录 B)的 random()函数。这个函数生成 0～RAND_MAX(通常是 2 147 483 647)之间的一个伪随机整数。编写函数

   ```
   int myRand(int min, int max);
   ```

   使其返回 min 到 max - 1 之间的一个伪随机整数。
3. 编写函数

   ```
   QVector<int> randomPerm(int n, unsigned key);
   ```

   它使用 myRand()函数(用 key 为种子)产生 0,…,n 的一个排列。
4. 加密与隐私正变得越来越重要。加密的一种思路是将一个字符串送入多个转换函数，经过这些转换函数转换的结果是一段可以更加安全地进行传送或保存的密文。此加密字符串的接收者可以使用转换函数的逆向过程来对密文进行逆向操作(即解密)，这样就能够获得原始字符串。加密字符串的发送者必须与接收者共享一些信息(即一个密钥)，以便解密字符串。下面的练习探索了一些简单的转换函数的设计。这些练习利

用了这样一个事实：random()返回的值序列完全由初始值(种子值)决定，因此这个序列是可重复的。

a. 编写函数

```
QString shift(const QString& text, unsigned key) ;
```

其中 shift()函数通过调用 srandom(),使用参数 key 设置随机函数的种子值。对于给定字符串 text 中的每一个字符 ch，加上下一个伪随机整数来获得一个移位的字符。然后，将这个移位的字符放入新字符串中对应的位置。当 text 中的全部字符都处理完之后，shift()返回这个新的字符串。

在将随机整数添加到用于处理字符的代码中时，必须进行“mod n”的加法运算，其中 n 是所使用的底层字符集中的字符数。对于这个练习，可以假定使用的是 ASCII 字符集，它有 128 个字符。

b. 下一个要编写的函数是

```
QString unshift(const QString& cryptext, unsigned key);
```

这个函数逆转上一个练习中描述的过程。

c. 编写代码，测试上面给出的 shift()函数和 unshift()函数。

d. 另外一种加密的方法(可以与上面描述的方法结合使用)是改变给定字符串中字符的顺序。编写函数

```
QString permute(const QString& text, unsigned key);
```

它使用 randomPerm()函数来生成原始字符串 text 的一个转换字符序列。

e. 编写函数

```
QString unpermute(const QString& scrtext, unsigned key);
```

它反转上面描述的 permute()函数的动作。

f. 编写代码，测试 permute()函数和 unpermute()函数。

g. 编写代码，测试对 shift()函数和 permute()函数使用同一个字符串的结果，再测试 unpermute()函数和 unshift()函数。

5. 实现封装前面练习中的函数的 Crypto 类。可以从图 5.4 中使用的 UML 框图开始。m_OpSequence 是一个 QString，它由代表 permute()函数和 shift()函数的字符'p'和's'组成。encrypt()函数将这两个函数用于给定的字符串，使其以在 m_OpSequence 字符串中出现的顺序排列。示例 5.21 中包含了测试这个类的一些代码。

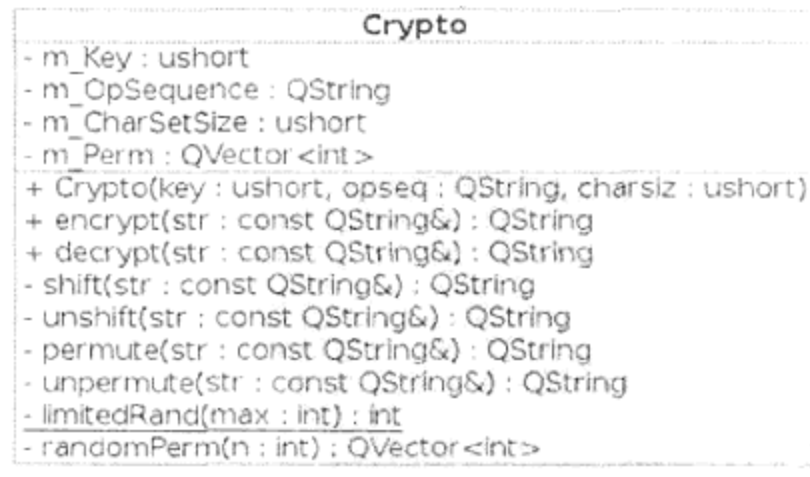


图 5.4 Crypto 的 UML 类框图

示例 5.21 src/functions/crypto-client.cpp

```
#include <QTextStream>
#include "crypto.h"

int main()  {
   QTextStream cout(stdout);
```

```
    QString str1 ("asdfghjkl;QWERTYUIOP{}}|123456&*()_+zxcvnm,,, ./?"),
            str2;
    cout << "Original string: " << str1 << endl;
    cout << "length: " << str1.length() << endl;
    QString seqstr("pspsp");
    ushort key(12579);
    Crypto crypt(key, seqstr);
    str2 = crypt.encrypt(str1);
    cout << "Encrypted string: " << str2 << endl;
    cout << "Recovered string: " << crypt.decrypt(str2) << endl;
}
```


关于加密的主题将在 17.1.4 节中再次探讨，这一节中将介绍一个进行加密哈希操作的 Qt 类。

5.13 复习题

1. 函数声明与函数定义有什么不同？
2. 为什么默认实参指示符出现在函数声明中而不在其定义中？
3. 如果位于同一作用域中的两个函数具有相同的签名而返回类型不同，则会导致错误。请给出解释。
4. 如果要对 Fraction 对象重载算术运算符(+, -, *, /)，则成员函数与非成员全局运算符相比哪一个更可取？给出理由。
5. 对于重载的左修饰运算符(比如-=和+=)，成员函数运算符与(非成员)全局运算符相比，哪一个更可取？
6. 解释按值传递与按引用传递的差异。在哪些情况下应使用其中的一个而不是另一个？
7. 解释预处理器宏与 inline 函数的差异。

第6章　继承与多态

本章讲解如何在C++类之间定义继承关系，并会给出一些示例。还将讲解重写方法、virtual关键字以及使用多态的一些简单示例。

6.1　简单派生

继承是组织类的一种特殊方式，所有面向对象的语言都支持这种特性，它使得类能够以多种不同的方式共享代码，并且可以揭示类之间的自然关系。它也可以使设计良好的类更具可复用性。

为了使用继承，需要将一组相关类的共同性质放置在一个基类之中，然后由它派生出其他更专门化的类。每一个派生类都继承基类的所有成员，当然也可以根据需要重写或者扩展基类中的每一个函数。从一个共同的基类继承各种成员大大简化了派生类，利用某些设计模式，还可以去除冗余代码。事实上，当消除一组相关类中重复的代码块时，就应当推荐使用继承。

提示

包含相同或相近重复代码的软件易于出错且难以维护。如果程序中含有重复代码，那么跟踪所有重复项将是一件非常困难的事情。

由于这样或者那样的原因，经常需要修改代码。如果需要修改一段重复出现多次的代码，就必须在程序中找到所有的重复项，然后逐一修改，或者需要仔细判断哪些必须修改。如此一来就非常有可能漏掉一个或者多个代码段，从而无法做到对所有的副本进行期望的修改。

重构(refactoring)是一个提高软件设计而不会改变其底层行为的过程。其中一个步骤就是将相似的代码块转换成对可复用函数的调用。

下面用一个简单的示例演示继承的用法。基类Student包含所有学生都有的共同属性。这个示例中只给出了这些属性的一小部分，但轻易就能想像出其他合适的共同属性。

从Student类派生出了两个类，它们分别描述了具有特殊属性的两种学生。第一个派生类是Undergrad，它包含本科生特有的属性；第二个派生类是GradStudent，它包含研究生特有的属性。图6.1中的UML框图描述了它们之间的关系。

Student::m_Year前面的符号#表明m_Year是Student类的一个protected成员。前面说过，派生类的成员函数可以访问类中的protected成员。Student类中的其他数据成员为private类型，因此

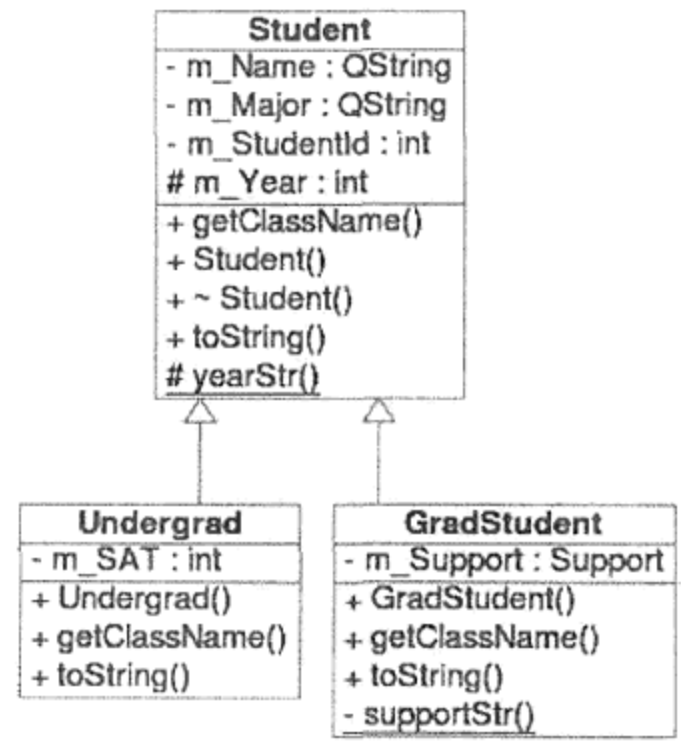


图6.1　继承的UML框图

在派生类的成员函数中无法访问它们。图中使用空三角箭头(指向基类)来表示类之间的继承关系。这个箭头也称为泛化(generalization)，因为它是从更具体的(派生)类指向更普通的(基)类。派生类也被称为基类的子类。

示例 6.1 中给出了这三个类的定义。

示例 6.1　src/derivation/qmono/student.h

```
#ifndef STUDENT_H
#define STUDENT_H

#include <QString>

class Student  {
 public:
    Student(QString nm, long id, QString major, int year = 1);
    ~Student() {}
    QString getClassName() const;
    QString toString() const;
 private:
    QString m_Name;
    QString m_Major;
    long m_StudentId;
 protected:
    int m_Year;
    QString yearStr() const;
};

class Undergrad: public Student {
 public:
    Undergrad(QString name, long id, QString major, int year, int sat);
    QString getClassName() const;
    QString toString() const;
 private:
    int m_SAT;  Scholastic Aptitude Test score total.

};

class GradStudent : public Student {
 public:
    enum Support { ta, ra, fellowship, other };
    GradStudent(QString nm, long id, QString major,
                int yr, Support support);

    QString getClassName() const ;
    QString toString() const;
 protected:
    static QString supportStr(Support sup) ;
 private:
    Support  m_Support;
};

#endif        //  #ifndef STUDENT_H
```

每一个派生类的类首部(classHead)都指明了它的基类以及所采用的派生方式。这里采用的是 public 派生方式[①]。

这三个类都有一个名称为 getClassName() 的函数和一个名称为 toString() 的函数。派生类中的这两个函数版本重写(override)了基类中对应的函数。这种派生类函数必须与它所重写的基类函数具有相同的签名和返回类型。

示例 6.2 中定义了 Student 类的成员函数。

示例 6.2 src/derivation/qmono/student.cpp

```
[ . . . . ]

#include <QTextStream>
#include "student.h"

Student::Student(QString nm, long id, QString major, int year)
    : m_Name(nm), m_Major(major), m_StudentId(id), m_Year(year) {}


QString Student::getClassName() const {
    return "Student";
}

QString Student::toString() const {
    QString retval;
    QTextStream os(&retval);                                    1
    os << "[" << getClassName() << "]"
       << " name: " << m_Name
       << "; Id: " << m_StudentId
       << "; Year: " << yearStr()
       << "; Major: " << m_Major  ;
    return retval;
}
```

1 写入流并返回它的字符串。

示例 6.3 给出了 Undergrad 成员函数定义。显然，Undergrad::toString()产生的字符串除了包含 Student 数据外，还包含自己的数据成员 m_SAT(SAT 表示 Scholastic Aptitude Test，即学术能力测试)[②]。注意，Student 类中的数据成员是 private 类型的，针对这一问题的解决办法是让 Undergrad::toString()调用 public 函数 Student::toString()。这样，封装就得以保留而任务得以合适地分配了：Student 类负责 Student 数据，Undergrad 类负责 Undergrad 数据。

示例 6.3 src/derivation/qmono/student.cpp

```
[ . . . . ]

Undergrad::Undergrad(QString name, long id, QString major,
                                                    int year, int sat)
```

① 22.4 节中将探讨派生的三种类型：public,protected 和 private。

② 在美国，SAT 是高中学生要参加的一种标准化考试，俗称“美国高考”，大多数大学的招生办公室都采用它。

```
                : Student(name, id, major, year), m_SAT(sat)          1
              { }

QString Undergrad::getClassName() const {
    return "Undergrad";
}

QString Undergrad::toString() const {
    QString result;
    QTextStream os(&result);
    os <<  Student::toString()                                        2
       << "\n  [SAT: "                                                3
       << m_SAT
       << " ]\n";
    return result;

}
```

1 基类对象被当成派生对象的子对象。类成员和基类都必须被初始化，初始化的顺序由它们在类定义中出现的顺序决定。

2 调用基类版本。

3 然后，添加 Undergrad 特有的项。

基类的成员初始化

由于每一个 Undergrad 都为 Student，所以只要创建了一个 Undergrad 对象，就必须同时创建并初始化一个 Student 对象。此外，还必须调用 Student 构造函数来初始化任何派生对象的 Student 部分。

在构造函数的成员初始化器中，可以将基类名称当成一个隐式的派生类成员。

- 它在派生类成员初始化之前首先进行初始化。
- 如果没有指定如何初始化基类，那么初始化过程将会调用默认构造函数。
- 如果基类没有默认构造函数，则编译器会报告错误。

分析一下 Undergrad 构造函数的签名。参数表中包含的 Student 数据成员的值为 private 类型，所以 Undergrad 成员函数无法对它们进行赋值。进行这种赋值的唯一途径是将这些值传递给 Student 构造函数，它不是 private 类型。

GradStudent 中增加的一些特性需要适当地处理，见示例 6.4。

示例 6.4　src/derivation/qmono/student.cpp

```
[ . . . . ]

GradStudent::
GradStudent(QString nm, long id, QString major, int yr,
                    Support support) :Student(nm, id, major, yr),
            m_Support(support) { }

QString GradStudent::toString() const {                               1
   return QString("%1%2%3 ]\n")
       .arg(Student::toString())                                       2
       .arg("\n  [Support: ")                                          3
```

```
        .arg(supportStr(m_Support));
}
```

1 另一种 QString 风格。

2 调用基类版本。

3 然后，添加 GradStudent 特有的项。

扩展

在 toString() 的两个派生类版本的内部处理派生类属性之前，都显式地调用了 Student::toString()，它用于处理(private 类型的)基类属性。toString()的每一个派生类版本都扩展了 Student::toString()的功能。

有必要再次强调的是，由于 Student 类的多数数据成员都是 private 类型的，需要一个非 private 类型的基类函数(例如，toString())来使派生类能够访问基类中的 private 数据成员。派生类对象不能直接访问 Student 类中的 private 成员，即使它包含这些成员也不行。这种设计对有些人来说需要一些时间来适应。

6.1.1 使用继承的客户代码示例

GradStudent 是一个 Student，因此无论何时，只要能够使用 Student 对象的地方就能够使用 GradStudent 对象。示例 6.5 中的客户代码创建了一些实例，并对一个 GradStudent 或 Undergrad 实例执行了一些直接操作或者通过指针进行间接操作。

示例 6.5 src/derivation/qmono/student-test.cpp

```
#include <QTextStream>
#include "student.h"

static QTextStream cout(stdout);

void finish(Student* student) {
    cout << "\nThe following "
         << student->getClassName()
         << " has applied for graduation.\n "
         << student->toString() << "\n";
}

int main() {
    Undergrad us("Frodo Baggins", 5562, "Ring Theory", 4, 1220);
    GradStudent gs("Bilbo Baggins", 3029, "History", 6, GradStudent::fellowship);
    cout << "Here is the data for the two students:\n";
    cout << gs.toString() << endl;
    cout << us.toString() << endl;
    cout << "\nHere is what happens when they finish their studies:\n";
    finish(&us);
    finish(&gs);
    return 0;
}
```

可以通过使用 qmake 和 make 来链编这个应用，步骤如下：

```
src/derivation/qmono> qmake -project
src/derivation/qmono> qmake
src/derivation/qmono> make
```

然后用下面的方式运行它：

```
src/derivation/qmono> ./qmono
Here is the data for the two students:
[Student]① name: Bilbo Baggins; Id: 3029; Year: gradual student; Major: History
  [Support: fellowship ]

[Student] name: Frodo Baggins; Id: 5562; Year: senior; Major: Ring Theory
  [SAT: 1220 ]

Here is what happens when they finish their studies:

The following Student has applied for graduation.
 [Student] name: Frodo Baggins; Id: 5562; Year: senior; Major: Ring Theory

The following Student has applied for graduation.
 [Student] name: Bilbo Baggins; Id: 3029; Year: gradual student; Major: History
src/derivation/qmono>
```

在 finish()函数中，参数 student 是一个基类指针。不管 student 指向何种对象，只要调用 student->toString()，就会激活 Student::toString()的调用。如果 student 指向 GradStudent，则应该在输出消息中有所提示。此外，在 toString()消息中应该能看到“[GradStudent]”字样，但是它没有出现。

如果程序用间接的运行时绑定的函数调用来决定每个对象应该使用哪一个 toString()函数，则会更合适一些。

因为 C++源于 C 语言，它的编译器也试图在编译时绑定函数调用，这主要是基于性能方面的考虑。编译器无法仅凭继承关系和基类指针就确定它正在操作何种对象。如果没有运行时检查，就无法保证运行时调用正确的函数。C++要求使用一个特殊的关键字来允许运行时通过指针和引用进行函数调用的绑定。这个关键字就是 virtual，它能够使得程序具有多态性(polymorphism)，将会在下一节中讲解。

6.1.2　练习：简单派生

1. 链编并运行 6.1 节和 6.1.1 节中描述的 Student 应用，然后从 Undergrad 构造函数中移除基类初始化器。当再次链编这个应用时会发生什么？请给出理由。
2. 修改客户代码，使其采用消息框而不是标准的输出格式来输出消息。
3. 修改这个应用，使得 finish()函数检查任何 Undergrad 学生的年级信息，并给出适当的消息。例如，对 GradStudent 学生而言，Freshman, Sophomore, Junior(大一、大二、大三)等年级信息就没有意义。对 GradStudent 学生应该如何处理？

6.2　具有多态性的派生

现在讲解面向对象编程中的另一种功能强大的特性：多态(polymorphism)。示例 6.6 与前一个示例的唯一不同是基类定义中 virtual 关键字的用法。

---

① 这里的输出中如果包含“[GradStudent]”字样，则会更好一些。

示例 6.6 src/derivation/qpoly/student.h

```
[ . . . . ]

class Student  {
 public:
    Student(QString nm, long id, QString major, int year = 1);
    virtual ~Student() {}                                          1
    virtual QString getClassName() const;                          2
    QString toString() const;                                      3
 private:
    QString m_Name;
    QString m_Major;
    long m_StudentId;
 protected:
    int m_Year;
    QString yearStr() const;
};
```

1 这里添加了 virtual 关键字。

2 这里也添加了 virtual 关键字。

3 此处也应有 virtual 关键字。

只要在一个成员函数中添加了关键字 virtual，就创建了一种多态类型。virtual 函数被称为方法(method)。这一术语与 Java 中的用法相同，Java 中的成员函数默认就是方法。示例 6.7 中再次给出了同样的客户代码。

示例 6.7 src/derivation/qpoly/student-test.cpp

```
#include <QTextStream>
#include "student.h"

static QTextStream cout(stdout);

void finish(Student* student) {
    cout << "\nThe following "
         << student->getClassName()
         << " has applied for graduation.\n "
         << student->toString() << "\n";
}

int main() {
    Undergrad us("Frodo Baggins", 5562, "Ring Theory", 4, 1220);
    GradStudent gs("Bilbo Baggins", 3029, "History", 6, GradStudent::fellowship);
    cout << "Here is the data for the two students:\n";
    cout << gs.toString() << endl;
    cout << us.toString() << endl;
    cout << "\nHere is what happens when they finish their studies:\n";
    finish(&us);
    finish(&gs);
    return 0;
}
```

运行该程序后的输出如下所示。

```
Here is the data for the two students:
[GradStudent] name: Bilbo Baggins; Id: 3029; Year: gradual student; Major: History
  [Support: fellowship ]

[Undergrad] name: Frodo Baggins; Id: 5562; Year: senior; Major: Ring Theory
  [SAT: 1220 ]


Here is what happens when they finish their studies:
The following Undergrad has applied for graduation.
 [Undergrad] name: Frodo Baggins; Id: 5562; Year: senior; Major: Ring Theory①

The following GradStudent has applied for graduation.
 [GradStudent] name: Bilbo Baggins; Id: 3029; Year: gradual student; Major:
History②
```

现在，“[GradStudent]”和“[UnderGrad]”字样出现在输出中，因为 getClassName() 为 virtual 函数。但是，对 GradStudent 而言，finish() 函数的输出中依然存在问题：Support 片段遗失了。

利用多态，解决了运行时方法的间接调用(通过指针和引用)，这被称为动态绑定或者运行时绑定。针对方法的直接调用(不通过指针或者引用)仍然是由编译器解析的，这被称为静态绑定或者编译时绑定。

这个例子中，如果 finish() 函数接收到一个 GradStudent 对象的地址，就会导致 student->toString() 调用该函数的 Student 版本。而当使用 Student::toString() 调用 getClassName() 时(通过 this 间接地调用，this 是一个基类指针)，它是一个 virtual 方法调用。

C++中，当使用 virtual 关键字时，动态绑定是必须开启的一个选项。

注意

由于“this”是在执行它的构造函数时被初始化的(或者执行它的析构函数时被销毁)，因此不要期望在这两种条件下能够执行正确的运行时绑定。特别地，由于构造函数设置的 virtual 函数表(它对运行时绑定至关重要)可能是不完整的(或者析构函数只是部分地销毁了它)，所以当在构造函数或者析构函数里面调用任何 this 方法时，将由编译时绑定决定应该调用哪一个方法，就好像不存在 virtual 关键字一样。正如 Scott Meyers 所说的那样：“构造函数或者析构函数中不存在 virtual 方法”[Meyers]。

注意

一般而言，如果类中包含一个或者多个 virtual 函数，则也应包含一个虚析构函数。这是因为，当对多态对象集合进行操作时，通常是通过基类指针删除这些对象，这会导致对析构函数的间接调用。如果析构函数不为 virtual 类型，则编译时绑定将决定应该调用哪一个析构函数，从而可能导致派生对象的不完整析构。

① 缺少 SAT 成绩。
② 缺少奖学金信息。

6.2.1 练习：具有多态性的派生

1. 在 Student 类定义中为 toString() 的声明添加 virtual 关键字。然后链编并运行这个程序，解释得到的结果。
2. 预测示例 6.8 到示例 6.12 中程序的输出结果。然后编译并运行它们，检查你的答案。

a. 示例 6.8 src/polymorphic1.cc

```
#include <iostream>
using namespace std;

class A {
public:
    virtual ~A() { }
    virtual void foo() {
        cout << "A's foo()" << endl;
        bar();
    }
    virtual void bar() {
        cout << "A's bar()" << endl;
    }
};

class B: public A {
public:
    void foo() {
        cout << "B's foo()" << endl;
        A::foo();
    }
    void bar() {
        cout << "B's bar()" << endl;
    }
};

int main() {
    B bobj;
    A *aptr = &bobj;
    aptr->foo();
    cout << "-------------" << endl;
    A aobj = *aptr;
    aobj.foo();
    cout << "-------------" << endl;
    aobj = bobj;
    aobj.foo();
    cout << "-------------"<< endl;
    bobj.foo();
}
```

b. 示例 6.9 src/polymorphic2.cc

```
#include <iostream>
using namespace std;
```

```
class A {
public:
    virtual void foo() {
        cout << "A's foo()" << endl;
    }
};

class B: public A {
public:
    void foo() {
        cout << "B's foo()" << endl;
    }
};

class C: public B {
public:
    void foo() {
        cout << "C's foo()" << endl;
    }
};

int main() {
    C cobj;
    Bx *bptr = &cobj;
    bptr->foo();
    A* aptr = &cobj;
    aptr->foo();
}
```

c. 示例 6.10　src/derivation/exercise/Base.h

```
[ . . . . ]
class Base {
public:
    Base();
    void a();
    virtual void b() ;
    virtual void c(bool condition=true);
    virtual ~Base() {}
};


class Derived : public Base {
public:
    Derived();
    virtual void a();
    void b();
    void c();
};
[ . . . . ]
```

d. 示例 6.11　src/derivation/exercise/Base.cpp

```
[ . . . . ]
Base::Base() {
```

```
    cout << "Base::Base() " << endl;
    a();
    c();
}
void Base::c(bool condition) {
    cout << "Base::c()" << endl;
}
void Base::a() {
    cout << "Base::a()" << endl;
    b();
}
void Base::b() {
    cout << "Base::b()" << endl;
}

Derived::Derived() {
    cout << "Derived::Derived() " << endl;
}

void Derived::a() {
    cout << "Derived::a()" << endl;
    c();
}
void Derived::b() {
    cout << "Derived::b()" << endl;
}

void Derived::c() {
    cout << "Derived::c()" << endl;
}
[ . . . . ]
```

e. 示例 6.12 src/derivation/exercise/main.cpp

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

    Base b;
    Derived d;
    cout << "Objects Created" << endl;
    b.b();
    cout << "Calling derived methods" << endl;
    d.a();
    d.b();
    d.c();
    cout << ".. via base class pointers..." << endl;
    Base* bp = &d;
    bp->a();
    bp->b();
    bp->c();
    //d.c(false);
}
[ . . . . ]
```

6.3　抽象基类的派生

考虑图 6.2，它给出了动物王国中一小部分动物的继承关系图。下面将使用这个图来解释抽象类与具体类的不同。抽象基类用于封装具体派生类的共同特性。尽管不能实例化抽象类，但是当整理不断积累的庞大而复杂的生物世界的知识时，这种机制是非常实用和高效的。例如，灵长类动物是一种具有某些额外特征的哺乳动物，原始人是拥有额外特征的灵长类动物，而大猩猩是拥有额外特征的原始人，等等。

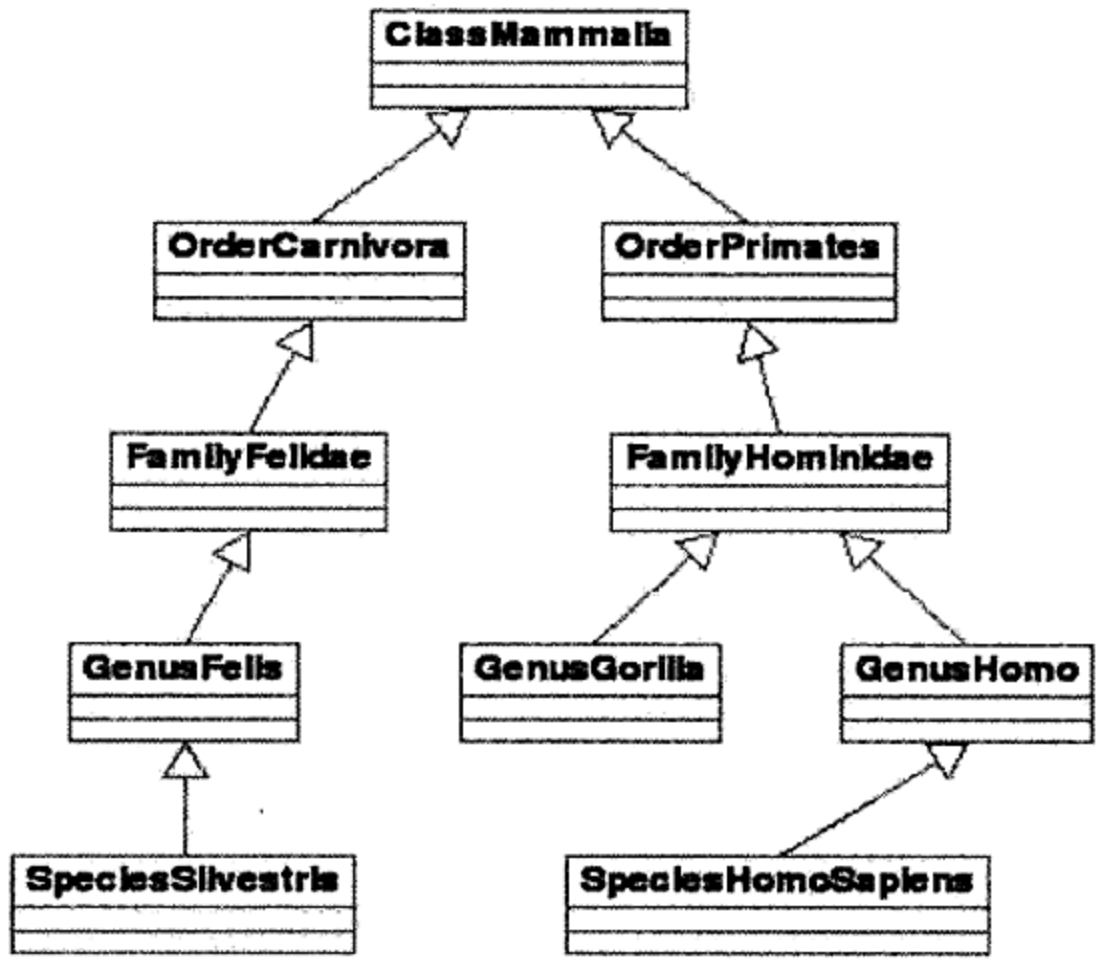


图 6.2　动物的分类

具体类代表某一类特定的实体，它们是真实存在的东西，是可以被实例化的。例如，当走在树林中时，永远不可能遇到一个能够被名称“肉食动物”或者“猫科动物”完整描述的真实的、活的动物。但是，根据所在地区的不同，有可能遇到狮子、野猫或者普通的家猫。但是，现实世界中没有任何一个 Hominidae(人科，也就是基类)。如果某位生物学家发现了一个具体的动物无法归入任何一个目前已经存在的物种类别中，那么他就可以定义并命名这个新的物种，这样就出名了。

总结来说，较为普通的物种类别(纲、目、科、子科)都是在现实世界中无法实例化的抽象基类。人们使用这些概念有助于分类和组织具体类(物种)。

回到编程

乍看起来，定义一个没有具体实现的抽象类似乎有违常理，但类仅仅是函数和数据的分组，是用来实现某种方式的组织和复用的有效工具。将事物分类，可以使人和计算机更加简单地管理世界。

当研究设计模式，开发框架和类库时，经常要设计继承树，其中只有叶节点才能被实例化，而所有的内部节点都是抽象的。

抽象基类是无法或者不适合实例化的类。通知编译器要遵守这个规则的类的特性如下：

- 至少具有一个纯 virtual 函数。
- 没有 public 构造函数。

图 6.3 中给出的抽象 Shape 类就具有纯 virtual 函数。在 UML 框图中，抽象类的名称以斜体显示。

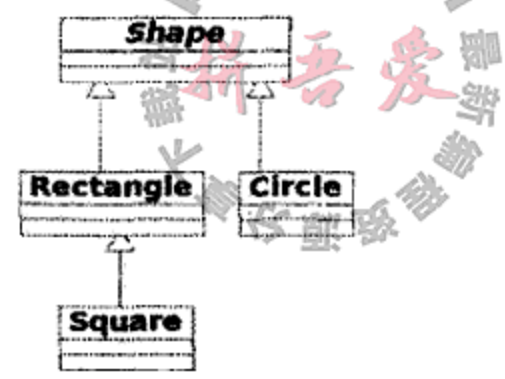


图 6.3 Shapes 类的 UML 框图

纯 virtual 函数的声明语法如下：

```
virtual returnType functionName(parameterList)=0;
```

示例 6.13 中给出了一个基类定义。

示例 6.13 src/derivation/shape1/shapes.h

```
[ . . . . ]

class Shape {                                                    1
 public:
    virtual double area() = 0;                                   2
    virtual QString getName() = 0;
    virtual QString getDimensions() = 0;
    virtual ~Shape() {}
};
```

1 抽象基类。

2 纯 virtual 函数。

getName(),area()和 getDimensions()都是纯 virtual 函数。由于它们被定义成纯 virtual 类型，所以在 Shape 类中不要求有函数定义。任何具体派生类都必须重写并定义全部的纯 virtual 基类函数，以进行实例化。换句话说，任何没有重写并定义全部纯 virtual 基类函数的派生类，都是抽象类。示例 6.14 中给出了一些派生类的定义。

示例 6.14 src/derivation/shape1/shapes.h

```
[ . . . . ]

class Rectangle : public Shape {
 public:
    Rectangle(double h, double w) :
        m_Height(h), m_Width(w) {}
    double area();
    QString getName();
    QString getDimensions();

 protected:                                                      1
    double m_Height, m_Width;
};

class Square : public Rectangle {
 public:
    Square(double h)
        : Rectangle(h,h)
    { }
    double area();
    QString getName();
```

```
    QString getDimensions();
};

class Circle : public Shape {
 public:
    Circle(double r) : m_Radius(r) {}
    double area();
    QString getName();
    QString getDimensions();
 private:
    double m_Radius;
};
```

1 希望在 Square 类中访问 m_Height。

2 成员初始化表中的基类名称，将实参传递给基类构造函数。

Rectangle 类和 Circle 类派生自 Shape 类，Square 类派生自 Rectangle 类，它们的实现见示例 6.15。

示例 6.15　src/derivation/shape1/shapes.cpp

```
#include "shapes.h"
#include <math.h>
    double Circle::area() {
        return(M_PI * m_Radius * m_Radius);                    1
    }

    double Rectangle::area() {
        return (m_Height * m_Width);
    }

    double Square::area() {
        return (Rectangle::area());                            2
    }
[ . . . . ]
```

1 M_PI 来自于<math.h>，它位于 cstdlib 库中。

2 对 this 调用基类版本。

示例 6.16 中提供了练习这些类的一些客户代码。

示例 6.16　src/derivation/shape1/shape1.cpp

```
#include "shapes.h"
#include <QString>
#include <QDebug>

void showNameAndArea(Shape* pshp) {
    qDebug() << pshp->getName()
             << " " << pshp->getDimensions()
             << " area= " << pshp->area();
}

int main() {
    Shape shp;                                                 1
```

```
    Rectangle  rectangle(4.1, 5.2);
    Square     square(5.1);
    Circle     circle(6.1);

    qDebug() << "This program uses hierarchies for Shapes";
    showNameAndArea(&rectangle);
    showNameAndArea(&circle);
    showNameAndArea(&square);
    return 0;
}
```


1 错误——对具有纯 virtual 函数的类进行实例化是不允许的。

在全局函数 showNameAndArea() 中，基类指针 pshp 接连指向三个子类对象。对于每一个地址赋值，pshp 都多态地调用正确的 getName() 函数和 area() 函数。示例 6.17 是这个程序的输出结果。

示例 6.17　src/derivation/shape1/shape.txt

```
This program uses hierarchies for Shapes

RECTANGLE  Height = 4.1 Width = 5.2   area = 21.32
CIRCLE  Radius = 6.1   area = 116.899
SQUARE  Height = 5.1    area = 26.01
```

6.4　继承设计

有时引入继承关系之后，起初会对问题有所帮助(例如减少冗余代码)，但到后来必须将其他的类添加到继承层次中时就会引起一些问题。预先的分析往往有助于简化问题，从而避免后续的麻烦。

示例 6.14 中派生了抽象 Shape 类，展示了两个层次上的继承关系。Rectangle 类用来将对象归类，同时也作为一个具体类。

正方形是一种矩形吗？从几何上来说应该算是。下面是从基础几何学中借用的几个概念：

- 形状是平面上的一个二维封闭对象，它能够使用图形化的方式来表达。每一个形状都有一个称为“中心”的点。
- 矩形是一个由四条直线组成的形状，两条直线的夹角为 90°。
- 正方形是一个四条边都相等的矩形。

尝试为正在设计的应用给出类之间的一棵继承关系树，这有助于列出需要给每一个类提供的各种功能。对于几何形状，这些功能可能是：

- 可绘制性
- 可伸缩性
- 可加载性
- 可保存性

在更深入地描述接口之后，可能会发现给形状分类的经典几何定义在这个应用的情形下并不是理想的形状分类。

当我们进行分析时，会发现存在如下的问题：

- 希望对所有形状执行的共同操作是什么？
- 应用中还会用到另外哪些种类的形状？
- 为什么需要一个 Rectangle 类作为 Square 的基类？
- Square 能够用 Rectangle 替换吗？
- 和 Rectangle 一样，Rhombus(菱形)也有四条边，那么 Rectangle 是否应该从 Rhombus 派生？
- 是否应为全部的四边形构造一个基类？
- 是否应为全部的五边形构造另外一个基类？
- 是否应该构造一个普通的多边形基类并使用一个属性来代表多边形的边数？
- 程序要进行几何证据搜索来区分对象吗？

利用 UML 建模工具，可以在编写代码之前更容易地尝试不同的想法。UML 框图对关注和描述较大系统的各个小部分尤其有用。图 6.4 中的几个具体类可也充当创建更“特定”形状的模板。

在这个树框图中，位于叶节点上的类是基类的“特定”版本。箭头表示的接口，对象的绘制、加载和保存都是在抽象基类中处理的。

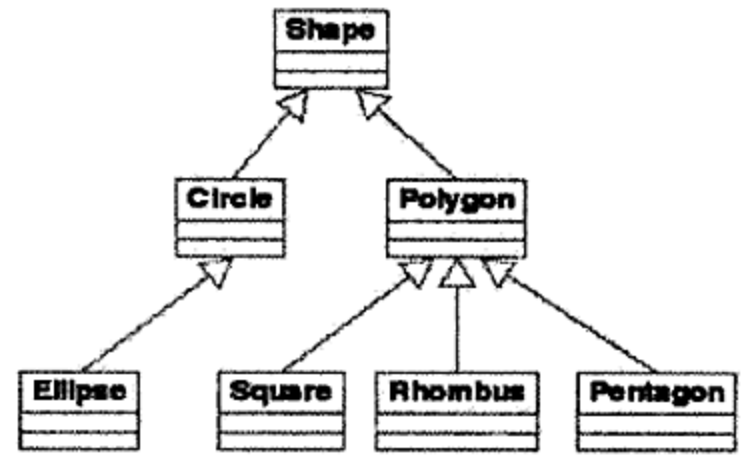


图 6.4　表示形状的另一种方法

问题

1. 从几何意义上来说，只要给定一个圆，就能证明它也是一个椭圆，因为确实存在一个描述椭圆的方程，只要它的两个焦距相等就为圆。反过来，图 6.4 表明了椭圆也是一种圆，只不过它具有一个额外的点(或者自由度)。如果将继承关系反过来，是否更具说服力？或者是否还有另外一棵完全不同的继承树？
2. 能否描述这些类中间的某两个具有更好的“是”(*is-a*)关系？
3. 考虑图 6.5，它们是来自于 Qt GraphicsView 库的 Shape 类[①]。

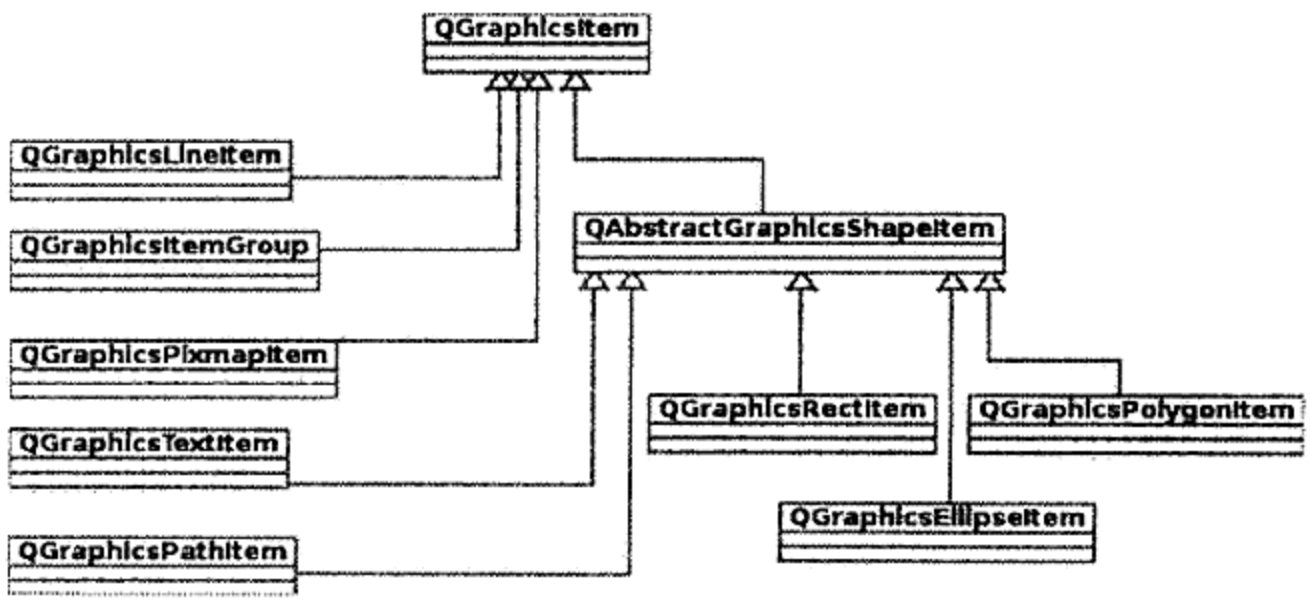


图 6.5　QGraphicsItem 的继承关系

① 关于这些类的更多信息，请参见 Qt Assistant 中的 *Qt Graphics View Framework Overview*。

注意,不存在具体的 Circle 项或 Square 项。请给出设计这样一种继承关系的理由。为什么同时存在 Rectangle 项和 Polygon 项?

6.5 重载，隐藏与重写

首先来回顾一下两个经常混淆的术语的定义。

- 当函数 foo 在同一个作用域内存在两个或者多个版本(具有不同的签名)时,就称 foo 函数被“重载”了。
- 当基类中的一个 virtual 函数在派生类中也存在，并且它们具有相同的签名和返回类型时，就称生类中的版本“重写”了基类中的版本。

示例 6.18 演示了重载与重写的区别，同时还引入了另外一种允许函数同名的关系。

示例 6.18 src/derivation/overload/account.h

```
[ . . . . ]
class Account {
public:
    Account(unsigned acctno, double bal, QString owner);
    virtual ~Account() { }
    virtual void deposit(double amt);
    virtual QString toString() const;
    virtual QString toString(char delimiter);                    1
protected:
    unsigned  m_AcctNo;
    double    m_Balance;
    QString   m_Owner;
};

class InsecureAccount: public Account {
public:
    InsecureAccount(unsigned acctno, double bal, QString owner);
    QString toString() const;                                    2
    void deposit(double amt, QDate postDate);                    3
};
[ . . . . ]
```

1 函数重载。

2 重写基类方法并隐藏 toString(char)。

3 没有重写方法，但是隐藏了全部的 Account::deposit()方法。

函数隐藏

派生类中的成员函数，会隐藏基类中与之同名的全部函数。如果出现这种情况，则:

- 只有派生类函数可以被直接调用。
- 类作用域解析运算符::必须用来显式地调用基类函数。

示例 6.19 展示了被隐藏的成员函数与不可访问的成员函数之间的区别。

示例 6.19　src/derivation/overload/account-client.cpp

```
#include  "account.h"
#include  <QTextStream>

int main() {
    InsecureAccount acct(12345, 321.98, "Luke Skywalker");
    acct.deposit(6.23);                                          1
    acct.m_Balance += 6.23;                                      2
    acct.Account::deposit(6.23);                                 3
    // ... more client code
    return 0;
}
```

1 错误：没有匹配的函数——被 deposit(double, int)隐藏。

2 错误：成员函数被包含，不可访问。

3 隐藏并不意味着不可访问。依然可以通过作用域解析来访问隐藏的公共成员。

6.6　构造函数，析构函数与复制赋值运算符

有三种特殊的成员函数从来不会被继承：

1. 复制构造函数
2. 复制赋值运算符
3. 析构函数

编译器会为没有定义它们的类自动生成这三种函数。

为什么这三种函数比较特殊

基类函数不足以初始化、复制或者销毁一个派生类的实例。

构造函数

对于继承自另一个类的类，基类的构造函数必须作为派生类初始化过程的一部分被调用。派生的构造函数在它的初始化表中可能需要指定应该调用基类的哪一个构造函数。

如果一个类没有构造函数，那么编译器就会自动产生一个默认的 public 构造函数，而这个构造函数会调用它的每一个基类的默认构造函数。如果某个类有一些构造函数但没有默认构造函数，那么它就不会进行默认的初始化工作。这种情况下，任何派生类构造函数都必须在其初始化表中明确地指定应该调用哪一个基类构造函数。

初始化的顺序

初始化过程按照下面的顺序进行：

1. 首先是基类初始化，按照它们在派生类的类首部中出现的顺序依次进行。
2. 数据成员初始化，按照声明的顺序进行。

复制赋值运算符

如果类没有明确定义复制赋值运算符，那么编译器会自动为它产生一个 public 版本的

复制赋值运算符。由于基类数据成员通常为 private 类型的，所以派生类复制赋值运算符必须(为每一个基类)调用基类的赋值运算符，以对遇到的那些数据成员进行逐成员(memberwise)的复制。随后，才可以对派生类数据成员执行逐成员的赋值。

其他的成员函数运算符，都是像常规的成员函数那样被继承下来的。

复制构造函数

如同复制赋值运算符一样，如果类没有定义复制构造函数，编译器也会自动为它产生一个 public 类型的复制构造函数。编译器产生的复制构造函数通过复制它的实参对象的数据成员，对所有成员进行初始化。

示例 6.20 中只定义了一个要求接收三个实参的构造函数，所以 Account 没有默认构造函数(换句话说，编译器将不会为它产生默认构造函数)。这里将基类析构函数声明成 virtual 类型，以确保在销毁通过基类指针访问的派生对象时，会调用合适的派生类析构函数。

示例 6.20 src/derivation/assigcopy/account.h

```
[ . . . . ]

class Account {
 public:
    Account(unsigned acctNum, double balance, QString owner);
    virtual ~Account(){
      qDebug() << "Closing Acct - sending e-mail to primary acctholder:"
               << m_Owner; }
    virtual QString getName() const {return m_Owner;}
    // other virtual functions
 private:
    unsigned  m_AcctNum;
    double    m_Balance;
    QString    m_Owner;
};
```

这里没有定义复制构造函数，也就意味着编译器将会自动产生一个。因此，这个类实际上可以按照两种方式进行实例化：(1)调用三参数构造函数；(2)调用编译器产生的复制构造函数并对其采用 Account 对象实参。

示例 6.21 中定义的派生类有两个构造函数，它们都要求进行基类初始化。

示例 6.21 src/derivation/assigcopy/account.h

```
[ . . . . ]

class JointAccount : public Account {
 public:
  JointAccount (unsigned acctNum, double balance,
                QString owner, QString jowner);
  JointAccount(const Account & acct, QString jowner);
  ~JointAccount() {
     qDebug() << "Closing Joint Acct - sending e-mail to joint acctholder:"
              << m_JointOwner; }
  QString getName() const {
    return QString("%1 and %2").arg(Account::getName()).arg(m_JointOwner);
```

```
    }
    // other overrides
  private:
    QString m_JointOwner;
};
```


示例 6.22 中，虽然并没有定义 Account(const Account&) 函数，但编译器依然允许 JointAccount::JointAccount 调用它。这是因为，编译器产生的复制构造函数会按照类定义中数据成员列出的顺序进行逐成员的复制/初始化工作。

示例 6.22　src/derivation/assigcopy/account.cpp

```
[ . . . . ]

Account::Account(unsigned acctNum, double balance, QString owner) :
      m_AcctNum(acctNum), m_Balance(balance), m_Owner(owner)
    { }

JointAccount::JointAccount (unsigned acctNum, double balance,
                            QString owner, QString jowner)
    :Account(acctNum, balance, owner),
     m_JointOwner(jowner)                                          1
     { }

JointAccount::JointAccount (const Account& acc, QString jowner)
    :Account(acc),                                                 2
     m_JointOwner(jowner)
    { }
```

1 基类初始化所要求的。

2 编译器产生的复制构造函数调用。

示例 6.23 中定义了 Bank 类用于维护 Account 指针的列表。

示例 6.23　src/derivation/assigcopy/bank.h

```
[ . . . . ]
class Account;

class Bank {
 public:
    Bank& operator<< (Account* acct);                              1
    ~Bank();
    QString getAcctListing() const;
 private:
    QList<Account*> m_Accounts;
};
[ . . . . ]
```

1 这就是将对象指针添加到 m_Accounts 中的方法。

示例 6.24 中，对象 a4 的构造函数使用编译器提供的 JointAccount 的复制构造函数，而它会调用编译器提供的 Account 的复制构造函数。

示例 6.24 src/derivation/assigcopy/bank.cpp

```
[ . . . . ]
#include <QDebug>
#include "bank.h"
#include "account.h"

Bank::~Bank() {
    qDeleteAll(m_Accounts);
    m_Accounts.clear();
}


Bank& Bank::operator<< (Account* acct) {
   m_Accounts << acct;
   return *this;
}

QString Bank::getAcctListing() const {
   QString listing("\n");
   foreach(Account* acct, m_Accounts)
      listing += QString("%1\n").arg(acct->getName());          1
   return listing;
}
int main() {
  QString listing;
   {                                                            2
      Bank bnk;
      Account* a1 = new Account(1, 423, "Gene Kelly");
      JointAccount* a2 = new JointAccount(2, 1541, "Fred Astaire",
         "Ginger Rodgers");
      JointAccount* a3 = new JointAccount(*a1, "Leslie Caron");
      bnk << a1;
      bnk << a2;
      bnk << a3;
      JointAccount* a4 = new JointAccount(*a3);                 3
      bnk << a4;
      listing = bnk.getAcctListing();
    }                                                           4
    qDebug() << listing;
    qDebug() << "Now exit program" ;
}
[ . . . . ]
```

1 getName()为 virtual 类型。

2 内部语句块的开始。

3 这条语句的作用是什么?

4 在这里，作为销毁 bank 类对象的一部分，全部四个 Account 对象都被销毁了。

析构函数

析构函数不会被继承。正如复制构造函数和复制赋值运算符一样，如果没有显式地为某

个类定义析构函数，编译器就会为它产生一个析构函数。当销毁派生的对象时，会自动调用基类的析构函数。数据成员和基类部分的销毁过程将按照与初始化过程相反的顺序进行。

6.7　处理命令行实参

从命令行运行的应用程序，经常通过命令行实参对其进行控制，这些实参可以是开关或者参数。ls, g++和 qmake 都是常见的接收命令行实参的应用。

可以按照各种方式处理不同类型的命令行实参。假设正在编写一个支持如下选项的程序：

```
Usage:
   a.out [-v] [-t] inputfile.ext [additional files]
     If -v is present then verbose  = true;
     If -t is present then testmode = true;
```

通常而言，程序并不会在意这些可选开关出现的顺序。在 Usage 描述中，可选实参总是被放置在方括号中，而必选实参没有方括号。这个程序接收至少包含一个文件名的任意长度的参数表，并且对每一个文件都执行同一种操作。

一般来说，命令行实参可以是下面的任何一种。

- 开关，例如-verbose 或者-t。
- 参数(通常是对文件的描述)，与开关不相关的简单字符串。
- 开关参数，例如 gnu 编译器的可选-o 开关，它要求一个对应的参数，即要生成的可执行文件的名称。

下面的这一行包含了所有三种类型的实参：

```
g++ -ansi -pedantic -Wall -o myapp someclass.cpp someclass-demo.cpp
```

示例 6.25 揭示了 C 语言程序是如何处理命令行实参的。

示例 6.25　src/derivation/argumentlist/argproc.cpp

```
[ . . . . ]
#include <cstring>

bool test = false;
bool verbose = false;

void processFile(char* filename) {
[ . . . . ]
}

/*
  @param argc - the number of arguments
  @param argv - an array of argument strings
*/
int main (int argc, char* argv[]) {
  // recall that argv[0] holds the name of the executable.
  for (int i=1; i < argc; ++i) {

     if (strcmp(argv[i], "-v")==0) {
        verbose = true;
```

```
        }
        if (strcmp(argv[i], "-t") ==0) {
           test = true;
        }
    }
    for (int i=1; i < argc; ++i) {
        if (argv[i][0] != '-')
          processFile(argv[i]);
    }
}
[ . . . . ]
```


1 首次处理开关。

2 第二遍操作处理非开关实参。

通过使用更多的面向对象的构造，就可以在 Qt 中避免使用数组、指针和<cstring>。

示例 6.26 中可以看到通过使用更高级的 QString 类和 QStringList 类来极大地简化类似于上面的实现代码。

6.7.1 派生与 ArgumentList

ArgumentList 就是一个可复用类的例子，它是从一个更通用的 Qt 类派生而来的，具有特定的作用。它复用了 QString 类和 QStringList 类，以简化命令行实参的处理。

从操作上说，ArgumentList 是一个用 main() 函数的 int 参数和 char**参数进行初始化的类，这两个参数会取得命令行实参的值；从概念上说，ArgumentList 就是一个 QString 的列表；从结构上说，ArgumentList 派生自 QStringList，并且添加了一些功能。Java 程序员会说 ArgumentList 扩展自 QStringList。示例 6.26 中包含了 ArgumentList 类的定义。

示例 6.26 src/derivation/argumentlist/argumentlist.h

```
#ifndef ARGUMENTLIST_H
#define ARGUMENTLIST_H

#include <QStringList>

class ArgumentList : public QStringList {
  public:
    ArgumentList();

    ArgumentList(int argc, char* argv[]) {
        argsToStringlist(argc, argv);
    }
    ArgumentList(const QStringList& argumentList):
       QStringList(argumentList) {}
    bool getSwitch(QString option);
    QString getSwitchArg(QString option,
                         QString defaultRetVal=QString());
  private:
    void argsToStringlist(int argc,  char* argv[]);
};
#endif
```

因为 ArgumentList 公共地派生自 QStringList，所以它支持 QStringList 的全部接口，并且可以用在任何能够使用 QStringList 的地方。除了自己的构造函数之外，ArgumentList 还定义了一些额外的函数：

- argsToStringList() 函数从给定的 char 数组中提取命令行实参并将其加载到 QStringList 中。这个函数是 private 类型的，因为它是这个类的实现部分，而不是 public 接口部分。构造函数需要这个函数，而客户代码并不需要它。
- getSwitch() 函数会找出并移除字符串表中的一个开关(只要这个开关存在)。如果找到了开关，则返回 true，否则返回 false。
- getSwitchArg() 函数会找出并移除字符串表中的开关以及与之对应的实参。如果找到了开关，则返回这个实参；如果没有找到开关，则什么也不做并返回默认值。

示例 6.27 中展示了这些函数的实现代码。

示例 6.27　src/derivation/argumentlist/argumentlist.cpp

```
#include <QCoreApplication>
#include <QDebug>
#include "argumentlist.h"
ArgumentList::ArgumentList() {
   if (qApp != NULL)
      *this = qApp->arguments();
}


void ArgumentList::argsToStringlist(int argc, char * argv []) {
   for (int i=0; i < argc; ++i) {
      *this += argv[i];
   }
}

bool ArgumentList::getSwitch (QString option) {
   QMutableStringListIterator itr(*this);
   while (itr.hasNext()) {
     if (option == itr.next()) {
        itr.remove();
        return true;
     }
   }
    return false;
}

QString ArgumentList::getSwitchArg(QString option, QString defaultValue) {
   if (isEmpty())
      return defaultValue;
   QMutableStringListIterator itr(*this);
   while (itr.hasNext()) {
      if (option == itr.next()) {
         itr.remove();
         if (itr.hasNext()) {
            QString retval = itr.next();
```

```
            itr.remove();
            return retval;
         }
         else {
            qDebug() << "Missing Argument for " << option;
            return QString();
         }
      }
   }
   return defaultValue;
}
```

1 指向当前 QApplication 的全局指针。

示例 6.28 里给出的客户代码中，所有的实参处理代码都已经从 main() 中移走。这里没有循环和 char*，也没有 strcmp 函数。

示例 6.28 src/derivation/argumentlist/main.cpp

```
#include <QString>
#include <QDebug>
#include "argumentlist.h"

void processFile(QString filename, bool verbose) {
   if (verbose)
      qDebug() << QString("Do something chatty with %1.")
                           .arg(filename);
   else
      qDebug() << filename;
}

void runTestOnly(QStringList & listOfFiles, bool verbose) {
   foreach (const QString &current, listOfFiles) {
      processFile(current, verbose);
   }
}

int main( int argc, char * argv[] ) {
   ArgumentList al(argc, argv);                          1
   QString appname = al.takeFirst();                     2
   qDebug() << "Running " << appname;
   bool verbose = al.getSwitch("-v");
   bool testing = al.getSwitch("-t");                    3
   if (testing) {
      runTestOnly(al, verbose);                          4
      return 0;
   } else {
      qDebug() << "This Is Not A Test";
   }
}
```

1 用命令行实参实例化 ArgumentList。

2 继承自 QStringList，即列表中的第一项是可执行文件的名称。

3 现在，所有的开关都已经从列表中移除，只剩下文件名称。

4 ArgumentList 可用来代替 QStringList。

以下是运行示例 6.28 中的程序得到的输出样本。

```
src/derivation/argumentlist> ./argumentlist
Running  "./argumentlist"
This Is Not A Test
src/derivation/argumentlist> ./argumentlist item1 "item2 item3" item4 item5
Running  "./argumentlist"
This Is Not A Test
src/derivation/argumentlist> ./argumentlist -v -t "foo bar" 123 space1 "1 1"

Running  "./argumentlist"
"Do something chatty with foo bar."
"Do something chatty with 123."
"Do something chatty with space1."
"Do something chatty with 1 1."
src/derivation/argumentlist>
```

6.7.2　练习：处理命令行实参

编写一个生日提醒程序，其名称为 `birthdays`。

- 将任意格式的姓名/生日对保存到 `birthdays.dat` 文件中。
- 不带任何命令行实参的 `birthdays` 命令将打开这个文件，并按时间顺序列出将在未来 30 天内过生日的全部生日信息。
- 命令 `birthdays -a "john smith" "yyyy-mm-dd"`将在这个文件中添加一项。
- 命令 `birthdays -n 40` 会显示将在未来 40 天内过生日的全部生日信息。
- 命令 `birthdays` *nameSpec* 就搜索姓名为 *nameSpec* 的人的生日信息。

6.8　容器

Qt 的容器类是一种值类型(能够被复制的事物)的集合，包括指向对象类型的指针[①](但不包括对象类型)。Qt 容器被定义成模板类，这样就使得它所包含的类型是未指定的。每一种数据结构都针对不同种类的操作进行了优化。在 Qt 中，有多种模板容器类可供选择。

- `QList<T>`是用数组实现的，数组的两端都有预分配的空间。它针对按索引的随机访问以及少于 1000 项的列表进行了优化。对于 `prepend()` 和 `append()` 这样的操作，它也有很好的性能表现。
- `QStringList` 是派生自 `QList<QString>`的一个便利类。
- `QLinkedList<T>`针对迭代器的顺序访问以及快速、常量时间的列表插入操作进行了优化，但排序和搜索比较缓慢。它提供多个便利函数来处理那些经常用到的操作。
- `QVector<T>`以连续的内存位置保存数据，并针对按索引的随机访问进行了优化。通常而言，`QVector` 对象都是用其初始大小进行构造的，在其两端都不存在预先自动分配的内存空间，所以中间插入、末端插入以及前端插入都是耗时的。
- `QStack<T>`是派生自 `QVector<T>`的 `public` 类，所以 `QVector` 类的 `public` 接

① 第 8 章中将探讨 `QObject` 和对象类型。

口可以用于 QStack 对象。不过，push()，pop()以及 top()函数采用的是后入先出(LIFO)的语法。

- QMap<*Key*, *T*>是一个有序的关联容器(associative container)，它保存的是键/值对，其作用是根据键来快速找到对应的值。它也被设计成支持适量的快速插入操作和删除操作。它将键有序排列，以便能够快速搜索和快速缩小搜索范围，采用的是一个跳跃列表字典(skip-list dictionary)[①]，这个字典在概率上是平衡的并且高效地利用了内存。键的类型必须是 operator<()和 operator==()。
- QHash<*Key*, *T*>也是一个关联容器，它使用哈希表来进行键的查找。它提供快速的查找(键精确匹配)和插入操作，但其搜索速度较慢，且没有排序功能。键的类型必须是 operator==()。
- QMultiMap<*Key*, *T*>是 QMap 的一个子类，而 QMultiHash<Key, *T*>是 QHash 的一个子类。这两个类使得一个键可以与多个值相关联。
- QCache<*Key*, *T*>是一个关联容器，它对最近使用过的项提供最快速的访问，并会根据几个开销函数的结果自动移除那些不常用的项。
- QSet<*T*>用 QHash 保存 *T* 类型的值，QHash 中的键位于 *T* 中，而其中的哑值与每一个键相关联。这种安排可优化查找和插入操作。QSet 中的几个函数用于常规的集合操作(例如，集合的并、交、差等)。它的默认构造函数会创建一个空集合。

用于模板容器类的类型参数 *T*，或者用于关联容器的键类型，都必须为可赋值数据类型，即值类型(参见 8.1 节)。这意味着 *T* 必须具有 public 类型的默认构造函数、复制构造函数和赋值运算符。

基本类(例如，int, double, char 等)和指针都是可赋值的。有些 Qt 类型是可赋值的(例如，QString, QDate, QTime)。QObject 以及派生自 QObject 的类型都是不可赋值的。如果需要使用某种不可赋值类型的对象，则可以定义一个指针容器，比如 QList<QFile*>。

6.9 托管容器，组合与聚合

Qt 的值容器是一致值(同一种类型的值)的容器，比如 QString, byte, int, float 等。指针容器是指向(同一种多态类型化的)对象的容器。它们可以是受管理的，也可以是未受管理的。

在需要时，这两种容器都能够在运行时通过分配更多的堆内存来使其容量变大。分配堆内存时是以一种能够安全地处理异常的方式进行的，所以无须担心内存泄漏。

但是，当对堆对象使用指针容器时，必须决定由哪一个类来负责管理这些堆对象。通过使用组合(实心菱形)和聚合(空心菱形)，UML 框图可以区分托管容器和非托管容器。如图 6.6 所示。

通常而言，托管容器为组合类型，因为容器会管理它所指向的对象。换句话说，当组合关系被销毁时，也会完整地销毁(清除)自身，因为较小的对象是组合的一部分。

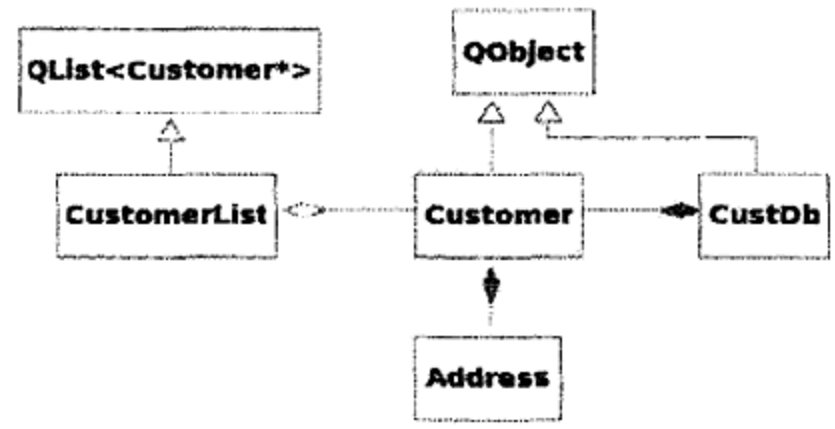


图 6.6 组合和聚合

① 参见 ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf。

当一个对象将另一个对象作为它的子对象被嵌入时，也会认为它们是一种组合关系。

图 6.6 中存在两种类型的 Customer 容器：CustomerList 和 CustDb，它们复用了模板容器。CustomerList 对象为聚合，即一种保存查询结果或者用户选择的临时结构。另一方面，CustDb 是一个独立的组合，它管理存在的全部 Customer 对象。

对于 Customer 和 Address 关系，这个框图表明一个或者多个 Address 对象应该与一个特定的 Customer 相关联。当 Customer 对象被销毁时，应该同时销毁它的所有 Address 对象。因此，Customer 对象管理它的 Address 对象，这是组合关系的又一个例子。

注意

这种设计对 Address 的可能用法强加了一些限制，特别地，不存在一种简单的办法来找到某个 Address 处的全部 Customer。如果 Address 和 Customer 被彼此独立地管理，则可以在这些类之间形成一种双向关系。

通常而言，当托管容器被销毁时，也会删除它所“拥有”的任何堆对象。对于 Qt 指针容器，可以使用 qDeleteAll(container)，它是一个对容器中的每一个元素调用 delete 函数的算法。

对托管容器的复制操作，可以有多种定义方式：

- 对某些容器，可以禁用这个复制特性。
- 对于其他容器，复制操作可以被定义成深度复制，所包含的全部对象都会被复制并放入一个新容器中。
- 另一种方法为隐式共享，它针对的是 Qt 容器的设计，将在 11.5 节中探讨。

聚合容器是一种只对它的内容提供索引或者引用导航机制的容器。

这种情况下，容器不管理它的对象，而只提供一种访问对象的便利途径。当复制聚合容器时，只会复制所包含对象的引用；当删除聚合容器时，只会移除这些引用，而位于容器底层的对象不会受到影响。

注意

托管容器是一个组合，而非托管容器在 UML 框图中通常被表示成聚合(但不总是这样)。

6.9.1　练习：托管容器，组合与聚合

扑克牌已经以各种形式存在 600 多年了，它被用于大量的机会游戏中，也是数学、统计学和计算机科学中受欢迎的练习题主角。

在欧洲和西方国家，有一种称为 deck(一副牌)的标准扑克牌，许多人对它并不陌生。deck 由 52 张牌组成，被分成 4 组，每一组称为一个 suit(花色)。每一种花色由 13 张牌组成，它们的名称分别为 A, 2, 3, 4, 5, 6, 7, 8, 9, T(10), J, Q 和 K。许多扑克牌游戏开始时都只向每一位玩家发很少的几张牌(从 deck 中随机抽取)，称为 hand(一手牌)。

这个练习中，需要设计数据类型来表示 deck 和 hand。后面将再次访问这些类，以制订规则并添加图形。图 6.7 中给出了表示这些类的一种方式。

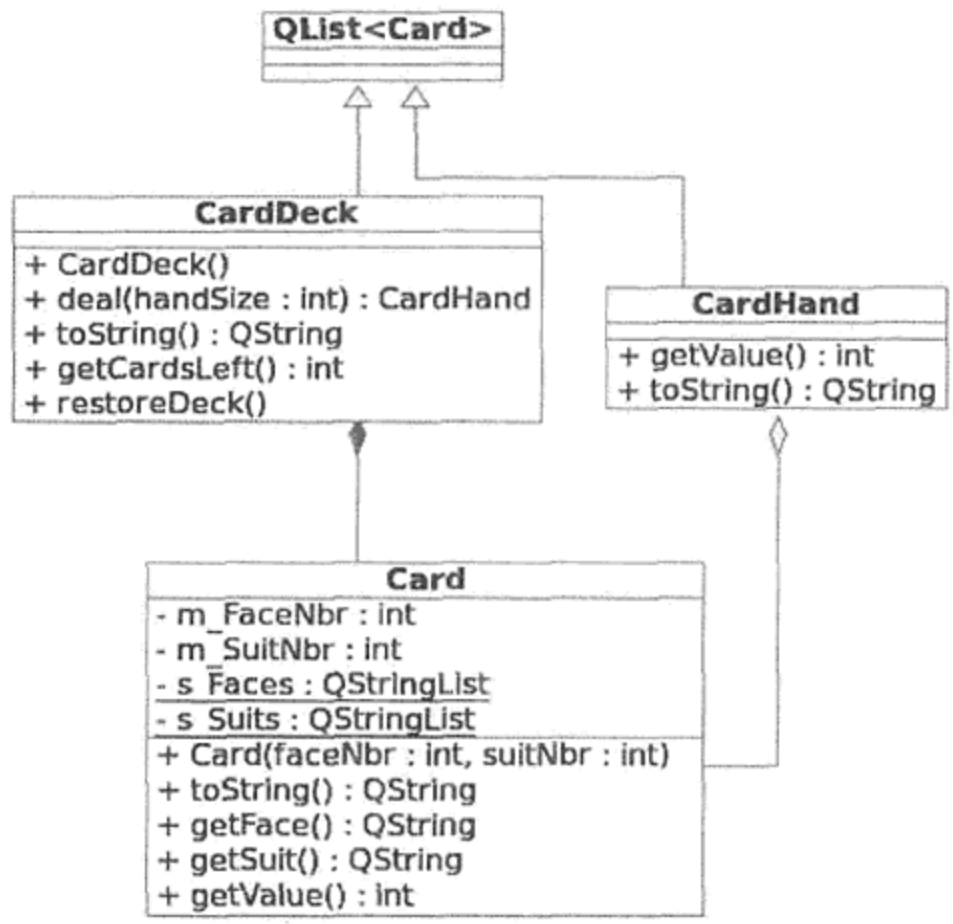


图 6.7 扑克牌游戏的 UML 框图

以下是一些提示。

- `CardDeck` 构造函数以一种方便的顺序产生一副完整的牌。
- `CardDeck::deal(int k)` 应使用来自于<cstdlib>的 `random()` 函数，从一副牌中挑选出 `k` 个 `Card` 对象(每选出一个，就将其移走)，用这些 `Card` 对象填充 `CardHand` 对象。
- 根据系统时间初始化 `random()` 函数，以便每次运行这个应用时都会得到不同的结果。语法为：

```
srandom(time(0));
```

- 利用桥牌的规则来求值 hand：A = 4, K = 3, Q = 2, J = 1，所有其他的牌都为 0 值。可以使用这个公式来计算 `getValue()` 函数的返回值。
- 示例 6.29 中的客户代码可用于测试这些类。

示例 6.29 src/cardgame/datastructure/cardgame-client.cpp

```
[ . . . . ]
#include "carddeck.h"
#include <QTextStream>
#include <QtGui>
int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    QTextStream cout(stdout);
    CardDeck deck;
    CardHand hand;
    int handSize, playerScore, progScore;
    cout << "How many cards in a hand? " << flush;
    handSize = QInputDialog::getInt(0, QString("getInt()"),
        QString("How many cards in hand?"), 1, 5);
    QMessageBox::StandardButton sb;
    do {
        hand = deck.deal(handSize);
```

```
    cout << "Here is your hand:" << endl;
    cout << hand.toString() << endl;
    playerScore = hand.getValue();
    cout << QString("Your score is: %1 points.")
                    .arg(playerScore) << endl;
    // Now a hand for the dealer:
    hand = deck.deal(handSize);
    progScore = hand.getValue();
    cout << "Here is my hand:" << endl;
    cout << hand.toString() << endl;
    cout << QString("My score is: %1 points.")
                    .arg(progScore) << endl;
    cout << QString("%1 win!!")
          .arg((playerScore > progScore)?"You":"I") << endl;
    sb = QMessageBox::question(0, QString("QMessageBox::question()"),
        QString("Another hand?"), QMessageBox::Yes | QMessageBox::No);
    } while (sb == QMessageBox::Yes);
}
```

6.10　指针容器

1.15.1 节中讲过，所有的指针都具有相同的大小(较小)，这就是应该使用指针容器而不是对象容器的唯一理由。本书的后面将用到的许多类都是继承关系树中的成员，尤其是那些用于 GUI 编程的类。正如本章中的许多示例那样，基类指针可以包含派生对象的地址。这样，基类指针的容器就能够包含任何派生对象的地址。进而，多态使得在运行时能够通过这些指针调用合适的函数。为了使用多态，必须在基类中为希望调用的每一个函数定义一个原型，即使该函数不能在基类中定义也应该如此。这就是有时为什么需要纯 virtual 函数的原因。基类提供的接口能够用于具体的派生对象。

指针容器要求小心地设计析构过程，以避免内存泄漏。此外，对指针的访问和维护必须小心地控制，以防止解引用空指针或者未定义的指针。这项工作并不如想像的那样困难。

- 当向容器添加指针时，必须确保它已经被立即初始化了。如果不方便初始化，则应将其赋值为 0。
- 当某个指针不再需要时，应移走并删除它。如果由于某些原因不方便立即将其移走，则被删除的指针应被重新赋值或者设置成 0。
- 销毁 Qt 指针容器时，应调用 qDeleteAll() 函数。
  - qDeleteAll() 函数是针对全部 Qt 容器类的通用算法。
  - 每一个特定的 Qt 容器类都有自己的 clear() 成员函数。

复制，复制，复制

使用指针时会导致另一个重要的问题出现，尤其是当使用指针容器时。一定要记住的是，如果允许复制包含指针的对象，则可能会导致错误出现。让编译器提供复制构造函数和复制赋值构造函数几乎总会造成这种灾难，它们都只会简单地复制主对象的指针。利用这种方法，如果指针容器(或者包含指针成员的对象)在函数调用中被当作一个实参传递给值参数时，这个函数将会产生一个该对象的局部副本，当函数返回时副本会被销毁。如果其析构函数正常

地删除了这些指针，则原始对象将会包含一个或者多个指向已被删除的内存区的指针，从而会导致极其难于跟踪的内存冲突问题。

避免出现这种情况的一种方法是确保复制构造函数和复制赋值构造函数都对主对象进行深度复制。即，位于新复制地址内存中的指针，包含的是主对象的指针所指向的数据的精确副本。遗憾的是，这种方法对系统资源是一种极大的消耗，通常是被禁止的。11.5 节中探讨了一种利用资源共享的更有效方法。

前面提到的 QObject 类可以包含指针容器，其处理方法是让复制构造函数和复制赋值构造函数为 private 类型的。这样，进行复制的任何企图都会导致编译错误[①]。

在第 8 章及后续的几章中，将有机会接触到各种指针类型的容器。在那些地方，QObject 指针的容器能够包含各种不同对象的地址，包括用于图形用户界面(GUI)的窗件(widget)。在第 9 章中，当用许多不同类型的窗件和布局(它们都为 QObject)编写 GUI 程序时，基类指针的容器将扮演至关重要的角色。

补充示例：一个简单的图书馆

对于这个示例，我们将图书馆当成各种参考资料的一个汇总。首先，定义几个类来是实现一个简化了的图书馆，它以图 6.8 中经过缩减的 UML 框图为基础。

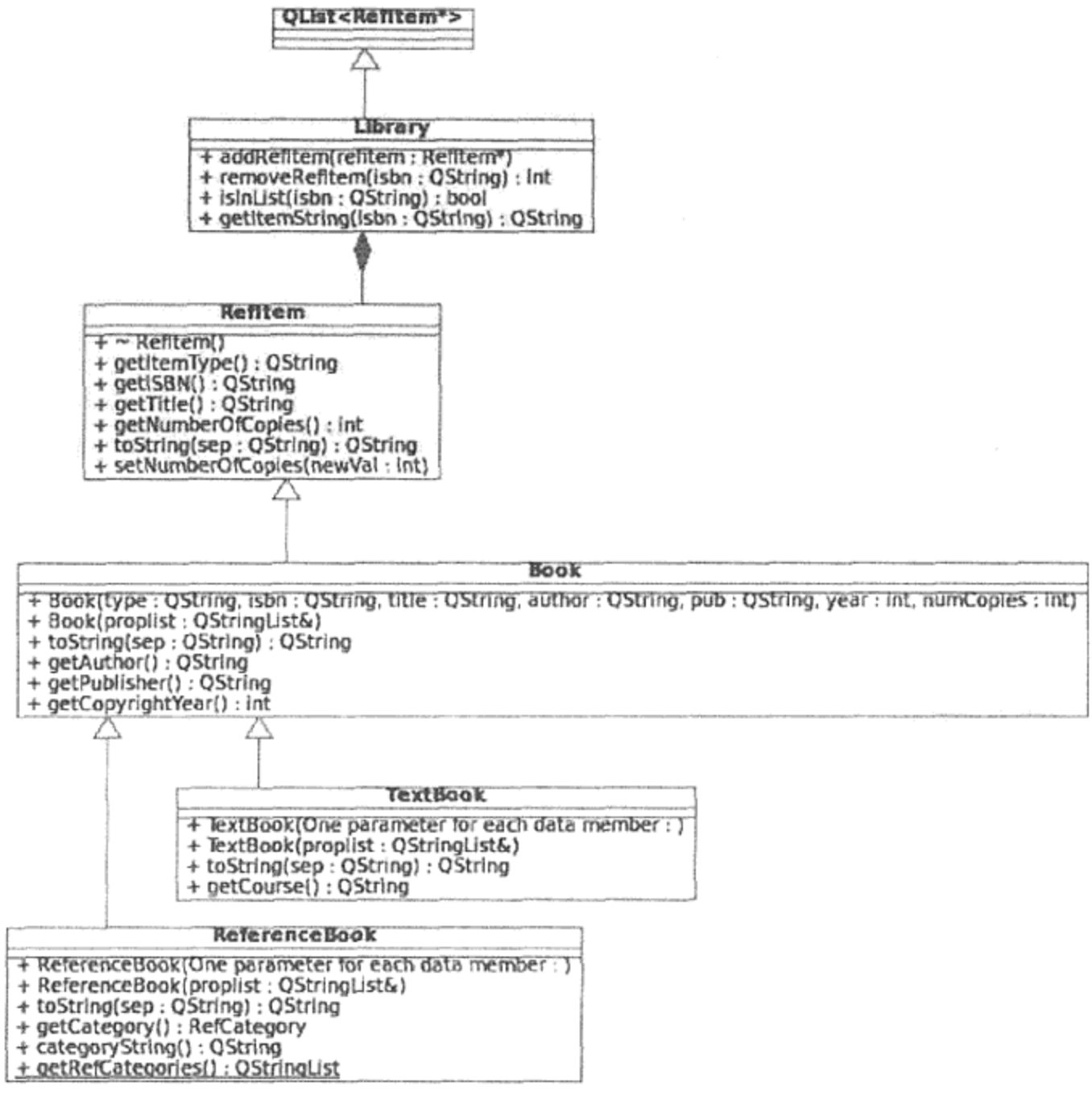


图 6.8 参考资料图书馆的 UML 框图

① 参见 8.1 节。

示例 6.30 中给出了基类的定义。由于它的构造函数都为 protected 类型的，所以客户代码无法构造 RefItem 对象。因此，RefItem 是一个抽象基类。

示例 6.30　src/pointer-container/library.h

```
[ . . . . ]

class RefItem {
public:
   virtual ~RefItem();
   QString getItemType() const;
   QString getISBN() const;

   QString getTitle() const;
   int getNumberOfCopies() const;
   virtual QString toString(QString sep="[::]") const;
   void setNumberOfCopies(int newVal);
protected:
   RefItem(QString type, QString isbn, QString title, int numCopies=1);
   RefItem(QStringList& proplist);
private:
   QString m_ItemType, m_ISBN, m_Title;
   int m_NumberOfCopies;
};
```

示例 6.31 中给出了一些派生类的定义。基类和每一个派生类都有一个构造函数，其唯一的参数为一个 QStringList 引用。当从文件读取数据或者从用户处获取信息时，这会极大地简化和方便对象的创建。RefCategory 为 enum 类型，是在 ReferenceBook 里定义的一种 public 对象，其作用是各种图书类别的枚举，比如文学、音乐、数学、科学、艺术、建筑等。在这些类中不存在任何输入/输出(I/O)操作。对于这个示例，所有的 I/O 都是由客户代码处理的，将在练习中完成它。

示例 6.31　src/pointer-container/library.h

```
[ . . . . ]

class Book : public RefItem {
public:
   Book(QString type, QString isbn, QString title, QString author,
        QString pub, int year, int numCopies=1);
   Book(QStringList& proplist);
   virtual QString toString(QString sep="[::]") const;
   QString getAuthor() const;
   QString getPublisher() const;
   int getCopyrightYear() const;
private:
   QString m_Author, m_Publisher;
   int m_CopyrightYear;
};

class ReferenceBook : public Book {
public:
   enum RefCategory {NONE = -1, Art, Architecture, ComputerScience, Literature,
                     Math, Music, Science};
```

```
    ReferenceBook(QString type, QString isbn, QString title, QString author,
         QString pub, int year, RefCategory refcat, int numCopies=1);
    ReferenceBook(QStringList& proplist);
    QString toString(QString sep="[::]") const;
    RefCategory getCategory() const;
    QString categoryString() const; //returns string version of m_Category
    static QStringList getRefCategories();  //returns a list of categories
private:
    RefCategory m_Category;
};
```

大多数实现代码都是非常固定的，因此这里将其省略。我们的重点是用来传输和接收数据的技术。由于数据的传输和接收可以有多种途径(例如，通过文件、网络等)，所以这里提供两种类型的转换：对象到 QString 和 QStringList 到对象。针对 I/O 的具体细节则放到了这些类之外的地方。示例 6.32 展示的是对象到 Qstring 的转换过程。

示例 6.32 src/pointer-container/library.cpp

```
[ . . . . ]

QString RefItem::toString(QString sep) const {
    return
    QString("%1%2%3%4%5%6%7").arg(m_ItemType).arg(sep).arg(m_ISBN).arg(sep)
                             .arg(m_Title).arg(sep).arg(m_NumberOfCopies);
}
[ . . . . ]

QString Book::toString(QString sep) const {
    return QString("%1%2%3%4%5%6%7").arg(RefItem::toString(sep)).arg(sep)
                .arg(m_Author).arg(sep).arg(m_Publisher).arg(sep)
                .arg(m_CopyrightYear);
}
[ . . . . ]

QString ReferenceBook::toString(QString sep) const {
    return QString("%1%2%3").arg(Book::toString(sep)).arg(sep)
                            .arg(categoryString());
}
[ . . . . ]

QString ReferenceBook::categoryString() const {
    switch(m_Category) {
      case Art: return "Art";
      case Architecture: return "Architecture";
      case ComputerScience: return "ComputerScience";
      case Literature: return "Literature";
      case Math: return "Math";
      case Music: return "Music";
      case Science: return "Science";
    default: return "None";
    }
}
```

QString 提供了传输数据的方便途径。如果 QString 由多块数据组成，且这些数据已

经被小心地放到一起，则可以利用 QString::split(QString separator)函数将它们重新打包成一个 QStringList。在这个示例中，数据项的顺序和分隔符由 toString(QString sep)函数确定。仔细研究示例 6.33，深入思考一下每一个构造函数中非 const 引用参数的用法。所有的操作都是在成员初始化表中发生的。

示例 6.33　src/pointer-container/library.cpp

```
[ . . . . ]

RefItem::RefItem(QStringList& plst) : m_ItemType(plst.takeFirst()),
         m_ISBN(plst.takeFirst()), m_Title(plst.takeFirst()),
         m_NumberOfCopies(plst.takeFirst().toInt())
{ }
[ . . . . ]

Book::Book(QStringList& plst) : RefItem(plst), m_Author(plst.takeFirst()),
         m_Publisher(plst.takeFirst()), m_CopyrightYear(plst.takeFirst().toInt())
{ }
[ . . . . ]

ReferenceBook::ReferenceBook(QStringList& plst) : Book(plst),
               m_Category(static_cast<RefCategory>(plst.takeFirst().toInt()))
{ }
```

示例 6.34 中包含了 Library 类的定义。由于 public 类型的 Library 是从 QList-<RefItem*>派生的，所以它是一个指针容器。复制构造函数和复制赋值运算符都为 private 类型。这可以阻止编译器提供它们的 public 版本(针对这种灾难的解决办法前面已经提到过)，从而可以确保不会存在 Library 对象的副本。这也会阻止编译器提供默认构造函数，因此程序员必须提供一个。

示例 6.34　src/pointer-container/library.h

```
[ . . . . ]

class Library : public QList<RefItem*> {
public:
   Library() {}
   ~Library();
   void addRefItem(RefItem*& refitem);
   int removeRefItem(QString isbn);
   QString toString(QString sep="\n") const;
   bool isInList(QString isbn);
   QString getItemString(QString isbn);
private:
   Library(const Library&);
   Library& operator=(const Library&);
   RefItem* findRefItem(QString isbn);
};
```

1 指针容器必须有析构函数!

Library 类成员函数的实现在下面给出。示例 6.35 中的第一部分给出的是复制构造函数和复制赋值运输发的实现，此外还包括向 Library 添加和移除项目的实现。复制构造函数和

复制赋值运算符从来不会用到，这里为它们提供的实现，仅仅是为了防止编译器发出没有在构造函数中初始化基类的警告，或者警告赋值运算符没有任何返回值。

在将项目添加到列表中之前，需检查该项目是否已经准备好。如果项目已经在列表中，则只需将它的m_NumberOfCopies增加1。为了移除某项目，需将它的m_NumberOfCopies减1。如果减1后的结果为0，则从列表中移除它并删除它的指针[①]。

示例 6.35 src/pointer-container/library.cpp

```
[ . . . . ]

Library::~Library() {
   qDeleteAll(*this);
   clear();
}

Library::Library(const Library&) : QList<RefItem*>() {}

Library& Library::operator=(const Library&) {
   return *this;
}

void Library::addRefItem(RefItem*& refitem) {                          1
   QString isbn(refitem->getISBN());
   RefItem* oldItem(findRefItem(isbn));
   if(oldItem==0)
      append(refitem);
   else {
      qDebug() << isbn << " Already in list:\n"
               << oldItem->toString()
               << "\nIncreasing number of copies "
               << "and deleting new pointer." ;
      int newNum(oldItem->getNumberOfCopies() + refitem->getNumberOfCopies());
      oldItem->setNumberOfCopies(newNum);
      delete refitem;                                                  2
      refitem = 0;                                                     3
   }
}

int Library::removeRefItem(QString isbn) {
   RefItem* ref(findRefItem(isbn));
   int numCopies(-1);
   if(ref) {
      numCopies = ref->getNumberOfCopies() - 1;
      if(numCopies== 0) {
         removeAll(ref);
         delete ref;
      }
      else
         ref->setNumberOfCopies(numCopies);
```

① 想一想，为什么不先删除指针然后再移除项目？

```
    }
    return numCopies;
}
```

1 参数为一个指针引用，所以删除之后有可能是一个空赋值。

2 没有位于托管容器中。

3 引用参数!

对于更真实的应用而言，Library 类和 RefItem 类将需要更多的数据和函数成员。示例 6.36 为激发灵感而提供了更多的 Library 成员函数。Library::findRefItem()为 private 类型，因为它返回一个指针，而且正如前面提到过的，让客户代码操作指针通常不是一个好主意。

示例 6.36　src/pointer-container/library.cpp

```
[ . . . . ]

RefItem* Library::findRefItem(QString isbn) {
   for(int i = 0; i < size(); ++i) {
      if(at(i)->getISBN().trimmed() == isbn.trimmed())
         return at(i);
   }
   return 0;
}


bool Library::isInList(QString isbn) {
   return findRefItem(isbn);
}

QString Library::toString(QString sep) const {
   QStringList reflst;
   for(int i = 0; i < size(); ++i)
      reflst << at(i)->toString();
   return reflst.join(sep);
}

QString Library::getItemString(QString isbn) {
   RefItem* ref(findRefItem(isbn));
   if(ref)
      return ref->toString();
   else
      return QString();
}
```

注意

在 Library::findRefItem()和 Library::toString()的实现中，并没有使用 foreach()宏的选项，因为这个宏需要复制它所遍历的容器。由于复制构造函数为 private 类型，所以这是不可能的。当在后面的几章中使用 QObject 时，一定要记住这一点。

下面编写客户代码，以测试这些类。由于使用的是标准 I/O，所以引入了一些 enum 数据类型，以便能够更容易地设置菜单系统，正如示例 6.37 中那样。

示例 6.37 src/pointer-container/libraryClient.cpp

```
[ . . . . ]

QTextStream cout(stdout);
QTextStream cin(stdin);
enum Choices {READ=1, ADD, FIND, REMOVE, SAVE, LIST, QUIT};
enum Types {BOOK, REFERENCEBOOK, TEXTBOOK, DVD, FILM, DATADVD};
const QStringList TYPES = (QStringList() << "BOOK" << "REFERENCEBOOK"
   << "TEXTBOOK" << "DVD" << "FILM" << "DATADVD");
bool saved(false);
```

示例 6.38 演示了应该如何使用这些 enum 数据类型。

示例 6.38 src/pointer-container/libraryClient.cpp

```
[ . . . . ]

Choices nextTask() {
   int choice;
   QString response;
   do {
      cout << READ << ". Read data from a file.\n"
           << ADD << ". Add items to the Library.\n"
           << FIND << ". Find and display an item.\n"
           << REMOVE << ". Remove an item from the Library.\n"
           << SAVE << ". Save the Library list to a file.\n"
           << LIST << ". Brief listing of Library items.\n"
           << QUIT << ". Exit from this program.\n"
           << "Your choice: " << flush;
     response = cin.readLine();
     choice = response.toInt();
   } while(choice < READ or choice > QUIT);
   return static_cast<Choices>(choice);
}

void add(Library& lib, QStringList objdata) {
   cout << objdata.join("[::]") << endl;
   QString type = objdata.first();
   RefItem* ref;
   switch(static_cast<Types>(TYPES.indexOf(type))) {
   case BOOK:
      ref = new Book(objdata);
      lib.addRefItem(ref);
      break;
   case REFERENCEBOOK:
      ref = new ReferenceBook(objdata);
      lib.addRefItem(ref);
         break;
[ . . . . ]

   default: qDebug() << "Bad type in add() function";
   }
}
```

从示例 6.39 可以看出，将数据保存到文件中是非常简单的。

示例 6.39　src/pointer-container/libraryClient.cpp

```
[ . . . . ]

void save(Library& lib) {
   QFile outf("libfile");
   outf.open(QIODevice::WriteOnly);
   QTextStream outstr(&outf);
   outstr << lib.toString();
   outf.close();
}
```

示例 6.40 是从文件读取数据，每次一行。这个方法只适合于 Library::toString() 采用换行符来分隔两个对象的数据的情况。

示例 6.40　src/pointer-container/libraryClient.cpp

```
[ . . . . ]

void read(Library& lib) {
   const QString sep("[::]");
   const int BADLIMIT(5); //max number of bad lines
   QString line, type;
   QStringList objdata;
   QFile inf("libfile");
   inf.open(QIODevice::ReadOnly);
   QTextStream instr(&inf);
   int badlines(0);
   while(not instr.atEnd()) {
      if(badlines >= BADLIMIT) {
         qDebug() << "Too many bad lines! Aborting.";
         return;
      }
      line = instr.readLine();
      objdata = line.split(sep);
      if(objdata.isEmpty()) {
         qDebug() << "Empty Line in file!";
         ++badlines;
      }
      else if(not TYPES.contains(objdata.first())) {
         qDebug() << "Bad type in line: " << objdata.join(";;;");
         ++badlines;
      }
      else
         add(lib, objdata);
   }
}
```

从键盘获取数据要复杂一些，因为必须请求每一个数据项，且要尽可能地对它进行验证。在客户代码中(如示例 6.41 所示)，每一个 RefItem 类都有一个 prompt 函数，它返回的 QStringList 可以传递给这个类中合适的构造函数。

示例 6.41 src/pointer-container/libraryClient.cpp

```
[ . . . . ]

QStringList promptRefItem() {
   const int MAXCOPIES(10);
   const int ISBNLEN(13);
   int copies;
   QString str;
   QStringList retval;
   while(1) {
      cout << "ISBN ("<< ISBNLEN << " digits): " << flush;
      str = cin.readLine();
      if(str.length() == ISBNLEN) {
         retval << str;
         break;
      }
   }
   cout << "Title: " << flush;
   retval << cin.readLine();
   while(1) {
      cout << "Number of copies: " << flush;
      copies = cin.readLine().toInt();
      if(copies > 0 and copies <= MAXCOPIES) {
         str.setNum(copies);
         break;
      }
   }
   retval << str;
   return retval;
}

QStringList promptBook() {
   static const int MINYEAR(1900), MAXYEAR(QDate::currentDate().year());
   int year;
   QStringList retval(promptRefItem());
   QString str;
   cout << "Author: " << flush;
   retval << cin.readLine();
   cout << "Publisher: " << flush;
   retval << cin.readLine();
   while(1) {
      cout << "Copyright year: " << flush;
      year = cin.readLine().toInt();
      if(year >= MINYEAR and year <= MAXYEAR) {
         str.setNum(year);
         break;
      }
   }
   retval << str;
   return retval;
}
```

```
QStringList promptReferenceBook() {
   int idx(0);
   bool ok;
   QString str;
   QStringList retval(promptBook());
   QStringList cats(ReferenceBook::getRefCategories());
   while(1) {
      cout << "Enter the index of the correct Reference Category: ";
      for(int i = 0; i < cats.size(); ++i)
         cout << "\n\t(" << i << ") " << cats.at(i);
      cout << "\n\t(-1)None of these\t:::" << flush;
      idx = cin.readLine().toInt(&ok);
      if(ok) {
         retval << str.setNum(idx);
         break;
      }
   }
   return retval;
}
[ . . . . ]

void enterData(Library& lib) {
   QString typestr;
   while(1) {
      cout << "Library item type: " << flush;
      typestr = cin.readLine();
      if(not TYPES.contains(typestr)) {
         cout << "Please enter one of the following types:\n"
              << TYPES.join(" ,") << endl;
         continue;
      }
      break;
   }
   QStringList objdata;
   switch (TYPES.indexOf(typestr)) {
   case BOOK: objdata = promptBook();
         break;
   case REFERENCEBOOK: objdata = promptReferenceBook();
         break;
[ . . . . ]

   default:
         qDebug() << "Bad type in enterData()";
   }
   objdata.prepend(typestr);
   add(lib, objdata);
}
```

示例 6.42 中给出了 `main()` 函数的定义。

示例 6.42　src/pointer-container/libraryClient.cpp

```
[ . . . . ]

int main() {
```

```
Library lib;
while(1) {
   switch(nextTask()) {
   case READ: read(lib);
      saved = false;
      break;
   case ADD: enterData(lib);
      saved = false;
      break;
   case FIND: find(lib);
      break;
   case REMOVE: remove(lib);
      saved = false;
      break;
   case SAVE: save(lib);
      saved = true;
      break;
   case LIST: list(lib);
      break;
   case QUIT: prepareToQuit(lib);
      break;
   default:
      break;
   }
  }
}
```

6.10.2 节中将优化并改善这个应用。

6.10.1 练习：指针容器

1. 假设要为汽车零部件编写一个库存控制系统。
   - 编写一个 UML 类框图，基类名称为 AutoPart，子类名称可以是 EnginePart，BodyPart，Accessory 等，而具体的零部件类名称可以是 Alternator，Fender，Radiator，SeatBelt 等。
   - 编写(但不实现)这些类的类定义。需要哪些类型的基类函数?
   - 应该如何保证只有具体的零部件类才能够被实例化?
2. 图 6.9 中的这些类是用于整理某学院图书馆中的电影胶片藏品的。
   a. 实现 Film 类。应确保构造函数具有足够的参数来初始化全部的数据成员。建议在 Entertainment 类中使用 enum 类型 FilmTypes(Action, Comedy, SciFi, …)和 MPAARatings(G, PG, PG-13, …)。
   b. 将 FilmList 类实现成 Film 指针的容器。确保 addFilm()函数不允许同一个电影胶片被添加多次。
   这里可以使用基类函数吗? 例如 contains()或者 indexOf()。
   c. 编写客户代码，测试这些类。将 Entertainment 和 Educational 电影胶片同时放入 FilmList 中，练习全部的成员函数。

如果使用指针容器，则会出现如何销毁它的问题。如果只是按照“常规的方式”处理，定义一个 FilmList 析构函数来访问每一个指针并删除它，则一定会担心客户代码

中有一个函数具有 FilmList 值参数。前面说过，每一个值参数都会导致对应的实参被复制。函数返回时，会销毁这个实参副本。该如何处理 FilmList 对象的复制和销毁？

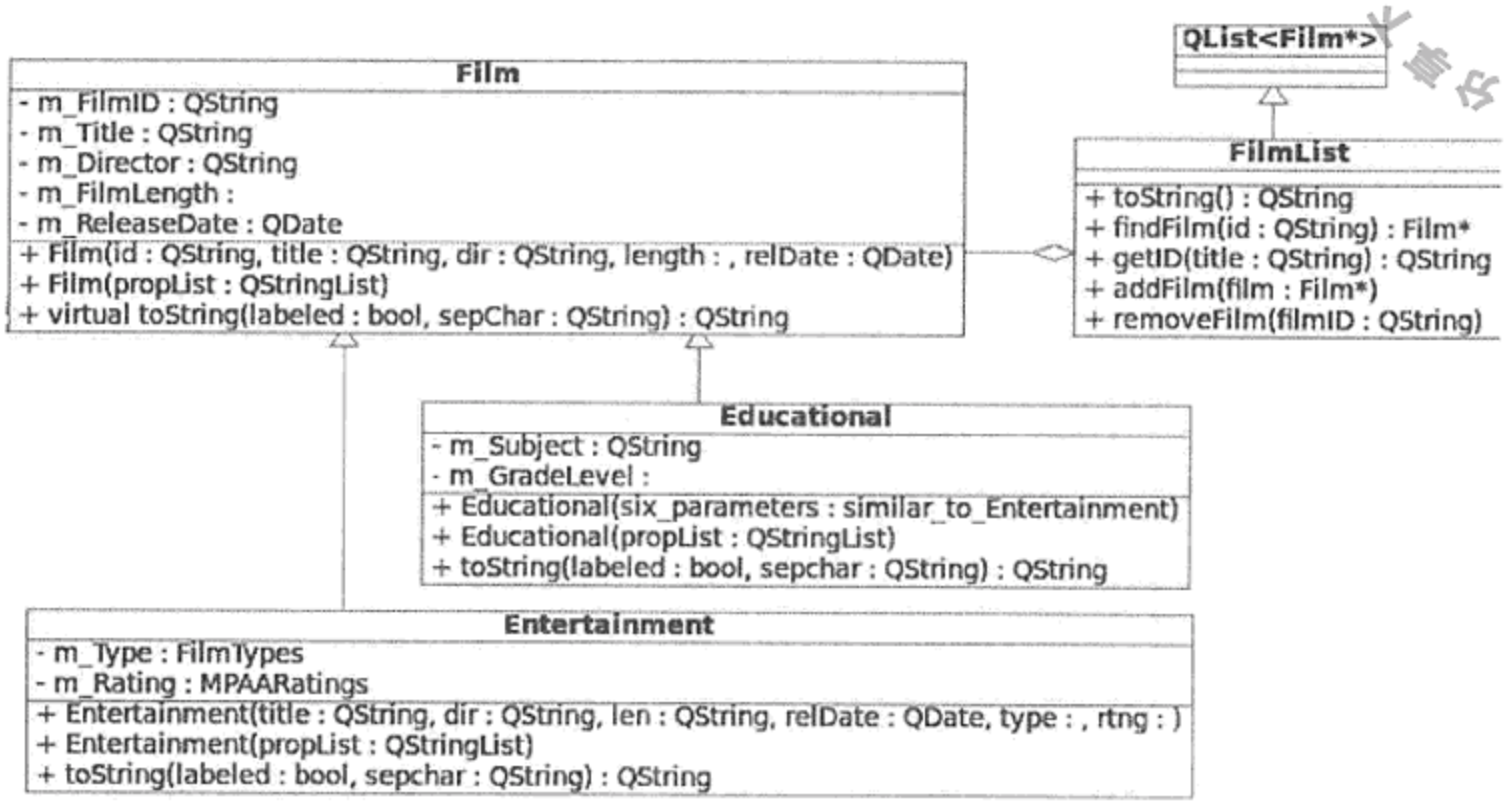


图 6.9　电影胶片类

FilmList 类使程序员能够利用多态，但会向客户代码暴露 Film 指针。因此，这种设计违背了前面提到的使用指针的警告。通常而言，指针应对客户代码隐藏。16.3 节中将探讨解决这一问题的一种方法。

3. 优化补充示例(一个简单的图书馆)中的解决方案：添加一个 LibraryUI 类，它处理与用户的交互，如图 6.10 所示。

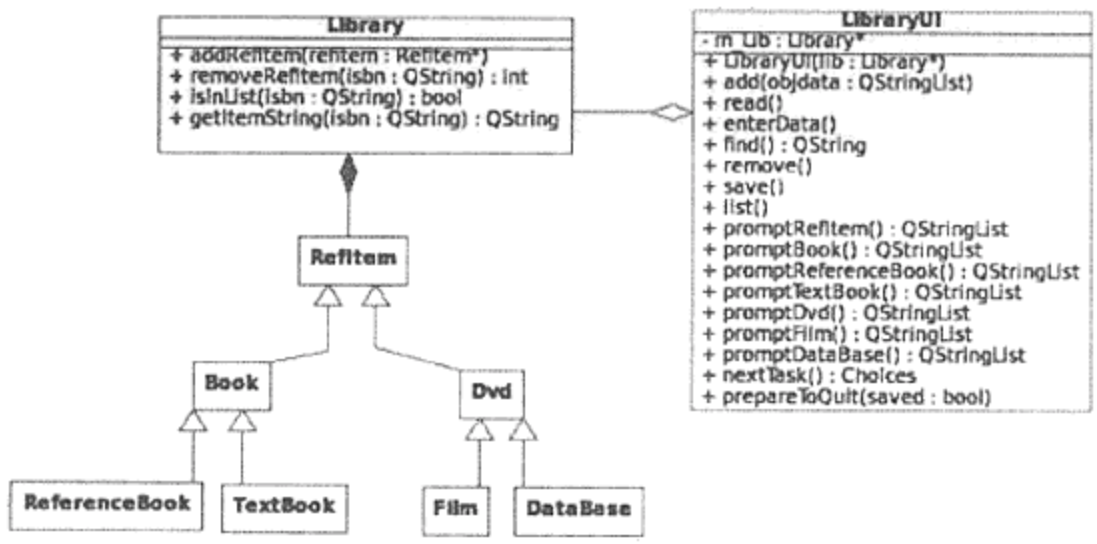


图 6.10　图书馆(版本 2)

当添加这个类时，应使用示例 6.43 中的客户代码来测试系统。

示例 6.43　src/pointer-container/libraryClient-v2.cpp

```
#include "libraryui.h"
#include "library.h"

bool saved(false);

int main() {
   Library lib;
```

```
LibraryUI libui(&lib);
while(1) {
   switch(libui.nextTask()) {
   case LibraryUI::READ: libui.read();
      saved = false;
      break;
   case LibraryUI::ADD: libui.enterData();
      saved = false;
      break;
   case LibraryUI::FIND: libui.find();
      break;
   case LibraryUI::REMOVE: libui.remove();
      saved = false;
      break;
   case LibraryUI::SAVE: libui.save();
      saved = true;
      break;
   case LibraryUI::LIST: libui.list();
      break;
   case LibraryUI::QUIT: libui.prepareToQuit(saved);
      break;
   default:
      break;
   }
  }
}
```

4. 优化问题 3 中的解决方案，使每一个类都有自己对应的 UI 类，如图 6.11 所示。
5. 改变 Library 类的实现，使其派生自 QMap<QString, RefItem*>，其中 QString 键是图书的 ISBN(书号)。

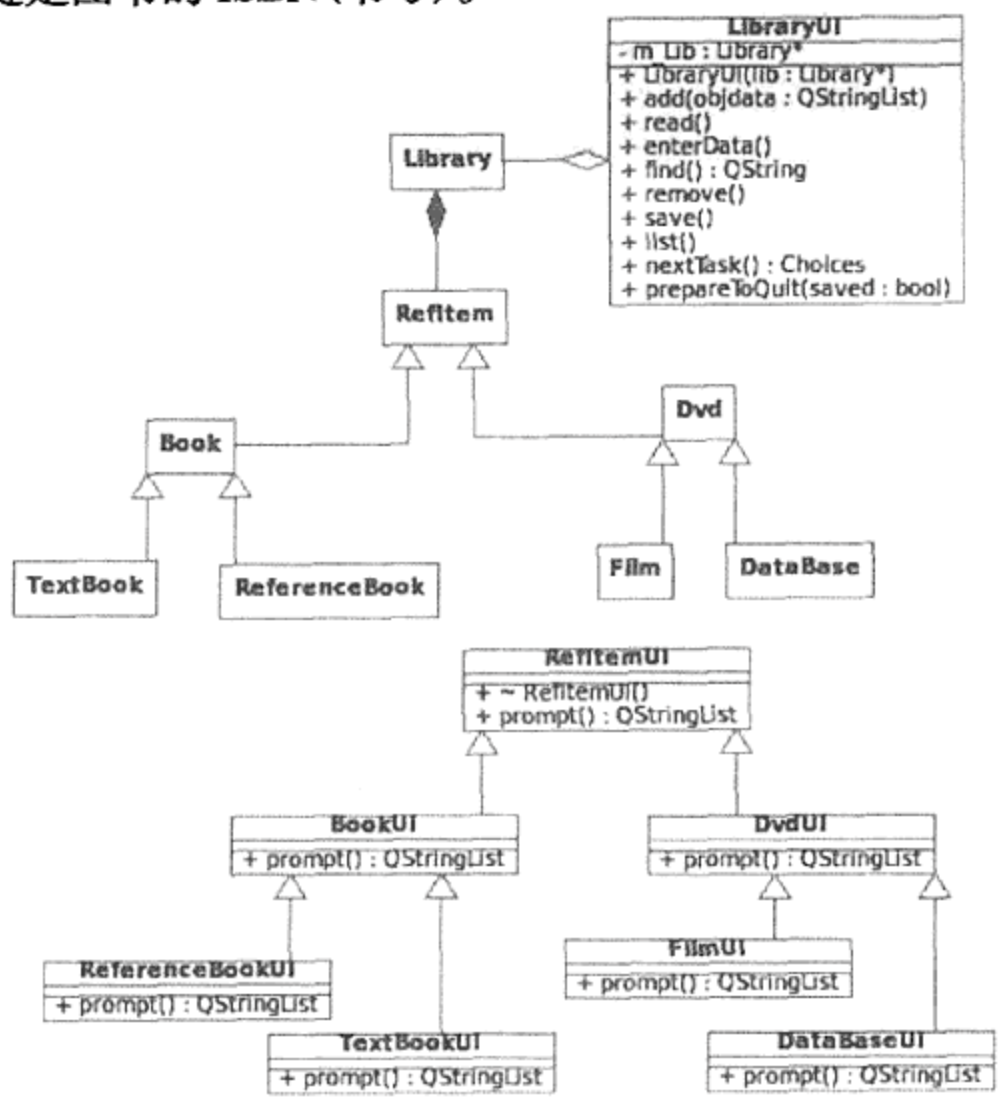


图 6.11　图书馆(版本 3)

6.11　复习题

1. 函数与方法有什么不同?
2. 隐藏基类函数有何意义? 如何才能隐藏基类函数?
3. 哪些成员函数不能从基类继承? 给出原因。
4. 根据示例 6.44 中的定义，回答示例 6.45 中的问题。

示例 6.44　src/quizzes/virtual-quiz.cpp

```
class Shape {
public:
    virtual void draw(Position p);
    virtual void draw(PaintEngine* pe, Position p);
};

class Square : public Shape {
public:
    void draw(Position p);
private:
    void draw(int x, int y);
};

int main() {
    Position p(4,3);
    Position q(5,6);
    PaintEngine *pe = .....;
    Square sq;

    sq.draw(p);                                          1
    sq.draw(pe, p);                                      2
    sq.draw(3,3);                                        3

    Shape* sp = &sq;
    sp->draw(q);                                         4
    sp->draw(pe, q);                                     5
    sp->draw(3,2);                                       6

}
```

1 _____

2 _____

3 _____

4 _____

5 _____

6 _____

示例 6.45　src/quizzes/virtual-questions.txt

1. 调用的是哪一个方法?
   a. `Shape::draw()`

b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

2. 调用的是哪一个方法?
a. Shape::draw()
b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

3. 调用的是哪一个方法?
a. Shape::draw()
b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

4. 调用的是哪一个方法?
a. Shape::draw()
b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

5. 调用的是哪一个方法?
a. Shape::draw()
b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

6. 调用的是哪一个方法?
a. Shape::draw()
b. Square::draw()
c. 错误，方法被隐藏
d. 错误，方法不可访问
e. 错误，没有这样的方法

5. 考虑另一种形状类并回答如下问题。答案可能有多个。

示例 6.46 src/quizzes/abstract-quiz.cpp

```
/* Consider the following header file, assume the
function definitions are in a .cpp file somewhere. */
class Shape {
public:
    explicit Shape(Point origin);
    virtual void draw(PaintDevice* pd) = 0;
    virtual void fill(PaintDevice* pd) = 0;
    virtual String name() const;
    Point origin() const;
private:
    Point m_origin;
};
```

```
class Rectangle : public Shape {
public:
    Rectangle(Point origin, int width, int height);
    void draw(PaintDevice* pd);
private:
    int m_width, m_height;
};

class Square: public Rectangle {
public:
    Square(Point origin, int width);
};
```

1. 下列哪些方法在所有类中都为纯 virtual 方法?
   a. origin
   b. draw
   c. fill
   d. draw 和 fill
   e. draw, fill 和 name

2. 下列哪些类是抽象的?
   a. Shape
   b. Rectangle
   c. Square
   d. Shape 和 Rectangle
   e. 以上都是

3. 下列哪些构造函数的实现是有效的?

```
Rectangle::Rectangle(Point origin, int width, int height)

a. { m_width = width; m_height = height; }
b. : m_origin(origin), m_width(width), m_height(height) {}
c. : Shape(origin) {m_width = width; m_height = height; }
d. { m_origin = origin; m_width = width; m_height = height; }
e. : Shape(origin), m_width(width), m_height(height) {}
*/
```

6. 阅读示例 6.47 中的代码，并回答示例 6.48 中的问题。

示例 6.47　src/quizzes/virtual-destructors-quiz.cpp

```
#include <QDebug>

class Base {
public:
    Base() { ++sm_bases; }
    ~Base() { --sm_bases; }
    void a();
    virtual void b();
protected:
    static int sm_bases;
};

class Derived : public Base {
public:
    Derived() { ++sm_deriveds; }
```

```
    ~Derived() { --sm_deriveds; }
    void a();
    void b();
    static void showCounts() {
        qDebug() << sm_bases << sm_deriveds;
    }
protected:
    static int sm_deriveds;
};

int Base::sm_bases(0);
int Derived::sm_deriveds(0);
void Base::a() { qDebug() << "Base::a()" ;}
void Base::b() { qDebug() << "Base::b()" ;}
void Derived::a() { qDebug() << "Derived::a()" ;}
void Derived::b() { qDebug() << "Derived::b()" ;}

void foo() {
    Derived d1;
    Base b1;
    Base* bp = new Derived();
    bp->a();
    bp->b();
    delete bp;
    Derived::showCounts();                                    1
}
int main() {
    Base b;
    Derived d;
    foo();
    Derived::showCounts();                                    2
}
```

1 _____

2 _____

示例 6.48 src/quizzes/virtual-destructors-quiz.txt

1. 第一次调用 showCounts()后的输出是什么?
   a. 4 3
   b. 3 2
   c. 2 2
   d. 1 1
   e. 3 4
2. 第二次调用 showCounts()后的输出是什么?
   a. 1 2
   b. 1 1
   c. 0 0
   d. 2 2
   e. 2 1

7. 在补充示例(一个简单的图书馆)中，有可能会将项目添加到 ReferenceBook 类中的 RefCategory 枚举中。这样做时应该避免哪些陷阱？应该强制使用哪些规则来使得添加项目时不会遇到这些陷阱?

第7章　库与设计模式

库是一组代码模块，它按照可复用的方式组织。本章将探讨如何构建、复用和设计库。也会介绍和探讨设计模式。

术语“平台”指的是硬件体系结构与软件架构的一种特定组合，前者尤指中央处理单元(CPU)[①]，后者通常指操作系统(OS)[②]。每一种计算机系统都只能执行以其自己低级的、特定于平台的语言编写的代码。这种低级的机器语言与任何自然界的人类语言都不相同，几乎没有程序员能够方便地直接使用它。

为了最优化编程资源的使用(尤其是在程序员的时间)，我们需要使用高级语言(例如，C++)来编写程序，以便以尽量接近于自然语言(例如，英语)的形式来表达并分享思想和精确的指令。将高级语言代码翻译成机器语言以便能够在特定的计算机平台上执行的工作由编译器[③]负责。

对代码复用的价值的广泛认知，导致了对代码库(及其产品)的急剧需求。代码库中保存的是有用的、可复用的、编译后的代码，这样，程序员不必处理代码库的任何源代码就能够利用它的功能。当用“`#include`”指令包含库模块的头文件时，就可以复用这个库模块。这个指令会在适当的源代码模块中指定应用编程接口(API)。前面已经复用过来自于标准库和Qt的几个库模块，前者如`iostream`和`string`，后者如`QString`，`QTextStream`和`QList`。当复用来自于库中的任何模块时，其工作由链接器(linker)在链编(build)过程完成，以在编译后代码中的项目引用与编译后库代码中的项目定义之间构建适当的链接，得到的可执行文件必须在运行时找到并动态地链接到编译后的库(称为运行时库)。编译后的库代码并不需要集成到可执行文件中，因为它能够在运行时动态地链接。这样就可以得到更小的可执行文件，并能更高效地使用内存。

库(lib)是一个文件，它包含一个或者多个编译后文件(称为目标文件)，并对其进行了索引，以便链接器能够更容易地找到符号(例如，类的名称、类成员、函数、变量等)以及它们的定义。将多个目标文件集成在一个库中，能够显著提速链接过程。

C++库能够以多种途径被打包：

- 开源包
- `dev`包
- 运行时库

开源包通常以压缩的档案文件形式发布，它包含全部的源代码、头文件以及链编脚本和文档。`dev`包在Linux包管理程序中有时被称为“`-devel`”包，它通常以档案文件的形式发布，包含一个库以及相关的头文件。这种格式使得在发布库时可以不带源代码，而其他程序

---

① 例如，Intel Core 2 Duo 或者 SPARC64 “Venus”。

② 例如，Linux, Windows7, Mac_OS_X。

③ 通过将 Java 代码编译成称为字节码的中间状态，Java 编译器能够产生独立于平台的代码。然后，用特定于平台的 Java 虚拟机就可以将字节码翻译成机器语言。这种安排利用了平台独立性的许多优点，但在性能上有所损失。

员依然可以编译他们的应用。运行时库由 lib 文件组成，没有相关联的头文件，所以它只能用于执行已经用这个库链编过的应用。

能够用来组织和打包 C++代码的各种方式总结如下，表 7.1 中定义了描述代码容器的一些术语。

表 7.1 可复用组件

术 语	可视化属性	描 述
类	class Classname{ body } ;	函数、数据成员以及其生命周期管理描述(构造函数和析构函数)的集合
命名空间	namespace name{ body } ;	类、函数及其静态成员的声明的集合，可能会分布在多个文件中
头文件	.h	类定义、模板定义、函数声明(带有默认实参定义)、inline 函数定义、静态对象声明
源代码模块	.cpp	函数定义、静态对象定义
编译后“对象”模块	.o 或者.obj	每一个.cpp 模块都被编译进一个二进制模块中，作为链编库或者可执行文件的中间一步
库	.lib 或者.la(如果是动态库，则也可以是.so 或者.dll)	对象文件的索引集合被连接在一起。库中的任何代码模块不必存在 main() 函数
devel 包	.lib 加上头文件	库与相关联的头文件的结合
应用程序	Windows 下的扩展名为.exe，*nix 下没有特定的扩展名	目标文件与库相连接的一个集合，形成一个应用程序。只能包含一个称为 main() 的函数定义

7.1 建立并复用库

本书中的许多示例都连接了各种各样的库。可以下载 tarball 文件 src.tar.gz，它包含 dist 目录下使用的代码和库[①]。解包这个 tarball 文件并创建一个 shell/环境变量 CPPLIBS，使其包含到 src/libs 目录的绝对路径。

注意

当设置复用这些库的工程时，总是假定 shell/环境变量 CPPLIBS(或者 Windows 系统下的 %CPPLIBS%)已经被正确地设置成包含 libs 根目录。

这个变量有两个作用：它是库的全部 C++源代码的父目录，同时也是这些库的已编译过的共享目标代码的目标目录。

qmake 可以在工程文件里访问 CPPLIBS 环境变量，其访问语法是$$(CPPLIBS)。qmake 也可以包含其他的工程文件(片断)。例如，示例 7.1 中的工程文件包含文件 common.pri，它位于示例 1.6 中，用于常见的应用链编设置。

示例 7.1 src/xml/domwalker/domwalker.pro

```
# include common qmake settings
include (../../common.pri)

# this project depends on libdataobjects:
LIBS += -ldataobjects
```

① URL 可以在参考文献中找到。

```
# this directory contains the libraries:
LIBS += -L$$(CPPLIBS)

# Search here for headers:
INCLUDEPATH += . $$(CPPLIBS)/dataobjects

QT += xml gui

CONFIG += console
TEMPLATE = app

SOURCES += main.cpp slacker.cpp domwalker.cpp xmltreemodel.cpp
HEADERS += slacker.h domwalker.h xmltreemodel.h
```

此外，这个工程还为 LIBS 和 INCLUDEPATH 变量添加了一些值，以使它能够找到所依赖的库和头文件。

命令：

```
qmake -project
```

产生的工程文件所包含的信息，只会以当前工作目录下的内容为基础。特别地，qmake 无法知道用来构建工程所需要的外部库。如果工程依赖于某个外部库，则必须编辑这个工程文件，将它的值赋予变量 INCLUDEPATH 和 LIBS。然后，再次运行 qmake -project 会彻底破坏这些改变，所以不要这么做!

例如，假设应用需使用 dataobjects 库，其头文件位于$CPPLIBS/dataobjects 下，而共享目标文件的库位于$CPPLIBS 下，那么必须将如下这些行添加到工程文件中：

```
INCLUDEPATH += $$(CPPLIBS)/dataobjects  # the source header files
LIBS += -L$$(CPPLIBS)                   # add this to the lib search path
LIBS += -ldataobjects                   # link with libdataobjects.so
```

给变量 LIBS 赋值通常包含两种直接传递给编译器和链接器的开关。关于链接器开关的更多信息，请参见 C.3 节。

7.1.1　组织库：依赖性管理

如果某个程序元素复用了另一个程序元素，则它们之间就存在依赖性。也就是说，构建、使用或者测试某个程序元素(复用者)时，要求另一个程序元素(被复用者)存在并且是正确的。对于类，只要被复用者类的接口发生改变，就使得复用者类的实现必须改变，则它们之间就存在依赖性。

描述这种关系的另一种方法是：如果构建 ProgElement1 时需要 ProgElement2，则就称 ProgElement1 依赖于 ProgElement2。

如果编译 ProgElement2.cpp 时必须包含 ProgElement1.h，则称这种依赖性为编译时依赖性；如果目标文件 ProgElement2.o 包含 ProgElement1.o 中定义的符号，则称其为连接时依赖性。

图 7.1　依赖性

图 7.1 中通过 UML 框图给出了复用者 ClassA 和复用者 ClassB 之间的依赖性。

ClassA 与 ClassB 之间的依赖性能够以各种形式出现。在下面各种情形下，ClassB 接口的变化会导致 ClassA 的实现也必须改变。

- ClassA 的数据成员为 ClassB 的对象或者指针。
- ClassA 派生自 ClassB。
- ClassA 的函数具有 ClassB 类型的参数。
- ClassA 的函数使用 ClassB 的静态成员。
- ClassA 向 ClassB 发送消息(例如，信号)[①]。

在每一种情况下，都必须在 ClassA 的实现文件中包含指令#include ClassB。

图 7.2 给出的包框图中,列出了一部分 libs 库集合。其中存在直接依赖性和间接依赖性。这一节将关注库之间的依赖性(用虚线箭头表示)。

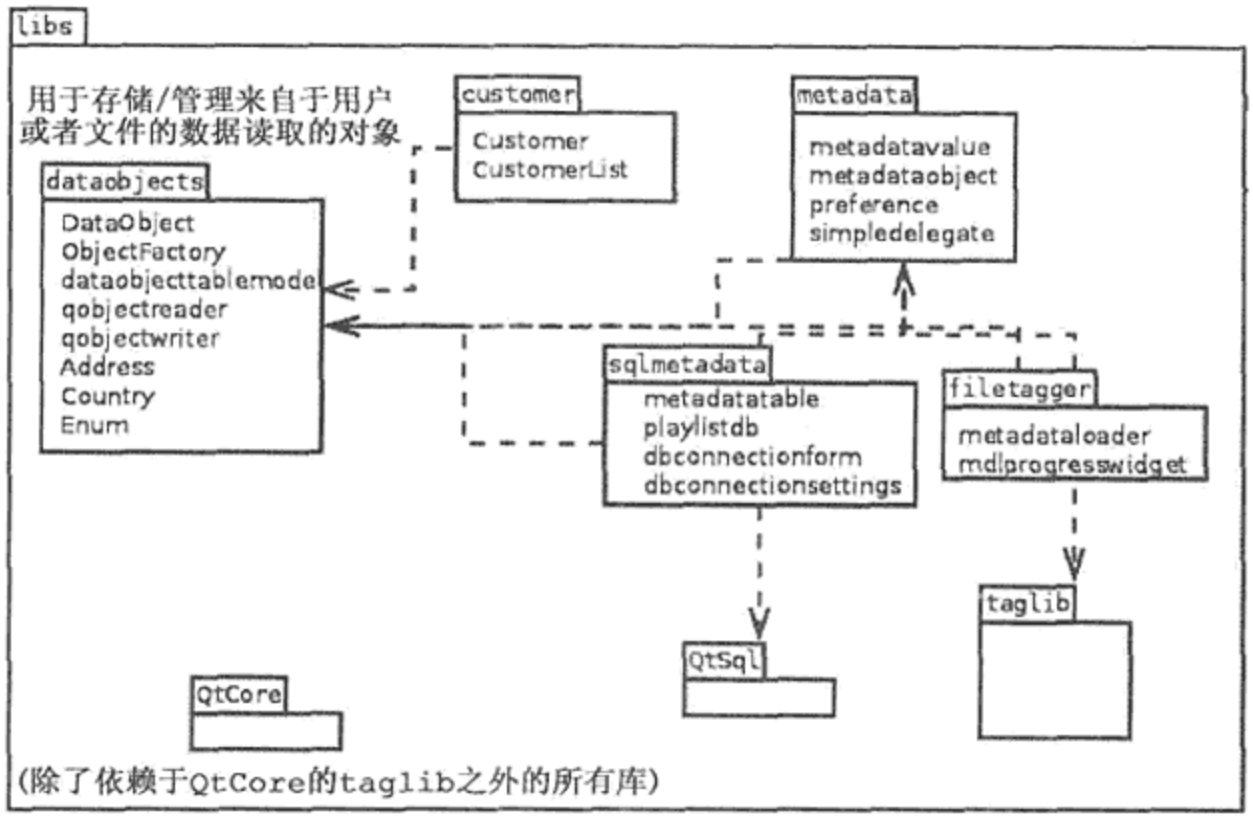


图 7.2 库及其依赖性

如果希望复用图 7.2 中的任何一个库，就需要确保它的所有依赖库也是工程的一部分。例如，如果使用 filetagger 库，则会存在一个依赖链，要求也使用 dataobjects 库(例如，派生自 DataObject 的那些 MetaData 类)和 taglib 库(例如，filetagger 使用 taglib 来加载元数据)。如果希望使用 sqlmetadata，则需要 QtSql，即 Qt 的 SQL 模块。

代码复用，这是一个有价值且重要的目标，但总会生成依赖。当设计类和库时，需要确保尽可能地减少不必要的或者无意的依赖性，因为它们会延长编译时间，降低类和库的可复用性。每一个#include 指令都会产生一种依赖性，所以要仔细检查，以保证它确实是必要的。这一点对头文件尤其重要：每一次用#include 指令包含头文件时，都会带入这个头文件中所包含的其他头文件，这样它们之间的依赖性就会相应地增多。

类的前置声明(forward declaration)将它的名称声明成一个有效的类名称，但不给出类的定义。这会使得类名称能够被用作指针和引用的类型，在遇到类的定义之前，不会将这些指针和引用进行解引用操作。对于类而言，前置声明使得头文件之间可以具有环形关系而没有环形依赖性(编译器不允许这样)。

① 8.5 节中将探讨信号和槽。

对于类定义头文件，需遵循的一条规则是：如果可以使用前置声明，就不要使用#include指令。例如，头文件 classa.h 的内容可能是这样的：

```
#include "classb.h"
#include "classd.h"
// other #include directives as needed
class ClassC;     // forward declaration
class ClassA : public ClassB {
  public:
    ClassC* f1(ClassD);
  // other stuff that does not involve ClassC
};
```

这个定义中，存在(至少)两个故意设置的复用依赖性：ClassB 和 ClassD，所以两个#include 指令都是必要的。但是，ClassC 有前置声明就足够了，因为类定义只使用了这个类的指针。

依赖性管理是一个重要的问题，它是许多文章的主题，且已经为它开发出了各种工具，其中的两个开源工具如下。

- cinclude2dot[①]，一种 Perl 脚本，它会分析 C/C++代码并产生一个依赖性图。
- Makedep[②]，一个用于大型软件工程的 C/C++依赖性生成程序，它会分析目录树下的全部源文件，并会构造出一个大型的依赖性文件用于包含在 Makefile 中。

7.1.2　安装库

当编写并测试完库之后，在链编过程结束后它将被安装到由 qmake 变量 DESTDIR 指定的目录下。例如，dataobjects 库的工程文件包含如下相关的行：

```
TEMPLATE = lib      # Build this as a library, not as an application
DESTDIR=$$(CPPLIBS) # Place the compiled shared object code here
```

对于库模板，qmake 将产生一个包含 install 目标的 Makefile。这样，在成功链编之后，命令

```
make install
```

将会把这个库复制到某个特定的位置。例如，在*nix 平台上，可以在 dataobjects 库的工程文件中添加如下的行：

```
target.path=/usr/lib
INSTALLS += target
```

然后，如果对目标位置有写权限，则命令

```
make install
```

会将 libdataobjects.so 文件以及与它们相关联的符号链接复制到/usr/lib 目录,使得这个库能被登录到计算机的任何人使用。

如果要迁移库，则过程会根据平台的不同而有所不同。在 Windows 系统中，可以将它的.dll 文件复制到 PATH 变量中给出的合适目录下。在*nix 系统中，可以将共享的目标文件以及相关的符号链接复制到/etc/ld.so.conf 中列出的目录下，也可以复制到 LD_LIBRARY_PATH 变量中列出的目录下。

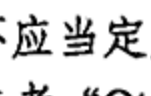
---

① 参见 http://www.flourish.org/cinclude2dot/。
② 参见 http://sourceforge.net/projects/makedep。

在开发过程中，通常只需在 CPPLIBS 目录下编译并安装库，并相应调整 LD_LIBRARY_PATH 设置即可。在*nix 系统中，如果遵循了上面的建议，并且一直使用了 CPPLIBS 环境变量，则只需在.bashrc 文件中添加下面的一行：

```
export LD_LIBRARY_PATH=$CPPLIBS
```

将这一行置于定义 CPPLIBS 环境变量的那一行的下面即可。在*nix 平台下部署库时，理想的安装目录是/usr/local，它是一个全部用户都能够访问的系统级位置。这个操作要求有超级用户权限。



注意

Qt 中并不要求 QTDIR 环境变量，但本书中有时在不同的地方会将其作为解压的 Qt tarball 的基本目录。它提供了一种方便的途径来引用 Qt 的示例、教程、二进制文件以及库。但是，在诸如 Ubuntu 或者 Debian 的系统中，Qt 被分成了多个文件夹并被安装在不同的位置，所以不应当定义 QTDIR。这种情况下，可以将 QTDIR 的使用理解成“Qt 二进制文件的父目录”或者“Qt 示例的父目录”（可以有两个不同的目录）。

提示

在 Microsoft 平台上建立动态链接库(DLL)要复杂得多，因为需要为每一个库定义唯一的“导出者”宏。

根据头文件是否从自己的库里包含还是从外部程序包含，这个宏会扩展成适当的 declspec()导出或者导入。

可以使用在<qglobal.h>中定义的预处理器宏 Q_DECL_EXPORT，它能够有条件地开启或者关闭 declspec，并可以在合适的时刻导入或者导出这个标志符。

例如，libdataobjects 中定义了一个名称为 DOBJS_EXPORT 的导出宏，见示例 7.2。

示例 7.2 src/libs/dataobjects/dobjs_export.h

```
#include <QtGlobal>
#ifndef DOBJS_EXPORT
/* This macro is only for building DLLs on Windows. */
#ifndef Q_OS_WIN
#define DOBJS_EXPORT
#elif defined(DATAOBJECTS_DLL)
#define DOBJS_EXPORT Q_DECL_EXPORT
#else
#define DOBJS_EXPORT Q_DECL_IMPORT
#endif
#endif
```

只有当 dataobjects.pro 中定义了 DATAOBJECTS_DLL 时，这个宏才会导出符号：

```
win32 {
   CONFIG(dll) {
      DEFINES += DATAOBJECTS_DLL
   }
}
```

现在，对于希望导出到 DLL 的任何类，只需将这个宏置于 class 关键字与类定义中的类名称之间即可：

```
class DOBJS_EXPORT DataObject : public QObject {
  [ . . . ]
}
```

当构建第一个 DLL 时，qtcentre.org 提供了关于这个主题的一些思路[①]，它能帮助节省许多时间。

注意

QLibrary 类中的几个方法可用来以独立于平台的方式在运行时加载或者卸载动态库。

7.2　练习：安装库

本书中的大量示例使用了库中的一些类，这些库是专门为这本书而编写的。可以下载这些类的源代码[②]，其中包含用 Doxygen 生成的 API 文档。这个练习中将看到如何建立并安装一些库。

随后，将给出在*nix 平台上安装库供本书示例使用的指导。在 Windows 系统上用 MinGW 安装 Qt 和 MySQL 的帮助，请查看 QtCentre Wiki[③]。

正如 7.1 节和 7.1.2 节中建议的那样，需依次执行如下步骤：

- 创建一个专门用于 C++/Qt 的目录，例如~/oop/projects/。
- 从 dist 目录下载 src.tar.gz[④]。
- 将这个 tarball 文件解压到一个新目录。结果应当是一个 libs 目录和许多的子目录，其中包括 libs/dataobjects 和 libs/customer。
- 在 libs 目录下检查名称为 libs.pro 的 subdirs 工程文件。
- 这个工程文件是用来建立库并进行测试的。
- 可随意将那些不准备使用的库注释掉，但是不要改变它们的顺序。
- 创建一个名称为 CPPLIBS 的 shell/环境变量，使之包含新的 libs 目录的绝对路径。为了方便，可以将这个环境变量的定义放入 shell 脚本中，如示例 7.3 所示。

注意

如果决定要将 libs.pro 中的某个特定目录注释掉，也应将 libs/tests/tests.pro 中相应的测试目录注释掉(否则不会建立库的测试部分)。

需提醒的是：在工程文件中某一行的开头插入一个#字符，就能将它注释掉。

① 参见 http://www.qtcentre.org/forum/showthread.php? t=1080。

② 来自于 dist 目录。

③ 参见 http://wiki.qtcentre.org/index.php? title=Building_the_QMYSQL_plugin_on_Windows_using_MinGW。

④ URL 可以在参考文献中找到。

libs.pro

```
TEMPLATE = subdirs
CPPLIBS=$$(CPPLIBS)
isEmpty(CPPLIBS) {
   error("Define CPPLIBS environment variable to point to this location.")
   }
SUBDIRS += dataobjects \
           actioneditor \
           customer \
#          metadata \
#          sqlmetadata
```

从 libs 目录建立库需要两个步骤：

1. qmake -recursive //在$CPPLIBS 和每一个子目录下创建 Makefile。
2. make //建立库并测试(每一个 Hc)[①]。

验证这些库已经建立，且共享的目标文件(例如，libdataobjects.so[②])位于 CPPLIBS 目录下。下面列出的是一个典型的 Linux 系统下 libs 目录下的内容：

```
libs> ls -l
drwxr-xr-x  5 dataobjects
lrwxrwxrwx  1 libactioneditor.so -> libactioneditor.so.1.0.0
lrwxrwxrwx  1 libactioneditor.so.1 -> libactioneditor.so.1.0.0
lrwxrwxrwx  1 libactioneditor.so.1.0 -> libactioneditor.so.1.0.0
-rwxrwxr-x  1 libactioneditor.so.1.0.0
[...]
lrwxrwxrwx  1 libdataobjects.so -> libdataobjects.so.1.0.0
lrwxrwxrwx  1 libdataobjects.so.1 -> libdataobjects.so.1.0.0
lrwxrwxrwx  1 libdataobjects.so.1.0 -> libdataobjects.so.1.0.0
-rwxr-xr-x  1 libdataobjects.so.1.0.0
-rw-r--r--  1 libs.pro
-rw-r--r--  1 Makefile
libs>
```

以 drwxr-xr-x 开头的行是目录，以 lrwxrwxrwx 开头的行是符号链接。可以通过搜索引擎找出为什么每一个共享目标文件都有三个符号链接的答案。

固定链接器路径

- 用合适的语法，更新 shell/环境变量 LD_LIBRARY_PATH(*nix)或者 PATH(win32)，以包含 CPPLIBS。
- 创建一个 projects/tests 目录，它就是保存用于测试各种库组件代码的地方。
- 运行随 libs tarball 文件一起提供的测试程序，它们位于与所建立库对应的 libs/tests 子目录下。

---

① 在建立某些库之前，可能还需要安装一下专门的 Qt 包。例如，phononmetadata 库就要求 libphonon-dev 包。链接器会提示这些依赖性。

② 在 Windows 系统下为 libdataobjects.lib 和 dataobjects.dll，后者必须位于 PATH 中列出的目录下，所以一定要在 PATH 中包含%CPPLIBS%。

在*nix 平台上，通常用一个 shell 脚本来定义环境变量。示例 7.3 中给出了处理这项工作的一个 bash shell 脚本。

示例 7.3　src/bash/env-script.sh

```
export CPPLIBS=$HOME/oop/projects/libs
export LD_LIBRARY_PATH=$CPPLIBS
```

应注意这个脚本中的 bash 语法细节：

- 环境变量 HOME 包含到个人主目录的绝对路径。也可以使用符号“~”。
- 位于等于号左边的环境变量没有美元符号前缀。
- 位于等于号右边的环境变量必须有美元符号前缀。
- 如果环境变量是环境的一部分，而不仅仅是针对这个脚本文件的，则需要使用 export 命令。

输入如下两种命令之一，都可以运行这个脚本：

```
source env-script.sh
```

或者

```
. env-script.sh
```

注意后一个命令开始处的点号。在 bash shell 中，点号与 source 命令等价。

如果希望在启动每一个 shell 时能够被自动设置，则可以将这个脚本放入~/.bashrc，只要启动 bash(例如，启用终端或者控制台)，它就会自动运行。

关于如何编写 shell 脚本，Hamish Whittal 提供了一个很好的在线指南[①]。

7.3　框架与组件

如何对类进行组织，已经超出了简单的继承的范围。对这种框架进行仔细设计，可以使查找和复用组件变得更加容易。所有大型软件工程都建立在框架的基础之上，这里将探讨一些当今流行的框架。

在现代编程技术中，代码复用具有最高优先级。过去，计算机时间较为昂贵而程序员时间相对便宜，现在，形势逆转了。如今，所有的软件都是基于块构建的，块本身也可以看成是一个软件。几乎没有从头开始进行应用设计的，因为这样做是将程序员时间浪费在重复设计和实现那些已经由公认的专家设计、实现、优化和测试过的模块。

框架是一个(通常非常大的)通用(或者针对特定领域的)类与约定的集合，其目的是提高设计的一致性。框架通常被用来创建图形化应用、数据库应用或者其他复杂的软件。

框架一般都具有文档丰富的公共 API。API 是库中公共函数、类和接口的描述。为了实现框架，可以采用设计模式。用设计模式进行开发涉及寻找合适的对象和可能的层次。所使用的类和模式都采用富于描述性的名称，这样可以真正做到一次定义、处处复用。7.3 节中将简要讨论设计模式。

① 参见 http://www.cc.puv.fi/~jd/course/Linux/Shell_Scripting/index.html。

Qt 是许多开源的面向对象框架中的一种，它提供一组可复用的组件，用于创建跨平台的应用。其他一些知名的类似工具如下。

- boost[①]，一种开源的跨平台 C++工具类库。
- mono[②]，Microsoft.NET 的一种开源实现，它是用于 C#的 API，以 libgtk 为基础而创建。
- libgtk, libgtk++，定义了一些窗件的库，这些窗件用于 Gnome 桌面、Mozilla、Dia、GAIM、GIMP、Evolution、OpenOffice 以及许多其他的开源程序。
- wxWidgets[③]，另一种 C++跨平台的窗件工具集。
- Wt[④]，一种类似于 Qt 的框架，用于用 boost 和 AJAX[⑤]创建 Web 应用。

利用 Qt 这样的多平台框架，就可以从其他人的创造性工作中获得大量的好处。严格使用 Qt 创建的软件，其基础是将那些已经在 Windows, Linux 和 Mac OS/X 上经过大量程序员测试过的组件。

Qt(以及跨平台的 Gnu ToolKit，Gtk++)这样的工具集，在不同平台下有不同的实现。这就是为什么基于 Qt 的应用在 Linux 上看起来与 KDE 应用类似、在 Windows 上看起来与 Windows 应用类似的原因。

7.4 设计模式

对于面向对象的软件设计所面对的共同问题而言，设计模式是一种高效且精炼的解决方案。设计模式是一种高度抽象的模板，可以将它们应用到不同类型的设计问题。

在 Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides(他们经常被戏称为“四人组”)所著的图书 *Design Patterns*[Gamma95]中，分析了 23 种特定的模式。每一种模式都用专门的一个小节进行讲解，描述了下面这些事情：

- 模式名称。
- 对可以使用该模式的某种问题的描述。
- 对关于设计问题以及如何获得解决方案的抽象描述。
- 应用该模式后能够得到什么结果以及关于得失的探讨。

设计模式可用于许多不同场合，其中的大多数都描述如何根据职责来区分代码。这些模式被分成三种类别：创建模式(Creational)、结构模式(Structural)和行为模式(Behavioral)。结构模式描述如何组织对象并连接它们；行为模式描述如何组织代码；创建模式描述如何组织那些用于管理对象创建的代码。

“四人组”宣称：设计模式是“在特定环境下用于解决某类设计问题的类与对象间通信关系的描述”。当在后面继续用 Qt 开发应用时，将会看到几个设计模式的描述和示例。

当使用更多流行的编程语言和 API 时，就会遇到工作中常用的一些设计模式。另一方面，类似于 Python 之类的现代的、动态的编程语言，已经具有对几种设计模式的内置支持，所以

---

① 参见 http://www.boost.org。
② 参见 http://www.mono-project.com/。
③ 参见 http://www.wxwidgets.org。
④ 参见 http://www.webtoolkit.eu/wt。
⑤ AJAX 是 Asynchronous JavaScript And XML 的首字母缩写，一种 JavaScript 和 XML 的客户系统，在 Web 页面中提供列表/树/表格视图且具有 GUI 行为。

在软件中实现它们是学习这种语言的一种天然收益。Qt 中也是如此，后面将会遇到几种 Qt 类，它们都实现了一些流行的设计模式。

7.4.1　序列化器模式：QTextStream 和 QDataStream

序列化器模式

序列化器(serializer)是一种只负责读取或者写入对象的对象。Qt 的 QTextStream 序列化器用于读写人可读的文件，而 QDataStream 序列化器用于读写结构化的二进制数据。这些类是用序列化器模式实现的，被用于 C++和 Qt 中[Martin98]。

利用 QDataStream，就可以序列化和解序列化(deserialize) QVariant 支持的全部类型，包括 QList, QMap, QVector 以及其他类型[①]。利用 QTextStream，如果希望在定制的类型上将提取运算符(>>)用于插入运算符(<<)的输出，则必须为字符串类型定义正确的字段和记录分隔符，并要用样本数据测试这些运算符的正确性。

由于这两种流都可以从 QIODevice 创建，且存在许多使用 QIODevice 进行通信的其他 Qt 类，所以这两种运算符能够将对象发送到网络、管道或者数据库。

示例 7.4 给出了 MetaDataValue 的输人/输出运算符函数的友元声明，MetaDataValue 是用来存储歌曲元数据的一个类。这些运算符不是成员函数，因为它的左操作数是一个类无法修改的 QDataStream 或者 QTextStream。此外，运算符还不能是 MetaDataValue 的成员函数，因为序列化器模式的思想是将 I/O 代码与类本身分开。METADATAEXPORT 宏有利于在 Windows 平台上复用这些代码[②]。

示例 7.4　src/libs/metadata/metadatavalue.h

```
[ . . . . ]
class METADATAEXPORT MetaDataValue {
public:

    friend METADATAEXPORT QTextStream& operator<< (QTextStream& os,
                                                   const MetaDataValue& mdv);
    friend METADATAEXPORT QTextStream& operator>> (QTextStream& is,
                                                   MetaDataValue& mdv);
    friend METADATAEXPORT QDataStream& operator<< (QDataStream& os,
                                                   const MetaDataValue& mdv);
    friend METADATAEXPORT QDataStream& operator>> (QDataStream& is,
                                                   MetaDataValue& mdv);
    friend METADATAEXPORT bool operator==(const MetaDataValue&,
                                          const MetaDataValue&);
[ . . . . ]
    virtual QString fileName() const ;
    virtual Preference preference() const ;
    virtual QString genre() const;
    virtual QString artist() const;
    virtual QString albumTitle() const;
    virtual QString trackTitle() const;
    virtual QString trackNumber() const;
```

① 参见 http://doc .qt.nokia.com/latest/datastreamformat.html。

② 7.1.2 节的“提示”小节中探讨过这类宏。

```
    virtual const QImage &image() const;
    virtual QTime trackTime() const;
    virtual QString trackTimeString() const;
    virtual QString comment() const;
[ . . . . ]
protected:
    bool m_isNull;
    QUrl m_Url;
    QString m_TrackNumber;
    QString m_TrackTitle;
    QString m_Comment;
    Preference m_Preference;
    QString m_Genre;
    QString m_Artist;
    QTime m_TrackTime;
    QString m_AlbumTitle;
    QImage m_Image;
};
Q_DECLARE_METATYPE(MetaDataValue);                                    1
[ . . . . ]
```

1 添加到 QVariant 类型系统。

每一个运算符都处理一个 MetaDataValue 对象。operator<<()将它的数据插入到输出流中，operator>>()从输入流中提取它的数据。每一个运算符都返回一个左操作数的引用，以便能够进行链式操作。

示例 7.5 中的 QTextStream 运算符需要考虑到空白符和分隔符，因为进行流式输出时全部内容都是以字符串表示的。

示例 7.5 src/libs/metadata/metadatavalue.cpp

```
[ . . . . ]


QTextStream& operator<< (QTextStream& os, const MetaDataValue& mdv) {
    QStringList sl;
    sl << mdv.url().toString() << mdv.trackTitle() << mdv.artist() << mdv.
albumTitle()
            << mdv.trackNumber() << mdv.trackTime().toString("m:ss")
            << mdv.genre() << mdv.preference().toString() << mdv.comment();
    os << sl.join("\t") << "\n";                                      1
    return os;
}

QTextStream& operator>> (QTextStream& is, MetaDataValue& mdv) {
    QString line = is.readLine();
    QStringList fields = line.split("\t");                            2
    while (fields.size() < 9) {
        fields << "";
    }
    mdv.m_isNull = false;
    mdv.setUrl(QUrl::fromUserInput(fields[0]));
    mdv.setTrackTitle(fields[1]);
```

```
    mdv.setArtist(fields[2]);
    mdv.setAlbumTitle(fields[3]);
    mdv.setTrackNumber(fields[4]);
    QTime t = QTime::fromString(fields[5], "m:ss");
    mdv.setTrackTime(t);
    mdv.setGenre(fields[6]);
    Preference p(fields[7]);
    mdv.setPreference(p);
    mdv.setComment(fields[8]);
    return is;
}
```

1 输出到 TSV(制表符分隔的值)。

2 以 TSV 读取。

示例 7.6 中的 QDataStream 运算符使用起来要简单得多，因为它们不需要程序员来区分数据项。

示例 7.6　src/libs/metadata/metadatavalue.cpp

```
[ . . . . ]

QDataStream& operator<< (QDataStream& os, const MetaDataValue& mdv) {
    os << mdv.m_Url << mdv.trackTitle() << mdv.artist() << mdv.albumTitle()
            << mdv.trackNumber() << mdv.trackTime() << mdv.genre()
            << mdv.preference() << mdv.comment() << mdv.image();
    return os;
}

QDataStream& operator>> (QDataStream& is, MetaDataValue& mdv) {
    is >> mdv.m_Url >> mdv.m_TrackTitle >> mdv.m_Artist >> mdv.m_AlbumTitle
            >> mdv.m_TrackNumber >> mdv.m_TrackTime >> mdv.m_Genre
            >> mdv.m_Preference >> mdv.m_Comment >> mdv.m_Image;
    mdv.m_isNull= false;
    return is;
}
```

示例 7.7 中给出了如何将这些运算符用于不同的流。使用 QDataStream 时唯一的缺点是结果文件为二进制的(即人不可读的)。

示例 7.7　src/serializer/testoperators/tst_testoperators.cpp

```
[ . . . . ]
void TestOperators::testCase1()
{
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
    }
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
```

```
}
QList<MetaDataValue> values;
while (!textStream.atEnd()) {
    MetaDataValue mdv;
    textStream >> mdv;                                    1
    values << mdv;                                        2
    dataStream << mdv;                                    3
}
textFile.close();
binaryFile.close();
textFile.setFileName("playlist2.tsv");
if (binaryFile.open(QIODevice::ReadOnly)) {
    dataStream.setDevice(&binaryFile);
    for (int i=0; i<values.size(); ++i) {
        MetaDataValue mdv;
        dataStream >> mdv;                                4
        QCOMPARE(mdv, values[i]);                         5
    }
}
}
[ . . . . ]
```

1 以 TSV 读取。

2 添加到列表。

3 写入 `binaryFile`。

4 读取二进制数据。

5 它与以前读取的相同吗?

7.4.2 反模式

反模式(antiPattern)最早是由[Koenig95]提出的一个术语，它常用来描述那些已经被证明是没有效果的、低效率的或者达不到预期目标的编程实践。对于挑选出来的一些反复出现的问题，当其解决方案在学生以及无经验的程序员之间传递时，几种反模式就出现了。在 Wikipedia[①]中有关于这一主题的一篇信息丰富、讲解透彻的文章，它罗列并简要描述了大量的反模式。研究一些反模式的例子，可以帮助程序员避免类似的陷阱。以下是来自于 Wikipedia 文章 antiPatterns 的一小部分内容。

- 软件设计反模式
  - 输入杂乱：没有指定和实现如何对可能的无效输入进行处理。
  - 接口膨胀：接口的功能强大而复杂，以至于难以复用或者实现。
  - 竞争风险：没有注意事件的不同顺序所造成的后果。
- 面向对象设计反模式
  - 循环依赖性：在对象或者软件模块之间引入了不必要的直接或者间接彼此依赖性。
  - “上帝”对象：具有太多信息或者太多责任的对象。这可能是由于在一个类中包含太多函数而导致的。这种对象可以在许多种情况下出现，但经常发生的情形是：将模型和视图的代码组合在同一个类中。

① 参见 http://en.wikipedia.org/wiki/AntiPattern#Programming_antiPatterns。

- 编程反模式
  - 难以编码：在实现中嵌入了关于系统环境的假设。
  - 魔幻数字：算法中包含未解释的数字。
  - 魔幻字符串：代码中包含直接的字符串作为事件类型用于比较。
- 方法学反模式
  - 复制—粘贴编程：复制并修改已有的代码，而不是创建更通用的解决方案。
  - 一切从头开始：不采用已有的解决方案，而是采用(执行起来表现要差得多的)定制的解决方案。

图 7.3 中，Customer 包含的成员函数用于以 XML 格式导入和导出它的每一个数据成员。

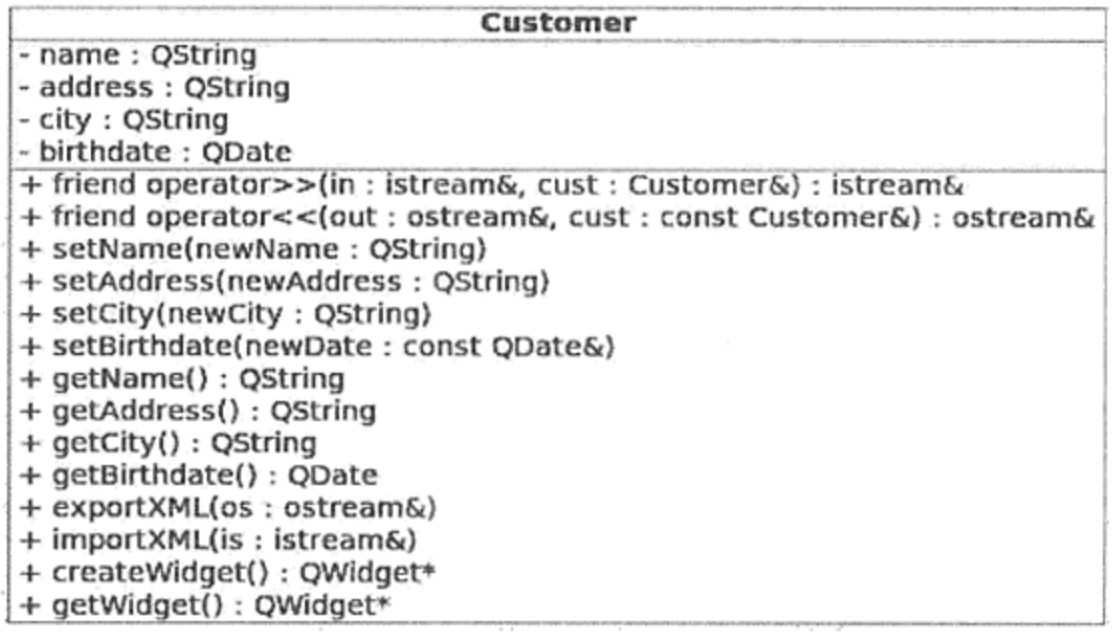


图 7.3　反模式示例

getWidget()提供了一个特殊的 GUI 窗件，用户可以用它来从图形化应用输入数据。此外，还存在几个通过 iostream 进行输入/输出的友元函数[①]。

这个类是一个模型，因为它拥有数据并表示一些抽象实体。但是，这个类也包含了一些视图代码，因为存在 createWidget()和 getWidget()成员函数。此外，它还包含专门针对特定 I/O 流的序列化代码[②]。如此看来，这个类对数据模型承担了太多的责任。这是接口膨胀(以及其他)反模式的一个例子。

当实现其他的数据模型类，比如 Address，ShoppingCart，Catalog，CatalogItem 等，接口膨胀反模式导致的问题就会立即出现。这些类还需要如下这些方法：

- createWidget()
- importXML()
- exporXML()
- operator<<()
- operator>>()

这会导致另一种反模式“复制—粘贴编程”的使用。一旦改变了数据结构，就需要相应改变全部的数据表示以及 I/O 方法。当维护这样的代码时，就有可能出现 bug。如果 Customer 是反射的(reflective)，则意味着它对自己的成员具有判断能力(例如，有多少个属性？属性的

① 2.6 节中探讨了友元。

② 序列化是一种将对象的数据转换成某一种形式的过程，这种形式使得数据能够保存到文件中或者通过网络传输，以便今后可以重构这个对象。7.3.1 节中探讨了序列化器模式。

名称是什么？属性的类型是什么？如何加载/存储这些属性？子对象是什么？)，然后，就可以定义一种通用的方法来读写用于 Customer 的对象以及任何其他相似的反射类。第 12 章中探讨了一个例子，它给出了如何用反射编写更为通用的代码。

7.4.2.1 关于设计模式的更多探讨

有数不清的 Web 站点探讨并给出了关于设计模式的例子。除了 Wikipedia 之外，还可以访问如下的站点：

- Vince Huston 的 Web 页面[①]。
- Douglas Schmidt 的设计模式教程[②]。
- Wiki 图书 C++ Programming/Code/Design Patterns[③]的站点。

术语“反模式”已经被 Pattern Community(模式社区)[④]采纳，这个社区维护着一个反模式的目录[⑤]。

7.5 复习题

1. 什么是平台？你使用的是什么平台？在你的学校或者工作地有哪些平台？
2. 什么是代码复用？如何实现它？其优点是什么？
3. 编译器的角色是什么？
4. 链接器扮演的角色是什么？
5. 给出三种 C++库的名称。
6. 什么是 CPPLIBS？为什么需要它？
7. 对下面的每一个项，判断它是否会正常出现在某个头文件(.h)或者某个实现文件(.cpp)中，并给出理由。
   a. 函数定义
   b. 函数声明
   c. static 对象声明
   d. static 对象定义
   e. 类定义
   f. 类声明
   g. inline 函数定义
   h. inline 函数声明
   i. 默认实参指示符
8. 编译时依赖性与链接时依赖性有什么不同？
9. 什么是框架？你在使用它吗？
10. 什么是设计模式？最常见的设计模式有哪些？
11. 什么是反模式？

---

① 参见 http://www.vincehuston.org/dp/。
② 参见 http://www.cs.wustl.edu/~schmidt/tutorials-patterns.html。
③ 参见 http://en.wikibooks.org/wiki/C++_Programming/Code/Design_Patterns。
④ 参见 http://c2.com/cgi/wiki? PatternCommunity。
⑤ 参见 http://c2.com/cgi/wiki? AntiPatternsCatalog。

第8章 QObject，QApplication，信号和槽

QObject 是 Qt 库中许多重要的类的基类，如 QEvent，QApplication，QLayout 和 QWidget。我们会把任何 QObject 派生类的对象都看作是一个 QObject 对象。一个 QObject 可以有一个父对象和一些子对象，这是组合模式(Composite pattern)的另一种实现方式。它可以使用信号和槽，即观察者模式(Observer pattern)的一种实现，与其他 QObject 通信。QObject 使基于事件的编程成为可能，其中用到了 QApplication 和 Qt 的事件循环。

下面先简单看一下 QObject 的定义。

```
class QObject {
  public:
    explicit QObject(QObject* parent=0);
    QObject * parent () const;
    QString objectName() const;
    void setParent ( QObject * parent );
    const ObjectList & children () const;
    // ... more ...
};
```

QObject 没有公有的复制构造函数或复制赋值运算符。向下到 QObject 类定义的结尾处有一个宏 Q_DISABLE_COPY(QObject)，它显式地确保任何 QObject 都不能被复制。QObject 也不是设计用于复制的。通常来说，QObject 会用来代表具有唯一身份的对象，也就是说，它们对应于现实世界中具有同样某种永久身份的东西。这种不带复制构造函数策略的一个直接后果就是永远无法通过值传递方式向函数传递 QObject。尽管把一个 QObject 对象的数据成员复制给另外一个 QObject 对象是可能的，但这两个 QObject 对象仍被认为是不同的。

显式构造函数

QObject(及其派生类)的单参数构造函数应当予以显式(explicit)声明，以避免意外的隐式转换的发生[①]。QObject 不被用来持有一个临时值，而且也不应该有从一个指针或单值生成另一个对象的可能。

每个 QObject 都可以有(至多)一个父 QObject，且可以拥有任意数量的子 QObject。换句话说，每个子对象的类型必须是 QObject 或者必须派生自 QObject。每个 QObject 都将指向各个子对象的指针存放在一个 QObjectList 中[②]。这一列表自身是通过一种松散的风格创建的，以尽量降低那些不带子孙的对象的开销。因为每个子对象都是一个 QObject，也可以有任意数量的子对象，所以很容易明白为什么不允许在 QObject 对象之间进行复制。

① 在 2.12 节曾第一次讨论到关键字 explicit。

② QObjectList 是 QList<QObject *>的别名，只是通过 typedef 进行了重定义而已。

子对象的概念有助于我们理解身份的概念和 QObject 之间不允许复制的策略。如果把一个一个的人看成是 QObject，则每个 QObject 的身份都有所不同这一点就很容易理解了。同样，每个 QObject 都可以有子对象的概念也非常清楚了。而对于每个 QObject 都至多有一个父对象的规则也就可以看成是一种简化类的实现的方式。最后，也就可以看出 QObject 不允许复制策略的必要性了。即使有可能去“克隆”一个人(也就是，将一个 QObject 对象的数据成员复制到另一个 QObject 对象中)，也无法去处理这个人所拥有的子孙问题，因此可以得到一个非常清楚的答案：克隆对象是一个拥有不同身份、完全分离的、不同的对象。

每个 QObject 父对象都会管理自己的子对象。这就意味着，在调用 QObject 的析构函数时会自动销毁该对象的子对象。

子对象列表会在各个 QObject 对象之间建立一种双向的关联：

- 每个父对象都知道它的子对象的地址。
- 每个子对象都知道其父对象的地址。

给某个对象设置父对象，将会隐含地把此对象添加到父对象的子对象列表之中，例如：

```
objA->setParent(objB);
```

会把 objA 的指针添加到 objB 的子列表中。如果随后再运行语句

```
objA->setParent(objC);
```

那么 objA 的指针就会从 objB 的子对象列表中移除，然后添加到 objC 的子对象列表中。这一行为称作重父化(reparenting)。

父对象与基类的比较

不应将父对象和基类混为一谈。父—子关系是为了描述对象运行时的约束和管理关系。基类派生关系是编译时各个类进行判定的一种静态关系。

当然，父对象也可能是其某些子对象的基类的实例。但这两种关系仍是不同的，务必不要混淆，尤其是考虑到许多类会直接或间接地派生自 QObject 时更是如此。图 8.1 应该可以澄清这一思想。

正如之前看到的那样，图形用户界面(GUI)中的所有窗件都派生自 QWidget，而 QWidget 则从 QObject 派生而来。像对待 QObject 一样，我们会把任意的 QWidget 派生类的对象都看作是一个 QWidget 对象(或者在某些时候，仅仅是看作一个窗件)。在 GUI 中，父—子关系通常是可见的：子窗件会显示在父窗件中。图 8.1 中，对话框窗件有数个子对象，包括：一个标签窗件，一个行编辑窗件以及两个按钮窗件。它还有一个标题栏窗件，既可以是该对话框的父对象也可以是其兄弟对象。标题栏窗件通常是由窗口管理器提供且含有数个子窗件，包括几个可以让用户能够最小化、最大化或者关闭该对话框的按钮窗件。

从图 8.1 中也可以看出对子对象管理需求的必要性。关闭对话框时(例如，通过单击右上角处的 X 按钮)，希望整个对话框窗口可以从屏幕上消失掉(也就是说，要销毁掉)。并不希望将对话框窗口打散开(例如，剩下按钮或标签窗件)而仍旧停留在屏幕上，也并不希望把这项清理工作的重担压在编程人员的身上。这就是为什么要在调用 QObject 的析构函数时要由它负责销毁自己所有的子对象的原因。这是一个自然递归的过程。当销毁 QObject 时，它首先会调用自己每个子对象的析构函数，每个子对象必须再调用自己每个子对象的析构函数，等等。

图 8.1　父-子对象和基于派生的类

现在可以来考虑一下，假设要复制 QObject，将会引起一些什么样的问题。例如，副本对象会和原对象拥有同一个父亲吗？副本对象应该(在某种意义上)拥有原对象的子对象吗？子对象列表的浅复制(shallow copy)将无法工作，因为如果那样可行的话，则每个子对象都将会拥有两个父对象。在那种情况下，如果需要销毁副本对象(例如，如果该副本是一个函数调用中的值参数)，那么每一个子对象也需要销毁，这样即使采用资源共享的方法也会遇到一些严重的问题。而在子对象列表非常庞大且指向的对象很大的情况下，子对象列表的深复制(deep copy)操作会付出高昂的代价。因为每个子对象同样也可以拥有任意数量的子对象，这种本就问题多多的方法自然会遇到更为严重的问题。

注意

通常而言，没有父对象的 QObject 应当在程序栈区(stack)中进行定义，而那些有父对象的 QObject 则应当在堆区(heap)动态创建出来。这样可有助于确保发生正确的析构操作：位于栈区的对象是没有父对象的，在它超出作用域时就会被销毁掉(例如，当它在函数中定义而函数结束返回时，或者在控制流离开它所定义的地方时)。然后，在被析构之前，该(栈)对象会销毁它(堆区)的子对象。

8.1　值和对象

C++类型可以分成两类：值(value)类型和对象(object)类型。

值类型的实例通常相对简单，占用相邻的内存空间，而且可以进行快速复制或者比较。值类型的例子有：*Anything**，int，char，QString，QDate 和 QVariant。

QVariant

QVariant 是一种特殊的联合体类型，可保存所有可复制的内置类型和编程人员定义的类型。有关 QVariant 以及 QVariant 已支持的诸如 QList, QImage, QString, QMap,

QHash 等各类型的一件趣事是：它们都支持隐式共享、写时复制(copy on write)和引用计数(reference counting)。这就意味着，通过值对它们复制、传递和返回的代价要相对低些。可参阅 QMetaType::Type 来查看所支持类型的列表，11.5 节中也讲解了实现隐式共享的更多细节。

带有公有默认构造函数、复制构造函数和复制赋值运算符的任何类都是值类型的。

相反，对象类型的实例通常要复杂得多，它要维护一些类型的身份。对象类型通常很少进行复制(克隆)。如果允许克隆，为这一操作所付出的代价也会很大，而且生成的新对象(图)与原来对象的身份也不会相同。

QObject 的设计人员毫不犹豫地采用了“无复制”策略，该方法就是将赋值运算符与复制构造函数设置为 private 类型。这种设计方法有效地阻止了编译器给 QObject 派生类生成赋值运算符和复制构造函数。采用这一机制后，如果试图通过值传递的方式向函数传递或者从函数返回 QObject 派生类对象，都会导致编译时错误。

提示

从来没有令人信服的要在堆区创建 QList，QString，QHash，QImage 或者其他与 QVariant 相关类型的理由，也不要那样做。还是让 Qt 为你完成引用计数和内存管理吧。

8.2 组合模式：父对象和子对象

根据[Gamma 95]，组合模式(composite pattern)的意图是通过将部分—整体的层次结构表示成树状结构，以便于使用较为简单(组件)的部分来创建出复杂(复合)的对象。组合模式的使用场景是，客户(client)无须区分简单的组成部分与由简单部分所组成(例如，包含)的较为复杂的部分。

图 8.2 描述的是组合模式。在这个示意图中，有两个不同的类可以方便地描述上述两种角色：

- 复合对象是包含可以包含子对象的类。
- 组件对象是可以拥有一个父对象的类。

许多 Qt 类都用到了组合模式：QObject，QWidget，QTreeWidgetItem，QDomNode，QHelpContentItem 和 QResource。在任何基于树的结构体中都可以找到组合模式。

图 8.3 中，可以看出 QObject 既是复合对象也是组件对象。可以把 QObject 对象之间的整体—部分关系表达成父-子关系。在这样的一棵树中，最高层的(即最为“复合”的)QObject 对象(即树的根)将会有若干个子对象而没有父对象。最简单的 QObject(即这棵树的叶子节点)都有一个父对象而无子对象。客户代码可以递归地处理此树中的每一个节点。

下面给出一个可能会用到组合模式的例子。萨福克大学由 Gleason Archer 在 1906 年创建，当时他决定开始向一些想要成为律师的零售商讲授法律的基本原则。最开始时，他只有一位秘书，后来又雇佣了一些讲师。这个学校的组织图非常简单：一间办公室、几个任务分工不同的雇员。随着企业不断发展壮大，组织图开始变得越来越复杂，比如逐渐开始增加了新的办公室和公寓。而在 100 多年后的今天，当时的法律学校得到了发展，已拥有一个艺术与科学学院、一个商学院、一个艺术与设计学院、海外学校以及许多专用的办公室，因而使组织

图变得异常复杂起来，同时也注定以后还会越来越复杂。图 8.4 所示为目前萨福克大学的一个简化过的、简明的组织图。

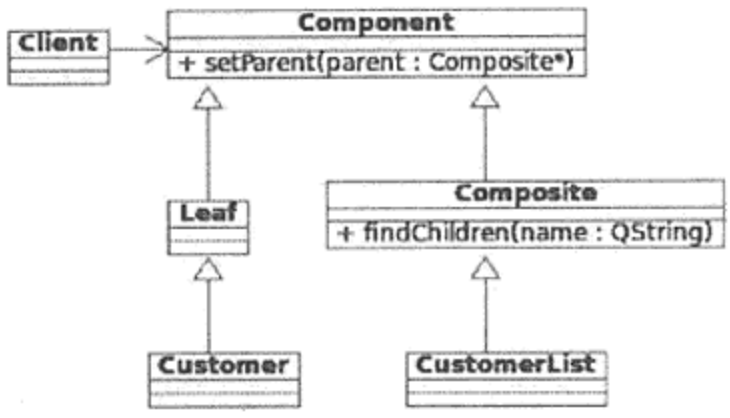


图 8.2　组件和复合

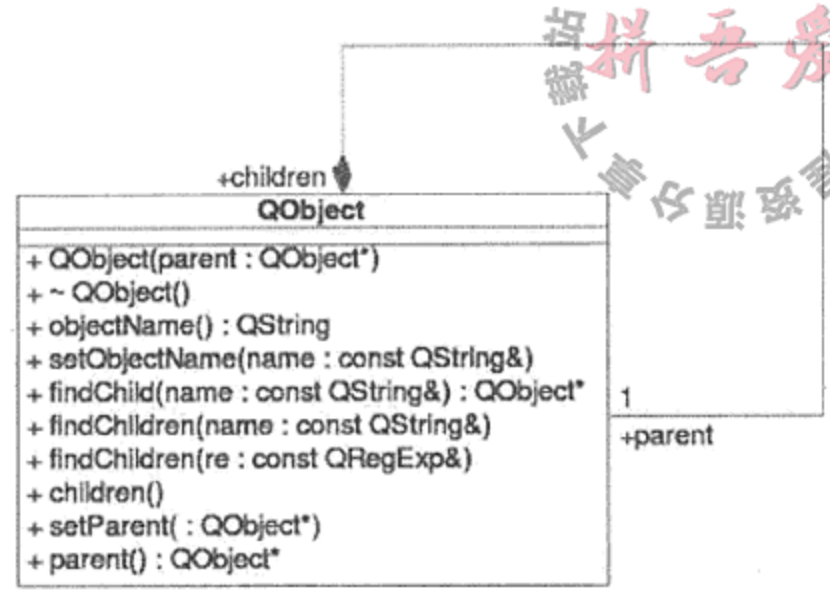


图 8.3　QObject：复合和组件

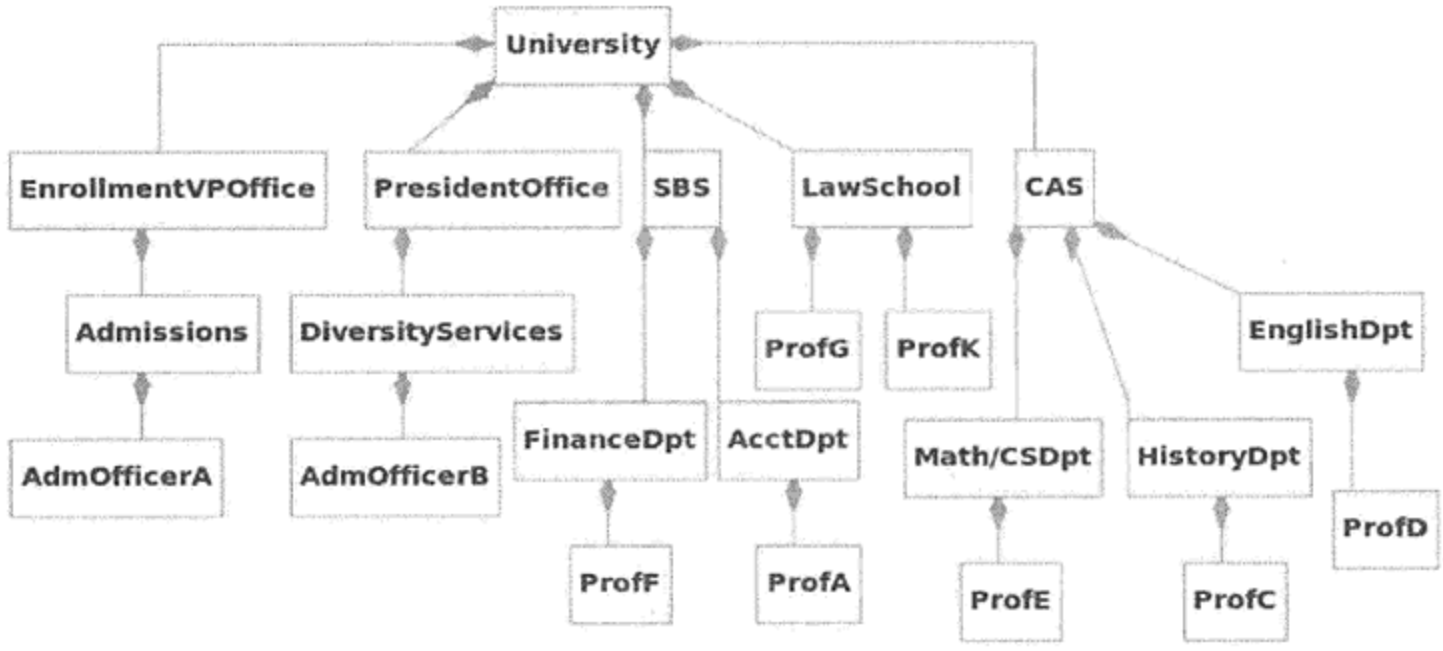


图 8.4　萨福克大学的组织图

这幅图中的每个矩形框都是一个组件。它可能是一个复合组件并拥有子组件，而反过来，这些子组件又可能是复合组件或者是单独的一个组件。例如，PresidentOffice 有单独的雇员(例如，校长及其助手)和子办公室(例如，DiversityServices)。这棵树的叶子是整个组织的个人雇员。

可以使用组合模式来对这所大学进行建模。树中的每一个节点都可以使用下面的类的一个 OrgUnit 型对象来进行表示。

```
class OrgUnit : public QObject {
  public:
    QString getName();
    double getSalary();
  private:
    QString m_Name;
    double m_Salary;
};
```

通过 QObject 的公有接口，可以构建一个类似于树的表达方式来描述这所大学的组织结构，然后编码实例化一个 OrgUnit 并调用 setParent()将其添加到合适的子对象列表中。

对于树中的每一个 OrgUnit 指针 ouptr，可以按照下面的规则来初始化其 m_Salary 数据成员。

- 如果 ouptr 指向一个单一雇员，那么使用此雇员的实际工资。
- 否则，将其初始化为 0。

可以像下面这样来实现 getSalary()方法。

```
double OrgUnit::getSalary() {
  QList<OrgUnit*> childlst = findChildren<OrgUnit*>();
  double salaryTotal(m_Salary);
  if(!childlst.isEmpty())
    foreach(OrgUnit* ouptr, childlst)
      salaryTotal += ouptr->getSalary();
  return salaryTotal;
}
```

可以从任意特定的节点调用 getSalary()方法，返回的结果是以此节点为根的子树所代表的大学中相应部门的工资总和。例如，如果 ouptr 指向 University，那么 ouptr->getSalary()会返回整个大学的总工资；如果 ouptr 指向 EnglishDpt，那么 ouptr->getSalary()会返回英语系的总工资；如果 ouptr 指向 ProfE，那么 ouptr->getSalary()仅返回 ProfE 的个人工资。

8.2.1 查找子对象

每个 QObject 都可以有不限数量的 QObject 子对象。这些子对象的地址会存放在一个特殊的 QObject 指针容器内[①]。QObject 有一个成员函数，可以返回一个指向主对象中全部子对象的指针列表。这个函数的函数原型是

```
const QObjectList& QObject::children() const
```

子对象在该列表中的出现次序是它们在添加到该列表中时的(最初)次序。有一些特定的运行时操作可以改变该次序[②]。

QObject 还提供两个名称为 findChildren()的重载(递归)函数。每个都会返回一个满足特定条件的子对象列表。该函数被重载之后的一种函数原型具有形式

```
QList<T> parentObj.findChildren<T> ( const QString& name = QString() ) const
```

该函数返回一个类型为 T 的子对象列表，其对象名与 *name* 相等。如果 *name* 是空字符串，那么将会起到一个类过滤器的作用，返回结果是一个 QList，其中包含指向全部子对象的指针，而这些子对象都可以通过类型转换变成类型 T 的对象[③]。

示例 8.1 说明如何来调用这个函数。必须在函数名称之后提供一个模板参数。

示例 8.1 src/findchildren/findchildren.cpp

```
[ . . . . ]
/* Filter on Customer* */
    QList<Customer*> custlist = parent.findChildren<Customer*>();
    foreach (const Customer* current, custlist) {
        qDebug() << current->toString();
    }
[ . . . . ]
```

---

① 回忆一下：从 QObject 派生的任何对象都称为 QObject。也就是说，QObject 指针也就可以保存一个派生对象的地址。

② 例如，提升(或者降低)QWidget 子对象，会将该窗口部件置于所有部分重叠同级窗口部件的前面(或者后面)。

③ 19.7 节将讨论类型变换和强制类型转换。

这是一个简单的示例，因此为了让它更真实，假定父对象是 SalesManager，其子列表中或许就会包含指向 Supplier、SupportStaff 和 SalesPerson 对象的指针，相应地，这些对象拥有的子列表或许会包含指向 Customer、TravelAgent 及其他相关对象的指针。

8.2.2　QObject 的子对象管理

示例 8.2 给出的是一个 QObject 派生类[①]。

示例 8.2　src/qobject/person.h

```
[ . . . . ]

class Person : public QObject {
 public:
    explicit Person(QString name, QObject* parent = 0);
    virtual ~Person();
};
[ . . . . ]
```

该类的全部实现如示例 8.3 所示。值得注意的是，~Person() 的确没有做任何显式的对象删除。它只是显示了即将要销毁的对象的名称(出于教学目的)。

示例 8.3　src/qobject/person.cpp

```
#include "person.h"
#include <QTextStream>

static QTextStream cout(stdout);

Person::Person(QString name, QObject* parent)
          : QObject(parent) {
    setObjectName(name);
    cout << QString("Constructing Person: %1").arg(name)
         << endl;
}

Person::~Person() {
    cout << QString("Destroying Person: %1").arg(objectName())
         << endl;
}
```

示例 8.4 中显示的是 growBunch()，它创建了一些对象，将它们添加到其他对象中，然后退出。而当 growBunch() 返回时，会销毁它的所有本地对象。

示例 8.4　src/qobject/bunch.cpp

```
[ . . . . ]
void growBunch() {
    qDebug() << "First we create a bunch of objects." << endl;
    QObject bunch;
    bunch.setObjectName("A Stack Object");                    1
    /* other objects are created on the heap */
```

① 这个例子以一部风行于 1969 年至 1974 年的美国家庭情景喜剧(脱线家族)为基础，该剧曾连续风靡过几十年。如果你出生比较晚而没看过这部电视剧，可从维基文章 http://en.wikipedia.org/wiki/The_Brady_Bunch 中寻找一些有趣的事。

```
    Person* mike = new Person("Mike", &bunch);
    Person* carol = new Person("Carol", &bunch);
    new Person("Greg", mike);                                          2
    new Person("Peter", mike);
    new Person("Bobby", mike);
    new Person("Marcia", carol);
    new Person("Jan", carol);
    new Person("Cindy", carol);
    new Person("Alice");                                               3
    qDebug() << "\nDisplay the list using QObject::dumpObjectTree()"
             << endl;
    bunch.dumpObjectTree();                                            4
    cout << "\nReady to return from growBunch() -"
         << " Destroy all local stack objects." << endl;
int main(int , char**) {
    growBunch();
    cout << "We have now returned from growBunch()."
         <<  "\nWhat happened to Alice?" << endl;
    return 0;
}
[ . . . . ]
```

1 本地栈对象——不是指针。

2 无须让指针指向子对象，因为可以借助对象导向而抵达它们那里。

3 Alice 没有父对象——内存泄漏?

4 只有使用的 Qt 库自身被编译时启用了调试选项，dumpObjectTree()的输出才会在屏幕上出现。

以下是这个程序的输出。

```
src/qobject> ./qobject
First we create a bunch of objects.
Constructing Person: Mike
Constructing Person: Carol
Constructing Person: Greg
Constructing Person: Peter
Constructing Person: Bobby
Constructing Person: Marcia
Constructing Person: Jan
Constructing Person: Cindy
Constructing Person: Alice

Display the list using QObject::dumpObjectTree()
QObject::A Stack Object
    QObject::Mike
        QObject::Greg
        QObject::Peter
        QObject::Bobby
    QObject::Carol
        QObject::Marcia
        QObject::Jan
        QObject::Cindy
```

```
Ready to return from growBunch() - Destroy all local stack objects.
Destroying Person: Mike
Destroying Person: Greg
Destroying Person: Peter
Destroying Person: Bobby
Destroying Person: Carol
Destroying Person: Marcia
Destroying Person: Jan
Destroying Person: Cindy
We have now returned from growBunch().
There is no way to access Alice.
src/qobject>
```


8.2.2.1　练习：QObject 的子对象管理

1. 示例 8.4 中，当 growBunch() 返回时，哪些本地栈中的对象会被销毁掉?
2. 需要注意的是，Alice 并没有在 dumpObjectTree() 的输出中出现。Alice 是何时销毁的?
3. 在 main.cpp 中，写出你自己的 void showTree(QObject* theparent) 函数。在全部对象创建后，这个函数的输出应具有形式

```
Member: Mike - Parent: A Stack Object
Member: Greg - Parent: Mike
Member: Peter - Parent: Mike
Member: Bobby - Parent: Mike
Member: Carol - Parent: A Stack Object
Member: Marcia - Parent: Carol
Member: Jan - Parent: Carol
Member: Cindy - Parent: Carol
```

4. 修改你的 showTree() 函数，以便让它能够与 dumpObjectTree() 函数产生同样的输出。

8.3　QApplication 和事件循环

具有 GUI 的交互式 Qt 应用程序与控制台应用程序和过滤器应用程序[①]的控制流有所不同。这是因为它们是基于事件的。在这些应用程序中，对象之间频繁地通过间接对象相互发送消息。这就使得通过手动线性地跟踪整个代码变得异常复杂。

观察者模式

在编写事件驱动的程序时，GUI 视图需要对数据模型对象的状态变化做出响应，以便它们可以显示最新信息。

当任意数据模型对象发生状态改变时，就需要一种间接的方式来提醒(并且可能向外发送额外的信息)观察者。观察者就是一些正在监听(并且响应)状态变化事件的对象。使用这种消息传递机制的设计模式就称为观察者模式(Observer pattern，有时也称为“发布-预定”模式，publish-subscribe pattern)。

① 过滤器应用程序都不是交互的。它通常从标准输入设备读入数据，然后将其写入标准输出设备。

这种模式有许多种不同的实现，但是它们都具有一些共同的特征：

1. 允许实体观察者类与实体主体类之间解耦。
2. 支持广播风格(一对多)的通信。
3. 所采用的从主体向观察者发送信息的机制完全由主体的基类给定。

Qt 的 QEvent 类封装了底层事件的概念。QEvent 类是若干特定的事件类的基类，例如 QActionEvent, QFileOpenEvent, QHoverEvent, QInputEvent, QMouseEvent 等。QEvent 对象可以由窗口系统创建以响应用户的动作(例如，QMouseEvent)，或按照指定的时间间隔(QTimerEvent)完成创建，也可以由应用程序显式地创建。成员函数 type() 会返回一个枚举，其中含有近百个特定的值，以区分不同种类的各式事件(例如，关闭、DragEnter、DragMove、放下、输入、GrabMouse、HoverEnter、KeyPress、MouseButtonDblClick、MouseMove、Resize 等)。

事件循环是一个程序结构，它能够将事件划分优先级，排队并分派给一些对象。编写一个基于事件的应用程序就是实现由函数组成的被动接口，且这些函数仅仅在某些特定事件发生时才会得到调用，如鼠标单击、触摸手势、敲击按键、发射信号、各类窗口管理器事件或来自其他程序的消息等。事件循环通常会一直运行，直到遇到某个终止事件。(例如，用户发出了退出的动作，才会关闭最后的窗口，等等。)

一个典型的 Qt 程序会创建对象，连接各个对象，然后再告诉应用程序开始 exec()。在运行时，应用程序就进入了事件循环。各个对象之间可以通过各种方式相互发送信息。典型的 main() 函数具有如下形式：

```
int main(int argc, char ** argv) {
  QApplication app(argc, argv);
  FancyWidget fwidg;
  fwidg.show();          // returns immediately
  return app.exec();     // enters event loop
}
```

在 main() 的最后，在 return 语句中才出现对 QApplication::exec() 函数的调用。应用程序的整个工作部分开始于该函数的调用，终止于该函数的返回。详细部分位于 FancyWidget 类的定义和实现中。那是相当典型的一个 Qt 事件循环应用程序。

事件与信号和槽的比较

事件可认为是低级消息，目标是某个特定的对象。信号可以认为是高级消息，很有可能会连接到许多槽上。只有在事件循环，特别是由 QApplication::exec() 进入的事件循环中，信号才能发送到槽上。这是因为信号和槽在其外表之下是使用事件循环来传递消息的。

这里的信号是指对事件进行封装的信号。

8.4 Q_OBJECT 和 moc 一览表

QObject 支持一些普通 C++对象通常所没有的特性：

- 信号和槽(参见 8.5 节)。
- 元对象、元属性、元方法(参见第 12 章)。

- qobject_cast(参见 12.2 节)。

这些特性中的某些特性只有借助生成的代码才能够实现。元对象编译器，即 moc，会针对每个使用 Q_OBJECT 宏的 QObject 派生类生成额外的函数，生成的代码可以在名称为 moc_filename.cpp 的文件中找到。

这也就意味着，当 moc 无法发现或处理工程中某个类时，编译器或者链接器会报出的一些莫名其妙含混不清的错误。为了保证 moc 能够处理所编写的全部 QObject 派生类，下面是编写 C++代码和 qmake 工程文件时应该遵守的一些指导原则。

- 每个类的定义都应该放在各自对应的.h 文件中。
- 每个类的实现都应当放在相应的.cpp 文件中。
- 为避免头文件的多次包含，头文件应该“封装”起来(例如，用#ifndef)。
- 每个.cpp 源文件都应当列举在工程文件的 SOURCES 变量中，否则，它将不会被编译。
- 每个头文件都应当列举在.pro 工程文件的 HEADERS 变量中。否则，moc 将不会对其进行预处理。
- Q_OBJECT 宏必须出现在每个 QObject 派生类定义的头文件中，以便让 moc 知道要为其生成代码。

注意

因为每个 Q_OBJECT 宏都会产生代码，所以它需要使用 moc 进行预处理。moc 是在这种假设下工作的：只会从 QObject 类派生了一次。更进一步说，QObject 类应该是其基类列表中的第一个基类。如果在实际应用中不小心多次继承了 QObject，或者它不是继承列表中的第一个基类，那么就可能会从 moc 生成的代码中发现一些非常奇怪的错误。

注意

如果定义了一个 QObject 派生类，构建了一个应用程序，然后才意识到需要在类的定义中添加一个 Q_OBJECT 宏，而且是在该工程是使用了旧的 Makefile 文件构建之后添加的，此时必须使用 qmake 更新一下该 Makefile。

不然，make 并不能智能到在 Makefile 中对这样的文件添加 moc 的步骤。通常执行 clean rebuild 命令并不能修复这种问题。该问题经常使得经验不足的 Qt 开发者非常头痛。关于该错误消息的更多信息，可参阅附录 C 的 C.3.1 节。

8.5　信号和槽

信号是在类定义中给出的类似于 void 函数声明的一种消息。它有参数列表却没有函数体。信号是一个类的接口的一部分。它看起来像函数，但不用同样的方式进行调用——它被此类的对象发射。

槽通常是一个 void 成员函数。它可以像普通的成员函数一样进行调用，或者可以由 QMetaObject 系统进行间接调用。

一个对象的信号可以与一个或者多个对象的槽相连接，前提是这些对象存在并且参数列表从信号到槽都是赋值兼容的[①]。连接语句的语法是

```
bool QObject::connect(senderQObjectPtr,
                      SIGNAL(signalName(argumentList)),
                      receiverQObjectPointer,
                      SLOT[②](slotName(argumentList))
                      optionalConnectionType);
```

任何拥有信号的 QObject 都可以发射出那样的信号。这就会引起对全部连接的槽的间接调用。

类似于函数调用，在发射语句中传递的参数可以在槽函数内通过参数进行访问。参数列表是从一个对象向另一个对象传递信息的方式。optionalConnectionType 让你可以明确说明，你是否希望从发射点处同步(阻塞)或者非同步(排队)地调用目标槽[③]。

在 1.11 节，我们讨论过一个用到了 QInputDialog 窗件(见图 8.5)的例子。在运行那个应用程序时，用户通过输入一个值，然后再通过左键单击 Cancel 按钮或者 OK 按钮，实现与第一个对话框的交互。

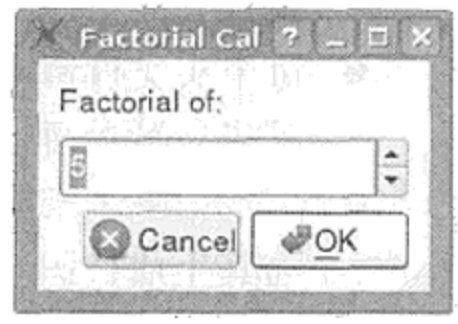
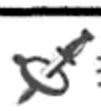


图 8.5 QInputDialog

鼠标左键的释放事件，是鼠标单击中的最后一步，引起选中按钮窗件发射 clicked()信号。那个信号是一系列说明鼠标指针(在按钮的矩形框内)位置的鼠标底层事件的集合，并可以确保鼠标按钮操作的正确次序。例如，鼠标左键按下然后再释放时，可能仍旧还在该矩形框内。换句话说，鼠标事件已经组合起来构成了 clicked()信号。设计良好的窗件 API 应该适当包含一组信号，使得用户不必与底层事件打交道，除非是要开发自定义窗件。


提示

如果有多个信号连接到同一个槽上且需要知道是哪一个 QObject 发射的信号，则可以在该槽中调用 sender()，它会返回一个指向那个对象的指针。

8.5.1 更多探讨

与 Qt 的 QObject 模型类似，还有其他一些信号和槽的开源实现。其中之一称为 XLObject[④]。与 Qt 相比，它不要求任何 moc 风格的预处理，但非常依赖模板，因此只有(2002 年以后的)现代 C++编译器才支持它。Boost 库[⑤]也包含一种信号和槽的实现。

8.6 QObject 的生命周期

这一节将给出一些很好地管理 QObject 生命周期的实践经验。

---

① 当列表是赋值兼容时，意味着对应的参数必须是兼容的。在 Qt 中，槽必须至少有与信号一样多的参数。槽可以忽略多余的参数。

② 或者 SIGNAL——从一个信号连接另一个信号也是可行的。

③ 连接不会局限于当前线程(参见 17.2 节)。

④ 参见http://sourceforge.net/projects/xlobject。

⑤ 参见http://www.boost.org。

警告

确保每一个 QObject 在 QApplication 之后创建，在 QApplication 销毁前销毁，这一点至关重要。静态存储区创建的对象将在 main()返回后才被销毁，这就太迟了。这就意味着绝不应该定义静态存储类的 QObject。

栈还是堆

一般情况下，没有父对象的 QObject 应当在栈上创建，或者定义成另一个类的子对象。有父对象的 QObject 不应在栈上创建。因为那样的话，它有可能会被删除两次。在堆上创建的所有 QObject 都应当或者是有父对象的，或者是由其他对象进行管理的。

提示

不推荐直接删除 QObject。在带有事件循环的程序中，最好是利用 QObject::deleteLater()来删除 QObject。这样做，可以在应用程序处理事件并在当前槽返回之后就安排该对象的销毁。

希望能够在槽内删除信号的 sender()，这么做实际上也是必须的(参见示例 17.20)。

8.7 QTestLib

编写测试代码最常用的方式是将其组织在一个基于单元的框架内。Qt 4 引入了由辅助宏和测试运行器构成的 QTestLib 框架，用来简化使用 Qt 编写的应用程序和库的单元测试的编写。在这个框架中，所有的公有方法都位于 QTest 命名空间中。下面的几个示例会用到这个框架。

TestClass 是一个派生自 QObject 的类，有一些用于一个或多个测试函数的私有槽。测试用例就是一系列将要执行的测试函数。每个测试用例都必须在其自己的工程中，该工程有一个含有 QTEST_MAIN()宏的 source 模块。QTEST_MAIN()用来说明哪个类是这个测试的入口点。它会扩展成一个创建实例并依照声明次序执行其私有槽的 main()函数。还可以提供一些用于初始化和清理的其他方法，分别是 initTestCase()和 cleanupTestCase()，会分别在测试用例的开头和结尾处调用它们。

为了演示 QTestLib 的用法，我们为 QCOMPARE 和 QVERIFY 写一个测试，这两个宏会在 Qt 的测试用例中用作断言。这些宏仅能用在测试类中，而 Q_ASSERT 则可用于任何地方。当断言失败时，它们能比 Q_ASSERT 提供更多的信息。

任何用到了 QTestLib 模块的工程，都必须在.pro 文件中用下面这行代码进行启用：

```
CONFIG += qtestlib
```

如示例 8.5 所示，第一步要定义一个含有测试函数的 QObject 派生类。包含 QtTest 头文件并把各个测试函数声明成私有槽是必要的。

示例 8.5　src/libs/tests/assert/testassertequals.h

```
[ . . . . ]

#include <QtTest>
```

```
class TestAssertEquals:public QObject {
    Q_OBJECT
private slots:
    void test ();
};

[ . . . . ]
```

这个测试会试验所有各种验证表达式的类型。示例 8.6 是处理 bool 表达式的部分代码。

示例 8.6 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

void TestAssertEquals::test () {
    qDebug() << "Testing bools";
    bool boolvalue = true;
    QVERIFY (1);
    QVERIFY (true);
    QVERIFY (boolvalue);
    qDebug () << QString ("We are in file: %1  Line: %2").
            arg (__FILE__).arg (__LINE__);
    QCOMPARE (boolvalue, true);        1
```

1 用布尔值测试 EQUALS。

示例 8.7 是处理 QString 表达式部分的实现代码。

示例 8.7 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing QStrings";
    QString string1 = "apples";                                    1
    QString string2 = "oranges";
    QString string3 = "apples";
    QCOMPARE ("apples", "apples");                                 2
    QCOMPARE (string1, QString("apples"));
    QCOMPARE (QString("oranges"), string2);
    QCOMPARE (string1, string3);
    QVERIFY (string2 != string3);
```

1 用字符串值测试 EQUALS。

2 测试 QString 和 char *的比较。

示例 8.8 用来处理 QDate 和 QVariant 表达式。通常，test()函数会在第一个测试失败的地方停下，当 QVERIFY(*condition*)碰到 *condition* 的值是 false 或者是碰到 QCOMPARE(*actual*, *expected*)在 *actual* 和 *expected* 不相等时，就会导致失败。因此，示例 8.8 中故意包含了一个 QCOMPARE()错误。要继续测试，就需要在这个故意失败的前面放上 QEXPECT_FAIL()宏。

示例 8.8 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing QDates";
    QString datestr ("2010-11-21");
```

```
QDate dateobj = QDate::fromString (datestr, Qt::ISODate);
QVERIFY (dateobj.isValid ());
QVariant variant (dateobj);
QString message(QString ("comparing datestr: %1 dateobj: %2 variant: %3")
       .arg (datestr).arg (dateobj.toString ()).arg (variant.toString ()));
qDebug() << message;
QCOMPARE (variant, QVariant(dateobj));                                    1
QCOMPARE (QVariant(dateobj), variant);
QCOMPARE (variant.toString(), datestr);                                   2
QCOMPARE (datestr, variant.toString());
QEXPECT_FAIL("","Keep going!", Continue);
QCOMPARE (datestr, dateobj.toString());                                   3
```

1 比较 QDate 和 QVariant。

2 比较 QVariant 和 String。

3 比较 QDate 和 QString。

示例 8.9 处理含有 int，long 和 double 项的表达式。在示例 8.9 中我们插入了两个 QVERIFY()失败。我们在第一个失败的前面放一个 QEXPECT_FAIL()宏。允许第二个失败可以停止该测试。注意位于函数定义下面的宏 QTEST_MAIN，它将生成适当的 main()函数代码。

示例 8.9　src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing ints and doubles";
    int i = 4;                                                            1
    QCOMPARE (4, i);
    uint u (LONG_MAX + 1), v (u / 2);
    QCOMPARE (u, v * 2);
    double d (2. / 3.), e (d / 2);
    QVERIFY (d != e);
    QVERIFY (d == e*2);
    double f(1./3.);
    QEXPECT_FAIL("","Keep going!", Continue);
    QVERIFY (f * 3 == 2);
    qDebug() << "Testing pointers";
    void *nullpointer = 0;
    void *nonnullpointer = &d;
    QVERIFY (nullpointer != 0);
    qDebug() << "There is one more item left in the test.";
    QVERIFY (nonnullpointer != 0);
}

// Generate a main program
QTEST_MAIN(TestAssertEquals)
```

1 整数测试。

示例 8.10 给出了这一测试的输出。值得注意的是，并没有看到最后的 qDebug()的输出信息。

示例 8.10　src/libs/tests/assert/testassert.txt

```
********* Start testing of TestAssertEquals *********
Config: Using QTest library 4.6.2, Qt 4.6.2
```

```
PASS   : TestAssertEquals::initTestCase()
QDEBUG : TestAssertEquals::test() Testing bools
QDEBUG : TestAssertEquals::test() "We are in file:
testassertequals.cpp  Line: 15"
QDEBUG : TestAssertEquals::test() Testing QStrings
QDEBUG : TestAssertEquals::test() Testing QDates
QDEBUG : TestAssertEquals::test() "comparing datestr: 2010-11-21
dateobj: Sun Nov 21 2010 variant: 2010-11-21"
XFAIL  : TestAssertEquals::test() Keep going!
   Loc: [testassertequals.cpp(46)]
QDEBUG : TestAssertEquals::test() Testing ints and doubles
XFAIL  : TestAssertEquals::test() Keep going!
   Loc: [testassertequals.cpp(59)]
QDEBUG : TestAssertEquals::test() Testing pointers
FAIL!  : TestAssertEquals::test() 'nullpointer != 0' returned FALSE.
()
   Loc: [testassertequals.cpp(63)]
PASS   : TestAssertEquals::cleanupTestCase()
Totals: 2 passed, 1 failed, 0 skipped
********* Finished testing of TestAssertEquals *********
```


8.8 练习：QObject, QApplication，信号和槽

1. 重写4.4节中的Contact和ContactList，以使它们都从QObject派生出来。当在ContactList中添加一个Contact时，要让Contact成为ContactList的子对象。
2. 用你在本练习中编写的客户代码将Contact和ContactList替换成新版本。

8.9 复习题

1. 当QObject A是QObject B的父对象时，含义是什么?
2. 哪些QObject不需要父对象?
3. 当一个QObject重父化(reparent)时，会发生什么事情?
4. 为什么QObject的复制构造函数不是公有的?
5. 什么是组合模式?
6. 在什么情况下，QObject既可以是复合对象又可以是组件对象?
7. 如何访问一个QObject的子对象?
8. 什么是事件循环?它是如何启动的?
9. 什么是信号?如何调用一个信号?
10. 什么是槽?如何调用一个槽?
11. 信号和槽是如何进行连接的?
12. 在多个信号连接到同一个槽的情况下，如何判定是哪一个QObject发射的信号?
13. 信息是如何从一个对象传递给另外一个对象的?
14. 一个类从QObject派生多次可以引起一些问题。什么情况下会意外地发生这种情况?
15. 值类型和对象类型之间有何区别?试举出几个例子。

第9章　窗件和设计师

这一章将简要介绍 Qt 库中的图形用户界面(GUI)的构成模块，即窗件(widget)，并会包含一些如何使用窗件的简单示例。将用 QtCreator 和设计师(Designer)探索各个窗件、窗件的特性以及它们与用户代码的结合方式。

窗件就是一个 QWidget 派生类的对象，它能够在屏幕上显示。QWidget 的基本结构形式如图 9.1 所示。

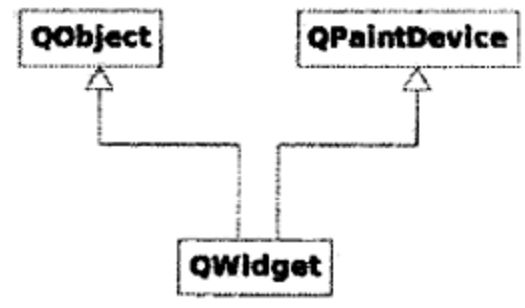


图 9.1　QWidget 的继承层次

QWidget 是一个采用了多重继承(参见 22.3 节)的类。首先，QWidget 是一个 QObject，因此它可以有父对象、信号、槽以及可受管理的子对象。同时，QWidget 也是一个 QPaintDevice，这个类是所有可在屏幕上进行“绘制”的对象的基类。

QWidget 与其子对象交互的方式非常有趣。没有父对象的窗件称为窗口(window)。如果一个窗件是另外一个窗件的父对象，那么子窗件的边界将完全置于父窗件的边界内部[①]。所有被包含的窗件将按照布局规则(参见 9.6 节)进行显示。

QWidget 可以接收来自窗口系统内各种实体的信号(例如，鼠标、键盘、计数器、其他进程，等等)，通过对这些信息的响应，QWidget 可以处理各种事件。QWidget 能够将其自身的矩形图像绘制在屏幕上。同样也可以从屏幕上移除自身而并不妨碍当前显示在屏幕上的所有其他事物。

典型的桌面 GUI 应用程序可以包含许多(即使达到数百个也不稀奇)不同的 QWidget 派生类对象，它们会根据父-子关系进行配置，并根据具体应用程序的布局进行排列。

QWidget 被认为是所有 GUI 类中最为简单的，这是因为它看上去就像一个空盒子。但是，该类本身非常复杂，它包含了数百个函数。当复用 QWidget 及其子类时，其实就已经站在了巨人的肩膀上，这是因为 QWidget 是基于几个 Qt 代码层创建的，而取决于你所处的平台(Linux 中的 X11，MacOS 中的 Cocoa 以及 Windows 上的 Win32)，这些代码又在不同的本地系统代码层次之上。

9.1　窗件的分类

Qt 窗件可以按照几种方式进行划分，如此就可以更轻松地查找那些想要使用的类。而较为复杂的 QWidget 可能会被划入不止一个的子类中。这一节将简要介绍一些在开始学习 GUI 编程时可能需要用到的类。

QWidget 可以分为四类基本窗件。按钮窗件(button widget)的 Windows 风格的显示效果如图 9.2 所示。输入窗件(input widget)的 Plastique 风格的显示效果如图 9.3 所示。

---

① 这一规则存在一些例外。例如，悬浮的 QDockWidget 或者 QDialog 就可以位于父窗口部件的边界之外但仍会在父窗口部件的“前面”。10.2 节将讨论 QDockWidget。

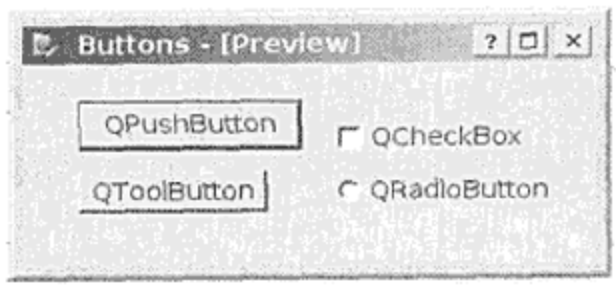


图 9.2 按钮窗件

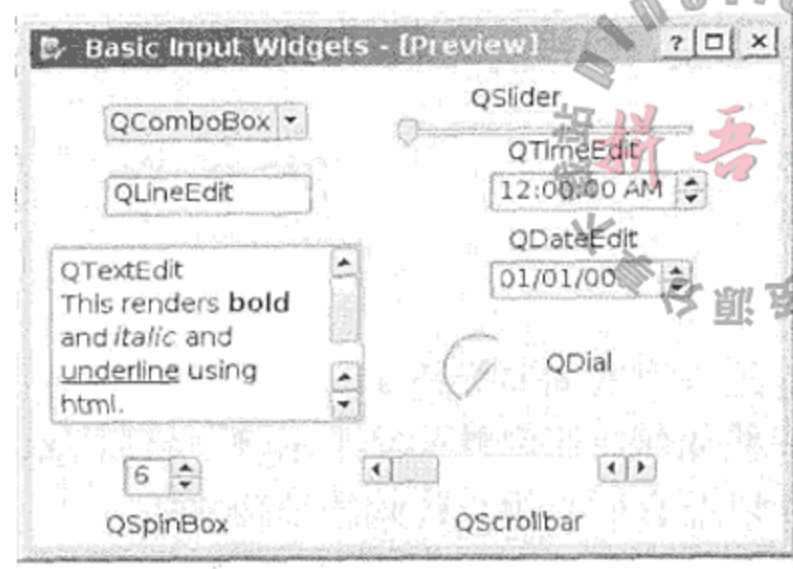


图 9.3 输入窗件

显示窗件(display widget)是不可交互的，如 QLabel，QProgressBar 和 QPixmap。容器窗件(container widget)，如 QMainWindow，QFrame，QToolBar，QTabWidget 和 QStackedWidget，可以包含其他窗件。

- 上述各类窗件可用作构建模块来创建其他更为复杂的窗件，比如 QFileDialog，QInputDialog 和 QErrorMessage。
- 能够显示数据集合的视图，诸如 QListView，QTreeView，QColumnView 和 QTableView，这四类的显示效果如图 9.4 所示。

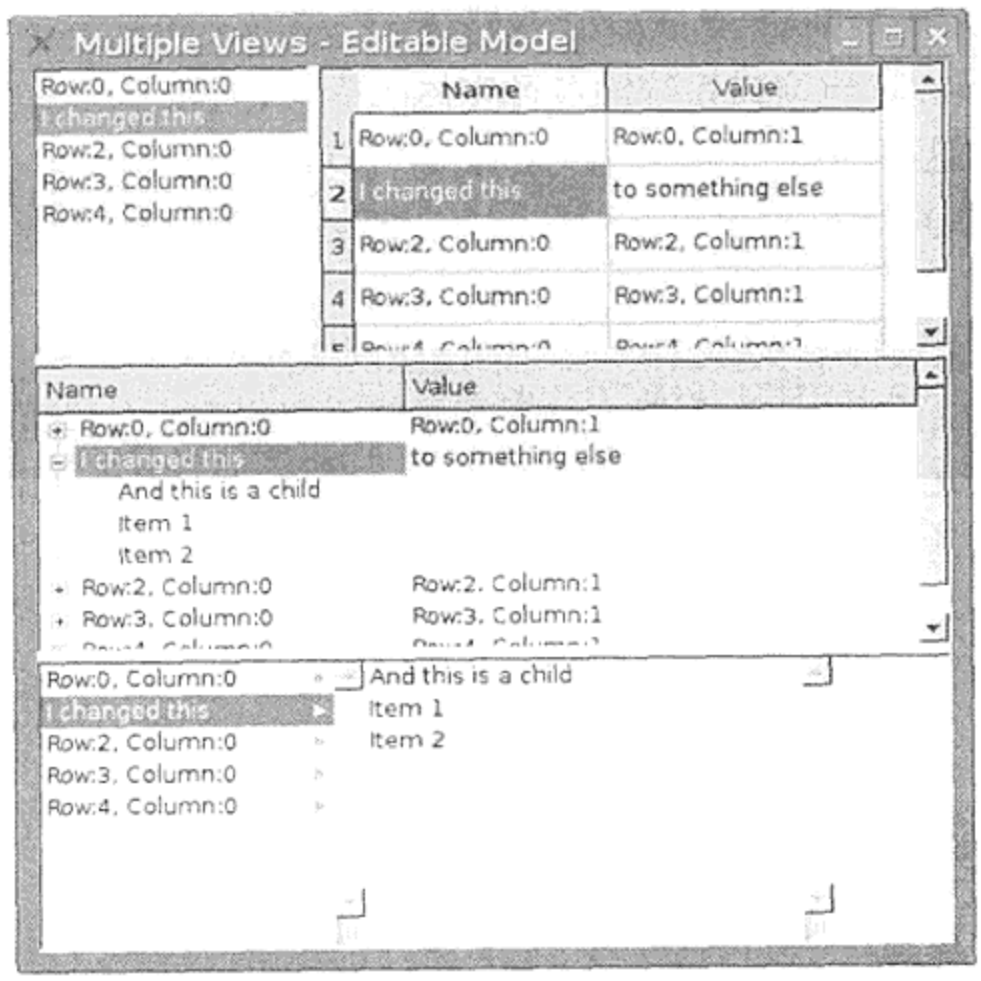


图 9.4 一个模型的四种视图

此外，还有一些 Qt 类没有任何图形化的显示(所以它们都不是窗件)，但是会在 GUI 开发中使用到。这些类主要包括如下几种。

- Qt 数据类型——QPoint，QSize，QColor，QImage 和 QPixmap 是在处理图形对象时经常用到的类型。
- 布局——这些类能够动态地管理窗件的布局。其中有些是常用的特殊布局，包括 QHBoxLayout，QVBoxLayout，QGridLayout，QFormLayout 等。

- 模型——QAbstractItemModel 及其各个派生类，如 QAbstractListModel 和 QAbstractTableModel，外加一些已有的可继承实体类，如 QSqlQueryModel 和 QFileSystemModel，都是 Qt 模型/视图框架中的一部分，该框架内置将一个模型和其他不同视图相连接的机理，以便对一个组件的修改可以自动变换到其他组件上。
- 控制器类——QApplication 和 QAction 两者都是管理 GUI 应用程序控制流的对象。QItemDelegate 用作模型和视图之间的控制器。

如果希望更详细地了解不同风格的窗件，可参考 *The Qt Widget Gallery*[①]，其中包含了各种屏幕截图和源代码，并以不同的风格渲染窗件。

9.2　设计师简介

设计师(Designer)是用于应用程序构图和编辑的一个图形化程序，它拥有拖放式的接口和大量节省编程时间和编程精力的一系列特性。设计师的设计阶段的输出是一个称为 *classname*.ui 的 XML 文件，其中的 *classname* 通常是由设计阶段的开始部分确定的。

在 QtCreator 中，无论何时打开一个 .ui 文件，设计师都会以“设计模式”的形式包含到 QtCreator 中。只要有相应合适的集成包存在，设计师也可作为嵌入型应用程序而集成到其他集成开发环境(IDE)中去（例如，Eclipse, Microsoft Developer Studio, Xcode 等）。

classname.ui 文件可以描述一个包含子对象、布局和各个内部连接的设计师窗件。如 9.7 节所述，XML 文件会被翻译成 C++头文件，但本节仅介绍用设计师来包合 GUI 的用法。

可以从窗件工具箱（Widget Box）将窗件拖放到中央的窗件编辑器（Widget Editor）处。拖放了一些窗件后，就可以在对象检查器（Object Inspector）中选择其中的任意窗件，并在属性编辑器（Property Editor）中查看它的各个属性，如图 9.5 所示。

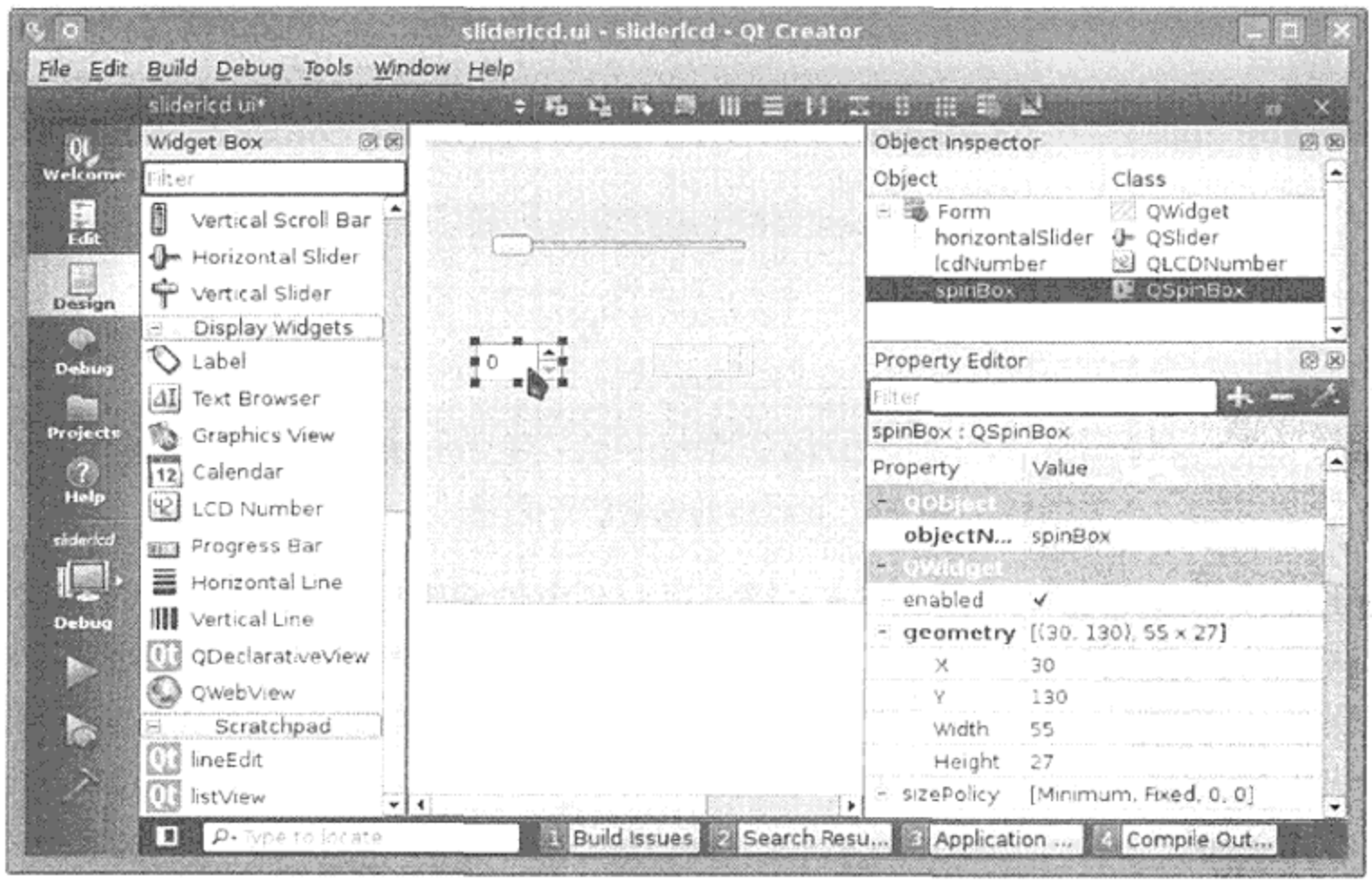


图 9.5　处于设计模式的 QtCreator

① 参见http://doc.trolltech.com/4.6/gallery.html。

在属性编辑器中可以修改任意的属性，而这些修改将会使 `classname.ui` 文件内发生相应的变化。如果编辑的属性会改变窗件的外观，那么在窗件编辑器中可以即刻看到所做的修改。

注意

绝大多数的对象可以从窗件编辑器或者对象检查器中进行重命名，或者是通过单击鼠标左键来选中该窗件然后再按下 F2 键进行重命名，或者是通过在该窗件上双击来重命名。如果愿意，也可以设置属性编辑器上的 *objectName*。然后，需要重点注意的是，输入的各个窗件名是大小写敏感的，因为输入的那些名称将还会成为由用户界面编译器(User Interface Compiler，UIC)生成的类的数据成员名。

通过切换到编辑信号/槽(Edit Signals/Slots，快捷键 F4)模式，可以在各窗件之间拖放各个连接，如图 9.6 所示。在进入编辑信号/槽模式时，窗件工具箱会无法使用，此时就可以从一个带信号的窗件上拖动连接到另一个带槽的窗件上(在窗件编辑器上)。

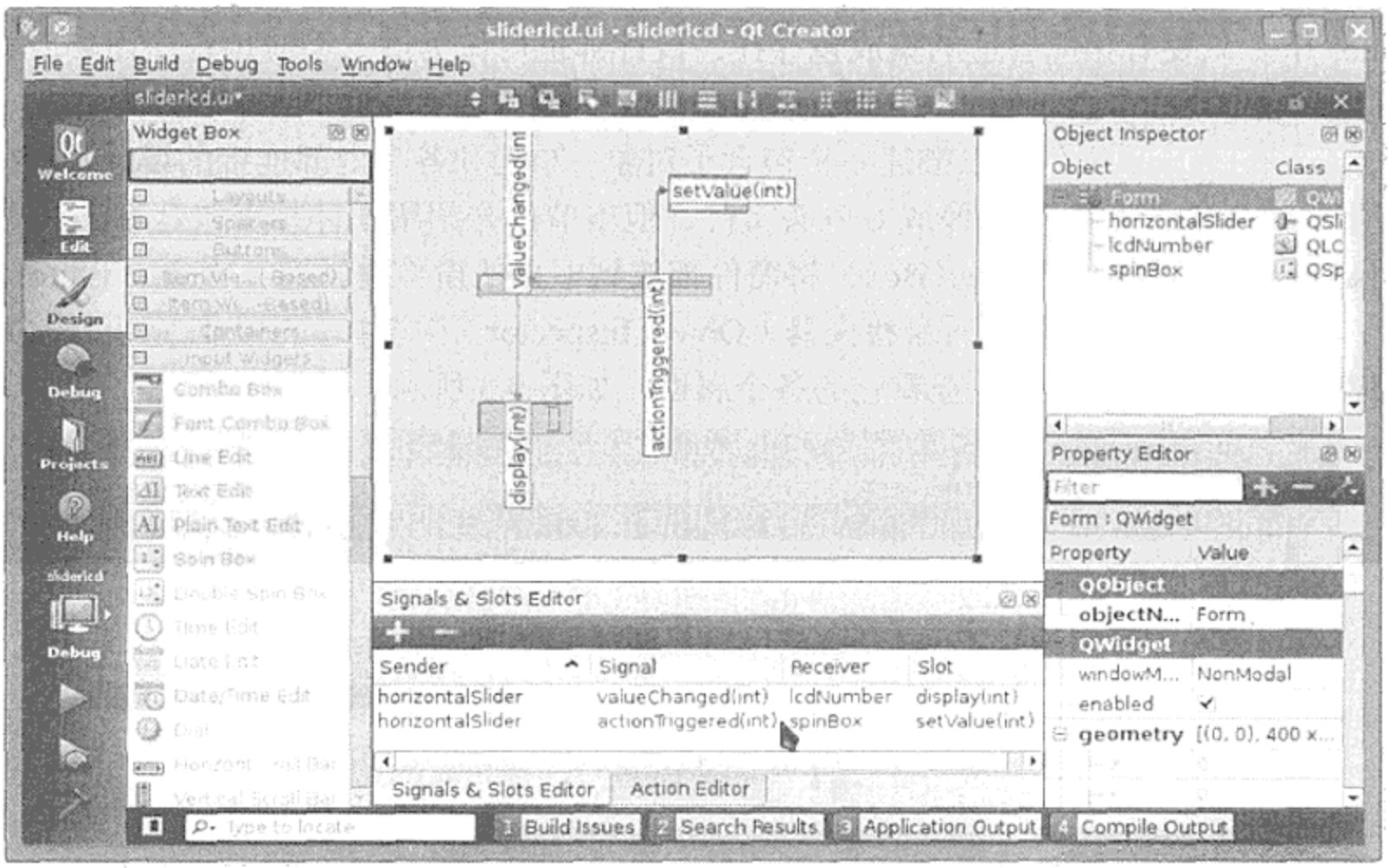


图 9.6 设计师的编辑信号/槽模式

在新的连接构造完成后，通过按 F3 键可以回到窗件编辑模式。通常可以让信号/槽编辑器变成可停靠的方式来观察或者编辑已有的各个连接(如图 9.6 靠近底部的位置所示)，而无须考虑窗件编辑器是窗件编辑模式、连接模式还是伙伴(buddy)编辑模式。利用预览窗件(Preview Widget，快捷键为 Ctrl + Alt + R)模式，就可以对刚刚构建的各个连接进行动态行为预览，如图 9.8 所示。

尽管到现在为止还没有编写任何代码，但是可以立即通过设计师对新创建的 ui 文件的动态行为进行预览。9.7 节描述了如何将 ui 文件和自己(处理数据)的类集成。

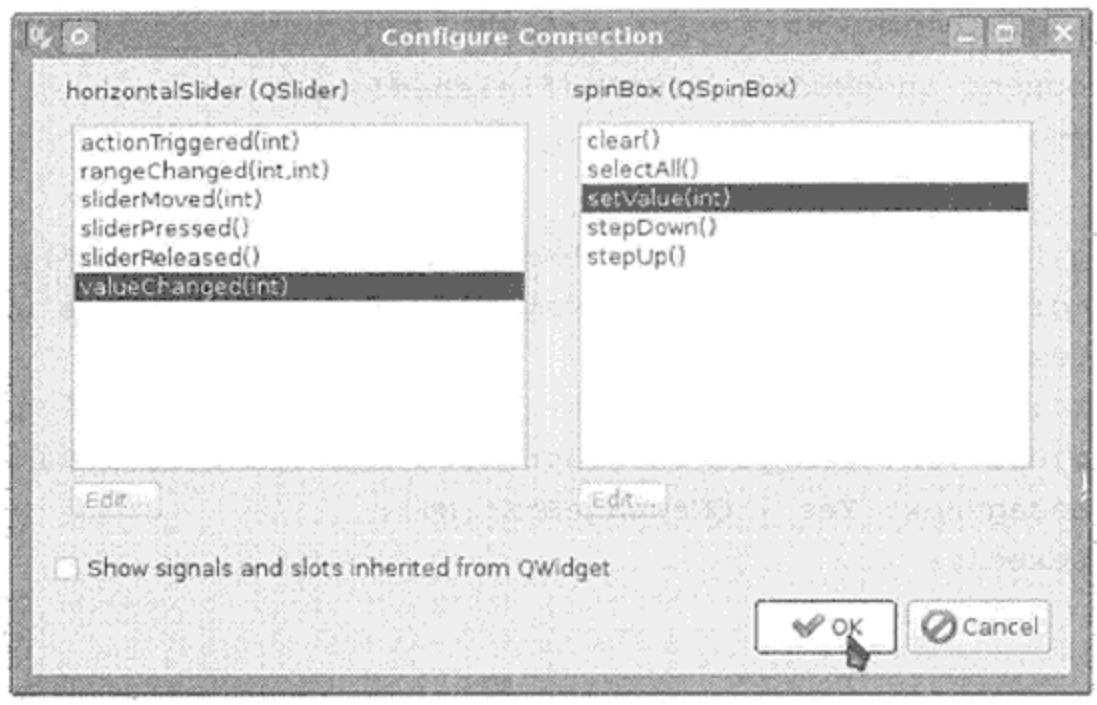


图 9.7　配置连接对话框

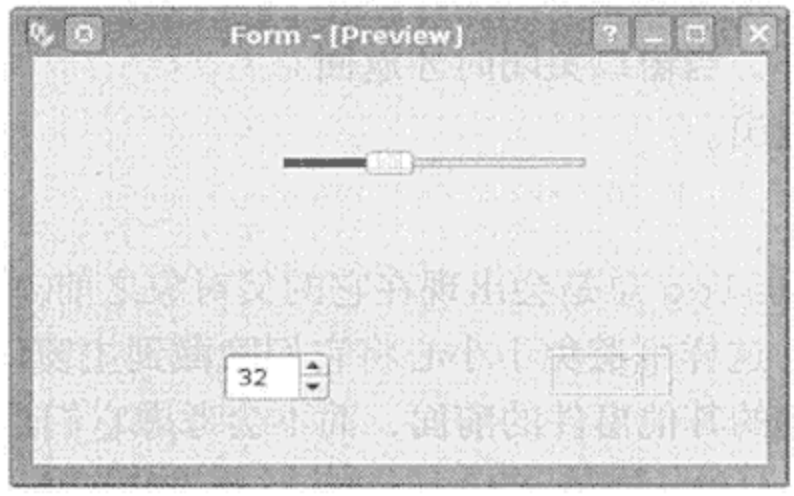


图 9.8　窗件预览

9.3　对话框

QDialog 是 Qt 所有对话框的基类。对话框窗口通常用于和用户进行简单交互。对话框窗口可以是模态(modal)对话框也可以是非模态(nonmodal)对话框。当程序调用静态的便利函数“QMessageBox::”或者“QFileDialog::”时，弹出的对话框就是模态对话框。当模态对话框显示在屏幕上时，它会冻结同一应用程序中的其他所有可见窗口的输入功能。用户解除模态对话框后，与应用程序的常规交互才可以继续下去。QDialog::exec()是将模态对话框放到屏幕上的另一种方式。当用户完成了所需的响应后，对话框就可以返回数据(例如，字符串或者数值)，也可以返回对话框代码(QDialog::Accepted 或者 QDialog::Rejected)。

可以像 QWidget 一样通过 show()显示一个 QDialog。在此情况下，对话框是非模态的，用户也就可以与应用程序的其他窗口继续交互。示例 9.1 给出了模态对话框和用 show()弹出的非模态对话框的区别。

示例 9.1　src/widgets/dialogs/modal/main.cpp

```
[ . . . . ]
#include <QtGui>
int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
    QProgressDialog nonModal;
    nonModal.setWindowTitle("Non Modal Parent Dialog");
```

```
    nonModal.show();                                                    1
    nonModal.connect(&nonModal, SIGNAL(finished()),
            &app, SLOT(quit()));                                        2
[ . . . . ]
    QFileDialog fileDialog(&nonModal, "Modal File Child Dialog");
    // 2 modal dialogs. exec() takes over all user interactions until closed.
    fileDialog.exec();                                                  3
    QMessageBox::question(0, QObject::tr("Modal parentless Dialog"),
            QObject::tr("can you interact with the other dialogs now?"),
            QMessageBox::Yes | QMessageBox::No);
    return app.exec();                                                  4
}
[ . . . . ]
```

1 立即返回。

2 结束的情形。

3 类似进入了事件循环，当窗口关闭时才返回。

4 当非模态关闭时才退出。

父对象和子对象

一个带有父窗件的 QDialog 总是会出现在它的父对象之前。单击父对象同时会显示该对话框。对于非模态对话框，这样可避免不小心将它们隐藏到主窗口的后面。模态对话框通常会出现在由该应用程序创建的其他窗件的前面，而不会考虑它们是何种父-子关系。

输入对话框

已经预定义了一些可复用的输入对话框。如图 9.9 所示，在目录$QTDIR/examples/dialogs/standarddialogs 中，给出了其中一些最为常用的输入对话框。

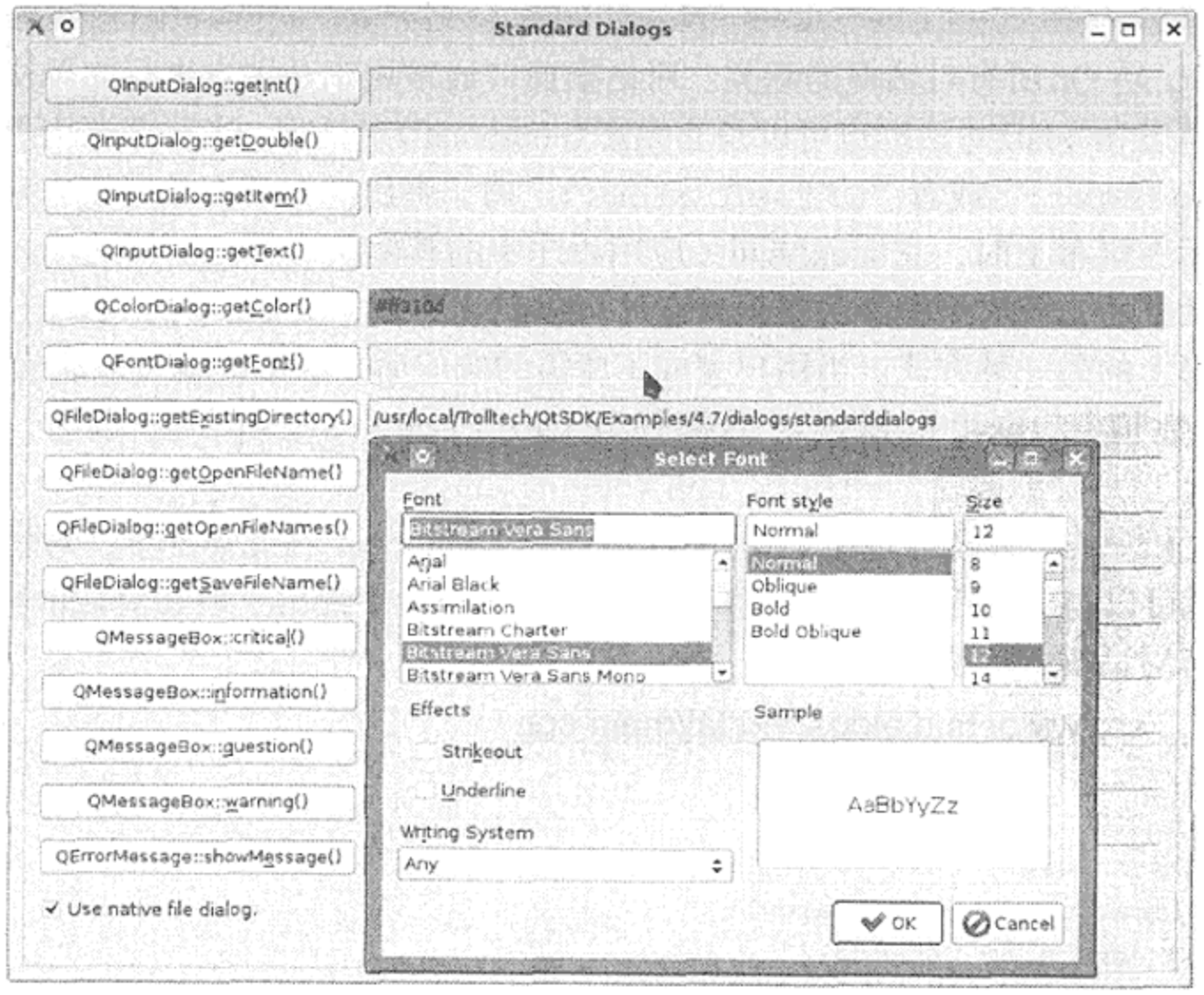


图 9.9 标准对话框

9.4　窗体的布局

有时，可能需要用一些自定义的输入域来创建自己的对话框。图 9.10 中给出了位于 QVBoxLayout 中的 QformLayout，它可从用户那里获得 Qstring，QDate 和 QColor 值。可以用一些 QLayout 型对象将窗件组织到网格中，再嵌套到行、列或者窗体中去。

无论是通过手工方式还是通过设计师，都可以通过 QFormLayout 类来简化窗体的创建过程。它会创建两列：一列用于大小固定的标签，另一列用于大小可变的输入窗件。下面给出的是 InputForm 的类定义。

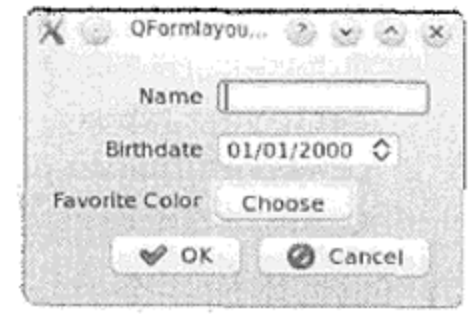


图 9.10　QFormLayout示例

示例 9.2　src/layouts/form/inputform.h

```
#ifndef INPUTFORM_H
#define INPUTFORM_H

#include <QDialog>
class QLineEdit;
class QDateEdit;
class QPushButton;
class QDialogButtonBox;

class InputForm : public QDialog {
    Q_OBJECT
public:
    explicit InputForm(QWidget* parent = 0);
    void updateUi();
protected slots:
    void accept();
    void chooseColor();
private:
    QColor m_color;
    QLineEdit* m_name;
    QDateEdit* m_birthday;
    QPushButton* m_colorButton;
    QDialogButtonBox* m_buttons;
};

#endif // INPUTFORM_H
```

示例 9.3 给出了 InputForm 构造函数中的几行代码。对 addRow() 的每个调用都会在该行代码所对应的布局中添加一个输入窗件以及它所对应的 QLabel。没有必要明确地构建每个 QLabel。

示例 9.3　src/layouts/form/inputform.cpp

```
[ . . . . ]

    m_name = new QLineEdit;
    m_birthday = new QDateEdit;
    m_birthday->setDisplayFormat("dd/MM/yyyy");
    m_colorButton = new QPushButton(tr("Choose"));
    m_colorButton->setAutoFillBackground(true);
```

```
m_buttons = new QDialogButtonBox(QDialogButtonBox::Ok |
                QDialogButtonBox::Cancel);

QVBoxLayout* vbox = new QVBoxLayout;
QFormLayout* layout = new QFormLayout;

layout->addRow(tr("Name"), m_name);                          1
layout->addRow(tr("Birthdate"), m_birthday);
layout->addRow(tr("Favorite Color"), m_colorButton);

vbox->addLayout(layout);                                     2
vbox->addWidget(m_buttons);

Q_ASSERT(vbox->parent() == 0);
Q_ASSERT(m_birthday->parent() == 0);
setLayout(vbox);                                             3
Q_ASSERT(vbox->parent() == this);
Q_ASSERT(m_birthday->parent() == this);
```

1 创建/添加一个 QLabel 并在这一行中创建/添加一个输入窗件。

2 用来说明如何在一个布局中嵌套另一个布局。

3 对之前布局过的各窗件重新设置父对象。

可以用 QDialogButtonBox 来显示一些面向用户的标准按钮，这样可以确保它们每次都能够以同样的顺序显示出来。这样做，还有可能让这些对话框可以在不同的平台上依据风格和屏幕尺寸而显示在不同的地方。

没有必要设置窗体中不同窗件的父对象，它们都会添加到以 vbox 为根而构成的部件树的不同布局中。对 setLayout(vbox) 的调用会把 vbox 的父对象设置为宿主的 InputForm 对象上。它还会把添加到 vbox 不同子布局中的 QWidget 的父对象重定义到 vbox 的父对象(同样是宿主的 InputForm)。上面的和下面的 Q_ASSERT 声明都证实了这一点。

9.5 图标，图像和资源

使用图形化的图片可以给应用程序添加可视化效果，这一节将讲解如何在工程中构建一个含有图形化图片的应用程序或者库。

Qt 允许工程使用二进制资源，比如图像、声音、图标、某些外来字体的文字，等等。这些资源通常存储在独立的二进制文件中。在实际工程中集成二进制文件的好处在于它们可以使用不依赖于本地文件系统的路径寻址，并且可以随可执行文件进行自动部署。

Qt 提供至少两种方式来获得标准的图标。一种方式来自桌面样式的 QStyle::standardIcon()，另一种则来自插件型图标主题：QIcon::fromTheme()。但是，你或许还希望使用一些来自其他来源的额外图标。

接下来的一个例子给出了如何创建和复用一个包含图像的库——每一个都代表了一副扑克牌中的一张牌。

第一步是在一个资源集合文件中列出希望使用的二进制文件，该资源文件是一个以.qrc为后缀的 XML 文件。libcard2d 中使用的资源文件的片段如下。

```
<!DOCTYPE RCC>
<RCC version="1.0"><qresource>
<file alias="images/qh.png">images/qh.png</file>
<file alias="images/qd.png">images/qd.png</file>
<file alias="images/jc.png">images/jc.png</file>
<file alias="images/js.png">images/js.png</file>
[...]
</qresource>
</RCC>
```


提示

诺基亚基于 Qt 的开源集成开发环境 QtCreator，是一个 qrc 文件编辑器，如图 9.11 所示。可以创建一些新的(或者编辑已存在的)资源文件，而且通过 GUI 窗体和文件选择器来添加和修改资源，可以确保它包含那些希望的资源。假设在.pro 文件中添加了.qrc 文件，并且正好在使用 QtCreator，那么对任何可被设置资源的属性，都可以通过属性编辑器的省略号按钮来访问这些资源。

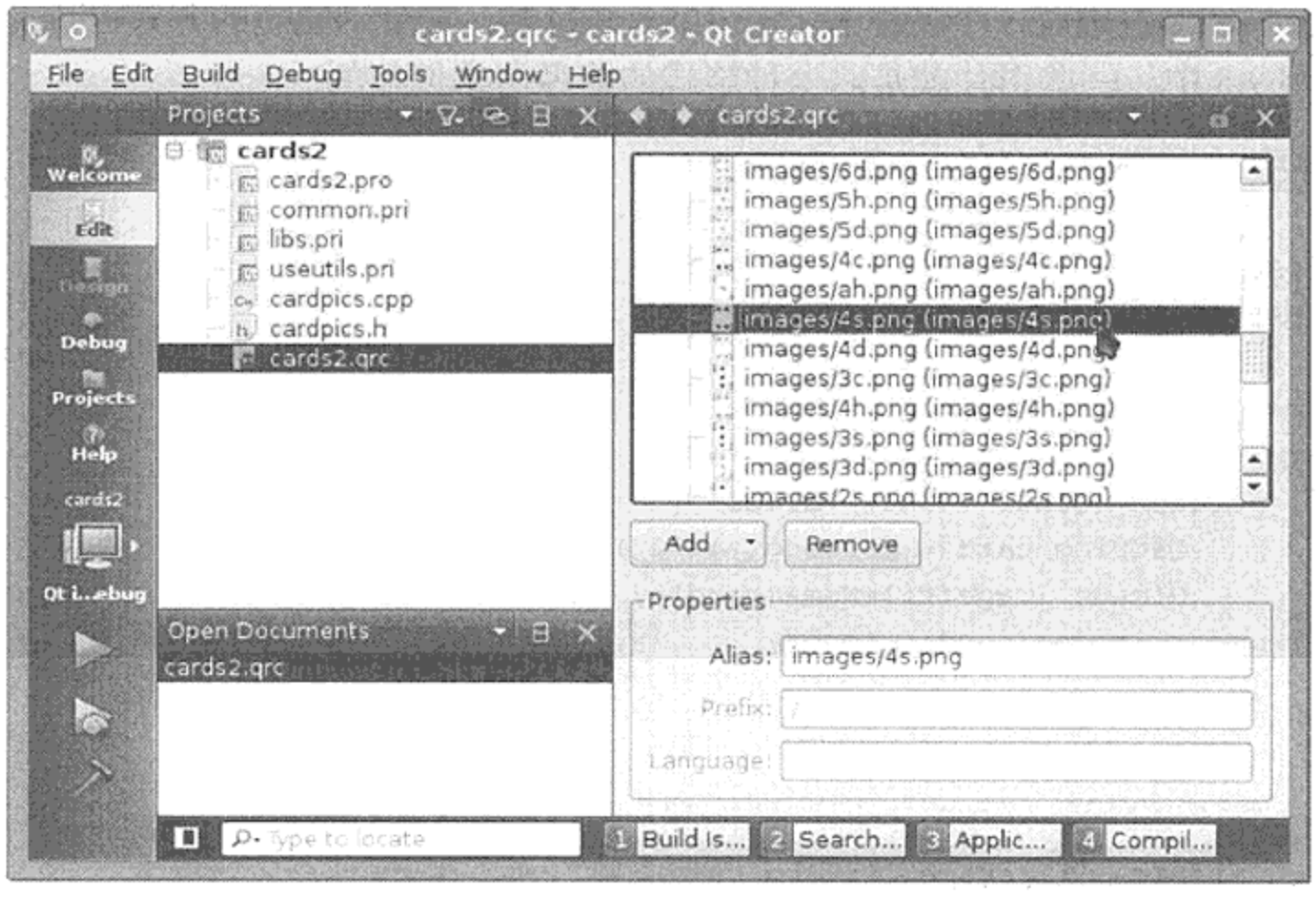


图 9.11　QtCreator 的属性编辑器

在每个含有 qresource 文件名称的工程文件中添加 RESOURCES 行，如示例 9.4 所示。

示例 9.4　src/libs/cards2/cards2.pro

```
include (../libs.pri)

TEMPLATE = lib
QT += gui

# For locating the files.
RESOURCES = cards2.qrc
SOURCES += cardpics.cpp \
    card.cpp
```

```
HEADERS += cardpics.h \
    card.h \
    cards_export.h

win32 {
        DEFINES += CARDS_DLL
}
```


在构建该工程时，rcc 会额外生成一个名称为 cards2_qrc.cpp 的文件，它含有 C++中定义的字节数组。该文件而不是原始的 card 图像文件会被编译并链接到工程的二进制文件中(可执行文件或者库文件)。DESTDIR 这一行指定了用于 libcards2 的共享对象文件将会与已构建过的其他库一起位于$CPPLIBS 中。

把所需的二进制数据文件作为资源附加到工程中会让工程更为健壮。源代码无须为资源文件使用一些不可移植的路径名。要引用一个存储成资源的文件，可以使用在.rcc 文件中指定的别名并在前面带一个前缀“:/”。于是，每个资源都好像位于一个私有虚文件系统中，其根为“:/”。然而，得到这些好处的确还是需要付出一些代价的。可执行文件会更大，程序也会需要更多的内存。

在 libcards2 中有一个名称为 CardPics 的类，它会以便捷的方式提供 QImage 型 card。示例 9.5 中，一些图片就是以这种格式的路径名来创建的。

示例 9.5 src/libs/cards2/cardpics.cpp

```
[ . . . . ]
const QString CardPics::values="23456789tjqka";
const QString CardPics::suits="cdhs";

CardPics::CardPics(QObject* parent) : QObject(parent) {
    foreach (QChar suit, suits) {
        foreach (QChar value, values) {
            QString card = QString("%1%2").arg(value).arg(suit);
            QImage image(fileName(card));
            m_images[card]= image;
        }
    }
}

QString CardPics::fileName(QString card) {
    return QString(":/images/%1.png").arg(card);                    1
}
QImage CardPics::get(QString card) const {
   return m_images.value(card.toLower(), QImage());
}
[ . . . . ]
```

1 来自资源。

有三个 Qt 类可以简化处理图片。

- QImage——用于离屏(off-screen)操作，输入输出操作，并可直接访问像素。
- QPixmap——用于在屏幕上进行绘制并优化。仅用在主线程中。
- QIcon——用于视频内存的缓冲且经常用到，但仅用在主线程中。
- QPicture——存储绘制的操作而不是实际的位图图片。

在 libcards2 中，会先从资源中创建每个 QImage 并添加到 CardPics 中，以便可以用 get() 函数进行快速访问。

9.6　窗件的布局

窗件可以像对话框一样弹出并显示在屏幕上，也可以作为一个较大窗口的一部分。无论何时，只要打算在较大的窗件中排放一些较小的窗件，就必然要用到布局。所谓布局，就是一个仅属于某一窗件的对象(即，是其子对象)。布局的唯一任务就是合理地组织其拥有的子窗件所占据的空间。

尽管每个窗件都有一个 setGeometry() 函数，也可以通过此函数来设置其大小和位置，但是在窗口应用程序中很少采用绝对大小和绝对位置，其原因就是这种方式通常会使得设计过于死板。通过使用布局，可以灵活自然地安排所有的可见空间，从而使得窗口能够成比例地改变大小、拖动或者为窗口加入滚动条。

在屏幕上安排窗件位置和顺序的主要过程，就是将屏幕空间合理地划分成几个区域，并使用 QLayout 来管理每个区域。布局可以将它们的窗件排列成如下几种形式。

- 垂直型(QVBoxLayout)
- 水平型(QHBoxLayout)
- 网格型(QGridLayout)
- 窗体型(QFormLayout)
- 栈型，任何时候都只有一个窗件可见(QStackedLayout)

可以使用 addWidget() 函数向 QLayout 添加窗件。当窗件添加到布局中时，它会成为拥有该布局的窗件的子对象。窗件永远不会成为布局的子对象。

布局不是窗件，它们也没有任何可见的表达形式。Qt 提供了一个名称为 QLayout 的抽象基类，还提供了几个特殊的 QLayout 子类：QBoxLayout(可以细分为 QHBoxLayout 和 QVBoxLayout)、QGridLayout 和 QStackedLayout。这些布局类型的每一种都拥有一组合适的函数来控制空间、大小、对齐方法以及对其窗件的访问方式。

为了能够顺利地管理其几何形状，每个 QLayout 对象都必须有一个父对象，这可以是一个 Qwidget，也可以是一个 Qlayout。可以在创建布局时通过向构造函数传递一个指向父窗件或者布局的指针来指定其父对象。当然，也可以先创建一个 QLayout 而不指定 QLayout 的父对象，这种情况下，可以稍后通过调用 QWidget ::addLayout() 来指定其父对象。

布局可以拥有子布局。通过调用 addLayout() 函数，可以将一个布局添加为另外一个布局的子布局。当然，具体的做法取决于所采用的布局类型。如果布局的父对象是一个窗件，那么该窗件将再也无法成为另外一个布局的父对象。

示例 9.6 中定义的 CardTable 类复用了 libcards2，以此来更为轻松地访问扑克牌(参见 9.5 节)的 QImage。构建 CardTable 对象之后，屏幕显示如图 9.12 所示。

示例 9.6　src/layouts/boxes/cardtable.h

```
#ifndef CARDTABLE_H
#define CARDTABLE_H
#include <cardpics.h>
#include <QWidget>
```

```
class CardTable : public QWidget {
  public:
    explicit CardTable(QWidget* parent=0);
  private:
    CardPics m_deck;
};

#endif          //  #ifndef CARDTABLE_H
```

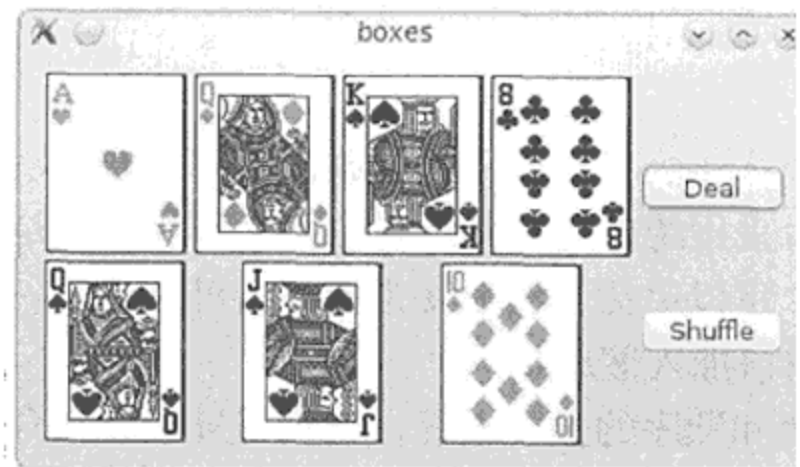


图 9.12 显示几行和几列扑克牌

示例 9.7 中实现的 CardTable 充分地利用了 QLabel 可以存放图像的事实，这个实现展示了一些简单却非常有用的布局技术。

示例 9.7 src/layouts/boxes/cardtable.cpp

```
[ . . . . ]

CardTable::CardTable(QWidget* parent)
: QWidget(parent) {

    QHBoxLayout* row = new QHBoxLayout();                    1
    row->addWidget(new Card("ah"));                          2
    row->addWidget(new Card("qd"));
    row->addWidget(new Card("ks"));
    row->addWidget(new Card("8c"));

    QVBoxLayout* rows = new QVBoxLayout();                   3
    rows->addLayout(row);                                    4
    row = new QHBoxLayout();                                 5
    row->addWidget(new Card("qs"));
    row->addWidget(new Card("js"));
    row->addWidget(new Card("td"));
    rows->addLayout(row);                                    6

    QVBoxLayout* buttons = new QVBoxLayout();                7
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));
    QHBoxLayout* cols = new QHBoxLayout();                   8
    setLayout(cols);                                         9
    cols->addLayout(rows);                                   10
    cols->addLayout(buttons);                                11
}
[ . . . . ]
```

1 第一行。
2 父对象会通过布局得到设置，因此不必指定。
3 垂直地布局各行。
4 在垂直布局中嵌入一行。
5 第二行。
6 再次嵌套。
7 用于各个按钮的一列。
8 把它们全都放在一起。
9 这个窗件的“根布局”。
10 把这两行扑克牌都加为一列。
11 把一列按钮都加为另一列。

示例 9.8 中给出的客户代码足以在屏幕中显示该窗口。

示例 9.8　src/layouts/boxes/boxes.cpp

```
#include <QApplication>
#include "cardtable.h"

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

如果构建并运行这个例子，然后用鼠标改变窗口的大小，可以注意到，按钮的宽度会首先进行伸展来占用那些多出来的空间。但是，在各个纸牌之间和各个按钮之间也有可供伸展的空间。如果将按钮移除，就可以观察到纸牌之间的水平空间将会均匀地增长。

9.6.1　分隔，伸展和支撑

不使用 Qt 设计师时，可以使用 QLayout 类的 API 来直接指定各个窗件之间的分隔(spacer)、伸展(stretch)和支撑(strut)。

- addSpacing(int size)会向布局的末尾添加固定数量的像素。
- addStretch(int stretch = 0)会添加数目不定的像素。此函数由一个最小的数目开始，然后逐渐扩展到使用所有的可用空间。如果在同一个布局中进行多次扩展，可以用此作为一个增长因子。
- addStrut(int size)将给垂直方向施加一个最小的数值(也就是，QVBoxLayout 的宽度或者 QHBoxLayout 的高度)。

回到示例 9.7 中，我们将试图使此布局在改变大小时表现得更加自然。图 9.13 给出了在该应用程序中添加一定的伸展量和分隔量后的结果。

通常情况下，布局会试图平等对待所有的窗件。当打算让一个窗件与另一个窗件不相邻或者相互远离时，可以使用伸展和分隔功能来与常规效果加以区分。示例 9.9 给出了如何使用伸展和分隔的做法。

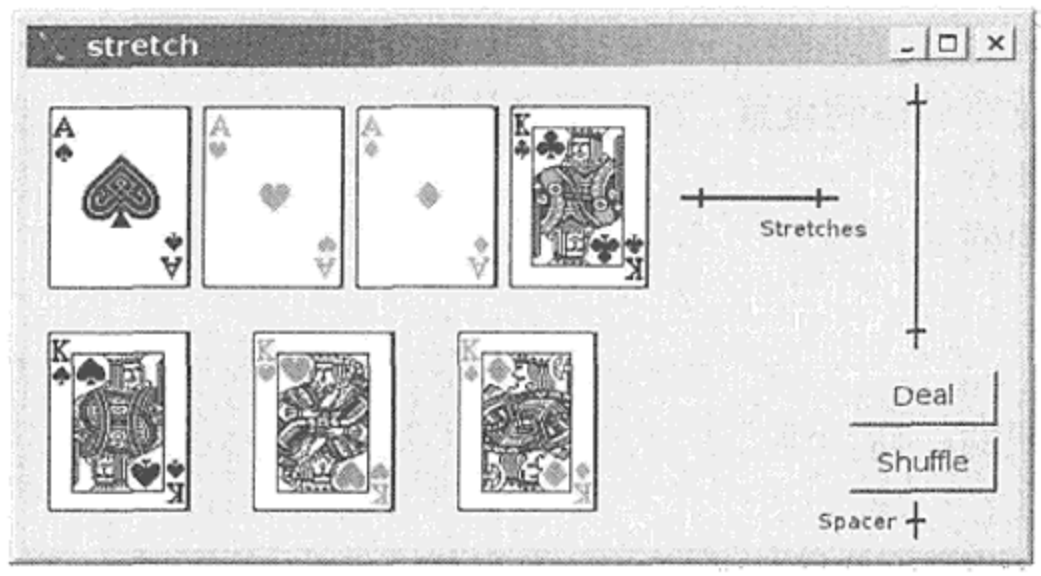


图 9.13 改进的带有伸展量和分隔量的布局

示例 9.9 src/layouts/stretch/cardtable.cpp

```
[ . . . . ]
    row = new QHBoxLayout();
    row->addWidget(new Card("td"));
    row->addWidget(new Card("js"));
    row->addWidget(new Card("kc"));
    rows->addLayout(row);
    rows->addStretch(1);                                          1
    QVBoxLayout* buttons = new QVBoxLayout();
    buttons->addStretch(1);                                       2
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));
    buttons->addSpacing(20);                                      3
    QHBoxLayout* cols = new QHBoxLayout();
    setLayout(cols);
    cols->addLayout(rows);
    cols->addLayout(buttons);
    cols->addStretch(0);                                          4
}
[ . . . . ]
```

1 用于各行的可伸展空间。

2 在列中各按钮之前的可扩展空间。

3 按钮之后的固定空间。

4 这会如何影响各按钮的尺寸呢?

如果使用示例 9.9 而不是示例 9.7 中的代码来构建和运行此应用程序,然后再次调整主窗口的大小,就可以注意到,按钮不会再增长了,而是被推到了一个角上。扑克牌之间的水平空间不再增长,但是其垂直空间仍会增加。

9.6.2 大小策略和大小提示

每个 QWidget 都有一些与屏幕尺寸相关的属性。一些窗件可以在运行时使用一个或多个方向上的额外空间。通过设置自定义窗件的 sizePolicy 值和 sizeHint 值,可以控制其默认的大小变化行为。

sizeHint 会为窗件保存推荐的 QSize 值。也就是说,该值是其第一次出现在屏幕上时的给定尺寸。QSize 定义了窗件的宽度和高度大小。还有一些成员函数可以用来完全控

制自定义窗件在运行时的大小，包括 setMinimumSize()，setMaximumSize()，setMinimumHeight()，setMaximumWidth()，setSizeHint()等。

每个 Qwidget 和 QLayout 都有一个水平方向和垂直方向的 QsizePolicy。Minimum，Maximum，Fixed，Preferred，Expanding，MinimumExpanding 和 Ignore 这些大小策略表达了窗件或者布局大小被改变时自身的意愿。默认的大小策略是 Preferred/Preferred，这表示 sizeHint 会让窗件拥有较好的大小。Ignore 表示 minimumHeight 和 minimumWidth 不会对改变尺寸的各种操作进行约束。

以固定尺寸显示的窗件将不会从可改变大小的窗件中获得任何多余的空间。因此，应当使用 Fixed 或 Preferred 型水平和垂直尺寸策略。比较而言，可滚动的窗件、容器型窗件以及文本编辑型窗件应当使用扩展型尺寸策略，以便可以向用户显示出更多的信息，因为这可能也是用户之所以使窗口变大的首要原因。多余的空间可由使用柔性尺寸策略的窗件给定或获得。

在同一布局中使用扩展型策略的窗件，根据所基于的伸展因子的不同，会以不同的比率来给定多出来的空间。

从图 9.14 可以看出，按钮通常不会占用任何方向上的多余空间。在开始的两行上，按钮使用了水平扩展尺寸策略，随后的一行显示了各个拉伸因子是如何影响可扩展型按钮的。第三行中，可以看到按钮会占用垂直方向上的全部空间并会逼迫起初的两行变得尽可能地高。

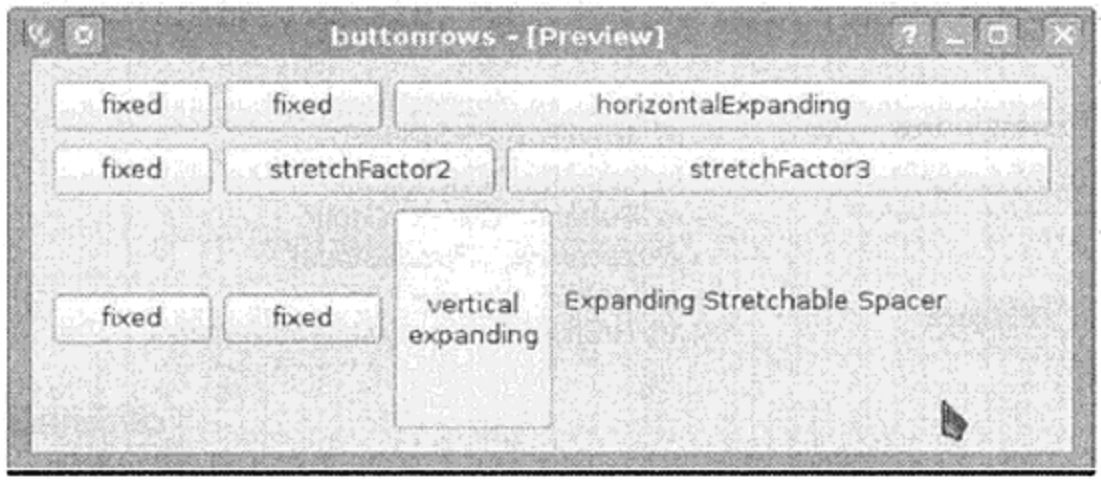


图 9.14　拉伸了的按钮

9.6.3　练习：窗件的布局

1. 在 Qt 设计师中，试着重做排放按钮的布局，使其能够拥有如图 9.14 所示的形式。
2. 在 15 拼图(或 $n^2 - 1$)游戏中含有一个 4 × 4(或 $n \times n$)的网格，其中含有 15 个从 1 到 15 编号的麻将牌和一个空白空间。只有与该空白空间相邻的牌才可以移动。

- 在 QGridLayout 中用 QPushButton 创建一个 15 拼图游戏，如图 9.15 所示。
- 在游戏开始时，各个牌会向玩家以“随机的”顺序显示出来。游戏的目的就是重新排列它们，以便可以让它们显示成升序的形式，使最小数字的牌位于左上角。
- 如果玩家成功完成了这个拼图，则弹出一个 QMessageBox，显示“获胜!”(或是其他一些更聪明的东西)。
- 添加一些按钮：
  - Shuffle（洗牌）——通过执行一个更大的合法的牌数(最大 50)来使牌随机排列。
  - Quit（退出）——终止游戏。

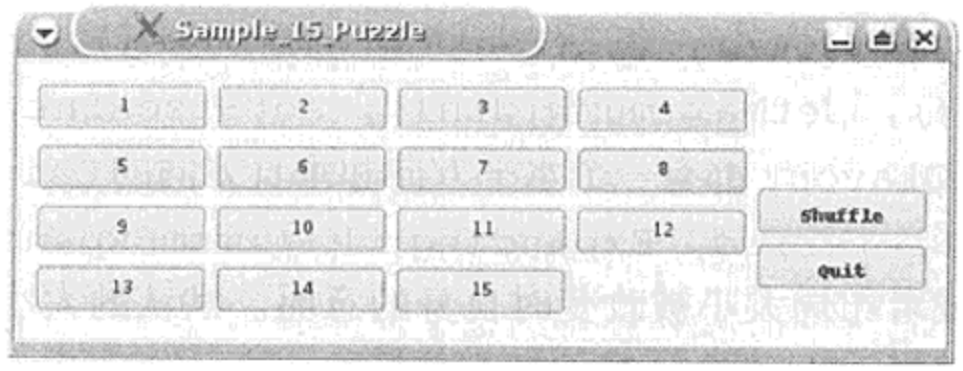


图 9.15　15 拼图的示例

提示

为了避免对每个按钮都使用信号和槽相连接，可以把各按钮群组织到一个 QButtonGroup 中，它有一个 QAbstractButton 的单参数信号。

提示

以图 9.16 所示的类开始，试着将自己的代码适当地整合到其中。“视图”类会处理 GUI，“模型”类则应当没有任何的 QWidget 代码或代码依赖，并且仅用于处理本游戏的逻辑。

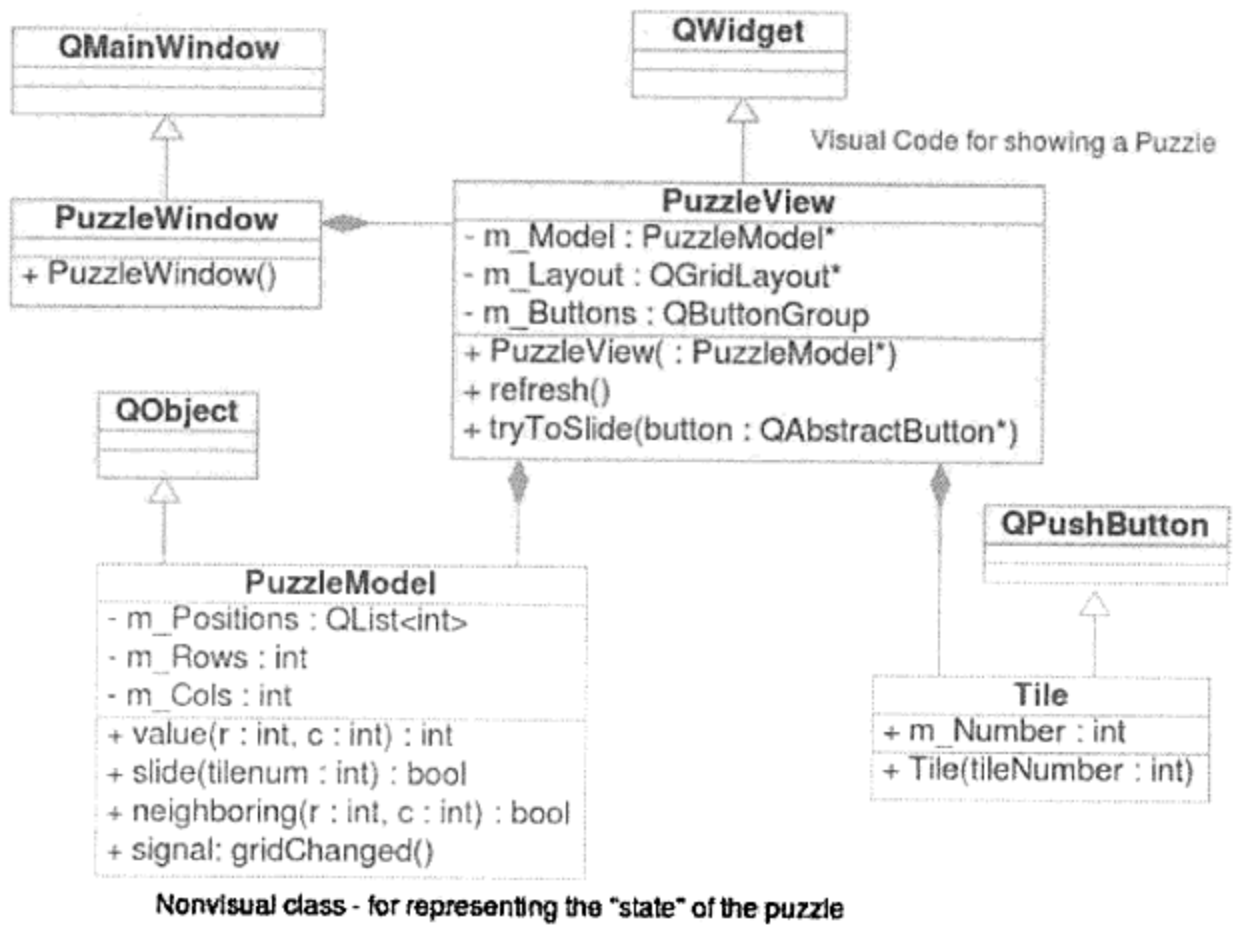


图 9.16　用于拼图的“模型—视图—控制器”设计模式

9.7　设计师和代码的集成

考虑如图 9.17 所示的 ProductForm，它是一个用于 Product 示例的窗体。QFormLayout 是一个便利的窗件，可方便地将 QLabel 和输入窗件组织成两列。

ProductForm 窗件可以接受一个新 Product 对象中来自用户的数据，可用来显示(只读)或者编辑已保存的 Product 实例中的值。根据使用模式，按钮可以有文字和角色。例如，当对用户新增一个 Product 响应时，单击 OK 按钮后应当用窗体中的数据创建一个新 Product 实例。如果单击的是 Cancel 按钮，则应当丢弃这些值。在编辑模式下单击 OK 按钮

时，窗体中的这些值应当去替换那些已保存实例的值；在消息(info)模式下单击 OK 按钮时，窗体应当消失，而 Cancel 按钮也应当隐藏起来。

使用它时，ProductForm 会有一个指向它所编辑的 Product 的引用，如图 9.18 所示。

图 9.17　Product 窗体

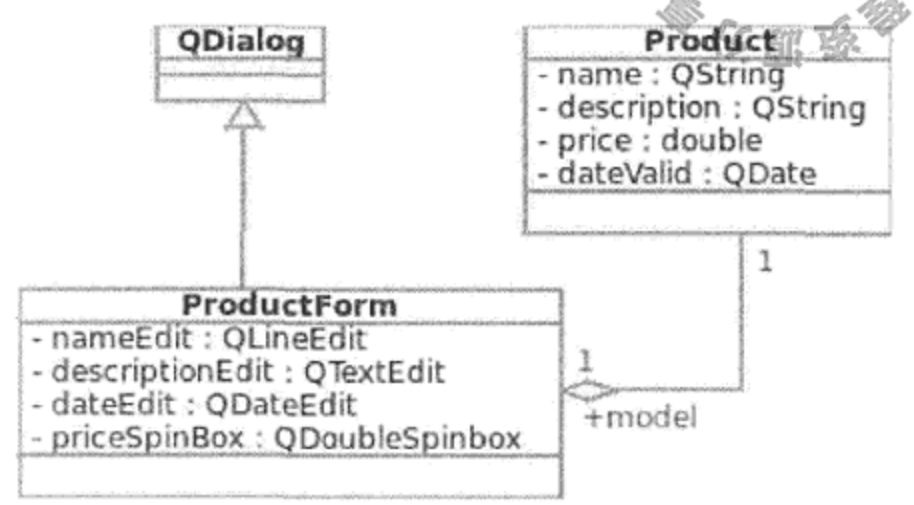


图 9.18　Product 及其窗体

使用设计师设计窗体

跟往常一样，第一步是选择一个基类名称(例如，ProductForm)并为其写出一个头文件(ProductForm.h)和一个实现文件(ProductForm.cpp)。然后，在设计师中用同样的基类名称(ProductForm.ui)创建一个窗体。设置该窗体根对象的 objectName 的名称(ProductForm)，然后启用 uic 为相应的 Ui 类(Ui_ProductForm)生成头文件。

图 9.19 给出了 uic 如何将通过设计师产生的带 .ui 扩展名的特殊 XML 文件当作输入，并生成可以包含进自己代码中的头文件。

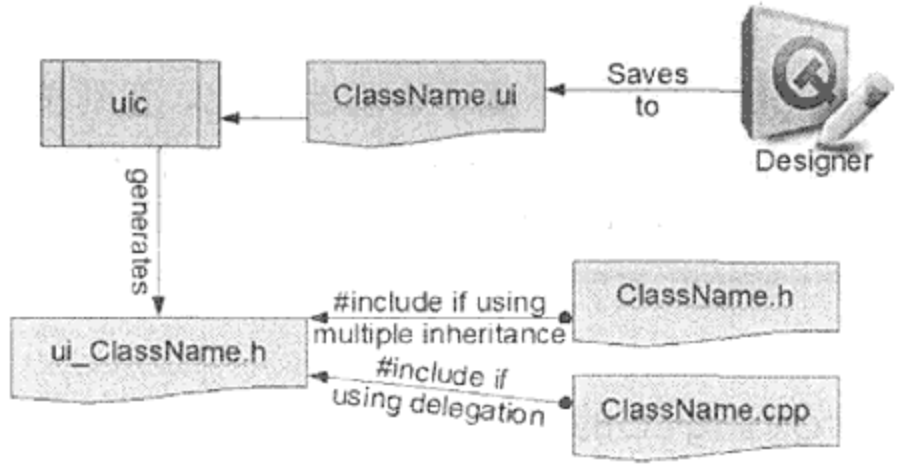


图 9.19　从设计师到代码

示例 9.10 给出了生成的一小段代码片段，它既定义了一个 Ui_ProductForm 类又定义了一个 Ui::ProductForm 派生类。在 C++代码中使用哪一个类由用户决定。正如所看到的，有数据成员会指向设计师窗体中的每一个窗件。每个成员的名称都来自设计师窗体中所设定的对象名，在生成的代码中，赋予了成员名的完全控制权。

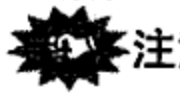
示例 9.10　src/designer/productform/ui_ProductForm.h

```
[ . . . . ]
class Ui_ProductForm
{
public:
    QVBoxLayout *verticalLayout;
    QFormLayout *formLayout;
```

```
    QLabel *label;
    QDoubleSpinBox *priceSpinbox;
    QLabel *label_2;
    QLabel *label_3;
    QLineEdit *nameLineEdit;
    QLabel *label_4;
    QTextEdit *descriptionEdit;
    QDateEdit *dateEdit;
    QSpacerItem *verticalSpacer;
    QDialogButtonBox *buttonBox;

    void setupUi(QDialog *ProductForm)
[ . . . . ]
};
namespace Ui {
    class ProductForm: public Ui_ProductForm {};
} // namespace Ui
[ . . . . ]
```

什么是 Ui 类?

Ui 类是一个类，仅含有由 uic 工具自动生成的代码。



注意

设计师一开始会把一些来自 QdialogButtonBox 的信号与 QDialog 的 accept()和 reject()槽相连接。从可停靠的信号和槽编辑器中可以看到这一点。这也就意味着，在 OK 按钮按下之前，后台的 Product 对象将一直不会发生变化，直到按下 OK 按钮——可能正如所期望的——reject()的基类行为会关闭对话框。你也可能会希望覆盖基类的 accept()，但不必把按钮的信号与之连接。基类版也会关闭对话框。

集成的方法

以下是将 Ui 类和基于 QWidget 的自定义窗体类相集成的三种方法。

1. 作为指针成员集成。
2. 多重(私有)继承。
3. 作为嵌入式对象集成。

推荐使用通过指针的集成方法(也是默认的)，因为这样可使修改 Ui 文件而不造成带有 ProductForm 头文件的二元破坏(binary breakage)成为可能。示例 9.11 给出了通过指针成员进行集成的示例。ProductForm.h 使用了一个前置类声明而没有直接包含 Ui_Product-Form.h。

示例 9.11 src/designer/delegation/productform.h

```
[ . . . . ]
#include <QDialog>
class Product;
class Ui_ProductForm;
```

```
class QWidget;
class QAbstractButton;
class ProductForm : public QDialog {
        Q_OBJECT
public:
    explicit ProductForm(Product* product = 0, QWidget* parent=0); 1
    void setModel(Product* model);

public slots:
    void accept();
    void commit();
    void update();

private:
    Ui_ProductForm *m_ui;
    Product* m_model;
};
[ . . . . ]
```

1 明确标示以避免在各指针间出现隐式转换!

只有其实现文件，如示例 9.12 所示，需依赖于 uic 生成的头文件。例如，这样可使在把其放入库中时变得更为简单。

示例 9.12　src/designer/delegation/productform.cpp

```
#include <QtGui>
#include "productform.h"
#include "ui_ProductForm.h"
#include "product.h"

ProductForm::ProductForm(Product* product, QWidget* parent)
: QDialog(parent), m_ui(new Ui::ProductForm),  m_model(product) {
    m_ui->setupUi(this);                                          1
    update();
}

void ProductForm::setModel(Product* p) {
    m_model =p;
}

void ProductForm::accept() {
    commit();
    QDialog::accept();                                            2
}

void ProductForm::commit() {
    if (m_model == 0) return;
    qDebug() << "commit()";
    m_model->setName(m_ui->nameLineEdit->text());
    QTextDocument* doc = m_ui->descriptionEdit->document();
    m_model->setDescription(doc->toPlainText());
    m_model->setDateAdded(m_ui->dateEdit->date());
    m_model->setPrice(m_ui->priceSpinbox->value());
```

```
}

void ProductForm::update() {
    if (m_model ==0) return;
    qDebug() << "update()";
    m_ui->nameLineEdit->setText(m_model->name());
    m_ui->priceSpinbox->setValue(m_model->price());
    m_ui->dateEdit->setDate(m_model->dateAdded());
    m_ui->descriptionEdit->setText(m_model->description());
}
```

1 让 Ui 对象和来自 .ui 文件中设置的适当值的实例一起工作。

2 关闭对话框。

9.7.1 QtCreator 和设计师的集成

QtCreator 用一种简便的方式将集成开发环境和设计师集成一体，在它能理清 Ui 文件和类文件对应关系的地方，会为这些类生成集成代码和各个槽的函数体。通过菜单 QtCreator File->New->Qt->Designer Form Class，可以一次创建所有新的 .ui 文件、头文件和相应的类文件。图 9.20 给出了它所支持的三种代码生成方式。

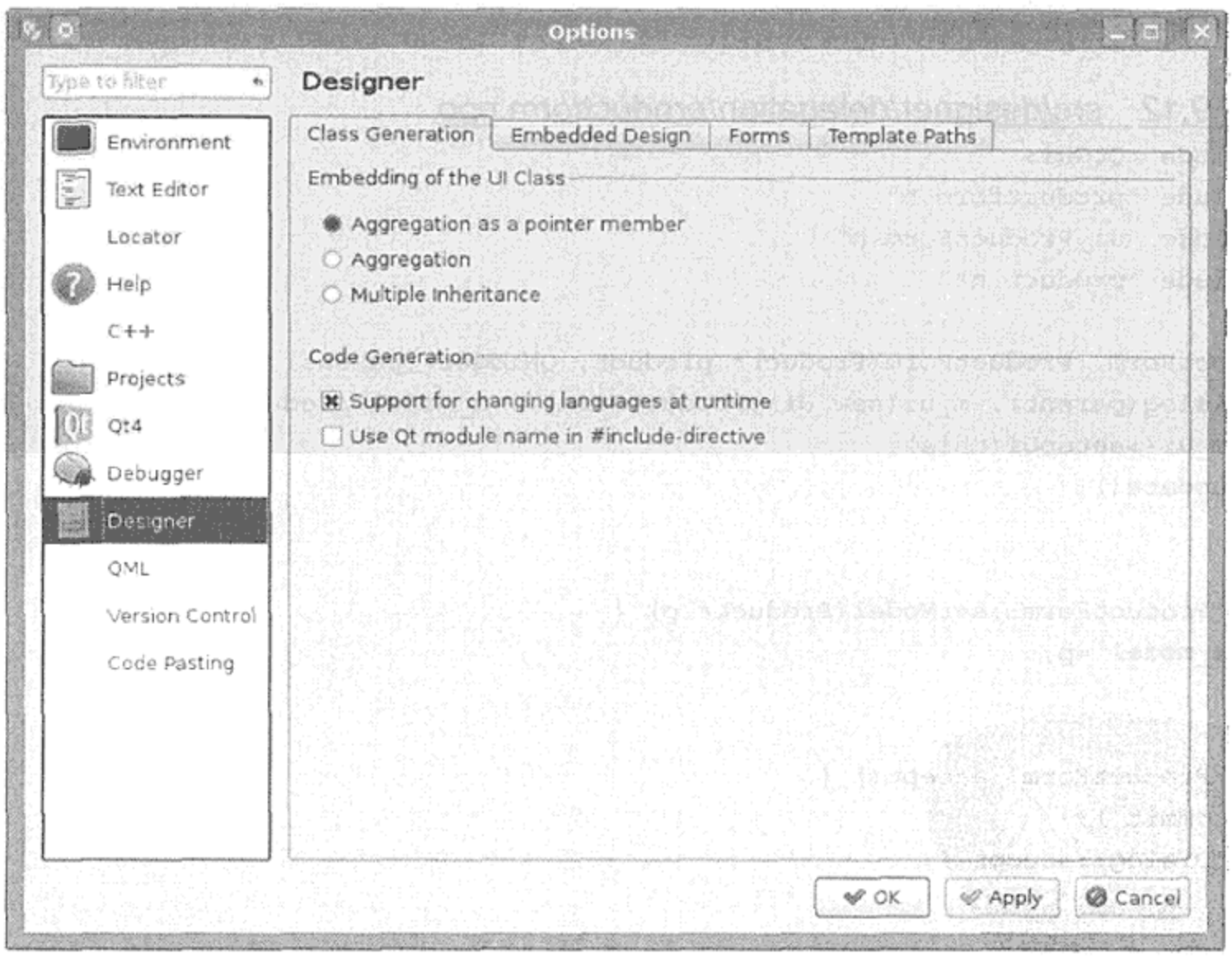


图 9.20 QtCreator 中设计师的代码生成选项

对于需要相互连接的各个子对象(例如，QMainWindow 的各个 GUI 组件)，QtCreator 提供了一种非常方便的特性，如图 9.21 所示。可以在 Widget 编辑器中发射信号的任意组件(例如，Save 按钮)上或者在 Action 编辑器中的任意动作(例如，actionSave)上右键单击，并选择 Go to slot 即可。

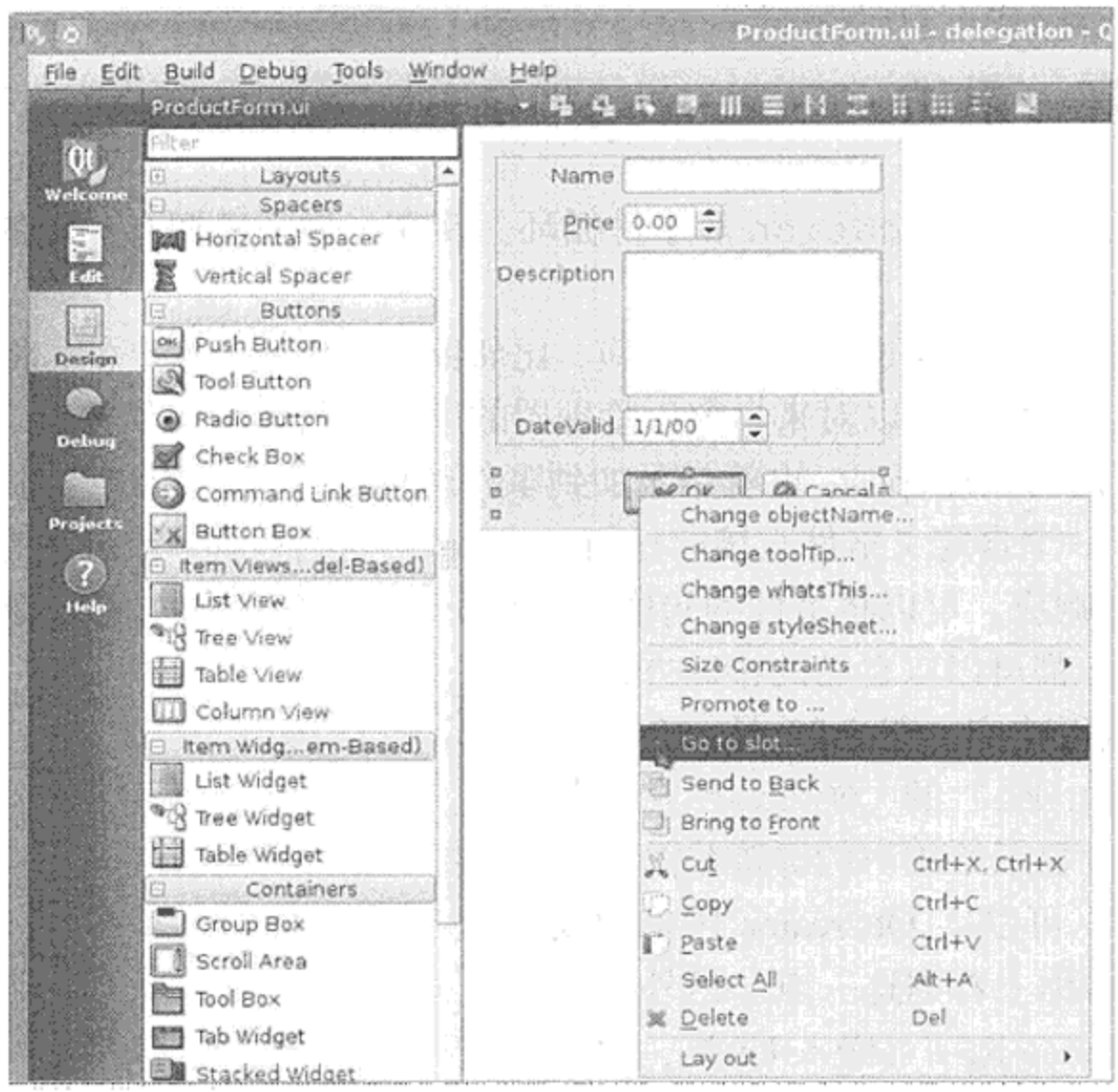


图 9.21　QtCreator 中的 Go to Slot 选项

接下来，从弹出的清单中选择一个信号，例如 `triggered()`，QtCreator 会生成一个私有槽——头文件中的一个原型和实现文件中的一个代码片段——如果还没有一个带有生成名称的槽，例如 `on_actionOpen_triggered()`。通过添加必要的操作细节，即可完成该新槽的定义。所选择的信号在运行时会自动连接到刚定义的槽上。在实现文件(`.cpp`)中，将无法找到连接语句。相反，这些连接是由 `QMetaObject::connectSlotsByName()` 函数完成的。

10.5 节讨论了一个在 QtCreator 中完全使用这些工具编写的应用程序。

9.8　练习：输入窗体

1. 实现 `OrderForm` 类中为创建 `Order` 对象输入对话框窗体所缺少的方法。在 `src/handouts/forms/manual` 中给出了一些开始的东西。对话框看起来会与图 9.22 类似。
   代码放在 `orderform.cpp` 中。用户应当单击 OK 按钮来给 `Order` 发送数据，或者单击 Cancel 按钮来取消该对话框并且不给 `Order` 发送任何数据。
   `totalPrice` 域应当是只读的，且计算值要基于 `quantity` 和 `unitPrice` 的值。
2. 接下来，用设计师创建同样的输入对话框，并使用委托或者多重继承的方式将其与你的代码相集成，如 9.7 节所述。

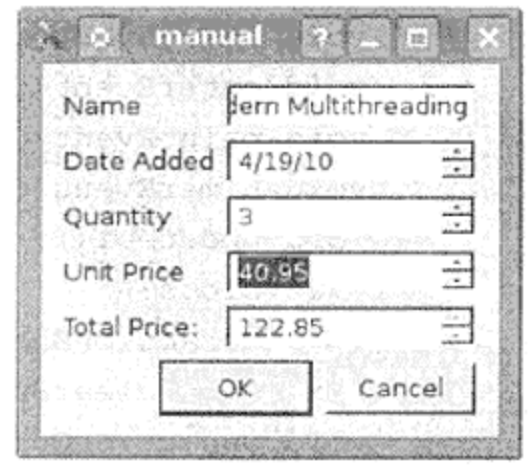


图 9.22　Order 窗体对话框

9.9 事件循环：重访

8.3 节中讲解过 QApplication 和事件循环。现在，你已经拥有一定的 GUI 编程经验，可以更深入地探讨事件了。

QWidget 根据用户产生的动作进行响应，比如鼠标事件、键盘事件，将 QEvent 发送给其他的 QObject。窗件还可以对来自窗口管理器的事件进行响应，如重绘、更改大小或者关闭事件等。事件可进行投递(post，也就是添加到事件队列中)，也可以进行过滤和排列优先级。此外，也有可能将一个自定义事件直接向任何 QObject 进行投递，或者是有选择地对可以投递的其他对象进行响应。可以将事件的句柄委托给一个特殊定义的对象处理程序。

每个 QWidget 都可以以自己的特殊方式对键盘和鼠标事件进行响应。图 9.23 是一个名称为 KeySequenceLabel 的 QWidget 在捕获 QKeySequence 并显示给用户的屏幕截图。

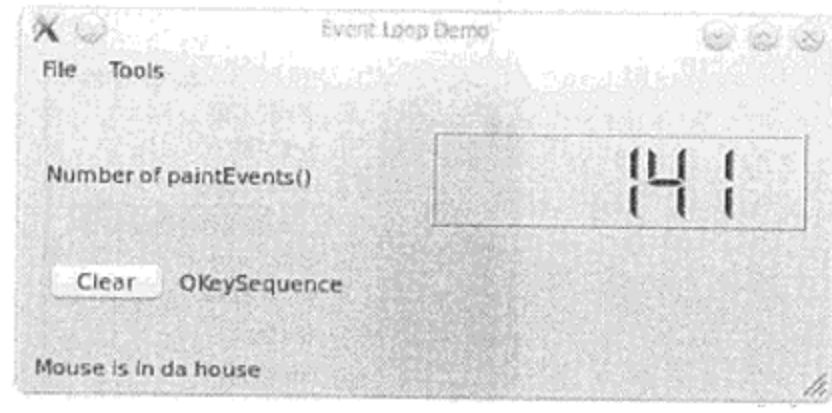


图 9.23 KeySequenceLabel 窗件

示例 9.13 中，可以从 QtCreator 用来构成 GUI 的私有槽的名称上看到这一点。另外，选中 Multiple Inheritance 选项可嵌入 Ui 类，因而 KeySequenceLabel 类会从 QMainWindow 类和设计师生成的 Ui 类中派生出来。

示例 9.13 src/eventloop/keysequencelabel.h

```
[ . . . . ]
#include "ui_keysequencelabel.h"
#include <QList>
#include <QPair>
class QObjectBrowserAction;

class KeySequenceLabel : public QMainWindow, private Ui::KeySequenceLabel {
    Q_OBJECT
 public:
    explicit KeySequenceLabel(QWidget* parent = 0);
 protected:                                                          1
    void changeEvent(QEvent* e);
    void keyPressEvent(QKeyEvent*);
    void leaveEvent(QEvent*);
    void enterEvent(QEvent*);
    void paintEvent(QPaintEvent*);
    void timerEvent(QTimerEvent*);                                   2
    void updateUi();
private slots:
    void on_actionShow_ObjectBrowser_triggered(bool checked);
    void on_m_clearButton_clicked();
    void on_actionQuit_triggered();
private:
    QObjectBrowserAction* m_browserAction;
```

```
    QList<QPair<int, int> > m_keys;
    int m_paints;
};
[ . . . . ]
```


1 重载 QWidget 事件处理程序。

2 重载 QObject 事件处理程序。

在这个类的实现中包含了一些有意思的东西。

在对话框中有一个 QLabel，用来显示 QKeySequence。QKeySequence 类封装了一个序列，它可存储多达四次的键击，一般用作 QAction 调用的快捷方式。在 KeySequenceLabel 窗件中重载了基类的 keyPressEvent()处理程序。示例 9.14 中，可以发现，键击事件在子窗件看到它们之前已被捕捉并保存到一个列表中。在 updateUi()内，这些事件会先转换成一个 QkeySequence，然后再转换成一个 QString，该字符串通过 setText()送到 QLabel 显示。

示例 9.14　src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

void KeySequenceLabel::keyPressEvent(QKeyEvent* evt) {
    bool doNothing = false;

    if (evt->key() == 0) doNothing = true;
    if (m_keys.size() > 3) doNothing = true;
    if (doNothing) {
        QMainWindow::keyPressEvent(evt);                    1
        return;
    }
    QPair<int, int> pair = QPair<int, int>(evt->modifiers(), evt->key());
    m_keys << pair;
    evt->accept();
    updateUi();
}

void KeySequenceLabel::updateUi() {
    if (!m_keys.isEmpty()) {
        int keys[4] = {0,0,0,0};
        for (int i=0; i<m_keys.size(); ++i) {
            QPair<int, int> pair = m_keys[i];
            keys[i] = pair.first | pair.second;
        }
        QKeySequence seq = QKeySequence(keys[0], keys[1], keys[2], keys[3]);
        m_label->setText(seq.toString());
    }
    else m_label->clear();
}
```

1 QWidget 的基类处理程序对弹出窗口的 ESC 响应。

示例 9.15 中定义了一些鼠标事件处理程序，用来说明鼠标指针是何时进入或者离开该窗件的。

示例 9.15 src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

void KeySequenceLabel::enterEvent(QEvent* evt) {

    statusBar()->showMessage(tr("Mouse is in da house"));
    evt->accept();

}

void KeySequenceLabel::leaveEvent(QEvent* evt) {
    statusBar()->showMessage(tr("Mouse has left the building"));
    evt->accept();
}
```

示例 9.16 中，构造函数使用 QObject 内置的定时器并调用了 QObject::startTimer()，它会每 2 s 就产生一个计时事件。可以来这样处理 timerEvent()：每 2 s 使用当前绘制事件被执行的次数来更新一个 QLCDNumber。例如，每当 LCDNumber 的值发生改变时，都会间接造成该窗件的重绘。可以运行一下这个应用程序，试着对窗件进行移动和改变大小，看看它是如何影响绘制事件次数的。

示例 9.16 src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

KeySequenceLabel::KeySequenceLabel(QWidget* parent) :
    QMainWindow(parent), m_browserAction(new QObjectBrowserAction(this)) {

    setupUi(this);
    startTimer(2000);                                                    1
    m_paints = 0;
}

void KeySequenceLabel::timerEvent(QTimerEvent*) {
    m_lcdNumber->display(m_paints);
}

void KeySequenceLabel::paintEvent(QPaintEvent* evt) {
    ++m_paints;
    QMainWindow::paintEvent(evt);
}
```

1 每 2 s 发生一次定时器事件。

到目前为止，所有与 GUI 活动有关的讨论都是关于事件和事件处理程序的。现在要讨论该示例中一个更为有趣的特点。从 Tools 菜单可以选择 Show ObjectBrowser，即可看到如图 9.24 所示的窗件。

QObjectBrowser 是一个开源的代码调试工具，在程序执行过程中，可以对 QObject

的属性、信号和槽进行图形化显示[①]。使用它的一种简单方法是将 QObjectBrowserAction 直接添加到 QMainWindow 的菜单上。

在 src/libs 目录中，已含有一个对该工具进行简单修改的版本。

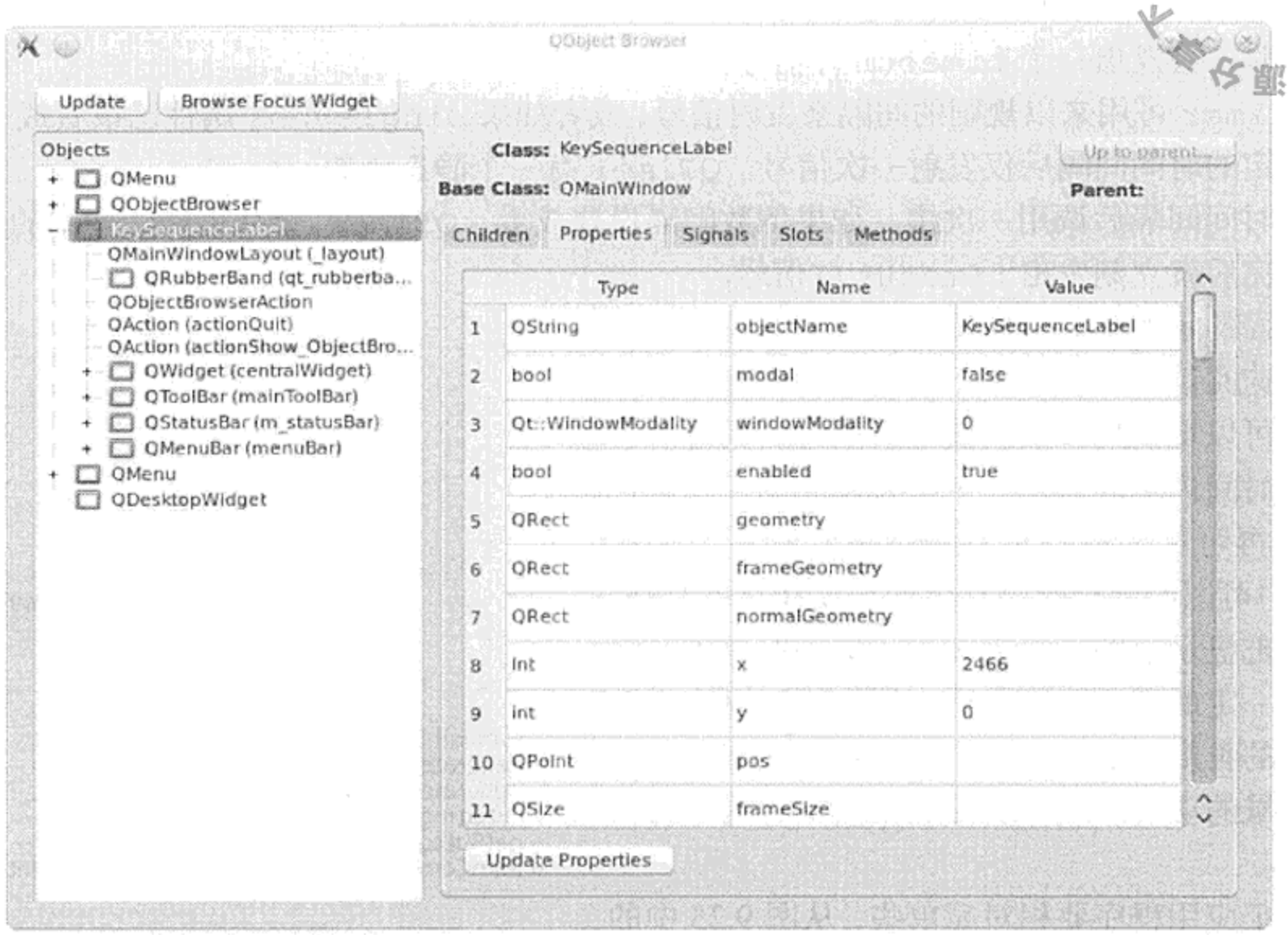


图 9.24　QObjectBrowser 窗件

自定义事件与信号和槽的比较

一般情况下，从语义学的角度看，可以将信号看成是来自对象的更为高级的消息，事件则可以认为是来自用户或系统的更为底层的消息。这两者的机理都体现了 Publish/Subscribe（发布/订阅）模式。

定义一些自定义的事件或者信号和槽还是可能的。利用自定义事件，可以用 QObject::postEvent()对任意的 QObject 进行发布，并可以用 installEventFilter()订阅任意 QObject 的有意向的事件。

绝大多数情况下，如果可以选择，信号—槽连接推荐用于自定义的 QEvent，因为信号和槽要更为安全一些、高级一些，且在消息传递方面更具柔性机制。信号和槽更具柔性机制，是因为发射状态不需要目的对象，而信号可用于多个目标槽中。

信号和槽也需要依靠事件循环，如果打算将线程中对象的信号连接到另一个线程中多个对象的槽上，就会立即注意到这一点。信号可以（以阻塞或者非阻塞的方式）传递给同一个线程或（自身拥有事件循环的）另一个线程中的槽。

① 由 Magland 开发。从其最初发布的地方可以跟踪这个工具开发过程，也可以在 http://tinyurl.com/2a6qkpy 中看到社区的一些建议和作者的回复。

9.9.1 QTimer

上一节中给出了一个使用 QObject 内置定时器的例子。Qt 还有一个 QTimer 类，它为定时器提供一种高级接口。QTimer 对象是一个倒数计时器，以毫秒级时间间隔启动。当其到达零时，会发出一个 timeout() 信号。

QTimer 可用来以规则的间隔来发射信号，或者如果 singleShot 属性已设置成 true，则在给定的时间间隔后仅发射一次信号。QTimer 有一个静态函数 singleShot()，可以在给定的时间间隔后调用一次槽。如果倒数间隔设置成零，QTimer 会在事件队列中的全部事件处理完后就立刻发出 timeout() 信号。

下面的示例是一个使用 QTimer 编写的训练用户快速阅读的应用程序。该程序的设计者们还声称它可以增加用户的阅读理解能力。其思想就是一段时间内简单显示一个字符串序列，让用户在其不再可见之前尽可能快地输入每一个字符串。用户可以明确给定字符串的长度和显示时间。程序会把显示过的字符串和敲入的字符串进行对比并以一定的形式记下得分。用户可以逐步增加字符串的长度并缩短字符串的显示时间来增强对这一领域的认知，进而过渡成阅读文字的一种高级能力。

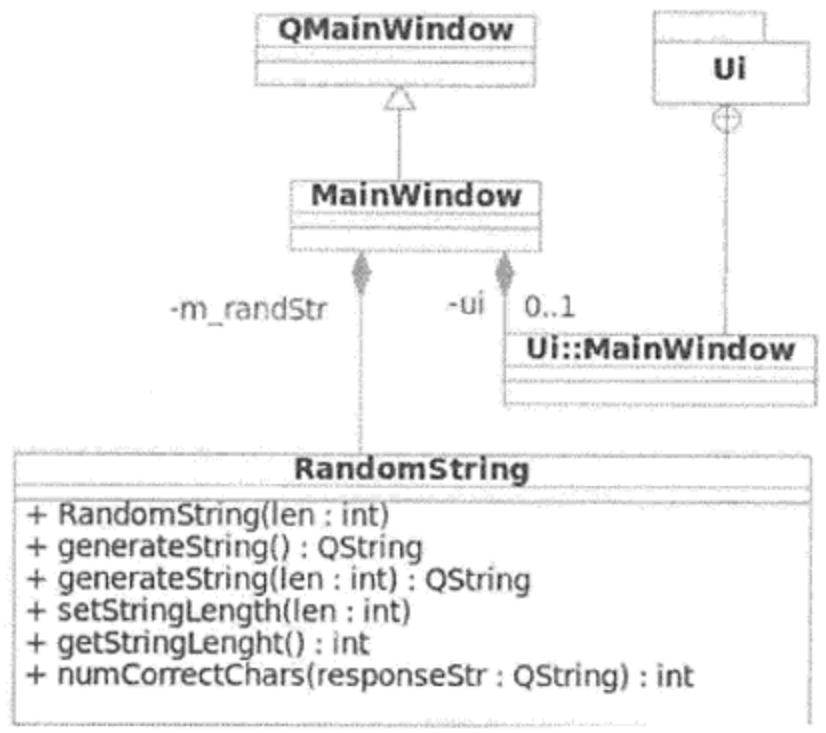


图 9.25 速度—读者的 UML 框图

这个应用程序要相对简单些，从图 9.25 中的 UML 框图即可看出。

示例 9.17 中给出了 MainWindow 的类定义。可以看到它的若干个槽使用了让 QMetaObject :: connectSlotsByName (MainWindow) 可以正常工作的命名惯例。这些名称是由 QtCreator 基于这种命名惯例生成的——这就是为什么在代码中看不到它们的连接语句的原因。

示例 9.17 src/timer/speed-reader/mainwindow.h

```
[ . . . . ]
class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    explicit MainWindow(QWidget* parent = 0);
    ~MainWindow();
protected:
    void changeEvent(QEvent* e);
    void processTrial();
private:
    Ui::MainWindow* ui;
private slots:
    void on_nextButton_clicked();
    void on_responseString_returnPressed();
    void on_startButton_clicked();
    void on_lengthSlider_valueChanged(int value);
```

```
    void on_exposureSlider_valueChanged(int value);
    void timerDisplayRandStr();
private:
    int m_expInterval;
    RandomString m_randStr;
    int m_trials;
    int m_correctChars;
    int m_totTrials;
    int m_totCorrectChars;
};
[ . . . . ]
```


这个应用程序使用 QTimer 的静态函数 singleShot() 来控制每个随机字符串的显示时间。在选中的滚动时间间隔耗尽后，singleShot() 会向 timerDisplayRandStr() 槽发射一个 timeout() 信号。如示例 9.18 所示，processTrial() 给出了在没有显式的连接语句的情况下是如何创建连接的。

示例 9.18　src/timer/speed-reader/mainwindow.cpp

```
[ . . . . ]
void MainWindow::processTrial() {
    //clear response text editor
    ui->responseString->setText("");
    //display the random string
    ui->targetString->setText(m_randStr.generateString());
    ui->responseString->setEnabled(false);
    ui->nextButton->setEnabled(false);
    //count the number of trials
    m_trials++;
    m_totTrials++;
    ui->nextButton->setText(QString("String %1").arg(m_trials));
    //begin exposure
    QTimer::singleShot(m_expInterval, this, SLOT(timerDisplayRandStr()));
}


void MainWindow::timerDisplayRandStr() {
    ui->targetString->setText(QString(""));
    //enable the response line editor and next button
    ui->responseString->setEnabled(true);
    ui->responseString->setFocus();
    ui->nextButton->setEnabled(true);
}

[ . . . . ]
```

图 9.26 给出了在选中显示时间和字符串长度后程序的运行截图。

9.9.1.1　练习：QTimer

1. 修改速度—读者应用程序(从示例 9.17 开始)，以便让敲入响应字符串的时间量可限制在用户所给定的时间量之内。需要在用户界面中添加另一个输入窗件。
2. 让用户在工作时可以用一些语言单词而不是随机的字符串。对于这个问题，可自由使

用 src/handouts/canadian-english-small 文件，其中含有英语中的一些特殊方言单词。在本书的 dist 目录中可以下载到 src 压缩包。应该如何控制这些单词字符串的长度?

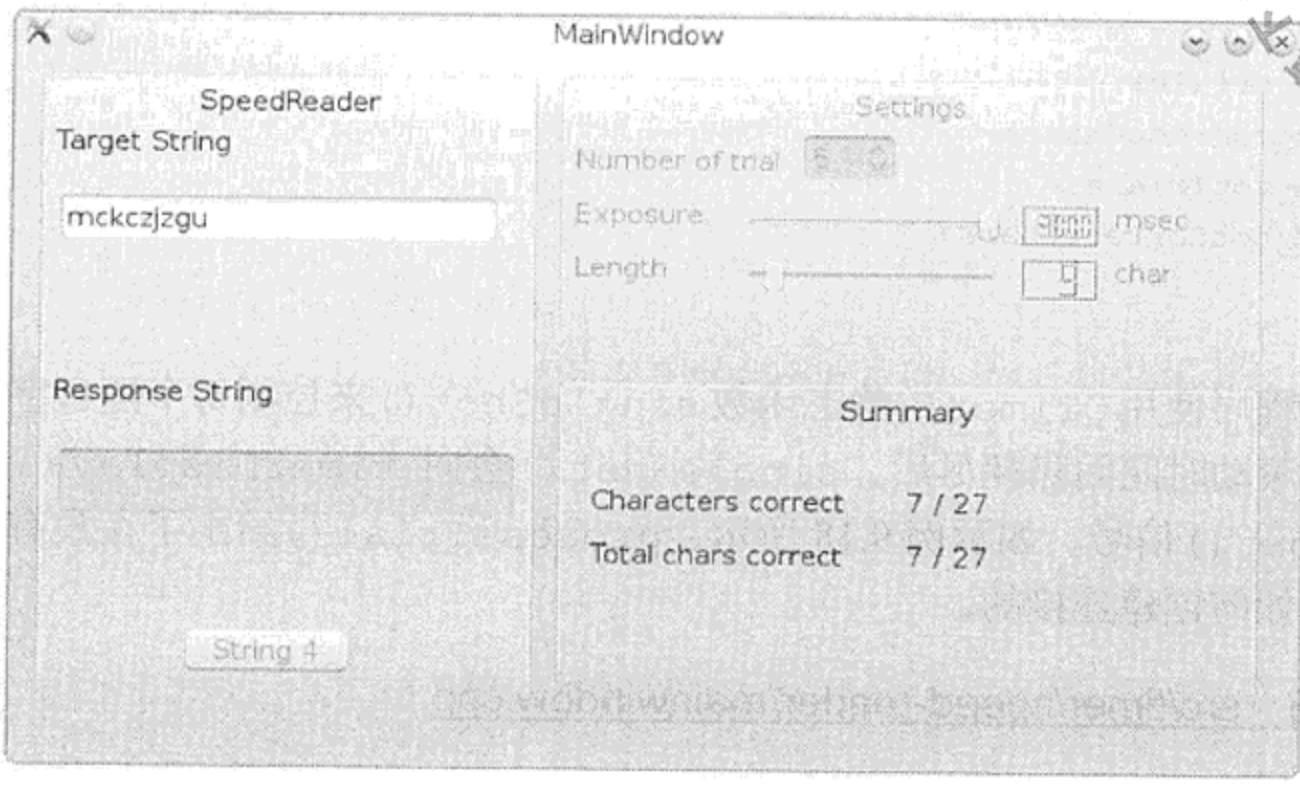


图 9.26 速度—读者屏幕截图

9.10 绘制事件和画图

一个窗件应当在其 paintEvent() 方法中执行适当的绘制操作。这是 QWidget 中唯一可以创建 QPainter(this) 的地方。

正如所看到的，有下面的几个原因使得 QPaintEvent 可以被发送到 QWidget 上。

1. 窗件是隐藏的，然后又显露了出来。
2. 窗件改变了大小或者进行了重新排布。
3. 调用了 update() 或者 repaint()。

示例 9.19 中定义了一个重载了 paintEvent() 的自定义 QWidget。

示例 9.19 src/widgets/life/lifewidget.h

```
[ . . . . ]
class LifeWidget : public QWidget
{
    Q_OBJECT
public:
    explicit LifeWidget(QWidget* parent = 0);
    ~LifeWidget();
    QSize sizeHint() const;
    void paintEvent(QPaintEvent* evt);
public slots:
    void setImage(const QImage& image);
private:
    QImage m_image;
    QSize m_size;
};
[ . . . . ]
```

1 自定义绘制事件。

为 QWidget 获得一个 QPainter 的步骤如下。

1. 创建一个 QPainter(this)。
2. 使用 QPainter 的 API，在 QWidget 上进行绘制。

示例 9.20 给出了一个带有离屏 QImage 的 paintEvent()例子，并直接将其绘制到 QWidget 上。

示例 9.20　src/widgets/life/lifewidget.cpp

```
[ . . . . ]

void LifeWidget::paintEvent(QPaintEvent* evt) {
    QPainter painter(this);                                    1
    if (!m_image.isNull())
        painter.drawImage(QPoint(0,0), m_image);
}
```

1 绝大多数 paintEvent 的第一行。

这个程序基于在康威的《游戏人生》(Conway's Game of Life)[①]中描述的规则，绘制了连续数代的人口地图。图 9.27 给出了一代人口的截屏。17.2.3 节中对其进行并行化时还会再次用到这个游戏。

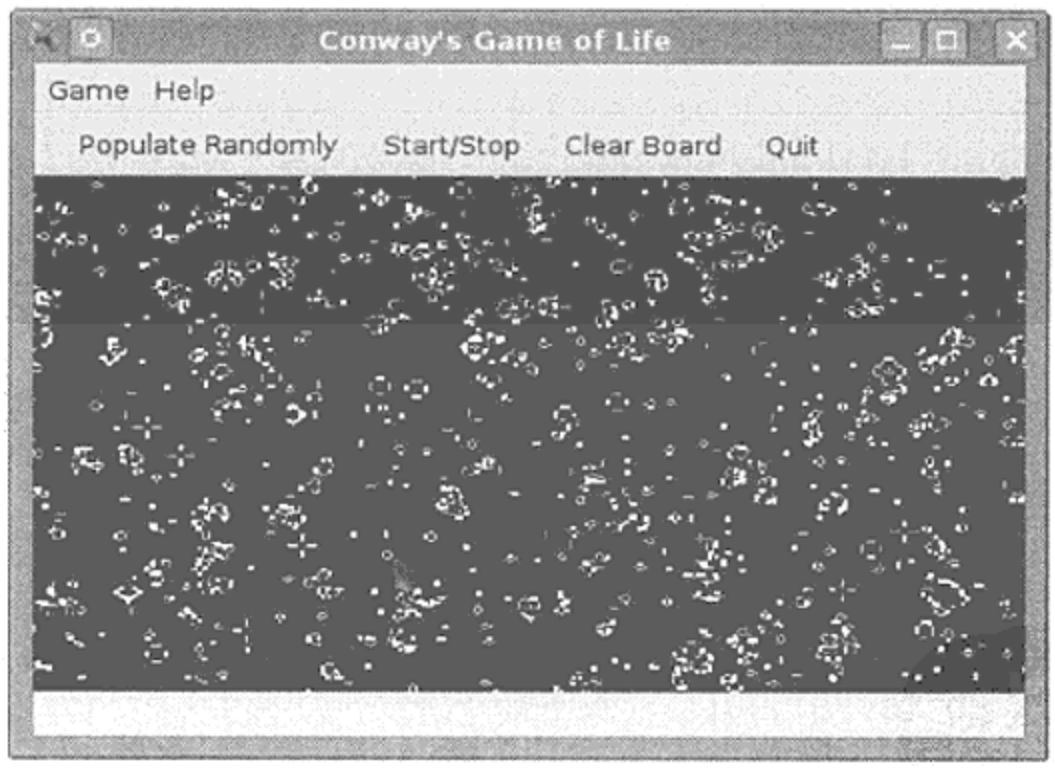


图 9.27　康威的《游戏人生》

通常情况下，不会直接调用 paintEvent()，但可以同步或者异步地将其调度到调用序列中。repaint()在 paintEvent()得到调用后才会返回。update()在 QPaintEvent 被放进事件队列后会立即返回。示例 9.21 是把图片缩放到正确的尺寸并且对其加以保存，在调用 update()之前，确保 LifeWidget 显示的是不久时间内的新图片。

① 参见http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life。

示例 9.21 src/widgets/life/lifewidget.cpp

```
[ . . . . ]

void LifeWidget::setImage(const QImage& image) {
    m_size = image.size();
    m_image = image.scaled(size());
    update();                                          1
}
```

1 异步——立刻返回。

9.11 复习题

1. 列举所有 QWidget 都有的 6 件事物。
2. 什么是对话框？使用对话框的场合有哪些？
3. 什么是 QLayout？其目的是什么？列举一个具体的 QLayout 类的例子。
4. 窗件可以是布局的子对象吗？
5. 布局可以是窗件的子对象吗？
6. 一个布局可以是另一个布局的子对象吗？
7. 在资源文件中列举图像的优点是什么？
8. 分隔(spacer)和伸展(stretch)的区别是什么？

第 10 章　主窗口和动作

绝大多数的 QApplication 都会管理一个单独的 QMainWindow。如图 10.1 所示，QMainWindow 拥有一些与大多数桌面应用程序相同的特性：

- 中央窗件
- 菜单
- 工具栏
- 状态栏
- 停靠区域

在绝大多数的应用程序中，QMainWindow 都会是(在此主窗口内的)所有 QAction，QWidget 以及 QObject 型堆对象的(祖)父对象。如示例 10.1 所示，为应用程序扩展该类，是常见的做法。

示例 10.1　src/widgets/mainwindow/mymainwindow.h

```
[ . . . . ]
class MyMainWindow : public QMainWindow {
    Q_OBJECT
 public:
    explicit MyMainWindow(QWidget* parent=0);
    void closeEvent(QCloseEvent* event);                    1

 protected slots:
    virtual void newFile();
    virtual void open();
    virtual bool save();
[ . . . . ]
```

1 对基类的函数进行重写，用以捕捉用户打算关闭该窗口的时机。

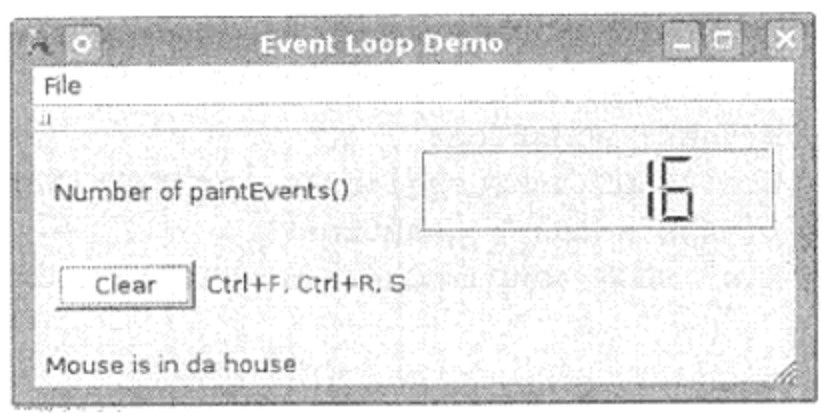


图 10.1　QMainWindow

10.1　QAction，QMenu 和 QMenuBar

QAction 从 QObject 派生而来，是用于用户选定动作的一个基类。它提供了丰富的接口，正如所看到的那样，它还可以用于许多种动作中。QWidget 接口使得每个窗件都可以维护一个 QList<QAction*>。

所有 QWidget 都可以拥有 QAction。一些窗件借助上下文菜单提供了 QAction 的清单，其他则借助菜单栏。有关如何为窗件提供上下文菜单的细节，可以参考 setContextMenuPolicy() 的 API 文档。

QMenu 是一个能够给 QAction 集合提供特殊视图的 QWidget。QMenuBar 是菜单的一个集合，常见于 QMainWindow 中。

若 QMenu 的父对象是 QMenuBar，QMenu 就会表现为一个拥有常规接口的下拉菜单；若其父对象不是 QMenuBar，它可以像对话框一样弹出，这种情况下，它就是一种上下文菜单[①]。当然，一个 QMenu 也可以把另外一个 QMenu 作为父对象，此时，前者就变成了子菜单。

为了帮助用户做出正确的选择，每个 QAction 都可拥有如下元素。

- 可显示于菜单或按钮上的文本或图标。
- 加速键或者快捷键。
- 一个“What's this?”和一个工具提示。
- 用来切换可见/不可见、启用/禁用、选中/未选中等动作状态的方法。
- changed()，hovered()，toggled()和 triggered()信号。

图 10.2 中的 QMainWindow 只有一个菜单栏，其中含有单一的菜单，提供了两个选项。

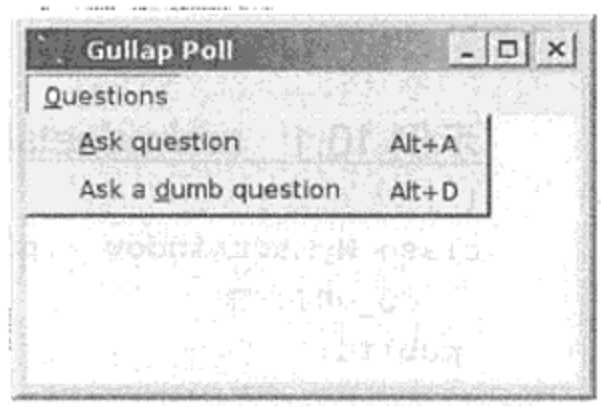


图 10.2 QMenu

示例 10.2 给出了创建该菜单栏的代码。QMainWindow::menuBar() 函数会返回一个指向 QMenuBar 的指针，它是 QMainWindow 的子对象。如果菜单栏尚未存在的话，menuBar() 函数会创建并返回一个指向空 QMenuBar 子对象的指针。

示例 10.2 src/widgets/dialogs/messagebox/dialogs.cpp

```
[ . . . . ]

    /* Insert a menu into the menubar. */
    QMenu* menu = new QMenu(tr("&Questions"), this);

    QMainWindow::menuBar()->addMenu(menu);

    /* Add some choices to the menu. */
    menu->addAction(tr("&Ask question"),
                    this, SLOT(askQuestion()), tr("Alt+A"));
    menu->addAction(tr("Ask a &dumb question"),
                    this, SLOT(askDumbQuestion()), tr("Alt+D"));
}
```

每次调用 QMenu::addAction(text, target, slot, shortcut) 函数，都会创建一个未命名的 QAction，并且会将其添加到这个 QMenu 上。然后会调用它的基类函数，QWidget::addAction(QAction *)，这样把新创建的 QAction 添加到用于上下文菜单的 QMenu 的 QAction 清单中。后者会在菜单的 QList<QAction*>中添加该新动作。

---

① 上下文菜单一般通过单击鼠标右键进行调用，或者通过按下“菜单”按钮激活。之所以称为上下文菜单，是因为这种菜单总是依赖于上下文(哪一个 QWidget 或项被选中或者拥有焦点)。

10.1.1　Qaction，QToolbar 和 QActionGroup

因为应用程序可能会给用户提供各种不同的方式(例如，菜单、工具栏按钮、键盘快捷键)来发出同一个命令，将每个命令都封装成一个动作有助于确保整个应用程序拥有一致、同步的行为。QAction 发射信号并根据需要连接到槽上。

在 Qt GUI 应用程序中，动作通常使用以下几种方式进行触发。

- 用户单击菜单项。
- 用户按下快捷键。
- 用户单击工具栏按钮。

QMenu::addAction()有若干个重载形式。下面将使用继承自 QWidget 的版本，即 addAction(QAction *)，如示例 10.3 所示。

在这里，你会看到如何向菜单、动作群组和工具栏中添加动作。其中，首先是从 QMainWindow 派生一个类，并用数个 QAction 成员外加一个 QActionGroup 和一个 QToolBar 来对其进行封装。

示例 10.3　src/widgets/menus/study.h

```
[ . . . . ]
class Study : public QMainWindow {
    Q_OBJECT
 public:
    explicit Study(QWidget* parent=0);
 public slots:
    void actionEvent(QAction* act);
 private:
    QActionGroup* actionGroup;                                    1
    QToolBar* toolbar;                                            2

    QAction* useTheForce;
    QAction* useTheDarkSide;
    QAction* studyWithObiWan;
    QAction* studyWithYoda;
    QAction* studyWithEmperor;
    QAction* fightYoda;
    QAction* fightDarthVader;
    QAction* fightObiWan;
    QAction* fightEmperor;
protected:
    QAction* addChoice(QString name, QString text);

};
[ . . . . ]
```

1 为了捕捉信号。

2 为了将动作显示为按钮。

这个类的构造函数创建了菜单，然后将其安装到已经是基类一部分的 QMenuBar 中，具体可以参见示例 10.4。

示例 10.4 src/widgets/menus/study.cpp

```
[ . . . . ]

Study::Study(QWidget* parent) : QMainWindow(parent) {
    actionGroup = new QActionGroup(this);
    actionGroup->setExclusive(false);
    statusBar();

    QWidget::setWindowTitle( "to become a jedi, you wish?  " );
                                                                    1

    QMenu* useMenu = new QMenu("&Use", this);
    QMenu* studyMenu = new QMenu("&Study", this);
    QMenu* fightMenu = new QMenu("&Fight", this);

    useTheForce = addChoice("useTheForce", "Use The &Force");
    useTheForce->setStatusTip("This is the start of a journey...");
    useTheForce->setEnabled(true);
    useMenu->addAction(useTheForce);                                2
[ . . . . ]

    studyWithObiWan = addChoice("studyWithObiWan", "&Study With Obi Wan");
    studyMenu->addAction(studyWithObiWan);
    studyWithObiWan->setStatusTip("He will certainly open doors for you...");
    fightObiWan = addChoice("fightObiWan", "Fight &Obi Wan");
    fightMenu->addAction(fightObiWan);
    fightObiWan->setStatusTip("You'll learn some tricks from him"
                              " that way for sure!");
[ . . . . ]

    QMainWindow::menuBar()->addMenu(useMenu);
    QMainWindow::menuBar()->addMenu(studyMenu);
    QMainWindow::menuBar()->addMenu(fightMenu);

    toolbar = new QToolBar("Choice ToolBar", this);                 3
    toolbar->addActions(actionGroup->actions());

    QMainWindow::addToolBar(Qt::LeftToolBarArea, toolbar);


    QObject::connect(actionGroup, SIGNAL(triggered(QAction*)),
            this, SLOT(actionEvent(QAction*)));                     4

    QWidget::move(300, 300);
    QWidget::resize(300, 300);
}
```

1 这里所用到的一些“ClassName::”前缀并不是必须的，这是因为，这个函数可以根据“this”进行调用。类名称可以用来显式地调用某个基类版本，或者用来向读者说明调用的是类的哪个版本。

2 它已经在 QActionGroup 中了，但是仍然将其添加到 QMenu 中。

3 这将会给每个 QAction 在可停靠的窗件中添加一个可见按钮。

4 没有连接每个动作的信号，而是针对包含所有这些动作的 ActionGroup 进行了一次连接。

将每个单独的 QAction triggered()信号连接到独立的槽中也是可能的。示例 10.5 中，我们将相关的 QAction 一起连接到一个 QActionGroup 中。如果这个群组中的任意一个成员得到了触发，QActionGroup 就会发射一个单独的信号 triggered(QAction *)，这样就可以按照一种统一的方式来处理一组动作。该信号带有一个指向所触发的特定动作的指针，以便可以选择对应的响应。

示例 10.5　src/widgets/menus/study.cpp

```
[ . . . . ]


// Factory function for creating QActions initialized in a uniform way
QAction* Study::addChoice(QString name, QString text) {
    QAction* retval = new QAction(text, this);
    retval->setObjectName(name);
    retval->setEnabled(false);
    retval->setCheckable(true);
    actionGroup->addAction(retval);                                    1
    return retval;
}
```

1 把每一个动作都添加到 QActionGroup 中，因此只需将信号连接到一个槽上。

在创建之后，每个 QAction 都会通过 addAction()被添加到其他三个对象中。

1. 一个 QActionGroup，用于信号处理。
2. 一个 QMenu，它是 QMenuBar 中三个可能的下拉菜单之一。
3. 一个 QToolBar，这里被渲染成一个按钮。

为了让这个例子更为有趣，在每两个菜单选择项之间创建了一些逻辑依赖关系，使其与各种电影的情节一致。这种逻辑关系在示例 10.6 中的 actionEvent()函数中进行了描述。

示例 10.6　src/widgets/menus/study.cpp

```
[ . . . . ]


void Study::actionEvent(QAction* act) {
    QString name = act->objectName();
    QString msg = QString();

    if (act == useTheForce ) {
        studyWithObiWan->setEnabled(true);
        fightObiWan->setEnabled(true);
        useTheDarkSide->setEnabled(true);
    }
    if (act == useTheDarkSide) {
        studyWithYoda->setEnabled(false);
        fightYoda->setEnabled(true);
        studyWithEmperor->setEnabled(true);
        fightEmperor->setEnabled(true);
```

```
        fightDarthVader->setEnabled(true);
    }


    if (act == studyWithObiWan) {
        fightObiWan->setEnabled(true);
        fightDarthVader->setEnabled(true);
        studyWithYoda->setEnabled(true);
    }
[ . . . . ]

    if (act == fightObiWan ) {
        if (studyWithEmperor->isChecked()) {
            msg = "You are victorious!";
        }
        else {
            msg = "You lose.";
            act->setChecked(false);
            studyWithYoda->setEnabled(false);
        }
    }
[ . . . . ]

    if (msg != QString()) {
       QMessageBox::information(this, "Result", msg, "ok");
    }
}
```

因为所有的动作都在一个 QActionGroup 中，可以只把一个 triggered(QAction*) 信号连接到 actionEvent()槽上。

初始化时，所有菜单选项中只有一个选项是启用的。当用户从可用选项中进行选择时，其他的选项会变成启用的或者禁用的，如图 10.3 所示。值得注意的是，按钮和菜单中的选项之间存在一致关系。单击启用的按钮将会使得相应的菜单变成选中状态。QAction 存储了该状态(启用的/禁用的)，而 QMenu 和 QToolBar 提供了 QAction 的视图。

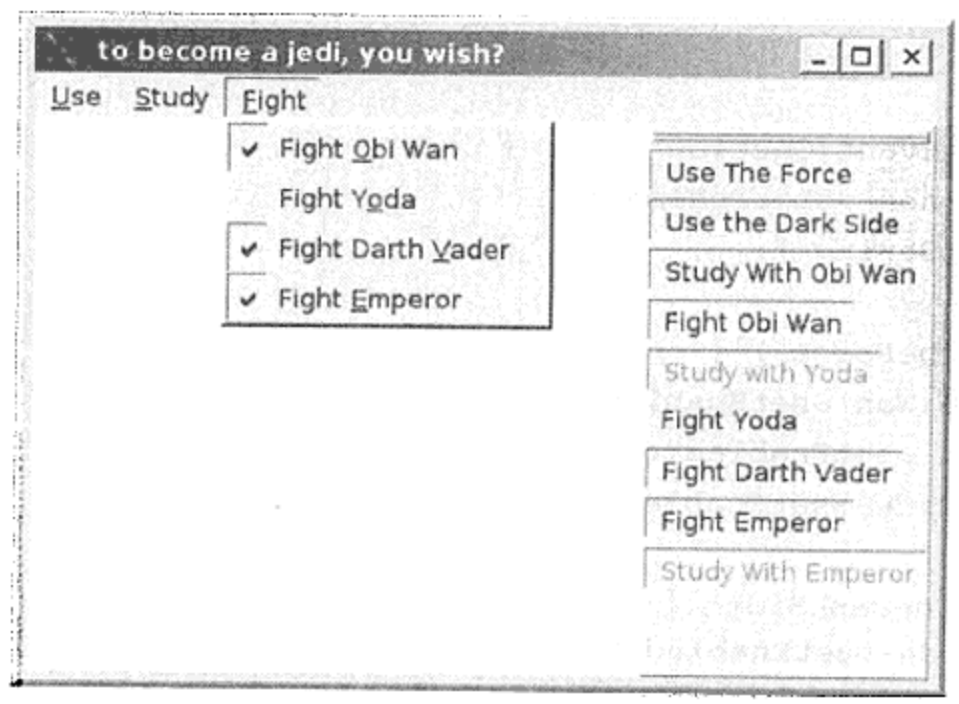


图 10.3 菜单和工具栏中可被选中的动作

10.1.2　练习：CardGame GUI

编写一个 21 点游戏(blackjack game)，其中包含下列动作：

- New game(开始新游戏)
- Deal new hand(新发一手牌)
- Shuffle deck(洗牌)
- Hit me(要另一手牌)
- Stay(计算手中的牌)
- Quit(退出游戏)

借助菜单栏和工具栏，如图 10.4 中应用程序的主窗口所示，这些动作应当是可以使用的。接下来介绍这个游戏的规则。

游戏开始时，会分别给玩家和庄家分发一手牌。每手牌刚开始都只有两张。玩家先玩，可通过单击“Hit Me”动作向其手中添加零次或者多次牌。每单击一次，就添加一张牌。如果玩家不再需要任何牌，那么可以单击“Stay”来触发相应的动作。

游戏的目标是获得不大于 21 点的最大点数。

为了计算手中牌的点数，需将每一张花牌（J, Q 和 K）算作 10 点，而 A 可以算作 1 点或者 11 点，可根据哪种值最好而决定。其他牌的值与其牌面值相等。如果手中有一张 A 外加一张 J，则 A 应算作 11，因为这样的总点数就是 21。但是，如果手中有一张 8 外加一张 7，而手中又增加了一个 A，那么将 A 算作 1 更好一些。

如果玩家的点数大于 21 点，那么玩家在这一局就算“失败”(busted)(输)，这一局也就结束了。

如果玩家手中有 5 张牌，其总点数没有超过 21 点，那么玩家就会赢得这一局。

图 10.4　21 点游戏

在玩家赢、输或者暂停之后，庄家可以根据需要单击多次，目的是使总点数大于 18。当达到这个状态时，庄家必须暂停，这一局也就结束了。如果玩家的点数更加接近 21 点，却没有超过 21 点，那么玩家就赢得此局。如果两者的点数相等，那么庄家赢。

一局结束后，玩家只能选择“Deal Hand”、“New Game”或者“Quit”（“Hit Me”和“Stay”会被禁用）。

在玩家选择“Deal Hand”之后，在当前一局完成之前应会禁用该选项。

跟踪庄家和玩家各自赢的局数，开始时分别初始化为 0，每赢一局就对应地为赢家加 1。在扑克牌的上方显示这两个数字。

如果用户没有选择“Shuffle Deck”或者牌桌还有扑克牌，需不断地发牌而不能重新洗牌。

可以尝试复用或者扩展 6.9.1 节中开发的 `CardDeck` 类及其相关类。通过给每张牌显示一个图片来为自己的游戏添加图形化表示，大致如在 9.5 节和 9.6 节中显示的那样。

为每个 `QAction` 提供一个下拉菜单和一个工具栏。确保 Hit Me 和 Stay 按钮只有在游戏开始之后才能使用。

在窗口顶部用一个只读的 QSpinBox 来显示还剩多少张牌。

新游戏应该将胜负次数归零并将扑克牌归位。

设计建议

尝试将模型类与视图类区分开来，而尽量不要将 GUI 添加到模型类中。总是尽量将模型和视图之间的代码分开是一种良好的编程风格，这样也会让你的项目从头至尾都变得更易于管理。

图 10.5 给出了一种可能的设计方案。

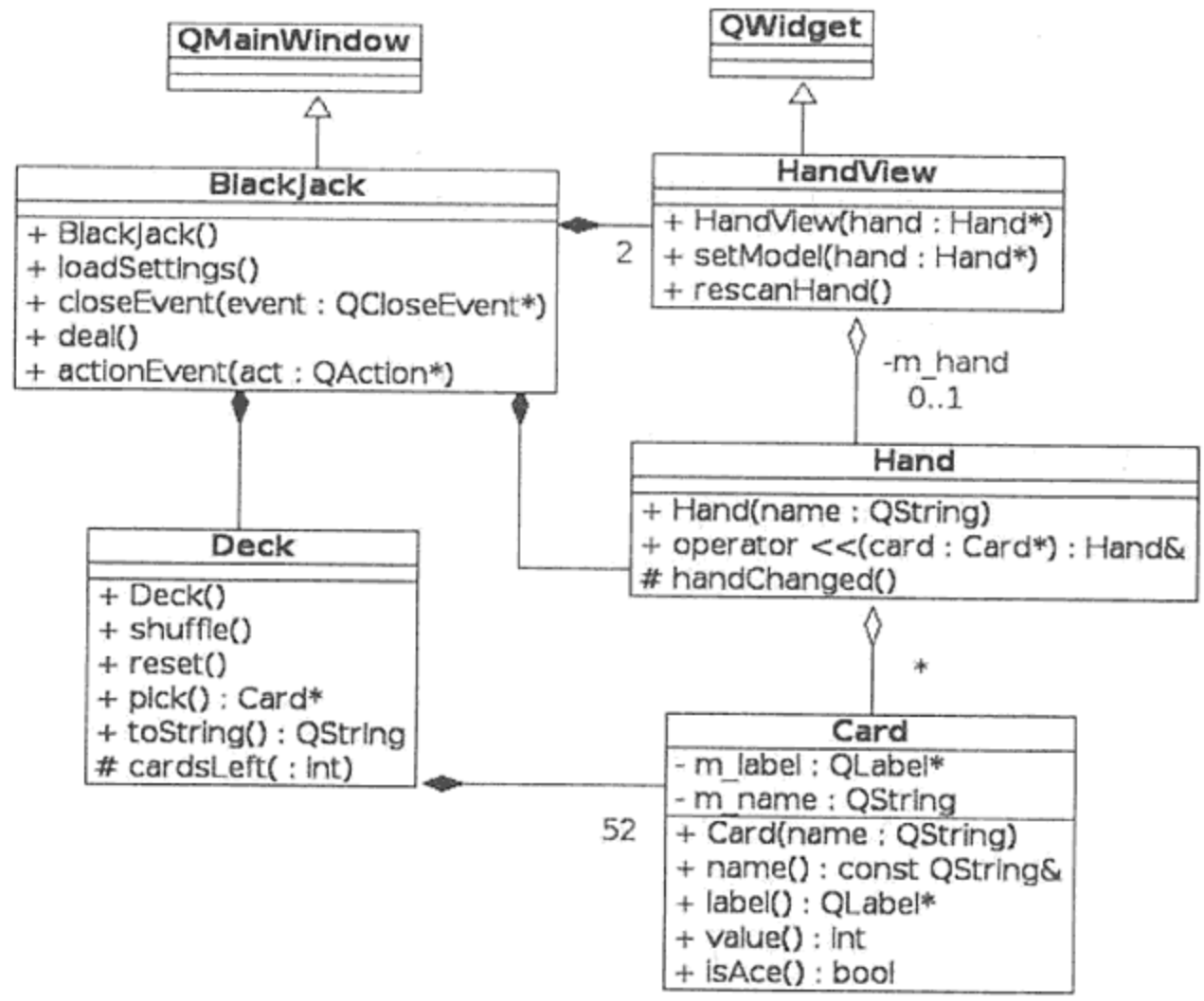


图 10.5 解决方案中的一种设计

10.2 区域和 QDockWidget

任何 QMainWindow 的派生类都有一些停靠窗口区域，分别位于中央窗件的四条边上，如图 10.6 所示。这四个区域可以用来向中央窗件添加二级窗口，也可以称为停靠窗口(dock window)。

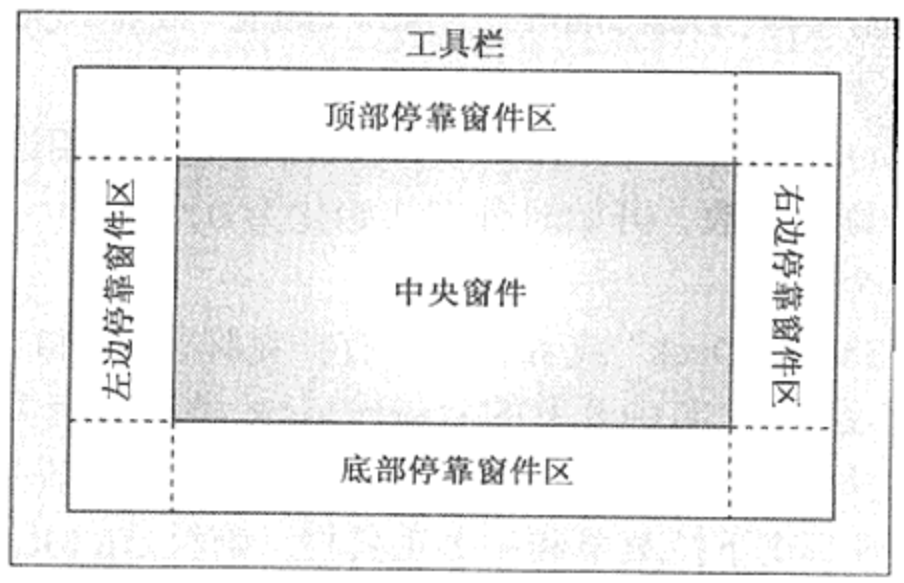


图 10.6 QMainWindow 的四个停靠区域

QDockWidget 可以看作是另外一个窗件的封装。它有一个标题栏、一个可以包含其他窗件的内容区域。根据属性设置值的不同，终端用户可以将 QDockWidget 拖离（以便让它“浮动”）、改变大小、关闭、拖到不同的位置，或者是将其停靠到相同或者不同的停靠窗件区域上。让数个停靠窗件同时占用同一个停靠区域是没有问题的。

QMainWindow 在中央窗件和 QDockWidget 之间恰当地创建了数个可以滑动的 QSplitter。有两个主要的 QMainWindow 函数可以用来管理停靠的窗口区域。

1. setCentralWidget(QWidget *)，此函数的作用是确立中央窗件。
2. addDockWidget(Qt::DockWidgetAreas, QDockWidget *)，此函数的作用是把给定的 QDockWidget 添加到指定的窗口区域。

在集成开发环境中，这些停靠的窗口非常重要，因为在不同的情况下往往需要不同的工具或者视图。每个停靠窗口都是一个可使用停靠机制轻易地将视图拖进主窗口中的窗件，如图 10.7 所示。

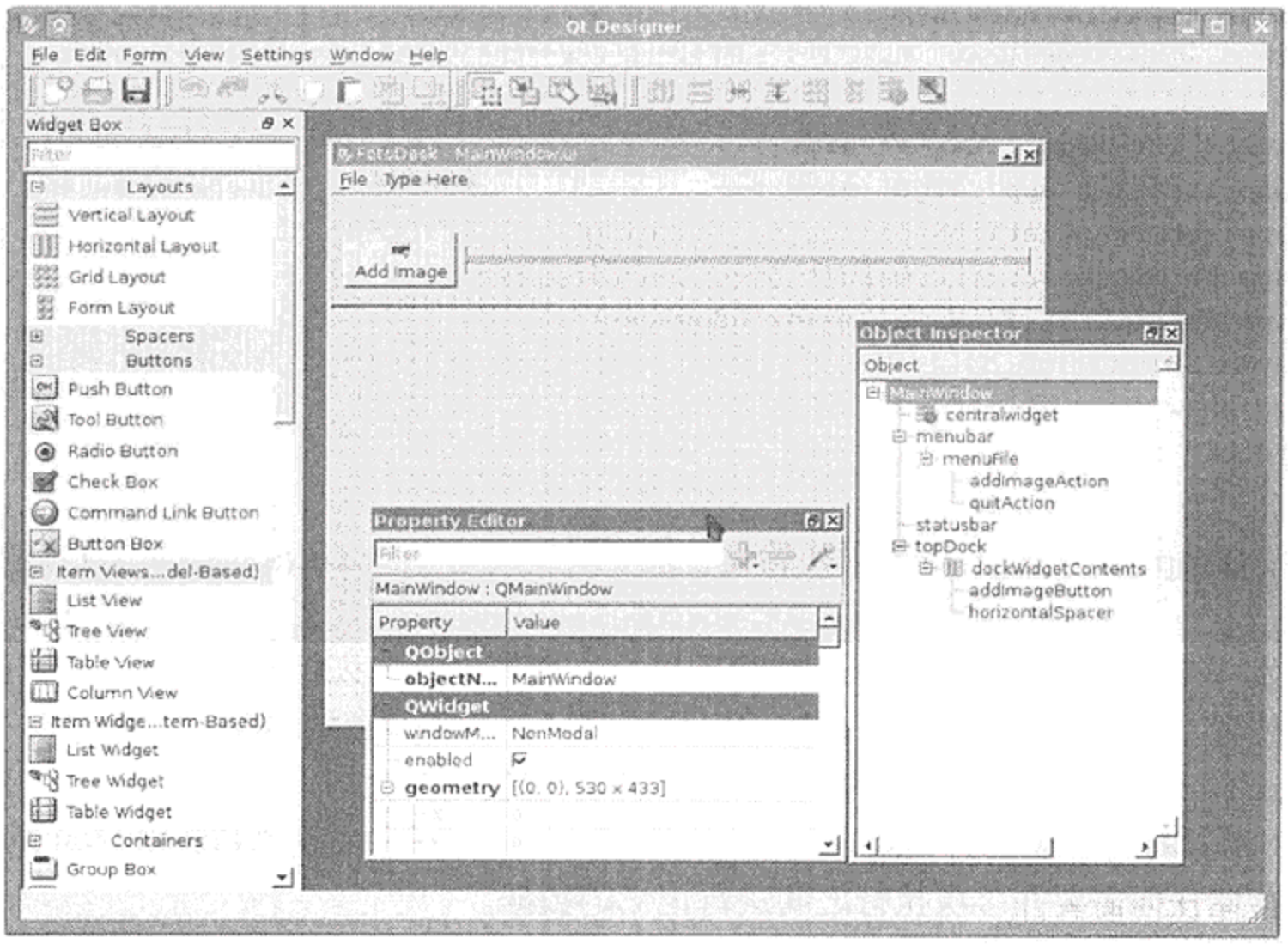


图 10.7　包含几个 QDockWindow 的设计师界面

正如大多数 Qt 应用程序一样，用于设计和构建 GUI 的工具——设计师，也大量使用了停靠窗口。设计师有一个窗件编辑器（中央窗件）和一些用于工具中的可停靠视图，例如：

- Widget Box（窗件工具箱）
- Object Inspector（对象检查器）
- Property Editor（属性编辑器）
- Signal/Slot Editor（信号/槽编辑器）
- Action Editor（动作编辑器）
- Resource Editor（资源编辑器）

这些可停靠窗件并非都是需要同时显示的，因此会有一个视图(View)菜单来允许对它们进行选择或者取消选择。QMainWindow::createPopupMenu()函数可以返回这个菜单，从而可以从上下文菜单中使用它们，也可以将其添加到工具栏或者下拉菜单中。

10.3 QSettings：保存和恢复应用程序的状态

大多数桌面应用程序都有让用户对设置进行配置的方式。而且，所有的设置、喜好和可选项都是需要是持久不变的。实现这一目标的机制都包含在 QSettings 中。

QSettings 在首次使用之前必须用应用程序的名称、组织名和组织域进行初始化。这样可以防止一个应用程序的设置值不小心覆盖了另一个应用程序的设置值。然而，如果 QApplication 自身设置了这些信息，如在示例 10.7 中给出的那样，则 QSettings 的默认构造函数将会使用这些值。

示例 10.7 src/widgets/mainwindow/mainwindow-main.cpp

```
#include "mymainwindow.h"
#include <QApplication>

int main( int argc, char ** argv ) {
    QApplication app( argc, argv );
    app.setOrganizationName("objectlearning");
    app.setOrganizationDomain("objectlearning.net");
    app.setApplicationName("mainwindow-test");
    MyMainWindow mw;
    mw.show();
    return app.exec();
}
```

现在，QApplication 得到了初始化，就可以使用 QSettings 的默认构造函数创建一些实例。

当开发一个新的 QMainWindow 应用程序时，最先希望持久保存的设置值很可能是窗口的大小和位置。也可能希望保存由应用程序最近打开的那些文档的名称。

QSettings 会管理键/值对的永久映射关系。它是一个 QObject，并且会使用一些与 QObject 相似的属性接口——setValue()和 value()——来设置和获得它的值。这个类可以用来存储任何需要在多次执行之间进行记忆的数据。

QSettings 需要一个组织名和一个应用程序名，但当使用默认的构造函数时，QSettings 会从 QApplication 中获得这些值。每个名称组合都会定义一个唯一的永久映射，这使得不会与其他命名的 Qt 应用程序产生冲突。

Monostate 模式

允许多个实例共享相同状态的类，可以看成是 Monostate 模式的一种实现。拥有相同组织/应用程序名称的两个 QSettings 实例，可以用来访问同一个永久映射数据。这简化了应用程序从不同源文件访问公共设置值的过程。

QSettings 是 Monostate 模式的一种实现。

对于 QSettings 数据永久存储的具体机制与实现方法相关且非常灵活。数据存储中的

一些可能实现甚至包括了 Win32 的注册项(Windows 平台)和$HOME/.settings 目录(Linux 平台)。更多详情，可以参阅 Qt 的 QSettings API 文档。

QMainWindow::saveState()函数返回一个包含有主窗口工具条和停靠窗件信息的 QByteArray。为了达到这一目的，使用了每个子窗件的 objectName 属性，因此确保每个名称都不相同是非常重要的。saveState()有一个可选的 int versionNumber 参数。这个 QSettings 对象用关键字符串"state"来存储 QByteArray。

QMainWindow::restoreState()带一个 QByteArray，大概是由 saveState()创建的，它使用其中包含的信息来将工具体和停靠的窗件放入一个之前的镜像中。该函数也有一个可选的 versionNumber 参数。在读取这些设置值后会创建其对象，如示例 10.8 所给出的那样。

示例 10.8　src/widgets/mainwindow/mymainwindow.cpp

```
[ . . . . ]

void MyMainWindow::readSettings() {
    QSettings settings;
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    QByteArray state = settings.value("state", QByteArray()).toByteArray();
    restoreState(state);
    resize(size);
    move(pos);
}
```

应当在用户打算退出应用程序后、窗口关闭前写入设置值。使用事件处理器会更为恰当一些，以便在窗件响应之前可由自己来处理这些事件。示例 10.9 给出了一个用于 closeEvent()的事件处理器，用来向 QSettings 存储这些信息。

示例 10.9　src/widgets/mainwindow/mymainwindow.cpp

```
[ . . . . ]

void MyMainWindow::closeEvent(QCloseEvent* event) {
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}


void MyMainWindow::writeSettings() {
    /* Save postion/size of main window */
    QSettings settings;
    settings.setValue("pos", pos());
    settings.setValue("size", size());
    settings.setValue("state", saveState());
}
```

10.4 剪贴板和数据传输操作

某些时候，将数据从一个地方取出并“发送”到另一个地方是很有必要的。一种方式是使用剪贴板的“剪切并粘贴”，另一种方式是“拖动并放下”。对于这两种操作，用于数据传输的类是相同的。

每个 Qt 应用程序都可以使用 qApp->clipboard()访问系统的剪贴板。剪贴板会保存带类型的数据(文本、图片、URL 或者自定义数据)。要往剪贴板中放置数据，可以创建一个 QMimeData，以一定的方式对数据进行编码，并且调用 QClipBoard->setMimeData()。

- setText()用来保存文本。
- setHtml()用来保存富文本。
- setImageData()用于图片。
- setUrls()用于 URL 或者文件名的列表。

示例 10.10 中，将系统剪贴板变化的信号和 MainWindow 的 clipboardChanged()槽进行了连接。无论何时，只要任意应用程序往剪贴板上复制了任何东西，都会触发该信号。可以运行这个例子并看看当从 QTextEdit 或者其他应用程序复制数据时可以使用哪种数据类型。

示例 10.10 src/clipboard/mainwindow.cpp

```
[ . . . . ]
MainWindow::MainWindow(QWidget* parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect (qApp->clipboard(), SIGNAL(changed(QClipboard::Mode)),
             this, SLOT(clipboardChanged(QClipboard::Mode)));
}

void MainWindow::clipboardChanged(QClipboard::Mode) {
    QStringList sl;
    QClipboard *cb = qApp->clipboard();
    const QMimeData *md = cb->mimeData();
    sl << "Formats: " + md->formats().join(",");
    foreach (QString format, md->formats()) {
        QByteArray ba = md->data(format);
        sl << "    " + format + ": " + QString(ba);
    }
    ui->clipboardContentsEdit->setText(sl.join("\n"));
}
[ . . . . ]
```

图 10.8 给出了从 QTextEdit 复制文本时发生的变化。这种情况下，剪贴板数据以三种方式进行编码：普通文本、HTML 和 OASIS 开放文档格式。选择何种格式取决于在粘贴或者放下对象时所需的数据类型。

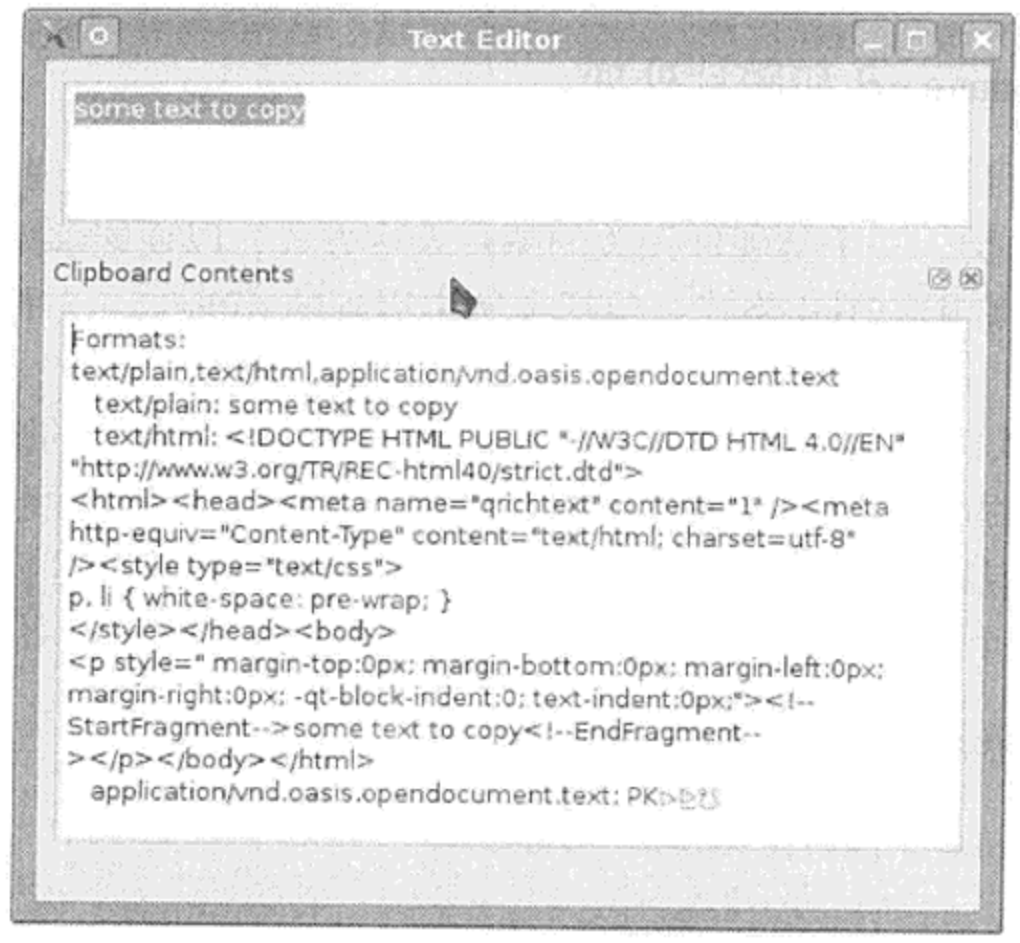


图 10.8　系统的剪贴板演示

10.5　命令模式

在软件中，可能会提供不同的撤销动作(undo)来降低用户的焦虑感并鼓励其尝试新功能。与确认对话框相比，这可能是一种更好的选择。Qt 提供了一些基本的 undo 功能，以使编写带 undo 功能的应用程序更为简单。

命令模式

命令模式，如文献[Gamma95]描述的那样，把操作封装为具有公共执行接口的对象。这样使得将操作放置在一个队列之中、日志维护操作以及取消已执行操作的影响成为可能。该模式完美的实现还可能提供一种处理错误或者异常情况的公用位置。

Qt 类中的 `QUndoCommand`，`QRunnable` 和 `QAction`，可认为在一定程度上实现了这一模式。

利用 Qt 实现命令模式非常简单：

- 可以创建一些命令并在适当的容器(例如，`QQueue`)内对它们进行排队。
- 通过把 `QUndoCommand` 放到 `QUndoStack` 上即可得到执行。
- 如果需要同时执行各个命令，则可以从 `QRunnable` 派生它们，并使用 `QtConcurrent::Run()` 在线程池中对它们进行调度[①]。
- 或许会把一些命令序列化到文件并在随后再次对它们进行调用(也有可能是在另一台机器上)，以用来实现批处理或者分布式执行[②]。

① 17.2 节会讨论线程。

② 17.4 节会讨论序列化模式(Serializer Pattern)。

10.5.1 QUndoCommand 和图片处理

下面的例子显示了 QUndoCommand 的用法，这个程序用到了一些图片处理的操作[①]。这个例子使用 QImage 类，一个与硬件无关的图片表达方式，以便能够处理每个单独的像素。QImage 支持多种最常用的图片格式，包括本例中用到的 JPEG[②]，一种对摄影图像进行有损压缩的系统。

首先，如示例 10.11 所示，从 QUndoCommand 派生一系列典型的图片处理操作。第一个操作通过对每个像素的红、绿和蓝成分进行双倍乘法运算而修正其颜色。第二个操作会根据用户给定的参数，或者水平或者竖直地对图片的另一半进行镜像而替换一半图片。每个操作的构造函数都带有一个源 QImage 的引用，并用同样的尺寸和格式初始化一个空的 QImage。

示例 10.11 src/undo-demo/image-manip.h

```
[ . . . . ]
class AdjustColors : public QUndoCommand {
public:
   AdjustColors(QImage& img, double radj, double gadj, double badj)
     : m_Image(img), m_Saved(img.size(), img.format()), m_RedAdj(radj),
     m_GreenAdj(gadj), m_BlueAdj(badj)   {setText("adjust colors"); }
   virtual void undo();
   virtual void redo();
private:
    QImage& m_Image;
    QImage m_Saved;
    double m_RedAdj, m_GreenAdj, m_BlueAdj;
    void adjust(double radj, double gadj, double badj);

};

class MirrorPixels : public QUndoCommand {
public:
   virtual void undo();
   virtual void redo();
[ . . . . ]
```

这两个操作都会在修改原始图片的任何像素之前生成一个它的备份。示例 10.12 给出了这些操作类中的一个实现，AdjustColors。它的构造函数会通过遍历每个 QImage 的所有像素并对每个像素都调用 pixel() 函数。pixel() 函数会返回颜色的 ARGB 四元组，这是一个无符号、8 字节的 int 值，格式为 AARRGGBB，其中每个字节都表示颜色的一个分量。我们使用 qRed()， qGreen() 和 qBlue() 函数对这个四元组（已由 typedef QRgb 给定）进行操作，以提取独立的三原色值，它们都在 0 到 255 之间[③]。然后它会用修正后的红、绿、蓝的值替换该像素。需要记住的是，颜色的修正操作是用 int 值乘以双精度浮点数后再将乘积赋值给 int 变量的，这就产生了截断(truncation)。换句话说，该调整操作是不能通过执行逆乘操作进行复原的。

---

① 这个例子的灵感来自 Guzdial 和 Ericson 所做的工作[Guzdial07]以及他们的 MediaComp 项目。

② 参见http://www.jpeg.org。

③ 第四个数值对保存 alpha 分量的值，它用来表示像素的透明度。该值的默认值是 ff(255)，表示不透明。

undo()函数会返回已存储的图像备份，redo()函数则会调用像素修改功能。

示例 10.12　src/undo-demo/image-manip.cpp

```
[ . . . . ]

void AdjustColors::adjust(double radj, double gadj, double badj) {
   int h(m_Image.height()), w(m_Image.width());
   int r, g, b;
   QRgb oldpix, newpix;
   m_Saved = m_Image.copy(QRect()); // save a copy of entire image
   for(int y = 0; y < h; ++y) {
      for(int x = 0; x < w; ++x) {
         oldpix = m_Image.pixel(x,y);
         r = qRed(oldpix) * radj;
         g = qGreen(oldpix) * gadj;
         b = qBlue(oldpix) * badj;
         newpix = qRgb(r,g,b);
         m_Image.setPixel(x,y,newpix);
      }
   }
}

void AdjustColors::redo() {
   qDebug() << "AdjustColors::redo()";
   adjust(m_RedAdj, m_GreenAdj, m_BlueAdj);
}

void AdjustColors::undo() {
    qDebug() << "AdjustColors::undo()";
    m_Image = m_Saved.copy(QRect());
}
```

我们使用 QtCreator 来设计 GUI。在转换成 QPixmap 之后，QImage 会显示在屏幕上的 QLabel 中。图 10.9 给出了程序处理之前的一幅照片。

图 10.9　原始照片

图 10.10 给出了这幅照片在经 AdjustColors 和 MirrorPixels 操作处理之后的不成功的结果。

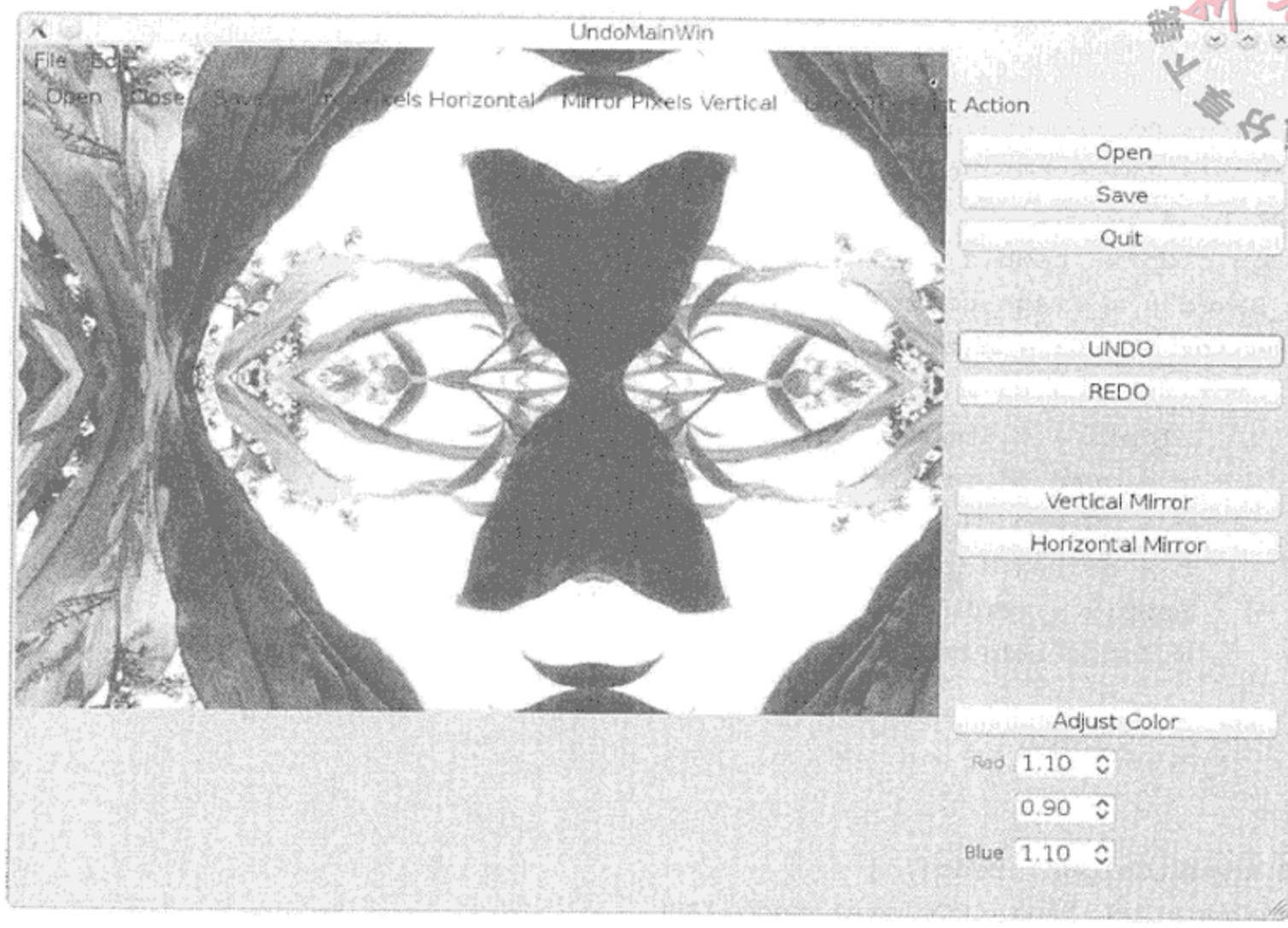


图 10.10 不成功的操作

UndoMainWin 类派生自 QMainWindow 并用到了 QUndoStack。默认情况下，QtCreator 会把 Ui 类作为指针成员嵌入到 UndoMainWin 中。示例 10.13 中，当从设计师中在窗件和动作上使用 Go to slot 特性时，私有槽会先从 QtCreator 生成的部分代码片段开始。

示例 10.13 src/undo-demo/undomainwin.h

```
#ifndef UNDOMAINWIN_H
#define UNDOMAINWIN_H

#include <QMainWindow>
#include <QUndoStack>

class QWidget;
class QLabel;
class QImage;
class QEvent;
namespace Ui {
    class UndoMainWin;
}

class UndoMainWin : public QMainWindow {
    Q_OBJECT
 public:
    explicit UndoMainWin(QWidget* parent = 0);
    ~UndoMainWin();
```

```
  public slots:
     void displayImage(const QImage& img);

  private:
     Ui::UndoMainWin* ui;
     QLabel* m_ImageDisplay;
     QImage m_Image;
     QUndoStack m_Stack;

private slots:
     void on_redoButton_clicked();
     void on_openButton_clicked();
     void on_actionAdjust_Color_triggered();
     void on_actionUndo_The_Last_Action_triggered();
     void on_actionHorizontal_Mirror_triggered();
     void on_actionVertical_Mirror_triggered();
     void on_actionQuit_triggered();
     void on_actionSave_triggered();
     void on_actionClose_triggered();
     void on_saveButton_clicked();
     void on_quitButton_clicked();
     void on_adjustColorButton_clicked();
     void on_undoButton_clicked();
     void on_verticalMirrorButton_clicked();
     void on_horizontalMirrorButton_clicked();
     void on_actionOpen_triggered();
};

#endif // UNDOMAINWIN_H
```

·示例 10.14 中可以看到 QtCreator 用来把各部分连接到一起的实现风格。还要注意的是，示例 10.13 中也给出了一个紧凑的私有槽完整列表。

QImage 在像素处理上得到了优化。QPixmap 使用了视频存储器(video memory)，它也是需要在屏幕上显示图片的多种窗件要用到的类。正如之前提到的，可以把 QImage 转换成 QPixmap 并将其在 QLabel 进行显示。

示例 10.14　src/undo-demo/undomainwin.h

```
[ . . . . ]
#include "image-manip.h"
#include "ui_undomainwin.h"
#include "undomainwin.h"

UndoMainWin::UndoMainWin(QWidget *parent)
: QMainWindow(parent), ui(new Ui::UndoMainWin),
  m_ImageDisplay(new QLabel(this)), m_Image(QImage()) {
  ui->setupUi(this);
  m_ImageDisplay->setMinimumSize(640,480);
}

UndoMainWin::~UndoMainWin() {
  delete ui;
```

```
}

void UndoMainWin::displayImage(const QImage &img) {
    m_ImageDisplay->setPixmap(QPixmap::fromImage(img));
}

void UndoMainWin::on_actionOpen_triggered() {
    m_Image.load(QFileDialog::getOpenFileName());
    displayImage(m_Image);
}

void UndoMainWin::on_horizontalMirrorButton_clicked() {
    MirrorPixels* mp = new MirrorPixels(m_Image, true);
    m_Stack.push(mp);
    displayImage(m_Image);
}

void UndoMainWin::on_adjustColorButton_clicked() {
    double radj(ui->redSpin->value()), gadj(ui->greenSpin->value()),
    badj(ui->blueSpin->value());
    AdjustColors* ac = new AdjustColors(m_Image, radj, gadj, badj);
    m_Stack.push(ac);
    displayImage(m_Image);
}
[ . . . . ]
```

1 既不是 QObject 也不是子对象，必须显式地将其删除。

10.5.1.1 练习：QUndoCommand 和图片处理

向示例 10.11 中添加一些其他的可撤销的图片处理操作。以下是一些值得尝试的做法。

1. 单色摄影术(monochrome)——将一幅三色图片转换成一幅灰度级单色图片。灰度是通过把所有三种颜色分量都设置为同一值而产生的。遗憾的是，如果只是简单地把图片中的每个像素的颜色都用三种颜色的平均值进行简单替换，那么所得的图片的总体效果将显得偏暗。产生可接受的灰度级图片的标准方法是基于纠正这样一个事实：蓝色是一种比红色更“深”一些的颜色。可以通过对这三种颜色中的每种颜色使用通用的权重因子的方法来调整每个像素[①]，如下所示。

   ```
   redVal *= 0.30; greenVal *= 0.59; blueVal *=0.11;
   ```

   然后，可以计算该像素的亮度值。亮度(lūminance，或者称为强度，intensity)是一个等于三种颜色值进行加权平均后的 int 值。这种情况下，因为已经对它们进行了加权处理，因此 、

   ```
   luminance = redVal + greenVal + blueVal
   ```

   最后，用该亮度值替换每一个像素的三种颜色值。
2. 底片——把一幅带三种颜色的图片转换成其负片值。要实现这一点，只需简单地把每一种颜色值 v 用 255 - v 进行替换即可。

---

① 例如，在 http://tinyurl.com/ydpjvgk 中就讨论了亮度值。

3. 打乱颜色——对于每一个像素，交换这些颜色的值，以便让红色值可以得到蓝色的初始值，绿色值可以得到红色的初始值，而蓝色值可以得到绿色的初始值。
4. 三重色——对于每个像素，计算它的颜色强度值 ci(其三种颜色的平均值)。如果 ci 低于 85，把它的红色和蓝色值降为 0。如果 ci 大于等于 85 但低于 170，那么把它的蓝色和绿色的值降为 0。如果 ci 是 170 或者更高，那么把它的红色和绿色的值降为 0。
5. 曝光边界——对于每个像素，把它的颜色强度值与它下方的像素的强度值进行比较。如果差值的绝对值超过阈值(以参数的形式提供)，就把它的颜色设置成黑色(三种颜色的所有值都设置成 0)；否则，把它的值设置成白色(三种颜色的所有值都设置成 255)。

10.6　tr()和国际化

如果所编写的程序可能会被翻译成另外一种语言(国际化)，那么 Qt Linguist 和 Qt 的翻译工具已经为解决这个问题提供了一种解决方案，它们提供了如何组织和何处放置翻译后字符串的方法。为了准备进行代码翻译，需要使用 QObject::tr()函数来包围应用程序中任何需要翻译的字符串。当将其用作非静态函数时，它会将 QMetaObject 所提供的对象的类名称，作为要翻译字符串分组的“上下文”。

tr()函数可以起到两个作用：

1. 它使得 Qt 的 lupdate 工具可以提取所有可翻译的字符串文字。
2. 如果翻译文件可用，并且该语言已被选中，该函数就会返回翻译后的字符串。

如果没有可以使用的翻译文件，那么 tr()函数返回原始字符串。

注意

确保每个可翻译的字符串都完全在 tr()函数内部且在编译时可被进行提取是非常重要的。对于带参数的字符串，可以使用 QString::arg()函数来将参数放置在翻译过的字符串中。例如

```
statusBar()->message(tr("%1 of %2 complete. progress: %3%")
    .arg(processed).arg(total).arg(percent));
```

这样一来，翻译过程中就可以将参数按照不同的顺序进行放置以应对语言导致文字/思想的顺序发生改变。

下列工具可用于翻译过程。

1. lupdate——扫描查找设计师的.ui 文件和 C++文件中可供翻译的字符串。生成一个.ts 文件。
2. linguist——编辑.ts 文件并让用户可以进行翻译工作。
3. lrelease——读取.ts 文件并生成.qm 文件，它们可用来由应用程序载入其翻译。

有关如何使用这些工具的详情，可以参阅 linguist 参考手册[①]。

① 参见 http://doc.qt.nokia.com/latest/internationalization.html。

10.7 练习：主窗口和动作

1. 编写一个文本编辑器应用程序，以 QTextEdit 作为其中央窗件。

   - 在窗口标题栏中显示文件名并说明是否有需要保存的变化。
   - File 菜单：添加用于 Open、Save as 和 Quit 的动作。
   - Help 菜单：添加用于 About 和 About Qt 的动作。
   - 如果有需要保存的变化，退出时询问用户确认退出。

2. 编写一个允许用户从磁盘选择并打开文本（或者是富文本）文件的应用程序，可通过滚动来查看其内容。滚动视图应当每次至少显示 10 行文本。
   该应用程序也应当允许用户搜索文件中的一个字符串。如果找到了该字符串，含有字符串的那一行应当显示在滚动视图中，以便可以让用户在文件中看到它的内容。如果该字符串没有找到，应当在状态栏中显示一个适当的消息。
   用户应当可以单击按钮来查找下一个出现的地方或者回到上一次出现字符串的地方。也应当有 Close 按钮，可从显示中移除文件并请求用户选择另一个文件或者退出。
   图 10.11 是一种可能的解决方案的屏幕截图，其中的两个菜单包含了除 Clear Search（只是用来清空搜索的文本）之外的全部动作的副本。

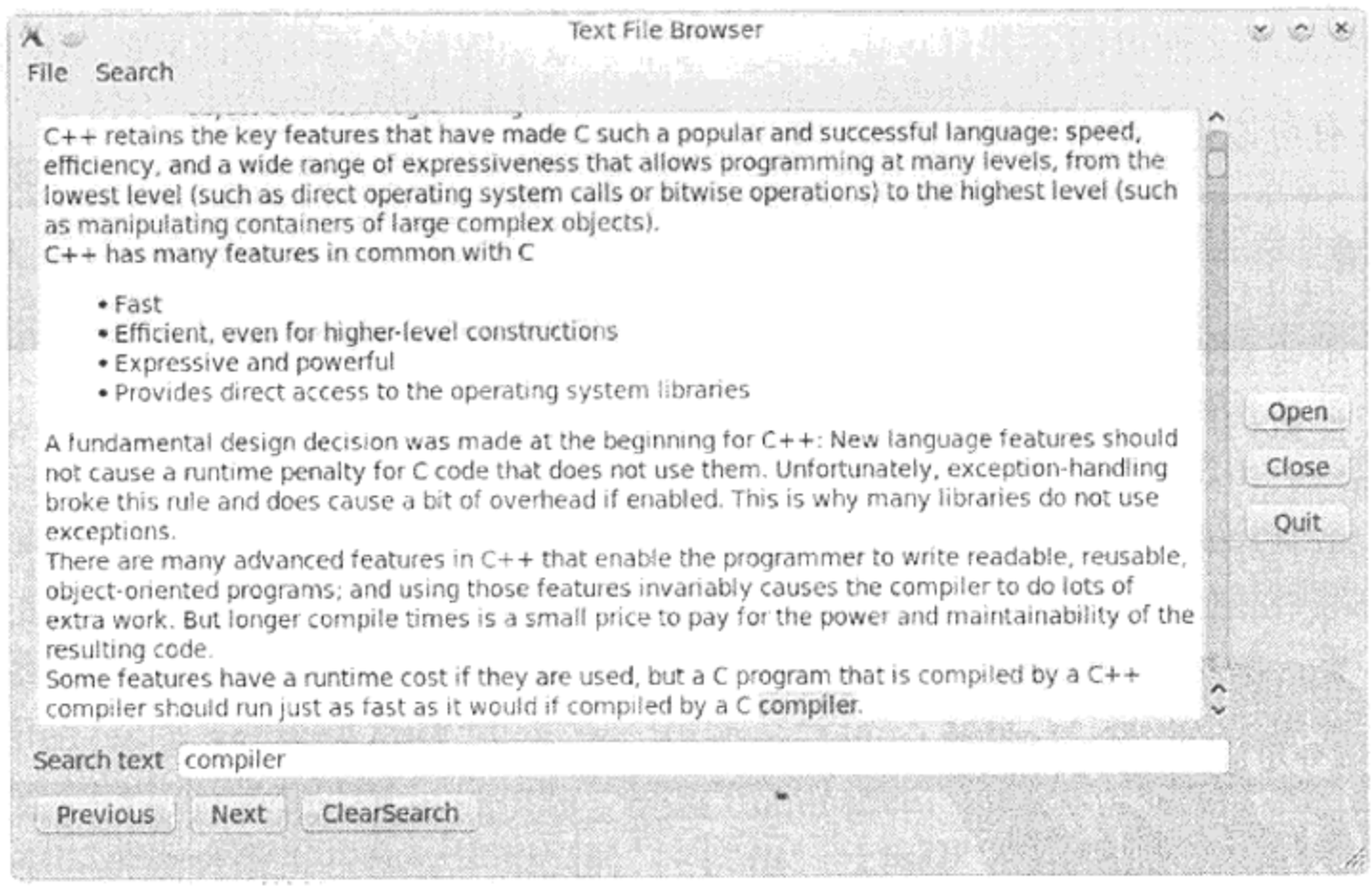


图 10.11 文本文件浏览器

3. QTextEdit 提供了用于 undo 和 redo 的方法。研究这些特性并讨论哪些操作可以进行 undo 和 redo。

10.8 复习题

1. QMainWindow 的主要特点是什么？
2. 如何在 QMainWindow 中安装 QMenuBar？

3. 可以采用什么方法来保存并随后再恢复一个 GUI 应用程序中窗件的大小、位置和排列次序？
4. 什么是中央窗件？
5. 如何让一个窗件变成中央窗件？
6. 停靠窗件是做什么用的？
7. 在一个应用程序中，可以有多少个停靠窗件？
8. 如何使用停靠窗件？
9. 什么是动作？
10. 如何使用动作？
11. 什么是动作群组？为什么愿意在应用程序中使用动作群组？
12. 如何让那些不懂英语的人也可以使用你的应用程序？

第 11 章　范型和容器

本章将会更为深入地讲解范型这一主题。所谓范型(generic)，是指那些能够像操作基本类型一样轻松操作对象的类和函数。Qt 容器类是范型类，也是基于模板的范型类，同时我们还会看到列表、集合和映射的用法。本章还会讨论重载运算符(overloaded operator)、托管容器(managed container)和隐式共享(implicit sharing)。

11.1　范型与模板

C++支持四种不同的类型：

- 基本类型：int, char, float, double 等
- 指针
- 类和结构的实例
- 数组

因为这四种不同类型之间不存在公共基类型，所以如果不使用模板(template)，要编写能够对多种类型进行操作的范型函数和类将会非常困难。模板为 C++编译器提供了一个途径，能够为带有参数化类型和相同行为的类和函数生成多个版本。模板用关键字 template 以及用尖括号<>包围的模板参数进行区分。

与函数参数不同，模板参数不仅可以传递变量和值，还可以传递类型表达式。

```
template <class T > class String { ... };
template <class T, int max > Buffer { ...
    T v[max];
};
String <char> s1;
Buffer <int, 10> intBuf10;
```

11.1.1　函数模板

函数模板用来创建能够按照相同的模式进行工作的类型检查函数。示例 11.1 定义了一个模板函数，该函数通过重复地使用 operator *=来生成一个类型为 T 的值的 exp 次幂。

示例 11.1　src/templates/template-demo.cpp

```
[ . . . . ]

template <class T> T power (T a, int exp) {
  T ans = a;
  while (--exp > 0) {
    ans *= a;
  }
  return (ans);
}
```

再次说明的是，编译器必须执行额外的工作以提供 C++的这一便利特性。编译器会扫描你的代码并基于函数调用时所提供的参数类型生成所需的不同版本的各个函数体，这样一来，诸如示例 11.2 所示的所有调用才能在编译时被解析。尽管模板参数中使用的关键字是 class，T 还是可以用作一个类或者基本类型。在这个例子中，对类型 T 唯一的限制是它必须是一个定义了 operator*=的类型。

示例 11.2　src/templates/template-demo.cpp

```
[ . . . . ]


int main() {
  Complex z(3,4), z1;
  Fraction f(5,6), f1;
  int n(19);
  z1 = power(z,3);                          1
  f1 = power(f,4);                          2
  z1 = power<Complex>(n, 4);                3
  z1 = power(n,5);                          4

}
```

1 第一次实例化，T 是一个复数。

2 第二次实例化，T 是一个分数。

3 如果实际参数不够“具体化”，则显式地提供模板参数，其结果是调用一个已经实例化的函数。

4 此语句会调用哪一个版本的函数呢?

每当编译器看到一个特定参数类型的组合首次用于一个模板函数时，就称此模板进行了实例化。随后用到的 power(Complex, int)和 power(Fraction, int)将会被转化为普通的函数调用。

11.1.1.1　练习：函数模板

1. 完成示例 11.2。特别是，编写一个通用的 Complex 类和 Fraction 类，并修改 main()函数，使其能够使用这两个类来正常工作。
2. 基于示例 5.13 编写一个模板函数 swap()。编写客户代码对其进行深入测试。
3. 是否存在让模板函数 swap()无法正常工作的类型?
4. 指出模板函数 swap()中参数的一些具体限制。

11.1.2　类模板

如同函数一样，类也可以使用参数化的类型。类模板主要用来生成数据的通用容器，其参数能够指明容器中的内容。所有的 Qt 容器类以及标准模板库(Standard Template Library，STL)中的所有容器类都是参数化的。参数就是问题“容器中有什么”的答案。图 11.1 给出了两个模板类的 UML 框图。

UML 类图把模板参数放在类框右上角的一个虚线框中。示例 11.3 中包含了这些类的定义。

示例 11.3 src/containers/stack/stack.h

```
[ . . . . ]

#include <QDebug>
template<class T> class Node {
 public:
    Node(T invalue): m_Value(invalue), m_Next(0) {}
    ~Node() ;
    T getValue() const {return m_Value;}
    void setValue(T value) {m_Value = value;}
    Node<T>* getNext() const {return m_Next;}
    void setNext(Node<T>* next) {m_Next = next;}
 private:
    T m_Value;
    Node<T>* m_Next;
};

template<class T> Node<T>::~Node() {
    qDebug() << m_Value << " deleted " << endl;
    if(m_Next) {
        delete m_Next;
    }
}

template<class T> class Stack {
 public:
    Stack(): m_Head(0), m_Count(0) {}
    ~Stack<T>() {delete m_Head;} ;
    void push(const T& t);
    T pop();
    T top() const;
    int count() const;
 private:
    Node<T>* m_Head;
    int m_Count;
};
```

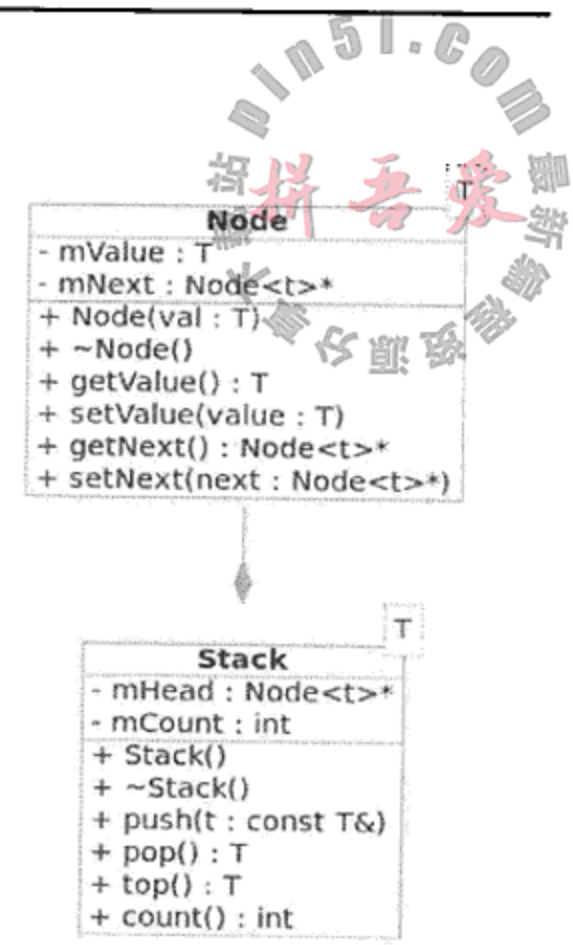


图 11.1 基于模板的栈

所有模板的定义（类和函数）都必须出现在头文件中，这是因为编译器需要用这些定义来根据模板声明生成代码。

值得注意的是，示例 11.3 和示例 11.4 中所需的模板声明代码 template<class T>会放在每个类或函数的定义之前，它们的名称中都有一个模板参数。

示例 11.4 src/containers/stack/stack.h

```
[ . . . . ]


template <class T> void Stack<T>::push(const T& value) {

    Node<T>* newNode = new Node<T>(value);
    newNode->setNext(m_Head);
    m_Head = newNode;
    ++m_Count;
}
```

```
template <class T>  T Stack<T>::pop() {
    Node<T>* popped = m_Head;
    if (m_Head != 0) {
        m_Head = m_Head->getNext();
        T retval = popped->getValue();
        popped->setNext(0);
        delete popped;
        --m_Count;
        return retval;
    }
    return 0;
}
```

对象创建通常是在模板函数 push() 中处理的。Node<T>类的析构函数会递归地删除 Node 指针，直到 m_Next 指针为零[①]。使用这种方法来控制 Node<T>对象的创建和析构使得 Stack<T>能完全地管理其动态内存。示例 11.5 中含有一些用来说明上述各个类的客户代码。

示例 11.5　src/containers/stack/main.cpp

```
#include <QDebug>
#include <QString>
#include "stack.h"

int main() {
    Stack<int> intstack1, intstack2;
    int val;
    for (val = 0; val < 4; ++val) {
        intstack1.push(val);
        intstack2.push(2 * val);
    }
    while (intstack1.count()) {
        val = intstack1.pop();
        qDebug() << val;
    }
    Stack<QString> stringstack;
    stringstack.push("First on");
    stringstack.push("second on");
    stringstack.push("first off");
    QString val2;
    while (stringstack.count()) {
        val2 = stringstack.pop();
        qDebug() << val2;
    }
    qDebug() << "Now intstack2 will self destruct.";
    return 0;
}
```

运行这个程序时，应当可以看到下列输出。

```
3 deleted
3
2 deleted
2
```

① 对一个指针的 delete 操作将会自动地激活与此指针相关联的析构函数。

```
1 deleted
1
0 deleted
0
first off deleted
"first off"
second on deleted
"second on"
First on deleted
"First on"
Now intstack2 will self destruct.
6 deleted
4 deleted
2 deleted
0 deleted
```


注意

因为每个 Q_OBJECT 都需通过 moc 为其生成代码，而且 moc 并未智能到知道如何生成模板类的特化，故而不允许使一个已经标记为 Q_OBJECT 的类再次成为模板类。

11.1.3 练习：范型与模板

1. 将 Stack 的函数定义放在一个单独的文件中(stack.cpp)，适当修改工程文件，然后编译并创建此应用程序。解释该程序的输出结果。
2. 这个应用程序的普适性如何？例如，类 T 必须满足什么条件才能正常工作？
3. Stack<T>的大小有何限制？
4. 编写一个模板类 Queue<T>以及用来测试它的客户代码。

11.2 范型，算法和运算符

通过重载运算符，可以为类定义与基本类型一致的常规接口成为可能。许多范型算法充分利用了这种优势，使用常规运算符来进行基本的函数操作，例如比较等。

11.2.1 qSort

qSort()函数是一个使用堆分类算法进行实现的范型方法[①]。示例 11.6 中给出了使用此函数如何将其用于两个类似但不相同的容器的方法。

qSort()可以应用到任何 Qt 容器，只要容器内的对象拥有公共接口 operator<()和 operator==()。基本数值类型的容器也可以使用此函数进行排序。

示例 11.6 src/containers/sortlist/sortlist4.cpp

```
#include <QList>
#include <QtAlgorithms>    // for qSort()
#include <QStringList>
#include <QDebug>
```

① Wikipedia 中有一篇非常好的介绍堆分类算法的文章，参见 http://en.wikipedia.org/wiki/Heapsort。

```
class CaseIgnoreString : public QString {
public:
    CaseIgnoreString(const QString& other = QString())
    : QString(other) {}
    bool operator<(const QString & other) const {
        return toLower() < other.toLower();
    }
    bool operator==(const QString& other) const {
        return toLower() == other.toLower();
    }
};

int main() {
    CaseIgnoreString s1("Apple"), s2("bear"),
                     s3 ("CaT"), s4("dog"), s5 ("Dog");
    Q_ASSERT(s4 == s5);
    Q_ASSERT(s2 < s3);
    Q_ASSERT(s3 < s4);

    QList<CaseIgnoreString> namelist;

    namelist << s5 << s1 << s3 << s4 << s2;                    1

    qSort(namelist.begin(), namelist.end());
    int i=0;
    foreach (const QString &stritr, namelist) {
        qDebug() << QString("namelist[%1] = %2")
                        .arg(i++).arg(stritr) ;
    }

    QStringList strlist;
    strlist << s5 << s1 << s3 << s4 << s2;                     2

    qSort(strlist.begin(), strlist.end());
    qDebug() << "StringList sorted: " + strlist.join(", ");
    return 0;
}
```

1 毫无规律地插入所有项。

2 值集合保存的是 QString，但是向其中添加了 CaseIgnoreString，因此需要进行转换。

operator<<()在 C 语言中是左移运算符，但在 QList 类中对其进行了重载，现在它的作用是向列表中添加一个项。

示例 11.7 给出了这个程序的输出结果。

示例 11.7　src/containers/sortlist/sortlist-output.txt

```
namelist[0] = Apple
namelist[1] = bear
namelist[2] = CaT
namelist[3] = dog
namelist[4] = Dog
StringList sorted: Apple, CaT, Dog, bear, dog
```

需要注意的是，CaseIgnoreString 对象添加到 QStringList 之后，其排序是区分大小写的。这是因为 CaseIgnoreString 在添加到 strlist 中时必须转换为 QString，因此当对 strlist 的项进行比较时，它们是作为 QString 比较并且排序是大小写敏感的。

11.2.1 练习：范型，算法和运算符

1. QStringList 是“写时复制”对象的值容器，从某种意义上来说，它像是一个指针容器，但是相对来说更加灵活。示例 11.6 中，CaseIgnoreString 会被添加到 QStringList，该过程需要进行(类型)转换。此时需要实际进行字符串数据的复制操作吗？为什么？
2. 向如图 11.2 所示的 ContactList 类中分别添加更多的函数 operator+=, operator-=, add()和 remove()。编写客户代码对这些函数进行测试。

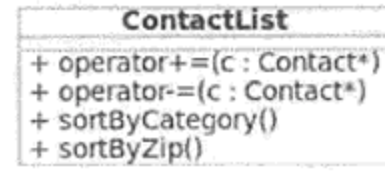


图 11.2 ContactList UML

11.3 有序映射示例

前面曾经提到，QMap 是一个关联数组，能够在添加和删除项时维护键的分类顺序。基于键的插入和删除操作非常快速，效率可达 $\lg n$，且迭代也是按照键的顺序进行的。

QMap 是一个值容器，但指针是简单值，因此可以使用 QMap 来存储分配在堆中的 QObject 的指针。尽管如此，在默认情况下，值容器不会管理堆对象，因此为了避免内存泄漏，必须确保这些对象在合适的时候进行了销毁操作。图 11.3 描述了一个类，它对 QMap 进行了扩展，以包含有关课本的信息。继承了 QMap 之后，QMap 的所有公共接口都变成了 TextbookMap 类公共接口的一部分，而我们仅仅添加了一个析构函数和两个便于添加与显示容器中课本(Textbook)的函数。

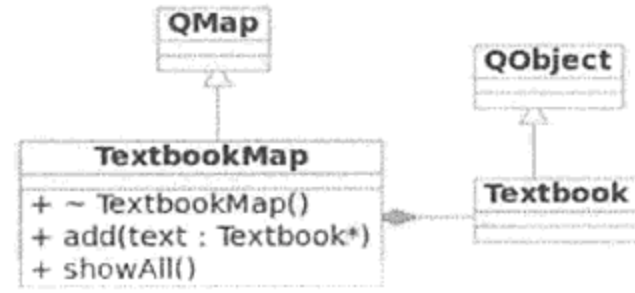


图 11.3 TextbookMap

TextbookMap 由以 ISBN 为键和 Textbook 指针为值的键值对（<key, value>）构成。示例 11.8 给出了该类的定义。

示例 11.8 src/containers/qmap/textbook.h

```
#ifndef _TEXTBOOK_H_
#define _TEXTBOOK_H_

#include <QObject>
#include <QString>
#include <QMap>

class Textbook : public QObject {
    Q_OBJECT
  public:
    Textbook(QString title, QString author, QString isbn, uint year);
[ . . . . ]
private:
    uint m_YearPub;
    QString m_Title, m_Author, m_Isbn;
```

```
};

/* Managed collection of pointers */
class TextbookMap : public QMap<QString, Textbook*> {
  public:
    ~TextbookMap();
    void add(Textbook* text);
    QString toString() const;

};
#endif
```


示例 11.9 中，析构函数使用对 QMap 的 values() 使用 qDeleteAll()，以删除所有的指针。这对于一个值容器来管理其对象是有必要的。

示例 11.9　src/containers/qmap/qmap-example.cpp

```
[ . . . . ]

TextbookMap::~TextbookMap() {
    qDebug() << "Destroying TextbookMap ..." << endl;
    qDeleteAll(values());
    clear();
}

void TextbookMap::add(Textbook* text) {
    insert(text->getIsbn(), text);
}

QString TextbookMap::toString() const {
    QString retval;
    QTextStream os(&retval);
    ConstIterator itr = constBegin();
    for ( ; itr != constEnd(); ++itr)
        os << '[' << itr.key() << ']' << ": "
        << itr.value()->toString() << endl;
    return retval;
}
```

不止一种的遍历映射方法

遍历映射中的每个元素并获得其值，可以有很多种方法。示例 11.9 中用到了 C 语言风格的循环和 STL 风格的迭代器。在单核计算机上，这一做法会非常有效。此处给出的代码是使用 Qt 的 foreach 循环对这一方法的重写，看起来也很不错，但是在运行时，它会创建一个临时列表并需要 $n \log_2 n$ 次的树查找运算，因而在空间或者时间上并不会非常高效。

```
void TextbookMap::showAll() const {
    foreach (QString key, keys()) {
        Textbook* tb = value(key);
        cout << '[' << key << ']' << ":"
             << tb->toString() << endl;
    }
}
```

重要的是要明白，正如示例 11.10 中看到的客户代码那样，当从 TextbookMap 中通过 remove() 移除一个指针时，会同时移除对该指针的管理责任。一旦将其移除，你就要承担删除其指针的职责！换句话说，客户代码很容易产生一些内存泄漏。同样的问题也存在于 QMap 的其他成员函数中，例如 QMap::erase() 和 QMap::take()。通过使用 TextbookMap 的成员函数隐藏这些危险的 QMap 函数可以减少这些问题，TextbookMap 版本能够移除和删除或者重父化(reparent)那些不再需要 Textbook 的指针。另外一种(可能较为安全的)做法是使用 private 派生而不是使用 public 派生。这样，TextbookMap 的公有接口将可能会只包含那些由你仔细定义过的、安全的公有成员函数。

示例 11.10 src/containers/qmap/qmap-example.cpp

```
[ . . . . ]

int main() {
    Textbook* t1 = new Textbook("The C++ Programming Language",
        "Stroustrup", "0201700735", 1997);
    Textbook* t2 = new Textbook("XML in a Nutshell",
        "Harold","0596002920", 2002);
    Textbook* t3 = new Textbook("UML Distilled",
        "Fowler", "0321193687", 2004);
    Textbook* t4 = new Textbook("Design Patterns", "Gamma",
        "0201633612",1995);
    {                                                          1
        TextbookMap m;
        m.add(t1);
        m.add(t2);
        m.add(t3);
        m.add(t4);
        qDebug() << m.toString();
        m.remove (t3->getIsbn());                              2
    }                                                          3
    qDebug() << "After m has been destroyed we still have:\n"
        << t3->toString();
    return 0;
}
```

1 出于演示目的而引入的内部块。

2 移除但未删除。

3 块的结束——局部变量都被销毁了。

从示例 11.11 的输出结果可以看出，当 TextbookMap::ShowAll() 函数对容器进行迭代时，各个 Textbook 已经按照 ISBN(键)的顺序进行了放置。

示例 11.11 src/containers/qmap/qmap-example-output.txt

```
src/containers/qmap> ./qmap
[0201633612]:Title: Design Patterns; Author: Gamma; ISBN: 0201633612;
Year: 1995
[0201700735]:Title: The C++ Programming Language; Author: Stroustrup;
ISBN: 0201700735; Year: 1997
[0321193687]:Title: UML Distilled; Author: Fowler; ISBN: 0321193687;
```

```
Year: 2004
[0596002920]:Title: XML in a Nutshell; Author: Harold; ISBN:
0596002920; Year: 2002
Destroying TextbookMap ...
After m has been destroyed we still have:
Title: UML Distilled; Author: Fowler; ISBN: 0321193687; Year: 2004
src/containers/qmap>
```


11.4　函数指针和仿函数

仿函数(functor)是可被广泛调用的结构。那些可较容易地转换为函数指针(function pointer)的常规 C 和 C++函数，都可以归纳为这一类型。泛型算法通常会通过重载来接受各类可调用结构作为参数。示例 11.12 给出了如何通过函数指针来间接地调用函数。

示例 11.12　src/functors/pointers/main.cpp

```
#include <QtGui>

QString name() {
    return QString("Alan");
}

typedef QString (*funcPtr)();                                    1
Q_DECLARE_METATYPE(funcPtr);                                     2

int main() {
    qRegisterMetaType<funcPtr>("funcPtr");                       3
    funcPtr ptr = name;                                          4

    QString v = (*ptr)();                                        5
    qDebug() << v << endl;                                       6
}
```

1 一个可以返回 `QString` 且不带参数的函数。

2 声明，因而可以用于 `QVariant` 中。

3 注册器，因而可以用于队列中的信号参数。

4 赋值给指向函数的指针的函数名称。

5 通过解引用函数 `ptr` 调用一个方法。

6 输出 “`Alan`”。

指向函数的指针经常用在 C 的回调函数中，或者用于对特定事件进行响应时调用的函数中。在 C++中使用面向对象的、基于模板的各种机制也是可能的。使用这种方式，在编译时就可以指定参数的类型和返回值的类型，从而让它们成为类型安全的。

从它可被解引用并可像函数一样调用的意义上讲，C++中的仿函数是一个行为类似于指针的可调用对象。遵守 RT1[①]或后续版本的 C++的标准库，在头文件<functional>中为这些仿函数提供基本的类类型(class type)。C++的函数调用运算符提供了部分让对象可以像函数一样工作的

① C++技术报告 1(C++ Technical Report 1，TR1)是一份草稿性质的文档，含有对 C++标准函式库建议追加的项目，这些项目很有可能包含在下一个官方标准中。详细信息请参阅 Wikipedia 文章：http://en.wikipedia.org/wiki/C%2B%2B_Technical_ Report_1。

语法糖(syntactic sugar)[①]std::unary_function 和 std::binary_function 类型会为 C++ 的仿函数提供额外的类型信息。它们都是一些可扩展的参数化基类类型，可用于 QtAlgorithms、C++ STL 以及 Qt 的 Concurrent 库中。示例 11.13 展示了如何定义可用于替代函数指针的仿函数。

示例 11.13 src/functors/operators/functors.h

```
[ . . . . ]
class Load : public std::unary_function<QString, QImage> {          1
public:
    QImage operator() (const QString& imageFileName) const {
        return QImage(imageFileName);
    }
};
class Scale {
    public:
    typedef QImage result_type;                                      2
    QImage operator() (QImage image) const {
        for (int i=0; i<10; ++i) {
            QImage copy = image.copy();
            copy.scaled(100, 100, Qt::KeepAspectRatio);
        }
        if (image.width() < image.height()) {
            return image.scaledToHeight(imageSize,
                                        Qt::SmoothTransformation);
        }
        else {
            return image.scaledToWidth(imageSize,
                                       Qt::SmoothTransformation);
        }
    }
};
class LoadAndScale : public std::unary_function<QString, QImage> { 3
public:
    Scale scale;
    QImage operator() (const QString& imageFileName) const {
        QImage image(imageFileName);
        return scale(image);
    }
};
[ . . . . ]
```

1 定义 result_type。

2 仿函数对象所需的一个属性(trait)。

3 还是定义 result_type。

示例 11.14 中，创建了 LoadAndScale 的一个临时实例并将其传递给 QtConcurrent 算法，该算法的映射函数进行了重载，以便可以接受函数指针和它的映射函数的 std::unary_function 对象。17.2 节中将会详细讨论 QtConcurrent。

---

① 语法糖表示“语法中的糖分”。该术语由 Peter J. Landin 创造，是指向一门计算机语言语法中添加附加物或附加成分，它不会影响语言的功能，但能使人类在使用该语言时显得“更甜美”一些。语法糖为程序设计人员提供了一种编写程序的替代方式，这一方式更具实用性和更有助于形成较好的程序设计风格，或者可使代码读起来更自然。但是，语法糖不会影响形式上的可表达性，也不会让语言拥有某种新功能。——译者注

示例 11.14　src/functors/operators/imagefunctor.cpp

```
[ . . . . ]

        connect(m_futureWatcher, SIGNAL(progressValueChanged(int)),
                this, SIGNAL(progressCurrent(int)));
        emit statusMessage("Loading and Transforming in parallel");
        m_futureWatcher->setFuture(QtConcurrent::mapped(files,
                                                        LoadAndScale()));
```

11.5　享元模式：隐式共享类

与 Java 不同，C++没有垃圾收集机制(garbage collection)。所谓的垃圾收集，实际是一个对不再引用的堆内存进行恢复的线程。垃圾收集会在 CPU 相对空闲或者内存不足时开始运行。当一个对象不再被引用后，它会被删除掉，这样，它所占用的内存就可用于其他对象。它具有减少开发人员工作，使其无须担心内存泄漏的好处[①]。但是，它也肯定会加重 CPU 的工作负担。

接下来的例子会给出一种通过引用计数(reference counting)将垃圾收集构建到类的设计中去的方法。引用计数是一个资源共享的例子。无论是从开发人员的角度还是从 CPU 时间的角度看，它都被认为是要比垃圾收集管理堆更为有效的方法。

注意

如果打算修改一个对象(例如，调用了一个非 const 成员函数)，且其引用计数值要比 1 大，那么首先需要的是将其分离，以便它不再共享。

对基于引用计数方式工作的隐式共享类，要避免删除那些共享的已管理对象。使用这个类的客户无须关注其引用计数或者内存的指针。

QString，QVariant 和 QStringList 都是采用这种方式实现的，也就是说，通过值进行参数传递和函数返回是相当快的。如果需要在一个函数内部修改存储在容器内部的值，可以通过引用来传递，而不是通过指针进行传递。

如果通过 const 引用来传递，速度可能还要更快一些，这样会允许 C++把整个复制操作都进行优化。使用 const 引用时，函数无法对引用进行任何修改，也就不会发生自动转换。

享元模式[②]

为了避免对同一对象的多个副本进行存储，在很多情况下，都可以在实际对象出现的地方用一个轻量级封装器(wrapper)来进行代替。封装器类会包含一个指向共享数据的指

① 实际上，Java 开发人员难免有些担心并总是会试图用各种技巧来尽可能少地创建堆对象。他们还会用多种方法来尽可能经常地强制运行垃圾收集程序(并且，这必然会消耗更多的 CPU 周期)。

② Flyweight 一词在拳击比赛中是指最轻量级，即“蝇量级”或者“羽量级”的意思。本书将其译作“享元模式”是为了能够更确切地反映该模式本意：享元模式以共享的方式高效地支持大量的细粒度对象。享元对象之所以可以实现共享，关键是要能够区分内蕴状态(Internal State)和外蕴状态(External State)。内蕴状态存储在享元对象内部，并且不会随环境的改变而改变，故而可以共享内蕴状态；外蕴状态则会随环境的改变而改变，因此也就无法共享。享元对象的外蕴状态必须由客户保存，并在享元对象创建之后，在需要使用时再传入到享元对象的内部。外蕴状态与内蕴状态是相互独立的。详情可参阅 http://tech.it168.com/KnowledgeBase/Articles/8/d/4/8d42715fe1b5939224f3823fcce731da.htm 一文。——译者注

针，而不是对数据的副本进行维护。通过这种方式工作的那些类就是对享元模式(Flyweight Pattern)的实现，有时也称为桥接模式(Bridge Pattern)或者私有实现模式(Private Implementation Pattern)。

在查看 Qt 源代码时，你无疑会注意到，file_p.cpp 和 file_p.h 文件中包含了大多数 Qt 类的实现细节。在改变一个类的实现时，这种模式可有助于确保源代码及其二进制之间的兼容性。此外，这种模式还可用来实现隐式共享，其中的数据可被多个实现共享。

桥接模式的灵活性是有代价的：其代码会更复杂(需要管理的类要两倍以上)，而且必须要实现一个和原始类有同样接口的封装器。

为了能够获得自己的轻量级隐式共享，你可以编写自己的引用计数代码，或者复用这两个 Qt 类：QSharedData 和 QSharedDataPointer。

QSharedData 提供了一个公有的 QAtomicInt ref 成员，可用作引用计数值。QAtomicInt 则提供一个 deref()操作，由 QSharedDataPointer 用来减少和测试引用计数值，以确定它是否可以安全地删除共享数据。QSharedDataPointer 会根据共享数据的复制或者分离来更新共享数据的引用计数值。

下一个例子，如图 11.4 所描绘，从一个相对常规的非共享的 MyString 类开始，它是利用动态字符数组实现的字符串，如示例 11.15 所示。

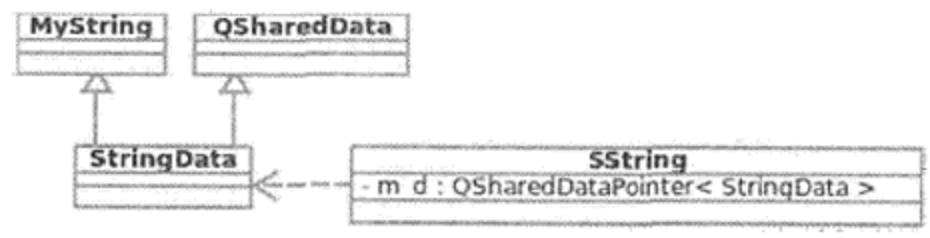


图 11.4 QSharedData 的私有实现示例

示例 11.15 src/mystring/shareddata/mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H
#include <iostream>
class MyString {
public:
    MyString(const MyString& str);
    MyString& operator=(const MyString& a);
    MyString();
    MyString(const char* chptr);
    explicit MyString(int size);
    virtual ~MyString();
    friend std::ostream& operator<<(std::ostream& os, const MyString& s);
    int length() const;
    MyString& operator+= (const MyString& other);
    friend MyString operator+(const MyString&, const MyString&);
protected:
    int   m_Len;
    char* m_Buffer;                                                    1
    void  copy(const char* chptr);
};
#endif        //  #ifndef MYSTRING_H
```

1 指向动态数组起始点的指针。

示例 11.16 通过增加引用计数功能来扩展 MyString 类，这是私有实现类。

示例 11.16　src/mystring/shareddata/stringdata.h

```
[ . . . . ]
class StringData : public QSharedData, public MyString {
public:
    friend class SString;
    StringData() {}
    StringData(const char* d) : MyString(d) {}
    explicit StringData(int len) : MyString(len) {}
    StringData(const StringData& other)
            : QSharedData(other), MyString(other) {}
};
[ . . . . ]
```

如示例 11.17 所示，隐式共享类 SString 是使用 QSharedDataPointer 来实现“写时复制”功能的一个类示例。

示例 11.17　src/mystring/shareddata/sstring.h

```
[ . . . . ]
class SString {
public:
    SString();
    explicit SString(int len);
    SString(const char* ptr);
    SString& operator+= (const SString& other);
    int length() const;
    int refcount() const {return m_d->ref;}
    friend SString operator+(SString, SString);
    friend std::ostream& operator<< (std::ostream&, const SString&);
[ . . . . ]
private:
    // Private Implementation Pattern
    QSharedDataPointer<StringData> m_d;

};
[ . . . . ]
```

SString 的公有方法委托给了 StringData。每当 m_d 以非 const 的方式解引用时，实际的共享数据就会被自动复制。示例 11.18 给出了修改共享元时 refcount()会减小引用值的一个范例。

示例 11.18　src/mystring/shareddata/sharedmain.cpp

```
#include <iostream>
#include "sstring.h"
using namespace std;

void passByValue(SString v) {
    cout << v << v.refcount() << endl;                    1
    v = "somethingelse";
```

```
    cout << v << v.refcount() << endl;
}

int main (int argc, char* argv[]) {
    SString s = "Hello";
    passByValue(s);
    cout << s << s.refcount() << endl;
}
```

1 引用数值等于 2。

2 引用数值等于 1。

3 引用数值等于 1。

QExplicitlySharedDataPointer 与 QSharedDataPointer 一样，但每当需要一个副本时都必须对 QSharedData 进行显式调用 detach()。

无论是出于隐式共享的原因还是出于其他原因，大多数的 Qt 类都实现了享元模式。只有当复制真正修改时，才会复制那些收集的对象，并和原来的容器相脱离。这时才产生和付出时间/内存方面的代价。

11.6 练习：范型

本练习要求给出一些数据结构来跟踪符号之间的关系。一个关系是元素满足下列两个属性的符号集合 S 上的一个布尔运算符 frop:

1. 自反性——对于 S 内的任意符号 s，s *frop* s 总是真（true）。
2. 对称性——对于 S 内的任意符号 s 和 t，如果 s *frop* t 为真，那么 t *frop* s 也为真。

frop 类似于社交网站上的布尔运算符 is-friends-with。is-friends-with 不具有传递性，一个人不会自动和自己的所有朋友的朋友成为朋友[①]。

在这个问题中，可构建一个数据结构来对符号之间的关系进行排序。每一个符号都是一个随机的字母和数字构成的字符串。这个数据结构应当使用一个或者多个 QSet，QMap 或者 QMultiMap。

1. 编写一个程序，重复让用户输入关系或者命令，并不断跟踪所能见到的字符对之间的关系。接口应该相当简单：用户输入要分析的字符串并由分配函数 processLine() 加以处理。换句话说，processLine()应当希望字符串具有下列形式：
   a. 要在两个字符串之间加一个 frop：string1 = string2。
   b. 列出 string1 的朋友：string1(该行中没有符号 =)。
2. 添加另外一个函数 takeback(int n)，其中 n 代表第 *n* 个断言。如果第 *n* 个断言添加了 frop，那么该函数应当“撤销”(undo)那个断言，应确保关系的完整性。在

① 传递性(transitivity)是没有给出的布尔运算符的第三个属性——对于 S 内的任意符号 s，t 和 u，如果 s *op* t 且 t *op* u 都是真，那么 s *op* u 也为真。如果布尔运算符 *op* 具有自反性、对称性和传递性，那么它就是一个等价关系(Equivalence Relation)。

运行此函数之后，给出涉及的两个符号(string1 和 string2)更新后的朋友清单。让 processLine() 函数扫描形如“takeback *n*”的信息，并随之调用 takeback() 函数作为响应。

11.7　复习题

1. 解释模板参数和函数参数之间的一个重要不同之处。
2. 实例化某个模板函数意味着什么？描述一种实例化的方法。
3. 通常来说，需要将模板定义放在头文件中。为什么？
4. 有些编译器支持 export。这有什么作用？
5. Qt 的容器类用来存储值类型。哪些东西不适于使用值集合中的值进行存储？
6. 哪些容器提供从键到值的映射？至少列出并描述两种，然后说明它们之间的不同之处。
7. 一个容器“管理”其堆对象的含义是什么？一个包含堆对象指针的容器怎样才能变成“托管容器”？
8. 至少举出三个实现了享元模式的 Qt 类示例。
9. 在定义一个类模板时，应当怎样将其代码分别放到头文件(.h)和实现文件(.cpp)中？解释你的答案。

第 12 章　元对象，属性和反射编程

本章将引入反射(reflection)的基本思想。所谓反射，就是指对象成员的自我检查。使用反射编程(reflective programming)，就可以编写出通用的操作，可对具有各种不同结构的类进行操作。使用通用的值存储器 QVariant，就可以按照一种统一的方式来对基本类型和其他普通类型进行操作。

12.1　QMetaObject——元对象模式

所谓元对象(meta object)，就是描述另一个对象结构的对象[①]。

> **元对象模式**
>
> QMetaObject 是元对象模式(MetaObject Pattern)的一个 Qt 实现，它提供了一个 QObject 对象所拥有的属性和方法的信息。元对象模式有时也称为反射模式(Reflection Pattern)。

一个拥有元对象的类就可以支持反射。这是一个许多面向对象语言都所具有的性质。虽然 C++中不存在反射，但 Qt 的元对象编译器(MetaObject compiler，moc)可以为 QObject 生成支持这种机制的代码。

像图 12.1 中的 Customer 和 Address 一样，只要满足一定的条件[②]，每个派生自 QObject 的类都会拥有一个由 moc 为其生成的 QMetaObject。QObject 拥有一个成员函数，它能够返回指向对象的 QMetaObject 的指针。这个函数的函数原型是

```
QMetaObject* QObject::metaObject() const [virtual]
```

可以使用 QMetaObject 的下面这些方法来获取一个 QObject 的信息。

- className()，它会将类的名称以 const char*格式返回。
- superClass()，如果存在基类的 QMetaObject，则返回其指针（如果不存在，则返回 0)。
- methodCount()，返回类的成员函数的个数。
- 另外还有几个非常有用的函数，将会在本章的后面进行讨论。

信号和槽机制也同样需要依赖于 QMetaObject。图 12.1 给出了各个 Qt 基类、QObject 的派生类以及由 moc 为其生成的元类(meta class)之间的继承关系。

通过使用 QMetaObject 和 QMetaProperty，就可以编写足够通用的代码来处理所有具有自我描述能力的类。

---

① meta，这个拉丁语词根的含义是“关于”，这里取其字面含义。

② 每个类都必须在头文件中定义，并且列举在工程文件的 HEADERS 中，同时此类的定义中还必须包括 Q_OBJECT 宏。

12.2　类型识别和 qobject_cast

RTTI，全称 Run Time Type Identification（运行时类型识别），如同其名称所显示的一样，是一个用来在运行时决定一个你可能仅仅拥有其基类指针的对象的实际类型的系统。

除了 C++的 RTTI 运算符 dynamic_cast 和 typeid(参见 19.10 节)之外，Qt 还提供了两种运行时的类型识别机制：

1. qobject_cast
2. QObject::inherits()

qobject_cast 是一个 ANSI 风格的类型转换运算符(参见 19.8 节)。ANSI 类型转换看起来很像模板函数：

```
DestType* qobject_cast<DestType*> ( QObject* qoptr )
```

类型转换运算符根据类型和语言的特定规则与约束将表达式从一种类型转换为另一种类型。如同其他转换运算符一样，qobject_cast 把目标类型看做模板参数，它返回指向同一个对象的 DestType 的指针。如果在运行时，实际的指针类型无法转换成 *DestType**，那么转换就会失败，此时返回值是 NULL。

如同 qobject_cast 的字面意思一样，它的参数类型受限于 ObjectType*，其中 ObjectType 类是 QObject 的派生类并且该类完全由 moc 进行处理（这需要其类定义中有一个 Q_OBJECT 宏)。

qobject_cast 实际是一个向下转换运算符，类似于 dynamic_cast。qobject_cast 允许把一个更为常规的指针和引用转换成某种特定的类型。取决于所使用的编译器，你或许可以发现，qobject_cast 的运行速度要比 dynamic_cast 快 5 到 10 倍。

拥有指向派生类的基类指针时，向下转换允许调用在基类接口中不存在的派生类方法。

找到 qobject_cast 的常见地方是在 QAbstractItemDelegate 的实际实现中，比如像示例 12.1 这样。大多数虚函数都把 QWidget*作为参数，所以可以执行类型检查来确定它究竟是何种类型的窗件。

示例 12.1　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    setEditorData(QWidget* editor,
                  const QModelIndex& index) const {
    QVariant val = index.data(Qt::EditRole);
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);          1
    if (starEditor != 0) {
        StarRating sr = qVariantValue<StarRating>(val);                  2
        starEditor->setStarRating(sr);
        return;
    }
```

```
    TimeDisplay* timeDisplay = qobject_cast<TimeDisplay*>(editor);   3
    if (timeDisplay != 0) {
        QTime t = val.toTime();
        timeDisplay->setTime(t);
        return;
    }
    SUPER::setEditorData(editor, index);                             4
    return;
}
```

1 动态类型检查。

2 从 QVariant 中抽取用户类型值。

3 动态类型检查。

4 让基类处理其他类型。

注意

qobject_cast 的实现没有使用 C++ RTTI，该运算符的代码是由元对象编译器生成的。

注意

把 qobject_cast 用于非 QObject 的基类时，需要把每个基类都放到一个形如 Q_INTERFACES(BaseClass1 BaseClass2)的代码行内，并把它放到类定义中 Q_OBJECT 宏的后面。

QObject 还提供了一个不再建议使用的、Java 风格的类型检查函数 inherits()。与 qobject_cast 不同，inherits()按树接收一个 char *类型名，而不是类型表达式。因为该运算符需要进行额外的哈希表查找操作，所以该函数比 qobject_cast 要慢一些，但是如果需要输入驱动(input-driven)型的类型检查，这个函数就非常有用了。示例 12.2 给出了一些使用 inherits()函数的客户代码。

示例 12.2 src/qtrtti/qtrtti.cpp

```
[ . . . . ]

//  QWidget* w = &s;                                                 1

    if (w->inherits("QAbstractSlider"))  cout << "Yes, it is ";
    else cout << "No, it is not";
    cout << "a QAbstractSlider" << endl;

    if (w->inherits("QListView")) cout << "Yes, it is ";
    else  cout << "No, it is not ";
    cout << "a QListView" << endl;

    return 0;
}
```

1 指向某个窗件的指针。

12.3　Q_PROPERTY 宏——描述 QObject 的属性

属性功能使得我们可以选择访问数据成员的方式：

- 直接访问，通过经典的获取函数和设置函数（getter/setter）。速度更快，更为有效。
- 间接访问，通过 QObject/QMetaObject 接口（可让代码复用性更好）。

通过省略 WRITE 函数，可以对一些属性给定只读访问。此外，也可以提供一个在属性发生更改时会发出的 NOTIFY 信号。

示例 12.3 中对 Customer 类的每个数据成员都定义了一个 Qt 属性。QVariant::Type 中列出了可用于属性的可能类型，还有一些加在 Q_DECLARE_METATYPE（参见 12.6 节）中的用户类型。这里采用了在程序开发中获得的经验，给每个属性赋予一个基于对应成员名的名称。例如，如果数据成员的名称是 m_DataItem，那么对应的属性就应当命名为 dataItem。

示例 12.3　src/properties/customer-props.h

```
[ . . . . ]
class Customer : public QObject {
    Q_OBJECT                                                          1

    /* Each property declaration has the following syntax:

    Q_PROPERTY( type name READ getFunction [WRITE setFunction]
    [RESET resetFunction] [NOTIFY notifySignal] [DESIGNABLE bool]
    [SCRIPTABLE bool] [STORED bool] )
    */

    Q_PROPERTY( QString id READ getId WRITE setId NOTIFY valueChanged);
    Q_PROPERTY( QString name READ getName WRITE setName
                NOTIFY valueChanged);
    Q_PROPERTY( QString address READ getAddress WRITE setAddress
                NOTIFY addressChanged);
    Q_PROPERTY( QString phone READ getPhone WRITE setPhone
                NOTIFY phoneChanged);
    Q_PROPERTY( QDate dateEstablished READ getDateEstablished );      2
    Q_PROPERTY( CustomerType type READ getType WRITE setType
                NOTIFY valueChanged);

  public:
    enum CustomerType
    { Corporate, Individual, Educational, Government };               3
    Q_ENUMS( CustomerType ) ;                                         4

    explicit Customer(const QString name = QString(),                5
        QObject* parent = 0);

    QString getId() const {
        return m_id;
    }
[ . . . . ]
```

```
    // Overloaded, so we can set the type two different ways:
    void setType(CustomerType newType);
    void setType(QString newType);
signals:
    void addressChanged(QString newAddress);
    void phoneChanged(QString newPhone);
    void typeChanged(CustomerType type);
    void valueChanged(QString propertyName,
        QVariant newValue, QVariant oldValue = QVariant());
private:
    QString m_id, m_name, m_address, m_phone;
    QDate m_date;
    CustomerType m_type;
};
[ . . . . ]
```

1 moc 预处理类所需要的宏。

2 只读属性。

3 枚举类型定义必须与 Q_ENUMS 宏的定义出现在同一类定义中。

4 特殊的宏可以实现生成字符串到枚举之间的转换功能；必须在同一个类中。

5 之所以声明为 explicit，是因为不希望从 QString 转换到 Customer 时出现意外。

值得注意的是，在 class Customer 的 public 部分定义的 enum CustomerType-defined，其中的 Q_ENUMS 宏告诉 moc 在 QMetaProperty 中为该属性生成一些函数来辅助字符串到枚举值的转换。

示例 12.4 中给出了设置函数和获取函数的定义，它们的实现都采用了常规的方式。

示例 12.4 src/properties/customer-props.cpp

```
[ . . . . ]
Customer::Customer(const QString name, QObject* parent)
:QObject(parent) {
    setObjectName(name);
}

void Customer::setId(const QString &newId) {
    if (newId != m_id) {
        QString oldId = m_id;
        m_id = newId;
        emit valueChanged("id", newId, oldId);
    }
}
[ . . . . ]
void Customer::setType(CustomerType theType) {
    if (m_type != theType) {
        CustomerType oldType = m_type;
        m_type = theType;
        emit valueChanged("type", theType, oldType);
    }
}

/* Method for setting enum values from Strings. */
```

```
void Customer::setType(QString newType) {                              1

    static const QMetaObject* meta = metaObject();                     2
    static int propindex = meta->indexOfProperty("type");
    static const QMetaProperty mp = meta->property(propindex);

    QMetaEnum menum = mp.enumerator();                                 3
    const char* ntyp = newType.toAscii().data();
    CustomerType theType =
                    static_cast<CustomerType>(menum.keyToValue(ntyp));

    if (theType != m_type) {                                           4
        CustomerType oldType = m_type;
        m_type = theType;
        emit valueChanged("type", theType, oldType);
    }
}

QString Customer::getTypeString() const {
    return property("type").toString();
}
[ . . . . ]
```

1 重载的版本，能够接收一个字符串作为参数。如果不清楚该如何设置，则可以将其设置为-1。

2 因为它们是静态局部变量，所以初始化操作仅仅执行了一次。

3 这些代码每次都会执行。

4 始终检测是否需要 valueChanged 信号。

重载函数 setType(QString)的实现充分利用了 QMetaProperty 的 Q_ENUMS 宏，它把 QString 转换成适当的枚举值。为了获得与一个枚举(enum)对应的正确的 QMetaProperty 对象，首先获得了 QMetaObject 对象，然后调用 indexOfProperty()函数和 property()函数进行查找。QMetaProperty 有一个称为 enumerator()的函数，可以用其将字符串转换成枚举值。如果给定的 QString 参数与任何一个枚举值都不匹配，那么 keyToValue()函数将返回-1。

静态局部变量

仔细观察下，我们在代码中是将三个局部(块作用域)变量——meta，propindex 和 mp——定义成静态变量的。静态局部变量仅初始化一次，这也是我们的目标——对此函数的重复调用将会使用同一个的 QMetaProperty 对象来进行转换工作。在一个函数中，这样使用静态局部变量可以极大地提高此函数的运行时性能[①]。

12.4　QVariant 类：属性访问

可以通过下面的函数来获得任意属性的值。

① 当然，这需要取决于创建对象的昂贵程度以及函数调用的频繁程度。

```
QVariant QObject::property(QString propertyName);
```

QVariant 是一个联合体[①]的封装，其中包含了所有基本类型和所允许的全部 Q_PROPERTY 类型。可以把 QVariant 创建为另外一个有类型值的封装。QVariant 会记住自己的类型，并且拥有获取和设置其值的成员函数。

QVariant 中包含丰富的函数来进行数据转换和合法性检查，尤其是有一个 toString() 函数能够为它支持的许多类型返回其 QString 表示[②]。这个类大大简化了属性接口。

示例 12.5 中给出了如何借助直接的获取和设置函数，或者借助间接的 property() 和 setProperty() 函数获得和设置同一个属性值的做法。

示例 12.5 src/properties/testcustomerprops.cpp

```
[ . . . . ]

#include "customer-props.h"
void TestCustomerProps::test() {
    Customer cust;
    cust.setObjectName("Customer");                                    1
    cust.setName("Falafal Pita");                                      2
    cust.setAddress("41 Temple Street; Boston, MA; 02114");
    cust.setPhone("617-555-1212");
    cust.setType("Government");                                        3
    QCOMPARE(cust.getType(), Customer::Government);                    4
    QString originalid = "834";                                        5
    cust.setId(originalid);
    QVariant v = cust.property("id");                                  6
    QString str = v.toString();
    QCOMPARE(originalid, str);
    QDate date(2003, 7, 15);
    cust.setProperty("dateEstablished", QVariant(date));               7
    QDate anotherDate = cust.getDateEstablished();                     8
    QEXPECT_FAIL("", "These are different dates", Continue);
    QCOMPARE(date, anotherDate);
    cust.setId(QString("anotherId"));
    qDebug() << objToString(&cust);
    cust.setType(Customer::Educational);
    qDebug() << " Educational=" << cust.getType();
    cust.setType("BogusType");
    qDebug() << " Bogus= " << cust.getType();
    return;
}

QTEST_MAIN(TestCustomerProps)
```

1 QObject 函数调用。

2 设置一些简单的属性。

3 用字符串来设置枚举属性。

---

① 在 C 和 C++中，联合体是一个数据结构，它声明了两种或者两种以上的数据成员，但将其分配在同一个地址。也就是说，联合体将会占据足够容纳最大已声明数据成员的内存。在初始化时，联合体仅仅可以为其中一个已声明成员存储值。

② 更多详情，可以参阅 QVariant::canConvert()。

4 与枚举值比较。
5 设置一个字符串属性。
6 通过 QObject 基类方法把值作为一个 QVariant 获取回来。
7 设置日期属性，并封装到 QVariant 中。
8 通过类型相关的获取函数取回日期。

示例 12.6 中给出了一个反射的方法 objToString()，它能够对任何定义了 Qt 属性的类进行操作。该函数的工作原理是通过对所有的 property()索引值进行迭代，这种方法与 java.lang.reflect 接口相对应。只有 canConvert(QVariant ::String)变量类型可被打印。

示例 12.6　src/properties/testcustomerprops.cpp

```
[ . . . . ]
QString objToString(const QObject* obj) {
    QStringList result;
    const QMetaObject* meta = obj->metaObject();                    1
    result += QString("class %1 : public %2 {")
        .arg(meta->className())
        .arg(meta->superClass()->className());
    for (int i=0; i < meta->propertyCount(); ++i) {
        const QMetaProperty qmp = meta->property(i);                2
        QVariant value = obj->property(qmp.name());
        if (value.canConvert(QVariant::String))
            result += QString("  %1 %2 = %3;")
            .arg(qmp.typeName())
            .arg(qmp.name())
            .arg(value.toString());
    }
    result += "};";
    return result.join("\n");
}
```

1 通过 QMetaObject 来深入地分析对象。
2 每个属性都有一个 QMetaProperty。

要构建这个程序，需要在工程文件中包含代码

```
CONFIG += qtestlib
```

该程序会以 C++的风格输出一个对象的状态，见示例 12.7。

示例 12.7　src/properties/output.txt

```
********* Start testing of TestCustomerProps *********
Config: Using QTest library 4.6.2, Qt 4.6.2
PASS   : TestCustomerProps::initTestCase()
QDEBUG : TestCustomerProps::test() "class CustProps : public QObject {
  QString objectName = Customer;
  QString id = anotherId;
  QString name = Falafal Pita;
  QString address = 41 Temple Street; Boston, MA; 02114;
  QString phone = 617-555-1212;
```

```
  QString phone = 617-555-1212;
  QDate dateEstablished = 2003-07-15;
  CustomerType type = 3;
};"
QDEBUG : TestCustomerProps::test()  Educational= 2
QDEBUG : TestCustomerProps::test()  Bogus=  -1
PASS   : TestCustomerProps::test()
PASS   : TestCustomerProps::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
********* Finished testing of TestCustomerProps *********
```


12.5 动态属性

即使未用 Q_PROPERTY 在类中定义，在 QObject 中加载和存储一些属性也是可能的。迄今为止，我们已经对用 Q_PROPERTY 宏定义的那些属性进行了专门处理。把这些属性对该类的 QMetaObject 是可知的，且有 QMetaProperty 与之对应。同一类的所有对象会共享同一个 metaObject，因而会有相同元属性组。

另一方面，在运行时会获得动态属性，这些属性对于获得它们的对象而言也是很特别的。换句话说，同一个类的两个对象会具有相同的元属性列表，但对于动态属性可以有不同的动态属性列表。示例 12.8 中定义了一个包含单一 Q_PROPERTY 的类，并给出了一个属性为 someString 的类。

示例 12.8 src/properties/dynamic/dynoprops.h

```
[ . . . . ]
class DynoProps : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString someString READ someString WRITE setSomeString);
 public:
    friend QDataStream& operator<<(QDataStream& os, const DynoProps& dp);
    friend QDataStream& operator>>(QDataStream& is, DynoProps& dp);
    QString someString() { return m_someString; }
    void setSomeString(QString ss) { m_someString = ss; }
    QString propsInventory();
 private:
    QString m_someString;
};
[ . . . . ]
```

示例 12.9 中，propsInventory() 的实现给出了一种显示固定属性和动态属性的方法。固定属性的清单来自 QMetaObject。使用 QMetaProperty::read() 或者 QObject::property() 可以查询属性的值。propertyCount() 函数会设置对 QMetaProperty 清单进行遍历的条件。

QMetaObject 无法知道动态属性。相反，要知道动态属性必须得使用 QObject 的一些方法。可以对利用 QObject::dynamicPropertyNames() 返回的 QList 清单中的名称进行遍历，并可用 QObject::property() 获得其值。

示例 12.9 src/properties/dynamic/dynoprops.cpp

```
[ . . . . ]

QString DynoProps::propsInventory() {
```

```
    static const QMetaObject* meta = metaObject();
    QStringList res;
    res << "Fixed Properties:";
    QString propData;
    for(int i = 0; i < meta->propertyCount(); ++i) {
       res << QString("%1\t%2").arg(QString(meta->property(i).name()))
              .arg(meta->property(i).read(this).toString());          1
    }
    res << "Dynamic Properties:";
    foreach(QByteArray dpname, dynamicPropertyNames()) {
       res << QString("%1\t%2").arg(QString(dpname))
              .arg(property(dpname).toString());
    }
    return res.join("\n");
}
```

1 这里本来也是可以使用 `property(propName)` 的。

除了访问它们时稍有不便之外，动态属性的使用在很大程度上还是与固定属性相同的，例如，可以对它们进行序列化。示例 12.10 中给出了两个序列化运算符的实现方法。在序列化中使用了一种与用于 `propsInventory()` 函数中相类似的技术。

示例 12.10　src/properties/dynamic/dynoprops.cpp

```
[ . . . . ]
QDataStream& operator<< (QDataStream& os, const DynoProps& dp) {
  static const QMetaObject* meta = dp.metaObject();
  for(int i = 0; i < meta->propertyCount(); ++i) {
       const char* name = meta->property(i).name();
       os << QString::fromLocal8Bit(name)                              1
          << dp.property(name);
  }
  qint32 N(dp.dynamicPropertyNames().count());                         2
  os << N;
  foreach(QByteArray propname, dp.dynamicPropertyNames()) {
     os << QString::fromLocal8Bit(propname) << dp.property(propname);
  }
  return os;
}

QDataStream& operator>> (QDataStream& is, DynoProps& dp) {
  static const QMetaObject* meta = dp.metaObject();
  QString propname;
  QVariant propqv;
  int propcount(meta->propertyCount());
  for(int i = 0; i < propcount; ++i) {
     is >> propname;
     is >> propqv;
     dp.setProperty(propname.toLocal8Bit(), propqv);                   3
  }
  qint32 dpcount;
  is >> dpcount;
  for(int i = 0; i < dpcount; ++i) {
```

```
        is >> propname;
        is >> propqv;
        dp.setProperty(propname.toLocal8Bit(), propqv);
    }
    return is;
}
```


1 把 char *序列化成 QString。

2 序列化 int。

3 用 QString 逆向转换来反序列化 char*。

示例 12.11 中给出了说明动态属性用法的客户代码。

示例 12.11 src/properties/dynamic/dynoprops-client.cpp

```
#include <QtCore>
#include "dynoprops.h"

int main() {
    QTextStream cout(stdout);
    DynoProps d1, d2;
    d1.setObjectName("d1");
    d2.setObjectName("d2");
    d1.setSomeString("Washington");
    d1.setProperty("AcquiredProp", "StringValue");
    d2.setProperty("intProp", 42);
    d2.setProperty("realProp", 3.14159);
    d2.setProperty("dateProp", QDate(2012, 01, 04));
    cout << d1.propsInventory() << endl;
    cout << d2.propsInventory() << endl;
    cout << "\nNow we save both objects to a file, close the file,\n"
            "reopen the file, read the data from the file, and use it\n"
            "to create new DynoProps objects.\n" << endl;
    QFile file("file.dat");
    file.open(QIODevice::WriteOnly);
    QDataStream out(&file);
    out << d1 << d2;
    file.close();
    DynoProps nd1, nd2;
    file.open(QIODevice::ReadOnly);
    QDataStream in(&file);
    in >> nd1 >> nd2;
    file.close();
    cout << "Here are the property inventories for the new objects.\n";
    cout << nd1.propsInventory() << endl;
    cout << nd2.propsInventory() << endl;
}
```

示例 12.12 中给出了这个程序的输出结果。

示例 12.12 src/properties/dynamic/output.txt

```
Fixed Properties:
objectName      d1
someString      Washington
```

```
Dynamic Properties:
AcquiredProp    StringValue
Fixed Properties:
objectName      d2
someString
Dynamic Properties:
intProp 42
realProp        3.14159
dateProp        2012-01-04

Now we save both objects to a file, close the file,
reopen the file, read the data from the file, and use it
to create new DynoProps objects.

Here are the property inventories for the new objects.
Fixed Properties:
objectName      d1
someString      Washington
Dynamic Properties:
AcquiredProp    StringValue
Fixed Properties:
objectName      d2
someString
Dynamic Properties:
intProp 42
realProp        3.14159
dateProp        2012-01-04
```

12.6　元类型，声明和注册

QMetaType 是一个用于值类型的辅助类(helper class)。对于 60 多种内置类型，QMetaType 为每个类型 ID 关联了一个类型名，从而使构造和析构可以在运行时动态发生。有一个名称为 QMetaType::Type 的公共枚举，它有所有 QVariant 兼容类型的值。在 QMetaType:: Type 中的枚举值与 QVariant:: Type 中的枚举值一样。

通过使用 Q_ENUMS 宏，我们已在 QVariant 系统中加入了一些自定义的枚举类型。使用 Q_DECLARE_METATYPE(*MyType*)宏也有可能把自己的值类型加到 QMetaType 列表中。如果 *MyType* 有公共的默认复制构造函数和公共的复制构造函数以及一个公共的析构函数，Q_DECLARE_METATYPE 宏使得它可用作 QVariant 中的自定义类型。

示例 12.13 中，我们将一个新的值类型 Fraction，引入到包含它定义的程序的已知元类型库。没有必要明确定义默认构造函数和复制构造函数，也没有必要明确定义析构函数，因为编译器会生成这些函数，它们会进行逐一复制或者逐一赋值，这正是我们所需要的。把这个宏放到头文件类定义的下面是一种标准的做法。

示例 12.13　src/metatype/fraction.h

```
[ . . . . ]
class Fraction : public QPair<qint32, qint32> {
public:
    Fraction(qint32 n = 0, qint32 d = 1) : QPair<qint32,qint32>(n,d)
```

```
    { }
};

Q_DECLARE_METATYPE(Fraction);
[ . . . . ]
```


12.6.1 qRegisterMetaType()

要注册的元类型必须已经用 Q_DECLARE_METATYPE 声明过。模板函数 qRegisterMetaType<T>()会注册类型 T 并返回由 QMetaType 使用的内部 ID。这个函数有一个重载版本，qRegisterMetaType<T>(const char* *name*)，它可以让你注册一个名称作为类型 T 的名称。对于这个函数的调用必须早早地出现在主程序中，一定要在任何试图以一种动态方式尝试使用该注册的类型之前。

声明了元类型之后，存储一个值到 QVariant 中是可能的。示例 12.14 展示了如何存储以及从 QVariant 中取回已声明元类型的值。

示例 12.14 src/metatype/metatype.cpp

```
[ . . . . ]

int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
    qRegisterMetaType<Fraction>("Fraction");
    Fraction twoThirds (2,3);
    QVariant var;
    var.setValue(twoThirds);
    Q_ASSERT (var.value<Fraction>() == twoThirds);

    Fraction oneHalf (1,2);
    Fraction threeQuarters (3,4);

    qDebug() << "QList<Fraction> to QVariant and back."

    QList<Fraction> fractions;
    fractions << oneHalf << twoThirds << threeQuarters;
    QFile binaryTestFile("testMetaType.bin");
    binaryTestFile.open(QIODevice::WriteOnly);
    QDataStream dout(&binaryTestFile);
    dout << fractions;
    binaryTestFile.close();
    binaryTestFile.open(QIODevice::ReadOnly);
    QDataStream din(&binaryTestFile);
    QList<Fraction> frac2;
    din >> frac2;
    binaryTestFile.close();
    Q_ASSERT(fractions == frac2);
    createTest();
    qDebug() << "If this output appears, all tests passed.";
}
```

已注册类型的值可以借助 QMetaType:: construct()动态构造，如示例 12.15 所示。

示例 12.15　src/metatype/metatype.cpp

```
[ . . . . ]

void createTest() {
    static int fracType = QMetaType::type("Fraction");
    void* vp = QMetaType::construct(fracType);
    Fraction* fp = reinterpret_cast<Fraction*>(vp);                    1
    fp->first = 1;
    fp->second = 2;
    Q_ASSERT(*fp == Fraction(1,2));
}
```

1 注意：这是第一次在本书中使用 reinterpret_cast!

QMetaType::construct()会返回一个 void 指针，所以应使用 reinterpret_cast 将它转换成一个 Fraction 指针。

12.7　invokeMethod()

Qt 把信号连接到槽需要一种机制：通过名称以类型安全的方式来间接调用这些槽。当调用槽时，实际是由 invokeMethod()完成的。示例 12.16 显示了它是如何接收一个作为函数名称的字符串的。除了槽，标记为 Q_INVOKABLE 的常规成员也可以用这种方法来间接调用。

示例 12.16　src/refl ection/invokemethod/autosaver.cpp

```
void AutoSaver::saveIfNecessary() {
    if (!QMetaObject::invokeMethod(parent(), "save")) {
        qWarning() << "AutoSaver: error invoking save() on parent";
    }
}
```

与 QObject::connect()类似，invokeMethod()接受一个可选参数 Qt::ConnectionType，该参数可让你来决定是要用同步调用还是要用异步调用。默认情况下为 Qt::AutoConnection，表示在发射者和接收者处于同一个线程中时会同步执行一个槽。

要通过 invokeMethod()向函数传递类型参数，可以用示例 12.17 中的 Q_ARG 宏创建一些值，这样会返回一个 QGenericArgument，它封装了单个参数的类型和值信息。

示例 12.17　src/refl ection/invokemethod/arguments.cpp

```
QByteArray buffer= ... ;
const bool b = QMetaObject::invokeMethod(m_thread, "calculateSpectrum",
                    Qt::AutoConnection,
                    Q_ARG(QByteArray, buffer),
                    Q_ARG(int, format.frequency()),
                    Q_ARG(int, bytesPerSample));
```

12.8　练习：反射

1. 编写一个可创建下列各个类实例的程序——QSpinBox, QProcess, QTimer——并向用户显示一个属性列表（以及一些值，如果它们可以转换成 QString），外加一个所有函数名称的清单。

2. 重写之前的某个 GUI 应用程序，从 `main()` 中使用 `invokeMethod()` 而不是通过直接调用来 `show()` 初始化窗件。

12.9 复习题

1. 从 `QMetaObject` 中可以获得何种类型的信息?
2. 用来实现数据反射的 Qt 类有哪些?
3. 每个 `QObject` 派生类的 `QMetaObject` 代码是如何生成的?
4. 什么是向下转换? 在何种情况下你愿意使用向下转换?
5. 什么是 RTTI? Qt 是如何提供 RTTI 的?
6. 讨论 `dynamic_cast` 和 `qobject_cast` 运算符的区别。
7. 什么是 `QVariant`? 应该如何使用它?
8. 使用 `property()` 和 `setProperty()`，比直接使用获得函数和设置函数有何好处?
9. `property()` 函数会返回什么? 如何才能获得实际的存储值?
10. 请解释向 `QObject` 添加新的可在运行时获得的属性是如何可能的。
11. 说明动态属性是如何实现序列化的。

第 13 章　模型和视图

模型-视图设计框架提供了将底层数据类型集(模型)与呈现给用户的 GUI 类型集(视图)进行分离的工具和技术。模型通常用来组织具有平面表格结构或者层级树形结构的数据。这一章中将给出如何在 Qt 中使用模型类来呈现多种不同类型数据的方法。

在先前的几个示例中，我们看到了那些努力将表现数据的模型类与呈现用户界面的视图代码进行清晰划分的代码。加强这种分离有几个重要的原因。

首先，将模型和视图分离可减少复杂性。模型和视图代码有完全不同的维护准则——变化是由完全不同的因素驱动的——因此当它们保持分离时将更易于维护。此外，将模型与视图分离使得维护使用同一数据的若干个不同的但保持一致的视图成为可能。那些可复用于设计良好的模型类的专业视图类的数量正在不断地增长。

许多 GUI 工具箱都提供有列表、表格，还有树形视图，但是需要开发者将数据存储在视图中。Qt 有从相应的视图类派生出来的部件，如图 13.1 所示。对于没有使用过模型-视图框架的开发者来说，这些部件类或许比与之相对应的视图类容易学习一些。尽管如此，将数据存储在这些部件中导致了在用户界面和底层的数据结构之间产生了很强的依赖性。这些依赖导致在其他类型数据中或者在其他应用程序中复用这些部件相当困难。它也使得维护使用相同数据的多个视图的一致性非常困难。因此，易用性和便利性(尤其在 Qt 设计师中)的代价是灵活性和复印性的降低。

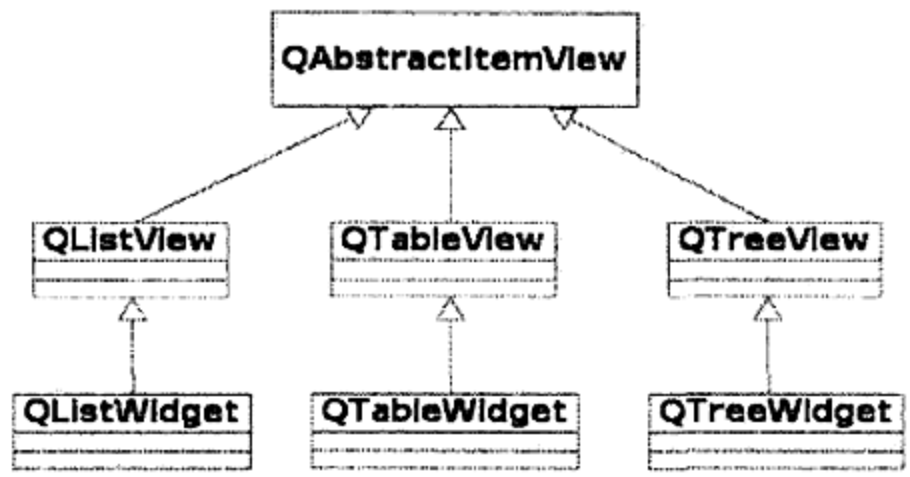


图 13.1　窗件和视图

13.1　模型-视图-控制器(MVC)

控制器代码管理在事件、模型和视图之间的相互交互。通常，工厂方法、委托以及创建和销毁代码都属于控制器的范畴。在 Qt 框架中，许多控制器机制可以在委托中找到。委托控制视图中单个项的渲染和编辑。视图提供了足以满足大多数场景的默认委托。尽管如此，如果有必要，我们可以通过从 `QAbstractItemModel` 派生一个自定义模型来优化默认委托渲染视图项的方式。

MVC：一个经典的设计模式

“四人组”(The Gang of Four)[Gamma95]提供了一个关于 MVC 的简要但精准的描述：

MVC 由三类对象组成。模型是应用程序对象，视图是它的屏幕展示，控制器定义了用户界面对用户输入的反应行为。在 MVC 概念之前，用户界面设计趋向于将这些对象归并在一起。MVC 将它们分拆开来以增强灵活性和复用性。

数据和角色

当获取和设置数据时，有一个可选的 role 参数允许指定 Qt::ItemDataRole 中的某个特定角色，该角色用于视图从模型中获取数据。一些角色指定了一般意义上的数据值，诸如 Qt::DisplayRole(默认)、Qt::EditRole(用于编辑的 QVariant 类型数据)以及 Qt::ToolTipRole(在工具提示中显示的 QString 类型数据)。其他角色用于描述外观，如 Qt::FontRole，使得默认委托可以指定一个特殊的 QFont 对象，或者 Qt::TextAlignmentRole，使得默认委托指定一个特殊的 Qt::AlignmentFlag 对象。Qt::DecorationRole 用于在视图中修饰数据值的图标。通常，应该在这个角色中使用一个 QColor，QIcon 或者 QPixmap 类型的数据值。Qt::UserRole 及其以上角色可以用于自定义数据值。可以将它们看成是表格模型中附加的数据列。

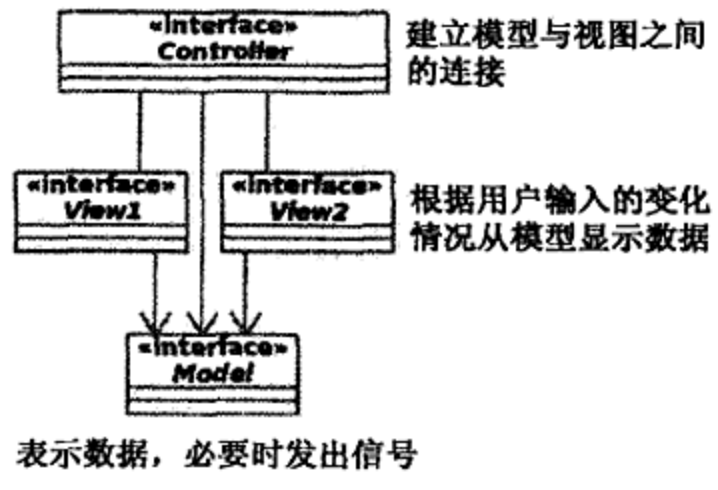


图 13.2 模型-视图-控制器类

如图 13.2 所示，模型-视图-控制器框架使用了多个设计模式，以支持多个视图使用相同数据的应用开发。它显示了模型代码（负责维护数据）、视图代码（负责以不同方式显示所有或部分数据）和控制器代码(负责处理影响到数据以及模型的事件，比如委托）要放置在分离的类中。这种分离使得在添加或删除视图和控制器时模型不需要做任何改变。它使得多个视图能够保持同步并且与模型保持一致，即使数据同时在不同的视图中被交互编辑。允许模型或视图均可被替换，使代码复用得以最大化。

控制器类的主要目标是封装控制器代码。一个复杂的应用可能在不同的组件或层级中具有多个控制器。

在 Qt 中，不同的控制器类的基类是 QAbstractItemDelegate。那些连接信号与槽的 connect 语句也可以认为是控制器代码。如你所见，将控制器代码放在模型和视图类之外能够获得额外的设计益处。

13.2 Qt 模型和视图

Qt 中包含有一个模型/视图框架，用于维护数据的组织管理和向用户的呈现方式之间的分离。三个最常用的视图类（列表、树和表格）都是默认提供的。另外，它还提供了抽象的和具体的数据模型，这些数据模型可被扩展和自定义以保存不同类型的数据。在应用中将一个模型以不同的方式同时呈现的情况比比皆是。

视图是获取、修改和呈现数据的对象。图 13.3 展示了 Qt 模型-视图框架中的四种视图。

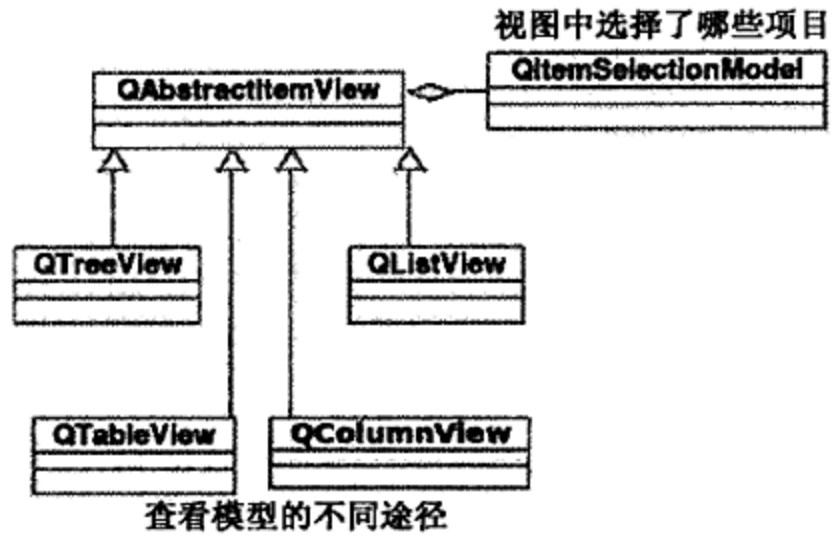


图 13.3　Qt 视图类

QAbstractItemModel 定义了视图(还有委托)访问数据的一个标准接口。模型中存储供显示和操作(例如排序、编辑、保存、获取、转换等)的具体数据。通过信号和槽，它们将数据的变化通知给所有相关联的视图。每个视图对象都有一个指向模型对象的指针。视图对象会频繁访问项模型的方法以获取或设置数据，或者做各种其他操作。图 13.4 展示了设计用来用各种视图类紧密配合的模型类。

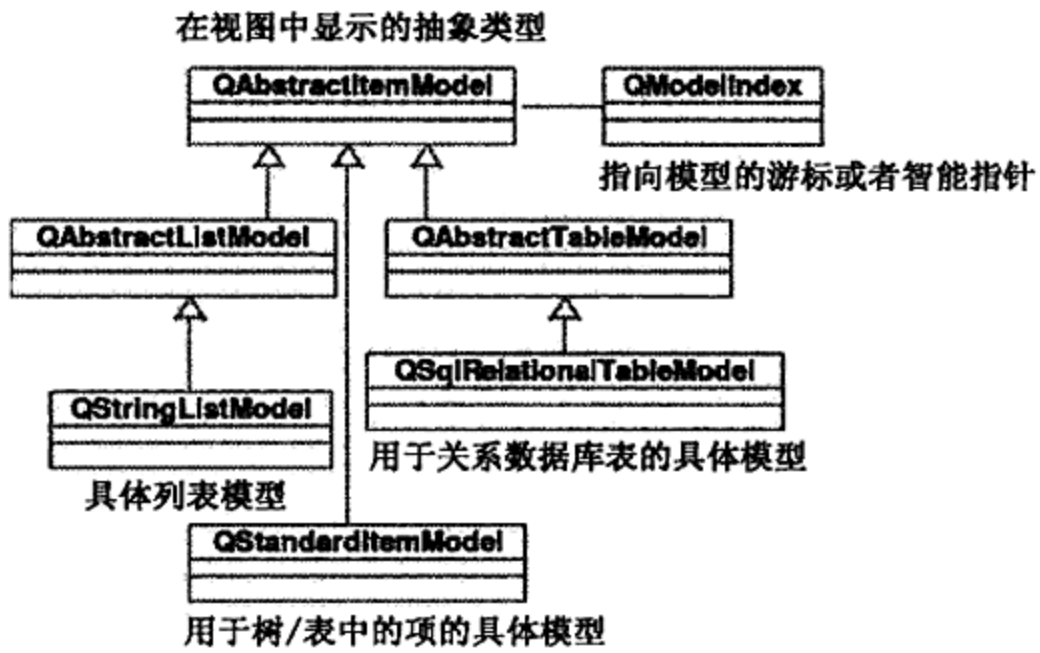


图 13.4　Qt 模型类

选择模型是用来描述在模型中被视图选中的那些数据项的对象。QModelIndex 的行为就像一个游标或智能指针，提供了统一的方式来遍历模型中的列表、树或表格项。在 setModel()方法调用之后，每当模型发生变化时视图会自动更新(假设模型是正确的)。

有两种方式来实现数据模型，每种方式都有各自的特性。

1. 重新实现 QAbstractItemModel 的被动接口，包括数据呈现。
2. 复用一个通用用途的具体的数据模型，比如 QStandardItemModel，填充其数据。

QAbstractItemModel 中的那些接口在实现上提供了更多的灵活性。使用为特定的访问模式或数据分布方式优化过的数据结构也是可行的。

第二种方式中复用了 QStandardItem(Model)类，使得可以用类似于使用 QListWidget，QTableWidget 和 QTreeWidget 的形式写树形或基于项的代码。

视图

QAbstractItemView 为这三个不同的模型类型的通用特性提供了接口：

- 各种布局的列表。
- 表格，或许带有交互元素。
- 表征父-子结构的树。

模型索引

模型中的每个数据项都用一个模型索引来表示。模型索引为视图和委托提供了在不知道其底层数据结构的情况下间接访问模型中数据项的方法。只有模型需要知道如何直接访问这些数据。QModelIndex 类提供了一个用于索引和访问派生自 QAbstractItemModel 的模型类中数据的接口。在列表、表格和树这几种模式下它工作得相当好。每一个索引都有一个指针指向创建它的模型，在具有层次关系的数据结构中(比如树)还可能有一个父项索引。QModelIndex 对待模型数据仿佛它们被安排在一个具有行和列索引的二维数组中一样，无论底层拥有这些数据是何种数据结构。

QModelIndex 对象由模型创建，可以被模型、视图或委托代码用于定位数据模型中的特定项。QModelIndex 对象具有很短的生命周期，可能在刚刚创建后就变成无效的状态，因此它们应该被立即使用而后丢弃。

如果使用一个在若干指令操作前已经存在的 QModelIndex，那么应该先调用 QModelIndex::isValid()。QPersistentModelIndex 对象具有更长的生命周期，但是在使用前仍应该先调用 isValid()方法来进行检查。

13.2.1 QFileSystemModel

QFileSystemModel 可以以列表、表格或树形视图呈现。图 13.5 展示了一个在 QTreeView 中显示的 QFileSystemModel 对象。

QFileSystemModel 早已将数据准备好了，所以可以简单地创建一个 QFileSystemModel 对象，创建一个视图，然后调用 view->setModel(model)。示例 13.1 展示了一个可能是最简单的 Qt 模型-视图示例。

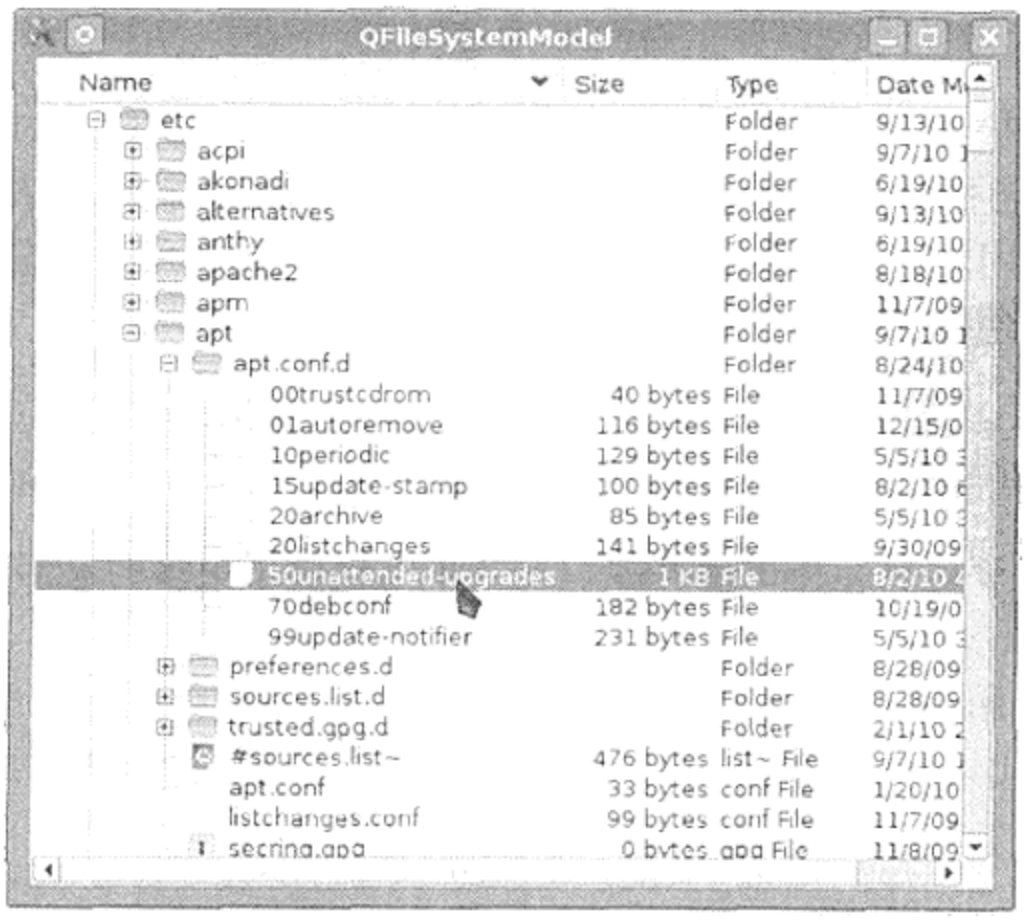


图 13.5 QTreeView 中的 QFileSystemModel

示例 13.1　src/modelview/filesystem/main.cpp

```
#include <QtGui>
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QFileSystemModel model;
    model.setRootPath("/");
    QTreeView tree;
    tree.setModel(&model);
    tree.setSortingEnabled(true);                                          1
    tree.header()->setResizeMode(QHeaderView::ResizeToContents);
    tree.resize(640, 480);
    tree.show();
    return app.exec();
}
```

1 使表头(HeaderView)的排序按钮可用。

通过设置 HeaderView 的 resizeMode 属性，表格或树中的列无论窗体大小是多少，都能够进行宽度调整。这些类是构建一个文件浏览部件的基本组成部分。

13.2.2　多重视图

示例 13.2 是一个用于创建可以以树、表格或简单的列表形式进行查看的 QStandard-ItemModel 的函数。这个函数也演示了 QStandardItem 类的使用。

示例 13.2　src/modelview/multiview/createModel.cpp

```
#include <QtGui>

QStandardItemModel* createModel(QObject* parent, int rows,
                                int cols, int childNodes) {
    QStandardItemModel*
        model = new QStandardItemModel(rows, cols, parent);
    for( int r=0; r<rows; r++ )
        for( int c=0; c<cols; c++)  {
            QStandardItem* item = new QStandardItem(
                QString("Row:%0, Column:%1").arg(r).arg(c) );
            if( c == 0 )                                                   1
                for( int i=0; i<childNodes; i++ ) {
                    QStandardItem* child = new QStandardItem(
                                                QString("Item %0").arg(i) );
                    item->appendRow( child );
                }
            model->setItem(r, c, item);
        }
    model->setHorizontalHeaderItem( 0, new QStandardItem( "Name" ));
    model->setHorizontalHeaderItem( 1, new QStandardItem( "Value" ) );
    return model;
}
```

1 给第一列元素添加子节点。

示例 13.3 中所示的主程序创建了四个不同的视图：QListView，QTableView，QTreeView 和 QColumnView。注意，QTableView 和 QListView 并不显示子节点，另外，

QColumnView 和 QListView 也不显示表格模型中的非首列。QColumnView 在右边显示当前选中节点下的树子节点，这与 MacOS X Finder 中显示一个选中目录下的文件的方式相似。

所有的视图共享同一个模型，所以在一个单元格中编辑后立即会在其他视图中显示出来。另外，所有的视图共享一个选择模型，所以选择状态在四个视图中也是同步的。

示例 13.3 src/modelview/multiview/multiview.cpp

```
[ . . . . ]

#include "createModel.h"

int main( int argc, char** argv ) {
  QApplication app( argc, argv );
  QStandardItemModel* model = createModel(&app);
  QSplitter vsplitter(Qt::Vertical);
  QSplitter hsplitter;                                          1

  QListView list;
  QTableView table;
  QTreeView tree;
  QColumnView columnView;
  [ . . . . ]

  list.setModel( model );
  table.setModel( model );
  tree.setModel( model );                                       2
  columnView.setModel (model);
  [ . . . . ]

  list.setSelectionModel( tree.selectionModel() );
  table.setSelectionModel( tree.selectionModel() );             3
  columnView.setSelectionModel (tree.selectionModel());
  table.setSelectionBehavior( QAbstractItemView::SelectRows );
  table.setSelectionMode( QAbstractItemView::SingleSelection );
```

1 默认的，子项水平布局。

2 共享同一个模型。

3 公共选择模型。

当执行这段代码时，你会看到如图 13.6 所示的窗口。需要注意的是，在视图中选择一项时会导致在其他视图中该项也会被选中。因为我们使用的是具体的 QStandardItemModel，模型项对每个视图来说都是可编辑的。此外，从任何一个视图产生的变化都会自动影响到其他视图，从而确保每个视图与模型是统一的。

根据 QAbstractItemView::EditTriggers 的设置情况，可以通过 F2 键、双击鼠标键或者仅仅进入到某个单元格来触发编辑状态。你或许注意到了在本地 windows 环境中熟悉的其他快捷键(剪切、复制、粘贴、Ctrl+光标键等)在视图和编辑区域中同样可以使用。

对于这个应用来说，我们使用了 QSplitter 部件。QSplitter 具有一些布局方面的特性，但是所管理的窗件是它的子部件。分隔栏部件允许用户在运行时通过拖动子窗件之间的分隔条来调整子窗件所占据空间的大小。示例 13.4 包含了建立分隔部件的代码。

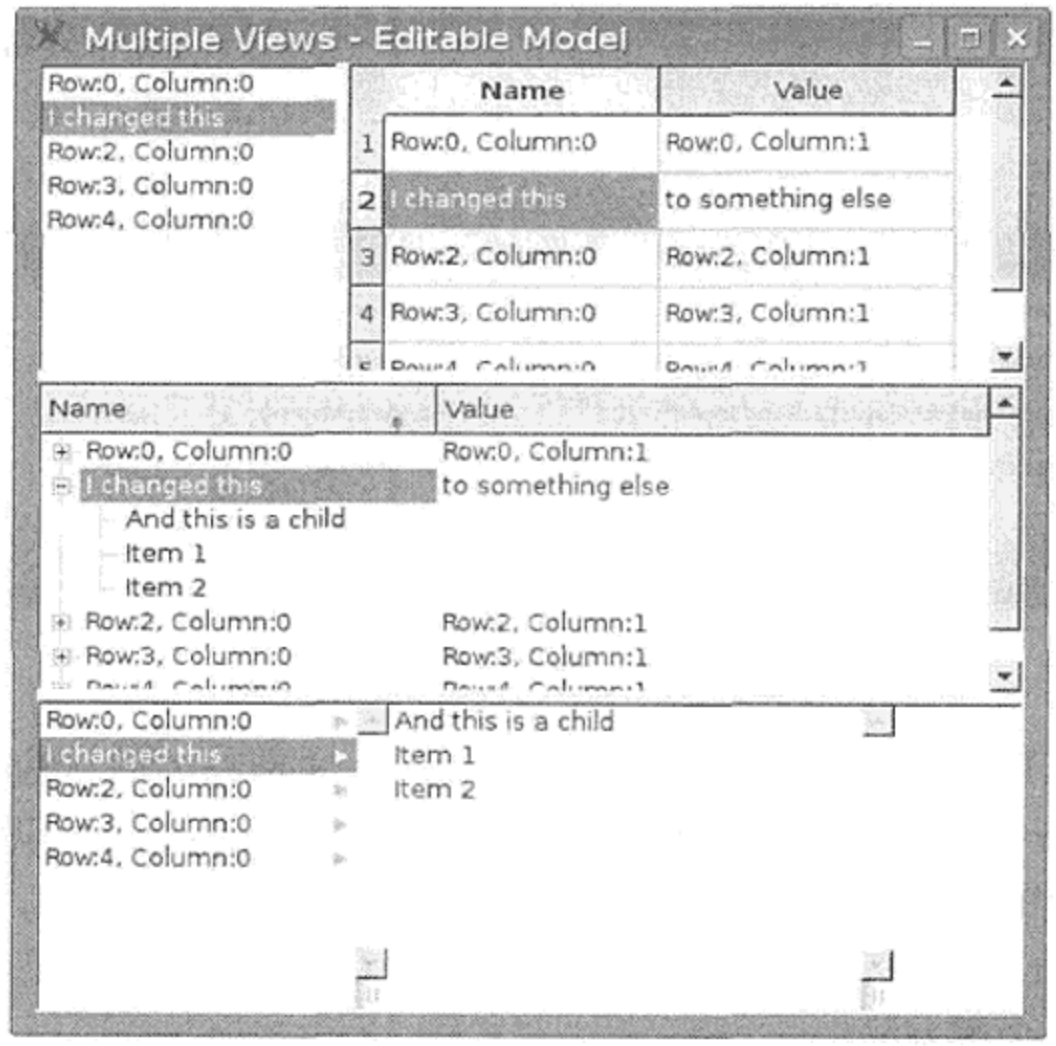


图 13.6　多重视图，同一个模型

示例 13.4　src/modelview/multiview/multiview.cpp

```
[ . . . . ]

  hsplitter.addWidget( &list );
  hsplitter.addWidget( &table );
  vsplitter.addWidget( &hsplitter );
  vsplitter.addWidget ( &tree );
  vsplitter.addWidget ( &columnView );

  vsplitter.setGeometry(300, 300, 500, 500);
  vsplitter.setWindowTitle("Multiple Views - Editable Model");
```

13.2.3　委托类

如图 13.7 所示，委托在模型和视图之间提供了另一层间接关联，它提升了自定义的可行性。

委托类通常派生于 QAbstractItemDelegate，为 Qt 的模型-视图框架中添加若干控制器特性。委托类能够提供一个工厂方法，以使得视图类能够创建编辑器以及虚函数 getter 和 setter，从模型中获取数据或者保存数据到模型中。它还能够提供一个 paint() 虚方法以自定义视图中项的显示。委托还能够设置为应用于整个 QAbstractItemView 或者仅仅应用于一列。

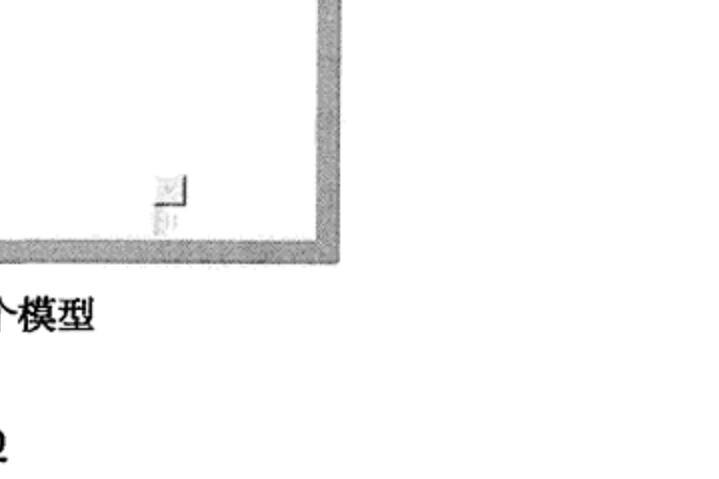


图 13.7　模型、视图和委托

图 13.8 展示了一个来自于 $QTDIR/examples/modelview/stardelegate 的 StarDelegate 示例的修改版。

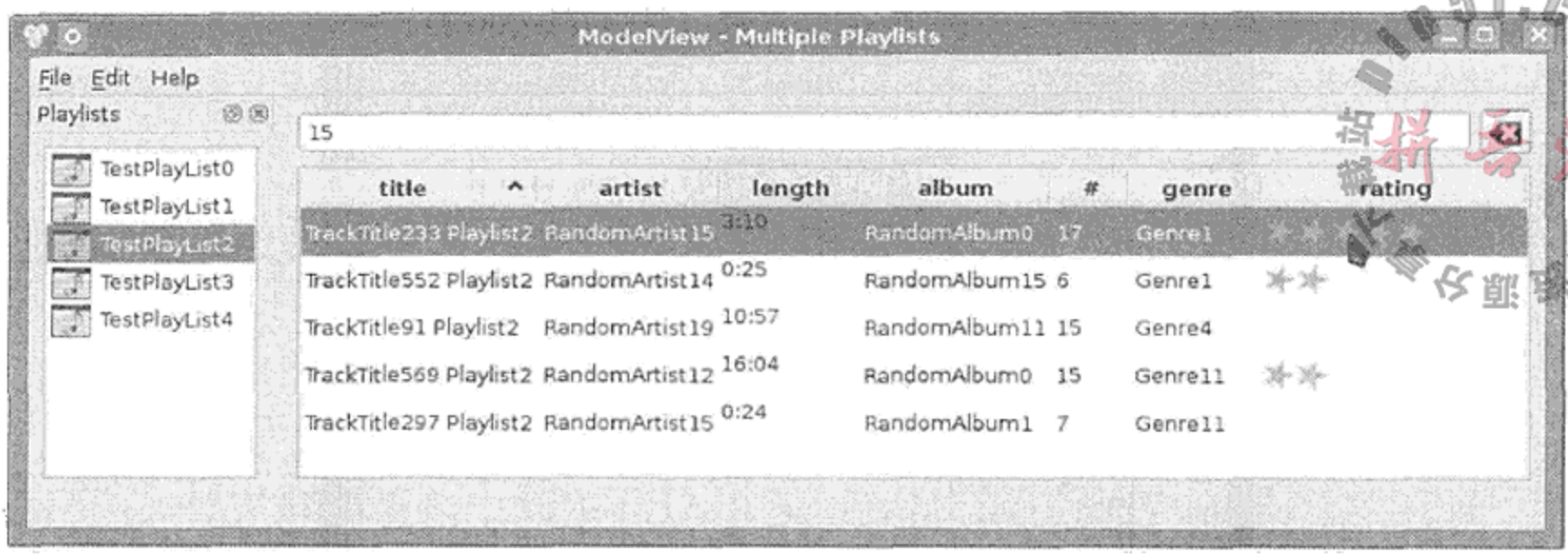


图 13.8 StarDelegate 示例

示例 13.5 中扩展了 QStyledItemDelegate。这是一个默认应用于 QListView，QTableView 和 QTreeView 中的具体类。它提供了一个 QLineEdit 以编辑 QString 类型属性，以及其他适合的编辑器以用于类型 boolean，QDate，QTime，int 或 double。对自定义的 StarRating 值，该自定义委托展示了要在表格中获得“星号”而不是简单的整数值，则必须重写一个虚方法。

示例 13.5 src/modelview/playlists/stardelegate.h

```
#ifndef STARDELEGATE_H
#define STARDELEGATE_H

#include <QStyledItemDelegate>
#include <QStyleOptionViewItem>
class StarDelegate : public QStyledItemDelegate {
    Q_OBJECT
public:
    typedef QStyledItemDelegate SUPER;
    StarDelegate(QObject* parent=0) : SUPER(parent) {};
    QWidget* createEditor(QWidget* parent,
                          const QStyleOptionViewItem& option,
                          const QModelIndex& index) const;
    void paint(QPainter* painter,
               const QStyleOptionViewItem& option,
               const QModelIndex& index) const;

    void setEditorData(QWidget* editor,
                       const QModelIndex& index) const;
    void setModelData(QWidget* editor,
                      QAbstractItemModel* model,
                      const QModelIndex& index) const;
};

#endif // STARDELEGATE_H
```

通过重写 paint()方法，委托为视图中的项如何显示提供了全面的控制，如示例 13.6 所示。

示例 13.6　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    paint(QPainter* painter,
          const QStyleOptionViewItem& option,
          const QModelIndex& index) const {
    QString field = index.model()->headerData(index.column(),
                                              Qt::Horizontal).toString();
    if (field == "length") {
        QVariant var = index.data(Qt::DisplayRole);
        Q_ASSERT(var.canConvert(QVariant::Time));
        QTime time = var.toTime();
        QString str = time.toString("m:ss");
        painter->drawText(option.rect, str, QTextOption());
        // can't use drawDisplay with QStyledItemDelegate:
        // drawDisplay(painter, option, option.rect, str);
        return;
    }
    if (field != "rating") {
        SUPER::paint(painter, option, index);
        return;
    }
    QVariant variantData = index.data(Qt::DisplayRole);
    StarRating starRating = variantData.value<StarRating>();
    if (option.state & QStyle::State_Selected)
        painter->fillRect(option.rect, option.palette.highlight());
    starRating.paint(painter, option.rect, option.palette,
                     StarRating::ReadOnly);
}
```

另外，委托还能在用户触发一个项的编辑状态时，确定应显示何种类型的部件。为此，可以如示例 13.7 所示重载 `createEditor()`，或者提供一个自定义的 `QItemEditorFactory`。

示例 13.7　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

QWidget* StarDelegate::
    createEditor(QWidget* parent,
                 const QStyleOptionViewItem& option,
                 const QModelIndex& index) const {
    QString field = index.model()->headerData(index.column(),
                                      Qt::Horizontal).toString();
    if (field == "rating")  {
            return new StarEditor(parent);
    }
    if (field == "length") {
        return new TimeDisplay(parent);
    }
    return SUPER::createEditor(parent, option, index);
}v
```

什么是编辑触发器

有各种方式来“触发”视图使其进入编辑模式。在大多数桌面平台中，F2 键是“平台编辑”键。在手持设备上，它可能是一个手势，如两次轻触(double-tap)或一个特殊的按钮。请查看 `QAbstractItemView::setEditTriggers(EditTriggers triggers)` 的 API 文档 http://doc.qt.nokia.com/latest/qabstractitemview.html#editTriggers-prop。

当触发一个编辑请求时，我们想要看到这样一个编辑器，其初始化值来自于你想查看或修改的模型。这是由 `setEditorData()` 方法完成的，如示例 13.8 所示。

示例 13.8 src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]
void StarDelegate::
    setEditorData(QWidget* editor,
                  const QModelIndex& index) const {
    QVariant val = index.data(Qt::EditRole);
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);          1
    if (starEditor != 0) {
        StarRating sr = qVariantValue<StarRating>(val);                  2
        starEditor->setStarRating(sr);
        return;
    }

    TimeDisplay* timeDisplay = qobject_cast<TimeDisplay*>(editor);       3
    if (timeDisplay != 0) {
        QTime t = val.toTime();
        timeDisplay->setTime(t);
        return;
    }
    SUPER::setEditorData(editor, index);                                 4
    return;
}
```

1 动态类型检查。

2 从 `QVariant` 中抽取用户定义类型值。

3 动态类型检查。

4 让基类来处理其他类型。

当用户完成编辑后，如示例 13.9 所示会调用 `setModelData()` 方法以将数据写回到 `QAbstractItemModel`。

示例 13.9 src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    setModelData(QWidget* editor, QAbstractItemModel* model,
                 const QModelIndex& index) const {
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);
```

```
    if (starEditor != 0) {
        StarRating r = starEditor->starRating();
        QVariant v;
        v.setValue<StarRating>(r);
        model->setData(index, v, Qt::EditRole);
        return;
    }
    TimeDisplay* td = qobject_cast<TimeDisplay*>(editor);
    if (td != 0) {
        QTime t = td->time();
        model->setData(index, QVariant(t));
        return;
    }
    SUPER::setModelData(editor, model, index);
    return;
}
```

13.3　表格模型

下一个示例如图 13.9 所示，是一个使用表格模型的能够显示和编辑动作和对应快捷键的视图。为了演示 Qt 模型-视图类中所支持的不同数据角色的使用和显示，我们获取了一个 QAction 对象列表并将它显示在一个表格中。每一个动作都可以有一个图标、工具提示、状态提示以及其他用户数据。这直接对应到了四种可用数据角色。

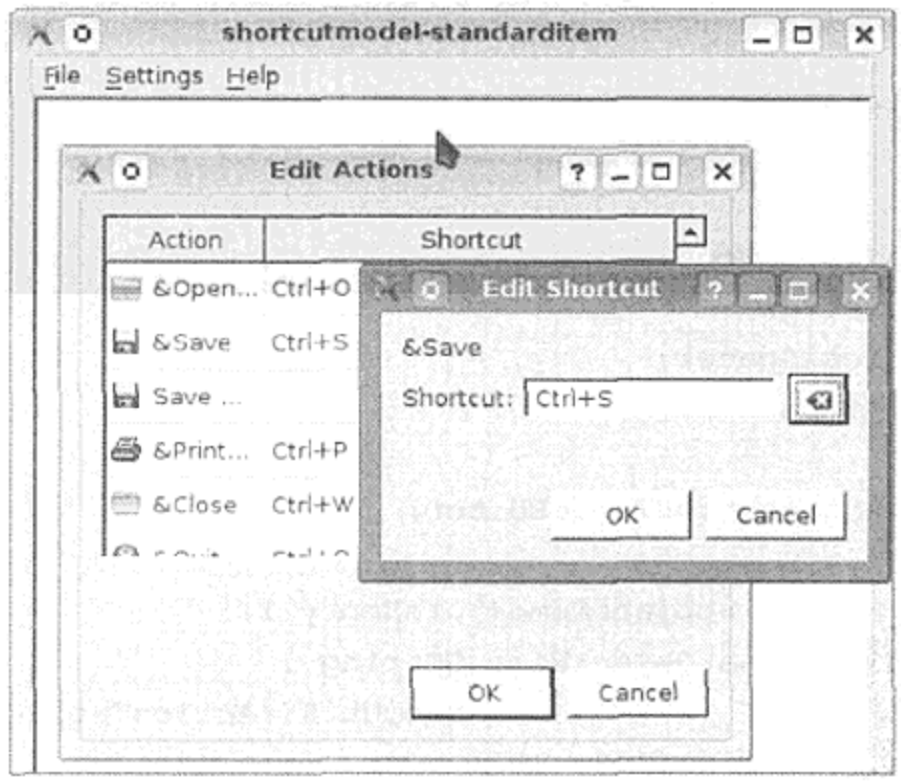


图 13.9　快捷键编辑器

13.3.1　标准模型与抽象模型的比较

当开发者刚开始熟悉 QStandardItem 时，他们有时会将其应用到一些或许并非是最佳选择的场景中。尽管 QStandardItemModel 使得无须从一个抽象基类派生即可轻易建立一个模型，如果你关注到 QStandardItem 中的数据与其他内存数据无法保持同步，或者用它创建标准项的初始化过程太繁琐，这就表明应该采用从 QAbstractItemModel 直接或间接派生的方式来建立模型。

示例 13.10 是一个基于 QStandardItem 类的快捷键表格模型。

示例 13.10 src/modelview/shortcutmodel-standarditem/actiontableeditor.h

```
[ . . . . ]
class ActionTableEditor : public QDialog {
    Q_OBJECT
public:
    ActionTableEditor(QWidget* parent = 0);
    ~ActionTableEditor();
protected slots:
    void on_m_tableView_activated(const QModelIndex& idx=QModelIndex());
    QList<QStandardItem*> createActionRow(QAction* a);
protected:
    void populateTable();
    void changeEvent(QEvent* e);
private:
    QList<QAction*> m_actions;
    QStandardItemModel* m_model;
    Ui_ActionTableEditor* m_ui;
};
[ . . . . ]
```

因为这是一个采用 Qt 界面设计师创建的窗口，所以自动生成的部件的创建和初始化代码看起来像示例 13.11 所示的样子。

示例 13.11 src/modelview/shortcutmodel-standarditem/actiontableeditor_ui.h

```
[ . . . . ]
class Ui_ActionTableEditor
{
public:
  QVBoxLayout *verticalLayout;
  QTableView *m_tableView;
  QSpacerItem *verticalSpacer;
  QDialogButtonBox *m_buttonBox;

  void setupUi(QDialog *ActionTableEditor)
  {
    if (ActionTableEditor->objectName().isEmpty())
      ActionTableEditor->setObjectName(QString::
                                       fromUtf8("ActionTableEditor"));
    ActionTableEditor->resize(348, 302);
    verticalLayout = new QVBoxLayout(ActionTableEditor);
    verticalLayout->setObjectName(QString::fromUtf8("verticalLayout"));
    m_tableView = new QTableView(ActionTableEditor);
    m_tableView->setObjectName(QString::fromUtf8("m_tableView"));
    verticalLayout->addWidget(m_tableView);
[ . . . . ]
```

示例 13.12 展示了如何在 QStandardItemModel 中创建数据行，每一行是一个 QAction 对象。

示例 13.12　src/modelview/shortcutmodel-standarditem/actiontableeditor.cpp

```
[ . . . . ]

QList<QStandardItem*> ActionTableEditor::
createActionRow(QAction* a) {
    QList<QStandardItem*> row;
    QStandardItem* actionItem = new QStandardItem(a->text());
    QStandardItem* shortcutItem =
        new QStandardItem(a->shortcut().toString());                    1
    actionItem->setIcon(a->icon());                                     2

    actionItem->setToolTip(a->toolTip());                               3
    actionItem->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEnabled);     4
    shortcutItem->setFlags(Qt::ItemIsSelectable| Qt::ItemIsEnabled);
    shortcutItem->setIcon(a->icon());                                   5
    row << actionItem << shortcutItem;
    return row;
}

void ActionTableEditor::populateTable() {
    foreach (QWidget* w, qApp->topLevelWidgets())                       6
        foreach (QAction* a, w->findChildren<QAction*>()) {             7
            if (a->children().size() > 0) continue;                     8
            if (a->text().size() > 0) m_actions << a;
        }

    int rows = m_actions.size();
    m_model = new QStandardItemModel(this);
    m_model->setColumnCount(2);
    m_model->setHeaderData(0, Qt::Horizontal, QString("Action"),
                           Qt::DisplayRole);
    m_model->setHeaderData(1, Qt::Horizontal, QString("Shortcut"),
                           Qt::DisplayRole);
    QHeaderView* hv = m_ui->m_tableView->horizontalHeader();
    m_ui->m_tableView->
       setSelectionBehavior(QAbstractItemView::SelectRows);
    m_ui->m_tableView->
       setSelectionMode(QAbstractItemView::NoSelection);
    hv->setResizeMode(QHeaderView::ResizeToContents);
    hv->setStretchLastSection(true);
    m_ui->m_tableView->verticalHeader()->hide();
    for (int row=0; row < rows; ++row ) {
        m_model->appendRow(createActionRow(m_actions[row]));
    }
    m_ui->m_tableView->setModel(m_model);                               9
}
```

1 从 QAction 复制数据到 QStandardItem。

2 复制更多的数据。

3 复制更多的数据。

4 只读模型，没有 Qt::ItemIsEditable 选项。

5 复制更多的数据。
6 所有顶层部件。
7 所有可被找到的 QAction 对象。
8 跳过群组动作。
9 将视图和模型关联起来。

QStandardItem 有它自己的属性，所以我们从每个 QAction 对象中复制数据值到两个相关项中。当工作于大型数据模型时，这种类型的复制工作会有严重影响性能和内存开销。其主要问题就是创建模型时它有着显著的开销，当完成它以后就会抛弃它。

这个示例并没有使用视图的编辑特性以修改模型中的数据。这需要写一个委托来提供自定义编辑器部件，并将作为一个练习留给读者(见 13.6 节的练习 4)。相反，当用户激活一行时，将弹出了一个如示例 13.13 所示的对话框。当对话框返回 Accepted 时，QAction 对象的快捷键将完全跳过这个模型被直接设置。下次再显示这个快捷键表格时，模型必须从动作列表重新生成，否则我们需要正确的处理变化。

示例 13.13 src/modelview/shortcutmodel-standarditem/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::
on_m_tableView_activated(const QModelIndex& idx) {
    int row = idx.row();
    QAction* action = m_actions.at(row);
    ActionEditorDialog aed(action);
    int result = aed.exec();                                    1
    if (result ==  QDialog::Accepted) {                         2
        action->setShortcut(aed.keySequence());                 3
        m_ui->m_tableView->reset();
    }
}
```

1 弹出模式对话框以供编辑动作的快捷键。
2 这将是检查重复的/不明确的绑定的好地方。
3 跳过模型直接设置 QAction 属性。

一种更好的方法

示例 13.14 是一个扩展自 QAbstractTableModel 的表格模型，它重新实现 data()和 flags()这两个纯虚方法以提供对模型数据的访问。模型是一个对已在内存中存在的 QAction 对象列表的代理，这意味着不再需要在模型数据中进行数据复制。

示例 13.14 src/libs/actioneditor/actiontablemodel.h

```
[ . . . . ]
class ACTIONEDITOR_EXPORT ActionTableModel : public QAbstractTableModel {
    Q_OBJECT
 public:
    explicit ActionTableModel(QList<QAction*> actions, QObject* parent=0);
    int rowCount(const QModelIndex&  = QModelIndex()) const {
        return m_actions.size();
```

```
    }
    int columnCount(const QModelIndex& = QModelIndex()) const {
        return m_columns;
    }
    QAction* action(int row) const;
    QVariant headerData(int section, Qt::Orientation orientation,          1
        int role) const;
    QVariant data(const QModelIndex& index, int role) const;               2
    Qt::ItemFlags flags(const QModelIndex& index) const;                   3
    bool setData(const QModelIndex& index, const QVariant& value,
        int role = Qt::EditRole);                                          4

protected:
    QList<QAction*> m_actions;
    int m_columns;
};
[ . . . . ]
```

1 可选重写。

2 必须重写。

3 必须重写。

4 对可编辑模型是必须的。

示例 13.15 展示了 data() 的实现。注意，对 QAction 对象的许多不同属性，这里都有一个相应的数据角色。可以将角色(尤其是用户角色)看成是一个数据附加列。

示例 13.15　src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

QVariant ActionTableModel::
data(const QModelIndex& index, int role) const {
    int row = index.row();
    if (row >= m_actions.size()) return QVariant();
    int col = index.column();
    if (col >= columnCount()) return QVariant();
    if (role == Qt::DecorationRole)
        if (col == 0)
            return m_actions[row]->icon();

    if (role == Qt::ToolTipRole) {
        return m_actions[row]->toolTip();
    }
    if (role == Qt::StatusTipRole) {
        return m_actions[row]->statusTip();
    }
    if (role == Qt::DisplayRole) {
        if (col == 1) return m_actions[row]->shortcut();
        if (col == 2) return m_actions[row]->parent()->objectName();
        else return m_actions[row]->text();
    }
    return QVariant();
}
```

从它无须创建/复制任何数据的意义上讲，ActionTableModel 是轻量级的。只有视图请求时它才会呈现数据给视图。这意味着有可能在数据模型底层实现一个松散的数据结构。这还意味着模型可以从另一个数据源以一种懒惰的方式获取数据，如同 QSqlTableModel 和 QFileSystemModel 那样。

13.3.2 可编辑模型

对于可编辑模型，必须重写 flags()和 setData()。如果想就地编辑(在真实视图中)，就需要在 flags()函数中返回 Qt::ItemIsEditable 标识。因为在某项被单击时依然会弹出一个 ActionEditorDialog，而不需要就地编辑，所以示例 13.16 中简单地返回了 Qt::itemIsEnabled。

示例 13.16 src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

Qt::ItemFlags ActionTableModel::
flags(const QModelIndex& index) const {
    if (index.isValid()) return Qt::ItemIsEnabled;
    else return 0;
}
```

示例 13.17 展示了在 setData()中如何在实际设置新值之前检查不明确的快捷键。数据修改之后，发射 dataChanged()信号是很重要的，这样还在显示旧数据的视图可以知道应该从模型中获取新数据了。

示例 13.17 src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

bool ActionTableModel::
setData(const QModelIndex& index, const QVariant& value, int role) {
    if (role != Qt::EditRole) return false;
    int row = index.row();
    if ((row < 0) | (row >= m_actions.size())) return false;
    QString str = value.toString();
    QKeySequence ks(str);
    QAction* previousAction = 0;

    if (ks != QKeySequence() ) foreach (QAction* act, m_actions) {
        if (act->shortcut() == ks) {
            previousAction = act;
            break;
        }
    }
    if (previousAction != 0) {
        QString error = tr("%1 is already bound to %2.").
                arg(ks.toString()).arg(previousAction->text());
        bool answer = QMessageBox::question(0, error,
                    tr("%1\n Remove previous binding?").arg(error),
                    QMessageBox::Yes, QMessageBox::No);
```

```
        if (!answer) return false;
        previousAction->setShortcut(QKeySequence());
    }
    m_actions[row]->setShortcut(ks);
    QModelIndex changedIdx = createIndex(row, 1);                    1
    emit dataChanged(changedIdx, changedIdx);                        2
    return true;
}
```


1 列 1(第二列)显示快捷键。

2 视图需要它以便知道什么时间什么内容要更新。

要支持数据行的插入/删除，有类似的信号 `rowsInserted()` 和 `rowsRemoved()` 存在，它们必须从实现函数 `insertRows()` / `removeRows()` 内发射。

在一个或多个快捷键修改后，将它们保存到 `QSettings` 对象中。示例 13.18 展示了如何对需要保存的 `QAction` 对象进行持续跟踪。

示例 13.18　src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::
on_m_tableView_activated(const QModelIndex& idx) {
    int row = idx.row();
    QAction* action = m_model->action(row);
    ActionEditorDialog aed(action);

    int result = aed.exec();
    if (result ==  QDialog::Accepted) {
        QKeySequence ks = aed.keySequence();
        m_model->setData(idx, ks.toString());
        m_changedActions << action;
    }
}
```

示例 13.19 展示了在用户在对话框中选择 Accepted 后如何保存这些快捷键到 `QSettings` 对象中。

示例 13.19　src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::accept() {
    QSettings s;
    s.beginGroup("shortcut");
    foreach (QAction* act, m_changedActions) {
        s.setValue(act->text(), act->shortcut() );
    }
    s.endGroup();
    QDialog::accept();
}
```

13.3.3 排序和过滤

不要劳神去查找用于创建图 13.10 中的 QLineEdit、清除按钮或二者之间的 connect 代码。它们都是在 Qt 设计师中定义的，并且由 uic 自动生成。

这得感谢 QSortFilterProxyModel，用少于 5 行的代码就可以为一个已存在的模型增加排序/过滤功能。图 13.11 展示了该代理是如何出现在视图和模型之间的。

图 13.10 过滤后的表格视图

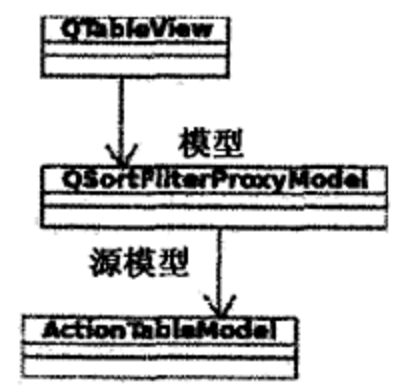


图 13.11 排序过滤代理

示例 13.20 展示了在前面的动作表格示例中设置排序过滤代理需要什么。

示例 13.20 src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::setupSortFilter() {
    m_sortFilterProxy = new QSortFilterProxyModel(this);
    m_sortFilterProxy->setSourceModel(m_model);                    1
    m_ui->m_tableView->setModel(m_sortFilterProxy);                2
    m_sortFilterProxy->setFilterKeyColumn(-1);                     3

}
void ActionTableEditor::on_m_filterField_textChanged               4
    (const QString& newText) {
    m_sortFilterProxy->setFilterFixedString(newText);              5
}
```

1 设置 SortFilterProxy 的源模型为 ActionTableModel。

2 表格视图的模型设置为代理模型而非 ActionTableModel。

3 所有数据列启用过滤。

4 自动连接的槽。

5 修改过滤字符串。

filterField_textChanged 是一个自动连接的槽，每当 QLineEdit 对象 filterField 的 textChanged 信号发射时会被调用。

13.4　树模型

在 QTreeView(具有父-子关系的项)中显示树形数据，有以下几个选项：

1. QAbstractItemModel 是一个可用于 QTreeView，QListView 或 QTableView 的通用抽象模型。
2. QStandardItemModel 是在示例 13.2 中使用过的可存储 QStandardItem 数据项的具体类，使得填充一个树形节点具体模型十分便利。
3. QTreeWidgetItem 不是一个模型类，但是它能用在从 QTreeView 派生而来的 QTreeWidget 部件中构造树形结构。

WidgetItem 类

QTreeWidget 和 QTreeWidgetItem 类主要用在 Qt 设计师中填充视图项。其 API 与 Qt 3 中的做法相似，它们仅被推荐应用于简单类型的数据以及单个视图中。这是因为使用基于部件/项的类，将模型与视图进行分离，或者当数据变化时多个视图自动更新都是不可能的。

QStandardItemModel 和 QTreeWidgetItem 类都是可被实例化或者进行扩展的树形节点。单个对象以树的形式连接起来，与 QObject 对象的子对象(参见 8.2 节)或者 QDomNode 节点(参见 15.3 节)相似。事实上，这些类都是组合模式的实现。

图 13.12 是下一个示例的截图，它给出当前内存中组成该应用程序用户界面的对象。

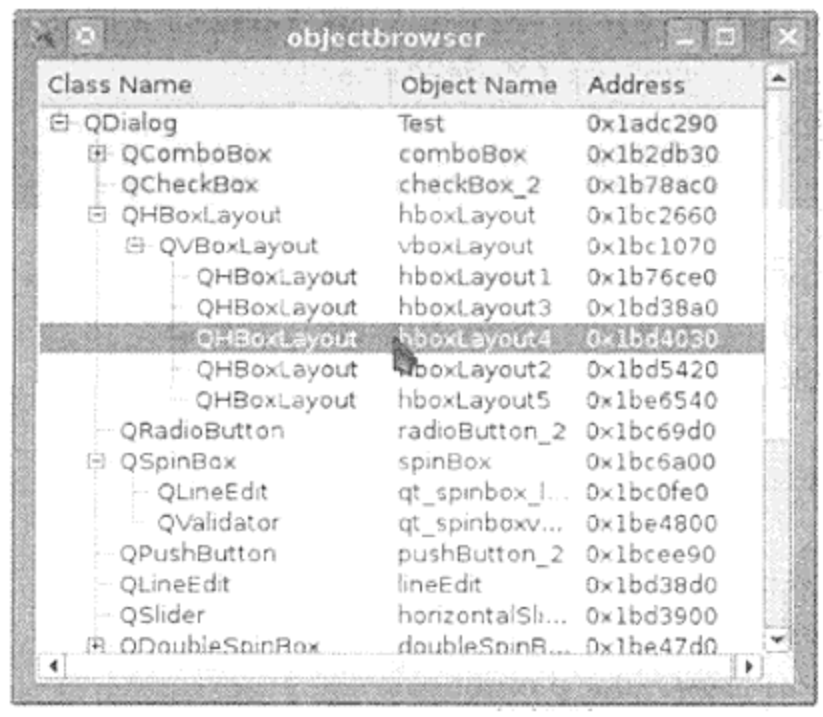


图 13.12　ObjectBrowser 树

这个应用程序的类定义在示例 13.21 中。ObjectBrowserModel 是一个扩展自 QAbstractItemModel 的具体树模型。它实现了一个只读对象浏览树所需的所有方法。

示例 13.21　src/modelview/objectbrowser/ObjectBrowserModel.h

```
[ . . . . ]
#include <QAbstractItemModel>
class ObjectBrowserModel :public QAbstractItemModel {
 public:
```

```
    explicit ObjectBrowserModel (QObject* rootObject);
    int columnCount ( const QModelIndex& parent = QModelIndex() ) const;
    int rowCount ( const QModelIndex& parent = QModelIndex() ) const;
    QVariant data ( const QModelIndex& index,
                    int role = Qt::DisplayRole ) const;
    QVariant headerData(int section, Qt::Orientation,
                        int role = Qt::DisplayRole) const;
    QModelIndex index ( int row, int column,
                        const QModelIndex& parent = QModelIndex()) const;
    QModelIndex parent ( const QModelIndex& index ) const;

protected:
    QList<QObject*> children( QObject* parent ) const;
    QString label( const QObject* widget, int column ) const;
    QObject* qObject( const QModelIndex& ) const;
private:
    QObject *rootItem;
};
[ . . . . ]
```

为了使树视图能够在 QAbstractItemModel 的父-子层次节点中上下移动,需要实现两个表格视图不需要的方法:index()和 parent()。示例 13.22 给出了它们的实现。

示例 13.22 src/modelview/objectbrowser/ObjectBrowserModel.cpp

```
[ . . . . ]

QModelIndex ObjectBrowserModel::
index(int row, int col, const QModelIndex& parent) const {
    if ((row < 0) || (col < 0) || row >= rowCount() ||
    col >= columnCount()) return QModelIndex();
    return createIndex( row, col, qObject(parent) );                    1
}

QModelIndex ObjectBrowserModel::parent( const QModelIndex& index ) const {
    if (!index.isValid()) return QModelIndex();
    QObject* obj = qObject(index)->parent();                            2
    if ( obj == 0 )
        return QModelIndex();

    QObject* parent = obj->parent();
    int row = children( parent ).indexOf( obj );
    return createIndex( row, 0, parent );
}

QObject* ObjectBrowserModel::
qObject(const QModelIndex& index) const {                               3
    if ( index.isValid() ) {
        QObject* parent = reinterpret_cast<QObject*>( index.internalPointer() );
        return children(parent)[index.row()];                           4
    }
    return 0;                                                           5
}
```

1 在索引中存储一个 internalPointer。
2 qObject()返回索引的直接行子项，但需要的是该索引的父项 QObject 指针，它存储在 index.internalPointer()。
3 索引的 internalPointer，父对象 QObject 指针。父项的直接子项，即 row()。
4 当前索引。
5 根索引。

index()也可以被想像成是“子索引”(childIndex)，因为它用于查找树中子项所对应的 QModelIndex，而 parent()用于在另一个方向上计算一步。在只应用于表格视图的模型中实现这些方法并不重要，但是如果希望在树形视图中查看这个模型的话，index()和 parent()就是必须的了。

13.4.1　Trolltech 模型测试工具

在模型中实现的方法会被一些尚未测试过的视图调用，或许用于尚未尝试过的情形。因为数据值常常是由用户驱动的，使用“模型测试”工具来测试模型或许很有帮助，它能快速地寻找通常的实现错误并且指出如何修复它们。

图 13.13 展示了在 QtCreator 调试器中使用 ModelTest 工具运行示例 13.21 中的 ObjectBrowser 时会发生的情况。错误断言和异常终止可能在栈踪迹中隐藏很深，所以需要一个调试器来查看全部栈踪迹。通常在终止行之前会有代码注释指明在该处正在执行什么样的测试。如示例 13.23 所示，它所包含的文件 Readme.txt 里面有使用 ModelTest 的简要指令说明。

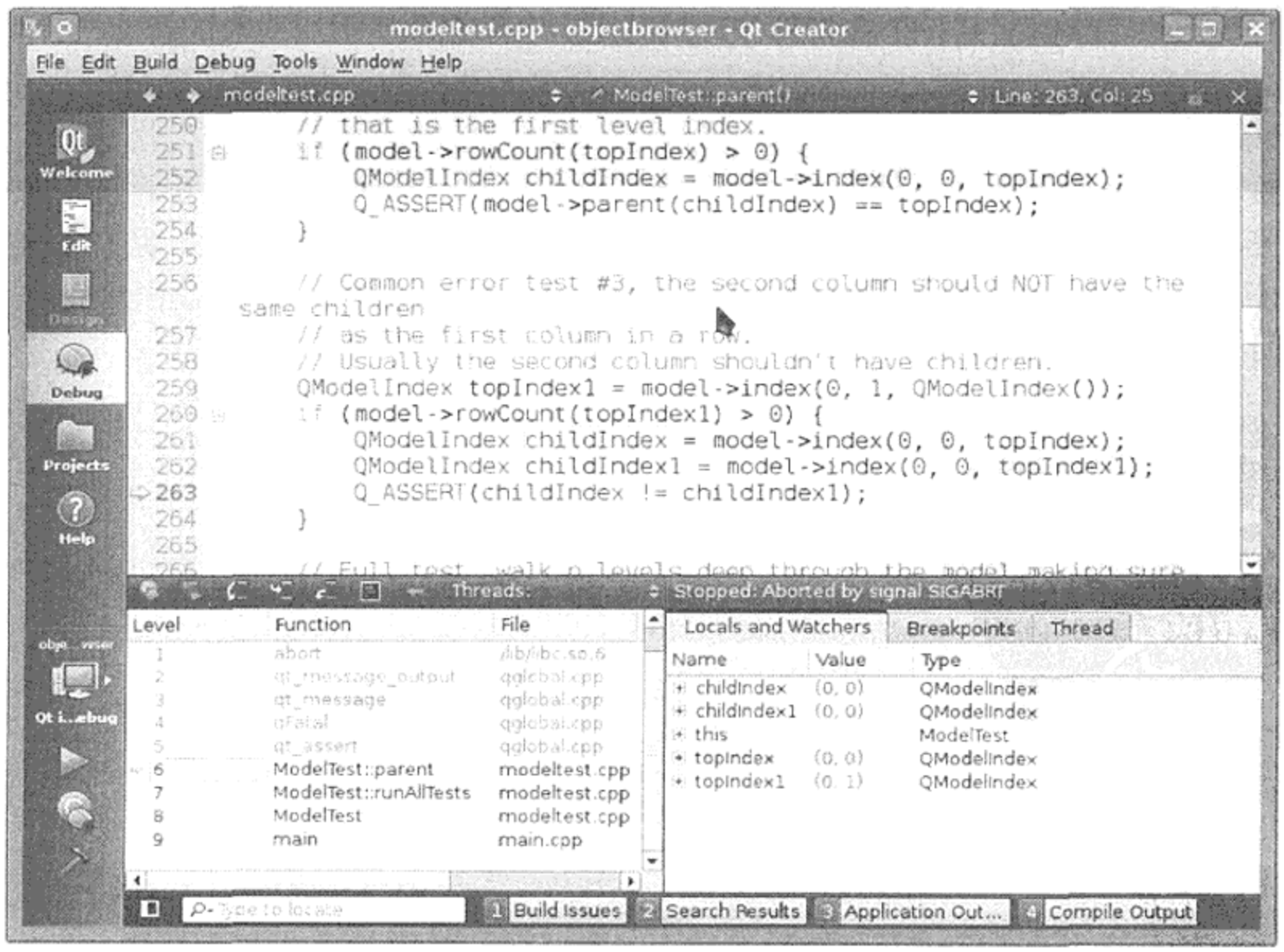


图 13.13　调试器中的 ModelTest

示例 13.23 src/libs/modeltest/readme.txt

应依照下面的指示来使用模型测试。

1) 像下面这样使用 include()命令将 pri 文件添加到工程文件.pro 中：

```
include(../path/to/dir/modeltest.pri)
```

2) 然后在源文件中包含 modeltest.h，并用模型初始化 ModelTest，这样测试才能在模型生存期内始终存在。例如：

```
#include <modeltest.h>

QDirModel *model = new QDirModel(this);
new ModelTest(model, this);
```

3) 这样就可以了。当测试找到问题时它会发出断言。

Modeltest.cpp 包含了一些关于如何修复该测试所找到问题的提示。

13.5 智能指针

尽管 C++不支持垃圾回收，但 C++对象的自动内存管理还是可以通过好几种方式来实现，主要是通过智能指针使用以及引用计数。Qt 提供了许多不同的智能指针类型，以适用于不同的用途。

一个重写了指针解引用操作 operator*()和 operator->()的类被称为智能指针。这使得类实例的行为就像它是一个内置指针一样。这样的类几乎总是模板类，因此定义时必须在模板参数中提供引用类型。最常见的能找到这些重写操作算子的地方是在迭代器以及智能指针中。使它们变得智能的通常是在构造、析构以及赋值中的自定义行为。

QScopedPointer 是一个在指针作用域结束后自动删除所引用对象的智能指针。它类似于 std::auto_ptr。复制 QScopedPointer 是毫无意义的，因为它会导致所引用的对象重复删除。指针的作用域明确地表明了所引用对象的生存期和所属。

类似于 QScopedPointer，QSharedPointer 是一个自动删除它所引用的对象的智能指针，但是它允许被复制，而且 QSharedPointer 会保持一个引用计数。共享的堆对象只有在最后一个指向它的智能指针销毁时才会被删除。示例 13.24 的 DataObjectTableModel 中使用了 QSharedPointer。

示例 13.24 src/libs/dataobjects/dataobjecttablemodel.h

```
[ . . . . ]

class DOBJS_EXPORT DataObjectTableModel : public QAbstractTableModel {
    Q_OBJECT
  public:
    explicit DataObjectTableModel(DataObject* headerModel = 0,
                                  QObject* parent=0);
    virtual bool insertRecord(QSharedPointer<DataObject> newRecord,
                              int position = -1,
                              const QModelIndex& = QModelIndex());
    QStringList toStringList() const;

    QString toString() const;
```

```
    virtual int fieldIndex(const QString& fieldName) const;
    virtual ~DataObjectTableModel();
[ . . . . ]

  public slots:
    void clear();
    void rowChanged(const QString& fileName);

  protected:
    QList<QSharedPointer<DataObject> > m_Data;
    QList<bool> m_isEditable;
    QStringList m_Headers;
    DataObject* m_Original;
    void extractHeaders(DataObject* hmodel);
  public:
    DataObjectTableModel& operator<<(QSharedPointer<DataObject> newObj) {
        insertRecord(newObj);
        return *this;
    }
};
```

可以通过智能指针间接地调用 DataObject 的 property()和 setProperty()方法，就像普通指针那样，使用 operator->的情形见示例 13.25。

示例 13.25　src/libs/dataobjects/dataobjecttablemodel.cpp

```
[ . . . . ]

QVariant DataObjectTableModel::
data(const QModelIndex& index, int role) const {
    if (!index.isValid())
        return QVariant();
    int row(index.row()), col(index.column());
    if (row >= rowCount()) return QVariant();
    QSharedPointer<DataObject> lineItem(m_Data.at(row));
    if (lineItem.isNull()) {
        qDebug() << "lineitem=0:" << index;
        return QVariant();
    }
    if (role == Qt::UserRole || role == Qt::ToolTipRole)
        return lineItem->objectName();
    else if (role == DisplayRole || role == EditRole) {
        return lineItem->property(m_Headers.at(col));
    } else
        return QVariant();
}


bool DataObjectTableModel::
setData(const QModelIndex& index, const QVariant& value, int role) {
    if (index.isValid() && role == EditRole) {
        int row(index.row()), col(index.column());
        QSharedPointer<DataObject> lineItem(m_Data.at(row));
```

```
            lineItem->setProperty(m_Headers.at(col), value);
            emit dataChanged(index, index);
            return true;
        }
        return false;
    }
```


如果每一行是由一个可在多个 DataObjectTableModel 中存在的 DataObject 对象所代表，通过使用带引用计数的指针，该表格在其拥有指向该对象的最后一个智能指针时可以清除这些 DataObject 对象。

13.6 练习：模型和视图

1. 写一个文件系统浏览器，具有地址栏、上一级按钮以及可选的可在其他浏览器中找到的其他常规按钮和特性。使用 QFileSystemModel 以及至少两个用 QSplitter 分隔开的视图类。一个视图要使用 QTableView。如图 13.14 所示，可以选择一个 Windows Explorer 样式树放于表格侧边，或者带 QColumnView 和表格的 Mac OS X 风格的浏览器。要使得能通过树/列视图、工具栏或者上一级按钮来选择一个目录，这样任意一种方式都能更新表格内容以反映新选择的目录。

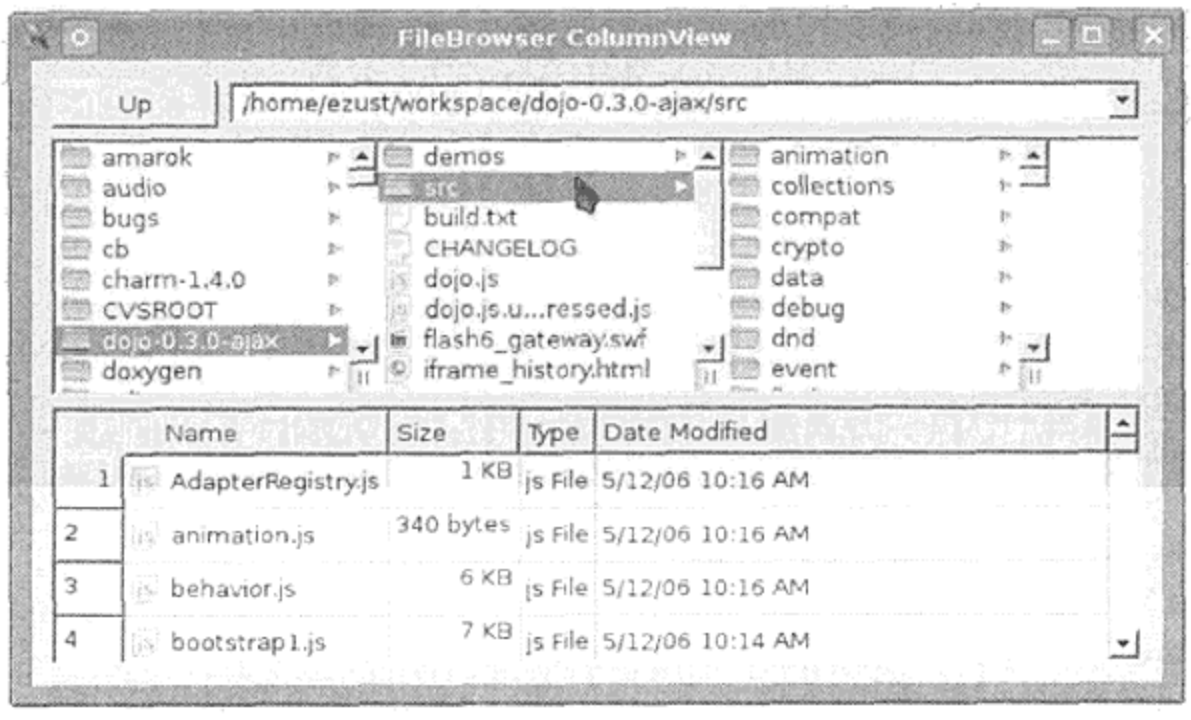


图 13.14 ColumnView 文件浏览器

2. 扩展 QAbstractTableModel 并且定义一个 PlayListModel，它应当表示一个 MetaDataValue(或者 MetaDataObject)对象列表。可以选择基于音轨或视频来设计。生成并显示测试数据，可以使用真实多媒体文件和MetaDataLoader或者使用自己的测试数据/工厂方法。实现加载/保存播放列表到磁盘的动作。
3. 重新温一下 11.6 节中的程序，实现一个显示好友列表或符号之间双向关系的界面，该界面应当显示两个 QListView 或者 QListWidget，就像图 13.15 所示那样。

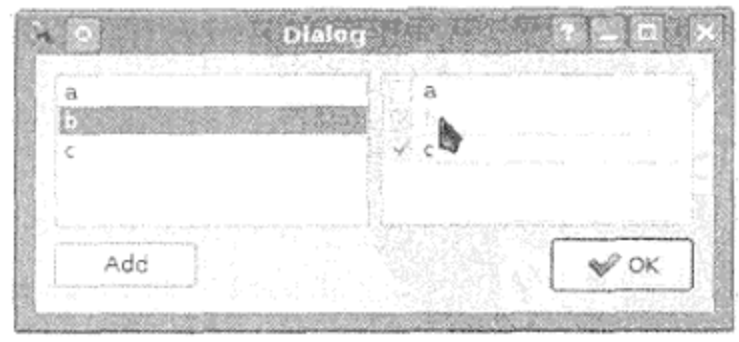


图 13.15 好友列表

- 两个列表都要显示可用符合集。
- 单击 Add 按钮，在两个列表中都添加一个新符号。
- 当左边的符号被选中时，右边列表中的好友显示为选中，陌生人置为未选中。
- 选中/取消右边的复选框，则添加/删除两个人之间的关系。
- 如果联系人与左侧选中的是同一个，则不允许用户取消选择它。始终将其显示为选中。

4. 重写快捷键编辑器示例来使用就地编辑。需要为表格视图写一个委托类，它提供一个自定义编辑器以供用户用来修改快捷键。

13.7　复习题

1. 什么是模型类？什么是视图类？在它们之间应保持什么样的关系？
2. Qt 提供了什么样的工具来与模型和视图共同工作？
3. 什么是 MVC？
4. 什么是控制器代码？哪些 Qt 类是控制器类？
5. 什么是委托？它们在哪里能发现？
6. 与委托相关的东西，角色有什么作用？
7. 如何判断一个 QListView 中的项是否被选中？
8. 如果想遍历一个 QAbstractItemModel 中的项，应该使用哪个类？
9. 有两套分层结构的类可用于存储和显示树形数据：*Widget/Item 和*ItemModel/View。使用其中一个而不是另一个的缘由是什么？
10. 为什么愿意使用 QStandardItemModel 而不是 QAbstractItemModel？反过来呢？

第 14 章　验证和正则表达式

输入数据通常需要进行过滤或者验证以确保输入是合适的。数字型数据一般需要在一定的范围内。字符串数据通常必须要有一定的格式。输入数据的验证是一项重要的课题，可以按照面向对象的方式使用各 Qt 类的组合来进行处理。本章将讨论一些对输入数据进行验证的有效方法，其中还包括了正则表达式(regular expression)的用法。

验证器(validator)和输入掩码(input mask)让开发人员可以精细地控制 QLineEdit 的行为，可用于对特定的输入类型进行限制。在使用它们之前需要记住的是，Qt 已经为许多常见的值类型提供了一些预定义的输入窗件和输入对话框(QDateEdit, QTimeEdit, QCalendarWidget, QColorDialog, QSpinBox 和 QDoubleSpinBox)，这在让开发人员限制输入值和缩小有效值范围的同时，允许用户轻松选择数值。对于 QDateEdit 来说，开发人员还可以从一系列本地化的日期格式中进行选用。参阅图 9.9，或者运行 Qt 的标准对话框示例程序(Standard Dialogs Example)来感受一下。

14.1　输入掩码

所谓的输入掩码，是一种控制用户在输入窗件中可键入的内容的主动模式。它有助于防止输入某些类型不正确的数据。每个 QLineEdit 都有一个 QString 属性用来存储掩码字符(mask character)——用于那些键入的数据上。输入掩码可以指定在键入 QLineEdit 的字符串中哪个位置处的何种字符是允许的。该字符串由一些特殊的、预定义的掩码字符和一些占据输入字符串相应位置的(可选的)普通字符构成。

表 14.1 中列出的小写版本的掩码字母，详细说明了在该位置上允许但并非必须的相应输入字符。使用 0 而不是 9 表明该位置允许但并非必须有一个 ASCII 数字。#表明该位置允许但并非必须有一个 ASCII 数字或者一个加号(+)或一个减号(–)。此外，表 14.2 中还给出了一些元字符(meta character)。

表 14.1　掩码字符

字　符	相应位置处所需字符	字　符	相应位置处所需字符
A	ASCII 字母型字符——大写或者小写	9	ASCII 数字
N	ASCII 字母数字型字符——大写或者小写	H	十六进制数字
X	任意的 ASCII 字符	B	二进制数字
D	ASCII 非零数字		

表 14.2　掩码元字符

字　符	效　果
>	随后的字母字符是大写的
<	随后的字母字符是小写的
!	结束大小写转换
\	转义字符

为了演示输入掩码的用法，下面给出一个简短的应用程序，它允许用户指定输入掩码字符，然后看看是如何限制输入的。示例 14.1 显示了该类的定义。

示例 14.1　src/validate/inputmask/masktestform.h

```
[ . . . . ]
class MaskTestForm : public QWidget {
   Q_OBJECT
public:
   MaskTestForm();
public slots:
   void showResult();
   void installMask();
   void again();
private:
   QLineEdit* m_InputMask;
   QLineEdit* m_StringEntry;
   QLabel* m_Result;
   void setupForm();
};
[ . . . . ]
```

示例 14.2 中实现了这个类。

示例 14.2　src/validate/inputmask/masktestform.cpp

```
[ . . . . ]
MaskTestForm::MaskTestForm(): m_InputMask(new QLineEdit),
    m_StringEntry(new QLineEdit), m_Result(new QLabel) {
    setupForm();
    move(500, 500); /*Start in mid screen (approx). */
}

void MaskTestForm::setupForm() {
    setWindowTitle("Mask Test Demo");
    QPushButton* againButton = new QPushButton("Another Input Mask", this);
    QPushButton* quitButton = new QPushButton("Quit", this);
    QFormLayout *form = new QFormLayout(this);
    form->addRow("Mask String:", m_InputMask);
    form->addRow("Test Input: ", m_StringEntry);
    form->addRow("Result:", m_Result);
    connect(m_InputMask, SIGNAL(returnPressed()),
            this, SLOT(installMask()));
    connect(m_StringEntry, SIGNAL(returnPressed()),
            this, SLOT(showResult()));
```

```
[ . . . . ]
}
void MaskTestForm::installMask() {
    m_StringEntry->setInputMask(m_InputMask->text());
}
[ . . . . ]
```

图 14.1 给出了应用程序在运行时的屏幕截图

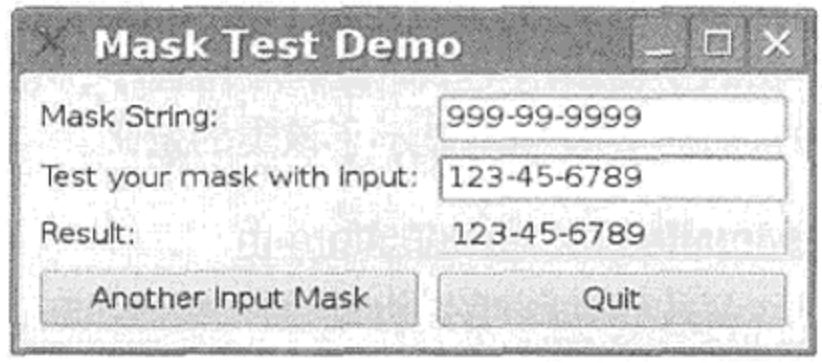


图 14.1 输入掩码

14.1.1 练习：输入掩码

1. 构建和运行 MaskTestForm 应用程序并用它来体验一下下列 inputMask 的值。

   a. AV99e77

   b. \AV99e77

2. 提供一个可接收以下情况的输入掩码(并使用该形式测试你自己的答案)。

   a. 仅允许格式为 123-45-6789 有效社会保障号码

   b. 仅允许格式为 2K3 Y4W(数字和字母相互交替)的加拿大邮政编码

   c. 仅允许格式为 02114-5678 的美国邮政编码

   d. 仅允许格式为(234)345-4567 的美国电话号码

   e. 仅允许格式为 1-123-234-5678 的美国电话号码

3. 能否让最后一个美国电话号码格式开头部分的 1 成为可选项(也就是说，以便也可以接收诸如 123-456-5678 这样的电话号码)？

14.2 验证器

验证器(validator)是可附加到输入窗件(例如 QLineEdit， QSpinBox 和 QComboBox)的不可见对象，能够提供一个检查用户输入的通用框架。Qt 有一个名称为 QValidator 的抽象类，它为所有内置的和自定义的验证器定义了接口。

QValidator 的两个实体子类可以用来对数值范围进行检查：QIntValidator 和 QDoubleValidator。另一个实体子类可以用一个指定的正则表达式来验证字符串。下一节将讨论正则表达式。

QValidator::validate()是一个纯虚函数，它会返回一个枚举值，其含义如下所示。

- Invalid：表达式不满足所要求的条件，进一步输入也于事无补。
- Intermediate：虽然表达式现在不满足所要求的条件，但是进一步的输入可能会产生一个可接受的结果。

- Acceptable：表达式满足所要求的条件。

通过使用 QValidator 类的一些其他成员函数，可以设置 validate()函数使用的条件，例如范围条件)。

一般情况下，正在工作的验证器不允许用户输入可使其返回无效值的数据。

示例 14.3 中给出了一个简短的应用程序，Work-Study Salary Calculator，其中用到了两个数值验证器。该程序可接收用户输入的表示产品信息的一个 int 值和显示其产品信息的一个 double 值并显示它们的乘积。用户按下回车键时，会计算并显示应付总额(Total Pay)的值。

示例 14.3　src/validate/numvalidate/inputform.h

```
[ . . . . ]
class InputForm : public QWidget {
    Q_OBJECT

 public:
    InputForm(int ibot, int itop, double dbot, double dtop);
 public slots:
    void computeResult();
 private:
    void setupForm();
    int m_BotI, m_TopI;
    double m_BotD, m_TopD;
    QLineEdit* m_IntEntry;
    QLineEdit* m_DoubleEntry;
    QLabel* m_Result;
};
[ . . . . ]
```

示例 14.4 中，验证器在构造函数中用范围值进行了初始化，然后又在 setupForm()函数中被指派给相应的输入窗件。

示例 14.4　src/validate/numvalidate/inputform.cpp

```
[ . . . . ]
InputForm::InputForm(int ibot, int itop, double dbot, double dtop):
    m_BotI(ibot), m_TopI(itop), m_BotD(dbot), m_TopD(dtop),
    m_IntEntry(new QLineEdit("0")),
    m_DoubleEntry(new QLineEdit("0")),
    m_Result(new QLabel("0")) {
    setupForm();
    move(500, 500); /*Start in mid screen (approx). */
}

void InputForm::setupForm() {
    [ . . . . ]
    QIntValidator* iValid(new QIntValidator(m_BotI, m_TopI, this));
    QDoubleValidator*
             dValid(new QDoubleValidator(m_BotD, m_TopD, 2, this));
    m_IntEntry->setValidator(iValid);
    m_DoubleEntry->setValidator(dValid);
    connect(m_IntEntry, SIGNAL(returnPressed()),
            this, SLOT(computeResult()));
```

```
    connect(m_DoubleEntry, SIGNAL(returnPressed()),
            this, SLOT(computeResult()));
}
[ . . . . ]
```

程序运行的屏幕截图如图 14.2 所示。

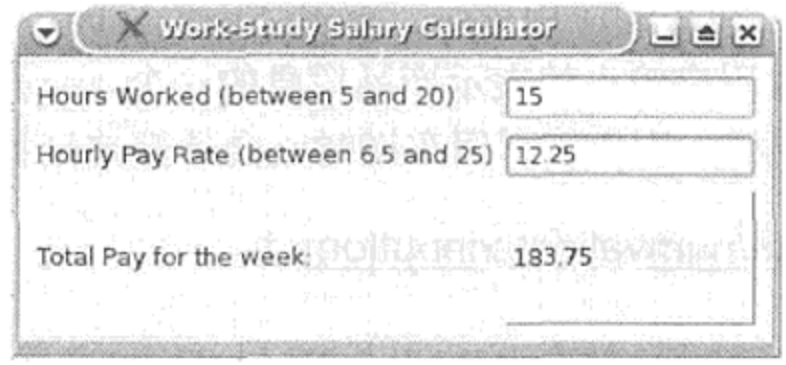


图 14.2 Work Study 计算器

14.2.1 练习：验证器

1. 构建并运行 Work-Study Salary Calculator 应用程序。在第一个 LineEdit 中试着输入一个非 `int` 型值，或者在第二个 LineEdit 中试着输入一个非 `double` 型值。
2. 对于每个 LineEdit，如果试图输入一个超出范围的值会发生什么。例如，对于每小时 3 美元的 2 个小时的工作，其应付总额是多少?

14.3 正则表达式

正则表达式是验证输入、从输入中提取数据以及对输入进行搜索和替换的强大工具。所谓正则表达式，regexp(或者缩写为 regex)，是一种利用模式匹配语言来描述字符串组成限制条件的方式。

正则表达式首先出现在 `vi`，`emacs`，`awk`，`sed` 等工具和 POSIX 标准库中。Perl 是第一个将正则表达式紧密集成到语言中的主流编程语言，那也是人们第一次真正开始学习正则表达式时。人们后来对 Perl 能够识别的正则表达式版本进行了许多改进。这些改进的机制已经成为我们所说的Perl风格的正则表达式扩展的一部分。这些扩展了的正则表达式同样在Java和 Python 中存在。C++0x 和 boost 也提供 C++正则表达式工具。

Qt 提供了一个 `QRegExp` 类，它实现了 Perl 风格的扩展正则表达式语言的大部分功能。

14.3.1 正则表达式语法

与字符串非常相似，正则表达式是一个字符的序列。然而，并非所有字符都只取字面意思。例如，尽管正则表达式中的一个‘a’可以和目标字符串中的‘a’相匹配，但字符‘.’则可以和任意字符相匹配。这里的‘.’就称为元字符(meta-character)。另外一个常用的元字符是‘*’，它可以用来说明目标字符串中可能存在零个或者更多个能够与‘*’前字符串相匹配的字符串。例如，‘a*’将可以和一行中任意数量(包括零个)的‘a’相匹配。如下所示，有许多不同种类的元字符。

下面是一些最经常使用的元字符。

(1) 特殊字符

- .(匹配任何字符)
- \n(匹配换行符)
- \f(匹配换页符)
- \t(匹配制表符)
- \x*hhhh*(匹配一个 Unicode 字符，其对应的码值是范围为 0x0000 到 0xFFFF 之间的一个十六进制数 *hhhh*)

(2) 量词——说明前面的字符(或字符组)在匹配的表达式中可出现次数的修饰符。

- +(出现 1 次或者更多次)
- ?(出现 0 次或者 1 次)
- *(出现 0 次或者更多次)
- {*i*,*j*}(出现至少 *i* 次但不超过 *j* 次)

(3) 字符集——在匹配表达式指定位置允许出现的字符集合。其中还预定义了几个字符集合：

- \s(匹配任何空白符)
- \S(匹配任何非空白符)
- \d(匹配任何数字字符：从‘0’到‘9’)
- \D(匹配任何非数字符合)
- \w(匹配任何“单词”字符，也就是任意的字母、数字或者下画线)
- \W(匹配任意的非单词字符)

字符集也可以使用方括号指定：

- [AEIOU](匹配这五个字符中的任意一个)
- [a-g](短线使此集合可匹配从‘a’到‘g’的字符)
- [^xyz](匹配任何除这三个字符以外的字符)

(4) 分组和捕获字符——(圆括号)是可以用来把字符划分成组的特殊字符。字符组可以是后向引用的。也就是说，如果存在一个匹配，那么分组了的值将可以通过各种方式来捕获和访问。

为简便起见，一般规定在一个正则表达式中最多可以引用 9 个分组，即使用\1 到\9 这样的修饰符。

此外还有一个 QRegExp 成员函数 cap(int nth)，它返回第 *n* 个分组(基于 QString 的形式返回)。

(5) 锚点字符(Anchoring Character)——确定尝试进行匹配操作的边界。

- 脱字符(^)，如果它是正则表达式中的第一个字符，则说明匹配过程从字符串的开头处开始。
- 美元符($)，如果它是正则表达式的最后一个字符，则表明匹配过程直到字符串的结尾才会结束。
- 此外，还有单词边界(\b)断言或者非单词边界(\B)断言，有助于我们关注于正则表达式本身。

表 14.3 中给出了一些正则表达式的例子。

表 14.3 正则表达式的例子

模　式	含　义
hello	匹配字面字符串 hello
c*at	量词：c 出现零次或者多次，at 紧跟其后，例如 at, cat, ccat 等
C?at	匹配 c 出现零次或者一次，之后紧跟 at：仅 at 或者 cat
c.t	c 后面紧跟任意字符，其后面又紧跟 t 的字符串匹配，例如 cat, cot, c3t, c%t 等
c.*t	字符 c 后面紧跟 0 个或者多个任意字符，然后紧跟 t，例如 ct, caaatttt, carsdf$#S8ft 等
ca+t	(量词)‘+’意味着前面的字符可以出现一次或者多次，因此符合条件的有 cat, caat, caaaat 等
c\.*t	反斜线在特殊字符之前将会“使其转义”，因此，只有字符串"c.*t"才匹配
c\\\.t	只和字符串"c\.t"匹配
c[0-9a-c]+z	在 c 和 z 之间有一个或者多个字符在集合[0-9a-c]之间，匹配的字符串类似"c312abbaz"和"caa211bac2z"
the (cat|dog) ate (fish|mouse)	(轮流交替)匹配的结果是 the cat ate the fish, the dog ate the mouse, the dog ate the the fish 或者 the cat ate the mouse
\w+	字母数字(单词字符)的序列，与[a-zA-Z0-9]+等价
\W	非单词字符(标点符号、空白符号等)
\s{5}	正好 5 个空白字符(制表符、空白符或者换行符)
^\s+	匹配字符串开头处的一个或者多个空白字符
\s+$	匹配字符串结尾处的一个或者多个空白字符
^Help	如果 Help 出现在字符串的开头，就匹配它
[^Help]	与字符串中任意地方(和元字符^的意思不一样)的除单词 Help 中任一字母之外的任何单个字符相匹配
\S{1,5}	至少 1 个、至多 5 个非空白字符(可打印的字符)
\d	一个数字[0-9](\D 是一个非数字，也就是[^0-9])
\d{3}-\d{4}	7 位电话号码：555-1234
\bm[A-Z]\w+	\b 代表单词边界：mBuffer 匹配而 StreamBuffer 不匹配

注意

反斜线也可用于转义 C++字符串中的特殊字符，因此 C++字符串内部的正则表达式字符串必须“双倍反斜线”。换句话说，每个\都需要变成\\，为了能够正确匹配反斜线字符本身，则需要 4 个反斜线：\\\\。

注意

如果编译器支持 C++0x，或许希望能够在正则表达式中使用原始引用字符串(raw quoted string)，以避免双倍转义反斜线。

```
R"(The String Data \ Stuff " )"
R"delimiter(The String Data \ Stuff " )delimiter"
```

当然，正则表达式还包括很多内容，还是非常值得花费一些时间来彻底地搞明白它。QRegExp 的文档是开始学习它的好地方。如果需要在更大的范围讨论，推荐参考文献[Friedl 98]。

在此期间，也可以继续探索 QRegExp 的功能，利用来自诺基亚 Qt 的一个例子来测试自己的正则表达式。在 src/regex-tester 目录中可以找到其代码。图 14.3 给出了该程序正在运行时的屏幕截图。

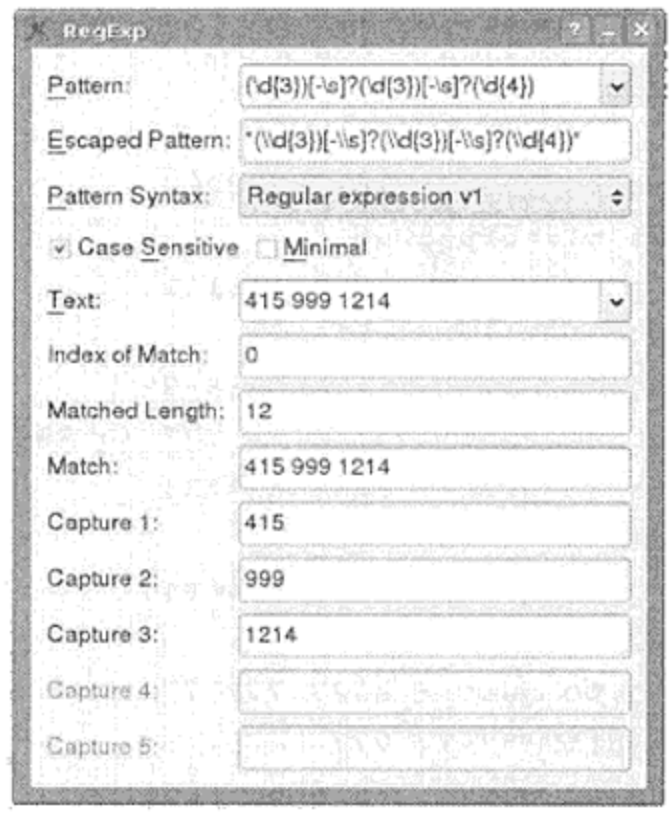


图 14.3　正则表达式测试程序

14.3.2　正则表达式：电话号码识别

14.3.2.1　问题描述

在几乎所有的应用程序中，都有这样一种需求：用一种简单但通用的方式来指定程序运行过程中输入数据时必须满足的一些条件。例如：

- 在美式地址中，每个邮政区号都是一个 5 位数，后接可选的短线(-)和一个 4 位数。
- 美式电话号码是由 10 位数字组成的，通常按照 3 + 3 + 4 的格式分组，此外还可能包含有一些可选的括号、短线以及一个可选的起始值 1。
- 对于美国州名的缩写，必须是 50 个已经得到认可的缩写语之一。

那么应该如何使用面向对象的方式来给定满足上述条件的输入数据的格式?

假设你想要编写这样一个程序，它可识别电话号码的格式并可以接收来自各个国家的各种电话号码，那么需要考虑下面的一些问题：

- 任何美国/加拿大格式的电话号码数字必须形如 *AAA EEE NNNN*，其中 *A* 为地区编码，*E* 为交换中心号码，*N* 为电话号码。
- 对于其他国家电话号码的格式[①]，可以假定其格式必须是 *CC MM*(或 *CCC MM*)后面紧跟 *NN NN NNN* 或者 *NNNNNNN*，此处的 *C* 代表国家，*M* 代表市府代码，*N* 则代表当地的号码数字。
- 在数字簇之间可能会存在短画线或空格。
- 在国家代码之前可能会有+或者 00。

① 在欧洲地区的电话号码情况相当复杂，许多专家已为之付出了多年的努力，希望能够开发一个系统，使其能够被所有欧盟成员国接受。通过访问这里给出的维基百科页面可大致知道其中所要考虑的问题并得到一些灵感。网址为：http://en.wikipedia.org/wiki/ Telephone_numbers_in_Europe 。

现在试想一下，如果仅使用 C++中可用的标准工具，那么应该如何编写这样一个应用程序呢？很有可能需要为每种格式编写一段长长的解析程序。示例 14.5 给出了这样一个程序的预期输出结果。

示例 14.5 src/regexp/testphone.txt

```
src/regexp> ./testphone
Enter a phone number (or q to quit): 16175738000
 validated: (US/Canada) +1 617-573-8000
Enter a phone number (or q to quit): 680111111111
 validated: (Palau) + 680 (0)11-11-11-111
Enter a phone number (or q to quit): 777888888888
 validated: (Unknown - but possibly valid) + 777 (0)88-88-88-888
Enter a phone number (or q to quit): 86333333333
 validated: (China) + 86 (0)33-33-33-333
Enter a phone number (or q to quit): 962444444444
 validated: (Jordan) + 962 (0)44-44-44-444
Enter a phone number (or q to quit): 56777777777
 validated: (Chile) + 56 (0)77-77-77-777
Enter a phone number (or q to quit): 351666666666
 validated: (Portugal) + 351 (0)66-66-66-666
Enter a phone number (or q to quit): 31888888888
 validated: (Netherlands) + 31 (0)88-88-88-888
Enter a phone number (or q to quit): 20398478
Unknown format
Enter a phone number (or q to quit): 2828282828282
Unknown format
Enter a phone number (or q to quit): q
src/regexp>
```

示例 14.6 是一种 C 风格的解决方案，用来说明如何使用 QRegExp 来解决这一问题。

示例 14.6 src/regexp/testphoneread.cpp

```
[ . . . . ]
QRegExp filtercharacters ("[\\s-\\+\\(\\)\\-]");                    1

QRegExp usformat                                                    2
("(\\+?1[- ]?)?\\(?(\\d{3})\\)?[\\s-]?(\\d{3})[\\s-]?(\\d{4})");

QRegExp genformat
("(00)?([[3-9]\\d{1,2})(\\d{2})(\\d{7})$");                         3

QRegExp genformat2
("(\\d\\d)(\\d\\d)(\\d{3})");                                        4


QString countryName(QString ccode) {
   if(ccode == "31") return "Netherlands";
   else if(ccode == "351") return "Portugal";
[ . . . . ]
   //Add more codes as needed ..."
   else return "Unknown - but possibly valid";
}
```

```
QString stdinReadPhone() {                                          5
    QString str;
    bool knownFormat=false;
    do {                                                            6
        cout << "Enter a phone number (or q to quit): ";
        cout.flush();
        str = cin.readLine();
        if (str=="q")
            return str;
        str.remove(filtercharacters);                               7
        if (genformat.exactMatch(str)) {
            QString country = genformat.cap(2);
            QString citycode = genformat.cap(3);
            QString rest = genformat.cap(4);
            if (genformat2.exactMatch(rest)) {
                knownFormat = true;
                QString number = QString("%1-%2-%3")
                                 .arg(genformat2.cap(1))
                                 .arg(genformat2.cap(2))
                                 .arg(genformat2.cap(3));
                str = QString("(%1) + %2 (0)%3-%4").arg(countryName(country))
                        .arg(country).arg(citycode).arg(number);
            }
        }
[ . . . . ]
        if (not knownFormat) {
            cout << "Unknown format" << endl;
        }
    } while (not knownFormat) ;
    return str;
}

int main() {
    QString str;
    do {
        str =  stdinReadPhone();
        if (str != "q")
            cout << " validated: " << str << endl;
    } while (str != "q");
    return 0;
}
[ . . . . ]
```

1 从用户提供的字符串中移除这些字符。

2 所有美式电话号码都有一个国家代码 1，并且拥有 3 + 3 + 4 = 10 位数字。这些数字之间的空白符、短画线和圆括号都应被忽略掉，但是这些符号能够帮助我们更好地理解电话号码。

3 欧洲国家的电话号码以 3 或者 4 开头，拉丁美洲以 5 开头，东南亚和印度洋地区以 6 开头，东亚地区以 8 开头，而亚洲的中部、南部和西部以 9 开头。国家代码的长度可能是两位数字也可能是三位数字。本地电话号码通常是 2 + 2 + 7 = 11 位或者 3 + 2 + 7 = 12 位。这个程序不会试图解析市府编码。

4 最后 7 位数字会被排列成 2 + 2 + 3 的形式。
5 确保用户输入的电话号码字符串必须遵守一个正则表达式并可从其中提取出合适的内容。返回一个适当格式的电话号码。
6 不停询问，直到得到一个有效的号码。
7 移除所有的短画线、空白符号、括号等。

在一个这样的程序中，用户的所有响应都会在他输入字符并按下回车键之后由 QRegExp 进行检查。没有办法可以阻止用户在输入流中键入那些不合适的字符。

14.3.3 练习：正则表达式

1. 许多操作系统都会保存一个单词列表，而各种程序都会使用此列表中的单词来进行拼写检查。在类*nix 系统中，这个单词列表通常(至少是间接地)命名为“words”。在类*nix 系统中，可以通过输入如下命令来确定该文件的位置：

```
locate words | grep dict
```

用 grep 过滤此命令的输出结果，可以将输出结果缩减为仅包含"dict"的那些行。在确定了操作系统中的单词列表文件之后，试着编写一个程序，从此文件中读入文本行，然后采用合适的正则表达式(或者，如果不太可能这样做，试试其他方法)来显示所有符合下列条件的单词。

- a 以一对重复字母开头的单词
- b 以“gory‘结尾的单词
- c 有重复字母超过一个的单词
- d 是回文的单词(即顺读和倒读都相同的单词)
- e 由按照字母表顺序严格升序排列的字母组成的单词(例如，knot)

如果在操作系统中无法找到合适的单词列表，可以使用本书源代码安装包中提供的文件：src/downloads/canadian-english-small[①]。如果之前没有这方面的准备，就需要用命令 gunzip canadian-english-small.gz 将其解压缩出来。

2. 编写一个程序，使其使用正则表达式来从 HTML 文件中提取超链接。超链接的形式如下：

```
<a href="http://www.web.www/location/page.html">The Label</a>
```

在输入文件中遇到每个超链接时，只打印其 URL 和标签，两者之间用 Tab 键分隔。需要记住的是，可选的空白字符可以在之前的示例模式中的不同部位出现。用一系列含有超链接的不同 Web 页面来测试该程序，并确认程序的确可以捕捉到所有的链接。

3. 假定你刚刚换了一家公司，并且要打算复用那些为前公司编写的开源代码。现在，需要将源代码中的所有数据成员都进行重命名。之前的公司对于数据成员的命名方式类似于：mVarName。然而，新公司希望它们的具有这样的命名方式：m_varName。请用 QDirIterator 对找到的每个文件中的文字都执行替换，以便让所有的数据成员都可以与新公司的编码标准一致。

---

① 可从附录中的 dist 目录中下载该文件。

14.4　正则表达式验证

QRegExpValidator 类使用了 QRegExp 来验证输入字符串。示例 14.7 中给出了一个包含 QRegExpValidator 和一些输入窗件的主窗口。

示例 14.7　src/validate/regexval/rinputform.h

```
[ . . . . ]

class RinputForm : public QWidget {
    Q_OBJECT
 public:
    explicit RinputForm(QWidget* parent=0);
    void setupForm();
 public slots:
    void computeResult();
 private:
    QLineEdit* m_PhoneEntry;
    QLabel* m_PhoneResult;
    QString m_Phone;
    static QRegExp s_PhoneFormat;
};
[ . . . . ]
```

我们借用了示例 14.6 中的正则表达式，用它来初始化示例 14.8 中的静态 QRegExp。示例 14.8 接受用户输入的一个电话号码，且只在通过有效性验证之后才能将其显示出来。值得注意的是，14.1.1 节中提出问题 3 时我们知道，美国式电话号码中起始的 1 是可有可无的。

示例 14.8　src/validate/regexval/rinputform.cpp

```
[ . . . . ]
QRegExp RinputForm::s_PhoneFormat(
  "(\\+?1[- ]?)?\\(?(\\d{3,3})\\)?[\\s-]?(\\d{3,3})[\\s-]?(\\d{4,4})");

RinputForm::RinputForm(QWidget* parent)
:   QWidget(parent),
    m_PhoneEntry(new QLineEdit),
    m_PhoneResult(new QLabel) {
    setupForm();
    move(500, 500); /*Start in mid screen (approx). */
}

void RinputForm::setupForm() {
    [ . . . . ]
    QRegExpValidator*
          phoneValid(new QRegExpValidator(s_PhoneFormat, this));
    m_PhoneEntry->setValidator(phoneValid);
    connect(m_PhoneEntry, SIGNAL(returnPressed()),
            this, SLOT(computeResult()));
}

void RinputForm::computeResult() {
    m_Phone = m_PhoneEntry->text();
```

```
        if (s_PhoneFormat.exactMatch(m_Phone)) {
            QString areacode = s_PhoneFormat.cap(2);
            QString exchange = s_PhoneFormat.cap(3);
            QString number = s_PhoneFormat.cap(4);
            m_PhoneResult->setText(QString("(US/Canada) +1 %1-%2-%3")
                  .arg(areacode).arg(exchange).arg(number));
        }
    }
    [ . . . . ]
```


QRegExpValidator 类不会接收任何可能产生无效结果的字符。图 14.4 给出了该程序运行过程中的一个屏幕截图。

图 14.4 电话号码验证器

QValidator 为输入数据验证提供了一种强大的机制。Qt 提供了两个数值范围验证器和一个正则表达式验证器。如果使用借助数字范围和正则表达式尚无法完成所需验证，可以针对具体情况较为容易地扩展并生成 QValidator 的自定义子类。

14.5 子类化 QValidator

当验证用户输入的需求超出了简单的数字范围或者正则表达式验证验证的能力时，可以通过对 QValidator 派生来定义自己的验证器类。在接下来的例子中，我们定义回文(palindrome)为这样一种字符串，在忽略大小写、空格和标点符号时，它从前往后或者从后往前读起来都相同。遗憾的是，还没有哪一个简单的正则表达式可以用来判定任意长度的字符串是否是一个回文字符串。示例 14.9 给出了一个 Palindate，这是一个可用来验证所给定的字符串是否是一个回文字符串的 QValidator。

示例 14.9 src/validate/palindrome/palindate.h

```
#include <QValidator>
#include <QString>

class Palindate : public QValidator {
    Q_OBJECT
public:
    explicit Palindate(QObject* parent = 0);
    QValidator::State validate(QString& input, int&) const;
};
```

QValidator 类有一个成员函数必须重载：validate()，其定义如示例 14.10 所示。该函数生成给定字符串的小写副本 inpStr，并从中移除所有的空白字符和标点符号。然后，它会把 inpStr 和 revStr 字符串作比较，revStr 字符串中含有同样的字符，但顺序与之相反。

示例 14.10 src/validate/palindrome/palindate.cpp

```
[ . . . . ]
QValidator::State Palindate::validate(QString& str, int& ) const {
    QString inpStr(str.toLower());
    QString skipchars("-_!,;. \t");
```

```
    foreach(QChar ch, skipchars)
        inpStr = inpStr.remove(ch);
    QString revStr;
    for(int i=inpStr.length(); i > 0; --i)
        revStr.append(inpStr[i-1]);
    if(inpStr == revStr)
        return Acceptable;
    else
        return Intermediate;
}
```

1 用 regex 来做的话可能会更快一些。

2 令人诧异的是，没有 reverse() 函数。

示例 14.11 定义了一个包含 QLineEdit 的窗件以用于测试该验证器。

示例 14.11　src/validate/palindrome/palindromeform.h

```
[ . . . . ]
class PalindromeForm : public QWidget {
    Q_OBJECT
public:
    PalindromeForm(QWidget* parent=0);
    QString getPalindrome();
public slots:
    void showResult();
    void again();
private:
    Palindate* m_Palindate;
    QLineEdit* m_LineEdit;
    QLabel* m_Result;
    QString m_InputString;
    void setupForm();
};
[ . . . . ]
```

示例 14.12 中构建了一些窗件，并在 QLineEdit 上设置了一个 Palindate 验证器。

示例 14.12　src/validate/palindrome/palindromeform.cpp

```
[ . . . . ]
PalindromeForm::PalindromeForm(QWidget* parent) : QWidget(parent),
  m_Palindate(new Palindate),
  m_LineEdit(new QLineEdit),
  m_Result(new QLabel) {
    setupForm();
}

void PalindromeForm::setupForm() {
    setWindowTitle("Palindrome Checker");
    m_LineEdit->setValidator(m_Palindate);
    connect(m_LineEdit, SIGNAL(returnPressed()),
            this, SLOT(showResult()));
[ . . . . ]
}
```

```
void PalindromeForm::showResult() {
   QString str = m_LineEdit->text();
   int pos(0);
   if(m_Palindate->validate(str,pos) == QValidator::Acceptable) {
      m_InputString = str;
      m_Result->setText("Valid Palindrome!");
   }
   else {
      m_InputString = "";
      m_Result->setText("Not a Palindrome!");
   }
}
[ . . . . ]
```

图 14.5 中给出了程序运行过程中的一个屏幕截图。

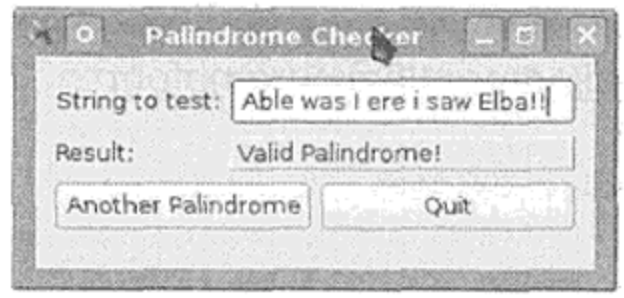


图 14.5 回文检验器

14.6 练习：验证和正则表达式

图 14.6 Address 窗体

1. 设计一个如图 14.6 所示的 Address 窗体。
   当用户从组合框中选择一个国家时，程序必须在三个 QLineEdit 上设置一个适当的验证器，即 stateEdit，zipEdit 和 phoneEdit。同样，对话框的标签应当以下列方式发生变化。
   a. 美国
      - 将 zipLabel 设置成 Zip。
      - 将 stateLabel 设置成 State。

   b. 加拿大
      - 将 zipLabel 设置成 Postal Code。
      - 将 stateLabel 设置成 Province。

   c. 用适当的验证器在选项中另外添加一个国家。

   确保永不接受非法输入(同时还要禁用 OK 按钮)。单击 OK 按钮后，应关闭对话框并打印出每个域的值。如果愿意，可以从 handout/validation/addressform.ui 文件开始。
2. 国际标准书号(International Standard Book Number，ISBN)是一种用于书籍和其他出版物的编码系统。ISBN 的大小可以不同(10 位和 13 位)，它们可以分解成标识语种、出版社和书名的子字符串。此外，最右端的那位数字是一个校验数字，它可从其他位中计算出来以用于错误校验[①]。

① Wikipedia 中有一篇解释该系统的有趣文章，参见 http://en.wikipedia.org/wiki/ISBN。

假定 ISBN-10 编码(包括校验位)看起来是这个样子：

```
d1d2d3...d9d10
```

那么，下式必须得平衡：

```
(10*d1 + 9*d2 + 8*d3 + ... + 2*d9 + d10) % 11 = 0
```

如果所需的校验位是10,可以在那个位置上放一个字母X。定义一个当且仅当ISBN-10编码的校验位正确时方可接受的 QValidator。可以用 0-306-40615-2(有效)和0-306-40615-5(无效)对其进行测试。

3. 找出(例如，从维基百科的文章中[①])如何验证 ISBN-13 编码校验位的方法，并定义一个当且仅当 ISBN-13 编码的校验位正确时方可接受的 QValidator。
4. 你能只用一个 QValidator 就既可处理 ISBN-10 编码又可处理 ISBN-13 编码吗？假设仅需验证校验位。

14.7　复习题

1. 什么是输入掩码？输入掩码可以用来干什么？
2. 什么是正则表达式？可以使用正则表达式来做些什么工作？
3. 什么是验证器？它有什么用处？
4. 说明三种不同类型的验证器。
5. 什么是正则表达式的元字符？有四种元字符：量词、字符集、分组和锚点。针对每种元字符给出一个具体的例子并解释其具体含义。
6. 为什么需要扩展 QValidator？

① 参见http://en.wikipedia.org/wiki/ISBN。

第 15 章　XML 解析

这一章将说明解析 XML 数据时的三种不同方法：两个来自 Qt 的 XML 模块，还有一个是新的，改进的来自 Qt 的 core 模块。这些例子会分别用来说明 SAX（Simple API for XML，XML 简单应用程序编程接口）的事件驱动解析（parse-event-driven parsing）方法、DOM（Document Object Model，文档对象模型）的解析树（tree-style parsing）方法和 `QXmlStreamReader` 的解析流（stream-style parsing）方法。

XML 是 Extensible Markup Language（可扩展标记语言）首字母的缩写词。它是这样一种标记语言，类似于 HTML（Hypertext Markup Language，超文本标记语言）但拥有更加严格的语法且没有定义任何语义（也就是说，其标记并没有关联任何含义）。

XML 严格的语法相较于 HTML 的语法而言，可构成强烈的对比。例如：

- 每个 XML <tag>都必须有一个对应的结束</tag>，或者是自我封闭，类似于这样的
。
- XML 标记区分大小写：<tag>与<Tag>是不一样的。
- 为了避免解析器产生混淆，在 XML 文档中类似于>和<等实际上不是标记一部分的字符，必须替换成被动等价的字符，例如>和<。
- 此外，和符号字符（&）必须由它的等价被动字符&进行替换，这也是出于同样的原因。

示例 15.1 是一个 HTML 文档，需要注意的是，这个文档并不符合 XML 的规则。

示例 15.1　src/xml/html/testhtml.html

```
<html>
<head> <title> This is a title </title>

 <!--Unterminated <link> and <input> tags are commonplace
         in HTML code:     -->
 <link rel="Saved Searches" title="oopdocbook"
   href="buglist.cgi?cmdtype=runnamed&amp;namedcmd=oopdocbook">
 <link rel="Saved Searches" title="queryj"
   href="buglist.cgi?cmdtype=runnamed&amp;namedcmd=queryj">
</head>
<body>
<p> This is a paragraph. What do you think of that?

HTML makes use of unterminated line breaks: <br>
And those do not make XML parsers happy. <br>

<ul>
<li> HTML is not very strict.
<li> An unclosed tag doesn't bother HTML parsers one bit.
</ul>
```

```
</body>
</html>
```

如果把 XML 的语法与 HTML 的元素语义结合起来，那么就可以得到称为 XHTML 的语言。示例 15.2 给出了示例 15.1 的 XHTML 版本。

示例 15.2　src/xml/html/testxhtml.html

```
<!DOCTYPE xhtml >
<html>
<head>
<title> This is a title </title>
<!-- These links are now self-terminated: -->
<link rel="Saved Searches" title="oopdocbook"
  href="buglist.cgi?cmdtype=runnamed" />
<link rel="Saved Searches" title="queryj"
  href="buglist.cgi?namedcmd=queryj" />
</head>
<body>

<p> This is a paragraph. What do you think of that? </p>
<p>
Html self-terminating line breaks are ok: <br/>
They don't confuse the XML parser. <br/>
</p>

<ul>
<li> This is proper list item </li>
<li> This is another list item </li>
</ul>

</body>
</html>
```

XML 是人类和应用程序都可以理解的一整类文件格式。目前，XML 已经发展成为 Web 应用程序存储和交换数据的流行格式，同时它也是一种擅长于表达层次(树状)信息的语言，而树状信息囊括了大部分的文档。

许多应用程序(例如，Qt 设计师、Umbrello 和 Dia)都使用某种特殊的 XML 文件格式来存储数据。Qt 设计师的 `.ui` 文件就是使用 XML 来描述一个 GUI 中 Qt 窗件的布局。本书英文版也是采用一种称为 Slacker 的 DocBook[①]的 XML 格式写成的，它类似于专门用来编写图书的 XML 语言 DocBook[②]，但是增加了一些来自 XHTML 的简写标记和自定义标记来描述课件。

XML 文档由节点(node)组成。基本元素(element)是形如<tag>文本或者元素</tag>的节点。一个开放标记(opening tag)可以包含属性，所有属性都有如下的形式：`name="value"`。相互交织的元素就组成了一种具有父-子关系的树状结构。

示例 15.3　src/xml/saxl/samplefile.xml

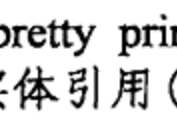
```
<section id="xmlintro">
    <title> Intro to XML </title>
    <para> This is a paragraph </para>
```

① 参见 http://slackerdoc.tigris.org。

② 参见 http://www.docbook.org。

```
    <ul>
        <li> This is an unordered list item. </li>
        <li c="textbook"> This only shows up in the textbook </li>
    </ul>
    <p> Look at this example code below: </p>
    <include src="xmlsamplecode.cpp" mode="cpp"/>
</section>
```


示例 15.3 中，<ul>有两个<li>子节点，而它的父亲是<section>。没有孩子的元素可以使用一个/>实现自我终止，例如，<include/>。一些元素拥有属性，例如<section>和<include>。尽管大部分解析器都会忽略多余的空白符，但把嵌套的元素进行缩进有助于提高可读性。



问题

<section>有多少个直接儿子?

XML 编辑器

有几种可用的开源 XML 编辑器。在转向寻求商业化软件方案之前，建议先试用一下这些软件。

1. jEdit[①]有一个使用起来非常不错的 XML 插件(要确保启用了 Sidekick)。
2. 对于 KDE 用户，可以使用 quanta[②]，就像 Kdevelop 一样，quanta 也基于 KDE 的高级文本编辑器 Kate。如果熟悉使用 emacs 的快捷键，那么可以下载并安装 emacs 版本的 Kate 插件：ktexteditor-emacsextensions[③]。

提示

自由工具 xmllint 对于检查 XML 文件中的错误非常拿手，它能够报告描述性的错误(不匹配的开始/结束标记、丢失的字符等)并可指出错误的位置。它可以用来将文档自动缩进成“pretty print”这种格式非常好的 XML 文档。同时，它还可以用来将使用 XInclude 或者外部实体引用(external entity reference)的多个部分的文档合并在一起——这两个特性都是 Qt XML 解析器尚不支持的。

15.1 Qt XML 解析器

XML 被广泛用作程序之间的信息交流媒介，也用作存储和传输层次化数据的一种方便的方式。这就意味着，对于每个可以生成 XML 输出的程序来说，就必须要有一个相应的读取程序来读取它并从中加工出有意义的东西。例如：把由 Umbrello 生成的 .xmi 文件转换成图形化的图表，把由设计师生成的 .ui 文件转换成屏幕上显示的窗件，把由 DocBook 生成的 .xml 文件转换成用于印刷的 .pdf 文件或者是可以在网络上阅读的 .tml 文件等。要完成这样的转

① 参见http://www.jedit.org。

② 参见http://quanta.kdewebdev.org/。

③ 参见http://www.kde-apps.org/content/show.php?content=21706。

换，就需要能够对 XML 文件进行解析。以上所提到的每个应用程序都有一种解析 XML 的方法，它们使用 SAX, DOM 或者其他的 XML 解析应用程序编程接口（Application Programming Interface，API）。一些 API，如 DOM 和 QXmlStreamWriter，还会有一种更高级的生成 XML 文件的方法。

QXmlStreamReader 和 QXmlStreamWriter

从 Qt 4.3 开始，出现了另外一种 XML API：QXmlStreamReader 和 QXmlStreamWriter。这些基于流（stream）的类位于 QtCore 中，并建议用于新的应用程序，因为它们比 Qt 的 XML 模块提供了更多的灵活性和更好的性能。在 15.4 节，很快就可以看到一个用它建立元素树的例子。

Qt 的 XML 模块包括两个标准（跨语言）的 XML API Qt 实现，分别如下所示。

SAX[①]详细说明了一种用于一系列事件驱动 API 的类，它们可以低层次、顺序访问地解析 XML 文档。它最初是用 Java 开发的，也是打算用于 Java 型的解析器中。要利用 SAX API，就很有必要定义一些回调函数（callback function），可对各种 SAX 事件进行被动响应，例如：

- 开始标记，结束标记。
- 字符数据（内容）。
- XML 处理指令（用<?和?>封闭的内容）[②]。
- XML 注释（用<!--和-->封闭的内容）。

SAX 可以处理几乎任何大小的 XML 文件。SAX 解析只可以向前处理，SAX 应用程序希望在解析文档的过程中就同时顺序完成它们的处理工作。

DOM[③]是一个可以把 XML 元素表示成导航树结构中对象（即节点）的标准 API。由于 DOM 会把整个 XML 文件加载到内存中，应用程序可以处理的最大文件大小要受限于可用的内存量。DOM 特别适用于需要对 XML 文档各个部分进行随机存取（而不是顺序访问）的应用程序。它不提供解析错误处理或者元素修改方法，但它提供了一个用于创建文档的 API。

要使用 Qt 的 XML 模块，需要在你的项目文件中添加下面一行代码：

```
QT += xml
```

15.2　SAX 解析

在使用 SAX 风格的 XML 解析器时，处理流程就完全取决于从文件或者流中顺序读取的数据。这种控制翻转（inversion of control）就意味着执行中的线程追踪需要一个栈，以跟踪对回调函数的被动调用。而且，Qt 库中的解析代码将会调用你的代码（重载虚函数）。

为了激活解析器，首先要创建一个 reader 和一个 handler，然后将它们连接起来，最后再调用 parse() 函数，如示例 15.4 所示。

---

① 参见http://www.saxproject.org。

② 所谓处理指令，是指一种 XML 节点类型，可以在文档的任何地方出现。原打算是包含应用程序的说明。

③ 参见http://www.w3c.org/DOM/。

示例 15.4 src/xml/sax1/tagreader.cpp

```
#include "myhandler.h"
#include <QFile>
#include <QXmlInputSource>
#include <QXmlSimpleReader>
#include <QDebug>

int main( int argc, char **argv ) {
    if ( argc < 2 ) {
        qDebug() << QString("Usage: %1 <xmlfile>").arg(argv[0]);
        return 1;
    }
    MyHandler handler;
    QXmlSimpleReader reader;                                    1
    reader.setContentHandler( &handler );                       2
    for ( int i=1; i < argc; ++i ) {
        QFile xmlFile( argv[i] );
        QXmlInputSource source( &xmlFile );
        reader.parse( source );                                 3
    }
    return 0;
}
```

1 通用解析器。

2 将各个对象连接在一起。

3 开始解析。

解析 XML 的接口在抽象基类 `QXmlContentHandler` 中进行描述，可以将它称为被动接口，因为调用 `MyHandler` 函数的不是自己的代码。`QXmlSimpleReader` 对象会读取 XML 文件并生成解析事件，然后通过调用 MyHandler 函数对其响应。图 15.1 给出了这一过程中所涉及的主要的类。

为了使 XML reader 能够提供有用信息，需要有一个对象能够接受解析事件，这个对象就是解析事件处理器，它必须实现一个由抽象基类指定的接口，这样才能够“插入”解析器中，如图 15.2 所示。

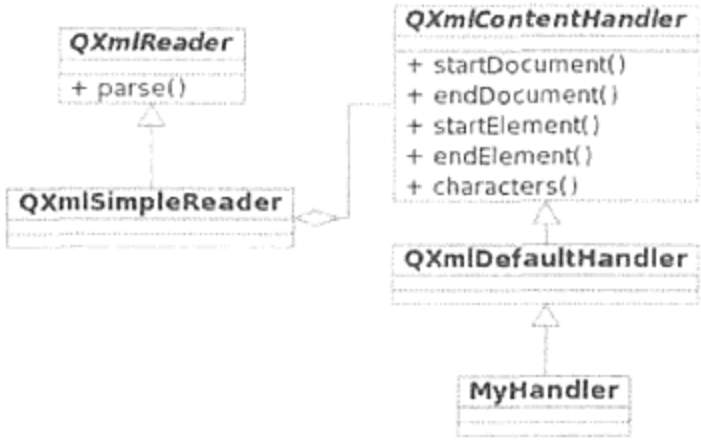


图 15.1 抽象 SAX 类和具体 SAX 类

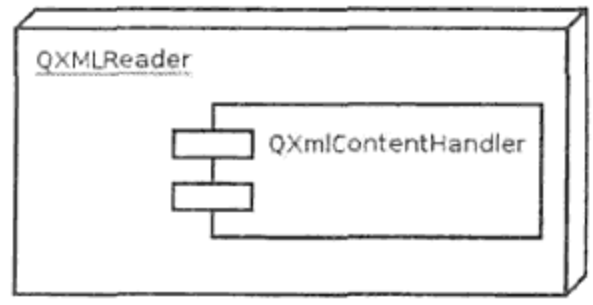


图 15.2 插件组件的结构

处理程序(直接或者间接地)继承自 `QXmlContentHandler`。在解析过程中遇到 XML 文件的各种元素时，解析器就会调用这些虚方法。无须直接调用这些函数。示例 15.5 给出了一个类，它扩展了默认的处理程序，以便根据应用程序所需的特定方式响应解析事件。

示例 15.5　src/xml/sax1/myhandler.h

```
[ . . . . ]
#include <QXmlDefaultHandler>
class QString;
class MyHandler : public QXmlDefaultHandler {
  public:
    bool startDocument();
    bool startElement( const QString & namespaceURI,
                       const QString & localName,
                       const QString & qName,
                       const QXmlAttributes & atts);
    bool characters(const QString& text);
    bool endElement( const QString & namespaceURI,
                     const QString & localName,
                     const QString & qName );
  private:
    QString indent;
};
[ . . . . ]
```

这些被动调用的函数通常称为回调函数，它们负责响应解析器生成的各种事件。例如，MyHandler 的客户代码是 Qt XML 模块中的 QXmlSimpleReader 类。

ContentHandler 还是 DefaultHandler

QXmlContentHandler 是一个包含许多纯虚函数的抽象类，任何派生实体类都必须重载这些虚函数。Qt 提供了一个 QXmlDefaultHandler 实体类，它在实现抽象基类的纯虚函数时将它们设置为不进行任何操作的空函数。可以把这个类作为实体基类。从这个类派生处理程序时，没有必要重载所有方法，但是为了完成工作必须重载其中的一些方法。

如果没有适当地重载应用程序中用到的每一个方法，那么程序将会调用对应的 QXmlDefaultHandler 方法，即使这些方法不做任何工作。在一个处理程序的函数体中，可以执行下列动作。

- 将解析结果存储在一个数据结构中。
- 根据特定的规则创建对象。
- 按照不同的格式打印或者转换数据。
- 做一些其他有意义的工作。

示例 15.6 包含了一个具体事件处理器的定义。

示例 15.6　src/xml/sax1/myhandler.cpp

```
[ . . . . ]
QTextStream cout(stdout);

bool MyHandler::startDocument() {
    indent = "";
    return TRUE;
}
```

```
bool MyHandler::characters(const QString& text) {
    QString t = text;
    cout << t.remove('\n');
    return TRUE;
}

bool MyHandler::startElement( const QString&,                    1
                              const QString&, const QString& qName,
                              const QXmlAttributes& atts) {
    QString str = QString("\n%1\\%2").arg(indent).arg(qName);
    cout << str;
    if (atts.length()>0) {
        QString fieldName = atts.qName(0);
        QString fieldValue = atts.value(0);
        cout << QString("(%2=%3)").arg(fieldName).arg(fieldValue);
    }
    cout << "{";
    indent += "    ";
    return TRUE;
}

bool MyHandler::endElement( const QString&,
    const QString& , const QString& ) {
    indent.remove( 0, 4 );
    cout << "}";
    return TRUE;
}
[ . . . . ]
```

1 忽略了那些没有使用的参数名，这就避免了编译器总是发出“unused parameter”警告的问题。

传递给 startElement() 函数的 QXmlAttributes 对象是一个映射，用来存放包含在 XML 元素中 *name* = *value* 的属性值对。

处理文件的过程中，parse() 函数在文件中某种“事件”出现时就会相应地调用 characters(), startElement() 和 endElement() 等函数。无论何时，只要在标记的开始处和结尾之间遇到一个普通字符组成的字符串，处理过程都会将它传递给 characters() 函数。

将该程序运行于示例 15.3 中的内容，它会将这个文档转换成示例 15.7 中所示的内容，看起来像 LaTeX 的另一种文档格式。

示例 15.7 src/xml/sax1/tagreader-output.txt

```
\section(id=xmlintro){
    \title{ Intro to XML }
    \para{ This is a paragraph }
    \ul{
        \li{ This is an unordered list item. }
        \li(c=textbook){ This only shows up in the textbook }     }
    \p{ Look at this example code below: }
    \include(src=xmlsamplecode.cpp){}}
```

15.3　XML，树结构和 DOM

文档对象模型(Document Object Model，DOM)是操作 XML 的一个高级接口。可以非常直观地处理和浏览 DOM 文档(如果对 QObject 子对象比较熟悉，就更容易懂些)。

图 15.3 给出了 DOM 涉及的主要类。setContent() 函数用于解析文件，在此之后的 QDomDocument 就会包含由 QDomNode 对象组成的一颗结构化树。而每个 QDomNode 可能是 QDomElement，QDomAttr 或者 QDomText 的一个实例。除了作为根的 QDomDocument 之外，每个 QDomNode 都有一个父亲。每个节点都可以从父亲那里找到自己。DOM 可以看成是组合模式(Composite Pattern)的另外一个应用。

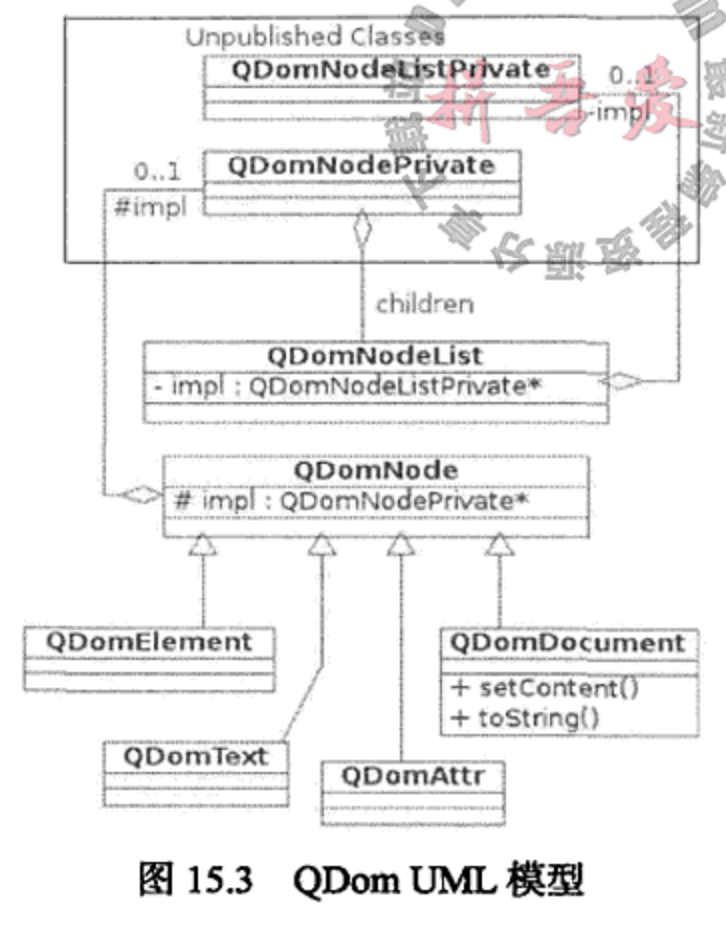


图 15.3　QDom UML 模型

注意

QDom 类是对一些私有实现类的封装。除了指针之外，它们不包含任何其他数据。这使得可以通过值将 QDomNode 传递给其他可能会改变其指向对象(通过添加属性或者儿子或者修改属性)的函数。这使得 QDom 接口就更加类似于 Java 风格的接口了。

15.3.1　DOM 树遍历

Qt 提供对 XML 数据树的完全读/写访问。可以通过一个与 QObject 的接口相类似但稍有不同的接口来遍历节点。在接口的下层，SAX 会直接进行解析操作，DOM 则定义了一个在内存中创建对象树的内容处理器。客户代码所需要做的全部工作就是调用 setContent() 方法，这将会对输入的 XML 和生成的树进行解析。

示例 15.8 在合适的位置将一个 XML 文档进行了转换。在对树进行操作之后，将其序列化为一个 QTextStream 流，这样就可以将其保存或者再次进行解析。图 15.4 给出了这个例子中用到的主要类。

示例 15.8　src/xml/domwalker/main.cpp

```
[ . . . . ]
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QString filename;
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " filename.xml" << endl;
        filename = "samplefile.xml";
    }
    else {
        filename = argv[1];
    }
```

```
    QFile f(filename);
    QString errorMsg;
    int errorLine, errorColumn;
    QDomDocument doc("SlackerDoc");
    bool result = doc.setContent(&f, &errorMsg,
        &errorLine, &errorColumn);                    1
    QDomNode before = doc.cloneNode(true);            2
    Slacker slack(doc);                               3
    QDomNode after = slack.transform();               4
    cout << QString("Before: ") << before << endl;
    cout << QString("After: ") << after << endl;
    QWidget * view = twinview(before, after);         5
    view->show();
    app.exec();
    delete view;
}
[ . . . . ]
```

1 把文件解析成一棵 DOM 树，然后将树存储在一个空文档中。

2 深层复制。

3 将树发送给 `Slacker`。

4 开始访问。

5 创建一对 `QTreeView` 对象，通过滑动条将其分隔开来，并使用 `QDomDocument` 作为模型。

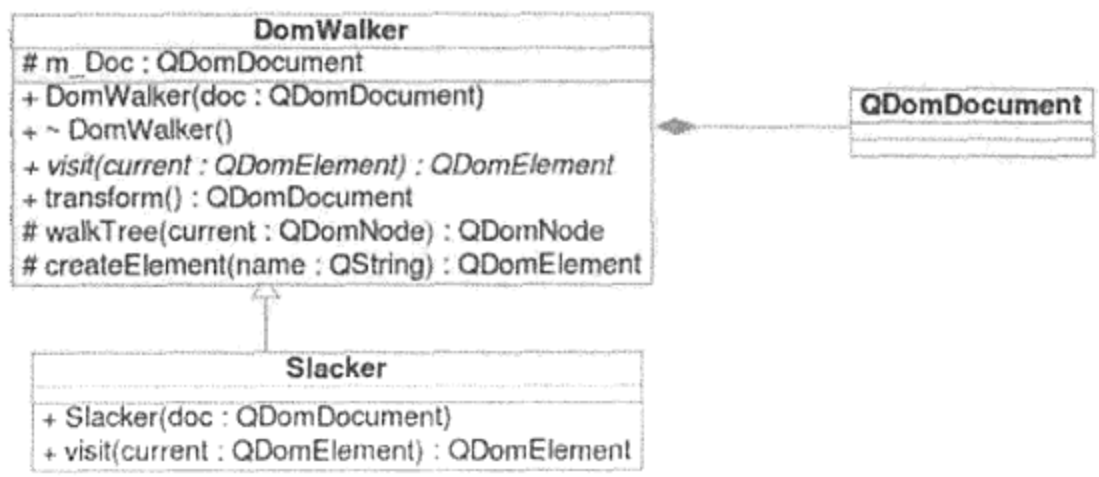


图 15.4 Domwalker 和 Slacker

`Slacker` 派生自 `DomWalker`，这是一个专门用来遍历 DOM 树的应用。

需要注意的是，示例 15.9 中定义的 `walkTree()` 函数没有使用任何指针或者类型转换。这里的 `QDom(Node|Element|Document|Attribute)` 类型是智能指针。我们使用 `QDomNode::toElement()` 或者 `QDomNode::toXXX()` 转换函数来把 `QDomNode` “向下”转换为一个 `QDomElement` 或者 `QDom`*XXX*。

示例 15.9 src/xml/domwalker/domwalker.cpp

```
[ . . . . ]
QDomDocument DomWalker::transform() {
    walkTree(m_Doc);
    return m_Doc;
}

QDomNode DomWalker::walkTree(QDomNode current) {
```

```
    QDomNodeList dnl = current.childNodes();
    for (int i=dnl.count()-1; i >=0; --i)
        walkTree(dnl.item(i));
    if (current.nodeType() == QDomNode::ElementNode) {
       QDomElement ce = current.toElement();
       return visit(ce);
    }
    return current;
}
[ . . . . ]
```

1 首先递归地处理所有儿子。

2 仅处理元素，不会改变其他节点。

3 取代类型转换。

提示

在对一棵树进行遍历时，可以只使用 QDomNode 接口。但是，当操作一个实际的 XML 元素时，“向下转换”为 QDomElement 就可以增加一些将元素及其属性(其本身也是 QDomNode 的子对象)作为整体进行处理的函数，这是非常方便的。

虽然这个例子中 QDomNode/QDomElement 对象是按值传递的，也是按值返回的，但是仍然可以通过临时复制改变底层的 DOM 对象。通过接口欺骗(interface trickery)，QDOM 看起来非常类似于 Java 风格的引用，它们在内部保存了指针而不是实际的数据。

Slacker 定义了将文档从一种 XML 格式转换成另外一种 XML 格式的方法。这是 DomWalker 的一个扩展，其中仅仅重载了一个 visit() 方法。该方法对于每种元素都有一个特定的规则，示例 15.10 中给出了它们的定义。

示例 15.10　src/xml/domwalker/slacker.cpp

```
[ . . . . ]
QDomElement Slacker::visit(QDomElement element) {
    QString name = element.tagName();
    QString cvalue = element.attribute("c", QString()) ;
    if (cvalue != QString()) {                                    1
        element.setAttribute("condition", cvalue);
        element.setAttribute("c", QString());
    }
    [ . . . . ]
    if (name == "b") {
        element.setAttribute("role", "strong");
        element.setTagName("emphasis");
        return element;
}
if (name == "li") {                                               2
    QDomNode parent = element.parentNode();
    QDomElement listitem = createElement("listitem");
    parent.replaceChild(listitem, element);                       3
    element.setTagName("para");                                   4
```

```
        listitem.appendChild(element);
        return listitem;
    }
    [ . . . . ]
```

1 修改属性——把所有的 c=变成 condition=。

2 这个转换更有意思，因为会用<listitem><para> text </para></listitem>来替换 text 。

3 移除 li 标记，在 li 的位置放一个 listitem。

4 就地修改标记的名称。

运行这个例子时，首先会弹出一个如图 15.5 所示的窗口，这个窗口中带有两个并排放置的树视图，应仔细分析一下转换之前和转换之后的 XML 文档。

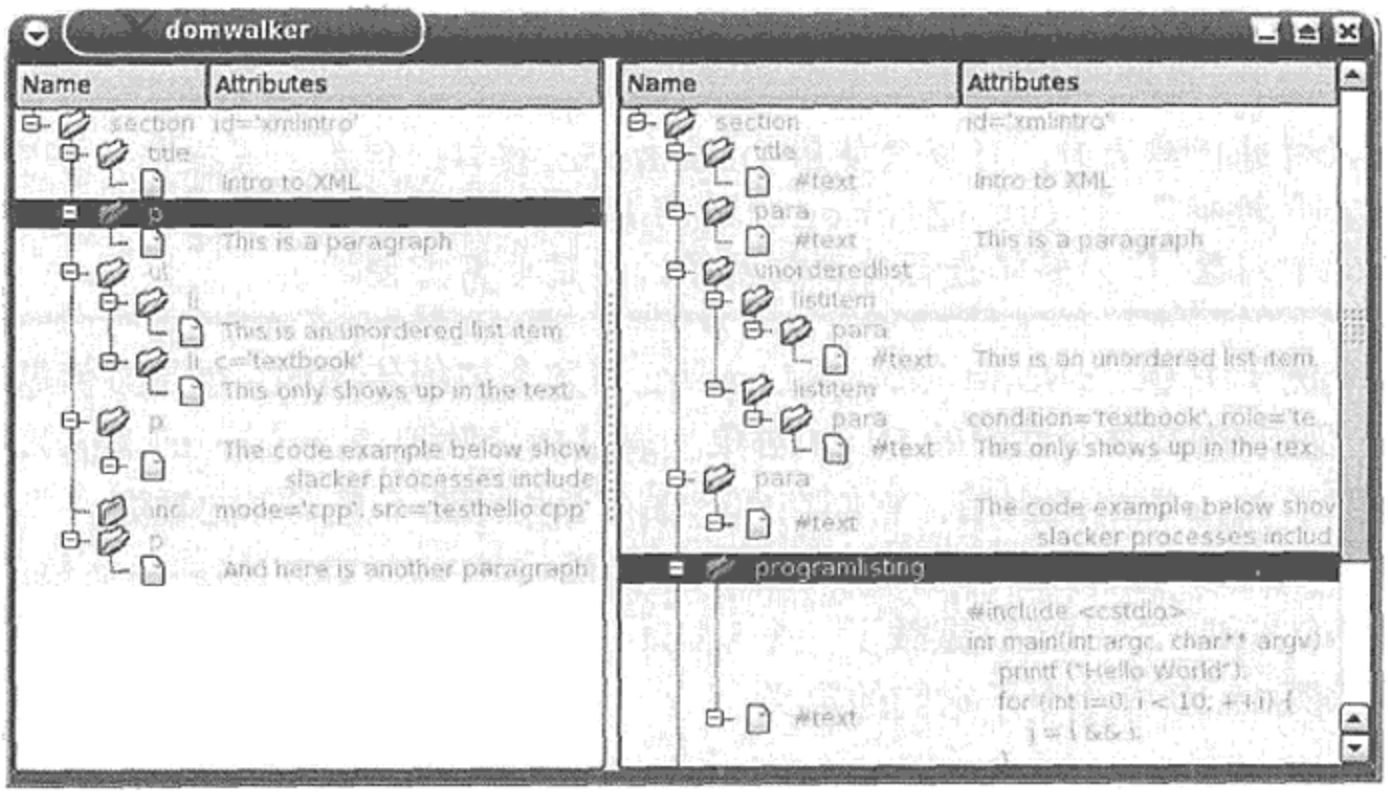


图 15.5 XML 的树视图

15.3.2 用 DOM 生成 XML

DOM 文档通常是由解析器从一个输入流中创建以表示 XML，但是 DOM 也可以用生成的 XML 结构作为输出。一般更倾向于使用 API 而不是通过打印格式化字符串来生成 XML，这是因为 DOM 生成的结果可确保它能被再次进行解析。

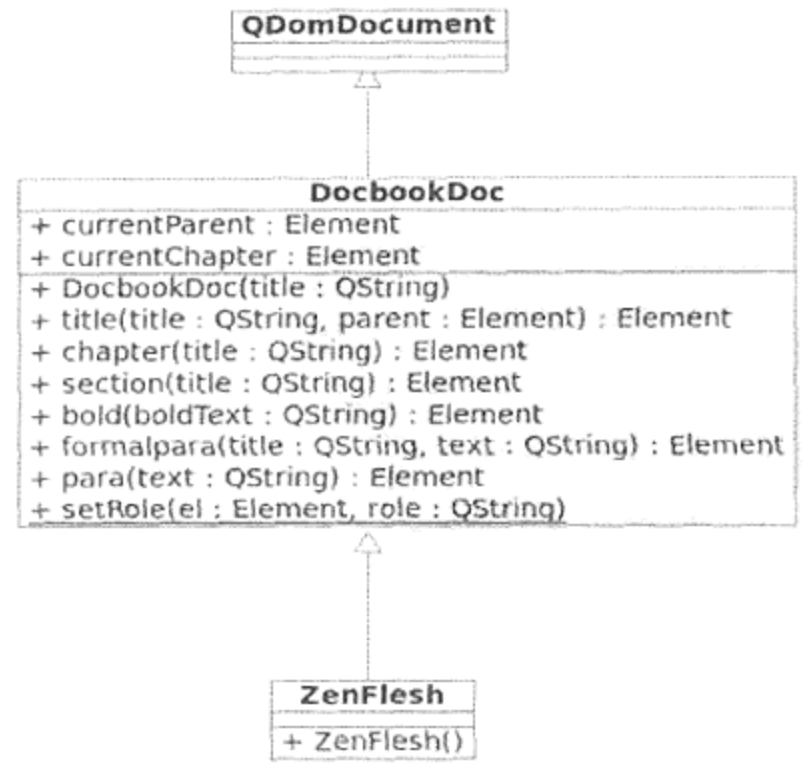


图 15.6 DocbookDoc

图 15.6 中，DocbookDoc 是 QDomElements 的一个工厂类，它从 QDomDocument 派生而来，专门用来创建 Docbook/XML 文档。

在这个类的头文件中，正如示例 15.11 中节选的部分代码一样，我们添加了一个 typedef 语句来提高代码的可读性。在 DOM 标准中，所有的 DOM 类都会分别命名为 Node，Element，Document 和 Attribute。

示例 15.11　src/libs/docbook/docbookdoc.h

```
[ . . . . ]

typedef QDomElement Element;  1
```

1 节省输入且与 Java DOM 一致。

正如示例 15.12 给出的那样，可以通过创建章、节、段落来构建一个文档。

示例 15.12　src/xml/dombuilder/zenfl esh.cpp

```
#include <QTextStream>
#include <docbookdoc.h>

class ZenFlesh : public DocbookDoc {
    public: ZenFlesh();
};

ZenFlesh::ZenFlesh() :
    DocbookDoc("Zen Flesh, Zen Bones") {

    chapter("101 Zen Stories");
    section("A cup of tea");
    para("Nan-in served a cup of tea.");
    section("Great Waves");
    QDomElement p = para("o-nami the wrestler sat in meditation and "
            "tried to imagine himself as a bunch of great waves.");
    setRole(p, "remark");
    chapter("The Gateless Gate");
    formalpara("The Gateless Gate",
      "In order to enter the gateless gate, you must have a ");
    bold(" mindless mind.");

    section("Joshu's dog");
    para("Has a dog buddha nature or not?");

    section("Haykujo's Fox");
    QDomElement fp = formalpara("This is a special topic",
        "Which should have a role= remark attribute");
     setRole(fp, "remark");
}

int main() {
   QTextStream cout(stdout);
   ZenFlesh book;
   cout << book << endl;
}
```

构造函数使用 XML 生成了一本小书，在经过优美的印刷之后，其结果会类似于示例 15.13。

示例 15.13　src/xml/zen.xml

```
<book>
    <title>Zen Flesh, Zen Bones</title>
    <chapter>
```

```
        <title>101 Zen Stories</title>
        <section>
            <title>A cup of tea</title>
            <para>Nan-in served a cup of tea.</para>
        </section>
        <section>
            <title>Great Waves</title>
            <para>
            o-nami the wrestler sat in meditation and tried
            to imagine himself as a bunch of great waves.
            </para>
        </section>
    </chapter>
    <chapter>
        <title>The Gateless Gate</title>
        <formalpara>
            <title>The Gateless Gate</title>
            In order to enter the gateless gate,
            you must have a <emphasis role="strong">
            mindless mind</emphasis>
        </formalpara>
        <section>
            <title>Joshu's dog</title>
            <para>Has a dog buddha nature or not?</para>
        </section>
        <section>
            <title>Haykujo's Fox</title>
            <formalpara role="remark">
                <title>This is a special topic</title>
                Which should have a role="remark" attribute
            </formalpara>
        </section>
    </chapter>
</book>
```

这种格式的优势是可以使用 xsltproc 和 Docbook/XSL 样式表[docbookxsl]等工具轻松地将它转换成 HTML[①](也可以转换成 PDF 或者 LaTeX)。示例 15.14 给出了生成 HTML 版本的做法。

示例 15.14 src/xml/zen2html

```
#!/bin/sh
# Translates zen.xml into index.html.
# Requires gnu xsltproc and docbook-xsl.
# DOCBOOK=/usr/share/docbook-xsl
xsltproc $DOCBOOK/html/onechunk.xsl zen.xml
```

现在，可以看一看创建元素的示例 15.15。Docbook 语言的每一个主要元素在 DocbookDoc 中都有一个对应的工厂方法。

示例 15.15 src/libs/docbook/docbookdoc.cpp

```
[ . . . . ]

Element DocbookDoc::bridgehead(QString titleStr) {
```

① 参见../docs/src/xml/zen.html。

```
    Element retval = createElement("bridgehead");
    Element titleEl = title(titleStr);
    currentParent.appendChild(retval);
    return retval;
}
Element DocbookDoc::title(QString name, Element parent) {
    Element retval = createElement("title");
    QDomText tn = createTextNode(name);
    retval.appendChild(tn);
    if (parent != Element())
        parent.appendChild(retval);
    return retval;
}

Element DocbookDoc::chapter(QString titleString) {
    Element chapter = createElement("chapter");
    title(titleString, chapter);
    documentElement().appendChild(chapter);
    currentParent = chapter;
    currentChapter = chapter;
    return chapter;
}

Element DocbookDoc::para(QString textstr) {
    QDomText tn = createTextNode(textstr);
    Element para = createElement("para");
    para.appendChild(tn);
    currentParent.appendChild(para);
    currentPara = para;
    return para;
}
```

另外，还有一些仅仅修改文本的字符级别的元素，示例 15.16 中给出了它们的具体示例。

示例 15.16　src/libs/docbook/docbookdoc.cpp

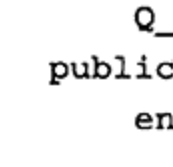
```
[ . . . . ]

Element DocbookDoc::bold(QString text) {
    QDomText tn = createTextNode(text);
    Element emphasis = createElement("emphasis");
    setRole(emphasis, "strong");
    emphasis.appendChild(tn);
    currentPara.appendChild(emphasis);
    return emphasis;
}

void  DocbookDoc::setRole(Element el, QString role) {
    el.setAttribute("role", role);
}
```

因为每个 QDomNode 都必须由 QDomDocument 创建，所以扩展 QDomDocument 来编写自己的 DOM 工厂是很有意义的。

取决于创建的是哪一种元素，DocbookDoc 会将新创建的元素作为孩子添加到之前所创建的元素中。

15.4 XML 流

QXmlStreamReader 和 QXmlStreamWriter 为读取和写入 XML 提供了一个更为快速、更为强大的 API。它们不是 Qt 的 XML 模块的一部分，自 Qt 4.3 以来，它们一直在 QtCore 模块中。



为什么不使用“标准”API

DOM 虽然方便和标准，但需要较大的内存和较高的处理器，而且理应可以进一步简化一些。SAX 虽然速度快且有着相对高效的内存，但对解析过程中创建特定标准结构的需求理应更快些，而且 SAX 也不是很简单，因为要恰当地使用它需要能够理解继承关系并恰当地使用一些纯虚函数。

Qt XML 模块中的接口基于 Java 标准中的相应部分，且其他语言中也有这些实现，因此如果之前使用过它们，就很容易上手。但如果已经对它们有所了解，那么上手 XML 解析的其他 API 也很容易，比如这个。

QXmlStreamReader API 并不会生成树结构。示例 15.17 给出了一个类，展示了如何用该 API 来生成一个树结构。该类本身是 QStandardItemModel，会为每个树节点都创建一个 QStandardItem 实例。

示例 15.17 src/xml/streambuilder/xmltreemodel.h

```
[ . . . . ]
class XmlTreeModel : public QStandardItemModel {
    Q_OBJECT
public:
    enum Roles {LineStartRole = Qt::UserRole + 1,
                LineEndRole};                                    1
    explicit XmlTreeModel(QObject *parent = 0);
public slots:
    void open(QString fileName);
private:
    QXmlStreamReader m_streamReader;
    QStandardItem* m_currentItem;
};
[ . . . . ]
```

1 用于 data() 的自定义角色。

示例 15.18 给出了实际用于解析操作的代码。

例 15.18 src/xml/streambuilder/xmltreemodel.cpp

```
[ . . . . ]
void XmlTreeModel::open(QString fileName) {
    QFile file (fileName);                                       1
    if (!file.open(QIODevice::ReadOnly)) {
```

```
        qDebug() << "Can't open file: " << fileName;
        abort();
    }
    m_streamReader.setDevice(&file);                                    2
    while (!m_streamReader.atEnd()) {
        QXmlStreamReader::TokenType tt = m_streamReader.readNext();
        switch (tt) {
            case QXmlStreamReader::StartElement: {
                QString name = m_streamReader.name().toString();
                QStandardItem* item = new QStandardItem(name);
                item->setData(m_streamReader.lineNumber(),
                              LineStartRole);                           3
                QXmlStreamAttributes attrs = m_streamReader.attributes();
                QStringList sl;
                sl << tr("Line# %1").arg(m_streamReader.lineNumber());
                foreach (QXmlStreamAttribute attr, attrs) {
                    QString line = QString("%1='%2'").arg(attr.name().toString())
                                   .arg(attr.value().toString());
                    sl.append(line);
                }
                item->setToolTip(sl.join("\n"));
                if (m_currentItem == 0)
                   setItem(0, 0, item);                                 4
                else
                   m_currentItem->appendRow(item);                      5
                m_currentItem = item;
                break; }
            case QXmlStreamReader::Characters: {
                QString tt = m_currentItem->toolTip();
                tt.append("\n");
                tt.append(m_streamReader.text().toString());
                m_currentItem->setToolTip(tt);
                break; }
            case QXmlStreamReader::EndElement:
                m_currentItem->setData(m_streamReader.lineNumber(), LineEndRole);
                m_currentItem = m_currentItem->parent();                6
                break;
            case QXmlStreamReader::EndDocument:
            default:
                break;
        }
    }
}
```

1 不在作用域时，自动关闭。

2 开始解析——可以使用任何 QIODevice。

3 自定义 data()。

4 在模型中设置 root 项。

5 作为子节点添加到当前项中。

6 遍历树。

如果运行这个例子，可以看到左侧的 QTreeView 会与右侧文本编辑器中 XML 文件结构的各个元素相互匹配，与图 15.7 类似。鼠标放在树中的元素上，就会显示其中的文字，单击该元素，文本编辑器中的光标会移动到文档中相应的位置。

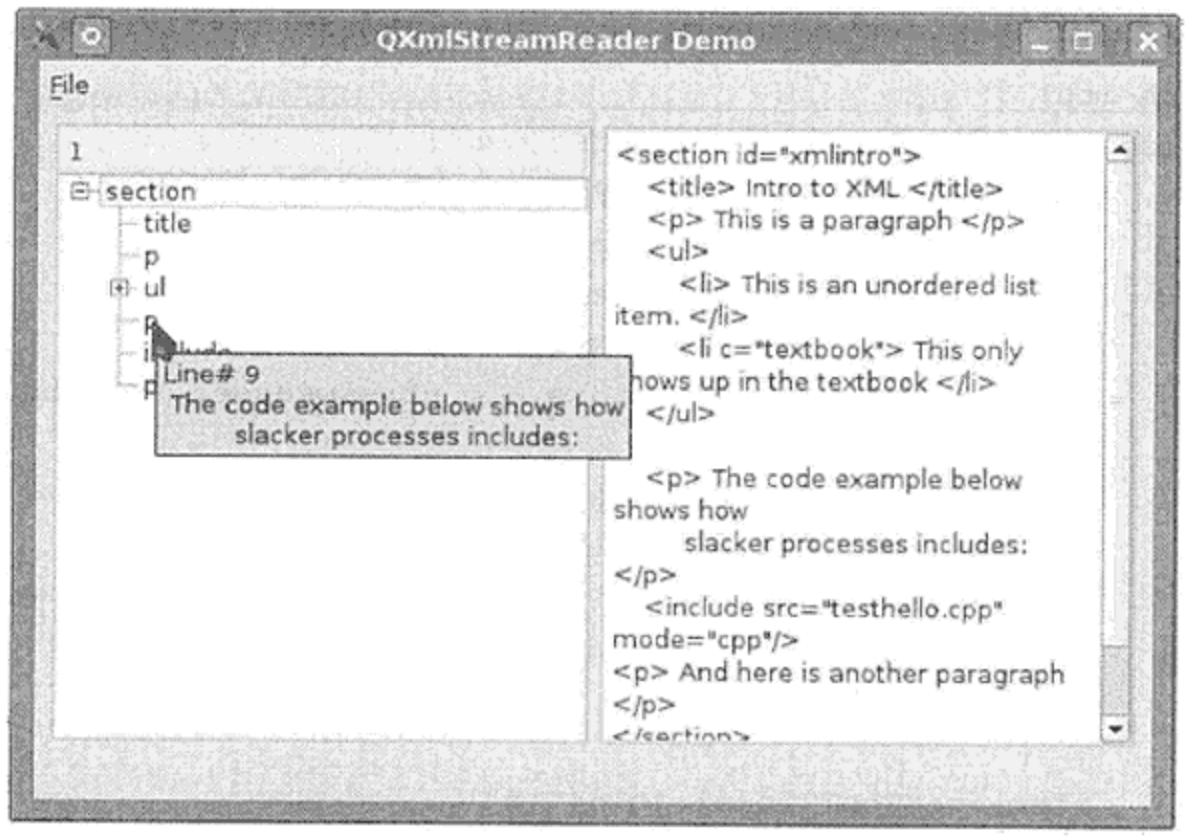


图 15.7 TreeBuilder 示例

15.5 复习题

1. 如果在 XML 文件中存在一个语法错误，那么应该如何判断错误的起因和位置?
2. SAX 是一个事件驱动解析器。它可以响应哪些种类的事件?
3. 对 SAX 和 DOM 做比较。为什么要选择一个而不选择另一个?
4. 如果有一个 QDomNode 且它实际上“指向”QDomElement，那么应该如何引用这个 QDomElement?
5. 使用 QXmlStreamReader 而不是 SAX 的好处是什么?
6. 使用 QXmlStreamReader 而不是 DOM 的好处是什么?

第 16 章　更多的设计模式

本章将介绍来自于两个类别的设计模式：创建模式(creational)和结构模式(structural)。

16.1　创建模式

使用管理对象创建模式，可以获得某种灵活性，使得在运行时能够挑选或者改变所创建或使用的对象的种类，并可用来管理对象的删除，尤其是在大型软件系统中更是如此。正确管理对象的创建，是管理不同的代码层的关键部分，也是确保对象可被正确销毁的重要一环。

C++中，工厂是一种程序组件(通常为一个类)，它负责对象的创建。工厂的思想是将对象创建与对象使用分离开来。

工厂类通常具有一个函数，它获得用于新对象的动态内存，并返回一个指向这个对象的(基类)指针。这种方法使得能够在不改变使用工厂的代码的情况下引入新的派生类型。

这一节将探讨几种使用工厂的设计模式，并会给出这些设计模式的一些例子。

当创建堆对象的责任委托给一个虚函数时，就称它是一个工厂方法。

如果让工厂方法为纯虚方法，并且编写了具体的派生工厂类，就使工厂基类成为了抽象工厂。

> 抽象工厂模式提供的接口用于定义一些工厂，它们共享抽象的特性但有不同的具体细节。客户代码可以实例化某个特定的子工厂，然后用抽象接口来创建对象。

通过强制使用防止直接实例化类的创建规则，就可以迫使客户代码只能使用工厂方法来创建所有的实例。例如，可能需要限制一次可以存在的某个特定类的对象数量。事实上，所用到的类只被严格地实例化一次是相当常见的。

> 单一模式(singleton pattern)可将类限制成只能创建一个实例。实现方式是让其构造函数为 `private` 类型或者 `protected` 类型，并且提供一个 `instance()` 成员函数。如果类的实例不存在，则这个成员函数返回一个指向新实例的指针，否则返回一个指向已存在实例的指针。

8.3 节中已经看到过并使用过单一模式。使用事件循环的 Qt 应用会首先实例化 `QApplication`，创建另外一些对象，然后调用 `QApplication::exec()`，启动事件循环。为了在程序的其他地方引用这个对象，可以使用 `qApp` 宏，它会返回一个指向这个单一 `QApplication` 模式的指针[①]。16.1.4 节中将更进一步探讨单一模式。

对象创建

考虑如下创建派生的 `QObject` 类 `Customer` 实例的三种方式：

---

① `QApplication` 派生自 `QCoreApplication`。当包含<QApplication>时，就是包含了宏`#define qApp (static_cast< QApplication *>(QCoreApplication ::instance()))`。`QCoreApplication: :instance()` 返回一个已定义实例的指针，或者如果没有创建实例的话，则返回 `null`。

- `Customer* c1 = new Customer(name);`
- `QMetaObject meta = Customer::staticMetaObject; Customer* cust = qobject_cast<Customer*>(meta.newInstance());`
- `Customer* cust = CustomerFactory::instance()->newCustomer(name);`

第一种情形中，类名称被硬编码到构造函数调用中，对象将在内存中用默认的堆存储创建。在客户代码中硬编码类名称，会限制代码的可复用性和灵活性。

第二种情形中，调用的是 QMetaObject::newInstance()。这是一个抽象工程，在每一个 QMetaObject 派生类中都会被重写，由 moc.newInstance()返回的 QObject 指向 Customer 的一个新实例。

第三种情形中，使用一个名称为 CustomerFactory::newCustomer()的专门化工厂方法，间接构造了一个 Customer 对象。与 Qt 库相比，这个接口使用起来会更加方便，但它只不过是另一个工厂的封装器。

在 QMetaObject::newInstance()之前[①]，有必要编写一条 switch 语句来处理工厂中所支持的每一个类。

16.1.1 抽象工厂

AbstractFactory(抽象工厂)的定义见示例 16.1，它是一个简单类。

示例 16.1 src/libs/dataobjects/abstractfactory.h

```
[ . . . . ]
class DOBJS_EXPORT AbstractFactory
{
  public:
    virtual QObject* newObject (QString className,
                                QObject* parent=0) = 0;
    virtual ~AbstractFactory();

};
[ . . . . ]
```

newObject()为一个纯虚方法，因此必须在派生类中重写它。示例 16.2 中给出了一些派生自 AbstractFactory 的具体类。

示例 16.2 src/libs/dataobjects/objectfactory.h

```
[ . . . . ]
class DOBJS_EXPORT  ObjectFactory : public AbstractFactory {
  public:
    ObjectFactory();
    virtual QObject* newObject (QString className, QObject* parent=0);
 protected:
    QHash<QString, QMetaObject> m_knownClasses;
};
[ . . . . ]
```

① 它是在 Qt 4.5 中引入的。

QMetaObject::newInstance()

通过将某个构造函数标记为 Q_INVOKABLE，就可以用 QMetaObject::newInstance() 创建派生自 QObject 类的实例。QMetaObject::newInstance()方法本身也是抽象工厂模式的一个例子。

前面已经定义了 ObjectFactory，这样它就知道如何创建两种具体类型。由于能够提供给 newObject()的可以是任何 QString，所以 ObjectFactory 需要处理传递的类未知的情况。这种情况下，它返回一个泛型 QObject 的指针并会设置动态属性 className，以便其他函数(例如，XML 导出例程)知道这个对象正在“欺骗”另一个类中的对象。

示例 16.3　src/libs/dataobjects/objectfactory.cpp

```
[ . . . . ]
ObjectFactory::ObjectFactory() {
    m_knownClasses["UsAddress"] = UsAddress::staticMetaObject;
    m_knownClasses["CanadaAddress"] = CanadaAddress::staticMetaObject;
}

QObject* ObjectFactory::newObject(QString className, QObject* parent) {
    QObject* retval = 0;
    if (m_knownClasses.contains(className)) {
        const QMetaObject& mo = m_knownClasses[className];
        retval = mo.newInstance();                                    1
        if (retval == 0) {
            qDebug() << "Error creating " << className;
            abort();
        }
    } else {
        qDebug() << QString("Generic QObject created for new %1")
                    .arg(className);
        retval = new QObject();
        retval->setProperty("className", className);
    }
    if (parent != 0) retval->setParent(parent);
    return retval;
}
```

1 要求 Qt 4.5 或者更高的版本。

示例 16.4 中给出的例子将 Q_INVOKABLE 宏用于 UsAddress 类的一个构造函数中。

示例 16.4　src/libs/dataobjects/address.h

```
[ . . . . ]

class DOBJS_EXPORT UsAddress : public Address {
    Q_OBJECT
  public:
    Q_PROPERTY( QString State READ getState WRITE setState );
    Q_PROPERTY( QString Zip READ getZip WRITE setZip );
    explicit Q_INVOKABLE UsAddress(QString name=QString(), QObject* parent=0)
```

```
                                        : Address(name, parent) {}
  protected:
    static QString getPhoneFormat();
  public:
[ . . . . ]

  private:
    QString m_State, m_Zip;
};
```


16.1.2 抽象工厂和库

现在考虑两个库：libdataobjects 和 libcustomer，它们都具有自己的(具体)对象工厂，如图 16.1 中的 UML 框图所示。示例 16.5 中定义的 CustomerFactory 扩展了 ObjectFactory 的功能性。

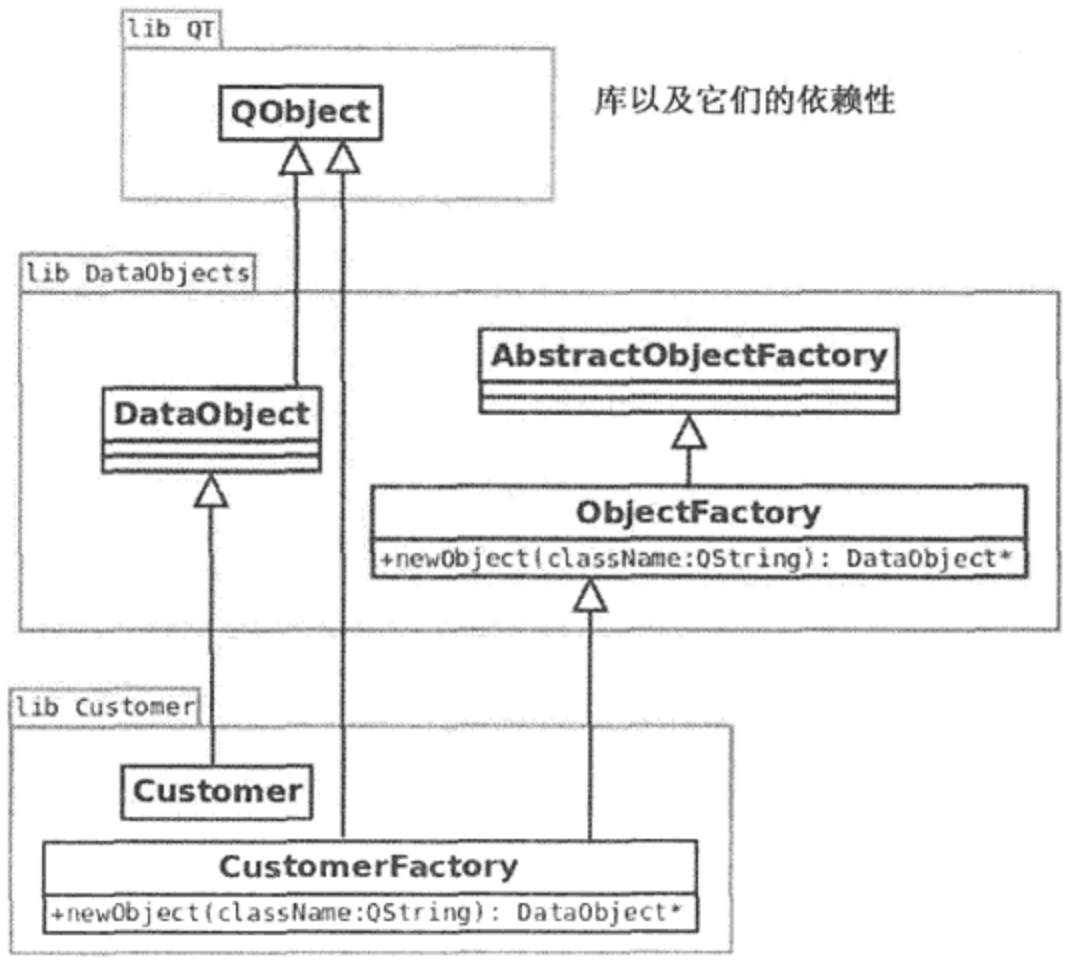


图 16.1 库与工厂

示例 16.5 src/libs/customer/customerfactory.h

```
[ . . . . ]
class CUSTOMER_EXPORT CustomerFactory :
                              public QObject, public ObjectFactory {
  public:
    static CustomerFactory* instance();                              1
    Customer* newCustomer(QString name, QObject* parent=0);          2
    Address* newAddress(QString countryType = "USA", QObject* parent=0);
  private:
    CustomerFactory(QObject* parent=0);
};
[ . . . . ]
```

1 单一工厂方法。

2 常规的工厂方法，不要求类型转换。

CustomerFactory 从 ObjectFactory 继承了创建 Address 对象的能力。此外，它还知道如何创建 Customer 对象。CustomerFactory 只需要在构造函数中将一些 QMetaObject 添加到 ObjectFactory::m_knownClasses 中(这是可能的，因为这个容器为 protected 类型)。基类 newObject()方法足够聪明，能够处理由 Customer 添加的类，只要它们被正确地在容器中被注册了。

示例 16.6　src/libs/customer/customerfactory.cpp

```
[ . . . . ]


CustomerFactory::CustomerFactory(QObject* parent) : QObject(parent) {
    m_knownClasses["Customer"] = Customer::staticMetaObject;
    m_knownClasses["CustomerList"] = Customer::staticMetaObject;
}
```

16.1.3　qApp 和单一模式

正如以前所讨论的，单一模式是一种专门化的工厂，用于希望限制所创建的实例数量或者类型的情形。

示例 16.7 中定义的 CustomerFactory::instance()方法，就是单一模式的一个例子。在需要时它会创建一个对象，但是只有在首次调用这个方法时才会这样做。对它的后续调用，会总是返回一个指向同一对象的指针。

示例 16.7　src/libs/customer/customerfactory.cpp

```
[ . . . . ]

CustomerFactory* CustomerFactory::instance() {
    static CustomerFactory* retval = 0;
    if (retval == 0) retval = new CustomerFactory(qApp);          1
    return retval;
}
```

1 当 QApplication 退出时，应确保这个对象以及它的全部子对象都被清除了。

当处理堆对象时，千万注意不要产生内存泄漏。可以利用 QObject 的父-子关系来防止这种情况出现。

正如前面提到的，qApp 是一个指向 QApplication 单一实例的指针，这个实例假定是在 main()中创建的。QApplication 实例的存在时间与应用运行的时间一致。

为什么不使用 static 父对象

如果堆对象的父对象为 static QObject，则它的子对象将在 QApplication 被销毁之后才销毁。除非存在足够的理由，否则在销毁 QApplication 之后，程序不应当对 QObject 有任何动作，包括对象清理。对于使用多个文件的应用程序而言，来自于不同文件的 static 对象是按照它们的链接器依赖性顺序被销毁的。这种销毁顺序可能导致无意的副作用(例如，终止时的段错误)。更多细节，请参见 8.6 节。

16.1.4 使用工厂的好处

工厂模式的好处之一是：可以在一个池(pool)中管理所创建的对象(复用可以被复用的那些对象)。

间接的对象创建还可以在运行时决定要创建哪些类对象。这样就使得可以进行替换类的“插入”，而不必要求客户代码有任何改变。16.2 节中给出的一个方法示例，它根据 XML 文件的内容使用工厂对象来创建链接的、客户定义的对象树。在 src/libs/metadata/abstractmetadataloader.h 中有工厂方法的另一个示例，它管理 MetaDataLoader 的单一实例，从而不必改变代码就能够轻易编写出在派生的元数据加载器之间切换的程序。

库和插件

建立大型系统时，一种好的做法是将库设计成包含共享某些共性或者需要一起使用的类。大量的应用都会使用来自于多个库的组件，这些库中的一些是由开发团队提供的，而另一些是由第三方开发人员提供的（例如，诺基亚的 Qt)。只有库类的公共接口才能出现在复用它们的客户代码中。库设计人员在改动库类的实现时，不应当导致客户代码的破坏。

插件（plugin）是一种集成的软件组件集，它为大型应用添加特定的功能。插件的一个例子是 Adobe Flash Player，它使得各种 Web 浏览器都能够显示某些类型的视频。好的插件支持使得用户能够定制应用的功能性。许多库通过发布接口、提供如何实现的文档，从而能够在类之外被插入。通过提供一个工厂基类，专门化的工厂类根据需要从这个工厂基类派生，这样就使得库方便了插件类的创建。

工厂方法的另一个好处（即通常所说的间接对象创建）是：可以强制对象在构造函数之后被初始化，包括 virtual 函数（虚函数）的调用。

来自于构造函数的多态

在构造函数完成执行之前，对象不会被认为是已经“完全构造的”。在构造函数执行完毕之前，对象的虚指针(vpointer)不会指向正确的虚表(vtable)。因此，从构造函数对这种对象调用方法不能使用多态。

当在对象初始化期间需要多态行为时，就要求使用工厂方法。示例 16.8 中就是这种情况。

示例 16.8 src/ctorpoly/ctorpoly.cpp

```
#include <iostream>
using namespace std;

class A {
public:
    A() {
        cout << "in A ctor" << endl;
        foo();
    }
    virtual void foo() {
        cout << "A's foo()" << endl;
    }
};
```

```
class B: public A {
public:
    B() {
        cout << "in B ctor" << endl;
    }
    void foo() {
        cout << "B's foo()" << endl;
    }
};

class C: public B {
public:
    C() {
        cout << "in C ctor" << endl;
    }
    void foo() {
        cout << "C's foo()" << endl;
    }
};

int main() {
    C* cptr = new C;
    cout << "After construction is complete:" << endl;
    cptr->foo();
    return 0;
}
```

其输出见示例 16.9。

示例 16.9　src/ctorpoly/ctorpoly-output.txt

```
src/ctorpoly> ./a.out
in A ctor
A's foo()
in B ctor
in C ctor
After construction is complete:
C's foo()
src/ctorpoly>
```

值得注意的是，当构造一个新的 C 对象时，调用的是错误的 foo() 版本。22.1 节中将详细探讨虚表。

16.1.5　练习：创建模式

1. 完成 Address，Customer 以及 CustomerList 类的实现。
   将 7.4.1 节中讲解的思想用于编写 CustomerWriter 类。确保使用了 Customer 和 Address 的 Q_PROPERTY 特性，以便当改变 Customer 类的实现时不必重写 CustomerWriter 类。
   要记住的是，Address 对象是作为 Customer 的子对象保存的。以下是值得考虑的一种输出格式：

```
Customer {
    Id=83438
    DateEstablished=2004-02-01
    Type=Corporate
    objectName=Bilbo Baggins
    UsAddress {
        Line1=52 Shire Road
        Line2=Suite 6
        City=Brighton
        Phone=1234567890
        State=MA
        Zip=02201
        addressName=home
    }
}
```


如果使用 Dataobject ::toString()函数，则会出现另一种情况。

2. 编写一个 CustomerReader 类，它通过复用所提供的 CustomerFactory 类来创建全部的新对象。

   编写一个 CustomerListWriter 类和一个 CustomerListReader 类，它们分别序列化和解序列化 Customer 对象的列表。编写它们时能够复用多少 CustomerWriter 类和 CustomerReader 类的代码?

   编写客户代码，测试这些类。

16.2 备忘录模式

这一节将 QMetaObject 与 SAX2 解析器结合起来，以展示如何编写通用的 XML 编码/解码工具，让其用于带有定义良好的 Q_PROPERTY 的 QObject 及其子对象。这样就可以将元对象模式与备忘录模式(Memento 模式)结合在一起使用。而且由于 XML 和 QObject 都能够表示层次结构，所以可以将这些想法与组合(Composite)模式以及抽象工厂模式结合在一起，以保存和加载全部的多态对象树。

这一节的目标是为许多不同类型的类提供序列化器和解序列化器，其中的编码和解码是由能够操作 QMetaObject 的方法处理的，处理时按照模型逻辑进行区分。

为了将 QObject 树编码和解码成 XML，必须定义一个映射模式，这种映射不仅必须获得 QObject 的属性、类型和值，而且还要知晓对象及其子对象之间已有的关系，每一个子对象以及它们的子对象之间的关系，等等。

XML 元素之间的父-子关系可以自然地映射成 QObject 的父-子关系。这些关系就定义了一种树结构。

前面说过，XML 和 QObject 都已经使用了组合模式，以支持对象的树状层次。图16.2中给出的 Customer 和 CustomerList 都派生自 QObject。这里使用组合模式，将 QObject 子对象映射成 XML 中对应的子元素。

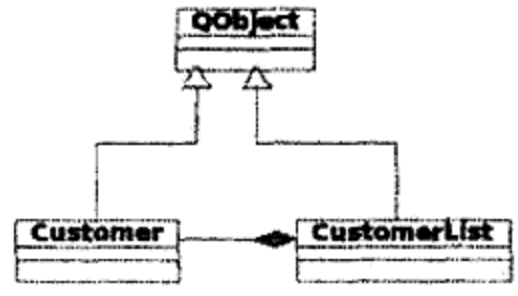


图 16.2 CustomerList 的 UML 框图

示例 16.10 中给出了一种所期望的 XML 格式，用于保存 CustomerList 的数据。

示例 16.10　src/xml/propchildren/customerlist.xml

```
<object class="CustomerList" name="Customers" >

   <object class="Customer" name="Simon" >
     <property name="Name" type="QString" value="Simon" />
     <property name="Date" type="QDate" value="1963-11-22" />
     <property name="LuckyNumber" type="int" value="834" />
     <property name="State" type="QString" value="WA" />
     <property name="Zip" type="QString" value="12345" />
     <property name="FavoriteFood" type="QString" value="Donuts" />
     <property name="FavoriteDrink" type="QString" value="YooHoo"/>
   </object>
   <object class="Customer" name="Raja" >
     <property name="Name" type="QString" value="Raja" />
     <property name="Date" type="QDate" value="1969-06-15" />
     <property name="LuckyNumber" type="int" value="62" />
     <property name="State" type="QString" value="AZ" />
     <property name="Zip" type="QString" value="54321" />
     <property name="FavoriteFood" type="QString" value="Mushrooms" />
     <property name="FavoriteDrink" type="QString" value="Jolt" />
   </object>

</object>
```

如果输入文件包含的是这种类型的信息，则不仅完全可以重新构造对象的属性以及类型，而且还可以重新定义 CustomerList 中各种 QObject 之间的父-子关系树结构。

16.2.1　导出到 XML

示例 16.11 中给出了一个反射递归函数 toString()，它为每一个对象的属性构造一个字符串，然后对该对象的子对象进行迭代，递归地对每个子对象调用 toString()函数。

> **备忘录模式**
>
> 如果对象的内部状态被捕获且被固定以便今后能够恢复时，这就是备忘录模式的一种实现。

示例 16.11　src/libs/dataobjects/qobjectwriter.cpp

```
[ . . . . ]
QString QObjectWriter::
toString(const QObject* obj, int indentLevel) const {
    QStringList result;
    QString indentspace;
    indentspace.fill(' ', indentLevel * 3 );
    QString className = obj->metaObject()->className();
    QString objectName = obj->objectName();
    QStringList propnames = propertyNames(obj);

    foreach (const QString &propName, propnames) {
        if (propName == "objectName") continue;
        QVariant qv = obj->property(propName.toAscii());
```

```
        if (propName == "className") {
                className = qv.toString();
                continue;
        }
        const QMetaObject* meta = obj->metaObject();
        int idx = meta->indexOfProperty(propName.toAscii());
        QMetaProperty mprop = meta->property(idx);

        result <<
        QString("%1  <property name=\"%2\" type=\"%3\" value=\"%4\" />")
            .arg(indentspace).arg(propName).
            arg(qv.typeName()).arg(toString(qv, mprop));
}
/* Query over QObjects */
if (m_children) {
        QList<QObject*> childlist =
                qFindChildren<QObject*>(obj, QString());

        foreach (const QObject* child, childlist) {
            if (child->parent() != obj) {
      //            qDebug() << "This is not my child!!";
                continue;
            }
            if (child != 0) {
                result << toString(child, indentLevel+1);
            }
        }
    }

    result.insert(0, QString("\n%1<object class=\"%2\" name=\"%3\" >")
        .arg(indentspace).arg(className).arg(objectName));
    result << QString("%1</object>\n").arg(indentspace);
    return result.join("\n");
}
[ . . . . ]
```

示例 16.11 中给出的 `toString()` 函数利用了 Qt 的属性和 `QMetaObject` 的便利性来反射类。迭代时，它会将每一行追加到 `QStringList` 的末尾。当迭代完成时，会关闭<object>。然后，通过调用 `QStringList::join("\n")`，可以快速获得返回的 `QString`。

16.2.2 导入具有抽象工厂的对象

先修课程

要求先阅读 15.2 节。

导入例程要比导出例程复杂得多，而且它具有几个有趣的特性。

- 它使用 SAX 解析器解析 XML。
- 它根据输入的情况来创建对象。

● 对象的数量和类型以及它们之间的父-子关系必须根据文件中的信息重新构造。

示例 16.12 中包含了 QObjectReader 类的定义。

示例 16.12　src/libs/dataobjects/qobjectreader.h

```
[ . . . . ]
#include "dobjs_export.h"
#include <QString>
#include <QStack>
#include <QQueue>
#include <QXmlDefaultHandler>

class AbstractFactory;
class DOBJS_EXPORT  QObjectReader : public QXmlDefaultHandler {
  public:
    explicit QObjectReader (AbstractFactory* factory=0) :
                        m_Factory(factory), m_Current(0) { }
    explicit QObjectReader (QString filename, AbstractFactory* factory=0);
    void parse(QString text);
    void parseFile(QString filename);
    QObject* getRoot();
    ~QObjectReader();
    // callback methods from QXmlDefaultHandler
    bool startElement( const QString& namespaceURI,
                       const QString& name,
                       const QString& qualifiedName,
                       const QXmlAttributes& attributes );
    bool endElement(  const QString& namespaceURI,
                      const QString& localName,
                      const QString& qualifiedName);
    bool endDocument();
  private:
    void addCurrentToQueue();
    AbstractFactory* m_Factory;
    QObject* m_Current;
    QQueue<QObject*> m_ObjectList;
    QStack<QObject*> m_ParentStack;
};
[ . . . . ]
```

图 16.3 中给出了这个示例中各种类之间的关系。

QObjectReader 派生自 QXmlDefaultHandler，后者是用于 QXmlSimpleReader 的一个插件。AbstractFactory 是用于 QObjectReader 的一个插件。当创建 QObjectReader 时，必须将它应用到 ObjectFactory 或者 DataObjectFactory 的具体实例。

现在，QObjectReader 就完全与它能够创建的对象的特定类型分开了。为了将 QObjectReader 用于自己的类型，只需从 AbstractFactory 派生出一个工厂即可。

要注意的是，当阅读根据示例 16.10 中的 XML 输出文件构造对象的代码时，应从示例 16.13 开始。

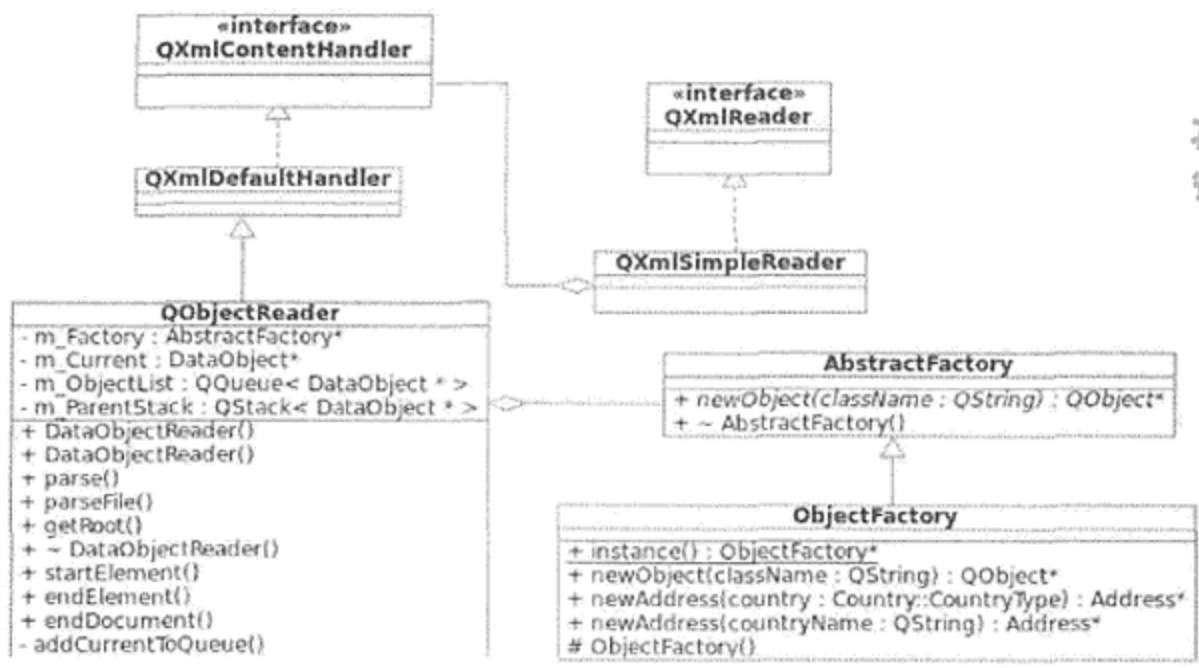


图 16.3 QObjectReader 以及它的相关类

示例 16.13 src/libs/dataobjects/qobjectreader.cpp

```
[ . . . . ]

bool QObjectReader::startElement( const QString&,
     const QString& elementName, const QString&,
     const QXmlAttributes& atts) {                                    1
    if (elementName == "object") {
        if (m_Current != 0)                                           2
            m_ParentStack.push(m_Current);                            3
        QString classname = atts.value("class");
        QString instancename = atts.value("name");
        m_Current = m_Factory->newObject(classname);
        m_Current->setObjectName(instancename);
        if (!m_ParentStack.empty()) {                                 4
            m_Current->setParent(m_ParentStack.top());
        }
        return true;
    }
    if (elementName == "property") {
        QString fieldType = atts.value("type");
        QString fieldName = atts.value("name");
        QString fieldValue = atts.value("value");
        QVariant qv = QVariant(fieldValue);
        m_Current->setProperty(fieldName.toAscii(), qv);

    }
    return true;
}
```

1 不需要命名未使用的参数。

2 判断是否在某个对象里。

3 将前一个流压入栈中。

4 ParentStack 栈顶或者前一个流应为父对象。

当 SAX 解析器遇到 XML 元素的初始标记时，就会调用 startElement() 函数。这个函数的参数包含创建对象所需的全部信息。在 startElement() 函数与对应的 endElement()

函数之间遇到的所有其他对象，都是 m_Current 的子对象。当到达 endElement() 函数时，这个对象就算“完成”了，如示例 16.14 所示。

示例 16.14　src/libs/dataobjects/qobjectreader.cpp

```
[ . . . . ]

bool QObjectReader::endElement( const QString& ,
                                const QString& elementName,
                                const QString& ) {
    if (elementName == "object") {
        if (!m_ParentStack.empty())
            m_Current = m_ParentStack.pop();
        else {
            addCurrentToQueue();
        }
        return true;
    }
```

QObjectReader 使用抽象工厂来进行实际的对象创建工作。

回调函数 newObject(QString className) 创建的对象能够保存 className 中描述的全部属性。ObjectFactory 会为具有 QMetaObject 的类型正确地创建对象。对于其他的类，它会创建常规的 QObject，但会通过设置它的 className 动态属性来“模仿”这种类型。为了支持自己的类型，可以编写一个具体派生工厂，将正确的字符串映射成 QMetaObject。

16.3　Façade 模式

使用 Façade 模式的类“为子系统中的接口集提供一个统一的接口。Façade 定义的高级接口使得更容易(且更安全地)使用子系统”[Gamma95]。如果类接口由于太复杂而无法有效地使用时(导致难以调试错误)，或者没有使用与大型框架相适应的编程风格，则应使用 Façade 模式。Façade 模式的另一种用法是将使用某个特定库的代码与应用的其他部分区别开，以达到减少库间依赖性的目的。Façade 是一个类(或者类集)，它具有清晰、简单的接口，封装且隐藏了复杂的类集或者函数集。两种紧密相关的设计模式是封装器封装器(wrapper)和适配器(adaptor)，其中一种设计模式的某些例子，可以用另一种设计模式体现。

许多核心 Qt 类都是具有一些用于不同平台的原始类的 Façade 设计模式，它们的实现要比表象复杂得多。例如，QString 是一个具有可增长的、被隐式共享的字符数组的封装器；QWidget 是一个具有原始窗件的封装器；而 QThread，QFile，QProcess 和 QSqlDatabase 是具有低级库的封装器，这些库根据平台的情况有相当不同的实现。如果只通过 Qt API 使用这些封装器，则代码就可以运行于全部平台上。

从复用具有不同接口的类中获得的经验，能够为设计自己的类中成熟的、友好的、有用的接口提供有价值的帮助。下面的这个示例探讨了用于多媒体元数据系统的 Qt 封装器。

多媒体文件包含音视频内容：声音效果、音乐、图形图像、动画、电影等。元数据是描述多媒体文件音视频内容的信息。元数据经常被称为标签数据，或简称为标签。

多媒体文件通常将元数据保存在包含媒体的同一个文件中。通过标签信息，媒体播放器可以显示关于当前要播放或即将播放的文件的相关信息(例如，标题、艺术家、风格等)。也可以用它来将文件组织成集合，或者创建播放列表。

有许多库都能用了读取文件中的元数据。本书的前一个版本使用的是 id3lib，但它已经不再被支持了。现在使用的是 Phonon，它是在 Qt 4.4 中增加的。Phonon 支持多种文件格式，但是不能用于 Windows 下的 MP3 文件，而且它只能读取而不能回写标签数据。还有一个 Taglib 1.6 库，它与 id3lib 类似但支持现代的文件格式，且可用于 Windows, Linux 和 Mac OS/X 系统。许多开源工具都使用 TagLib，比如 Amarok 和 kid3。

本书中的代码已经被组织成根据应用的情况能够在 Phonon 和 TagLib 之间轻易切换。图 16.4 中给出了 MetaDataLoader 和 MetaDataValue 彼此相互适应的方式，只需在应用与所使用的底层元数据库之间提供一个 Façade 封装器即可。如果希望在程序中以某种方式动态地挑选其中之一(也许需根据平台的不同或者所使用的文件格式)，则可以进一步将 phononmetadata 和 filetagger 包装到插件中。

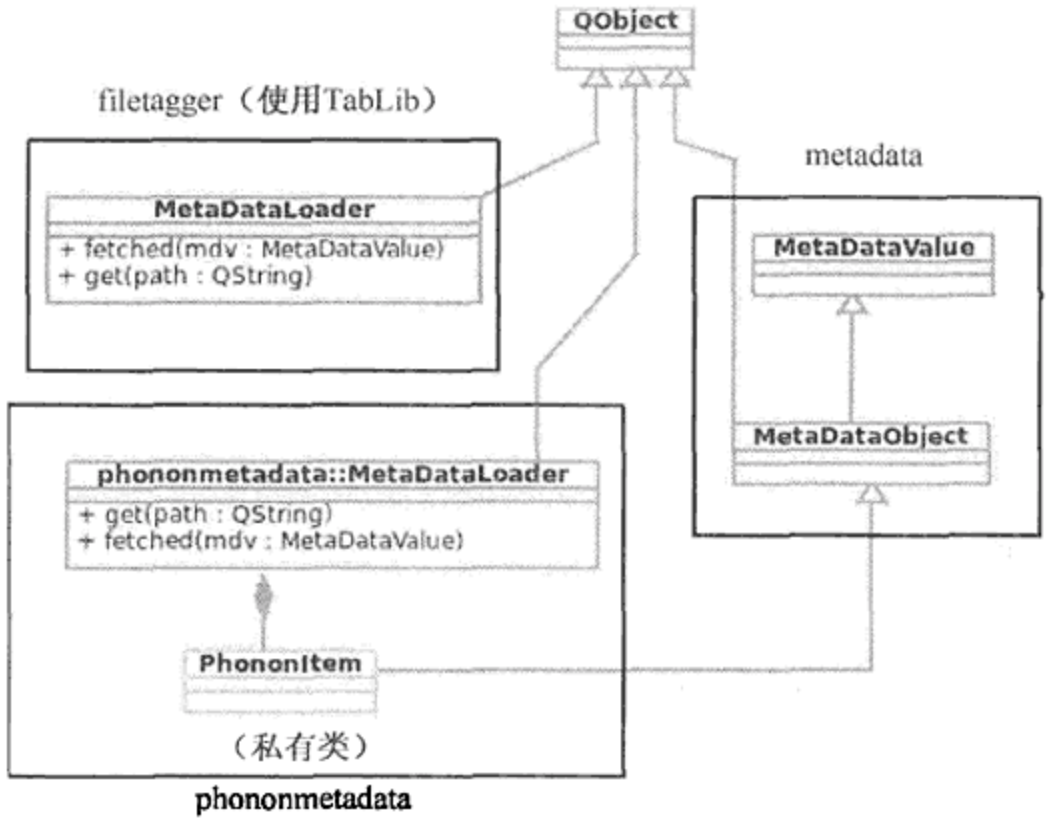


图 16.4 MetaDataLoader 和 MetaDataValue

16.15 中的 MetaDataValue 是用于操作元数据值的一个基类接口。

示例 16.15 src/libs/metadata/metadatavalue.h

```
[ . . . . ]
class METADATAEXPORT MetaDataValue {
public:

    friend METADATAEXPORT QTextStream& operator<< (QTextStream& os,
                                          const MetaDataValue& mdv);
    friend METADATAEXPORT QTextStream& operator>> (QTextStream& is,
                                          MetaDataValue& mdv);
    friend METADATAEXPORT QDataStream& operator<< (QDataStream& os,
                                          const MetaDataValue& mdv);
    friend METADATAEXPORT QDataStream& operator>> (QDataStream& is,
                                          MetaDataValue& mdv);
    friend METADATAEXPORT bool operator==(const MetaDataValue&,
                                           const MetaDataValue&);
```

```
[ . . . . ]
    virtual QString fileName() const ;
    virtual Preference preference() const ;
    virtual QString genre() const;
    virtual QString artist() const;
    virtual QString albumTitle() const;
    virtual QString trackTitle() const;
    virtual QString trackNumber() const;
    virtual const QImage &image() const;
    virtual QTime trackTime() const;
    virtual QString trackTimeString() const;
    virtual QString comment() const;
[ . . . . ]
protected:
    bool m_isNull;
    QUrl m_Url;
    QString m_TrackNumber;
    QString m_TrackTitle;
    QString m_Comment;
    Preference m_Preference;
    QString m_Genre;
    QString m_Artist;
    QTime m_TrackTime;
    QString m_AlbumTitle;
    QImage m_Image;
};
Q_DECLARE_METATYPE(MetaDataValue);                                    1
[ . . . . ]
```

1 添加到 QVariant 类型系统。

MetaDataValue 对象可以来自于磁盘、数据库或者用户的标签。不同的库都提供了一个自定义的 MetaDataLoader 类，它在一个信号参数中包含了这种类型的一个值。

示例 16.16　src/libs/metadata/abstractmetadataloader.h

```
[ . . . . ]
namespace Abstract {
class METADATAEXPORT MetaDataLoader : public QObject {
    Q_OBJECT
public:
    explicit MetaDataLoader(QObject *parent = 0)
        : QObject(parent) {}
    static MetaDataLoader* instance();
    virtual MetaDataLoader* clone(QObject* parent=0) = 0;
    virtual ~MetaDataLoader();
    virtual const QStringList &supportedExtensions() = 0;
    virtual void get(QString path) = 0;
    virtual void get(QStringList path) = 0;
    virtual bool isRunning() const = 0;
public slots:
    virtual void cancel() = 0;

signals:
```

```
        void fetched(const MetaDataValue & mdv);
        void progressValueChanged(int);
        void progressRangeChanged(int, int);
        void finished();

};
}

#endif // AMETADATALOADER_H
```


示例 16.16 中的 AbstractMetaDataLoader 具有一个简单的、无阻断的 get()方法，它会立即返回。它释放出一个 fetched(MetaDataValue)信号，这个信号能够被连接到另一个对象的槽，比如 PlayListModel。示例 16.17 中给出了使用 TagLib 的 MetaDataLoader 具体实现。

示例 16.17 src/libs/filetagger/tmetadataloader.h

```
[ . . . . ]
class FILETAGGER_EXPORT MetaDataLoader
                            : public Abstract::MetaDataLoader {
    Q_OBJECT
public:
    typedef Abstract::MetaDataLoader SUPER;
    explicit MetaDataLoader(QObject *parent = 0);
    static MetaDataLoader* instance();
    virtual ~MetaDataLoader() {}
    const QStringList &supportedExtensions() ;
    MetaDataLoader* clone(QObject *parent) ;
    void get(QString path);
    void get(QStringList path);
    bool isRunning() const {return m_running;}
public slots:
    void cancel();
private slots:
    void checkForWork();
private:
    bool m_running;
    bool m_canceled;
    int m_processingMax;
    QStringList m_queue;
    QTimer m_timer;
};
}

[ . . . . ]
```

示例 16.18 中给出的 checkForWork()方法在一个循环中执行操作，它调用 qApp->processEvents()方法，使得在一个长时间运行的循环中 GUI 依然能够保持响应性，比如这里的情况。

示例 16.18　src/libs/filetagger/tmetadataloader.cpp

```
[ . . . . ]

TagLib::MetaDataLoader::MetaDataLoader(QObject *parent) :
    SUPER(parent) {
    m_processingMax = 0;
    m_running = false;
    qDebug() << "TagLib::MetaDataLoader created.";
    connect (this, SIGNAL(finished()), this, SLOT(checkForWork()),
             Qt::QueuedConnection);
}

void TagLib::MetaDataLoader::get(QString path) {
    m_queue << path;
    m_timer.singleShot(2000, this, SLOT(checkForWork()));
}

void TagLib::MetaDataLoader::checkForWork() {
    MetaDataFunctor functor;
    if (m_queue.isEmpty() && !m_running) {
        m_processingMax = 0;
        return;
    }
    if (m_running ) return;
    m_running = true;
    m_canceled = false;
    QStringList sl = m_queue;
    m_queue = QStringList();
    m_processingMax = sl.length();
    emit progressRangeChanged(0, m_processingMax);
    for (int i=0; i<m_processingMax;++i) {
        if (m_canceled) break;
        emit fetched(functor(sl[i]));
        emit progressValueChanged(i);
        qApp->processEvents();                                    1
    }
    m_running = false;
    emit finished();
}
```

1 使得 GUI 能够处理事件(且信号也能被传送)。

示例 16.19 中能够找到实际的 TagLib 代码，这些代码被放在一个仿函数(functor)中，因为这里是第一次在 QtConcurrent 算法中使用它。经过一些测试后，我们发现仿函数并不是线程安全的，所以这里从循环中依次调用它。这里还完成了 TagLib 类型与 Qt 类型之间的全部转换。客户代码无须关心数据是如何被取出的，也不必担心所使用的字符串库，就可以继续使用来自于 libmetadata 的 MetaDataValue 接口。

示例 16.19　src/libs/filetagger/tmetadataloader.cpp

```
[ . . . . ]

MetaDataValue MetaDataFunctor::operator ()(QString path) {
```

```
    using namespace TagLib;
    MetaDataValue retval;
    FileRef f(path.toLocal8Bit().constData());
    const Tag* t = f.tag();
    Q_ASSERT( t != NULL ) ;
    retval.setFileName(path);
    retval.setTrackTitle(toQString(t->title()));
    retval.setArtist(toQString(t->artist()));
    retval.setAlbumTitle(toQString(t->album()));
[ . . . . ]

    QTime time(0,0,0,0);
    const AudioProperties* ap = f.audioProperties();
    time = time.addSecs(ap->length());
    retval.setTrackTime(time);
    return retval;
}
```

注意

在利用与 MetaDataValue 参数的排队连接发出信号之前，必须先调用 qRegisterMetaType<MetaDataValue>("MetaDataValue")。这是因为，在底层 QMetaType::construct()会动态创建另一个实例。

16.4 复习题

1. 创建模式如果能够管理对象的销毁?
2. 如何利用属性来编写更通用的写入器(Writer)?
3. 如何利用抽象工厂来编写更通用的读取器(Reader)?
4. 哪些 Qt 类是 Façade 模式的实现? 解释为什么它们是 Façade 模式或者封装器。

16.4.1 更多探讨

- 给出另一个 UserType,它会针对 InputField 的新类型而被添加到 QVariant 中。
- 关于使用元对象的 moc 对象以及 marshalling 对象的更多探讨，请参见 Qt 季刊(Qt Quarterly)[①]。

① 参见 http://doc.trolltech.com/qq/qq14-metatypes.html。

第17章 并 发

QProcess 和 QThread 提供了两种并发方式。本章将讨论进程和线程的创建及其通信方式，同时还会介绍进程与线程的监视和调试技术。

17.1 QProcess 和进程控制

QProcess 是一个能够非常方便(而且跨平台)的用于启动和控制其他进程的类。它从 QObject 派生而来，可充分利用信号和槽来简化与其他 Qt 类的“交互”。

现在考虑一个简单的例子，启动一个进程并观察其持续运行的输出结果[①]。示例 17.1 中给出了一个 QProcess 的简单派生类的定义。

示例 17.1 src/logtail/logtail.h

```
[ . . . . ]
#include <QObject>
#include <QProcess>
class LogTail : public QProcess {
    Q_OBJECT
  public:
    LogTail(QString fn = QString());
    ~LogTail();
  signals:
    void logString(const QString &str);

  public slots:
    void logOutput();
};
[ . . . . ]
```

一个 QProcess 可以使用 start() 函数来启动[②]另外一个进程。新进程将会成为一个子进程并且在父进程终止时而随之终止[③]。示例 17.2 给出了 LogTail 类构造函数和析构函数的实现。

示例 17.2 src/logtail/logtail.cpp

```
[ . . . . ]

LogTail::LogTail(QString fn) {
```

---

① tail -f 将会始终运行，随时显示所有添加到文件中的内容，这对于显示一个运行进程的日志文件的内容是非常有用的。

② 强调 QProcess API 的跨平台价值是因为这样一个事实，一个进程用于启动另一个进程的机制在两大主流操作系统家族中是非常不同的。有关这一机制在*nix 系统中的做法的更多信息，可参阅维基百科中的文章 http://en.wikipedia.org/wiki/Fork_%28operating_system%29。而描述 Microsoft Windows 中的方法位于维基百科中的文章 http://en.wikipedia.org/wiki/Spawn_%28computing%29。

③ 也可以使用 startDetached() 函数来启动一个在父进程结束之后仍能存活的进程。

```
    connect (this, SIGNAL(readyReadStandardOutput()),
        this, SLOT(logOutput()));                                    1
    QStringList argv;

    argv << "-f" << fn;                                              2
    start("tail", argv);                                             3
}
LogTail::~LogTail() {
    terminate();                                                     4
}
```

1 当输入准备好时，会调用这个槽。

2 `tail -f` 文件名。

3 立即返回，并且现在会有一个“依附”于当前进程的子进程在独立运行。当调用进程退出时，新产生的子进程也会终止。

4 试图终止此进程。

子进程可以看成是一个预先定义了两个输出通道的顺序 I/O 设备，这两个输出通道分别代表了两个独立的数据流：stdout 和 stderr。父进程可以使用函数 setReadChannel() 来选择一个输出通道(默认是 stdout)。当子进程中被选中通道的数据可用时，它将会发射出信号 readyRead()。此时父进程就可以通过调用函数 read()，readLine() 或者 getChar() 来读取其输出结果。如果子进程启用了标准输入，那么父进程就可以使用 write() 函数向其发送数据。

示例 17.3 给出了 logOutput() 槽的具体实现，这个槽与 readyReadStandardOutput() 信号相连接，并且使用了 readAllStandardOutut() 方法，因此它仅仅关注 stdout。

示例 17.3 src/logtail/logtail.cpp

```
[ . . . . ]

// tail sends its output to stdout.
void LogTail::logOutput() {                                          1
    QByteArray bytes = readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        emit logString(line);
    }
}
```

1 无论什么时候有要读取的输入，都会调用该槽。

各个信号的消除用法是需要有一个读循环。当没有更多的输入数据要读入时，就不会再调用槽。信号和槽使得并行代码的可读性更好，原因是它们隐藏了事件处理和派发的代码。示例 17.4 给出了一些相关的客户代码。

示例 17.4 src/logtail/logtail.cpp

```
[ . . . . ]

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

```
    QStringList al = app.arguments();
    QTextEdit textEdit;
    textEdit.setWindowTitle("Debug");
    textEdit.setWindowTitle("logtail demo");
    QString filename;
    if (al.size() > 1) filename = al[1];
    LogTail tail(filename);                                          1
    tail.connect (&tail, SIGNAL(logString(const QString&)),
        &textEdit, SLOT(append(const QString&)));
    textEdit.show();
    return app.exec();
}
```

1 创建对象，同时启动进程。

一旦数据行在指定的日志文件中出现，这个应用程序就会在 QTextEdit 中进行追加。为了演示 LogTail 这一应用程序，需要一个诸如某种活动日志的文本文件，它可以随着向其添加行而不断增长。如果无法找到一个这样的文件，可以使用诸如 top 这样的工具自行创建一个，top 是一个在典型 *nix 主机上都可用的一个实用工具。通常情况下，不带命令行参数的 top 会产生一个纯文本、格式化的屏幕列表，其中会按照资源的使用情况来降序排列当时使用系统资源最多的 25 个正在运行的进程。显示的开始位置是系统使用情况的总体说明，并且每隔几秒都会更新整个显示。top 会一直运行，直到它被用户终止。在这个例子中，我们可以用以下命令行参数来启动 top：

- -b，将其置为批处理模式，以便可以重定向其输出。
- -d 1.0，指明两次更新之间的间隔秒数。
- > toplog，将输出重定向到 toplog 文件中。
- &，将 top 作为后台进程运行。

然后，对所得的结果文件运行 logtail 例子：

```
top -b -d 1.0 > toplog &
./logtail toplog
```

图 17.1 是该程序运行过程中的屏幕截图。

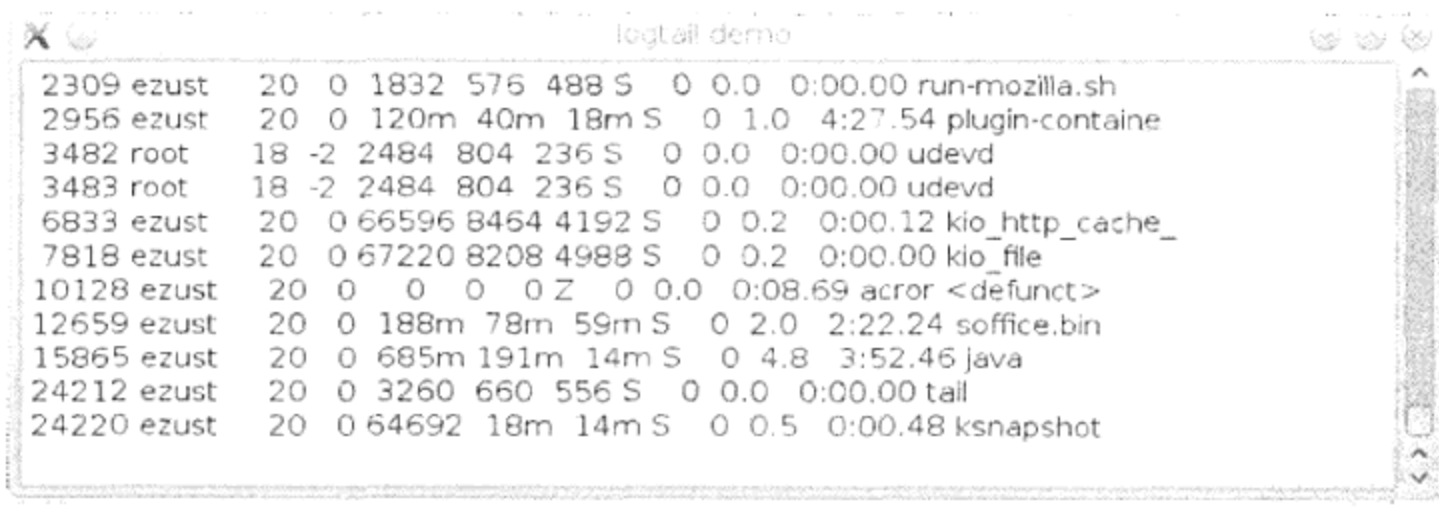


图 17.1　使用中的 LogTail

这个示例程序会一直运行直至被迫终止，随后必须杀死 top 作业，它的作业编号(job number)和进程 ID 在启动后就会显示出来。使用 bash 时，杀死作业时只需使用作业编号%1 即可。

```
src/logtail> top -b -d 1.0 > toplog &
[1] 24209
src/logtail> ./logtail toplog
      [[ logtail was terminated here. ]]
QProcess: Destroyed while process is still running.
src/logtail> kill 24209
src/logtail>
```


17.1.1 进程和环境

环境变量是<name, value>这样的名/值字符串对，可以非常容易地存储在一个映射或者哈希表中。环境变量名必须是合法的标志符，按照惯例，通常不会含有小写字母。每个正在运行的进程都有一个由环境变量的集合所构成的环境。大多数编程语言都会支持一种获取和设置这些变量的方法。

最常用的一些环境变量如下。

1. PATH。用于搜索可执行文件(在 Windows 上也包括动态链接库)的一个目录列表。
2. HOME。主目录的位置。
3. CPPLIBS。来自本书源代码示例程序的 C++库的安装位置[①]。
4. HOSTNAME(*nix)或者 COMPUTERNAME (win32)。通常用于获取机器名。
5. USER(*nix)或者 USERNAME (Win32)。通常用于获取当前登录的用户 ID。

环境变量和它们的值通常由父进程设置。依赖特殊变量或者值的程序通常是不可移植的，但在一定程度上也取决于它们的父进程。环境变量为进程间交流信息提供了一种便利的跨语言机制。

操作系统允许用户给进程及其将来的子进程设置环境变量。下面是一些例子。

- Microsoft Windows 桌面——通过 Start->Settings->System->Advanced->Environment Variables 可以得到类似图 17.2 的设置对话框。
- Microsoft 命令提示符——可以设置 VARIABLE = value 和 echo %VARIABLE%。在较新版本的 Windows 中也可以看到 setx 命令。
- Bash 命令行——利用 export VARIABLE = value 和 echo $VARIABLE。

许多编程语言也支持环境变量的获取和设置，列举如下。

- C++/Qt：QProcess::environment()函数和 setEnvironment()函数。
- Python：os.getenv()函数和 os.putenv()函数。
- Java：ProcessBuilder.environment()函数。
- C：<cstdlib>中的 getenv()函数和 putenv()函数(参见附录 B)。

所运行的任何程序涉及的进程树都可能有好几层深。这是因为，典型的桌面操作系统环境是由许多并发运行的进程一起组成的。图 17.3 中，缩进排列的层次显示了进程之间的父-子关系。在同一个缩进层次上的进程之间是兄弟关系(即它们拥有相同的父对象)。图 17.3 给出了在典型的 KDE/Linux 系统上该 environment 例子从 konsole 运行时各个进程的生命周期和进程之间的父-子关系。

---

① 参见 http://www.distancecompsci.com/dist/。

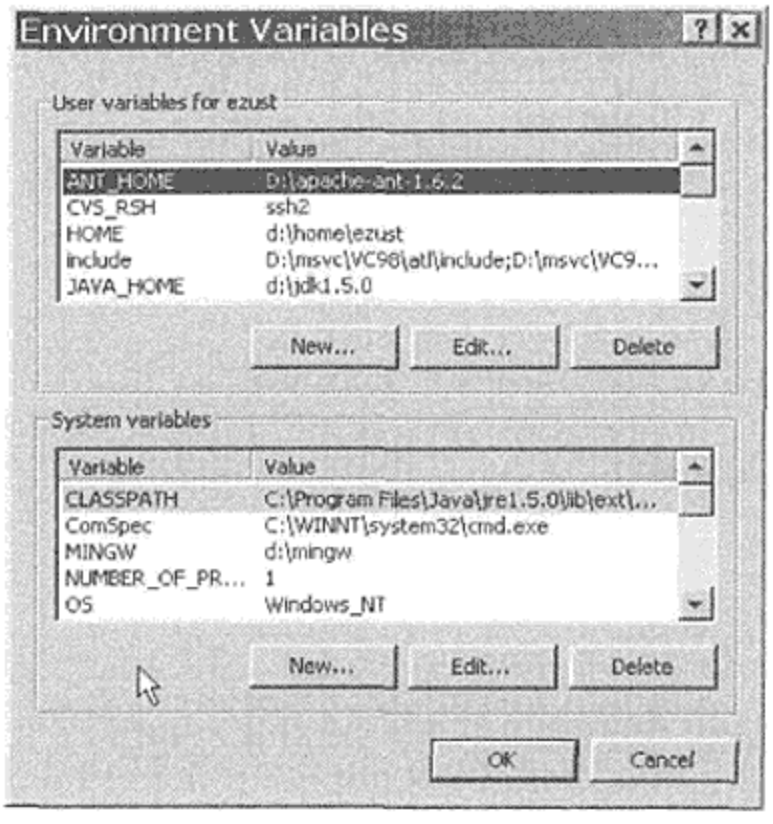


图 17.2　Windows 环境变量

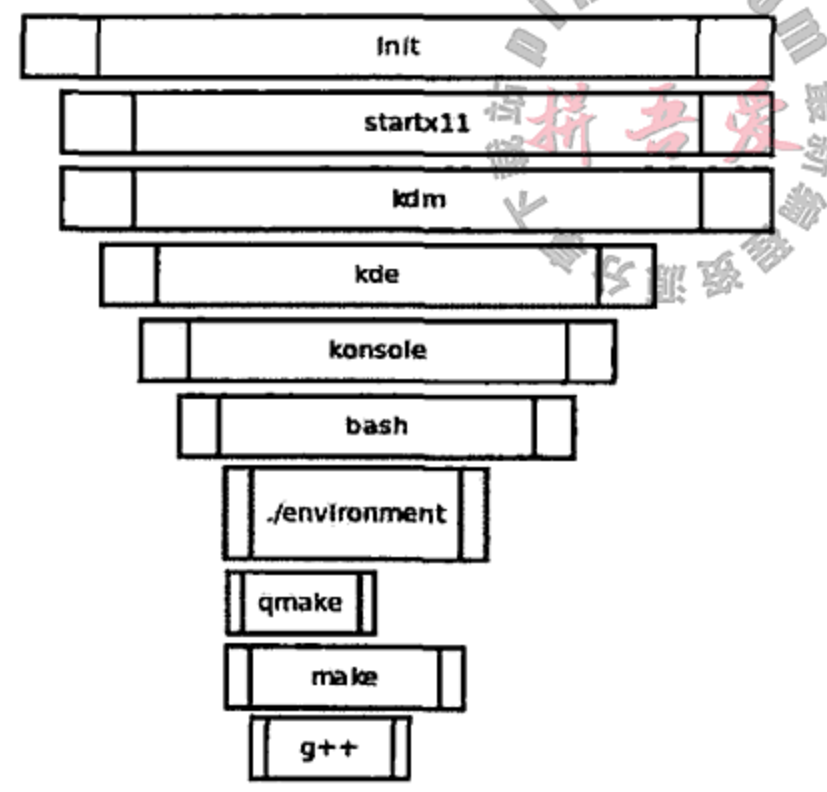


图 17.3　KDE/Linux 的进程层次

只要程序 A 启动了程序 B，进程 A 就是进程 B 的父亲。在创建进程 B 时，进程 B 会继承(作为一个副本)进程 A 的环境。在进程 B 内部对环境的改变将会影响进程 B 及其子孙进程，而这些修改对于进程 A 来说是永远不可见的。

示例 17.5 展示了提供给 `setenv()` 函数的值可传播到其子孙进程中。

示例 17.5　src/environment/setenv.cpp

```
#include <QCoreApplication>
#include <QTextStream>
#include <QProcess>
#include <QCoreApplication>
#include <QTextStream>
#include <QStringList>
#include <cstdlib>

class Fork : public QProcess {
    public:
    Fork(QStringList argv = QStringList() ) {
        execute("environment", argv);
    }
    ~Fork() {
        waitForFinished();
    }
};

QTextStream cout(stdout);
int main(int argc, char* argv[]) {

    QCoreApplication qca(argc, argv);
    QStringList al = qca.arguments();
    al.removeAt(0);
    bool fork=al.contains("-f");
    if(fork) {
       int i = al.indexOf("-f");
```

```
        al.removeAt(i);
    }

    QStringList extraVars;
    if (al.count()  > 0) {
        setenv("PENGUIN", al.first().toAscii(), true);
    }
    cout << " HOME=" << getenv("HOME") << endl;
    cout << " PWD=" << getenv("PWD") << endl;
    cout << " PENGUIN=" << getenv("PENGUIN") << endl;

    if (fork) {
        Fork f;
    }
}
```


1 将同一应用程序作为子进程运行。

当这个程序运行时，其输出结果如下所示。

```
src/environment> export PENGUIN=tux
src/environment> ./environment -f
HOME=/home/lazarus
PWD=src/environment
PENGUIN=tux
HOME=/home/lazarus
PWD=src/environment
PENGUIN=tux
src/environment> ./environment -f opus
HOME=/home/lazarus
PWD=src/environment
PENGUIN=opus
HOME=/home/lazarus
PWD=src/environment
PENGUIN=opus
src/environment> echo $PENGUIN
tux
src/environment>
```

17.1.2 Qonsole：用 Qt 编写一个 Xterm

命令行 shell 从用户读入命令，然后打印程序的输出结果。这个例子中用一个 QTextEdit 来给另外一个正在运行的进程提供输出结果视图。在这里这个进程是指 bash，它是大多数 *nix 系统中默认的命令行 shell 解释程序(Windows 上是 cmd)。Qprocess 是一个模型，代表一个正在运行的进程。图 17.4 所示为 Qonsole 运行时的一个截图，这是我们在图形用户界面(GUI)中提供命令 shell 第一次尝试[①]。

因为 Qonsole 把信号连接到了槽上并以此来处理用户交互问题，所以可以把它看成是一个控制器。而又因为它继承自 QMainWindow 类，所以其中也包含了一些视图代码。图 17.5 中的 UML 框图给出了此应用程序中的各个类之间的关系。

① 本书作者在 Qonsole 发音上尚存在一些分歧。两个选择或许应当是“Chon-sole”(使用标准的中文发音惯例，就好像 Qing)和“Khonsole”(使用标准的阿拉伯发音方法，就好像 Qatar)。

图 17.4　Qonsole1

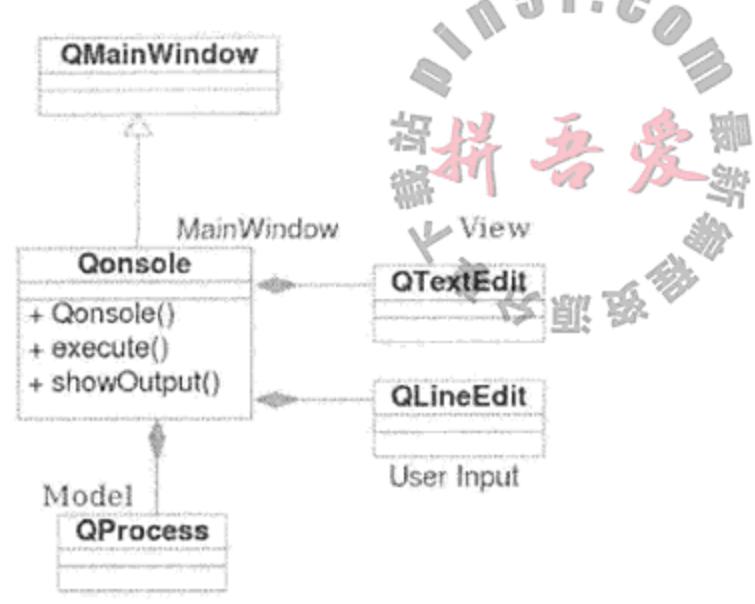


图 17.5　Qonsole UML：模型和视图

示例 17.6 给出了 Qonsole 类的定义。

示例 17.6　src/qonsole/qonsole1/qonsole.h

```
[ . . . . ]
class Qonsole : public QMainWindow {
    Q_OBJECT
 public:
    Qonsole();
 public slots:
    void execute();
    void showOutput();
 private:
    QTextEdit* m_Logw;
    QLineEdit* m_InputArea;
    QProcess* m_Shell;
};
[ . . . . ]
```

示例 17.7 中，我们可以看到构造函数是如何创建 Qonsole 的框架的，同时还可以看到 Qonsole 组件之间的一些重要连接。

示例 17.7　src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]
Qonsole::Qonsole() {
    m_Logw = new QTextEdit();
    m_Logw->setReadOnly(true);
    setCentralWidget(m_Logw);
    m_InputArea = new  QLineEdit();
    QDockWidget* qdw = new QDockWidget("Type commands here");
    qdw->setWidget(m_InputArea);
    addDockWidget(Qt::BottomDockWidgetArea, qdw);
    connect (m_InputArea, SIGNAL(returnPressed()),
                          this, SLOT(execute()));

    m_Shell = new QProcess(this);
    m_Shell->setReadChannelMode(QProcess::MergedChannels);
    connect (m_Shell, SIGNAL(readyReadStandardOutput()),
```

```
                                this, SLOT(showOutput()));
#ifdef Q_OS_WIN
    m_Shell->start("cmd", QStringList(), QIODevice::ReadWrite);
#else
    m_Shell->start("bash", QStringList("-i"), QIODevice::ReadWrite);  2
#endif

}
```

1 将 stdout 和 stderr 合二为一。

2 以交互方式运行 bash。

不管 shell 输出什么结果，Qonsole 都会将其发送到 QTextEdit。而无论用户何时按下回车键，Qonsole 都会捕捉 QLineEdit 中的所有文本并将其发送给 Shell，由 shell 将这些文本作为一个命令进行解释，正如示例 17.8 所示。

示例 17.8 src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]

void Qonsole::showOutput() {                                          1
    QByteArray bytes = m_Shell->readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        m_Logw->append(line);
    }
}

void Qonsole::execute() {
    QString cmdStr = m_InputArea->text() + "\n";
    m_InputArea->setText("");
    m_Logw->append(cmdStr);
    QByteArray bytes = cmdStr.toUtf8();                               2
    m_Shell->write(bytes);                                            3
}
```

1 只要输入准备好，就会调用该槽。

2 8 位的 Unicode 传输模式。

3 将数据发送到 Shell 子进程的 stdin 流。

示例 17.9 给出了启动这个应用程序的客户代码。

示例 17.9 src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]

#include <QApplication>

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

17.1.3　带有键盘事件的 Qonsole

在先前的例子中，Qonsole 为用户输入提供一个单独的窗件。为了提供更为真实的终端经验，用户应该能够在命令输出窗口中输入命令。为了满足这一基本需求，Qonsole 需要捕捉键盘事件。其中的第一步，就是重写 QObject 基类的 eventFilter()方法。示例 17.10 中修改的类定义中可以看到这一点。

示例 17.10　src/qonsole/keyevents/qonsole.h

```
[ . . . . ]
class Qonsole : public QMainWindow {
    Q_OBJECT
 public:
    Qonsole();
 public slots:
    void execute();
    void showOutput();
    bool eventFilter(QObject *o, QEvent *e)  ;
 protected:
    void updateCursor();
 private:
    QString m_UserInput;
    QTextEdit* m_Logw;
    QProcess* m_Shell;
};
[ . . . . ]
```

正如 8.3 节中讨论的那样，事件是一个从 QEvent 派生的对象。在一个应用程序的上下文中，这样一个 QEvent 与作为预期接收者的 QObject 相关联。负责接收的对象会有一个专门处理此事件的函数。事件过滤器首先对 QEvent 进行检查，然后决定是否允许其接收者进行处理。我们给修改后的 Qonsole 应用程序提供了一个 eventFilter()函数，它的作用是过滤来自 m_Logw 的键盘事件，其中 m_Logw 扩展自 QTextEdit。Qonsole 扩展自 QmainWindow，它是所有这些事件的预期接收者。此函数的具体实现会在示例 17.11 中给出。

示例 17.11　src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

bool Qonsole::eventFilter(QObject* o, QEvent* e) {
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent* k = static_cast<QKeyEvent*> (e);
        int key = k->key();
        QString str = k->text();
        m_UserInput.append(str);
        updateCursor();
        if ((key == Qt::Key_Return) || (key == Qt::Key_Enter) ) {
#ifdef Q_WS_WIN                                                    1
            m_UserInput.append(QChar(0x000A));
#endif
            execute();
            return true;                                           2
```

```
        }
        else {
            m_Logw->insertPlainText(str);
            return true;
        }
    }
    return false;                                                    3
}
```

1 Windows 进程需要的是一个回车+换行，并非只是一个回车。

2 我们处理了这个事件。这样，其他的窗件也就无法再感知此事件。

3 不要处理(touch)该事件。

当按下回车键时，就会调用成员函数 execute()，以便可以让命令字符串发送到 Shell 并随之得到重置。示例 17.12 给出了这两个函数的具体实现。

示例 17.12 src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

void Qonsole::updateCursor() {
    QTextCursor cur = m_Logw->textCursor();
    cur.movePosition(QTextCursor::End, QTextCursor::KeepAnchor);
    m_Logw->setTextCursor(cur);
}

void Qonsole::execute() {
    QByteArray bytes = m_UserInput.toUtf8();
    m_Shell->write(bytes);
    m_UserInput = "";
}
```

剩下的事情就是在 m_Logw 上调用其基类函数 installEventFilter()，该窗件的事件也正是我们想要捕捉的事件。这部分动作是在构造函数中完成的，如示例 17.13 所示。

示例 17.13 src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

Qonsole::Qonsole() {
    m_Logw = new QTextEdit;
    setCentralWidget(m_Logw);
    m_Logw->installEventFilter(this);                                1
    m_Logw->setLineWrapMode(QTextEdit::WidgetWidth);
    m_Shell = new QProcess();
    m_Shell->setReadChannelMode(QProcess::MergedChannels);
    connect (m_Shell, SIGNAL(readyReadStandardOutput()),
             this, SLOT(showOutput()));
#ifdef Q_WS_WIN
    m_Shell->start("cmd", QStringList(), QIODevice::ReadWrite);
#else
    m_Shell->start("bash", QStringList("-i"), QIODevice::ReadWrite);
#endif
}
```

1 截取送往 QTextEdit 的事件。

17.1.4　练习：QProcess 和进程控制

1. 在 Qonsole 的这一版本中，尚无法恰当地处理退格键。增加一个事件处理器，让它可以对退格键进行适当的响应。
2. 修改 Qonsole，使其能够在单独的标签中支持多个并行终端。
3. 根据美国国家标准与技术研究所[①](NIST)的研究成果，哈希函数可接受称为消息的二进制数据，并可生成一种称为消息摘要(message digest)的简明表示。加密哈希函数是一个旨在达到一定的安全属性的哈希函数。联邦信息处理标准(Federal Information Processing Standard)180-2，即安全哈希标准(Secure Hash Standard)，给出了 5 个用在计算方面的加密哈希函数算法：SHA-1, SHA-224, SHA-256, SHA-384 和 SHA-512。

   对于任意给定的数据块，一个好的加密哈希函数必须能够可靠地生成实质唯一的摘要。也就是说，绝不可能有其他任何数据块可以使用该函数得到同样的摘要。哈希法是一种单向操作。也就是说，这一过程通常是不可逆的，也不可能从该摘要中生成数据块。

   通过仅对每个密码的摘要进行存储即可处理安全密码。当用户登录并输入密码，那个字符串会即刻得到散列处理，其结果摘要会和存储的摘要加以比对。如果两者匹配，用户就是合法的。否则，登录就不成功。用户的密码永远不会被存储并且也只是在计算密码摘要时存在于内存中。

   Qt 有一个 QCryptographicHash 类，它提供了一个用于计算给定 QByteArray 加密哈希值的哈希函数。Qt 4.7 中提供了 SHA-1, MD4 和 MD5[②]。

   a. 编写一个带有两个命令行参数的简单应用程序：一个是用于哈希处理的字符串，一个用来说明所要使用的算法。该应用程序应当把摘要的处理结果发送到标准输出终端上。例如：

      ```
      crhash "my big secret" md5
      ```

      将输出由二进制数据所组成的摘要。

   b. 把上一题中的 crhash 应用程序用作单独进程使用，编写一个管理俱乐部成员数据的应用程序，包括用户的 ID、密码、电子邮件地址、街道地址、城市、州、邮政编码和电话号码。密码应当只存储成摘要。确保对组成成员的数据进行过适当的序列化处理。

17.2　QThread 和 QtConcurrent

在 Qt 之前，对于 C++开源程序开发人员来说，通常是没有跨平台多线程可用的，因为多线程是相对较新的概念，并且在每个操作系统中多线程的处理方式都有所不同。现在，在大多数的操作系统和许多现代编程语言中都提供多线程。多核处理器也已相当普遍，因此，来自同一进程中的多个线程都可被现代操作系统分配到不同的内核上运行。

---

① 参见http://csrc.nist.gov/groups/ST/hash/index.html。

② MD4 和 MD5 是由 Ron Rivest 在 SHA-1 之前设计用于消息—摘要的算法，而 SHA-1 已经被 SHA-2 的系列哈希函数所取代。更多详细情况，可以参阅http://en.wikipedia.org/wiki/MD5。

Qt 的线程模型允许线程的优先次序和控制。QThread 是一个低级(low-level)类，适合用于显式地构建长期运行的线程。

QtConcurrent 是一个命名空间，提供了用于编写并发软件的更高层次的类和算法。该命名空间中有一个重要的类，QThreadPool，这是一个管理线程池的类。每个 Qt 应用程序都有一个 QThreadPool::globalInstance()函数，它带有一个推荐的最大线程数，在大多数系统上，处理核的数量就是该值的默认值。

借助于 QtConcurrent 中函数式的 map/filter/reduce 算法(它们可将函数并行用到容器中的每个项)，通过将进程分布在由线程池管理的多个线程上，可编写一个能够自动利用系统多核的程序。另外，在命令模式和利用 QtConcurrent:: run()工作时可把 QRunnable 用作基类。在这些情况下，无须显式地创建线程或者直接管理它们，只需简单地把工作片段描述为具有正确接口的对象即可。

为什么使用线程

有时候，使用线程给软件所带来的复杂程度会超过所带来的性能优势。如果一个程序的性能受限于输入/输出，则将 CPU 的工作分散于多个线程将不会为程序的整体性能带来明显改善。然而，如果程序需要做相当数量的高级计算工作，并且也有空闲的处理核，多线程就可以提高性能。

线程指南

一般情况下，要尽可能避免使用线程，而是用 Qt 事件循环与 QTimer、非阻塞 I/O 操作、信号以及短持续时间槽相结合的方法来代替。此外，可以在主线程中长期运行的循环调用 QApplication::processEvents()，以使执行工作时图形用户界面可以保持响应[①]。

要驱动动画(animation)，建议使用 QTimer, QTimeLine 或者动画框架(Animation Framework)[②]。这些 API 并不需要额外创建其他线程。它们允许访问动画代码中的 GUI 对象而且不会妨碍图形用户界面的响应。

如果要完成 CPU 密集型工作并希望将其分配给多个处理核，可以把工作分散到 QRunnable 并通过以下这些推荐做法来实现线程的安全。

- 无论何时，都尽可能使用 QtConcurrent 算法把 CPU 密集型计算工作分散给多线程，而不是自己编写 QThread 代码。
- 除了主线程以外，不要从其他任何线程访问图形用户界面(这也包括那些由 QWidget 派生的类、QPixmap 和其他与显卡相关的类)。这包括读取操作，比如查询 QLineEdit 中输入的文本。
- 要其他线程中处理图像，使用 QImage 而不是 QPixmap。
- 不要调用 QDialog::exec()或者从除主线程之外的任何线程创建 QWidget 或 QIODevice 的子类。

① 参见http://doc.trolltech.com/qq/qq27-responsive-guis.html#manualeventprocessing。

② 在 Qt 4.6 中引入。

- 使用 QMutex，QReadWriteLock 或者 QSemaphore 以禁止多个线程同时访问临界变量。
- 在一个拥有多个 return 语句的函数中使用 QMutexLocker(或者 QReadLocker，QWriteLocker)，以确保函数从任意可能的执行路径均可释放锁。
- 创建 QObject 的线程，也称线程关联(thread affinity)，负责执行那个 QObject 的槽。
- 如果各 QObject 具有不同的线程关联，那么就不能以父-子关系来连接它们。
- 通过从 run()函数直接或者间接地调用 QThread::exec()，可以让线程进入事件循环。
- 利用 QApplication:: postEvent()分发事件，或使用队列式的信号/槽连接，都是用于线程间通信的安全机制——但需要接收线程处于事件循环中。
- 确保每个跨线程连接的参数类型都用 qRegisterMetaType()注册过。

17.2.1　线程安全和 QObject

可重入(reentrant)函数就是一个可以由多个线程同时调用的函数，其中任意的两次调用都不会试图访问相同的数据。线程安全的方法在任何时间都可以同时由多个线程调用，因为任何共享数据都会在某种程度上(例如，通过 QMutex)避免被同时访问。如果一个类的所有非静态函数都是可重入的或者是线程安全的，那么它就是可重入的或者是线程安全的。

一个 QObject 在它所“属于”或者有关联的线程中被创建。其各子对象也必须属于同一线程。Qt 禁止存在跨线程的父—子关系。

- QObject::thread()可返回它的所有者线程，或者是其关联线程。
- QObject::moveToThread()可将其移动到另一个线程。

moveToThread(this)

由于 QThread 是一个 QObject 而且在需要额外的线程时才会创建 QThread，因此，即使你会认为 QThread 和线程是可以相互指代的，也是可以理解的。尽管如此，那个额外的线程在调用 QThread::start()之前实际上都不会被创建，这使得问题更难于理解。

回想一下，每个 QThread 的本质都是一个 QObject，这决定了它与其创建的线程存在关联，而不是与它启动的线程存在关联。

正是因为这个原因，有人说 QThread 并不是线程本身，而是该线程的管理器。这或许也可以有助于理解这一方式。实际上，QThread 是一个底层线程 API 的封装器，也是一个基于 java.lang.thread API 的管理单个线程的管理器。

这就意味着，当信号连接到这个 QThread 的槽上时，槽函数的执行是在其创建线程，而不是在其管理的线程进行的。

一些程序通过改变 QThread 的定义使它可表示其管理的线程并在该线程内执行它的槽。这些程序使用一种变通方法：在 QThread 的构造函数中使用 moveToThread(this)。这一变通方法的主要问题是，在线程退出后，通过 post 方式派发给该对象的事件如何处理留下不确定性。

有一篇文章[①]详细地讨论了这个问题。要说明的是，尽管 Qt 自身文档和一些旧示例程

① 参见http://labs.qt.nokia.com/2010/06/17/youre-doing-it-wrong/。

序都使用了这种做法，它仍是不再被推荐的。一方面，在托管的线程终止时，对于事件和信号会发生什么并不确定。另一方面，它混淆了 QRunnable 和 QThread 的目的，并将太多的职责加到单一的类中。

线程安全的对象就是一个可以由多个线程同时访问并且可确保处于“有效”状态的对象。默认情况下，QObject 不是线程安全的。为了让一个对象线程安全，可以利用的方法有许多种。这里列出了一些，但推荐去更深入地了解 Qt 线程支持方面的文档[①]。

1. QMutex 用于保证互斥，可与 QMutexLocker 一起使用，它允许一个单独的线程 T 保护(锁住)一个对象或者一段代码，使其在线程 T 释放(解锁)之前不能被其他的线程访问。
2. QWaitCondition 与 QMutex 结合使用，可以把某个线程置于一种不忙的阻塞状态，这种状态下，可让其等待另外一个线程将其唤醒。
3. QSemaphore 是一个广义的 QMutex，可以用在一个线程在开始工作之前需要锁住不止一个资源的各种情况下。信号量使其能够保证线程仅在要进行工作所需的资源全部满足的情况下才锁住资源。

有更多的 Qt 示例展示了如何使用 QtConcurrent： $QTDIR/examples/qtconcurrent。

volatile 的合理性

volatile 就像 const 一样，可由编译器用于为程序强制确保一定的线程安全。如果对象是 volatile 的，则只有标记为 volatile 的方法才可以调用它。volatile 在 Qt 中用于实现一些原子操作，这些操作反过来又用于实现诸如 QMutex，QSharedPointer 和 QReadWriteLock 这样的更高级结构中。有关 volatile 的更多信息，可参考下列两篇文章。

- *Using Volatile with User Defi ned Types*(在用户定义的类型中使用 volatile)[②]。
- *Volatile Almost Useless for Multithreaded Programming*(volatile 对多线程编程几乎没有什么用)[③]。

17.2.2 并行素数计算器

这一节介绍两种不同的计算素数的方法，其工作共享于多个线程。

第一种方法是生产者—消费者模型(producer-consumer model)，带有一个负责收集结果的调停对象(mediator object)。

示例 17.14 给出了一个生产者类 PrimeServer。

示例 17.14 src/threads/PrimeThreads/primeserver.h

```
[ . . . . ]
class PrimeServer : public QObject
{
    Q_OBJECT
public:
```

① 参见http://doc.qt.nokia.com/latest/threads.html。

② 参见http://www.drdobbs.com/184403766。

③ 参见http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/。

```
    explicit PrimeServer(QObject* parent =0);
    void doCalc(int numThreads, int highestPrime, bool concurrent = false);
    int nextNumberToCheck();
    void foundPrime(int );
    bool isRunning() const;
public slots:
    void cancel();
private slots:
    void handleThreadFinished();
signals:
    void results(QString);
private:
    int m_numThreads;
    bool m_isRunning;
    QList<int> m_primes;
    int m_nextNumber;
    int m_highestPrime;
    QTime m_timer;
    QMutex m_nextMutex;
    QMutex m_listMutex;
    QSet<QObject*> m_threads;
private slots:
    void handleWatcherFinished();
    void doConcurrent();
private:
    bool m_concurrent;
    int m_generateTime;                                          1
    QFutureWatcher<void> m_watcher;
};
[ . . . . ]
```

1 花在生成输入数据上的时间。

PrimeServer 创建 PrimeThreads(消费者)来执行实际的工作。示例 17.15 中创建并启动了各个 PrimeThreads 子对象。

示例 17.15 src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]
void PrimeServer::
doCalc(int numThreads, int highestPrime, bool concurrent) {
    m_isRunning = true;
    m_numThreads = numThreads;
    m_concurrent = concurrent;
    m_highestPrime = highestPrime;
    m_primes.clear();
    m_primes << 2 << 3;
    m_threads.clear();
    m_nextNumber = 3;
    m_timer.start();
    if (!concurrent) {
        for (int i=0; i<m_numThreads; ++i) {
            PrimeThread *pt = new PrimeThread(this);             1
            connect (pt, SIGNAL(finished()), this,
                     SLOT(handleThreadFinished()));
```

```
            m_threads << pt;
            pt->start();                                          2
        }
    }
    else doConcurrent();
}
```


1 子线程还没有开始。

2 子线程执行 run()。

如示例 17.16 所示，PrimeThread 是一个重写了 run() 的自定义 QThread。

示例 17.16 src/threads/PrimeThreads/primethread.h

```
#ifndef PRIMETHREAD_H
#define PRIMETHREAD_H
#include <QThread>
#include "primeserver.h"

class PrimeThread : public QThread
{
    Q_OBJECT
public:
    explicit PrimeThread(PrimeServer *parent);
    void run();                                                   1
private:
    PrimeServer *m_server;

};

#endif // PRIMETHREAD_H
```

1 需要重写。

如示例 17.17 所示，在一个紧凑的循环中，run() 函数在调用 PrimeServer 的两个带有 QMutexLocker 的方法中间进行一个素数测试。

示例 17.17 src/threads/PrimeThreads/primethread.cpp

```
[ . . . . ]
PrimeThread::PrimeThread(PrimeServer *parent)
: QThread(parent), m_server(parent) { }

void PrimeThread::run() {
    int numToCheck = m_server->nextNumberToCheck();
    while (numToCheck != -1) {
        if (isPrime(numToCheck))
            m_server->foundPrime(numToCheck);
        numToCheck = m_server->nextNumberToCheck();
    }
}
[ . . . . ]
```

如示例 17.18 所示，PrimeServer 使用 QMutexLocker 来锁住 QMutex，这使得程序进入和离开封闭的块作用范围时都可以安全地锁定和解锁 QMutex。最初，这个程序使用一个简

单的互斥量来保护这两种方法，但因为要访问的数据是独立的，为每种方法都使用独立的互斥量可以增加并行的可能性。

示例 17.18　src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

int PrimeServer::nextNumberToCheck() {
    QMutexLocker locker(&m_nextMutex);                    1
    if (m_nextNumber >= m_highestPrime) {
        return -1;
    }
    else {
        m_nextNumber+= 2;
        return m_nextNumber;
    }
}

void PrimeServer::foundPrime(int pn) {
    QMutexLocker locker(&m_listMutex);                    2
    m_primes << pn;
}
```

1 基于作用范围的互斥量工作于有多个返回点的情况。

2 这个方法也必须要线程安全。

这些方法都是线程安全的，因为从多个线程同时调用会相互阻塞。这是序列化访问临界共享数据的一种方式。

示例 17.19　src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::cancel() {
    QMutexLocker locker(&m_nextMutex);
    m_nextNumber = m_highestPrime+1;
```

示例 17.19 中给出的 cancel 方法打算以非阻塞的方式来得到调用，以便让一个 GUI 可以在 PrimeThread 安全退出其循环并从 run() 返回时持续对事件作出响应。

示例 17.20 中，服务器会清空每个完成的线程并在它们全部完成时报告其结果。它使用了 QObject::sender() 来获取信号发送者并用 deleteLater() 安全地将其删除。这是一种推荐用于终止和清理线程的安全方式。

示例 17.20　src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::handleThreadFinished() {
    QObject* pt = sender();                               1
    m_threads.remove(pt);
    pt->deleteLater();
    if (!m_threads.isEmpty()) return;                     2
    int numPrimes = m_primes.length();
    QString result = QString("%1 mutex'd threads %2 primes in %3"
```

```
                                    "miliseconds. ").arg(m_numThreads)
                                    .arg(numPrimes).arg( m_timer.elapsed());
    QString r2 = QString(" %1 kp/s")
                    .arg(numPrimes / m_timer.elapsed());
    qDebug() << result << r2;
    emit results(result + r2);
    m_isRunning = false;
}
```

1 QThread 是发送者。

2 其他仍在运行中。

图 17.6 中对测试 100 000 000 个号码的结果进行了总结。标记有 Mutex'd 的行给出了在 n 个工作者线程上运行生产者—消费者算法时的加速比因子(speedup factor)。正如所看到的，最佳加速比因子是在 3 时获得的，之后性能就降了下来。这可能是因为还有一个生产者线程相当繁忙而在图中没有计算进来。

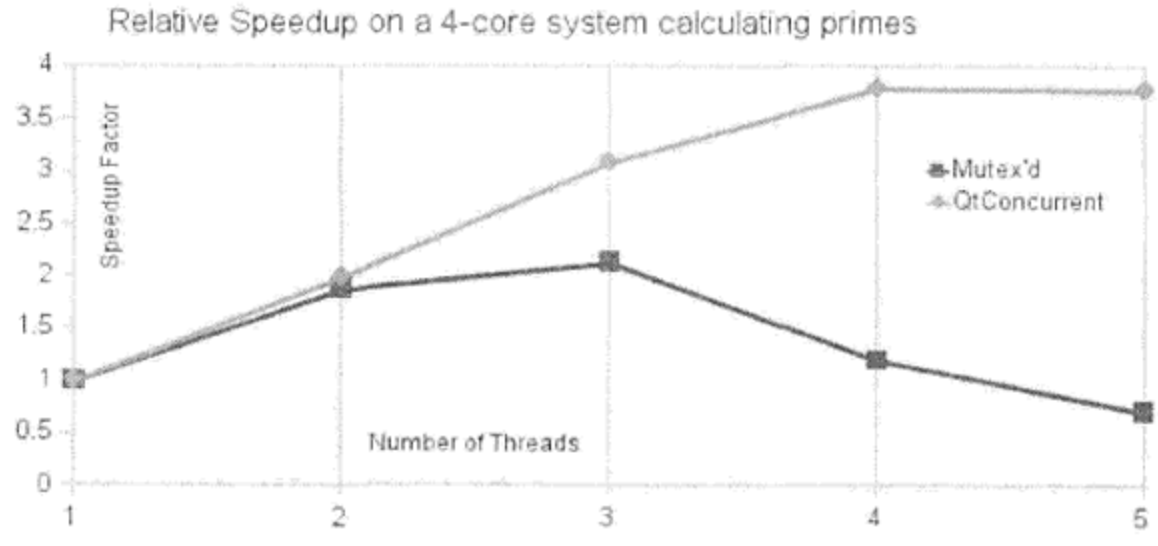


图 17.6 PrimeThreads 的加速比因子

标有 QtConcurrent 的另一行给出了几乎是最佳的加速比因子(1:1)，在 4 核系统上达到了 4，并在超过该值而继续增加线程数时并没有显著的退化。如示例 17.22 所示，这来自于对(前面)同一个 isPrime() 函数的并发使用。

这个曲线图主要关注在一个单独线程上运行相同算法时的相对加速因子的比较，但它的确给出的是绝对速度。如果尝试运行这个例子，会发现在所有测试中 QtConcurrent 方法总体至少会快 10 倍。

并行算法的效率取决于让所有核都忙着做计算，而不是等待对方的同步。相对越多的时间是花费在同步方面，随着线程增加得更多，算法的性能会变得越差。

QtConcurrent 方法

可以使用 QtConcurrent 的 filter()算法来从一系列的数中过滤非素数，而不是创建和管理自己的线程。QtConcurrent 算法可自动把工作分发给任意数量的线程，这可通过全局的 QThreadPool 给定。这些算法会接受一个容器和一个可作用于容器中每一项的函数指针或者仿函数。

示例17.21给出了在PrimeServer中并行算法用到的数据成员。可以使用QFutureWatcher以非阻塞的方式来等待计算的结束。

示例 17.21　src/threads/PrimeThreads/primeserver.h

```
[ . . . . ]

private slots:
    void handleWatcherFinished();
    void doConcurrent();
private:
    bool m_concurrent;
    int m_generateTime;                                          1
    QFutureWatcher<void> m_watcher;
};
```

1 用在生成输入数据上的时间。

示例 17.22 给出了一个函数式编程风格的解决方案。非阻塞的 `filtering()` 函数会立即返回一个 `QFuture` 类型的值。可以将其发送给 `QFutureWatcher` 来监测计算的进展情况。

示例 17.22　src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::doConcurrent() {
    QThreadPool::globalInstance()->setMaxThreadCount(m_numThreads);
    m_primes.clear();
    m_primes << 2;
    for (m_nextNumber=3; m_nextNumber<=m_highestPrime;
         m_nextNumber += 2) {
       m_primes << m_nextNumber;
    }
    m_generateTime = m_timer.elapsed();
    qDebug() << m_generateTime << "Generated "
             << m_primes.length() << " numbers";
    connect (&m_watcher, SIGNAL(finished()), this,
             SLOT(handleWatcherFinished()));
    m_watcher.setFuture(                                         1
        QtConcurrent::filter(m_primes, isPrime));                2
}

void PrimeServer::handleWatcherFinished() {
    int numPrimes = m_primes.length();
    int msecs = m_timer.elapsed();
    QString result =
      QString("%1 thread pool %2 primes in %4/%3 milliseconds"
              "(%5% in QtConcurrent).") .arg(m_numThreads)
              .arg(numPrimes).arg(msecs).arg(msecs-m_generateTime)
              .arg((100.0 * (msecs-m_generateTime)) / msecs);
    QString r2 = QString(" %1 kp/s").arg(numPrimes / msecs);
    qDebug() << result << r2;
    m_watcher.disconnect(this);
    emit results(result + r2);
    m_isRunning = false;
}
```

1 QFutureWatcher 用于检测进展情况。

2 非阻塞，此 filter()会返回一个 QFuture。

17.2.3 并发映射/规约示例

重新看一下 9.10 节中“康威的游戏人生”(Conway's Game of Life)示例，我们将使用 QtConcurrent 的 MapReduce()算法并行化这一计算。对于这项工作，必须先将问题分成较小的块。把每一块都定义成一个 LifeSlice，如示例 17.23 所示。

示例 17.23 src/threads/life/lifeslice.h

```
[ . . . . ]
struct LifeSlice {
    LifeSlice() {};
    LifeSlice(QRect r, QImage i) : rect(r), image(i) {}
    QRect rect;
    QImage image;
};
[ . . . . ]
```

一个 LifeSlice 由一个用来说明 QRect 是来自哪一块的 QImage 构成。这是映射函数的参数和返回类型(示例 17.24)。不要在 LifeSlice 中使用 QPixmap[①]。

示例 17.24 src/threads/life/lifemainwindow.cpp

```
[ . . . . ]
struct LifeFunctor : public std::unary_function<LifeSlice, LifeSlice> {
    LifeSlice operator() (LifeSlice slice);
};

LifeSlice LifeFunctor::operator()(LifeSlice slice) {                    1
    QRect rect = slice.rect;
    QImage image = slice.image;
    QImage next = QImage(rect.size(), QImage::Format_Mono);
    next.fill(DEAD);
    int h = rect.height();  int w = rect.width();

    for (int c=0; c<w; ++c) {
        for (int r=0; r<h; ++r) {
            int x = c+rect.x();
            int y = r+rect.y();
            bool isAlive = (image.pixelIndex(x, y) == ALIVE);
            int nc = neighborCount(image, x, y);
            if (!isAlive && nc == 3)
                 next.setPixel(c, r, ALIVE);
            if (!isAlive) continue;
            if (nc == 2 || nc == 3)
                next.setPixel(c,r, ALIVE);
        }
    }
    slice.image = next;
    return slice;
}
```

1 映射函数。

① 17.2 节中讨论过这一限制。通常情况下，在 GUI(主)线程之外使用 QPixmap 是不安全的。

LifeFunctor 派生自一个一元函数对象，来自标准库模板的 unary_function 会给仿函数附加额外的类型“特征”（traits），可被泛型算法用来获得仿函数的参数及返回类型。这个仿函数会定义一个带有 LifeSlice 参数并返回 LifeSlice 值的 operator()。

映射函数的返回类型必须是示例 17.25 中的规约函数的(第二个)参数类型。

示例 17.25　src/threads/life/lifemainwindow.cpp

```
[ . . . . ]
void stitchReduce(QImage& next, const LifeSlice &slice) {
    if (next.isNull())
        next = QImage(boardSize, QImage::Format_Mono);
    QPainter painter(&next);
    painter.drawImage(slice.rect.topLeft(), slice.image);          1
}
```

1 在一个图片上绘制另一个图片的一部分。

规约函数必须以某种方式共同带有每个工作者线程产生的部分结果并重新组合成一个连贯的 QImage。在这个拼图练习中，映射函数使用高层次的 QPainter API 来把一幅画绘制到另一幅画上，避免了单像素赋值对嵌套循环的需求。

示例 17.26 中的主循环必须将问题分成更小的问题，把它们发送到 QtConcurrent 的 blockingMappedReduced()，并且把结果发送到 LifeWidget。

示例 17.26　src/threads/life/lifemainwindow.cpp

```
[ . . . . ]

void LifeMainWindow::calculate() {
    int w = boardSize.width();
    // This might not be optimal, but it seems to work well...
    int segments = QThreadPool::globalInstance()->maxThreadCount() * 2;
    int ws = w/segments;                                           1
    LifeFunctor functor;                                           2
    while (m_running) {
        qApp->processEvents();                                     3
        m_numGenerations++;
        QList<LifeSlice> slices;                                   4
        for (int c=0; c<segments; ++c) {
            int tlx = c*ws;
            QRect rect(tlx, 0, ws, boardSize.height());
            LifeSlice slice(rect, m_current);
            slices << slice;                                       5
        }
        m_current = QtConcurrent::blockingMappedReduced(slices, functor,
                    stitchReduce, QtConcurrent::UnorderedReduce );  6
        m_lifeWidget->setImage(m_current);
    }
}
```

1 段的宽度。

2 映射仿函数。

3 确保 GUI 仍旧可以响应。

4 分成更小的块。

5 把小块添加到一个以并行方式处理的集合中。

6 开始并行工作。当其准备好后，可在每个块上都可以调用 stitchReduce。

这个循环中的 qApp->processEvents()是必须的，以便主事件循环来接收和处理其他 GUI 事件。如果注释掉这一行并试着运行应用程序，就会注意到，一旦计算开始就没有办法停止或退出。

在 1024 × 768 的版幅上，这个程序使用 4 个线程时可获得每秒 10 帧(frames per second，fps)，而用一个线程相对可获得 4 fps。这里并不是一个因子 4，但使用更大的幅面时，可能会观察到更好的改善。记住，在计算完每一代之后，都不可避免地有一个同步点。

17.3 练习：QThread 和 QtConcurrent

1. 在不使用 QtConcurrent 的情况下，编写一个多线程的游戏人生的例子，其中的 LifeServer 创建并管理 LifeWorker 线程(可多达 QThreadPool::maxThreadCount()个)，同时把计算分配到这些线程以获得更快的速度。让用户可以从 QSpinBox 中设置线程的数目。

   使用 QMutex 或 QReadWriteLock 对共享数据的访问进行同步。以示例 17.14 中 PrimeThreads 的生产者与消费者的例子为指南，并可复用 17.2.4 节中 Life 例子的任意代码。图 17.7 给出了一个用 UML 表示的可能的高层次设计。

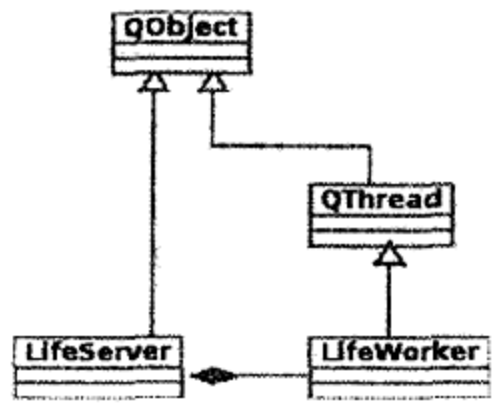


图 17.7 多线程的 Life UML

2. 从 17.2.2 节中给出的应用程序开始，在 GUI 中添加一个可使这两种算法工作的进度条。
3. 在这个练习中，重新回顾一下那些在 10.5.1 节中介绍过的图像处理技术并将其用于并行，通过多个线程来产生随机拼图。

   编写一个应用程序，让用户可从磁盘中选择尽可能多的图片，然后使用 QtConcurrent 算法在随机数量的图像副本上随机使用图像处理功能(如果适当，可以带一些随机参数)。

   在图像操作完成后，对每个处理过的图像使用 QtConcurrent 算法进行随机缩减，并将其绘制到拼图最初空白图像的随机位置。

   在拼图生成并保存到磁盘后，在屏幕上显示这个拼图。图 17.8 是使用 28 张照片产生的，其中的每个都被复制了 1 至 5 次，经操纵、缩放并插入到一个(默认的)640 × 480 图像中。最终拼图的大小可以用命令行参数设置。

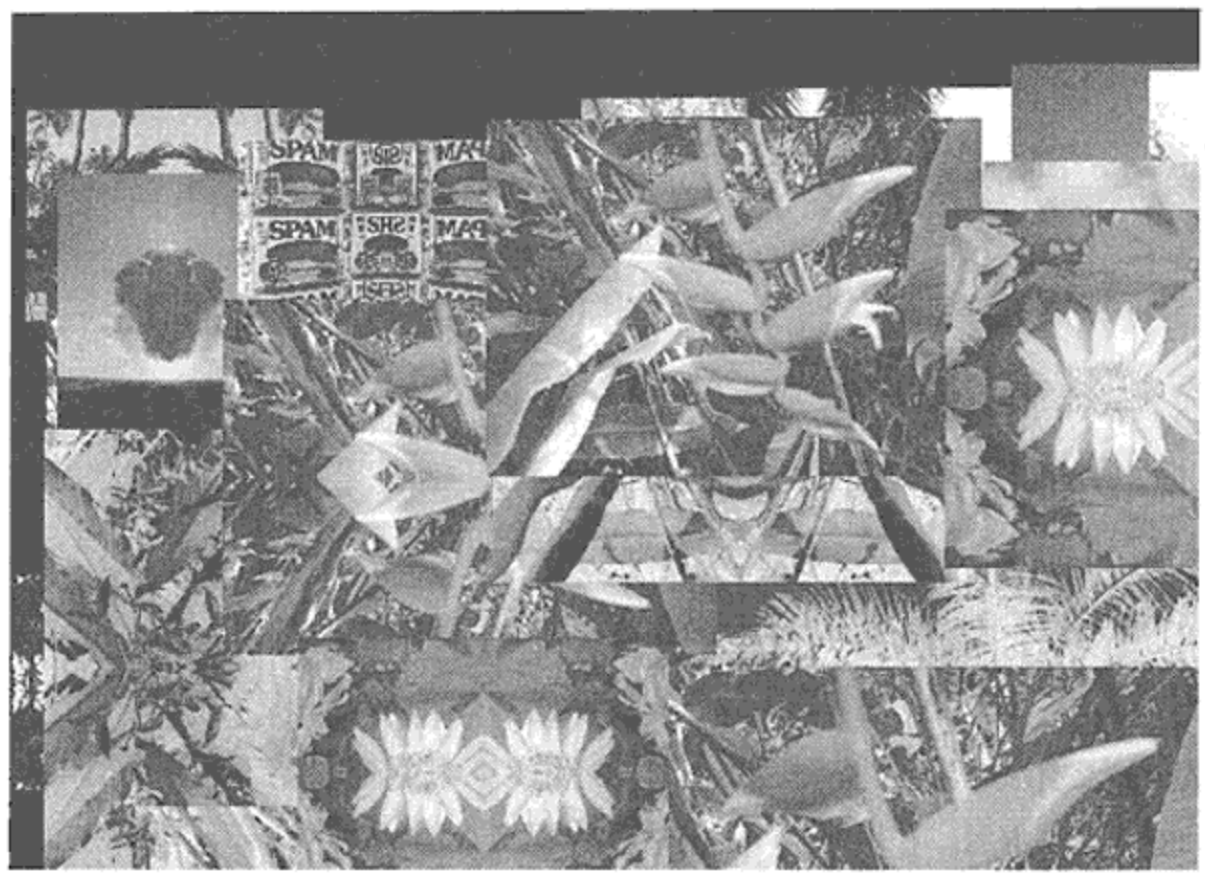


图 17.8　拼图示例

17.4　复习题

1. 列举并解释至少两种父进程可以用来向其子进程传递信息的机制。
2. 列举并解释至少两种线程之间彼此同步的机制。
3. 在什么情况下可以使用一个 `QTimer` 来代替一个 `QThread`？为什么要这么做？
4. 对于函数来说，线程安全意味着什么？
5. 哪个类可用于非 GUI 型线程？是 `QImage` 还是 `QPixmap`？
6. 对于函数来说，可重入(reentrant)意味着什么？
7. 怎样让一个非 GUI 线程进入事件循环？
8. 无须使用其他额外线程，在执行一个长期运行的循环同时，如何保持 GUI 的响应能力？

第 18 章　数据库编程

本章给出了 Qt 的 SQL 类功能的大致介绍，例子后端使用的是 sqlite。

需要学习结构化查询语言吗

学习 SQL 不要为寻找任何一本声称“标准”SQL(Structured Query Language)的书而苦恼。需要阅读的最好的手册就是所选数据库服务器软件的参考指南，因为每个服务器所支持的语法和数据类型都会略有不同。幸运的是，创建表和其他 DDL 查询是一个例外，通过 QSQL 应用程序编程接口(API)可以很轻松地实现 SQL 值和 QVariant 之间的映射。当使用了多个数据库后，就应该或者寻找一个抽象层(根据编写的好坏程度，可能最终会让阻隔层成为历史)，或者至少为共同特性编写一个 SQL(SQLite)。

Qt 提供了一个平台中立的与 JDBC 类似的数据库接口[①]。它需要为每一个可以连接的特定数据库提供驱动程序。为了构建面向特定数据库的驱动程序，开发数据库的头文件和库必须可用。可用 Qt 连接各种不同的 SQL 数据库，包括 Oracle, Postgre SQL 和 Sybase SQL 的数据库。下面的例子中已经在 Linux 上用 MySQL 和 SQLite 测试了代码。

如果要 QtSQL 开发新东西，建议使用 SQLite 的语法，因为它有如下特点。

1. 是开源的。
2. 是 Qt 自带的。
3. 不需要从源代码构建 Qt 中，不需要构建插件，也不需要设置独立服务器。
4. 每个数据库都会映射成磁盘上的一个单独的文件。
5. 支持 SQL 的子集，该子集在其他大多数系统上都是可用的。

SQLite 是一个进程内的、零配置的数据库类库。它并不单独作为服务器运行。建议在应用程序中使用 SQLite，因为它有较好的并发性、可靠性和高性能，还拥有更快的启动/关机时间和更小的内存等需求，因为所连接的并不是诸如 MySQL 这样的外部数据库进程。但是，SQLite 应足以应付简单的实验需求和 SQL 学习需要。

其他数据库驱动程序

欲了解更多信息，可以参阅 Qt 的 SQL 驱动程序[②]文档页面。

① Java Database Connectivity(Java 数据库连接，JDBC)API。

② 参见http://doc.qt.nokia.com/latest/sql-driver.html。

已经支持的驱动程序有哪些

欲要找出当前版本的 Qt 中哪种驱动程序是可用的，有下面几种途径。

1. 运行$QTDIR/demos/sqlbrowser，然后可以在初始的 Connection Settings 对话框的组合框中看到一个驱动程序列表，如图 18.1 所示。
2. 在代码中调用 QSqlDatabase::drivers()。

提示

如果打算和 SQL 撇清关系并直接把对象映射到永久存储设备上，你可能会对 Code Synthesis ODB 感兴趣[①]，这是一个开源的、支持 Qt 类型的对象-关系型映射层。

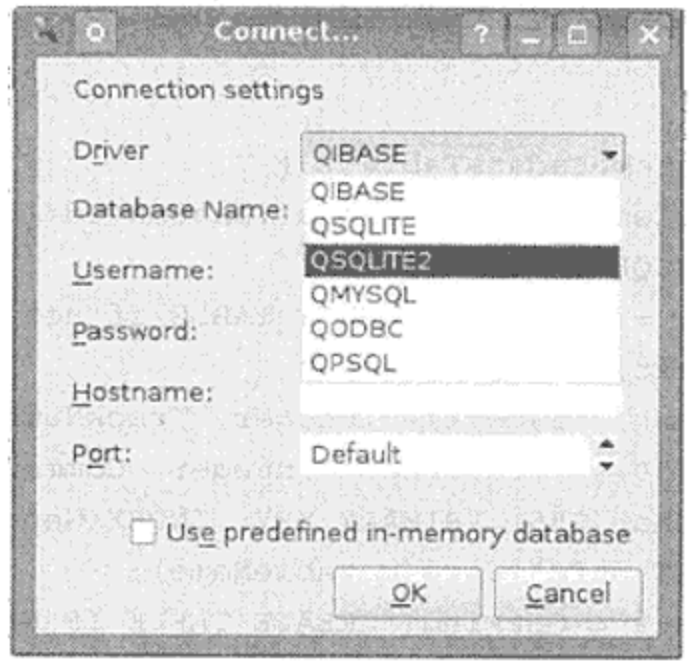


图 18.1　Sql Browser Connection Settings 对话框

18.1　QSqlDatabase：从 Qt 连接 SQL

QSqlDatabase 这个名称容易让人对这个类产生误解。它表示的并不是磁盘上的一个数据库，而是一个到数据库的连接。这个类更好的名称应该是 QSqlConnection。

与数据库服务器相连接需要这些信息：驱动程序类型、主机名、用户名、密码和数据库名称，如示例 18.1 所示。使用 SQLite 时，只需一个文件名，它会传递给 QSqlDatabase::setDatabaseName()。

用 static QSqlDatabase:: addDatabase() 函数可以创建一个初始连接(即，QSqlDatabase 的一个实例)。可以给该实例一个可选的连接名称，也可以在随后用这个名称获得该连接。默认连接可以使用 QSqlDatabase::database()函数。

示例 18.1　src/sql/testprepare/testprepare.cpp

```
[ . . . . ]

void testprepare::testPrepare() {
```

① 参见http://www.codesynthesis.com/products/odb/。

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
db.setHostName("localhost");
db.setUserName("amarok");
db.setPassword("amarok");
db.setDatabaseName("amarok");
QVERIFY(db.open());
```


18.1.1 数据定义语言语句：定义表

每个数据库都有一个表的集合。修改表定义的查询语句称为 DDL(Data Definition Language，DDL，数据定义语言)语句。表与结构数组非常相似，其中的每个数据成员对应数组中的一列，每个对象大致相当于一个记录，或者对应于表中的一行。要定义一个表，必须说明一个记录看起来像什么。这就意味着要定义每一列，可想像成字段、属性或数据成员。示例 18.2 定义了一个称为 MetaData 的 SQL 表。

示例 18.2 src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

bool MetaDataTable::createMetadataTable() {
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    if (m_driver == "QMYSQL")
       m_createTableQStr = QString("CREATE TABLE if not exists %1 ("
         "TrackTitle  text, Artist text, "
         "AlbumTitle  text, TrackTime integer, TrackNumber integer, "
         "Genre varchar(30),  Preference integer, Comment  text, "
         "FileName  varchar(255) PRIMARY KEY, INDEX(Genre) ) "
         "DEFAULT CHARSET utf8").arg(m_tableName);
    else m_createTableQStr = QString("CREATE TABLE IF NOT EXISTS %1 ("
         "TrackTitle  text, Artist text, AlbumTitle  text, "
         "TrackTime integer, TrackNumber integer, Genre varchar(30), "
         "Preference integer, Comment  text, FileName  varchar(255) "
         "PRIMARY KEY)").arg(m_tableName);
    QSqlQuery q(m_createTableQStr);
    if (!q.isActive()) {
       qDebug() << "Create Table Fail: " << q.lastError().text()
                << q.lastQuery();
       return false;
    }
    db.commit();
    return true;
}
```

1 用 SQLite 3 测试通过。

根据所用 QSqlDriver 的不同，会用 SQL 创建不同的字符串。想要额外支持的数据库必须单独测试，因为从一台服务器到另一台不同的服务器其 SQL 语法可能会有所不同。例如，在 MySQL 中，起初使用 time 作为 TrackTime 的列类型，但随后将其改变成 integer[①]，以便可以在两种数据库中都可以使用相同的模式。

① 注意，要与 C++的类型名称区分开。

数据库连接打开后，就可以使用一个强大的称为 QSqlQuery 的类，其中有一个成员函数 exec()。

18.1.2　预处理语句：插入行

在使用 QSqlQuery 时，有两种执行 SQL 语句的方法：

1. QSqlQuery .exec(QString)
2. QSqlQuery .prepare(QString)

exec(QString)要慢一些，因为它需要服务器来解析每个 SQL 语句。预处理语句(prepared statement)更安全些，因为不需要对字符串进行转义。它们也比较快，尤其是当重复执行带不同参数的相同 SQL 语句时。SQL 驱动程序只需解析一次查询字符串。

示例 18.3 给出了预处理语句插入或更新行的用法。这里使用的是命名参数(named parameter)，但也有可能使用诸如 addBindValue 和形如 ":1 "、" :2 "等的定位参数(positional parameter)。在 MySQL 中，诸如插入或更新一行这样的单一 SQL 操作语句与 SQLite 的语句略有不同。如此一来，就有两个不同的插入字符串。

示例 18.3　src/libs/sql/metadata/metadatatable.cpp

```
[ . . . . ]

MetaDataTable::MetaDataTable(QObject* parent)
    : QObject(parent), m_tableName("MetaData") {
    setObjectName(m_tableName);
    m_mdl = Abstract::MetaDataLoader::instance();
    m_driver = DbConnectionSettings::lastSaved().driverName();
    Q_ASSERT(createMetadataTable());
    QString preparedQuery = "INSERT into MetaData"                    1
        "(Artist, TrackTitle, AlbumTitle, TrackNumber, TrackTime, Genre,"
        "Preference, FileName, Comment) VALUES (:artist, :title, :album,"
        ":track, :time, :genre, :preference, :filename, :comment) "
        "ON DUPLICATE KEY UPDATE Preference=VALUES(Preference),"
        "Genre=VALUES(Genre), AlbumTitle=VALUES(AlbumTitle),"
        "TrackTitle=VALUES(TrackTitle), TrackNumber=VALUES(TrackNumber),"
        "Artist=VALUES(Artist), COMMENT=VALUES(Comment)";
    if (m_driver == "QSQLITE") {
        preparedQuery = "INSERT or REPLACE into MetaData"
            "(Artist, TrackTitle, AlbumTitle, TrackNumber, TrackTime, "
            "Genre, Preference, FileName, Comment)"
            "VALUES (:artist, :title, :album, :track, :time, :genre, "
            ":preference, :filename, :comment)";
    }
    bool prepSuccess = m_insertQuery.prepare(preparedQuery);
    if (!prepSuccess) {
        qDebug() << "Prepare fail: " << m_insertQuery.lastError().text()
                 << m_insertQuery.lastQuery();
        abort();
    }
```

1 用 MySQL 5 测试通过。

某些情况下，Qt 的 SQL 驱动程序可能不支持服务器端的预处理查询，但使用 Qt SQL，仍然可以使用客户的转义字符进行预处理查询，这是插入数据或处理用户提供的数据的最安全方式。预处理语句可以让你免受 SQL 注入攻击和其他可能出现的解析错误，且应该会比每次执行都需要解析的常规查询更快一些。

示例 18.4 给出了使用中的预处理语句。首先，对每一列的查询调用 `bindValue()`，然后再调用 `exec()`。

示例 18.4 src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

bool MetaDataTable::insert(const MetaDataValue &ft) {
    using namespace DbUtils;

    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QSqlRecord record = db.record(m_tableName);
    if (record.isEmpty() && !createMetadataTable()) {
        qDebug() << "unable to create metadata: "
                 << db.lastError().text();
        return false;
    }

    m_insertQuery.bindValue(":artist", ft.artist());
    m_insertQuery.bindValue(":title", ft.trackTitle());
    m_insertQuery.bindValue(":album", ft.albumTitle());
    m_insertQuery.bindValue(":track", ft.trackNumber());
    QTime t = ft.trackTime();
    int secs = QTime().secsTo(t);
    m_insertQuery.bindValue(":time", secs);
    m_insertQuery.bindValue(":genre", ft.genre());
    m_insertQuery.bindValue(":filename", ft.fileName());
    int pref = ft.preference().intValue();
    m_insertQuery.bindValue(":preference", pref);
    m_insertQuery.bindValue(":comment", ft.comment());

    bool retval = m_insertQuery.exec();
    if (!retval) {
        qDebug() << m_insertQuery.lastError().text()
                 << m_insertQuery.lastQuery();
        abort();
    }
    emit inserted(ft);
    return retval;
}
```

正如所看到的，Qt SQL 针对不同的数据库引擎并不提供一种“一次编写，随处运行”的方式。尽管把列映射成属性是有可能的，但还是需要编写对象—关系型映射代码并必须在不同的服务器上进行测试。

18.2　查询和结果集

示例 18.4 中在 MetaData 表中插入了几行。这个应用程序中会试着将可能需要的对 SQL 表 MetaData 的全部 SQL 操作都封装在名称为 MetaDataTable 的类中。示例 18.5 中给出了一个简单查询，它返回一个 QStringList。

示例 18.5　src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

QStringList MetaDataTable::genres() const {
    QStringList sl;
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QSqlQuery q("SELECT DISTINCT Genre from MetaData");
    if (!q.isActive()) {
        qDebug() << "Query Failed: " << q.lastQuery()
                 << q.lastError().text();
    } else while (q.next()) {
        sl << q.value(0).toString();
    }
    return sl;
}
```

当需要返回一行数据时，API 中最好的表达方式是什么呢？推荐返回一个带 getter 方法和 setter 方法的对象，但该对象应该是一个堆型的 MetaDataObject 还是一个栈型的 MetaDataValue，尚值得讨论。如果返回的是指向这里所创建堆对象的指针，那么必须注意的是，到底是谁在拥有它们并负责在随后删除它们。示例 18.6 给出了另外一种方法，把 QObject 转换成其基类的值类型并返回这种类型值。

示例 18.6　src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

MetaDataValue MetaDataTable::findRecord(QString fileName) {
    using namespace DbUtils;
    QFileInfo fi(fileName);
    MetaDataObject f;
    if (!fi.exists()) return f;                                    1
    QString abs = fi.absoluteFilePath();
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QString qs = QString("select * from %1 where FileName = \"%2\"")
                 .arg(m_tableName).arg(escape(abs));
    QSqlQuery findQuery(qs);
    if (!findQuery.isActive()) {
        qDebug() << "Query Failed: " << findQuery.lastQuery()
                 << findQuery.lastError().text();
        return f;
    }
    if (!findQuery.first()) return f;
    QSqlRecord rec = findQuery.record();
    for (int i=rec.count() -1; i >= 0; --i) {                      2
```

```
    QSqlField field = rec.field(i);
    QString key = field.name();
    QVariant value = field.value();
    if (key == "Preference") {
        int v = value.toInt();
        Preference p(v);
        f.setPreference(p);
    }
    else if (key == "TrackTime") {                                    3
        QTime trackTime;
        trackTime = trackTime.addSecs(value.toInt());
        f.setTrackTime(trackTime);
    }
    else {
        f.setProperty(key, value);                                    4
    }
    }
    return f;                                                         5
}
```

1 QObject 按值返回？别忘了，MetaDataValue 才是这个特殊 QObject 的基类。

2 QObject 中的各个属性会映射成表中的列名/字段值。

3 SQLite 没有 time 类型，所以必须存储为 int。

4 对于其他列，使用 QObject 的 setProperty 函数。

5 从本地将要销毁的栈 QObject 创建一个值类型。

这个例子中创建了一个 MetaDataObject 栈用于设置属性，然后按值的方式返回它，所以会返回一个临时的 MetaDataValue。这可以用来说明派生对象是如何隐式转换成基类类型的[①]。

18.3 数据库模型

图 18.2 给出了用于连接到 QTableView 的一些具体模型类。

如果打算显示一个在调用示例 18.4 后创建的表，只需用 QSqlTableModel 的 5 行代码即可实现。

示例 18.7 src/libs/tests/testsqlmetadata/testsqlmetadata.cpp

```
[ . . . . ]

void TestSqlMetaData::showTable() {
    QSqlTableModel model;
    model.setTable("MetaData");
    model.select();
    QTableView *view = new QTableView;
    view->setModel(&model);
    view->setItemDelegate(new SimpleDelegate(view));
```

① 提供的复制构造函数不是私有的。

示例 18.7 中摘录的这个测试用例会扫描通过环境变量 TESTTRACKS 设置选择的目录，还会将找到的每个 MP3 文件的元数据添加到表模型中。该测试示例已包含在 dist 目录的源代码压缩包中。

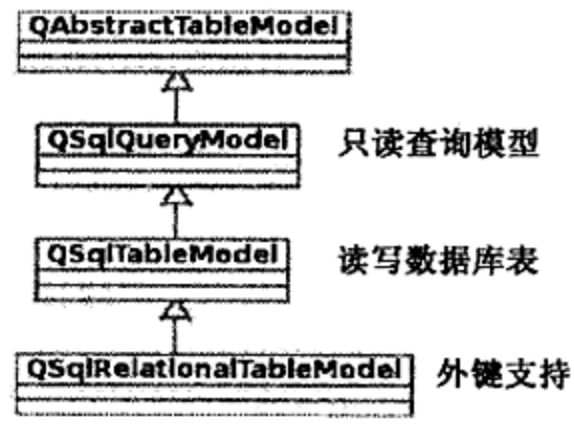


图 18.2　SQL 表模型

18.4　复习题

1. Qt 在所有平台上都包含的数据库是哪一个？
2. 如何确定你的 Qt 支持哪些数据库？
3. QSqlDatabase 是对什么的抽象：文件、连接、用户还是表？
4. DDL 查询和常规查询之间的区别是什么？
5. 为什么预处理查询要优于常规查询？
6. 类 QSqlDatabase 的名称最好应为什么？
7. 如果数据库驱动程序的 hasFeature(QSqlDriver:: PreparedQueries)返回 false，你还可以用预处理查询吗？
8. 非 DDL 查询可以修改表中的行吗？

第二部分　C++语言规范

第 19 章　类型与表达式

本章将深入探讨 C++的强类型化系统概念，也会讲解如何求解并转换表达式。

首先将形式化地给出一些术语的定义。运算符是一种特殊类型的函数，它对操作数执行计算并返回结果，操作数是提供给运算符的实参。

可以将运算符看成常规的函数，但有一些运算符使用中缀运算符符号(例如，+，-，*，/等)。因此，除了可以使用更长的函数调用句法之外(例如，`str3 = operator+(str1, str2);`)，还可以使用更具可读性的中缀句法(例如，`str3 = str1 + str2;`)。

表达式可以由带有单一操作数、多个操作数的运算符或者包含实参的函数组成。每一个表达式都具有类型和值。值是通过将运算符(或者函数)的定义应用于操作数(或者实参)而求得的。

19.1　运算符

根据用途的不同，运算符被分成如下几个大类：

赋值运算符	=, +=, *=, …
算术运算符	+, -, *, /, %
关系运算符	<, <=, >, >=, ==, !=
逻辑运算符	&&, ||, !
位运算符	&, |, ^, ~, <<, >>
内存管理运算符	new, delete, sizeof
指针运算符和访问运算符	*, &, ., ->, [], ()
作用域解析运算符	::
其他运算符	条件(?:)，逗号(,)

正如表 19.1 中所示，C++标准定义的一些关键字可以充当某些运算符符号的别名。

表 19.1　运算符的别名

运　算　符	别　　名	运　算　符	别　　名
&&	and(与)	^	xor(异或)
&	bitand(位与)	^=	xor_eq(异或等于)
&=	and_eq(与等于)	!	not(非)
||	or(或)	!=	not_eq(非等于)
|	bitor(位或)	~	compl(补)
|=	or_eq(或等于)		

对于内置类型，运算符具有预先定义的含义，但并不是所有的运算符都是为内置类型而定义的。

运算符的特性

运算符具有下列几个特性：

- 优先级
- 结合性
- 所要求的操作数数量

表 19.2 中列出了全部 C++运算符以及它们的特性，按照优先级和用途分组，高优先级的先列出。

- “操作数”列包含运算符所要求的操作数数量。
- “描述”列包含针对内置类型该运算符的传统含义。
- “结合性”列给出的结合性，表示如果同一个运算符在一个表达式出现多次，则应该如何对其进行求值。
  - “左”表示结合性为从左到右。例如：

```
d = a + b + c;      //首先求值 a + b，然后求值(a + b) + c
```

    赋值运算符是最后求值的，因为它具有更低的优先级。
  - “右”表示结合性为从右到左。例如：

```
c = b = a;          //先将 a 赋值给 b，然后赋值给 c
```

- “重载性”列表示这个运算符是否可以针对定制类型被重载(重定义)。

  这个列可能的值包括：

  - Y。运算符可以被重载成全局函数或者成员函数。
  - M。运算符只可以被重载成类成员函数。
  - N。运算符不能被重载。

19.1.1　运算符表

表 19.2　C++运算符

运算符	操作数	描述	例子	结合性	重载性
::	1	全局作用域解析	:: name	右	N
::	2	类/命名空间作用域解析	*className::memberName*	左	N
->	2	通过指针的成员选择器	*ptr->memberName*	左	N
.	2	通过对象的成员选择器	*obj.memberName*	左	N
->	1	智能指针	*obj->member*	右	M
[ ]	2	下标运算符	*ptr[expr]*	左	M
()	任意个	函数调用	*function(argList)*	左	N
()	任意个	值构造	*className(argList)*	左	M
++	1	后递增	*varName*++	右	Y
--	1	后递减	*varName*--	右	Y
typeid	1	类型识别	typeid(*type*)或者typeid(*expr*)	右	N
dynamic_cast	2	运行时经检验的类型转换	dynamic_cast<*type*>(*expr*)	左	N
static_cast	2	编译时经检验的类型转换	static_cast<*type*>(*expr*)	左	N
reinterpret_cast	2	未检验的转换	reinterpret_cast<*type*>(*expr*)	左	N
const_cast	2	常量转换	const_cast<*type*>(*expr*)	左	N
sizeof	1	字节大小	sizeof *expr*或者sizeof(*type*)	右	N

(续表)

运算符	操作数	描述	例子	结合性	重载性
++	1	前递增	*++varName*	右	Y
--	1	前递减	*--varName*	右	Y
~	1	逐位取反	*~ expr*	右	Y
!	1	逻辑非	*! expr*	右	Y
+, -	1	一元加，一元减	*+expr* 或者 *-expr*	右	Y
*	1	指针解引用	** ptr*	右	Y
&	1	取址	*& lvalue*	右	Y
new	1	分配	new *type* 或者 new *type*(*exprlist*)	右	Y
new [ ]	2	分配数组	new *type*[*size*]	左	Y
delete	1	解除内存分配	delete *ptr*	右	Y
delete [ ]	1	解除数组分配	delete [ ] *ptr*	右	M
()	2	C 风格的类型转换	(*type*) *expr*	右	N[b]
->*	2	通过指针的成员指针选择器	*ptr->*ptrToMember*	左	M
.*	2	通过对象的成员指针选择器	*obj.*ptrToMember*	左	N
*	2	乘	*expr1 * expr2*	左	Y
/	2	除	*expr1 /expr2*	左	Y
%	2	求余	*expr1 % expr2*	左	Y
+	2	加	*expr1 + expr2*	左	Y
-	2	减	*expr1 expr2*	左	Y
<<	2	位左移	*expr << shiftAmt*	左	Y
>>	2	位右移	*expr >> shiftAmt*	左	Y
<	2	小于	*expr1 < expr2*	左	Y
<=	2	小于或者等于	*expr1 <= expr2*	左	Y
>	2	大于	*expr1 > expr2*	左	Y
>=	2	大于或者等于	*expr1 >= expr2*	左	Y
==	2	等于[c]	*expr1 == expr2*	左	Y
!=	2	不等于	*expr1 != expr2*	左	Y
&	2	位与	*expr1 & expr2*	左	Y
^	2	位异或	*expr1^e2*	左	Y
|	2	位或	*expr1 | expr2*	左	Y
&&	2	逻辑与	*expr1 && expr2*	左	Y
||	2	逻辑或	*expr1 || expr2*	左	Y
=	2	赋值	*expr1 = expr2*	右	Y
*=	2	乘并赋值	*expr1 *= expr2*	右	Y
/=	2	除并赋值	*expr1 /= expr2*	右	Y
%=	2	求余并赋值	*expr1 %= expr2*	右	Y
+=	2	加并赋值	*expr1 += expr2*	右	Y
-=	2	减并赋值	*expr1 -= expr2*	右	Y
<<=	2	左移并赋值	*expr1 <<= expr2*	右	Y
>>=	2	右移并赋值	*expr1 >>= expr2*	右	Y
&=	2	与并赋值	*expr1 &= expr2*	右	Y

（续表）

运算符	操作数	描述	例子	结合性	重载性
|=	2	或并赋值	*expr1* |= *expr2*	右	Y
^=	2	异或并赋值	*expr1* ^= *expr2*	右	Y
?:	3	条件表达式	*bool* ? *expr* : *expr*	左	N
throw	1	抛出异常	throw *expr*	右	N
	2	顺序求值(逗号)	*expr* , *expr*	左	Y

a 可以将函数调用运算符声明成带任意数量的操作数。

b 类型转换运算符可以使用构造函数或者转换运算符来转换定制类型。

c 不能将这个运算符用于 float 或者 double 操作数。它要求的是精确匹配，这与体系结构有关且可能导致意料之外的结果。

19.2　语句与控制结构

语句是可执行的代码块。控制结构是控制其他语句的执行的语句。本章将形式化地定义一些语言元素，并会给出一些可用的控制结构类型。

19.2.1　语句

C++程序包含的语句可以改变由程序管理的存储状态，并且可以决定程序的执行流程。存在多种类型的 C++语句，其中的大多数都继承自 C 语言。首先，最简单的语句是以分号结尾的语句：

```
x = y + z;
```

其次，是复合语句或者语句块，它由包含在一对大括号中的语句序列组成：

```
{
    int temp = x;
    x = y;
    y = temp;
}
```

上面是一个简单的复合语句，它包含三条简单语句，从上到下依次执行。变量 temp 为这个语句块的局部变量，当到达语句块的结尾处时，它会被销毁。复合语句可以包含其他的复合语句。

通常而言，可以将复合语句放于简单语句能够存在的任何位置。但是，反过来并不成立。特别地，函数定义

```
double area(double length, double width) {
    return length * width;
}
```

不能被替换成

```
double area(double length, double width)
   return length * width;
```

函数定义的语句体必须总是一个语句块。

19.2.2　选择语句

任何编程语言都至少存在一种控制结构，它使得程序的执行流程能够根据某个布尔条件的输出而变化。C 和 C++语言中都有 if 语句和 switch 语句。if 语句通常具有下列形式：

```
if(boolExpression)
  statement
```

它还可以具有一个可选的 else 分支：

```
if(boolExpression)
  statement1
else
  statement2
```

条件语句可以嵌套，这意味着它们可以变得相当复杂。需记住的一个重要原则是：如果紧挨在 else 或者 else if 子句前面的 if 条件中的 *boolExpression* 求值为 false，则会执行这个子句。如果程序逻辑允许省略某些 else 子句，则这个原则可能会使人迷惑。考虑下面这个故意设计错误的例子，其中 x 为 int 类型：

```
if (x>0)
    if (x > 100)
        cout << "x is over a hundred";
else
    if (x == 0)  // no! this cannot be true -the indentation is misleading
        cout << "x is 0";
else
    cout << "x is negative"; // no! x is between 1 and 100 inclusive!
```

利用大括号，就可以找出并修复这个逻辑错误：

```
if (x>0) {
    if (x > 100)
        cout << "x is over a hundred";
}
else
    if (x == 0)  // now this is possible.
        cout << "x is 0";
    else
        cout << "x is negative";
```

没有 else 分支的 if 语句，可以通过将 if 下面的语句放入一对大括号中，使其成为一个复合语句。

switch 语句

switch 是另一种分支结构，它允许根据参数的值执行不同的代码。

```
switch(integralExpression) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
        ...

    case valuen:
        statementn;
        break;
    default:
        defaultStatement;
}
nextStatement;
```

switch 语句是一种具有计算功能的 goto 语句。switch 语句中的每一个 case 后面都有一个唯一的标记值，它会与 *integralExpression* 比较。

如果某个 case 标记值等于 *integralExpression*，switch 就会跳到这个 case 后面的语句执行，直到到达 switch 语句块的结尾处或者遇到一条分支语句(例如，break)。

如果 *integralExpression* 没有与之相等的 case 标记值，就会跳到可选的 default 标记处执行。如果没有 default 标记且不存在匹配的 case 标记，则会跳到 *nextStatement* 去执行。

integralExpression 必须是一个能够求值为整数的表达式。除了 default 标记外，每一个 case 标记值都必须是一个整数常量[①]。

与上面相同的任何 switch 语句，都可以被重写为一个长的 if…else 语句。但是，switch 语句的运行时性能要好得多，因为它只需一次比较且只会执行一个分支。对应的 if…else 语句如下：

```
if(integralExpression == value1)
   statement1;
else if(integralExpression == value2)
   statement2;
...
else if(integralExpression == valuen)
   statementn;
else
   defaultStatement;
```

注意

长的复合条件语句和 switch 语句应当避免在面向对象编程中使用(除非它们只位于工厂代码中)，因为它们会使函数变得复杂并难以维护。

如果每一个 case 都能够被重写为不同类的方法，则可以使用策略模式(以及虚表)，而不必自己编写 switch 语句。

19.2.2.1　练习：选择语句

假设你是计算机，预测示例 19.1 的输出结果。然后，在计算机上运行它并将结果与预测值进行比较。

示例 19.1　src/early-examples/nestedif.cpp

```
#include <iostream>
using namespace std;

void nestedif1 () {
    int m = 5, n = 8, p = 11;
    if (m > n)
        if (p > n)
            cout << "red" << endl;
        else
            cout << "blue"  << endl;
```

① case 标记与 goto 标记并不相同，后者被用作声名狼藉的 goto 语句的执行目标。goto 标记必须是标志符。特别地，这种标志符不能是整数。

```
}


void nestedif2() {
    int m = 5, n = 8,  p = 11;
    if (m > n) {
        if (p > n)
            cout << "red"  << endl;
    } else
        cout << "blue" << endl;
}


int main() {
    nestedif1();
    nestedif2();
    return 0;
}
```


迭代

C++提供了三种迭代结构：

1. while 循环

```
while ( loopCondition ) {
   loopBody
}
```

   a. 首先求值 *loopCondition*。
   b. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。

2. Do…while 循环

```
do {
   loopBody
} while ( loopCondition ) ;
```

   a. 首先执行 *loopBody*。
   b. 求值 *loopCondition*。
   c. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。

3. for 循环

```
for ( initStatement; loopCondition; incrStmt ) {
   loopBody
}
```

   a. 首先执行 *initStatement*。
   b. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。
   c. 每次执行完 *loopBody* 后，执行 *incrStmt*。

对于这些迭代结构，只要 *loopCondition* 求值为 true，就会重复地执行 *loopBody* 代码。do 循环与另外两种不同，它的 *loopCondition* 在循环的底部进行检验，所以它的 *loopBody* 总是至少会执行一次。

一种常见的编程错误是在 while 的后面放置一个分号：

```
while (notFinished());
    doSomething();
```

第一个分号会终止整个 while 语句，并会导致一个具有空 loopBody 的循环。尽管 doSomething()被缩进了，但是它不会在循环中执行。*loopBody* 负责改变 *loopCondition* 的值。如果开始时 notFinished()不为 true，则空 loopBody 会导致无限循环。如果开始时 notFinished()为 false，则循环会立即终止，而 doSomething()会恰好执行一次。

为了对循环中执行的代码进行更好的控制，C++提供了 break 语句和 continue 语句：

```
while ( moreWorkToDo ) {
    statement1;
    if ( specialCase ) continue;
    statement2;
    if ( noMoreInput ) break;
    statement3;
// continue jumps here
}
// break jumps here
```

break 会跳出当前的控制结构，不管这个控制结构是 switch，for，while 还是 do…while。

continue 只在循环内部执行，它会跳过当前迭代中剩下的语句而去检验 *moreWorkToDo* 条件。示例 19.2 中给出了 continue 语句和 break 语句的用法。

示例 19.2　src/continue/continue-demo.cpp

```
#include <QTextStream>
#include <cmath>

int main() {
   QTextStream cout(stdout);
   QTextStream cin(stdin);
   int num(0), root(0), count;
   cout << "How many perfect squares? "<< flush;
   cin >> count;
   for(num = 0;; ++num) {
      root = sqrt(num);                                    1
      if(root * root != num)
         continue;
      cout << num << endl;
      --count;
      if(count == 0)
         break;
   }
}
```

1 将 sqrt 转换成 int。

19.2.3.1　练习：迭代

1. 编写函数 isPrime(int n)，如果 n 为素数就返回 true，否则返回 false。提供一个交互式的 main()函数来测试这个函数。
2. 编写函数 primesBetween(int min, int max)，它在屏幕上显示位于 min 和 max 之间的全部素数。提供一个交互式的 main()函数来测试这个函数。

3. 编写函数 power2(int n)，它计算并返回 2 的 n 次幂。提供一个交互式的 main() 函数来测试这个函数。
4. 编写一个二进制算法函数 binLog(int n)，它计算并返回一个与 $\log_2 n$ 的整数部分相等的 int 值，其中 n 为正数。这等价于找出小于或者等于 n 的 2 的最大指数。例如，binLog(25)等于 4。至少存在两种简单的迭代方式可以执行这种计算。提供一个交互式的 main()函数来测试这个函数。

19.2.4 复习题

1. 复合语句与简单语句有什么不同?
2. 对于任意给定的 switch 值，如何保证至少有一个分支会被执行?
3. 三种迭代结构各自的优缺点是什么？对于每一种结构，在哪些情况下应该使用其中的一种而不是另外两种?

19.3 逻辑表达式的求值

在 C 和 C++语言中，一旦整个表达式的逻辑值已经确定，对逻辑表达式的求值过程就会停止。这种短路机制可能导致某些操作数不会被求值。当且仅当下面的全部操作数都为 true 时，表达式

$expr_1$ && $expr_2$ && ... && $expr_n$

的值才会被求值为 true。如果其中有一个或者多个操作数为 false，则表达式的值就为 false。对这个表达式的求值是从左到右依次进行的，且只要遇到某个操作数的值为 false，求值过程就会停止(且返回值 false)。

类似地，当且仅当下面的操作数全都为 false 时，表达式

$expr_1$ || $expr_2$ || ... || $expr_n$

的值才会被求值为 false。对这个表达式的求值是从左到右依次进行的，且只要遇到某个操作数的值为 true，求值过程就会停止(且返回值 true)。

程序员经常要开发包含类似如下语句的系统：

```
if( x != 0 &&  y/x < z) {
// do something ...
}
else {
// do something else ...
}
```

如果 x 等于 0，则第二个表达式会导致运行时错误。幸运的是，这不会发生。

逻辑表达式经常同时使用&&和||。需重点记住的是，&&的优先级比||高。换句话说，

$expr_1$ || $expr_2$ && $expr_3$

表示

$expr_1$ || ($expr_2$ && $expr_3$)

而不是

($expr_1$ || $expr_2$) && $expr_3$

19.4　枚举

第 2 章中简要探讨过如果通过定义类来在 C++语言中增加新的类型。C++中增加新类型的另一种途径也值得探讨。

关键字 enum(枚举)被用来向 C++标志符赋整数值。例如，当设计执行位操作的数据结构时，一种方便的做法是为各个位掩码命名。Qt 经常为此使用枚举。枚举的一个恰当例子位于 QFileDialog::Option 中。枚举的主要作用是使代码更具可读性，从而更易于维护。例如：

```
enum {UNKNOWN, JAN, FEB, MAR };
```

定义了四个常量标志符，它们从 0 开始按升序编号。这条语句等价于：

```
enum {UNKNOWN=0, JAN=1, FEB=2, MAR=3};
```

标志符 JAN，FEB，MAR 被称为枚举器(enumerator)。可以将它们初始化成任意的整数值。由于枚举器为 const 项，它们的名称经常以大写字母形式出现(但是，Qt 并没有遵循这个规范)。

```
enum Ages {MANNY = 10, MOE, JACK = 23,
           SCOOTER = JACK + 10};
```

如果第一个枚举器 MANNY 没有被初始化，则它会被自动赋予值 0。由于 MANNY 已经被初始化成 10 而 MOE 没有赋值，所以 MOE 的值为 11。各个枚举器的值可以相同。

为 enum 赋予一个名称，就定义了一个新类型。例如：

```
enum Winter {JAN=1, FEB, MAR, MARCH = MAR };
```

名称 Winter 被称为标记名(tag name)。现在，就可以声明 Winter 类型的变量了。

```
Winter m = JAN;
int i = JAN;    // OK - enum can be implicitly converted to int.
m = i;          // error - explicit cast is required.
m = static_cast<Winter>(i);  // OK
i = m;          // OK
m = 4;          // error
```

在各自的作用域内，标记名和枚举器必须具有不同的标志符。枚举器值可以被隐式地转换成普通的整数类型，但是如果没有进行显式地转型，反过来是不成立的。示例 19.3 演示了枚举的用法，并输出了几个枚举器的值。

示例 19.3　src/enums/enumtst.cpp

```
#include <iostream>
using namespace std;

int main(int, char** ) {
    enum Signal { off, on } sig;                          1
    sig = on;
    enum Answer { no, yes, maybe = -1 };                  2
    Answer ans = no;                                      3
//    enum Neg {no,false} c;                              4
    enum { lazy, hazy, crazy } why;                       5

    int  i, j = on;                                       6
    sig = off;
```

```
    i = ans;
//  ans = s
    ans = static_cast<Answer>(sig);
    ans = (sig ? no : yes);
    sig = static_cast<Signal>(9);
    Signal sig2(sig);
    why = hazy;
    cout << "sig2, ans, i, j, why "
         << sig2 << ans << i << j << why << endl;
    return 0;
}
```


1 在一行中给出了一个新类型，两个新枚举标志符以及一个变量定义。
2 为类型/枚举定义。
3 枚举的一个实例。
4 对标志符的非法重新定义。
5 一个未命名的枚举变量。
6 枚举总是可以被转换成 int 值。
7 枚举类型之间的转换不能隐式地进行。
8 可以用 cast 进行转换。
9 最好不要这样做。
10 添加了一个未命名的枚举器吗?

输出如下所示。

```
src/enums> ./enums
sig2, ans, i, j, why 91011
src/enums>
```

19.5 有符号整型类型与无符号整型类型

这一节讲解有符号(signed)整型类型与无符号(unsigned)整型类型的区别。

任何整型类型的对象 x 的底层二进制表示都是这样的(假设有 n 位存储空间):

$$d_{n-1}d_{n-2}...d_2d_1d_0$$

其中每一个 d 为 0 或者 1。对 x 计算等价的十进制值，与 x 为有符号类型还是无符号类型相关。如果 x 为无符号类型，则其十进制值为

$$d_{n-1}*2^{n-1} + d_{n-2}*2^{n-2} +...+ d_2*2^2 + d_1*2^1 + d_0*2^0$$

因此，能够用无符号整数表示的最大(正)值是:

$$2^n - 1 = 1*2^{n-1} + 1*2^{n-2} +...+ 1*2^2 + 1*2^1 + 1*2^0$$

如果 x 为有符号类型，则其十进制值为

$$d_{n-1}*-(2^{n-1}) + d_{n-2}*2^{n-2} +...+ d_2*2^2 + d_1*2^1 + d_0*2^0$$

因此，能够用有符号整数表示的最大(正)值是:

$$2^{n-1} - 1 = 0*-(2^{n-1}) + 1*2^{n-2} +...+ 1*2^2 + 1*2^1 + 1*2^0$$

这被称为“2 的补码”表示。为了确定负的有符号整数的表示，需进行如下步骤:

1. 计算这个数的“1 的补码”(即将每一个位都变成它的补)。
2. 将上一步中得到的值加 1。

8 位整数举例

假设一个小型系统只使用 8 位来表示一个数。在这个系统中，最大的无符号整数是：

```
11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

但是对于同一个数，如果将其作为有符号整数，则将是：

```
11111111 = -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1
```

19.5.1　练习：有符号整型类型与无符号整型类型

假设有一台计算机，对于示例 19.4、示例 19.5 和示例 19.6，模拟计算机的动作执行给定的代码，并设想输出结果。然后，在计算机上编译并运行这些代码，检验你的设想是否正确。如果存在差异，请给出理由。

1. 示例 19.4　src/types/tctest1.cpp

```
#include <iostream>
using namespace std;
int main() {
  unsigned n1 = 10;
  unsigned n2 = 9;
  char *cp;
  cp = new char[n2 - n1];
  if(cp == 0)
    cout << "That's all!" << endl;
  cout << "bye bye!" << endl;
}
```

2. 示例 19.5　src/types/tctest2.cpp

```
#include <iostream>
using namespace std;

int main() {
  int x(7), y = 11;
  char ch = 'B';
  double z(1.34);
  ch += x;
  cout << ch << endl;
  cout << y + z << endl;
  cout << x + y * z << endl;
  cout << x / y * z << endl;
}
```

3. 示例 19.6　src/types/tctest3.cpp

```
#include <iostream>
using namespace std;

bool test(int x, int y)
{ return x / y; }

int main()
{ int m = 17, n = 18;
  cout << test(m,n) << endl;
```

```
    cout << test(n,m) << endl;
    m += n;
    n /= 5;
    cout << test(m,n) << endl;
}
```


4. 在任何计算或者比较中，如果混用有符号数和无符号数，则通常不是一种好做法。预测示例 19.7 的输出结果。

示例 19.7 src/types/unsigned.cpp

```
#include <iostream>
using namespace std;
int main() {
  unsigned u(500);
  int i(-2);
  if(u > i)
    cout << "u > i" << endl;
  else
    cout << "u <= i" << endl;
  cout << "i - u = " << i - u << endl;
  cout << "i * u = " << i * u << endl;
  cout << "u / i = " << u / i << endl;
  cout << "i + u = " << i + u << endl;
}
```

19.6 标准表达式转换

这一节探讨表达式转换，包括通过提升和降级进行的隐式类型转换，以及通过各种转换机制进行的显式转型。

假设 x 和 y 都为数字型变量。“x op y” 形式的表达式同时具有值和类型。当求值这个表达式时，会使用 x 和 y 的临时副本。如果 x 和 y 具有不同的类型，则在执行操作之前，具有较短类型的那一个变量需要进行转换(变宽)。

对某个数进行的隐式转换如果能保持它的值，则被称为提升(promotion)。

x op y 的自动表达式转换规则

1. 任何 bool, char, signed char, unsigned char, enum, short int 或者 unsigned short int 类型，都可以被提升为 int 类型。这被称为整型提升(integral promotion)。
2. 上一步完成之后，如果表达式为混合类型，则较小类型的操作数会被提升为较大类型，而表达式的值就具有这种较大类型。
3. 类型的层次关系见图 19.1 中的箭头所示。

unsigned 与 long 之间的关系与它们的实现有关。例如，在将 int 实现成具有与 long 相同的字节的系统中，就不能将 unsigned 提升成 long，因此提升过程就会绕开 long 而将 unsigned 提升成 unsigned long。假定具有如下的声明：

```
double d;
int i;
```

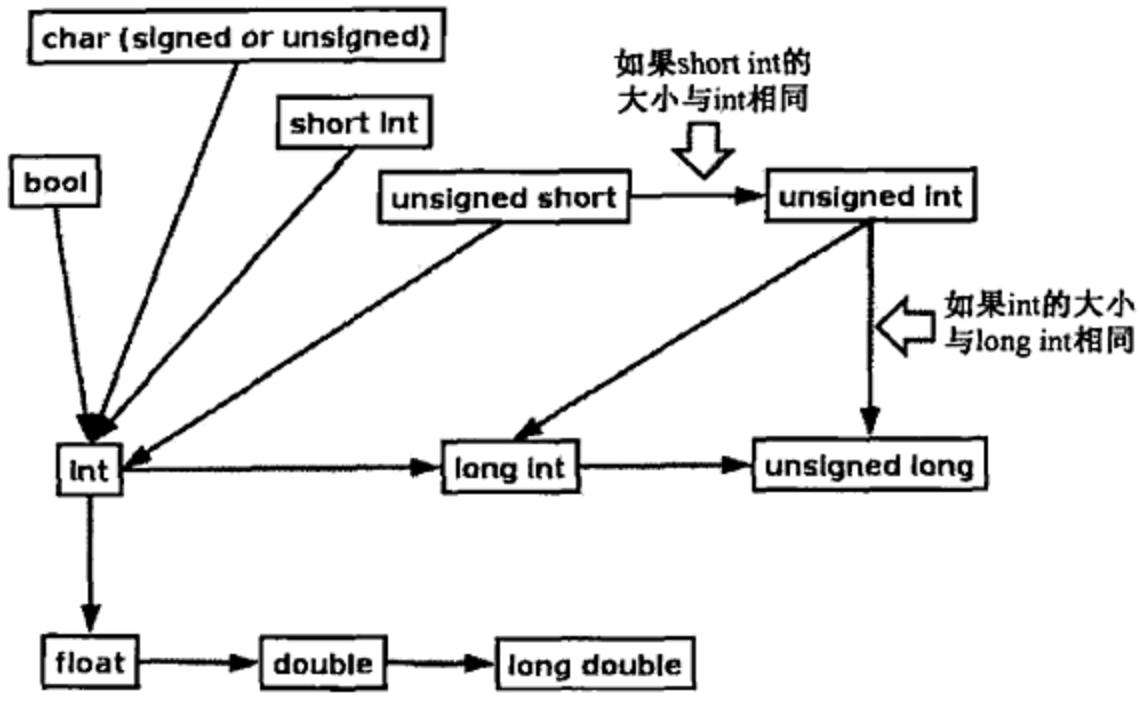


图 19.1　基本类型的层次关系

通常而言，执行语句“d = i;”时进行的提升不会存在什么问题。但是，赋值语句“i = d;”引起的降级会导致信息的丢失。如果编译器允许这种赋值，则 d 的小数部分将被丢弃。示例 19.8 中演示了前面讨论过的几种转换。

示例 19.8　src/types/mixed/mixed-types.cpp

```
#include <iostream>
using namespace std;

int main() {
    int i, j = 88;
    double d = 12314.8723497;
    cout << "initially d = " << d
         << "  and j = " << j << endl;
    cout << "The sum is: " << j + d << endl;
    i = d;
    cout << "after demoting d,  i = " << i << endl;
    d = j;
    cout << "after promoting j,  d = " << d << endl;
}
```

以下是编译并运行的结果。

```
src> g++ mixed-types.cpp
mixed-types.cpp: In function `int main()':
mixed-types.cpp:10: warning: converting to `int' from `double'
src> ./a.out
initially d = 12314.9  and j = 88
The sum is: 12402.9
after demoting d,  i = 12314
after promoting j,  d = 88
src>
```

存在另一种可以根据需要随意执行的隐式转换。能够被求值成指针、整型值或者浮点值的表达式，都可以被转换成布尔值。如果表达式的值为 0，则布尔结果为 false，否则为 true。示例 19.9 演示了这种转换。

示例 19.9 src/types/convert2bool.cpp

```
#include <iostream>
using namespace std;

int main() {
   int j(5);
   int* ip(&j);
   int* kp(0);
   double y(3.4);
   if(y)
      cout << "y looks like true to me!" << endl;
   else
      cout << "y looks like false to me!" << endl;
   cout << "ip looks like " << (ip ? "true" : "false") << endl;
   cout << "kp looks like " << (kp ? "true" : "false") << endl;
   while(--j) {
      cout << j << '\t';
   }
   cout << endl;
}
```

输出如下所示。

```
src> ./a.out
y looks like true to me!
ip looks like true
kp looks like false
4       3       2       1
src>
```

19.6.1 练习：标准表达式转换

假设有如下的声明：

```
double d = 123.456;
int i = 789, j = -1;
uint k = 10;
```

- d + i 的类型和值分别是什么?
- j + k 的类型和值分别是什么?
- 如果执行类似“d = i;”的提升，会发生什么?
- 如果执行类似“i = d;”的降级，会发生什么?

19.7 显式转换

显式转换被称为转型(cast)。有时候，转型是需要的，但是它有过度使用的迹象，是某些错误的主要来源。C++的创立者 Bjarne Stroupstrup 公开表示要尽可能少使用它。

由于 C++起源于 C 语言，所以 C++支持老式的(不安全的)C 风格的转型：

```
(type)expr
```

例如：

```
double d=3.14;
int i = (int)d;
```

C++还支持另一种构造函数风格的转型语法：

```
Type t = Type(arglist)
```

转型会创建一个指定类型的临时值并将其压入程序栈中。如果 *Type* 为类，则创建的是一个临时对象，且它会被合适的转换构造函数初始化；如果 *Type* 为原始类型，则 *Type(arg)* 与 *(Type)arg* 等价。临时值被置于栈中，其存在时间只到其所在的表达式被求值为止。自此以后，它就会被销毁。

例如：

```
double d = 3.14;
Complex c = Complex(d);
```

19.8　用 ANSI C++类型转换进行更安全的类型转换

ANSI C++中增加了四个转型运算符(见表 19.3)，它们具有临时风格的语法，能够更清晰地表达程序员的意图，并可使转型能容易在代码中体现出来。

表 19.3　ANSI 类型转换

`static_cast< type >(expr)`	用于相关类型之间的转换
`const_cast< type > expr`	用于转换掉 const(常量性)或者 volatile(易变性)
`dynamic_cast< type >(expr)`	用于在继承层次中安全地导航
`reinterpret_cast< type >(expr)`	用于不相关类型之间指针的类型转换

19.8.1　static_cast 与 const_cast

只要编译器知道从 *expr* 到 *DestType* 的隐式转换，*static_cast<DestType>(expr)* 就会将值 *expr* 转换成 *DestType* 类型。编译时会进行全部的类型检验。

```
static_cast<char>('A' + 1.0);
static_cast<double>(static_cast<int>(y) + 1);
```

static_cast 运算符会在相关类型之间进行转换，比如从一种指针类型转换成另一种指针类型，从枚举类型转换成整型类型，或者将浮点类型转换成整型类型。这些转换都被很好地定义了，且是可移植的、可逆的。编译器可以对每一个 static_cast 进行某些最小化的类型检验。

static_cast 不能转换掉常量性。如果要创建 *expr* 的一个非常量版本，则必须使用 const_cast<*DestType*>(*expr*)。

这种情况下，只有当存在或者不存在 const/volatile 时，*DestType* 的类型才可以与 *expr* 的类型不同。

对于 int i，static_cast<double>(i)会创建一个 double 类型的临时副本，它具有值 i。这个转型操作不会使变量 i 发生改变。示例 19.10 中包含了这两种转型操作。

示例 19.10　src/ansicast/m2k.cpp

```
// Miles are converted to kilometers.
#include <QTextStream>

QTextStream cin(stdin);
QTextStream cout(stdout);
```

```
QTextStream cerr(stderr);

const double  m2k = 1.609;     // conversion constant
inline double mi2km(int miles) {
    return (miles * m2k);
}

int main() {
    int  miles;
    double kilometers;
    cout << "Enter distance in miles: " << flush;
    cin >> miles ;
    kilometers = mi2km(miles);
    cout << "This is approximately "
         <<  static_cast<int>(kilometers)
         << "km."<< endl;
    cout << "Without the cast, kilometers = "
         << kilometers << endl;
    double* dp = const_cast<double*>(&m2k);
    cout << "m2k: " << m2k << endl;
    cout << "&m2k: " << &m2k << "  dp: " << dp << endl;
    cout << "*dp: " << *dp << endl;
    *dp = 1.892;                                                    1
    cout << "Can we reach this statement? " << endl;
    return 0;
}
```

1 这里的操作意图是什么?

输出如下所示。

```
Enter distance in miles: 23
This is approximately 37km.
Without the cast, kilometers = 37.007
m2k: 1.609
&m2k: 0x8049048  dp: 0x8049048
*dp: 1.609
Segmentation fault
```

根据示例 19.10 可以得出如下一些结论:

- 混合表达式 `miles * m2k` 被隐式地加宽成 `double` 类型。
- 安全的转型 `static_cast<int>(kilometres)` 会将 `double` 值截尾成 `int` 值。
- 转型操作不会改变 `kilometres` 变量的值。
- 对`*dp` 赋值的结果是未定义的。

转换掉常量性

通常而言，`const_cast` 只用在 `const` 引用和指向非 `const` 对象的指针上。用 `const_cast` 改变 `const` 对象的结果是未定义的，因为 `const` 对象可能被保存在只读内存中(操作系统会保护只读内存)。对于 `const int` 的情况，试图通过转换掉常量性来改变它的结果，与编译器的优化技术有关，经常的情况是将其优化成“消失了”(通过预编译值替换操作)。示例 19.11 中给出了可能发生的一些奇怪行为的结果。

示例 19.11　src/casts/constcast1.cpp

```
#include <iostream>
using namespace std;

int main() {
    const int N = 22;
    int* pN = const_cast<int*>(&N);
    *pN = 33;
    cout << N << '\t' << &N << endl;
    cout << *pN << '\t' << pN << endl;
}
```

输出如下所示。

```
22      0xbf91cfa0
33      0xbf91cfa0
```

前面的输出是用 gcc 4.4.5 获得的，它可能与你的系统上的输出有所不同，因为程序的行为是未定义的。

这个示例中使用了 const_cast 来获得一个 const int 的常规指针。由于 const int 位于栈存储类中，试图改变它的内存不会导致段错误。编译器不能“优化掉” int，const_cast 会告诉它不要这样做。

19.8.1.1　练习：static_cast 与 const_cast

1. 示例 19.11 中，将语句

   ```
   const int N = 22;
   ```

   移到

   ```
   int main() {
   ```

   的上面或者下面，输出有什么不同？请解释。

2. a. 预测示例 19.12 的输出。

   示例 19.12　src/casts/constcast2.cpp

   ```
   #include <iostream>

   void f2(int& n) {
       ++n;
   }

   void f1(const int& n, int m) {
       if (n < m)
           f2(const_cast<int&>(n));
   }

   int main() {
       using namespace std;
       int num1(10), num2(20);
       f1(num1, num2);
       cout << num1 << endl;
   }
   ```

   b. 将 f1() 中调用 f2() 的语句里移除 const_cast，再次预测输出。

19.8.2 reinterpret_cast

reinterpret_cast 被用于与表示相关或者与系统相关的转型中。这类例子包括不相关类型之间的转换(例如，int 到指针)，或者不相关指针类型之间的转换(例如，int*到double*)它不能转换掉常量性。

reinterpret_cast 是危险的，且通常是不可移植的，所以应避免使用。

考虑下面的情形。

```
Spam spam;
Egg* eggP;
eggP = reinterpret_cast<Egg*>(&spam);
eggP->scramble();
```

reinterpret_cast 具有一些 spam 且提供一个 Egg 类型的指针，而没有考虑类型兼容性。eggP 将会把猪肉(spam)重新解释成鸡蛋。

转型的真正用途是什么

有时，C 函数会返回一个 void*指针，指向开发人员已知的某种类型。这种情况下，有必要用类型转换将 void*转换成实际类型。如果确定它指向的是一个 Egg，则合适的转型操作是 reinterpret_cast<Egg*>。对于这种转型，编译器不会进行类型检验，运行时系统也不会。

例如，示例 12.15 的 QMetaType 中，就对 void*使用了 reinterpret_cast。

19.8.3 为什么不使用 C 风格的转型

C 风格的转型操作已经被弃用，不应该再使用。考虑下面的语句：

```
Apple apple;
Orange* orangeP;
// other processing steps ...
orangeP = (Orange*) &apple;
orangeP->peel();
```

这些语句存在的问题是：从这些代码中无法看出开发人员是否知道苹果(Apple)是否与橘子(Orange)“兼容”。这里看不出它是合适的类型转换还是不可移植的指针转换。尽管它们可能都有 peel()函数，但是你能像剥橘子皮一样给苹果剥皮吗?

由这种转型导致的错误，比较难以理解和改正。如果有必要使用依赖于系统的转型操作，则推荐的做法是使用 reinterpret_cast 而不是 C 风格的转型，这样当出现问题时，会更容易从源代码中找出原因。

19.8.4 关于 explicit 转换构造函数的更多说明

2.12 节中讲解过 explicit 转换构造函数，下面是使用它的另一个例子。

假设出于某个理由，需要编写一个自己的 String 类[①]，则可能会编写出带有如下几个构造函数的一个类定义：

① 来自 Sgt. Major 的声音：“QString 还不够吗?”

```
class String {
  String();                  // Creates an empty string of length 0
  String(const char* str); // Converts a char array to a String
  explicit String(int n);  // Creates length n string, filled
                             // with spaces
  ... other member functions - but not constructors ...
};
```

描述 String 对象 str1 和 str2 会得到如下的客户代码行：

```
...
String str1, str2;          // construct two empty strings
str1 = "A";
str2 = 'A';
...
```

如果让这两个 String 对象都只包含一个字符'A'，使 str1 和 str2 非常相似，这样做没有什么不合法的。但是，根据实现转换构造函数方式的不同(即如何处理终止的 null 字符)，str1 可能会包含字符'A'和 null 字符。对 str2 的赋值可能是另外一种完全不同的情况。如果移除 explicit 关键字，则 str2 将成为一个长度为 65 的 String 对象，其值为空。这是因为，字符'A'将被提升为 int 值 65('A'的 ASCII 码值)，这样就会隐式地调用第三个构造函数。

为了避免这种误解，一种好的做法是将第三个构造函数指定成 explicit。这样做会导致 str2 赋值产生一个编译错误，即第三个构造函数不能被隐式地调用，只能显式地调用。因此，String 类就无法处理这一行代码，编译器会报告错误。

19.9　重载特殊的运算符

本节探讨如何将运算符重载作为“句法糖果”的一种形式，这样就能够将复杂对象当成简单类型，即具有方括号的数组和带有圆括号的函数。

19.9.1　转换运算符

2.12 节中讲过，利用转换构造函数，可以在编译器的类型系统中实现类型转换功能。这样就使得从一种已有类型转换成一种新类型成为可能。为了定义反方向的转换，需要使用转换运算符。5.3 节中讲解过如何定义非简单类型上的运算符的行为。转换运算符使用与类型转换重载相同的机制。

转换运算符看起来与成员函数大不相同。它没有返回类型(甚至没有 void)，也没有任何参数。当使用转换运算符时，会返回一个具有它所命名的类型的对象：在函数体中指定的主对象的转换对象。转换运算符通常是隐式使用的，这样在需要时可以进行自动转换(例如确定函数调用时，见 5.1 节)。

当运行示例 19.13 中的程序时，可以看到它使用了从 Fraction 到 double 和从 Fraction 到 QString 的用户定义转换。

示例 19.13　src/operators/fraction/fraction-operators.cpp

```
#include <QString>
#include <QTextStream>

QTextStream cout(stdout);
```

```
class Fraction {
public:
    Fraction(int n, int d = 1)                                          1
        : m_Numerator(n), m_Denominator(d) {}

    operator double() const {                                           2
        return (double) m_Numerator / m_Denominator;
    }
    operator QString () const {
        return  QString("%1/%2").arg(m_Numerator).arg(m_Denominator);
    }
private:
    int m_Numerator, m_Denominator;
};

QTextStream& operator<< (QTextStream& os, const Fraction& f) {
    os << static_cast<QString> (f);                                     3
    return os;
}

int main() {

    Fraction frac(1,3);
    Fraction frac2(4);                                                  4
    double d = frac;                                                    5
    QString fs = frac;                                                  6
    cout << "fs= " << fs << "  d=" << d << endl;
    cout << frac << endl;                                               7
    cout << frac2 << endl;
    return 0;
}
```

1 转换构造函数。

2 转换运算符。

3 显式的转型操作调用转换运算符。

4 转换构造函数调用。

5 调用转换运算符。

6 另一个转换运算符调用。

7 直接调用 operator <<()。

以下是程序的输出。

```
src/operators/fraction> ./fraction
fs= 1/3  d=0.333333
1/3
4/1
src/operators/fraction>
```

19.9.2 下标运算符 operator[]

许多列表和与数组相似的类都提供与数组一致的接口，但还具有更多的功能。下标运算符 operator[]被限制成只使用一个参数。它通常被用来提供对容器中某个元素的访问，如

示例 5.16 所示。示例 19.14 定义的一个容器类使用了下标运算符 operator[]()，它能够防止数组下标越界。

示例 19.14　src/operators/vect1/vect1.h

```
[ . . . . ]
class Vect {
public:
    explicit Vect(int n = 10);
    ~Vect() {
        delete []m_P;
    }
    int& operator[](int i) {                                    1
        assert (i >= 0 && i < m_Size);
        return m_P[i];
    }
    int  ub() const {
        return (m_Size - 1);
    }                                                           2
private:
    int*  m_P;
    int   m_Size;
};

Vect::Vect(int n) : m_Size(n) {
    assert(n > 0);
    m_P = new int[m_Size];
}
[ . . . . ]
```

1 访问 m_P[i]。

2 上界。

示例 19.15 中的客户代码定义了一个 Vect 对象的数组，它提供一种与矩阵类似的结构，其中一个是固定的一维数组，另一个根据 Vect 实现的不同可能是变量或者稀疏矩阵。main() 中还为数字输出提供了一个简单的右对齐方法。

示例 19.15　src/operators/vect1/vect1test.cpp

```
#include "vect1.h"

int main() {
    Vect a(60), b[20];

    b[1][5] = 7;
    cout << " 1 element 5 = "<< b[1][5] << endl;
    for (int i = 0; i <= a.ub(); ++i)
        a[i] = 2 * i + 1;
    for (int i = a.ub(); i >= 0; --i)
        cout << ((a[i] < 100) ? " " : "" )
        << ((a[i] < 10) ? " " : "" )
        << a[i]
        << ((i % 10) ? "  " : "\n");
```

```
        cout << endl;
        cout << "Now try to access an out-of-range index"
        << endl;
        cout << a[62] << endl;
    }
```

以下是程序的输出。

```
src/operators/vect1> ./vect1
 1 element 5 = 7
119  117  115  113  111  109  107  105  103  101
 99   97   95   93   91   89   87   85   83   81
 79   77   75   73   71   69   67   65   63   61
 59   57   55   53   51   49   47   45   43   41
 39   37   35   33   31   29   27   25   23   21
 19   17   15   13   11    9    7    5    3    1
Now try to access an out-of-range index
vect1: vect1.h:16: int& Vect::operator[](int):
Assertion `i >= 0 && i < m_Size' failed.
Aborted
src/operators/vect1>
```

19.9.3 函数调用运算符

可以将函数调用运算符 operator()重载成非静态成员函数。它经常被用来提供可调用的接口、迭代器或者一个具有多个索引下标的运算符。它比 operator[]更为灵活，因为可以用不同的签名进行重载。示例 19.16 中，就为 Matrix 类提供了一个多下标运算符。

示例 19.16 src/operators/matrix/matrix.h

```
[ . . . . ]
class Matrix {
public:
    Matrix(int rows, int cols);                                    1
    Matrix(const Matrix& mat);                                     2
    ~Matrix();
    double& operator()(int i, int j);
    double operator()(int i, int j) const;
    // Some useful Matrix operations
    Matrix& operator=(const Matrix& mat);                          3
    Matrix operator+(const Matrix& mat) const;                     4
    Matrix operator*(const Matrix& mat) const;                     5
    bool operator==(const Matrix& mat) const;
    int getRows() const;
    int getCols() const;
    QString toString() const;
private:
    int m_Rows, m_Cols;
    double  **m_NumArray;
    //Some refactoring utility functions
    void sweepClean();                                             6
    void clone(const Matrix& mat);                                 7
```

```
    double rcprod(int row, const Matrix& mat, int col) const;
                                                    /* Computes dot product of the
                                                    host's row with  mat's col. */
};
[ . . . . ]
```


1 分配单元格并清空。
2 复制构造函数；复制 mat。
3 删除主矩阵内容；复制 mat。
4 矩阵加法。
5 矩阵乘法。
6 清除主矩阵全部单元格的值。
7 用新内存位置保存主矩阵的副本。

示例 19.17 中实现了所需要的多下标运算符的两个版本，一个用于获取 Matrix 的值，另一个用于设置它的值。注意，这个实现中使用了(不受保护的)数组下标用法，它与 C/C++ 数组中的下标用法相同。这一段用于正确地处理底层数组的警告代码，有助于产生安全而可靠的公共接口，它对 Matrix 类使用(具有范围检查的)函数调用下标标注法。

示例 19.17　src/operators/matrix/matrix.cpp

```
[ . . . . ]

double Matrix::operator()(int r, int c) const {
   assert (r >= 0 && r < m_Rows && c >= 0 && c < m_Cols);
   return m_NumArray[r][c];
}

double& Matrix::operator()(int r, int c) {
   assert (r >= 0 && r < m_Rows && c >= 0 && c < m_Cols);
   return m_NumArray[r][c];
}
```

示例 19.18 中给出的构造函数实现，需要知道希望为这个 Matrix 使用多少行和多少列。

示例 19.18　src/operators/matrix/matrix.cpp

```
[ . . . . ]

Matrix:: Matrix(int rows, int cols):m_Rows(rows), m_Cols(cols) {
    m_NumArray = new double*[rows];
    for (int r = 0; r < rows; ++r) {
        m_NumArray[r]  = new double[cols];
        for(int c = 0; c < cols; ++c)
           m_NumArray[r][c]  = 0;
    }
}
```

构造函数在 Matrix 的每一个单元格中为 double 值分配了空间，所以示例 19.19 中实现的析构函数必须删除每一个单元格。如果需要实现其他的成员函数，则需要去除这些删除代码。

示例 19.19 src/operators/matrix/matrix.cpp

```
[ . . . . ]

void Matrix::sweepClean() {
   for (int r = 0; r < m_Rows; ++r)
      delete[] m_NumArray[r] ;
   delete[] m_NumArray;
}

Matrix::~Matrix() {
   sweepClean();
}
```

19.9.3.1 练习：函数调用运算符

完成 Matrix 类的成员函数的实现，并编写全面测试这个类的客户代码。

19.10 运行时类型识别

这一节讲解 dynamic_cast 和 typeid，它们是能够执行运行时类型识别(RTTI)的两个运算符。

基类指针到派生类指针的转换，被称为向下转型(downcasting)，因为从基类到派生类的转型被认为是沿类层次“向下”移动。

当操作这种类型层次时，有时必须将指针“向下转型”为一种更具体的类型。如果没有向下转型，则只能使用指针类型(基类)的接口。用到向下转型的一种常见情形是接受基类指针的函数内部。

RTTI 使得程序员能够安全地将指针和引用从基类型转换成派生类型的对象。

dynamic_cast<*D**>(ptr)具有两个操作数：指针类型 *D**和多态类型 *B**的指针 ptr。如果 *D* 是 *B* 的基类(或者 *B* 与 *D* 具有相同的类)，则 dynamic_cast<D*>(ptr)是一个向上转型(或者根本不是一个转型)，它等价于 static_ cast<D*>(ptr)。但是，如果 ptr 具有类型为 *D* 的对象的地址，其中 *D* 派生自 *B*，则这个运算符返回一个类型为 *D**的向下转型指针，指向同一个对象。如果无法使用转型操作，则会返回一个空指针。

dynamic_cast 执行运行时检查，以判断指针/引用转换是否是有效的。例如，假设是在处理一个 QWidget*的集合。示例 19.20 给出了对 QWidget 集合的操作。进一步假设只对按钮和 spinbox 进行操作，而不理睬其他的窗件。

示例 19.20 src/rtti/dynamic_cast.cpp

```
[ . . . . ]
int processWidget(QWidget* wid) {

    if (wid->inherits("QAbstractSpinBox")) {                    1
        QAbstractSpinBox* qasbp =
            static_cast <QAbstractSpinBox*> (wid);
        qasbp->setAlignment(Qt::AlignHCenter);
    }
    else {
        QAbstractButton* buttonPtr =
```

```
            dynamic_cast<QAbstractButton*>(wid);
        if (buttonPtr) {
            buttonPtr->click();
            qDebug() << QString("I clicked on the %1 button:")
                .arg(buttonPtr->text());
        }
        return 1;
    }
    return 0;
}
[ . . . . ]
    QVector<QWidget*> widvect;

    widvect.append(new QPushButton("Ok"));
    widvect.append(new QCheckBox("Checked"));
    widvect.append(new QComboBox());
    widvect.append(new QMenuBar());
    widvect.append(new QCheckBox("With Fries"));
    widvect.append(new QPushButton("Nooo!!!!"));
    widvect.append(new QDateTimeEdit());
    widvect.append(new QDoubleSpinBox());
    foreach (QWidget* widpointer, widvect) {
        processWidget(widpointer);
    }
   return 0;
}
```

1 只针对由元对象编译器(moc)处理的 QObject。

2 如果非空，则为有效的 QAbstractButton。

qobject_cast(参见 12.2 节)要比 dynamic_cast 运行得快，但它只能用于 QObject 派生的类型。

就运行时成本而言，dynamic_cast 要昂贵得多，也许是 static_cast 的 10～50 倍。这两种转换是不可互换的操作，它们用在不同的环境中。

19.10.1　typeid 运算符

RTTI 中另一种运算符是 typeid()，它返回关于实参的类型信息。例如：

```
void f(Person& pRef) {
   if(typeid(pRef) == typeid(Student) {
     // pRef is actually a reference to a Student object.
     // Proceed with Student-specific processing.
   }
   else {
     // Nope! The object referred to by pRef is not a Student.
     // Proceed to do whatever alternative stuff is required.
   }
}
```

typeid()返回的 type_info 对象与实参的类型相对应。

如果两个对象具有相同的类型，则它们的 type_info 对象应当等价。可以将 typeid()运算符用于多态类型或者非多态类型，也可以将它用于基本类型和定制类型。此外，typeid()的实参可以是类型名称或者对象名称。

以下是 g++ 4.4 中 type_info 类的一种实现方式：

```
class type_info {
 private:
    type_info(const type_info& );
    // cannot be copied by users
    type_info& operator=(const type_info&);
    // implementation-dependent representation
 protected:
    explicit type_info(const char *name);
 public:
    virtual ~type_info();
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
    // returns a pointer to the name of the type
    // [...]
}
```

19.11 成员选择运算符

有两种形式的成员选择运算符：

- *pointer->memberName*
- *object->memberName*

它们貌似相同，但存在两个重要的不同点：

- 第一个是二元的，第二个是一元的。
- 第一个是全局的且不可重载，第二个是可重载的成员函数。

当用于类定义时，一元 operator->()应返回一个指针，指向其成员名称为 *memberName* 的一个对象。

实现了 operator->的对象，通常被称为智能指针。之所以这样称呼，是因为在程序中它会比常规的指针更“聪明”。例如，QSharedPointer 就是一个维护 QObject 引用计数指针的智能指针。如果删除了最后一个共享的指针，则共享的 QObject 也会被删除。

智能指针的例子包括：

- STL 风格的迭代器。
- QPointer, QSharedDataPointer, QSharedPointer, QWeakPointer, QScopedPointer, QExplicitlySharedDataPointer。
- auto_ptr, STL 智能指针。

这些智能指针都是模板类。

示例 19.21 中给出了 QSharedPointer 的部分定义。

示例 19.21　src/pointers/autoptr/qsharedpointer.h

```
template <class T>
class QSharedPointer {
public:
    QSharedPointer();
    explicit QSharedPointer(T* ptr);
    T& operator*() const;
    T* operator->() const;

    bool isNull() const;
    operator bool() const;
    bool operator!() const;
    // [ ... ]
};
```

如果指针<*T*>的对象被销毁时，它自动设置成 0，则称指针<*T*>为 *T* 类型的 QObject 的监控指针(guarded pointer)。QSharedPointer，QPointer 和 QWeakPointer 同样也提供了这种功能性。以下的代码段给出了智能指针是如何被当作常规指针使用的：

```
[. . .]
QPointer<QIntValidator> val = new QIntValidator(someParent);
val->setRange(20, 60);
[. . .]
```

第二行代码中，val->()返回一个指向新分配的对象的指针，然后被用来访问 setRange()成员函数。

19.12　练习：类型与表达式

1. 假设要求你使用一个编写得很不规范的类库，而且没有办法改进这些类的实现方式(但愿这不是真的)。示例 19.22 中给出了这样一个编写得很差的类样本，还给出了一个使用它的小程序。这个程序中创建了几个对象，并将它们作为 const 引用(隐含的意思是不会改变它们)传递给了一个函数，然后输出算术结果。遗憾的是，由于类编写得很差，运行过程中出现了一些错误。

示例 19.22　src/const/cast/const.cc

```
#include <iostream>
using namespace std;

class Snafu {
public:
    Snafu(int x) : mData(x) {}
    void showSum(Snafu & other) const {
        cout << mData + other.mData << endl;
    }

private:
    int mData;
```

```
};

void foo(const Snafu & myObject1,
         const Snafu & myObject2) {
    // [ . . . ]
    myObject1.showSum(myObject2);
}

int main() {

    Snafu myObject1(12345);
    Snafu myObject2(54321);

    foo(myObject1, myObject2);
}
```

回答下列问题。

a. 这些错误是什么?

b. 对类做哪些修改可改正这些错误?

不巧的是,无法对类进行修改。在不改变类定义的前提下,至少给出修复这个程序的两种方法。哪一个方法更好?为什么?

c. 示例 19.23 是一个尚未完成的类定义,它对每一个实例计算每一个对象的数据被输出的次数。请查看并改正这个程序。

示例 19.23 src/const/cast/const2.cc

```
#include <iostream>
using namespace std;

class Quux {
public:
    Quux(int initializer) :
        mData(initializer), printcounter(0) {}
    void print() const;
    void showprintcounter() const {
      cout << printcounter << endl;
    }

private:
    int mData;
    int printcounter;
};

void Quux::print() const {
    cout << mData << endl;
}

int main() {
    Quux a(45);
    a.print();
    a.print();
```

```
        a.print();
        a.showprintcounter();
        const Quux b(246);
        b.print();
        b.print();
        b.showprintcounter();
        return 0;
    }
```


19.13　复习题

1. 语句与表达式有什么不同?
2. 重载运算符与函数有什么不同?
3. 在 C++中引入新类型有哪些方法?
4. 对于数字值，最合适的转型运算符是什么?
5. 向 int 变量赋予一个 double 值时，会发生什么?
6. 对于通过多态层次进行的向下转型操作，最合适的转型运算符是什么?
7. 为什么推荐使用 ANSI 转型操作而不是 C 风格的转型操作?
8. 在什么情况下适合使用 reinterpret_cast?

第 20 章　作用域与存储类

标志符具有作用域，对象具有存储类，而变量二者都具有。本章将探讨声明与定义的差异，并将讲解如果确定标志符的作用域以及对象的存储类。

20.1　声明与定义

在使用之前，任何标志符都必须被声明或者定义。声明一个名称，就是告诉编译器与这个名称相关联的类型是什么。

定义一个对象或者变量，就是分配空间以及(可能的)赋初值。例如：

```
double x, y, z;
char* p;
int i = 0;
QString message("Hello");
```

定义一个函数，表示在一个 C++语句块中完整地描述它的行为。例如：

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

定义一个类，表示以函数和数据成员声明的顺序指定类的结构，见示例 20.1。此外，类定义还告知编译器它的对象要求多少内存空间。

示例 20.1　src/early-examples/decldef/point.h

```
class Point {                                                    1
 public:
    Point(int x, int y, int z);                                  2
    int distance(Point other);                                   3
    double norm() const {                                        4
        return distance(Point(0,0,0));
    }
 private:
    int m_Xcoord, m_Ycoord, m_Zcoord;                            5
};
```

1 类首部。

2 构造函数声明。

3 函数声明。

4 声明与定义。

5 数据成员声明。

示例 20.2 中包含的一些声明不是定义。

示例 20.2　src/early-examples/decldef/point.cpp

```
extern int step;                                                 1
class Map;                                                       2
int max(int a, int b);                                           3
```

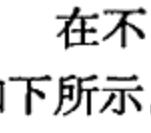
1 对象(变量)声明。

2 (前置)类声明。

3 全局(非成员)函数声明。

每一个不是为定义的声明，都会向编译器传递一个隐含的承诺(链接器会强制这样)：所声明的名称会在程序中某个合适的位置定义。

每一个定义就是一个声明。某个作用域内的任何名称的定义只能出现一次，但是可以存在多个声明。



C++中，变量初始化似乎是“可选的”，但是不管是否指定了初始化，对变量的初始化工作总是会发生。任何具有如下形式的语句：

```
TypeT var;
```

会导致变量 var 的默认初始化。默认初始化表示值是由编译器提供的。对于简单类型(例如，int, double, char)，默认值是未定义的。即赋予 var 的默认值可以是 0，也可能是某个恰好位于内存中的随机垃圾值。对于类对象，其默认值由默认构造函数(如果存在的话)确定，否则编译器会报告错误。所以，强烈建议为所有的变量定义提供精心挑选的初始值，否则，可能会出现无效的结果或者难于定位的、奇怪的运行时错误。有必要重申一下这个原则：所有的对象和变量都应当在创建时(或随后)就正确地初始化。

20.2　标志符的作用域

每一个标志符的作用域都由其所声明的位置确定。程序中标志符的作用域，是它能够被识别并使用的程序区域。如果在标志符的作用域之外使用它的名称，就会导致错误。

在不同的作用域内可以声明并使用同一个名称。同一个名称在不同作用域下的使用规则如下所示。

1. 来自于最近的作用域的名称被首先使用。
2. 如果在最近的作用域内没有定义这个名称，则会使用最近的包含作用域(enclosing scope)中的同一个名称。
3. 如果任何包含作用域内都没有定义这个名称，则编译器会报告错误。

下面探讨 C++中的六种不同的作用域：

1. 块作用域(只用于语句块中)
2. 函数作用域(函数的整个范围)[①]
3. 类作用域(类的整个范围，包括它的成员函数定义)
4. 命名空间作用域(具有类作用域特性的扩展块作用域)
5. 文件作用域(从某个源代码文件声明的开始处到结束处)
6. 全局作用域(与文件作用域类似，但可以扩展至多个源代码文件)

① 只有标记才具有函数作用域。

20.2.1 标志符的默认作用域

下面逐一分析这六种作用域并给出一些示例。

块作用域

在一对大括号(不包括 namespace 语句块)里或者在一个函数参数表中声明的标志符，具有块作用域。块作用域的范围从它的声明开始到右大括号结束。

函数作用域

标记(label)是一种后面跟有一个冒号的标志符。C/C++函数中的标记具有自己的作用域。在整个函数定义中，标记在其声明之前和之后都是可识别的。对于老式的、(正常情况下)应避免使用的、需要标记的 goto 语句，C 和 C++几乎不支持。函数作用域的独特之处在于：标记(声明它的位置)能够出现在它所指向的第一条语句(例如，goto 语句)的后面。示例 20.3 中给出的例子，使用了强烈建议丢弃的 goto 语句以及与之相关的标记。

示例 20.3 src/goto/goto.cpp

```
[ . . . . ]
int look() {
    int i=0;
    for (i=0; i<10; ++i) {
        if (i == rand() % 20)
            goto found;                    1
    }
    return -1;
found:                                     2
    return i;
}
[ . . . . ]
```

1 最好使用 break 或者 continue。

2 goto 充当了标记的前置声明。

关于标记的一种相关的但是危险性小一些的用法，是将其用在 switch 语句块中。switch 语句是一种具有计算功能的 goto 语句，且由于它的动作都位于一个语句块之内，所以不会导致像 goto 语句那样的有效性问题。示例 20.4 是使用 switch 语句的一个例子。

示例 20.4 src/switch/switchdemo.cpp

```
#include <QTextStream>
#include "usermanager.h"

QTextStream cout(stdout);
QTextStream cin(stdin);
enum  Choices {LOAD = 1, ADD, CHANGE, CHECK, SAVE, LIST, QUIT};    1

// Function Prototypes
void addUsers(UserManager&);
void changePassword(UserManager&);
Choices menu();
```

```
//etc.

int main() {
  // some code omitted
  while (1) {
    switch (menu()) {
    case LOAD:
      cout << "Reading from file ...\n"
           << um.loadList() << " loaded to list"
           << endl;
      break;
    case ADD:
      cout << "Adding users to the list ..." << endl;
      addUsers(um);
      break;
case SAVE:
  cout << "Saving changes ...\n"
       << um.saveList() << " users in file" << endl;
  break;
case CHANGE:
  cout << "Changing password ..." << endl;
  changePassword(um);
  break;
case CHECK:
  cout << "Checking a userid/pw combo ..." << endl;
  checkUser(um);
  break;
case LIST:
  cout << "Listing users and passwords ...\n";
  um.listUsers();
  break;
case QUIT:
  cout << "Exiting the program ..." << endl;
  return 0;
default:
  cout << "Invalid choice! " << endl;
    }
  }
}
```

1 枚举见第 19 章的探讨。

2 menu()获得来自于用户的一个值。

正如 19.2.2 节中所讲，case 标记(default 标记除外)与常规的标记不同，因为它要求是整型常量。case 标记的作用域为整个 switch 语句，所以可以称其具有 switch 作用域。

有时，标记被用来解决各种兼容性问题。例如，标记被用来阻止 C++编译器对某些类定义中的“signals:”和“slots:”声明给出错误信息(参见 8.3 节)。

注意

尽管 goto 是 C++语言的一部分，但绝对不要使用它。

命名空间作用域

在某个命名空间里声明的标志符，具有命名空间作用域。这种标志符可以用在声明之后、命名空间定义里面的任何地方。命名空间定义是开放的，可以扩展。同一个命名空间的后续定义，只会向其添加项目，如示例 20.5 中所示。如果在命名空间里面试图重新定义某个项目，则会导致编译错误。

示例 20.5 src/namespace/openspace/opendemo.txt

```
//File:  a.h"
#ifndef _A_H_
#define _A_H_

#include <iostream>
namespace A {
  using namespace std;
  void f() { cout << "f from A\n"; }
  void g() { cout << "g from A\n"; }
}
#endif

//File:  new-a.h
#ifndef NEW_A_H_
#define NEW_A_H_
#include <iostream>

namespace A {
  //void k() { h(); }                                          1
  //void g() { cout << "Redefine g()/n"; }                     2
  void h() {
    cout << "h from newA\n";
    g();
  }
}
#endif

File: opendemo.cpp
#include "a.h"
#include "new-a.h"

int main() {
  using namespace A;
  f();
  h();
}

/*Run

openspace> ./a.out
f from A
h from newA
g from A
openspace>

*/
```

1 错误!

2 错误!

类作用域

在某个类定义里声明的标志符，具有类作用域。类作用域的范围是类定义里的任何地方[①]，或者成员函数体里的任何地方[②]。

文件作用域

如果某个标志符的声明没有位于大括号对中，且被声明为静态的，则它就具有文件作用域。文件作用域的范围从它的声明开始到文件结束。关键字 static 使其他源文件看不到这个标志符，从而使其作用域被限制在声明它的文件内。文件作用域变量不能被声明成 extern 类型，其他文件也无法访问它。

由于文件作用域变量不会被输出，所以它不会扩展到(扰乱)全局命名空间。这种变量常用于 C 程序中，因为 C 语言中没有提供隐藏特性的实现方式，比如针对类成员的 private 声明。

注意

为了与 C 语言具有后向兼容性，C++中支持文件作用域，但是只要有可能，就应使用命名空间或者静态类成员。

全局作用域

如果某个标志符的声明没有位于大括号对中，且没有被声明为静态的，则它就具有全局作用域。这种标志符的作用域从声明处开始，到源文件的结尾处结束，但是通过 extern 关键字，它可以扩展至其他的源文件。可以利用 extern 声明来访问在其他源文件中定义的全局化标志符。

C++中，对变量采用全局作用域并不是必需的。通常而言，只有类和命名空间才应在全局作用域下定义。如果确实需要“全局”变量，则可以通过使用 public static 类成员或者命名空间成员来获得类似的功能。由于编译器一次只能处理一个源文件，所以只有链接器(或者模板编译器)才能够严格区分全局作用域与文件作用域，如示例 20.6 所示。

示例 20.6　全局作用域与文件作用域的比较

```
In file 1:

int g1;         // global
int g2;         // global
static int g3; // keyword static limits g3 to file scope
(etc.)


In file 2:

int g1;            // linker error!
```

① 包括位于所引用成员声明之前的 inline 函数定义。

② 要记住的是，非静态成员的作用域不包括静态成员函数的函数体。

```
extern int g2;     // OK, share variable space
static int g3;     // okay, 2 different variable spaces
(etc.)
```

位于命名空间中的标志符，可以通过使用作用域解析运算符“NamespaceName::”而成为全局标志符。也可以不用作用域解析运算符而用关键字 using，即可使命名空间中的标志符在其他作用域内可用。

命名空间变量和静态类成员具有静态存储类，能够在全局范围内被访问。它们就好像全局变量，但没有扩大(扰乱)全局命名空间。更多细节，请参见 20.4 节。

20.2.2 文件作用域，块作用域与“operator::”的比较

前面已经看到并使用过作用域解析运算符，它用来通过“ClassName::”扩展类的作用域或者访问它的成员。类似的语法“NamespaceName::”被用来访问命名空间中的每一个符号。C++还具有(一元)文件作用域解析运算符“::”，它提供从所包含的作用域内访问全局对象、命名空间对象或者文件作用域对象的能力。下面的练习将各种作用域用于这个运算符。

20.2.2.1 练习：文件作用域，块作用域与“operator::”的比较

1. 确定示例 20.7 中各个变量的作用域。
2. 假设你是计算机，预测这个程序的输出结果。

示例 20.7 src/early-examples/scopex.cpp

```
#include <iostream>
using namespace std;

long x = 17;
float y = 7.3;                                          1
static int z = 11;                                      2

class Thing {
    int m_Num;                                          3
public:
    static int s_Count;                                 4
    Thing(int n = 0) : m_Num(n) {++s_Count;}
    ~Thing() {--s_Count;}
    int getNum() { return m_Num; }
};

int Thing::s_Count = 0;
Thing t(11);
int fn(char c, int x) {                                 5
    int z = 5;                                          6
    double y = 6.933;
    { char y;                                           7
    Thing z(4);                                         8
    y = c + 3;
    ::y += 0.3;                                         9
    cout << y << endl;                                  10
    }
```

```
    cout << Thing::s_Count
         << endl;                                    11
    y /= 3.0;                                        12
    ::z++;                                           13
    cout << y << endl;
    return x + z;
}

int main() {
    int x, y = 10;
    char ch = 'B';                                   14
    x = fn(ch, y);
    cout << x << endl;
    cout << ::y << endl;                             15
    cout << ::x / 2 << endl;
    cout << ::z << endl;
}
```

1 作用域：_____

2 作用域：_____

3 作用域：_____

4 作用域：_____

5 作用域：_____

6 作用域：_____

7 作用域：_____

8 作用域：_____

9 作用域：_____

10 作用域：_____

11 作用域：_____

12 作用域：_____

13 作用域：_____

14 作用域：_____

15 作用域：_____

20.3　存储类

只要创建了对象，就会为它分配如下四种可能的位置空间，这些位置被称为存储类。

注意

作用域指的是标志符可以访问的代码区域。存储类指的是内存中的某个位置。

1. 静态区域——全局变量、静态局部变量、静态数据成员被保存在静态存储区域中。静态对象的生命周期从加载它的对象模块开始，到程序终止时结束。
   静态区域常用于指针、简单类型和字符串常量，较少用于复杂对象。

2. 程序栈(自动存储类 auto[①])——函数参数、局部变量、返回值以及其他的临时对象都保存于栈中。栈存储是在执行对象定义时自动分配的。位于这个存储类中的对象，对函数或者语句块而言是局部的[②]。
   对于局部(块作用域)变量，其生命周期由包围所执行代码的括号确定。
3. 堆存储或者自由存储(动态存储)——通过 new 关键字创建的对象。堆对象的生命周期完全由 new 和 delete 关键字的使用确定。
   通常而言，堆对象的分配和释放应当小心地置于所封装的类里面。
4. 另一个存储类是 C 语言遗留下来的，被称为寄存器(register)。它是自动存储类的一种特殊形式，由数量相对较小的、最快速的可用内存组成，通常位于 CPU 中。
   这种类型的存储类，可以通过在变量声明中使用关键字 register 来请求它。大多数 C++编译器都会忽略这个关键字，而将这种变量置于栈中，但是对寄存器内存可能具有更高的访问优先级。为某个对象请求这种存储类，意味着不能用取址运算符(&)获得它的地址。

20.3.1 全局变量，静态变量与 QObject

通常而言，存在如下两个理由来证明有必要赋予某个对象全局作用域：

1. 对象的生命周期需要与程序的执行时间相同。
2. 需要在程序的多个位置访问这个对象。

在 C++中，应当尽可能地少使用全局作用域，而应采用其他的机制。但是，下面情况依然可使用全局作用域标志符：

- 类名称
- 命名空间名称
- 全局指针 qApp 指向正在运行的 QApplication 对象。

如果将全局对象变成静态类成员或者命名空间成员，就可以避免扩大全局命名空间的规模，但依然可以使这个对象能被多个源代码模块访问。

静态变量与 QObject

当创建 QObject 或者其他感兴趣的类[③]时，要注意的是，在 qApp(单一的 QApplication)被销毁之后，即 main()完成之后，不应当再调用它们的析构函数。

静态 QObject(以及其他的复杂类)在 qApp 被销毁之后如果依旧存在，则可能会存在代码清理问题。这是因为，当 qApp 被销毁时，它会带走许多其他的对象。

通常而言，需要对所有复杂对象的析构顺序进行控制。一种途径是确保在栈(或者堆)上分配的每一个 QObject 都是某个已经位于栈上的 QObject 的子对象或者孙对象等。

QApplication(或者它的派生实例)是所有这些对象的“最根本”栈对象，所以应尽量使其成为“最后一个存在的 QObject”。这使得 qApp 成为了程序中任何孤立 QObject “最后的祖先”的一个好选择。

---

① 可选关键字 auto 几乎从不使用。
② 或者为另一个对象的成员。
③ 这里的“感兴趣的类”，指的是其析构函数需执行某些重要的清理工作的任何类。

20.3.1.1　全局变量与 const

const 全局变量的作用域与常规的全局变量的作用域稍有不同。

默认情况下，被声明成 const 的全局对象具有文件作用域。与在所有语句块之外声明的静态对象不同，如果在初始化时将全局 const 变量声明成 extern 类型，则可以将它扩展到其他的文件中。例如，在一个文件中，可以具有如示例 20.8 所示的代码。

示例 20.8　src/const/globals/chunk1.cpp

```
const int NN = 10;       // file scope
const int MM = 44;       // file scope
extern const int QQ = 7; // can be accessed from other files

int main() {
// NN = 12;          // error
   int array[NN];    // okay
// QQ++;             // error
   double darray[QQ];
   return 0;
}
```

在另一个文件中，可以具有如示例 20.9 所示的代码。

示例 20.9　src/const/globals/chunk2.cpp

```
extern const int NN = 22;      // a different constant
extern const int MM;           // error
// declare global constant - storage allocated elsewhere
extern const int QQ;     // external declaration

void newFunction() {
   int x = QQ + NN;
}
```

示例 20.9 中有一个 const int 变量 NN，它与示例 20.8 中同名的 const 变量彼此不相干。示例 20.9 中可以共享 const int QQ 的使用，因为存在 extern 修饰符。示例 20.9 不能通过用 extern 修饰符声明 MM 来访问具有文件作用域的 const MM。

20.3.2　练习：存储类

给出示例 20.10 中创建/定义每一个对象时的作用域/存储类。如果存在名称冲突，描述所发生的错误。

示例 20.10　src/storage/storage.cpp

```
#include <QString>


int i;                                    1
static int j;                             2
extern int k;                             3
const int l=10;                           4
extern const int m=20;                    5
```

```
class Point                                                    6
{
    public:
    QString name;                                              7
    QString toString() const;
    private:
    static int count;
    int x, y;                                                  8
};

int Point::count = 0;                                          9

QString Point::toString() const {
    return QString("(%1,%2)").arg(x).arg(y);
                                        /* Scope: _____   Storage class: _____
*/
}

int main(int argc, char** argv)                                10
{
    int j;                                                     11
    register int d;
    int* ip = 0;                                               12
    ip = new int(4);                                           13
    Point p;                                                   14
    Point* p2 = new Point();                                   15
}
```

1 作用域：_____ 存储类：_____
2 作用域：_____ 存储类：_____
3 作用域：_____ 存储类：_____
4 作用域：_____ 存储类：_____
5 作用域：_____ 存储类：_____
6 作用域：_____ 存储类：_____
7 作用域：_____ 存储类：_____
8 作用域：_____ 存储类：_____
9 作用域：_____ 存储类：_____
10 argc 和 argv 的作用域/存储类：_____
11 作用域：_____ 存储类：_____
12 作用域：_____ 存储类：_____
13 作用域：_____ 存储类：_____
14 作用域：_____ 存储类：_____
15 作用域：_____ 存储类：_____

20.4 命名空间

在 C 和 C++中，存在一个包含如下两类信息的全局作用域：

- 所有全局函数和变量的名称
- 通常对全部程序都可用的类名称和类型名称

类是分组一个共同的标题(类名称)下的名称(成员)的一种方式，但是有时希望对名称具有更高级的分组形式。

命名空间机制提供了将全局作用域分组成各个命名的子作用域的途径。这有助于避免命名冲突，当开发使用模块的程序时，有可能出现命名冲突。定义命名空间的语法如下：

```
namespace namespaceName { decl1, decl2, ...}
```

任何合法的标志符都可以用于可选的 *namespaceName*(命名空间名称)。示例 20.11 和示例 20.12 在不同的文件中定义了两个独立的命名空间，每一个都包含相同名称的函数。

示例 20.11　src/namespace/a.h

```
#include <iostream>

namespace A {
    using namespace std;
    void f() {
        cout << "f from A\n";
    }

    void g() {
        cout << "g from A\n";
    }
}
```

示例 20.12　src/namespace/b.h

```
#include <iostream>

namespace B {
    using namespace std;
    void f() {
        cout << "f from B\n";
    }

    void g() {
        cout << "g from B\n";
    }
}
```

示例 20.13 中包含两个头文件，并使用作用域解析来调用这两个文件中声明的函数。

示例 20.13　src/namespace/namespace1.cc

```
#include "a.h"
#include "b.h"

int main() {

    A::f();
    B::g();
}
```

输出如下所示。

```
f from A
g from B
```

using 关键字使得不必使用作用域解析即可引用命名空间中的各个成员。它的语句具有如下两种形式。

1. using 指令

   ```
   using namespace namespaceName
   ```

   将整个命名空间导入到当前作用域中。

2. using 声明

   ```
   using namespaceName::identifier
   ```

   将某个特定的标志符从命名空间导入到当前作用域中。

必须仔细地多加练习，以确保当标志符出现在多个命名空间中时不会导致歧义。示例 20.14 是一个具有歧义的函数调用的例子。

示例 20.14 src/namespace/namespace2.cc

```
#include "a.h"
#include "b.h"

int main() {
    using A::f;                          1
    f();
    using namespace B;                   2
    g();                                 3
    f();                                 4
}
```

1 声明——将“A::f()”带入作用域。

2 将命名空间 B 中的全部名称带入作用域。

3 可以这样做。

4 出现歧义。

输出如下所示。

```
f from A
g from B
f from A
```

提示

为了确保各种命名空间的名称是唯一的，有时程序员需要使用非常长的命名空间名称。利用类似下面的命令，就可以很轻易地为长的命名空间名称定义一个别名：

```
namespace xyz = verylongcomplicatednamespacename;
```

20.4.1 匿名命名空间

没有名称的命名空间，是一个匿名命名空间，它与静态全局标志符或者静态文件作用域标志符相似。匿名命名空间的作用域从它的声明点开始，到文件结尾处结束[①]。

① 除非这个匿名命名空间出现在另一个命名空间中(C++语言允许这样)。如果是这样，则匿名命名空间的作用域会被进一步由包含它的命名空间的方括号所限制。

示例 20.15 表明，使用匿名命名空间就可以不使用静态全局声明。

示例 20.15　src/namespace/anonymouse.h

```
namespace {
    const int MAXSIZE = 256;
}

void f1() {
    int s[MAXSIZE];
}
```

20.4.2　开放的命名空间

任何命名空间定义都是开放的，这意味着可以通过声明另一个名称相同但具有新项目的命名空间，向已有的命名空间增加成员。新项目被添加到命名空间中的顺序，是编译器遇到这些命名空间声明的顺序。

类与命名空间相似，但是类不是开放的，因为它必须充当对象创建的模式。

using 指令不会扩充使用它的作用域，它会将名称从指定的命名空间导入到当前的作用域中。

局部定义的名称的优先级，会被来自于命名空间的名称的优先级高(但这种名称依然可通过作用域解析运算符访问)。

20.4.3　命名空间，静态对象与 extern

在命名空间里面声明的对象隐含为静态的，这意味着在整个程序中它只能被创建一次。静态对象的初始化只能存在于一个 C++模块中。为了声明一个静态(全局或者命名空间)对象而不定义它，必须使用关键字 extern[①]。示例 20.16 展示了如何声明命名空间变量。

示例 20.16　src/qstd/qstd.h

```
[ . . . . ]
namespace qstd {

    // declared but not defined:
    extern QTextStream cout;
    extern QTextStream cin;
    extern QTextStream cerr;

    // function declarations:
    bool yes(QString yesNoQuestion);
    bool more(QString prompt);

    int promptInt(int base = 10);
    double promptDouble();
    void promptOutputFile(QFile& outfile);
    void promptInputFile(QFile& infile);
};
[ . . . . ]
```

① 即使位于命名空间里面也需要使用这个关键字!

函数和类可以在命名空间的头文件中声明或者定义。但是，如果在头文件中没有定义，则命名空间中的每一个顶级对象(它对命名空间函数不是局部的)都必须在.cpp 文件中定义，如示例 20.17 所示。

示例 20.17 src/qstd/qstd.cpp

```
[ . . . . ]
QTextStream qstd::cout(stdout, QIODevice::WriteOnly);
QTextStream qstd::cin(stdin, QIODevice::ReadOnly);
QTextStream qstd::cerr(stderr, QIODevice::WriteOnly);

/* Namespace members are like static class members */
bool qstd::yes(QString question) {
    QString ans;
    cout << QString(" %1 [y/n]? ").arg(question);
    cout.flush();
    ans = cin.readLine();
    return (ans.startsWith("Y", Qt::CaseInsensitive));
}
```

20.5 复习题

1. 什么是作用域？什么类型的“事物”具有作用域？
2. 什么是存储类？什么类型的“事物”具有存储类？
3. 何时初始化静态对象？要注意同时考虑全局对象和块作用域对象。
4. const 是如何充当作用域修饰符的？
5. extern 的含义是什么？
6. 根据所用场合的不同，关键字 static 具有多种含义。
   a. 解释 static 如何能用作作用域修饰符。
   b. 解释 static 如何能用作存储类修饰符。
   c. 给出关键字 static 的另一种用法。
7. 定义在某个命名空间中的对象的存储类是什么？
8. 对于使用关键字 register 声明的指向某个对象的指针，其特殊之处在哪里？
9. 类与命名空间有什么不同？
10. 如果希望在命名空间的头文件中声明某个对象而不定义它，则必须做什么？

第21章 内存访问

数组和指针是C程序中低级的构建块，它们提供了对硬件内存的快速访问。本章将探讨组织和访问内存的不同方式。

对内存的直接操作存在很大的风险，要求有很好的经验和全面的测试，以避免出现严重的错误。误用指针和动态内存，可以使程序崩溃，导致堆冲突和内存泄漏。堆错误尤其难以调试，因为它通常会导致段错误(segmentation fault)，挂起程序所在的代码点，可能会远离出现损坏的堆所在的那一点。

Qt 和标准库容器类都允许安全地使用动态内存而不会影响到性能[①]。数组实现了大多数容器类，这种实现对客户代码是隐藏的。这些安全因素来自于对每一个容器API的精心设计，使得不允许存在可能导致内存问题的动作。

Qt提供了许多容器，从高级模板类(如6.8节中讨论过的一个)到低级容器(如QBitArray和QByteArray)。

通常而言，当编写复用这些容器的程序时，很容易就能完全避免使用数组。如果无法使用Qt，或者需要编写一个与C代码的接口，则可能需要使用数组和指针，并要直接操作所分配的内存。

现代的软件经常包含彩色图形以及声音，对于它们的执行要求能够快速地处理。这通常意味着需大量使用动态内存。在一般的计算设备上，已经有大容量的内存和辅助存储设备可用——就在几年前，这还是不可想像的。尽管如此，图形、动画和声音所要求的大量内存，都必须小心而有效率地处理。这就是本章关注内存资源的正确管理以及分析错误管理如何会导致严重后果的原因。

21.1 指针误用

1.15节中讲解过指针，并演示过它的一些基本用法。下面给出的两个短的代码示例表明，如果没有正确地处理指针，就可能发生奇怪的和危险的事情。示例21.1中给出了声明指针的多种方法。

示例 21.1 src/pointers/pathology/pathologydecls1.cpp

```
[ . . . . ]

int main() {
    int a, b, c;                                    1
    int* d, e, f;                                   2
    int *g, *h;                                     3
    int* i, * j;                                    4
```

① 以后，当使用术语“容器”时，如果没有进一步的限制，指的就是Qt或者标准库容器。

```
    return 0;
}
```

1 正如所期望的，这一行会创建三个 int 变量。
2 这一行创建了一个指向 int 变量的指针和两个 int 变量。
3 这一行创建了指向 int 变量的两个指针。
4 这一行也创建了指向 int 变量的两个指针。

初学者会想当然地认为 main()中的第二行会创建三个指针——毕竟在第一行中，同样的语法创建了三个 int 变量。但是，当在一行中声明多个变量时，星号类型修饰符只会作用于紧跟在它后面的那个变量，而不会作用于前面的那个类型符号。正是由于这个原因，推荐对每一个指针都用一个单独的声明(单独一行)。示例 21.2 中包含了三组语句。

示例 21.2 src/pointers/pathology/pathologydecls2.cpp

```
[ . . . . ]
int main() {
    int myint = 5;
    int* ptr1 = &myint;
    cout << "*ptr1 = " << *ptr1 << endl;
    int anotherint = 6;
//  *ptr1 = &anotherint;                          1

    int* ptr2;                                    2
    cout << "*ptr2 = " << *ptr2 << endl;
    *ptr2 = anotherint;                           3

    int yetanotherint = 7;
    int* ptr3;
    ptr3 = &yetanotherint;                        4
    cout << "*ptr3 = " << *ptr3 << endl;
    *ptr1 = *ptr2;                                5
    cout << "*ptr1 = " << *ptr1 << endl;

    return 0;
}
[ . . . . ]
```

1 错误，从 int*到 int 的无效转换。
2 未初始化的指针。
3 不可预测的结果。
4 常规的赋值。
5 危险的赋值。

这段代码中存在一些严重的指针问题，其中最坏的情况没有被编译器检测到。首先出现的是一个简单的类型失配问题。编译结果如下：

```
src/pointers/pathology> g++ pathologydecls2.cpp
pathologydecls.cpp: In function `int main()':
pathologydecls.cpp:17: error: invalid conversion from `int*' to `int'
src/pointers/pathology>
```

将这个无效转换操作注释掉之后，再次尝试编译：

```
*ptr1 = 5
*ptr2 = -1218777888
*ptr3 = 7
*ptr1 = 6
Segmentation fault
```


对未初始化的指针 ptr2 进行解引用操作，会导致不可预测的(即未定义的)结果。

为读而解引用未初始化的指针，就已经是一种很不好的操作，更不用说写这种指针了。这是内存冲突(memory corruption)的一种形式，它能够导致程序在以后执行时出现问题。段错误是由解引用 ptr2 时发生的内存冲突而引起的。

段错误还不是最坏的情况。至少，它会警告程序员存在一个严重的问题。更糟糕的情形是，如果不中断程序的执行而是让它运行，则会产生错误的、看似合理的结果。没有几个人愿意花时间和精力来检验计算机的运算结果！如果发生内存冲突，则任何事情都有可能发生。

21.2 带有堆内存的更多指针误用

对拥有堆中一个有效对象的地址的指针进行删除操作的结果，是将这个堆内存的状态从“使用中”变为“可用”。对指针进行删除操作后，指针本身的状态是未定义的。它也许依旧会保存所删除内存的地址，也许不会。所以如果再次对同一个指针进行删除操作会导致运行时问题，可能是堆冲突。

通常而言，编译器无法检测到对同一个对象的多次删除操作，尤其是当内存块(或其一部分)被重新分配时。为了避免这种重复删除所带来的不良后果，一种好的做法是在删除指针后立即对它赋值 0 或者 NULL。

对空指针执行删除操作，不会有任何动作，也不会有错误发生。

对不是由 new 操作返回的非空指针执行删除操作，会导致未定义的结果。通常而言，编译器无法判断出指针是否是由 new 操作返回的，所以可能导致未定义的运行时行为。需牢记的一条根本原则是：正确地使用删除操作是程序员的责任。

运行时错误的最多来源之一是内存泄漏。如果程序分配了内存，但随后丢失了它的踪迹，导致既无法访问也不能删除它，这就是内存泄漏。没有被正确地删除的对象，在进程终止之前将一直占据内存。

有些程序(例如，服务器程序、操作系统程序)会长时间保持有效状态。假设这样的程序中包含了一个经常要执行的例程，每次运行它时都会导致内存泄漏。由于充满了这些不可访问的、未删除的内存块，堆会逐渐变得支离破碎。在某一刻，如果有例程需要大量的连续动态内存，就有可能拒绝这种请求。如果程序没有为处理这种事件做好准备，它就会中断。

运算符 new 和 delete 使 C++程序员具备更强的能力，同时也增加了责任性。

以下是演示内存泄漏的一些代码样本。在定义完两个指针后，内存应如图 21.1 所示。

```
int* ip = new int;          // allocate space for an int
int* jp = new int(13);      // allocate and initialize
cout << ip  << '\t'  <<  jp  << endl;
```

执行完下面的代码行之后，内存应如图 21.2 所示。

```
jp = new int(3);    // reassign the pointer - MEMORY LEAK!!
```

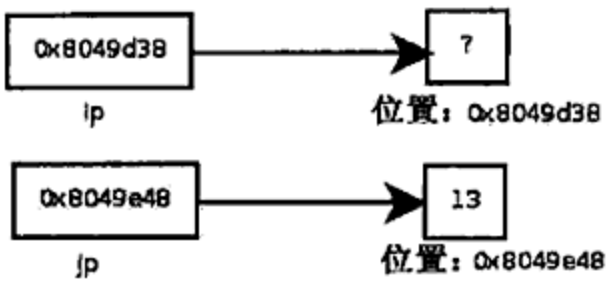


图 21.1 内存中的初始值

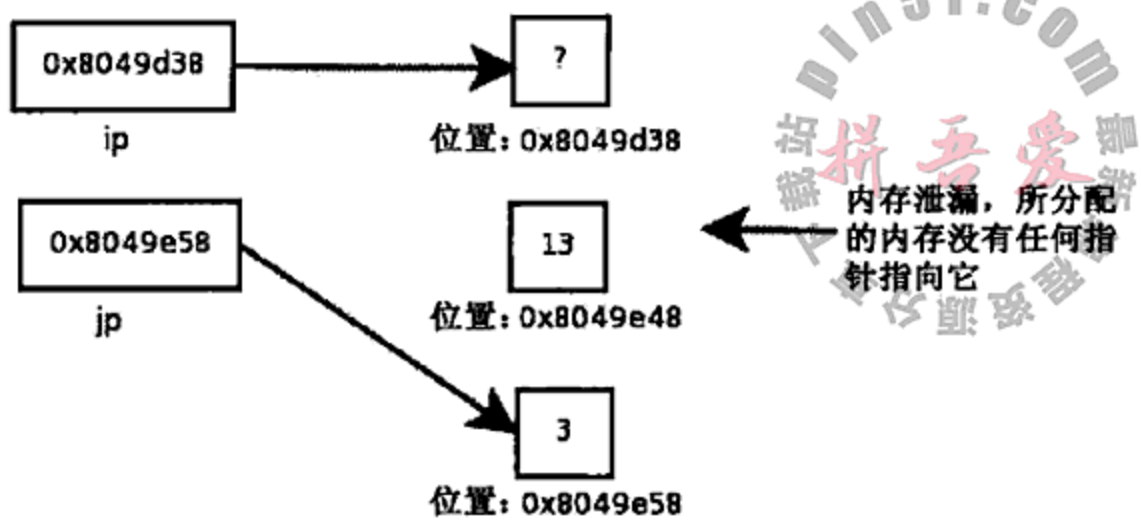


图 21.2 发生内存泄漏后的结果

示例 21.3 中，对指针 jp 删除了两次。

示例 21.3 src/pointers/pathology/pathologydemo1.cpp

```
#include <iostream>
using namespace std;

int main() {
    int* jp = new int(13);                                    1
    cout << jp << '\t' << *jp << endl;
    delete jp;
    delete jp;                                                2
    jp = new int(3);                                          3
    cout << jp << '\t' << *jp << endl;
    jp = new int(10);                                         4
    cout << jp << '\t' << *jp << endl;
    int* kp = new int(17);
    cout << kp << '\t' << *kp << endl;
    return 0;
}
```

1 分配并初始化。

2 错误：指针已经被删除。

3 重新分配指针，导致内存泄漏。

4 重新分配指针，导致内存泄漏。

输出如下所示。

```
OOP> g++ pathologydemo1.cpp
OOP> ./a.out
0x8049e08       13
0x8049e08       3
0x8049e08       10
Segmentation fault
OOP>
```

第二个删除操作是一种严重的错误，但是编译器不会报错。这个错误会损坏堆，使得无法进行进一步的内存分配，导致此后的程序行为是未定义的。例如，如果在重新为指针 jp 分配内存时出现内存泄漏，就不会再获得新的内存空间了。当试图使用另一个指针变量时，就会发生段错误。这是一种未定义的行为，且在不同的平台或者不同的编译器下有不同的表现。

21.3 内存访问小结

以下是关于内存访问的最重要的几点：

- 运算符 new 和 delete 使 C++程序员具备更强的能力，同时也增加了责任性。
- 误用指针和动态内存，可以使程序崩溃，导致堆冲突和内存泄漏。
- Qt 和标准库容器类都允许安全地使用动态内存而不会影响到性能。
- 当在一行中声明多个变量时，星号类型修饰符只会作用于紧跟在它后面的那个变量，而不会作用于前面的那个类型符号。
- 解引用一个未初始化的指针，是一种不会被编译器捕获的严重错误。
- 对指针进行删除操作后，指针本身的状态是未定义的。
- 一种好的做法是在删除指针后立即对它赋值 0 或者 NULL。
- 对不是由 new 操作返回的非空指针执行删除操作，会导致未定义的结果。
- 编译器无法判断出 delete 的错误使用，所以正确地使用删除操作是程序员的责任。
- 如果程序分配了内存，但随后丢失了它的踪迹，导致既无法访问也不能删除它，这就是内存泄漏。

21.4 数组简介

数组是一组连续的内存单元，这些内存单元具有相同的大小。每一个单元被称为数组元素或者数组项。

当声明数组时，必须告知数组的大小。数组大小可以显式地给出，也可以通过初始化来确定：

```
int a[10]; // explicitly creates uninitialized cells a[0], a[1],..., a[9]
int b[] = {1,3,5,7}; // implicitly creates and initializes b[0],..., b[3]
```

数组名称是指向数组第一个单元的 const 类型的指针的别名。指针声明

```
int* ptr;
```

只会创建这个指针变量。指针变量没有自动的默认初始化值。如果解引用未初始化的指针，则是一个错误。

指针索引是从基地址开始的相对偏移量：

a[k]等价于*(a + k)

示例 21.4 中给出了数组索引的一个有趣特性。

示例 21.4 src/pointers/pathology/pathologydemo2.cpp

```
#include <iostream>
using namespace std;
int main(){
  int a[] = {10, 11, 12, 13, 14, 15};

  int* b = a + 1;
  cout << "a[3] = " << a[3] << '\n'
       << "b[3] = " << b[3] << endl;
```

```
    //It gets even worse.
    int c = 123;
    int* d = &c;
    cout << "d[0] = " << d[0] << '\n'
         << "d[1] = " << d[1] << '\n'
         << "d[2] = " << d[2] << endl;

}
```

编译并运行这个程序的输出如下。

```
pointers/pathology> g++ -ansi -pedantic -Wall pathologydemo2.cpp
pointers/pathology>
a[3] = 13
b[3] = 14
d[0] = 123
d[1] = -1075775392
d[2] = -1219610235
```

注意，b 和 d 都没有被声明成数组，但是编译器允许使用使用下标运算符[]。c 是一个常规的 int 变量，而 d 是一个常规的 int 指针。d[0]，d[1]和 d[2]没有定义，但是编译器不会对它们的出现给出任何警告——尽管使用了三个命令行选项。

这是一种特殊的语法，用于定义由给定数量的某种类型的元素组成的动态数组：

```
uint n;
ArrayType* pt;
pt = new ArrayType[n];
```

这种 new 操作会分配 *n* 个连续的内存块，每一个内存块的大小为 sizeof(ArrayType)，并会返回一个指向第一个内存块的指针。新分配的数组的每一个元素都会被默认初始化。为了正确地解分配(deallocate)这个数组，需要使用语法：

```
delete[] pt;
```

进行 delete 操作删除动态数组时，如果没有包含空的方括号，则会导致未定义的结果。

关于系统无法完成动态内存请求时所发生的异常以及其他动作，在 dist 目录下单独的一篇文章中进行了讨论[①]。

21.5 指针的算术运算

对指针进行+, -, ++或--运算的结果，与它所指向的对象类型有关。当算术运算符用于类型为 *T**的指针 p 时，假定 p 指向的数组元素为 *T* 类型的对象。

- p + 1 指向数组中的下一个元素。
- p - 1 指向数组中的前一个元素。
- 通常而言，p + k 的地址值会比 p 的地址值大 k * sizeof(T)字节。

指针的减法只有当两个指针都指向同一个数组中的元素时才有意义。这种情况下，其结果是一个与两个元素间数组元素的个数等价的 int 值。

① 参见 articles/exceptions.html。

在数组环境之外进行的指针算术运算结果是未定义的，程序员有责任确保正确地使用了指针算术运算。示例21.5中演示了这种情况。

示例21.5 src/arrays/pointerArith.cpp

```
[ . . . . ]
int main()  {
    using namespace std;
    int y[] = {3, 6, 9};
    int x = 12;
    int* px;
    px = y;                                                    1
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    return 0;
}
```

1 y或者任何数组名称，是指向数组第一个元素的指针的别名。

编译并运行这个程序的输出如下[①]。

```
src/arrays> g++ -ansi -pedantic -Wall pointerArith.cpp
pointerArith.cpp: In function 'int main()':
pointerArith.cpp:6: warning: unused variable 'x'
src/arrays> ./a.out
What's next: 6
What's next: 9
What's next: -1080256152
What's next: 12
```

注意，编译器和运行时系统都不会给出错误消息。但是，给出了没有使用变量x的一个警告。C++很乐意读取任意内存地址，并以所选择的类型报告它们，这样就在为C++开发人员提供强大功能的同时，也为犯错提供了大量机会。

21.6 数组，函数与返回值

正如C语言中那样，为函数所声明的返回类型不能是数组(例如，不能是int[]，char[]或Point[]形式)。从函数返回数组(的地址)为指针类型，这是允许的。但是，对类的公共接口不推荐这样做。

我们已经看到，数组是一个没有受保护的内存块。封装了这块内存的类，不应具有返回指向它的指针的公共成员。如果这样做，客户代码就有可能错误地使用这块内存。正确设计的类，会全面封装它的实现中使用的任何数组与这块内存的交互操作。

数组从来不会将值传递给函数。也就是说，数组元素不会被复制。如果调用函数时其实参表中包含数组，例如，

```
int a[] = {10, 11, 12, 13, 14, 15};
void f(int a[]) {
```

① 通常而言，访问位于数组边界之外的内存会导致未定义的结果，这里由于是故意这样做的，所以结果是未定义的。

```
    [ ... ]
}
    [ ... ]
f(a);
```

则实际传递的值仅仅是数组中第一个元素的指针。示例 21.6 演示了这一点，输出了传递给函数的数组和返回的数组。

示例 21.6 src/arrays/returningpointers.cpp

```
#include <assert.h>

int paramSize;

void bar(int* integers) {
    integers[2]=3;                                          1
}

int* foo(int arrayparameter[]) {
    using namespace std;
    paramSize = sizeof(arrayparameter);
    bar(arrayparameter);                                    2
    return arrayparameter;                                  3
}

int main(int argc, char** argv) {
    int intarray2[40] = {9,9,9,9,9,9,9,2,1};
    char chararray[20] = "Hello World";                     4
    int intarray1[20];                                      5
    int* retval;                                            6

//  intarray1 = foo(intarray2);                             7

    retval = foo(intarray2);
    assert (retval[2] == 3);
    assert (retval[2] = intarray2[2]);
    assert (retval == intarray2);
    int refSize = getSize(intarray2);
    assert(refSize == paramSize);
    return 0;
}
```

1 改变输入数组中的第三个元素。
2 用指针将数组传递给函数。
3 从函数将数组当作指针返回。
4 用于初始化 char 数组的特殊语法。
5 未初始化的内存。
6 未初始化的指针。
7 错误：intarray1 类似 char* const，不能这样赋值。

21.7 不同类型的数组

基本类型的数组，例如 int，char 和 byte 类型的数组，被用来实现缓存。出于与 C 语言中 struct 数组后向兼容的考虑，对象数组在 C++语言中是支持的，但是只能将它用在同一种结构的相同集合中，而不能用于相似的多态对象的集合中。

如果需要随机访问所保存的项目，(Qt 中的)QList 或者(STL 中的)vector 都可以用来替换数组。在底层，它们都是用动态数组实现的。只要有可能，就应优先使用这样的容器而不是数组，因为容器能够正确且安全地分配和释放内存。

21.8 有效的指针操作

以下给出的是能够正确地在指针上执行的操作。

创建——指针的初始值可以有三种来源：

- 由所声明的指针变量或者 const 指针(例如，数组名称)获得的栈地址。
- 由取址运算符&获得的地址。
- 由动态内存分配运算符(如 new)获得的堆地址。

赋值

- 指针能够被赋予同一种类型或者派生类型的指针所保存的地址。
- 可以将 void*类型的变量赋予任何类型的指针，不需要显式的转型。
- 将另一种(或者非派生)类型的指针所保存的地址赋予非 void*类型的指针时，只能使用显式的转型。
- 数组名称就是一个 const 指针，不能被赋值。
- 可以将 NULL 指针(值为 0)赋予任何指针(Stroustrup 推荐使用 0 而不是 NULL)。

算术运算

- 可以对指针进行增 1 操作或者减 1 操作，即 p++或者 p--。
- 可以将指针与整数相加或者相减，即 p + k 或者 p - k。
- 只有当得到的指针值位于同一个数组的范围之内时，这种加减操作才是被定义的。唯一的例外情况是：只要指针不试图解引用位于数组末尾后一个位置处的内存单元地址，这个指针就被允许指向这个位置。
- 两个指针可以进行减法操作。指向同一数组两个成员的两个指针相减后，其结果是表示两个元素之间的数组元素数量的一个 int 值。

比较

- 指向同一个数组中不同元素的两个指针，可以用==, !=, <, >等运算符进行比较。
- 任何指针都能够与 0 进行比较。

间接赋值

- 如果 p 为 T*类型的指针，则*p 为 T 类型的变量，且可以将其用于赋值运算的左边。

索引

- 指针 p 可以用于数组索引运算符 p[i]，其中 i 为 int 类型。编译器会将这样的表达式解释为*(p + i)。
- 索引只有用于数组时才有意义并可定义，但是编译器不会阻止它用于非数组指针中，其结果是未定义的。

示例 21.7 中清楚地演示了最后一种情况。

示例 21.7 src/arrays/pointerIndex.cpp

```
#include <iostream>
using namespace std;

int main()  {
    int x = 23;
    int y = 45;
    int* px = &x;
    cout << "px[0] = " << px[0] << endl;
    cout << "px[1] = " << px[1] << endl;
    cout << "px[2] = " << px[2] << endl;
    cout << "px[-1] = " << px[-1] << endl;
    return 0;
}
```

输出如下所示。

```
// g++ on Mac OSX:
px[0] = 23
px[1] = 1606413624
px[2] = 32767
px[-1] = 45

// g++ on Linux (Ubuntu):
px[0] = 23
px[1] = -1219095387
px[2] = -1216405456
px[-1] = 45

// g++ on Windows XP (mingw)
px[0] = 23
px[1] = 45
px[2] = 2293588
px[-1] = 2009291924

// Windows XP with MS Visual Studio compiler:
px[0] = 23
px[1] = 45
px[2] = 1245112
px[-1] = 1245024
```

有一个小型的具体例子，它给出了“未定义行为”的含义。对初学者而言，对于程序栈中连续定义的变量是如何排列的，可以原谅他们对此做出的一些假设。对非数组指针使用数组下标，或者使用的下标超出了数组的范围，只不过是暴露这些假设的情形之一。现在，尝

试使用下标-1，然后努力使不同平台上的不同结果保持一致。这个例子太短了，以至于无法给出“未定义行为”的全貌。指针的未定义行为，通常会导致内存冲突，而内存冲突是编程的噩梦——程序员更期望看到的结果是运行时中断。

21.9 数组与内存

以下是本章中已经提到的最重要的几点：

- 数组是一组连续的内存单元，这些内存单元具有相同的大小。
- 数组名称是指向数组第一个单元的 const 类型的指针的别名。
- 指针变量没有自动的默认初始化值。
- 指针索引是从基地址开始的相对偏移量。
- 只有当用来访问数组的成员，且访问是位于数组范围之内时，数组下标才是有效的。
- C++标准不保证编译器会捕获到将指针用于非数组的下标运算符的企图。
- 将数组传递给函数和从函数返回，是通过指针进行的。
- 可以将算术运算符+, -, ++和--运用到数组指针，只要其结果是有效的。
- 在数组环境之外进行的指针算术运算结果是未定义的。
- C++标准不保证编译器能够捕获误用指针运算的企图。
- 指针只能通过如下方式获得值：
  - 在创建时初始化。
  - 创建后对其赋值。
  - 作为指针运算的结果。
- *ArrayType* 的 size 元素的动态数组是用如下语法分配内存的：

```
uint size;
ArrayType* pt;
pt = new ArrayType[size] ;
```

- 当为数组分配内存时，动态数组的每一个元素都会被默认初始化。
- 为了正确地解分配这个动态数组，需要使用语法：

```
delete[] pt;
```

如果无法执行分配内存的请求，ANSI/ISO 标准要求 new 运算符抛出 bad_alloc 异常而不是返回 NULL。关于异常的更多细节，请查看 dist 目录下的一篇文章[①]。如果无法执行分配内存的请求，则运算符 new(nothrow) 可以返回 0。应将动态数组小心地封装到类中，类应当包含合适的析构函数、复制构造函数以及复制赋值构造函数。

21.10 练习：内存访问

预测示例 21.8 的输出，然后链编并运行它。解释输出的结果。如果与你的预测有所不同，给出理由。

---

① 参见 articles/exceptions.html。

示例 21.8 src/arrays/arrayVSptr.cpp

```
#include <iostream>
using namespace std;

int main()  {
   int a[] = {12, 34, 56, 78};
   cout << a << "\t" << &a[1] - a << endl;
   int x = 99;
     a = &x;
   int* pa;
   cout << pa << endl;
   pa = &x;
   cout << pa << "\t" << pa - &a[3] << endl;
   cout << a[4] << "\t" << a[5] << endl;
   cout << *(a + 2) << "\t" << sizeof(int) <<endl;
   void* pv = a;
   cout << pv << endl;
   int* pi = static_cast<int*>(pv);
   cout << *(pi + 2) << endl;
   return 0;
```

21.11 复习题

1. 下面的语句中定义了什么?

   ```
   int* p, q;
   ```

2. 什么是内存泄漏? 它是如何发生的?
3. 比较将+, -, ++, --运算符用于指针和将它们用于 int 或者 double 类型的值上的不同。
4. 如果将 delete 运算符用于已经被删除的指针上，会发生什么?
5. 如果将数组作为参数传递给函数，程序栈中被复制的是什么?
6. 什么是动态内存? C++中如何获得它?

第22章 继承详解

本章将给出第6章中讲解的一些概念的形式化定义，并会详细分析它们。第6章中讲解了构造函数、析构函数和复制赋值运算符是如何产生并被派生类使用的，探讨了如何能将关键字public，private和protected用于基类和成员，还给出了多重继承的几个示例。

22.1 虚指针和虚表

每一个包含方法(虚函数)的类，都有一个虚跳表(virtual jump table)或者虚表(vtable)，它是作为“轻量级”C++执行环境的一部分而产生的。有多种方法可以实现虚表，其中最简单的实现(通常是最快和最轻量级的)包含一个指向该类全部方法的指针列表。根据优化策略的不同，虚表可以包含有助于调试的更多信息。编译器会将函数名称替换成方法调用的间接(相对于虚表列表)引用。

由此，可以显式地将多态类型定义成包含一个或者多个方法的类，从而要求使用虚表。多态类型的每一个实例都具有一个typeid属性，可以相当自然地将它实现成类的虚表地址。

> **使用虚表而不是switch语句**
>
> 为了通过虚表实现间接的方法调用，编译器会为每一个多态类产生一个跳表，它与switch语句相似。程序员经常能够利用虚表而不必编写switch语句或者大型的复合条件语句。这样做隐含了大量的设计模式，比如命令模式、访问者模式、解释器模式以及策略模式。

只有当类中全部重写的方法被完全定义了并且能够被链接器找到时，才会对这个类建立虚表。

当执行完对象的构造函数后，就会设置它的typeid。如果存在基类，则在初始化每一个基类之后，会多次为对象设置typeid。

示例22.1中定义的类表明，从构造函数或者析构函数调用虚函数可能导致意外的后果。

示例22.1 src/derivation/typeid/vtable.h

```
[ . . . . ]
class Base {
 protected:
    int m_X, m_Y;
 public:
    Base();
    virtual ~Base();
    virtual void virtualFun() const;
};

class Derived : public Base {
```

```
    int m_Z;
public:
    Derived();
    ~Derived();
    void virtualFun() const ;
};
[ . . . . ]
```

示例 22.2 表明了当从基类构造函数或析构函数调用虚函数时会发生什么。

示例 22.2 src/derivation/typeid/vtable.cpp

```
#include <QDebug>
#include <QString>
#include "vtable.h"

Base::Base() {
    m_X = 4;
    m_Y = 12;
    qDebug() << " Base::Base: " ;
    virtualFun();
}

Derived::Derived() {
    m_X = 5;
    m_Y = 13;
    m_Z = 22;
}

void Base::virtualFun() const {
    QString val=QString("[%1,%2]").arg(m_X).arg(m_Y);
    qDebug() << " VF: the opposite of Acid: " << val;
}

void Derived::virtualFun() const {
    QString val=QString("[%1,%2,%3]")
        .arg(m_X).arg(m_Y).arg(m_Z);
    qDebug() << " VF: add some treble: " ;
}

Base::~Base() {
    qDebug() << " ~Base() " ;
    virtualFun();
}

Derived::~Derived() {
    qDebug() << " ~Derived() " ;
}
int main() {
    Base *b = new Derived;                    1
    b->virtualFun();                          2
    delete b;                                 3
}
```

1 调用 Base::virtualFun( )。

2 使用虚表和运行时绑定调用 Derived::virtualFun( )。

3 调用 Base::virtualFun( )。

从随后的输出中可以看出，派生的 virtualFun()没有从 Base::Base()调用，因为基类初始化器位于还没有派生实例的一个对象中。

```
Base::Base:
VF: the opposite of Acid:  "[4,12]"
VF: add some treble:
~Derived()
~Base()
VF: the opposite of Acid:  "[5,13]"
```

不推荐从析构函数调用虚方法。从前一个输出中可以看出，基函数 virtualFun 总是从基类构造函数或者析构函数调用的。动态绑定不会在构造函数或者析构函数内发生。正如 Meyers 所说，“构造函数或者析构函数中不存在虚方法”。

22.2 多态与虚析构函数

当在继承层次中操作这些类时，经常需要维护包含有派生对象地址的基类指针的容器。示例 22.3 中定义的 Bank 类包含一个具有各种 Account 类型的容器。

示例 22.3 src/derivation/assigcopy/bank.h

```
[ . . . . ]
class Account;

class Bank {
 public:
    Bank& operator<< (Account* acct);                    1
    ~Bank();
    QString getAcctListing() const;
 private:
    QList<Account*> m_Accounts;
};
[ . . . . ]
```

1 这就是将对象指针添加到 m_Accounts 中的方法。

示例 22.4 中给出了这些 Account 类的定义。

示例 22.4 src/derivation/assigcopy/account.h

```
[ . . . . ]
class Account {
 public:
    Account(unsigned acctNum, double balance, QString owner);
    virtual ~Account(){
      qDebug() << "Closing Acct - sending e-mail "
               << "to primary acctholder:" << m_Owner; }
    virtual QString getName() const {return m_Owner;}
    // other virtual functions
```

```
 private:
    unsigned  m_AcctNum;
    double    m_Balance;
    QString    m_Owner;
};
class JointAccount : public Account {
 public:
  JointAccount (unsigned acctNum, double balance,
                QString owner, QString jowner);
  JointAccount(const Account & acct, QString jowner);
  ~JointAccount() {
     qDebug() << "Closing Joint Acct - sending e-mail "
              << "to joint acctholder:" << m_JointOwner; }
  QString getName() const {
    return QString("%1 and %2").arg(Account::getName())
                    .arg(m_JointOwner);
  }
  // other overrides
 private:
  QString m_JointOwner;
};
[ . . . . ]
```

通过对每一个所包含的 Account 调用 virtual 方法，Bank 类能够执行统一的操作。示例 22.5 中，delete acct 操作会导致对 Account 析构函数的间接调用，并随后会释放所分配的内存。

示例 22.5 src/derivation/assigcopy/bank.cpp

```
[ . . . . ]

#include <QDebug>
#include "bank.h"
#include "account.h"

Bank::~Bank() {
    qDeleteAll(m_Accounts);
    m_Accounts.clear();
}


Bank& Bank::operator<< (Account* acct) {
   m_Accounts << acct;
   return *this;
}

QString Bank::getAcctListing() const {
   QString listing("\n");
   foreach(Account* acct, m_Accounts)
      listing += QString("%1\n").arg(acct->getName());          1
   return listing;
}
```

1 getName()为虚函数。

尽管列表中的每一个地址都为 Account 对象，但其中的一些(甚至全部)有可能指向派生类对象，从而要求派生类析构函数调用。

如果析构函数为虚函数，则编译器会通过 Account 指针对任何析构函数启用运行时绑定，而不是简单地调用 Account::~Account()。如果没有在基类中将~Account()声明成虚函数，则运行示例 22.6 时将得到不正确的结果[①]。

示例 22.6　src/derivation/assigcopy/bank.cpp

```
[ . . . . ]

int main() {
  QString listing;
   {                                                          1
      Bank bnk;
      Account* a1 = new Account(1, 423, "Gene Kelly");
      JointAccount* a2 = new JointAccount(2, 1541, "Fred Astaire",
         "Ginger Rodgers");
      JointAccount* a3 = new JointAccount(*a1, "Leslie Caron");
      bnk << a1;
      bnk << a2;
      bnk << a3;
      JointAccount* a4 = new JointAccount(*a3);                  2
      bnk << a4;
      listing = bnk.getAcctListing();
    }                                                            3
    qDebug() << listing;
    qDebug() << "Now exit program" ;
}
```

1 内部语句块的开始。

2 这条语句的作用是什么?

3 在这里，作为销毁 bank 类对象的一部分，全部四个 Account 对象都被销毁了。

以下输出是在~Account 中删除 virtual 后的结果：

```
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
Closing Acct - sending e-mail to primary acctholder:Fred Astaire
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
[ ... ]
```

如果析构函数为虚函数，则这时示例中 Account 的两种类型都会被正确地销毁，且当销毁了 Bank 类对象时，联名账户中的两个账户持有人都能够得到合适的 E-mail 通知。

```
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
Closing Joint Acct - sending e-mail to joint acctholder:Ginger Rodgers
```

注意

如果在一个类中声明了一个或者多个虚方法，则应该为这个类定义一个虚析构函数，即使它的函数体为空也应定义。

① 编译器会给出警告，指出析构函数中缺少 virtual，其行为是未定义的，所以在不同的系统上可能会得到不同的结果。

22.3 多重继承

多重继承是继承的一种形式，其中的类会继承多个基类的结构和行为。多重继承的常见用途如下。

- 用于组合存在一定重叠的不同类的功能，如图 22.1 所示。
- 用于以各种不同的方式实现共同的“纯接口”(只有纯虚函数的类)。

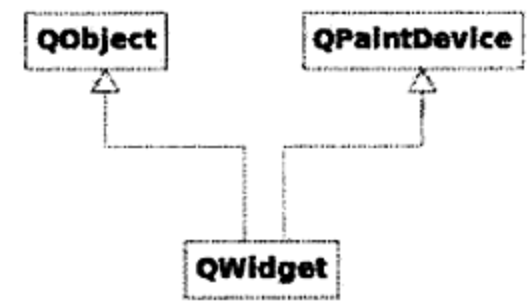


图 22.1 QWidget 的继承

与单一继承一样，多重继承在类之间也定义了一种静态的关系。在运行时不能改变这种关系。

与单一继承层次相比，多重继承层次要复杂得多，更难以设计、实现和理解。可以用它来解决一些设计难题，但如果存在更简单的方法(如聚合)，则不应使用多重继承。

22.3.1 多重继承语法

这一节中的例子给出了多重继承的语法及用法。

图 22.2 中给出了两个基类 Rectangle 和 ScreenRegion，每一个都在屏幕上扮演特定的角色。其中一个类关注的是形状和位置，而另一个关注的是颜色和可见性特征。Window 类必须同时为 Rectangle 类和 ScreenRegion 类，它们的定义参见示例 22.7。

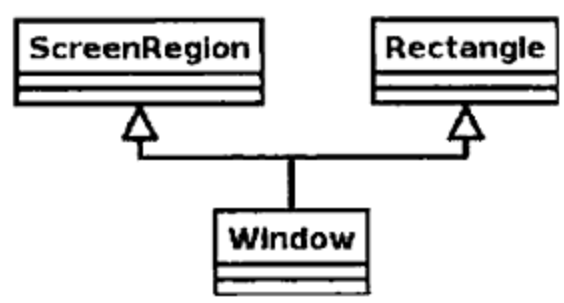


图 22.2 Window 类与 ScreenRegion 类和 Rectangle 类的关系

示例 22.7 src/multinheritance/window.h

```
[ . . . . ]

class Rectangle {
 public:
    Rectangle( Const Point& ul, int length, int width);
    Rectangle( const Rectangle& r) ;
    void move (const Point &newpoint);
 private:
    Point m_UpperLeft;
    int m_Length, m_Width;
};

class ScreenRegion {
 public:
    ScreenRegion( Color c=White);
    ScreenRegion (const ScreenRegion& sr);
    virtual color Fill( Color newColor) ;
    void show();
    void hide();
```

```
 private:
    Color m_Color;
    // other members...

};

class Window: public Rectangle, public ScreenRegion {
 public:
    Window( const Point& ul, int len, int wid, Color c)
        : Rectangle(ul, len, wid), ScreenRegion(c) {}          1

    Window( const Rectangle& rect, const ScreenRegion& sr)
        : Rectangle(rect), ScreenRegion(sr) {}                 2

    // Other useful member functions ...
};
```

1 使用基类 ctors。

2 使用基类复制 ctors。

Window 类的类首部中有一些语法项值得注意：

- 如果派生类不是 private 类型的，则访问指示符(public 或者 protected)必须出现在每一个基类名称的前面。
  - 默认的派生类是 private 类型的。
  - 派生类中可以混用 public， protected 和 private 类型。
- 多个基类用逗号分开。
- 初始化基类的顺序，应与类首部中基类出现的顺序相同。

示例 22.8 中提供了 Window 类的一些客户代码。

示例 22.8 src/multinheritance/window.cpp

```
#include "window.h"

int main() {
    Window w(Point(15,99), 50, 100, Color(22));
    w.show();                                                  1
    w.move (Point(4,6));                                       2
    return 0;
 }
```

1 调用 ScreenRegion::show()。

2 调用 Rectangle::move()。

成员初始化的顺序

对成员的默认初始化或者赋值的顺序，与类定义中声明数据成员的顺序相同：首先是基类，然后是派生类成员。

22.3.2 多重继承与 QObject

Qt 中的许多类都使用多重继承。图 22.3 中给出了 QGraphicsView 中使用的类之间的继承关系。QGraphicsItem 是一个轻量级对象，它不支持信号或者槽。QGraphicsObject

依次继承自 QObject 和 QGraphicsItem，因而具有这两种类的好处：QGraphicsObject 支持信号和槽，通常用在动画中。图 22.1 中所示的 QWidget 是使用了 QObject 多重继承的另一个类。

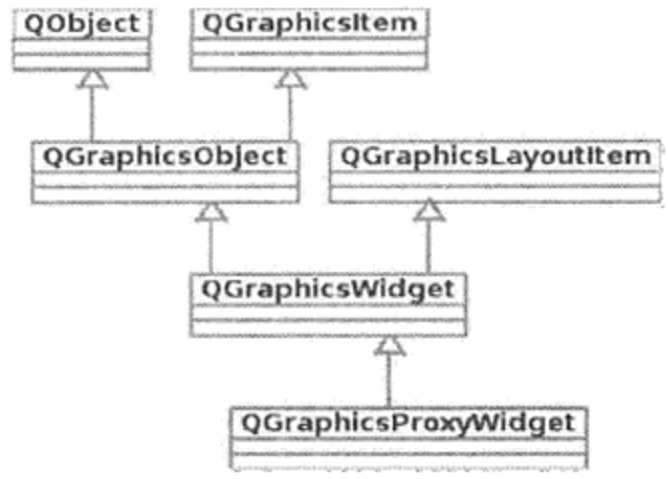


图 22.3　QGraphicsObject 与多重继承

注意

图 22.3 中，QObject 是被多重继承了的基类之一。Qt 中的一个限制是：QObject 只能被任何类继承一次(不支持虚继承)。此外，在基类列表中必须首先列出派生自 QObject 的基类。如果破坏这个原则，则元对象编译器(moc)可能产生奇怪的代码错误信息。

22.3.3　解决多重继承的冲突

图 22.4 给出的 UML 框图中，接口和实现中都错误地使用了多重继承。为了使事情更复杂一些，其中的一个类从同一个基类继承了两次。

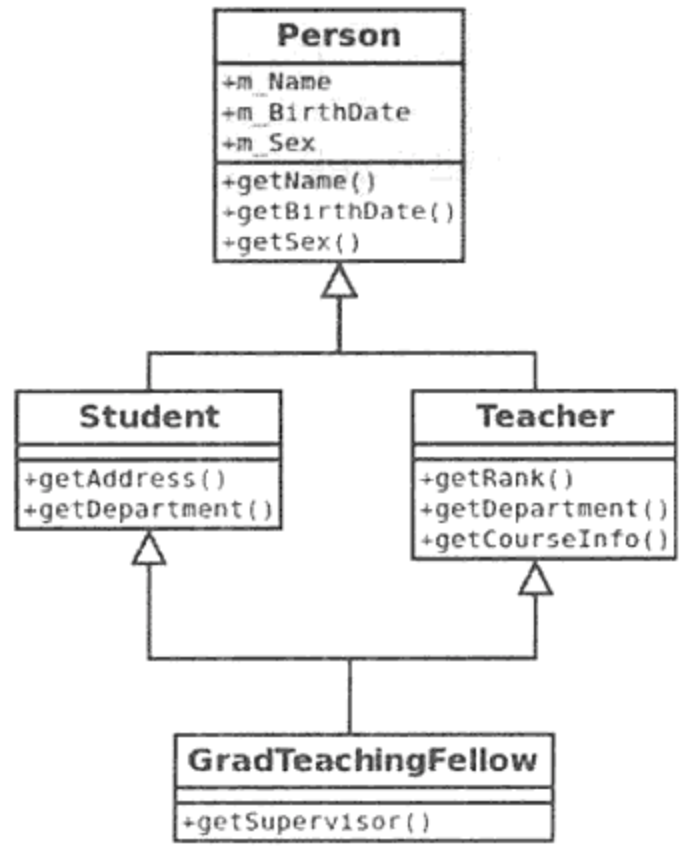


图 22.4　Person-Student-Teacher 继承层次

这里的 GradTeachingFellow 类派生自两个类：Student 和 Teacher。

```
class GradTeachingFellow : public Student,
                           public Teacher {
    // class member functions and data members
```

如果错误地使用了多重继承，就会出现命名冲突和设计问题。这个例子中，Student类和Teacher类都具有getDepartment()函数。这样，学生既可以在某个系学习，又可以在另一个系教书。

问题

当对GraduateTeachingFellow调用getDepartment()函数时，会出现什么问题？

```
GraduateTeachingFellow gtf;
Person* pptr = &gtf;
Student * sptr = &gtf;;
Teacher* tptr = &gtf;
gtf.Teacher::getDepartment();
gtf.Student::getDepartment();
sptr->getDepartment()
tptr->getDepartment()
pptr->getDepartment();    // Ambiguous: runtime error if virtual
gtf.getDepartment();  // Compiler error: ambiguous function call
```

当然，问题在于：没有在GradTeachingFellow类中提供getDepartment()函数。当编译器查找getDepartment()函数时，Student类和Teacher类具有同样的优先级。

应该避免类似这样的继承冲突，因为它会在以后招来许多设计上的麻烦。不过，如果能够通过作用域解析得到解决的话，也可以这样做。

22.3.3.1 虚继承

图22.4中，GraduateTeachingFellow从同一个基类中继承了多次。这种模型会存在另一个问题：冗余。这种多次继承类的实例会如图22.5那样。

Person类的属性应当只被继承一次。对GradTeachingFellow而言，具有两个生日和两个名称没有什么意义。虚继承(virtual inheritance)能够消除这种冗余。

当将多重继承用在有歧义的环境下时，会出现奇怪的问题，尤其是将虚继承/函数与非虚继承/函数混用，增加了复杂性时，这样就似乎提示了Java的设计者不要在Java中使用多重继承。实际上，Java允许程序员定义只由抽象函数(纯虚函数)组成的接口。这样，Java类就能够利用implements子句根据需要实现任意多的接口。

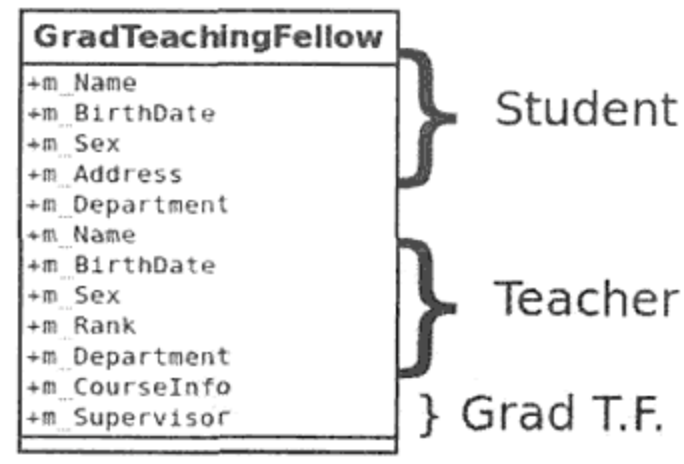


图22.5 GradTeachingFellow,没有虚继承

22.3.3.2 虚基类

可以将基类声明成虚基类。虚基类将它的表示与具有同一个虚基类的其他类共享。

在Student类和Teacher类的类首部中增加关键字virtual，而让类定义的其他细节保持不变，就得到示例22.9。

示例 22.9 src/multinheritance/people.h

```
#include "qdatetime.h"

class Person {
public:
    Person(QString name, QDate birthdate)
    QObject(name.ascii()),
    m_Birthdate(birthdate) {}

    Person(const Person& p) : QObject(p),
    m_Birthdate(p.m_Birthdate) {}

private:
    QDate m_Birthdate;
};

class Student : virtual public Person {                    1
    // other class members
};

class Teacher : virtual public Person {                    2
    // other class members
}
class GraduateTeachingFellow :
    public Student, public Teacher {                       3
public:
    GraduateTeachingFellow(const Person& p,
                           const Student& s, const Teacher& t):
    Person(p), Students(s), Teacher(t) {}                  4
}
```

1 注意这里的关键字 virtual。

2 虚继承。

3 这里并不需要关键字 virtual。

4 在多重派生的类中，有必要显式地初始化全部的虚基类，以消除应如何将它们初始化的歧义。

使用了虚继承之后，GradTeachingFellow 的实例如图 22.6 所示。

从另一个类虚继承的每一个实例，都具有一个指向它的虚基类子对象的指针(或者有可变的偏移量)。对程序员而言，虚基类指针是不可见的，而且通常情况下不需要改变它。

利用多重继承，每一个虚基类指针都指向同一个对象，从而有效地使基类对象能被所有的派生类部分共享。

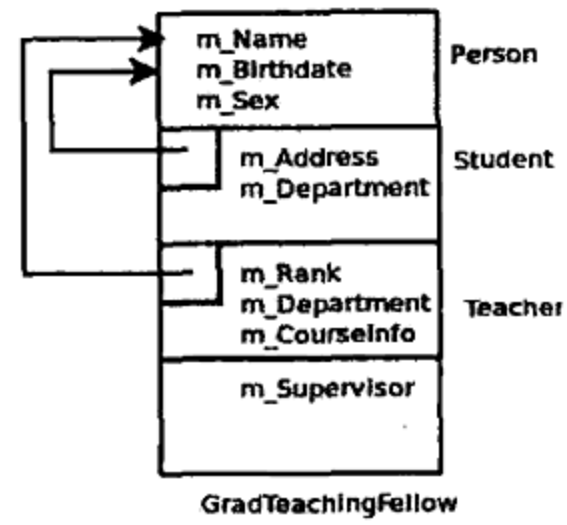


图 22.6 GradTeachingFellow，有虚继承

对于具有虚基类的任何类，对于这个虚基类的成员初始化项必须出现在这个类的成员初始化表中。否则，虚基类将会被默认初始化。

22.4 public, protected 和 private 派生

多数情况下，我们看到的子句都采用 public 派生，例如：

```
class Square : public Shape {
// ...  };
```

这种派生关系描述了两个类之间的接口关系。这意味着基类的接口(public 部分)会与派生类的接口合并。如果能够将派生类对象称为“是”基类对象时(即二者存在“是”关系)，就适合使用 public 派生。例如，正方形“是”一种(具有更多属性的)形状。

较少看到的是 protected 派生或者 private 派生，它们被认为是“实现”关系而不是“是”关系。基类接口(public 部分)会与派生类的实现(根据派生类型的不同，可以是 private 类型或者 protected 类型)合并。

在效果上，private 派生就如同向派生类添加了一个额外的对象作为其 private 数据成员。

类似地，protected 派生就如同向派生类添加了一个额外的对象作为其 protected 数据成员，但这个对象会共享 this 指针。

示例 22.10 就是这种情形的一个具体例子，其中的 private 派生就是合适的。模板类 Stack 是从 QList 类 private 派生的。这样做的理由是：根据定义，栈是一种被限制成只能访问栈顶项的数据结构。从 QVector 类 public 派生的 QStack 类，为栈提供了一个所期望的 public 接口，而且还允许客户代码无限制地访问栈中的项，因为 QStack 类包含 QVector 类的全部 public 接口。Stack 类是从 QList 类 private 派生的，所以它的 public 接口限制客户代码对栈的访问，以与这种数据结构的定义相一致。

示例 22.10 src/privatederiv/stack.h

```
#ifndef _STACK_H_
#define _STACK_H_

#include <QList>

template<class T>
class Stack : private QList<T> {
public:
    bool isEmpty() const {
        return QList<T>::isEmpty();
    }
    T pop() {
        return takeFirst();
    }
    void push(const T& value) {
        prepend(value);
    }
    const T& top() const {
        return first();
    }
    int size() const {
```

```
        return QList<T>::size();
    }
    void clear() {
        QList<T>::clear();
    }
};
#endif
```


示例 22.11 表明，客户代码企图使用 Stack 的基类(QList)接口是不允许的。

示例 22.11　src/privatederiv/stack-test.cpp

```
#include "stack.h"
#include <QString>
#include <qstd.h>
using namespace qstd;

int main() {
    Stack<QString> strs;
    strs.push("hic");
    strs.push("haec");
    strs.push("hoc");
//  strs.removeAt(2);                                    1
    int n = strs.size();
    cout << n << " items in stack" << endl;
    for (int i = 0; i < n; ++i)
        cout << strs.pop() << endl;
}
```

1 错误：继承的 QList 方法是 private 类型的。

因此，当基类只用于实现目的时，private 派生提供了隐藏基类 public 接口的一种途径。对于 protected 派生又如何呢?

假设希望派生 XStack 类，它是来自于这个 Stack 类的一种特殊的栈。对于从 QList 类 private 派生的 Stack 类，不能在 XStack 的实现中使用任何 QList 的成员函数。如果在实现 XStack 时需要使用某些 QList 函数，则从 QList 派生 Stack 时，必须使用 protected 派生。Protected 派生使 QList 的 public 接口在 Stack 中成为 protected 接口。在内部，这使得派生自 Stack 的类能够使用继承的 QList protected 接口。

22.5　复习题

1. 什么是虚表?
2. 什么是多态类型?
3. 哪些类型的成员函数不会被继承？为什么?
4. 在什么情况下应使用虚析构函数?
5. 当从基类构造函数调用虚函数时会发生什么?
6. 什么是虚继承？它能用来解决什么问题?
7. 为什么要使用非 public 派生?

第三部分　编程作业

第 23 章　MP3 自动点唱机作业

第 23 章　MP3 自动点唱机作业

这一章中将会分阶段编写一个可作为 MP3 播放列表生成器和数据库管理器的主窗口程序。它将会搜索文件系统中的 MP3 歌曲，并据此生成和播放选择的 MP3 文件，同时该程序将允许基于存储在 ID3v2（元标记信息）中的数据进行过滤查询。

前提条件：要继续本章的作业，需要你看完第 13 章。

我们将要实施的功能是受到 Amarok，aTunes 这样的开源项目和 iTunes，Musicmatch Jukebox 这样的商业程序的启发。所有这些程序都提供了类似的功能，但用户界面却大不相同。还有很多创意也可以用到播放器上。其中一些用键盘或鼠标会一样简单。只用鼠标或只用键盘工作试试。

在`$QTDIR/examples/phonon/qmusicplayer`中可以找到一个媒体播放器示例，可以以此作为工作的起点。然后，在做下列作业的过程中，可以向它添加一些其他组件。到撰写这部分内容时为止，`qmusicplayer` 允许向播放列表中添加文件并播放计算机上的歌曲列表。

如图 23.1 所示，这是一个最简单的媒体播放器，有如下两个主要的组件。

- 播放器视图，它的构成如下。
  - 一个停靠区，用来显示用户当前正在播放的歌曲，同时提供一些控制，用来改变歌曲的声音大小和位置。
  - Play/Pause 和 Stop 按钮（如果还有 Next 和 Previous 按钮会更好一些）。
- 用作中央窗件的歌曲列表视图，用来显示一列歌曲。

图 23.1　示例的屏幕截图

这些组件中的每一个都是显示数据视图，建议使用单独的类来保存实际数据，并始终保持模型代码和视图代码的分离。

Qt 的 MultiMediaKit

另外还有一个库，Qt 的 MultiMediaKit[①]，可以用作 Qt 的一个附加组件(add-on)。该组件包含在 Qt 软件开发包(Qt Software Development Kit，SDK)中，可以从 Qt Mobility[②]中下载。Qt 的 MultiMediaKit 提供用于媒体播放器方面的高级 API，包括保存记录和访问原始声音矢量数据的功能。它提供播放列表和许多流行播放列表格式的读取、写入实现。如果打算复用这个库而不是使用 Phonon，更好的起点将是 Qt Mobility 中的 demos/player 示例。Qt Mobility 的一部分只能用在诸如手机这样的移动平台上，而 Qt 的 MultiMediaKit 则可以用于所有平台。

Qt 的 MultiMediaKit 仍处于开发之中，但在未来的 Qt 版本中将可能打算替换 Phonon。

截至 Qt 4.6，有一个称为 libqtmultimedia 的库，包含于 Qt 安装包中。它含有之前提到的 libqtmultimediakit 库的一个子集。截至 Qt 4.7，这个库都没有包含 M3u 播放列表类以及其他一些你想用于媒体播放器的类。

23.1　Phonon/MultiMediaKit 配置

第一步要确保有一个带 Phonon(或 Mobility 的 MultiMediaKit)的 Qt 版本，还要确保你的计算机系统带有正确的可以播放 MP3 文件的编解码器。如果当前可以在计算机上播放 MP3 文件，那么设置就完成了。否则，对于 Qt 4.7 来说，Qt SDK 已经为大多数平台提供了这两个库现成的二进制文件，它们似乎都可以很好地工作。

构建并运行你所选择的起始范例程序，以确保你已经恰当地安装了可用于该工程的 Qt。如果无法播放预期格式的媒体，可以参阅“安装 Phonon”一节中的文档[③]。

23.2　播放列表

1. 从起始范例开始并使其可以加载、保存和清空当前的播放列表。用于播放列表的模型应当保存在一个名称为 PlayListModel 的类中，该类(直接或间接)派生自 QAbstractItemModel。
   PlayListModel 应当为 MetaDataValue(或类似的)实例提供一个模型。
   不要像 Qt Phonon 中的例子那样使用 QTableWidget，而是对 QTableView 进行扩展并称其为 PlayListView。图 23.2 给出了这些类之间的关系。
   如果愿意，可以复用 MetaDataObject，DataObjectTableModel，SimpleDelegate 以及可以在 dist 中 libmetadata 处找到的其他类[④]。
2. 让应用程序在 QSettings 中记住当前播放列表的文件名、表列的宽度以及当用户退出时主窗口的大小/位置。

---

① 参见http://doc.qt.nokia.com/qtmobility/multimedia.html。

② 参见http://qt.nokia.com/products/qt-addons/mobility/。

③ 参见http://doc.qt.nokia.com/latest/phonon-overview.html#installing-phonon。

④ 参见http://www.distancecompsci.com/dist/。

3. 在启动时，恢复这些 QSettings 的值。
4. 实现以下播放器/播放列表的动作：后一个(next)、前一个(previous)、播放下一首(playNext,在当前歌曲播放完成之后)、添加文件(addFiles)、清空(clear)、另存为(save as)、保存(save)和打开(open)。把这些动作放到主菜单和播放列表视图的上下文菜单中。确保给定名称的动作可以真正地控制播放器。
5. (选作)：实现剪贴板和在表视图中进行多重选择的动作，即移除文件(removeFiles)、复制(copy)、剪切(cut)和粘贴到此处(paste here)。

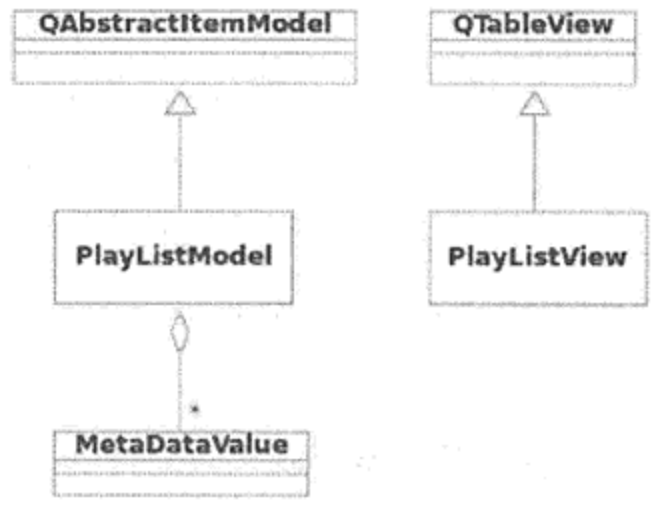


图 23.2 简单的播放列表

23.3 多种类型的播放列表

在这项作业中，会定义一些数据类型及加载和保存一些 M3U 格式播放列表的方式。

M3U 文件相当简单；它们每行都包括一个 URL，该 URL 可能是本地磁盘轨道上的一个相对路径。可选的扩展信息和注释后出现在以#打头的一行中。

之前的作业是从一个用作自我提高的示例性媒体播放器程序开始的。现在，可以向该程序添加一些加载/保存播放列表的动作。

Phonon 库有没有用来处理 M3U 文件的程序代码,所以需要自己编写一段序列化的代码。Qt Mobility 示例中有一个 QMediaPlaylist->load()方法，它可以载人 M3U 文件。

src/handouts 中提供了一些加载和向磁盘保存 M3U 文件的示例代码。可以随意复用这些代码。

MetaDataLoader 类使用了一些 dist 目录中的代码。有三种方式可以用：Phonon，taglib 或者 Qt Mobility。可以选择一种最适用于你的工作平台的方式。在 Linux 平台上，所有这三种方式都可以工作。截至本书出版时，无论是 Phonon 还是 Mobility，都不可以读取 Windows 上 ID3 格式的 MP3 文件，这就是为什么还要提供一个 Taglib 实现的原因。

如果想让播放器与其他播放器相兼容,在 http://help.mp3tag.de/en/main_tags.html 这个链接中所包含的信息[①]，可以帮助你确定流行的其他 MP3 播放器是使用哪一种标记来保存歌曲评分的。

用 QtTestLib 写一些测试用例，以验证可以读取和写人各种播放列表。

① 参见http://help.mp3tag.de/en/main_tags.html。

23.4　源选择器

如果能够从各种不同的源中选择歌曲，那么该播放器的功能将异常强大，也更为有用。所谓的源可能是下列中的任意一种：

- 一个播放列表。
- 一个 MP3 库，存储在一个数据库中。
- 一个在线电台的列表。
- CD 上的音轨。
- 一个音轨文件夹。
- 一个音乐库的子集，可通过流派、艺术家或者唱片进行分类过滤。

有趣的是，所有这些都有一个共同点：每种都可以看作一个 `QAbstractItemView`。

图 23.3 中，停靠在左侧的 `SourceSelector` 窗件允许用户选择当前的播放源，这种情况下，它对应的是一个许多歌曲的列表。

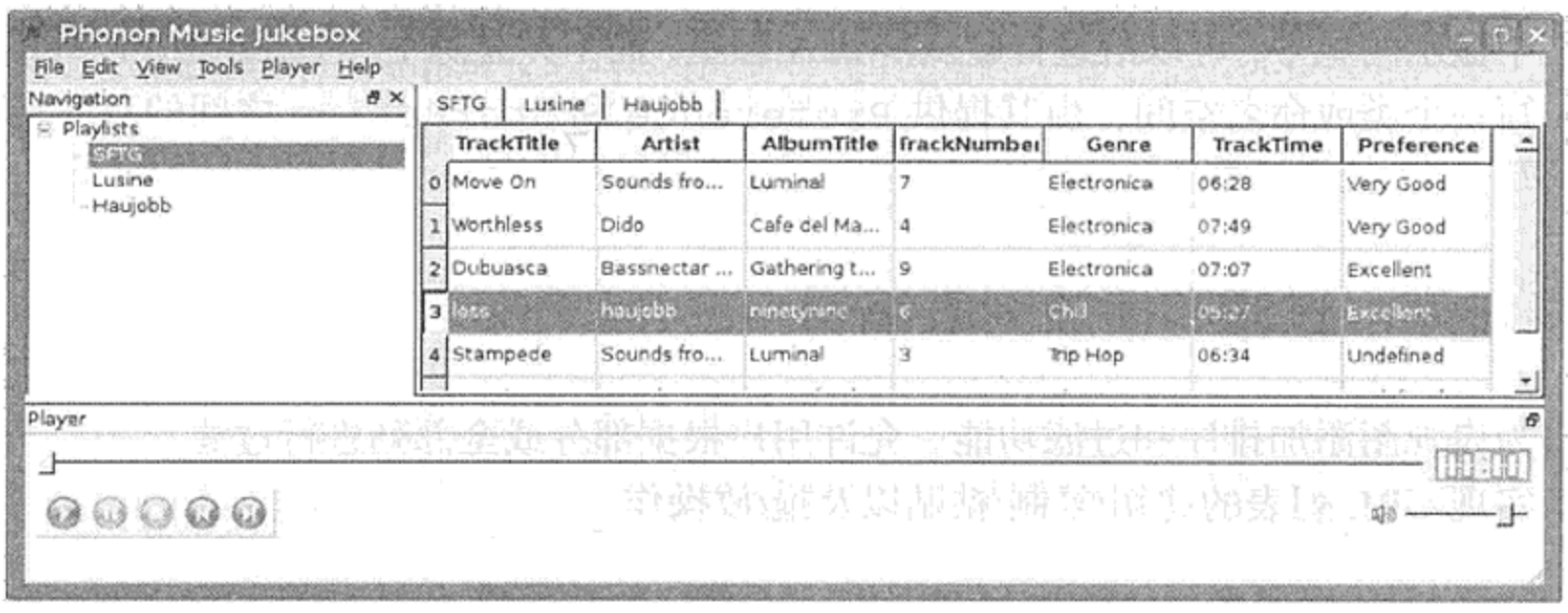


图 23.3　源选择器视图

任何时候，如果用户在窗件中单击另一个源，那么“当前播放列表”就应当切换到选中的那一个，且 MP3 播放器应当开始播放这个新的播放列表中的歌曲。

通过在选择器模型中单击一个源，应用程序应当改变目前可见的表或者中央窗件中的列表视图。图 23.4 给出了一种可能的方式来设计能够提供视图和一个源集合选择器的那些类。

在任何给定的时间，用户都可以执行一个动作，显示正在播放的(Show Now Playing)歌曲。它会改变中央窗件，以便可以显示当前正在播放的，而不是用户在源选择器中那个最后的选择。

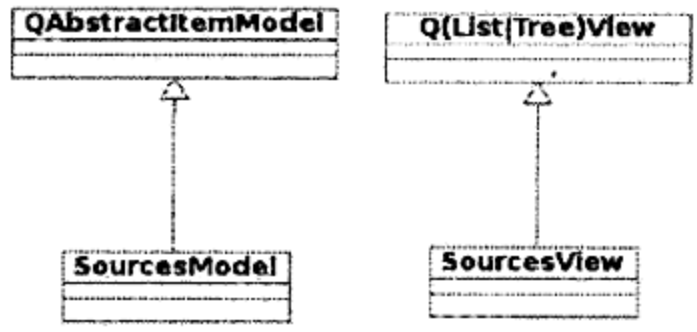


图 23.4　源选择器及其相关类

23.5 各播放列表数据库

要可以在数据库中加载和存储各个播放列表。如果不打算编写 SQL，可以复用 `$CPPLIBS/sqlmetadata` 中的代码。

- 添加一个队列和一个历史播放列表。
- 给放入队列和移出队列的音轨添加 `QAction`。
- 在播放列表视图中把播放位置显示成一个装饰条。
- 在播放一首曲目的同时，测试 next/previous 是否可以如预期的那样工作。

23.6 星号评分

1. 在 Qt 的例子中，有一个 stardelegate 的应用程序，它提供了一个用于 `StarRating` 的 `QItemDelegate` 实例。
2. 评价一首歌曲时，不要使用类似于“差”(bad)或者“优”(great)这样的词，用表格中显示的星号，可以给这首歌从零到 *N* 颗星，而不只是除 `Preference` 以外的评价。
3. 写一个类或命名空间，由其提供 `Preference` 和 `StarRating` 之间的双向转换。
4. 写一个 `StarDelegate`，让它用于 `Preference`。

23.7 排序，过滤和播放列表编辑

1. 为表视图添加排序和过滤功能。允许用户根据部分或全部列进行过滤。
2. 实现 URL 列表的剪切/复制/粘贴以及拖/放操作。

附录 A　C++的保留关键字

关键字是一种标志符，它是编程语言基本语法的一部分。关键字的名称具有固定的含义，不能用在不是这种固定含义的任何场合。

以下是 C++的关键字列表，其中以粗体显示的同时也为 ANSI C89 的关键字。

and	**extern**	**signed**
and_eq	FALSE	**sizeof**
asm	**float**	**static**
auto	**for**	static_cast
bitand	friend	**struct**
bitor	**goto**	**switch**
bool	**if**	template
break	inline	this
case	**int**	throw
catch	**long**	TRUE
char	mutable	try
class	namespace	**typedef**
compl	new	typeid
const	not	typename
const_cast	not_eq	**union**
continue	operator	**unsigned**
default	or	using
delete	or_eq	virtual
do	private	**void**
double	protected	**volatile**
dynamic_cast	public	wchar_t
else	**register**	**while**
enum	reinterpret_cast	xor
explicit	**return**	xor_eq
export	**short**	

附录 B 标准头文件

本书中用到了标准模板库(STL，也称为标准库)的一个小子集。书中所用的标准头文件在下面列出。为了有效地使用这些类和函数，知道去哪里查找文档是有用的。

表 B.1 中给出了一些标准头文件，其中有些头文件带有手册页，另外一些情况下，也可以找到某个函数的手册页。

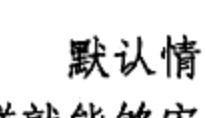
如果使用的是 Microsoft Developer 中的 Studio，则标准库的文档会随 MSDN 文档一起提供。

对于其他平台，尽管文档能够提供帮助，但是没有必要在本地保存手册页的副本，因为在 Web 上存在大量的文档副本[①]。

下面的这个表是通过运行 doxygen 而得到的标准头文件。

表 B.1 标准头文件

头 文 件	库	手 册 页
	C++ STL	
String	STL 字符串类型	std::string
sstream	stringstream，用于写入字符串流	std::stringstream
iostream	C++标准流库	std::ios, std::iostream
memory	C++中与内存相关的例程	std::bad_alloc, std::auto_ptr
	C 标准库	
cstring, string.h	用于 C char *字符串的函数	string, strcpy, strcmp
cstdlib, stdlib.h	C 标准库	random, srandom, getenv, setenv
cstdio, stdio.h	标准输入/输出	stdin, stdout, printf, scanf
cassert	断言宏	assert



默认情况下，C++标准库文档不会安装在系统中。用包管理器搜索字符串 `libstdc`，这样就能够安装某个文档，比如 `libstdc++6-4.5-doc`。

① 例如，可以查看 cplusplus.com (http://www.cplusplus.com/ref/) 或者 Dinkumware (http://www.dink umware.com/manuals/reader.aspx?lib=cpp)。

附录C 开发工具

本附录包含的中是一些有关C++开发环境的文章。

C.1 make和Makefile

以前曾讲过：不带参数的qmake能读取工程文件并构建出一个Makefile。示例C.1给出了对Makefile进行简要的回顾，该文件是从一个名称为qapp的简单工程中生成的。

示例C.1 src/qapp/Makefi le-abbreviated

```
# Exerpts from a makefile

####### Compiler, tools and options

CC            = gcc       # executable for C compiler
CXX           = g++       # executable  for c++ compiler
LINK          = g++       # executable for linker

# flags that get passed to the compiler
CFLAGS        = -pipe -g -Wall -W -D_REENTRANT $(DEFINES)
CXXFLAGS      = -pipe -g -Wall -W -D_REENTRANT $(DEFINES)
INCPATH       = -I/usr/local/qt/mkspecs/default -I. \
                -I$(QT4)/include/QtGui -I$(QT4)/include/QtCore \
                -I$(QT4)/include

# Linker flags
LIBS          = $(SUBLIBS) -L$(QT4)/lib -lQtCore_debug -lQtGui_debug -lpthread
LFLAGS        = -Wl,-rpath,$(QT4)/lib

# macros for performing other operations as part of build steps:
QMAKE         = /usr/local/qt/bin/qmake

####### Files

HEADERS       =     # If we had some, they'd be here.
SOURCES       = main.cpp
OBJECTS       = main.o
[snip]
QMAKE_TARGET  = qapp
DESTDIR       =
TARGET        = qapp  # default target to build

first: all            # to build "first," we must build "all"
```

```
####### Implicit rules

.SUFFIXES: .c .o .cpp .cc .cxx .C

.cpp.o:
        $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<

## Possible targets to build

all: Makefile $(TARGET)  # this is how to build "all"

$(TARGET):  $(OBJECTS) $(OBJMOC)     # this is how to build qapp
        $(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(OBJMOC) $(OBJCOMP) \
    $(LIBS)

        qmake:  FORCE                # "qmake" is a target, too!
        @$(QMAKE) -o Makefile qapp.pro  # what does it do?

dist:                                # Another target
        @mkdir -p .tmp/qapp \
    && $(COPY_FILE) --parents $(SOURCES) $(HEADERS) \
       $(FORMS) $(DIST) .tmp/qapp/ \
    &&  (cd `dirname .tmp/qapp` \ && $(TAR) qapp.tar qapp \
         && $(COMPRESS) qapp.tar) \
    && $(MOVE) `dirname .tmp/qapp`/qapp.tar.gz . \
    && $(DEL_FILE) -r .tmp/qapp

clean:compiler_clean                 # yet another target
        -$(DEL_FILE) $(OBJECTS)
        -$(DEL_FILE) *~ core *.core


####### Dependencies for implicit rules

main.o: main.cpp
```

make 命令会检查依赖并执行 Makefile 中指定的每一个构建步骤。最终结果的名称和位置可以用(.pro)工程中的变量 TARGET 和 target.path 设置。如果没有给定 TARGET，则会以工程文件所在位置的目录名会成为默认的 TARGET 名称。如果没有给定 target.path，则用工程文件所在的目录作为默认的位置。

清空文件

qmake 生成的 Makefile 文件使得 make 可执行若干个有用的命令，有时也称为“伪”(phony)目标[①]。这些命令中的第一个，make clean，会移除在构建或者执行工程的过程中可能产生的全部的目标文件和核心转储(core dump)文件[②]。

---

① 之所以称为“伪”，是为了和“真”目标文件进行区分。“真”目标通常是一些文件名，比如构建过程产生的可执行文件名。

② 核心转储文件有时是在执行程序遇到运行时错误并强制中断时产生的。它们给出了程序崩溃时主要内存的快照，或许可以用来确定造成崩溃的原因。

```
src/early-examples/example0> make clean
rm -f fac.o
rm -f *~ core *.core
src/early-examples/example0> ls
example0  example0.pro  fac.cpp  Makefile
src/early-examples/example0>
```


第二个命令，make distclean，除了会移除 make clean 移除的全部文件之外，还会移除 Makefile 和可执行文件，只剩下在其他机器上构建工程必须的那些文件。

```
src/early-examples/example0> make distclean
rm -f fac.o
rm -f *~ core *.core
rm -f example0
rm -f Makefile
src/early-examples/example0> ls
example0.pro  fac.cpp
src/early-examples/example0>
```

qmake 会在 Makefile 中生成其他的“伪”目标，但刚介绍过的那两个将会是最常用的。通过检查 Makefile 并查看那些标签，即以标示符开头且后跟一个冒号(比如，clean:)那些行，就可以熟悉其他的目标。要了解有关make 的更多信息，可以参阅作者推荐的[Rehman03]这本书。

注意

如果在 make 执行后又修改了项目文件，则应该在下次调用 make 前运行 qmake 以重新生成 Makefile。

提示

make dist 命令会创建一个包含工程文件所知的全部源文件的压缩包(dirname.tar.gz)。

C.2 用于#include 文件的预处理器

在 C++中，代码的复用是在源文件中由预处理器命令#include 来指明的。被包含的头文件中包含诸如类或命名空间的定义、常量的定义、函数原型等内容。这些文件在编译器开始编译代码之前就被逐字地包含到自己的文件中了。

只要编译器发现某个标志符被定义了不止一次，它就会报告错误。编译器可以容忍重复声明，但不能容忍重复定义[①]。为了防止重复定义，我们总是谨慎地用一个#ifdef 将每个头文件包起来。它会告诉 C 预处理器跳过已经出现过的内容。下面看一下示例 C.2 中的类定义。

示例 C.2 src/preprocessor/constraintmap.h

```
#ifndef CONSTRAINTMAP_H
#define CONSTRAINTMAP_H
```

① 20.1 节讨论了声明和定义之间的区别。

```
#include <QHash>
#include <QString>

class Constraint;                                                    1

class ConstraintMap : public QHash<QString, Constraint*> {           2

private:
    Constraint* m_Constraintptr;                                     3
    Constraint m_ConstraintObj;                                      4
    void addConstraint(Constraint& c);                               5
};
#endif        //  #ifndef CONSTRAINTMAP_H
```

1 前置声明。

2 需要 QHash 和 QString 的定义，但仅需要 Constraint 的声明，因为它是一个指针。

3 没问题，这只是一个指针。

4 错误：不完整的类型。

5 使用前置声明。

正如所看到的，在函数的参数列表中，可以使用指针或者引用指向那些仅声明没定义的类。指针的解引用和成员的访问操作是在示例 C.3 中给出的实现文件中进行的。这里，必须使用#include 包含出所用到的全部类型的完整定义。

示例 C.3 src/preprocessor/constraintmap.cpp

```
#include "constraintmap.h"

ConstraintMap map;                                                   1
#include "constraintmap.h"                                           2

Constraint* constraintP;                                             3

Constraint p;                                                        4
#include <constraint.h>
Constraint q;                                                        5

void ConstraintMap::addConstraint(Constraint& c) {
    cout << c.name();                                                6
}
```

1 没问题，已经包含了 ConstraintMap。

2 尽管多余，但只要已用#ifndef 包住就会成为多余但无害的东西。

3 从 constraintmap.h.那里使用前置声明。

4 错误：不完整类型。

5 现在，这是一个完整类型了。

6 这里需要完整类型。

为了使头文件之间“强依赖”(strong dependency)的头文件最少，应当尽可能多地使用类的声明而不是在只要需要的地方就使用#include 头文件。为了判断到底是需要一个前置声明还是需要将整个头文件都包含到头文件中，这里有几条指导原则：

- 如果 *ClassA* 派生自 *ClassB*，那么在处理 *ClassA* 的定义之前编译器必须知道 *ClassB* 的定义。这样，*ClassA* 的头文件中必须包含 *ClassB* 的头文件。
- 如果 *ClassA* 的定义中包含一个成员，而它是 *ClassD* 的对象，那么 *ClassA* 的头文件必须包含 *ClassD* 的头文件。如 *ClassA* 的定义包含一个函数，其参数之一或返回值是 *ClassD* 的对象，那么 *ClassA* 的头文件也必须包含 *ClassD* 的头文件。
- 如果 *ClassA* 的定义仅包含未被解引用的 *ClassE* 指针，那么在 *ClassA* 的头文件中有对 *ClassE* 的前置声明就足够了：`class ClassE;`。

只进行声明而没有进行定义的类称为不完整类型(incomplete type)。对不完整类型，试图解引用其指针或定义它的对象都会产生一个编译错误[①]。

ClassA 的实现文件 `classa.cpp` 应当既包含 `classa.h` 又包含 *ClassA* 所使用的所有类的头文件(除非该头文件已经包含进了 `class.h`)。所有的指针解引用都应当在 `.cpp` 文件中进行，这将会减小类之间的依赖性并且提高编译速度。

- 一个 `.cpp` 文件永远不应该包含另一个 `.cpp` 文件。
- 一个头文件应该尽可能少地包含其他头文件，这样它就能被快速地包含且只带有少量的依赖。
- 应该总是使用#ifdef 把头文件包起来，以防止它被多次包含。

循环依赖和双向关系

每当一个文件`#include` 另一个文件时，二者之间就会产生强依赖。当这样的依赖性出现在头文件之间时，它就不可能是双向的：预处理器没有能力处理头文件之间“彼此都包含对方”的这种循环依赖(circular dependency)。其中的某一个`#include` 语句必须替换成前置类声明。

前置声明有助于消除类之间的循环依赖，并且这个过程中允许它们存在双向关系。

C.3 了解链接器

图 C.1 显示了链接器如何接收由编译器生成的二进制文件并创建可执行二进制代码作为输出结果。链接器的可执行程序，在*nix 机器上称为 `ld`。在成功编译完所有的源文件之后，g++就会运行该命令。

上述所有的这些步骤都是在运行 `make` 命令时执行的，而 `make` 命令通常会在执行每条命令之前将其打印出来。通过阅读 `make` 的输出结果，就可以观察到哪些参数被传递给编译器和链接器。而当出现错误时，错误提示将会紧跟产生错误的代码行被打印出来。

示例 C.4 给出了传递给 g++的命令行选项，并且试图展示 g++如何运行链接器 `ld` 并给它传递一些参数。

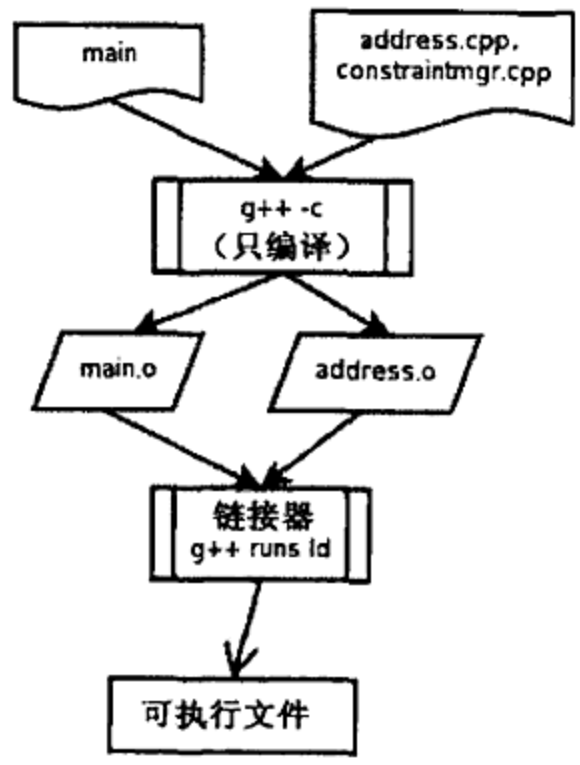


图 C.1 链接器的输入和输出

① 实际出错信息不一定总是很清晰，而且对于 `QObject`，错误可能源于 MOC 生成的代码而不是自己的代码。

示例 C.4 linker-invocation.txt

```
g++ -Wl,-rpath,/usr/local/qt-x11-free-3.2.3/lib                    1
    -o hw7                                                         2
    .obj/address.o .obj/ca_address.o .obj/constraintgroup.o
    .obj/customer.o .obj/dataobject.o .obj/dataobjectfactory.o
    .obj/hw07-demo.o .obj/us_address.o .obj/moc_address.o          3
    .obj/moc_ca_address.o .obj/moc_customer.o .obj/moc_dataobject.o
    .obj/moc_us_address.o
    -L/usr/local/qt-x11-free-3.2.3/lib  -L/usr/X11R6/lib
    -L/usr/local/utils/lib                                         4
    -lutils -lqt -lXext -lX11 -lm                                  5
```

1 告诉 g++运行链接器，并把这些选项传递给 ld。

2 指定输出文件的名称为 hw7。

3 把这些目标文件链接为可执行文件。

4 添加另外一个位置作为链接器搜索库的位置。

5 将此应用程序再与 4 个库进行链接：qt 工具库、ext、X11 和 m。

链接命令的详细步骤如下。

- 对于每个带有开关-l 的库名称，搜索库路径以及所有开关-L 的参数(由 qmake 通过指定 qmake 变量 LIB 来生成)，来找到它所对应的库文件。
  - 对于静态库，它包含了将被链接进可执行文件的二进制代码。
  - 对于动态库，它是一个目录列表(通常采用可读的 ASCII 格式)，其中描述了每个标签定义所对应的实际共享对象的位置。链接器将检查并确认那些共享对象确定位于正确的位置，否则就会报错。
- 对于正在链接的代码中调用的所有函数，找到其代码所在的对象并执行一个简单快速的检查，以确定在该位置确实存在一个定义完整且具有恰当名称/函数签名的函数。若找不到，或者名称/类型/大小有误，则报错。
- 对于变量名的每个引用，找到该变量所处的对象地址，并执行一个简单快速的检查，以确定这个地址对于该类型的对象来说是有效的。

总体的思路是：链接器解析对名称的引用，途径是查找它们在文件中的实际地址并检查该地址对该类型是否有效。这类似于为 C++编译器提供了一个字典查找服务。

C.3.1 常见的链接器错误信息

C++程序员经常花费大量时间来试图理解并修复编译和链接错误。如果不能理解那些消息，就无法继续。对于编译错误，往往可以很容易地诊断问题所在，因为它只与一个源码模块和它所包含的头文件的编译有关。编译器通常会告诉用户它所侦测到的每个错误的精确位置。而对于链接错误，问题涉及到源代码模块是如何链接到一起的。当到达链接阶段时，所有单个模块都已被无错地编译完毕。链接错误可能产生于 C++代码中的 bug，但也可能是工程文件问题所导致的。以下是四种错误类型。

1. Error: Unable to find libxxx.so.x

对于 Win32 用户

在创建过程中，IDE 需要能够找到.DLL 文件。要避免这种麻烦，研究一下菜单结构直到你找到 project->properties->c/c++build->libraries。在这里，你可以添加一个第三方的库，当然此时会有一个对话框来询问头文件和.DLL 文件的位置。

在运行时刻，系统环境变量 PATH 必须包含所需.DLL 文件所在的路径。

安装一个库意味着使其对系统上不只一个用户可用。我们有可能复用一个库而不去安装它，复用的所有库或者必须进行安装，或者必须位于 LD_LIBRARY_PATH 所指定的目录中。

当第一次复用一个库时，可能会看到这条出错的消息，它说明链接器无法找到库。当 gnu 链接器查找共享对象时，它至少会检查两个地方：

1. LD_LIBRARY_PATH 指定的目录。
2. 名称为/etc/ld.so.cache 的 cache 文件中列出的已安装库。

Cache 文件：ld.so.cache

cache 文件对在/etc/ld.so.conf 指定的目录中发现的共享对象提供快速查找。在此处能够发现的目录包括：

```
/lib
/usr/lib
/usr/X11R6/lib
/usr/i486-linuxlibc1/lib
/usr/local/lib
/usr/lib/mozilla
```

如果使用一个 Linux 包安装程序来安装一个库，它将会对 Id.so.conf 做出恰当的改动并重建 cache 文件。然而如果是手动编译并安装库，那么必须修改这个文件。之后可以通过 ldconfig 命令来重建 cache 文件。

2. undefined reference to *identifier*

这是最常见的也几乎是最讨厌的链接错误。它的意思是链接器无法找到代码中某个命令实体的定义。下面是 make 命令的一些输出。

```
.obj/ca_address.o(.gnu.linkonce.t._ZN10DataObject16getConstraintGroupEv+0x4):
In function `DataObject::getConstraintGroup()':
/usr/local/qt-x11-free-3.2.3/include/qshared.h:50:
undefined reference to `DataObject::s_Cm'
collect2: ld returned 1 exit status
make: *** [hw7] Error 1
```

编译器找到了声明，但链接器却找不到相应的定义。在代码中的某部分引用了一个符号，但是找不到其定义。下面是一些有用的信息：

- 它找不到的那个符号是 DataObject::sm_Cm。
- 试图使用它的函数是 DataObject::getConstraintmgr。

解决问题的第一步是确认，你能否找到那个丢失的定义。如果无法找到，那么链接器如何可以呢？如果在.cpp 文件中找到了它，那么你必须确认下面的两件事情。

- .cpp 和.h 文件都在工程中。

- 这个文件被我们正在连接的某个库包含。

由于正在使用良好的命名约定(参见 3.1 节),随意马上可以断定 sm_Cm 是 DataObject 类的一个静态数据成员。编译器发现了声明,但链接器无法找到定义。

而由于它是静态的(参见 2.9 节),sm_Cm 的定义隶属于 dataobject.cpp。编译器期望找到一个如下形式的定义语句:

```
ConstraintGroup DataObject::s_Cm;
```

如果它在那里,但链接器仍旧无法找到它,那么最有可能的原因是:

- 包含该定义的.cpp 文件没有在.project 文件中的 qmake 的 SOURCES 中列出。
- 代码位于其他库中但链接器无法找到那个库。这时可以通过在工程文件中添加一个遗漏的 LIBS 参数来解决问题。
- -lmyLib 添加一个要链接的库。
- -LmyLibDir 向链接器的库搜索路径列表中添加一个目录。

3. Error: unresolved external symbol

当使用 Microsoft 编译器链接自己的库时,可能会出现这样的错误:

```
customer.obj : error LNK2001: unresolved external symbol
"public: virtual bool __thiscall DataObject::readFrom(class QObject const &)"
(?readFrom@DataObject@@UAE_NABVQObject@@@Z)
```

这通常意味着,在构建 DLL 时该符号未被导出。确保在该类或者函数声明之前有一个 Q_DECL_EXPORT 宏,然后再次构建该 DLL。参见 7.1.2 节。

4. Error: undefined reference to vtable for ClassName

这是众多最令人费解的错误之一,它通常表示缺少一个虚函数定义。从字面意思来说,就是该类的 vtable(它存有每个虚函数的地址)无法完整地创建。

代码中缺少了函数定义就会产生这种错误,而在make/project文件的HEADERS或SOURCES中缺少某一项也会导致这种问题。如果你最近在某个已存在的头文件中添加了 Q_OBJECT 宏,那么需要重新运行 qmake,因为 Makefile 需要重新生成。这方面的更多细节请参阅 8.4 节。

提示

在这类链接器错误后,首先检查所有文件是否都恰当地列在了工程文件中。所有的 QObject 都有一个 Q_OBJECT 宏,试着执行命令 qmake && make clean,然后看看是否还会生成这个错误。

完全内联类(All-Inline Classes)

对于多态类,在与头文件相对应的源程序文件(.cpp)[①]中至少要有一个非内联的定义(函数或静态成员)。如果没有,则很多链接器将无法找到它的任何一个虚方法定义并报出类似的错误。

完全内联类在 C++中是合法的,但是当与多态混用时它将不能按预想的方式工作。

① 多态是指拥有至少一个虚方法的类。

C.4 调试

编译器可以定位并描述出语法上的错误，而链接器可以揭示出程序组件之间的不一致并有助于我们正确地定位它们。在 C++中最具有挑战性的方面是学习如何找到并修复各种各样的运行时错误。

运行时错误(run-time errors)是在一个语法正确且不包含未定义对象和函数的程序逻辑错误。通过有效地使用调试器(一个专门设计的、用于追踪运行时错误的程序)，能够极大减少花在处理这些错误上的时间。

一个调试器允许单步地执行代码，还允许查看对象值。由于调试器是工作在编译后的代码之上的，早期的版本只能由熟悉汇编语言的程序员使用。现在的调试器能够同时在编译后的机器代码和原始程序代码上面工作。GNU 的开发工具家族包含了 gdb，这是一个可以用于C/C++程序的源码级 GNU 调试器。gdb 设计有一个命令行接口，这个接口非常强大但对用户不甚友好。幸运的是，gdb 有了一些开源的图形界面外壳，我们下面会讨论其中的一个。商业的 C++ IDE(如 Visual Studio)通常有内建的源码级调试器。

C.4.1 建立一个可调试的目标

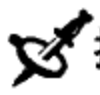
为了让 gdb 能够工作，就必须在编译时把调试符号创建到代码当中，否则机器指令将无法正确映射到 C++原文件中的对应位置上。我们可以在调用编译器时使用适当的命令行开关(-g)来简单地完成这件工作：

```
g++ -g filename.cpp
```

这通常会生成一个尺寸大得多的可执行文件。一般来说，这个尺寸增长是与源代码的尺寸和复杂程度成正比的。这个扩展了的可执行文件包含符号表信息，调试器可以用它来找到与机器码对应的源代码。要使得 qmake 生成的 makefile 开启调试选项，在 qmake 工程文件中加入下面一行：

```
CONFIG += debug
```

当创建带有调试符号的 Qt 库时，就可以像操作自己的代码一样单步调试 Qt 的源代码。有些程序会包含由 Qt 库直接进行调用的代码，在调试这些特定的程序时，可能需要创建带有调试符号的 Qt 库。



提示

在 Win32 中，这是一个可以单击的菜单选项。而在*nix 平台上，首先解开源代码的 tar 压缩文件，然后在创建之前给 configure 脚本传递一个参数，这样在创建 Qt 库时就会包含调试符号了。

```
./configure --enable-debug
make
make install
```

C.4.1.1 练习：建立一个可调试的目标

为了更好地理解调试的代价：

- 比较在工程文件中有和没有 CONFIG+=debug 的情况下创建的可执行文件大小。
- 记得稍候后用一个更复杂的应用再试一下。

C.4.2 gdb 入门

设想你正在运行一个程序，但不知什么原因，程序崩溃了。

```
[lazarus] app> ./playlistmgr
Segmentation fault
[lazarus] app>
```

当应用程序中止或崩溃时，如果精确地知道(尽可能快地)事故在哪里发生将是很有用的。我们可以使用 gdb 来快速简单地定位故障点。下面是一个 gdb 命令行会话的例子。

```
[lazarus] app> gdb playlistmgr
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
This GDB was configured as "i386-linux"...Using host libthread_db library "/lib/
tls/libthread_db.so.1".

(gdb)  r①
Starting program: ftgui/app/playlistmgr
[Thread debugging using libthread_db enabled]
[New Thread -1227622176 (LWP 17021)]
Qt: gdb: -nograb added to command-line options.
         Use the -dograb option to enforce grabbing.
This is a debug message

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread -1227622176 (LWP 17021)]
0xb7f03320 in FormDialog::createActions (this=0x80ae2a0) at formdialog.cpp:53
53          delete m_OkAction;
(gdb)
```

gdb 不仅可以向用户显示文件名和行号，还可以显示相应的源代码行。然而，我们可能仍然想要得到这个错误的上下文。list 命令显示当前文件中该错误周围的一些代码：

```
(gdb) list
51      void FormDialog::createActions() {
52
53          delete m_OkAction;
54          delete m_CancelAction;
55          m_OkAction = new OkAction(m_Model, m_View);
56          m_CancelAction = new CancelAction(m_Model, m_View);
57          QHBoxLayout *buttons = new QHBoxLayout(0);
(gdb)
```

where 命令的作用是显示栈的内容或者显示如何到达该处。

```
(gdb) where
#0  0xb7f03320 in FormDialog::createActions (this=0x80ae2a0) at formdialog.cpp:53
#1  0xb7f03058 in FormDialog::setModel (this=0x80ae2a0, fmodel=0x80c80d0)
    at formdialog.cpp:34
```

---

① r 表示的是“运行”命令。

```
#2  0x080664bd in SettingsDialog (this=0x80ae2a0, parent=0x0) at settingsdialog.
cpp:14
#3  0x0805f313 in MainWindow (this=0xbfffdec8) at mainwindow.cpp:42
#4  0x08066f14 in Controller (this=0xbfffdec0, argc=1, argv=0xbfffdfe4) at
controller.cpp:25
#5  0x0805a8a4 in main (argc=1, argv=0xbfffdfe4) at main.cpp:7
(gdb)
```


诸如 Eclipse 和 QtCreator 这样的多数开源 IDE 在后台使用的都是 gdb，它们分别提供了一个用户接口，以使某些功能更易于学习和使用。ccdebug[①]是使用 Qt 编写的，且专门设计用于调试 Qt 应用程序，并提供 QString 支持。

提示

在一些调试器内 QString 是很难看到的，因为它们是间接指向 Unicode 数据的指针。调试器要能正确显示 QString 的内容需要知道额外的信息。

如果你使用的是安装了带有 Qt 集成工具的 IDE，应当可以在调试器内看到 QString。

如果使用的是 QtCreator 并且无法看到 QString，可以到 Qt settings 中，检查是否可以为你所使用的 Qt 构建调试帮助。

如果正在使用的是 gdb 命令行，可以下载 kde-devel-gdb[②]，这是 KDE subversion 仓库中的一个 Qt 帮助宏集合，然后把以下内容放在~/.gdbinit 中。

```
source /path/to/kde/kde-devel-gdb
define pqs
    printq4string $arg0
end
```

现在应该就能够在调试时使用 pqs 宏来显示 QString 了。

提示

可以使用.gdbinit 文件自动加载一个可执行文件，设置一些断点，并开始调试任务。一旦发现在运行 gdb 时会重复输入相同的命令，可以试着将该命令序列加到工程所在目录下的.gdbinit 文件。

C.4.3 查找内存错误

若没有运行时分析工具的帮助内存错误是非常难以跟踪的。用来分析程序运行性能的程序称为 profiler(分析器)。Valgrind 是一个开源的 Linux 分析工具，用于跟踪代码的内存和 CPU 使用率并检测各种错误。这些错误包括：

- 内存泄漏(memory leak)——不可再被使用却又没有被删除的内存。
- 堆内存的非法指针使用，比如
  - 索引越界
  - 分配和释放的语法不匹配(例如，用 new[]分配却用 delete 释放)

① 参见http://sourceforge.net/projects/ccdebug。

② 参见http://websvn.kde.org/*checkout*/trunk/KDE/kdesdk/scripts/kde-devel-gdb。

- 使用未初始化的内存

刚才列出的每个错误都能在一小段程序中产生灾难性的结果。分析器也能够用于性能的调试以及确定哪些代码对拖慢程序的速度负有责任(即找速度瓶颈)。

示例 C.5 的程序包含了一个有意为之的内存使用错误。

示例 C.5 src/debugging/wrongdelete.cpp

```
void badpointer1(int* ip, int n) {
  ip = new int[n];
  delete ip;                                    1
}

int main() {
  int* iptr;
  int num(4);
  badpointer1(iptr, num);
}
```

1 错误的删除语法。

为了使输出结果更易读，我们加入了调试符号(-g)。

```
debugging/wrongdelete> g++ -g -Wall wrongdelete.cpp
debugging/wrongdelete> ./a.out
debugging/wrongdelete>
```

编译器并没有报错，而且甚至是在程序运行以后，也没表现出错误行为。尽管如此，该程序破坏了内存。

让我们看一下 valgrind 对我们程序的分析(略为删减)，我们去掉了每行开头的 valgrind 作业的进程 id。这个进程 id 在每次运行 valgrind 时都会不同。

```
src/debugging> valgrind a.out
--3332-- DWARF2 CFI reader: unhandled CFI instruction 0:50
--3332-- DWARF2 CFI reader: unhandled CFI instruction 0:50
 Mismatched free() / delete / delete []
    at 0x401C1CB: operator delete(void*) (vg_replace_malloc.c:246)
    by 0x80484BD: badpointer1(int*, int) (wrongdelete.cpp:3)
    by 0x80484F4: main (wrongdelete.cpp:9)
  Address 0x4277028 is 0 bytes inside a block of size 16 alloc'd
    at 0x401BBF4: operator new[](unsigned) (vg_replace_malloc.c:197)
    by 0x80484AC: badpointer1(int*, int) (wrongdelete.cpp:2)
    by 0x80484F4: main (wrongdelete.cpp:9)
```

valgrind 发现了错误，且根据调试符号就能够指出问题代码的位置。示例 C-6 稍微有趣一些，因为它存在内存泄漏和数组索引错误。

示例 C.6 src/debugging/valgrind-test.cpp

```
#include <iostream>

int badpointer2(int k) {
  int* ip = new int[3];
  ip[0] = k;
```

```
  return ip[3];
}


int main() {
  using namespace std;
  int* iptr;
  int num(4), k;
  cout << iptr[num-1] << endl;
  cout << badpointer2(k) << endl;
}
```

1
2
3
4
5
6

1 越界的索引。

2 内存泄漏：分配的内存已不再可用。

3 iptr 没有初始化。

4 k 没有初始化。

5 iptr 的状态是什么？

6 向函数发送了未初始化的 arg。

用 valgrind 运行示例 C.6 得出了一些错误的精确位置：

```
 For more details, rerun with: -v

--2164-- DWARF2 CFI reader: unhandled CFI instruction 0:50
--2164-- DWARF2 CFI reader: unhandled CFI instruction 0:50
 Use of uninitialised value of size 4
    at 0x80486AF: main (valgrind-test.cpp:17)
68500558
 Invalid read of size 4
    at 0x804867C: badpointer2(int) (valgrind-test.cpp:8)
    by 0x80486DD: main (valgrind-test.cpp:18)
  Address 0x4277034 is 0 bytes after a block of size 12 alloc'd
    at 0x401BBF4: operator new[](unsigned) (vg_replace_malloc.c:197)
    by 0x8048667: badpointer2(int) (valgrind-test.cpp:6)
    by 0x80486DD: main (valgrind-test.cpp:18)
0

ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 19 from 1)
malloc/free: in use at exit: 12 bytes in 1 blocks.
malloc/free: 1 allocs, 0 frees, 12 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 1 not-freed blocks.
checked 120,048 bytes.

LEAK SUMMARY:
   definitely lost: 12 bytes in 1 blocks.
     possibly lost: 0 bytes in 0 blocks.
   still reachable: 0 bytes in 0 blocks.
        suppressed: 0 bytes in 0 blocks.
Use --leak-check=full to see details of leaked memory.
```

如果根据这些信息还不足够找到内存在哪泄漏，可以重新运行 valgrind 并使用 --leak-check=full 开关。在示例 C.7 中，我们修复了其中的一些错误。

示例 C.7 src/debugging/valgrind-test2.cpp

```
#include <iostream>

int notSoBadPointer(int k) {
  int* ip = new int[3];
  ip[0] = k;
  delete[] ip;                                    1
  return k;                                       2
}


int main() {
  using namespace std;
  int* iptr;                                      3
  int num(4), k(4);                               4
  cout << iptr[num-1] << endl;                    5
  cout << notSoBadPointer(k) << endl;
}
```

1 清除内存泄漏。

2 一个可返回值。

3 未初始化的指针!

4. 至少 k 不再未初始化。

5. 这里有问题!

在不使用 valgrind 的情况下，编译并运行这个稍经修正的测试程序不会产生警告或者错误且输出了一个没有意义的值:

```
src/debugging> g++ -g -Wall valgrind-test2.cpp
src/debugging> ./a.out
-1078391036
4
```

使用 valgrind 运行它生成的错误比先前少一些:

```
src/debugging> valgrind ./a.out
 For more details, rerun with: -v

 Use of uninitialised value of size 4
    at 0x8048794: main (valgrind-test2.cpp:18)
-1096641724
4

 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 18 from 1)
 malloc/free: in use at exit: 0 bytes in 0 blocks.
 malloc/free: 1 allocs, 1 frees, 12 bytes allocated.
 For counts of detected errors, rerun with: -v
 All heap blocks were freed -- no leaks are possible.
src/debugging>
```

最后，在示例 C.8 中，我们清除了最后一个错误。

示例 C.8 src/debugging/valgrind-test3.cpp

```
#include <iostream>

int notSoBadPointer(int k) {
  int* ip = new int[3];
  ip[0] = k;
  delete[] ip;                                    1
  return k;                                       2
}

int main() {
  using namespace std;
  int num(4), k(4);                               3
  int* iptr = new int[num] ;                      4
  for (int i = 0; i < num; ++i)
     iptr[i] = i;
  cout << iptr[num-1] << endl;                    5
  cout << notSoBadPointer(k) << endl;
  delete[] iptr;
}
```

1 清除内存泄漏。

2 一个可返回值。

3 至少 k 不再是未初始化。

4 没有未初始化指针。

5 不再有问题了。

编译、运行并再次用 valgrind 运行：

```
src/debugging> g++ -g -Wall valgrind-test3.cpp
src/debugging> ./a.out
3
4
src/debugging> valgrind ./a.out
3
4
 ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 18 from 1)
 malloc/free: in use at exit: 0 bytes in 0 blocks.
 malloc/free: 2 allocs, 2 frees, 28 bytes allocated.
 For counts of detected errors, rerun with: -v
 All heap blocks were freed -- no leaks are possible.
src/debugging>
```

在 Mac OS X 下，Valgrind 尚无法使用，[①]但是在 Mac Developer Tools 中包含了一个称为 MallocDebug.app 的图形化工具，它可以代替 valgrind 的一些功能。

C.5 开源开发工具、库和集成开发环境

下面是一些构建与 Qt 之上的开源库，提供了一些额外的可复用组件：

① 可以用源代码进行安装，源代码可以在 http://valgrind.org/downloads/repository.html 中找到。

- Qwt[①]——用于科技类应用程序的 Qt(Qt Widgets for Technical Applications)。
- Qxt[②]——Qt 扩展库(Qt eXTension Library)。

C.5.1 集成开发环境

使用普通的文本编辑器进行面向对象开发是不太实际的。面向对象开发工作涉及到众多的类以及各种其他的文件(头文件和源代码文件)。在编辑窗口中写代码只是开发进程中的一部分，一个优秀的程序编辑器或者 IDE(集成开发环境)应当支持下列众多特性：

- 每个文件中对象/成员的树状结构导航。
- 重构(Refactoring)帮助，用于成员的移动和改名。
- 集成的调试器。
- 上下文敏感的链接到 API 文档的帮助。
- 内建的命令行外壳窗口，这样就能在不离开环境的情况下运行程序。
- 一个工程管理器，用来管理相关文件分组和子分组。
- 可用于其他语言的编辑模式。
- 易用的键盘制定能力，可以让任何键击都可以执行任何一项功能(尤其是光标移动，当然也包括窗口的移动)。
- 开放的插件构架，以便添加其他组件。
- 若能集成版本控制功能则更好，尤其是在窗口环境中。可参看 Subversion[③]或其他更高级的分布式版本控制系统，如 bzr, mercurial, git, monotone 或者 darcs。
- 易学习、可编码的宏。
- 根据语言特性在不同文件之间跳转(有查找声明、查找定义、查找引用等快捷方式)。

C.5.2 开源集成开发环境

Nokia/Qt Software 已发布其自由、开源的 QtCreator[④]，现在已包含在 Qt SDK 中。QtCreator 是一个用 Qt 和 C++ 编写的 IDE。它为 qmake/cmake 工程、上下文自动补全、代码导航、重构和集成化代码调试等提供完整支持。我们认为，无论是对初学者还是专家来说，它都是一个在所有主流平台上开发基于 Qt 的 C++编程集成开发环境的理想 IDE。

在使用 C++/Qt 开发前，Mac OS X 的用户或许需要先安装有 XCode。

对于所有平台，还有一个基于 Java 的自由、开源集成开发环境：Eclipse[⑤]。诺基亚提供了一个 Eclipse 集成软件包[⑥]，提供了许多功能，使 Eclipse 成为一个适用于 Qt 开发的集成开发环境。首先需要安装 C/C++开发工具(C/C++ Development Tools，CDT)或者一些用于 C++开发的插件[⑦]。Qt 集成工具使得你可以直接将 qmake 的.pro 文件导入成 Eclipse 的工程文件，并可将设计师的可停靠窗件停靠在 Eclipse 的主窗口中，以便可以使用设计师的全部功能而无须离开 Eclipse。此外，将 Qt API 的文档集成到 Eclipse 的上下文敏感帮助系统中也是可行的。

---

① 参见http://qwt.sourceforge.net/。
② 参见http://docs.libqxt.org/index.html。
③ 参见http://subversion.tigris.org/。
④ 参见http://qt.nokia.com/products/developer-tools。
⑤ 参见http://www.eclipse.org。
⑥ 参见http://qt.nokia.com/developer/eclipse-integration。
⑦ 参见http://www.eclipse.org/cdt/。

C.5.3 UML 建模工具

为了能够使用统一建模语言(UML)创建本书中的图表，我们使用了两个开源工具：Umbrello[①]和 Dia[②]。这两个工具都使用某种 XML 方言(XML dialect)作为其原生文件格式。

如图 C.2 所示，Umbrello 是 KDE 中的 UML 建模器(Modeler)——它可以直接导入 C++ 代码，这使得拖拽导入类到图表中变得非常容易。

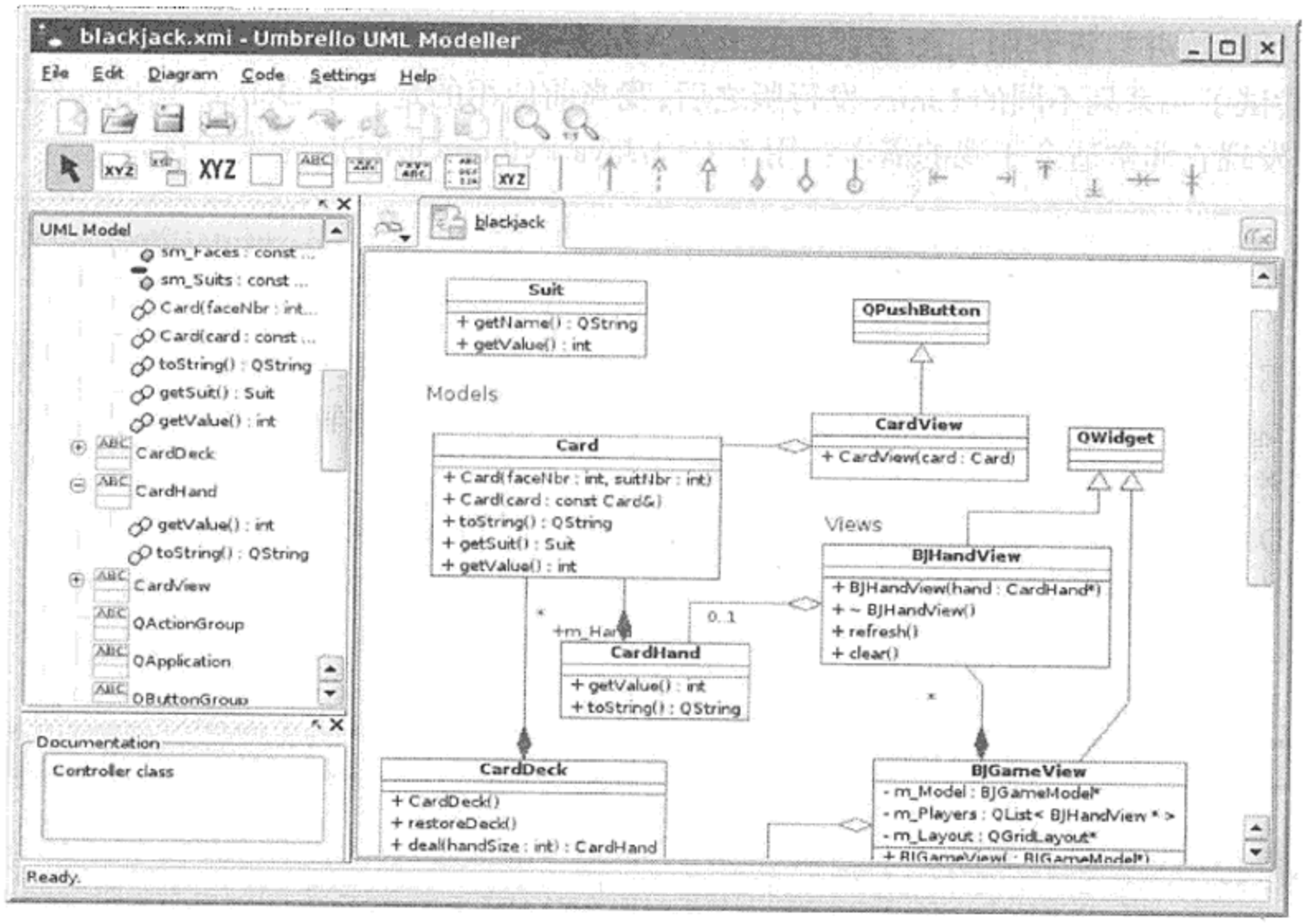


图 C.2 Umbrello 的屏幕截图

Dia 是一个 Gnome 实用工具，它是一个带有某些 UML 特性的通用性更强的图表工具。有很多插件和实用工具可用于往/从 Dia 中导入代码、导出图形到其他语言和格式。

① 参见http://uml.sourceforge.net/index.php/。

② 参见http://www.gnome.org/projects/dia/。

附录 D　Alan 的 Debian 程序员快速指南

这篇文章总结了我用来建立用于开发工作的新 Debian 桌面的步骤。这对不熟悉 Debian Linux 桌面环境并打算利用 apt 系统的新开发人员来说会非常有用。

在尝试了一系列不同的 Linux 发行版之后，我当前使用的是一个 Debian Testing KDE 安装工具[①]，发现它非常适合于我的需求：用于 C++/Java/Python/Qt/KDE 开发工作。

问题：为什么要用 Debian

1. 可用软件包的数量庞大的令人觉得可笑——许多软件包都难以从其他发行版中找到。这就意味着，你可以大把抓到几乎每一个库已编译的二进制软件包，而不是你自己从源代码开始构建。
2. Debian 测试版(Debain testing)似乎包含了最新的 Qt 和 KDE 软件包。
3. 许多 KDE/Qt 开发人员都是在 Debian 或者 Kubuntu 中完成工作的。
4. Debian 和 Ubuntu 的发行版提供了一个强大的名称为 apt 的软件包管理系统，它用起来相当容易，而且在出现问题时可以很轻松地进行修复。

问题：为什么要用 KDE

KDE 是这本书的两位作者最喜爱的桌面环境。我们两个都使用和喜欢 KDE 的桌面环境许多工具，而不是许多其他桌面环境(例如，GNOME)中相类似的应用程序。除了设计精良和用户环境友好之外，它还拥有自身是用 Qt 和 C++编写的这一优势。

推荐基于 Debian 的 Linux 发行版

以下给出的各发行版都有一些重要的共同点：它们是可自引导的，这就意味着可以从 CD 或 USB 记忆棒中的映射直接引导你的计算机并运行 Linux，而无须安装任何东西到你的硬盘中。但你也可以用同一个 CD 或 USB 记忆棒中在你的硬盘上安装一份完整的基于 Debian 的 Linux。举荐的所有这些发行版都是自由(免费)的，可以很容易地从各种网站下载到：

1. Debian[②]一直是过去几年中主要的几个“基本“发行版之一。最初，Debian 的卓越性能是公认的，但对于初学者来说，它的安装太困难了。其他几个发行版基本上都是对 Debian 的简化，提供简单、用户友好的安装脚本。硬件检测和识别一直是安装系统时主要的难点领域之一。Debian 的优势之一(可以自定义安装，以便可以准确适用于硬

① KDE 是一个桌面和一系列工具集，与基于 Gtk 的 Gnome 不同，该工具集是完全用 Qt 编写的。

② 参见http://www.debian.org。

件)也是它的主要缺点(安装系统时必须知道如何找到硬件驱动程序)。Debian 的近期版本已经有了更为友好的安装脚本。可以下载一个“netinst” ISO 映像，将其刻录到 CD 或者 USB 记忆棒中，从它启动，并只需一点点儿工作即可安装到硬盘中。推荐使用 debian.org[①]中最新的 debian-live-KDE-desktop.iso CD 映像安装包，因为它是以 liveCD 的形式启动并可安装一个最新的 KDE 桌面。

2. Ubuntu[②]可能是本书出版之时最为流行的基于 GNOME 的发行版。
3. Kubuntu[③](基于 KDE 的 Ubuntu)和 Ubuntu 类似，但它是基于 KDE 的，而不是基于 GNOME 的。

问题

稳定？测试？不稳定？

这听起来很吓人。为什么会有人愿意使用其他的而不是稳定的操作系统？许多 Debian 新手都会被用来形容任意 Linux 发行版的“不稳定”一词所吓倒。因为其他两个可用的选项是“测试”和“稳定”，人们自然会尽量先去试用稳定版。遗憾的是，稳定版中提供的大多数应用程序和库都已经相当陈旧(至少一年，有时甚至更久远)，还有一些你最喜欢的软件包可能尚不可用。对于桌面/开发系统来说，可能更倾向于使用较新的软件。“测试”是一个处于不稳定和稳定之间的妥协产物。通常情况下，Debian 测试版的安装程序也支持更新一些的硬件。在开发过程中，会给 Debian 的版本取些昵称，这些名称源于自玩具总动员(Toy Story)电影系列[④]。

稳定版(三眼仔，squeeze)是当前的稳定版本。它包括了 Qt 4.6。这些可用的版本都经过了彻底的测试。

测试版(企鹅吱吱，wheezy，2011 年)目前正在使用 Qt 4.7，并含有稳定版中没有但将被包含在下一个主要发行版中的版本。

不稳定版(席德，sid)含有比测试版更多的更新版本，根据决定升级系统的不同时间，或许会发现需要自己修复那些无法安装的软件包(broken package)。只有在使用 apt 系统相当顺手以后才建议你使用“席德”版。

D.1　apt 系统

apt 系统是一个用来管理软件包及其依赖关系的工具。下列程序可以用来管理基于 apt 的软件库：

- apt-get，一个让获得和升级软件包变成轻松事情的简便程序。
- dpkg，软件的低层。许多简单的 apt 命令都会被转换成更为复杂的命令并由 dpkg 加以执行。
- apt-cache，一个帮助你在本地 apt 软件包信息数据库中搜索的工具。

① 参见http://cdimage.debian.org/debian-cd/current-live。
② 参见http://www.ubuntu.com/。
③ 参见http://www.kubuntu.org/。
④ 文章 http://en.wikipedia.org/wiki/Debian 较为详细地介绍了这一点。

- aptitude，一个稍微智能些的 apt-get。当 apt 无效时，它可以用来获得和移除软件包。这是 dpkg 之上的另一个简便层。

以下是一些有用的提示和关于 apt 系统以及如何使用 apt 的基本知识。

etc/apt/sources.list

这个文件包含了一个 apt 用来检查软件包的资源列表。在安装新软件之后，应当对这个资源列表做两件事情：

修改主镜像(main mirror)文件，以便让它们指向本地镜像[①]而不是指向远程镜像。实现这一点可以有很多方法，但最为用户友好的方式是运行 apt-setup。

可选项：如果打算让发行版中的每样东西都是最新的，可以把来源设置成“不稳定”(Debian)或者“edgy”(Kubuntu)[②]。edgy 是 Kubuntu 中大约 6 年前的一个老版本。

apt-get 更新会从软件来源中下载软件列表，以便可以有一个清单、依赖关系和自己 dpkg 数据库说明的本地副本。

有时，可能需要某个软件包但没有记住它的具体名称。命令 apt-cache search *search-string* 会对本地已下载的软件包清单进行搜索，从每个软件包的名称或者描述中查找该字符串的出现地方。如果返回的清单中含有所查找的软件包，就可以从网络上把它轻松拉取下来并予以安装。

一旦知道了软件包的名称，就可以使用命令 apt-cache show *pkgname* 阅读它的完整说明。

apt-get install *packageName* 是功能最强大的命令，在基于 Debian 的系统中，只有拥有 root 权限才可以使用该命令。

apt-get remove *packageName*，顾名思义，会完全移除给定名称的软件包。

为了节约时间，下面列出了一些在 Debian 系统上进行 C++开发时建议安装的软件包。注意插入注释的“#”的用法。

```
# alias for following commands:
alias agi='apt-get install -y'
# Database Stuff
agi mysql-server mysql-client libmysqlclient15-dev
agi sqlite3 libsqlite3-dev

# For development
agi build-essential manpages-dev # manual pages for stdlib
agi global cscope exuberant-ctags # For C++ development - navigation and doc
generation
agi libqt4-dev qt4-dev-tools libphonon-dev
agi gstreamer0.10-fluendo-mp3 # for playing mp3 files in phonon
agi libtag1-dev libtag1-doc # taglib used by libfiletagger
agi gdb # gnu debugger
agi umbrello # UML Diagramming tool that reads/writes XMI and imports C++ source
```

① 参见http://www.debian.org/mirror/list。

② 仅 Ubuntu：确保在 main 之后加一个“universe”，以便可以获得由 Ubuntu 提供的完整软件包。

更多的 apt 中提示

apt-get (dist-|dselect-) upgrade

随着时间的流逝，会有新的安装包出现在你的软件仓库中。在你打算升级系统时，执行一下 dist-upgrade 会很合适。为了给 apt 赋予权限以移除那些陈旧软件包并替换成更新的软件包，可以使用 apt-get dselect-upgrade 命令（这个命令与 aptitude upgrade 命令相似但不完全一样）。

apt-get source packageName

除非需要的是一个软件包源无法提供的特定版本，只需通过简单请求 apt 即可方便地获得任何可以获得的软件包的源代码压缩包。

我刚安装了一样东西。它跑哪儿去了？

对于这个命令，需要深入研究一下 dpkg。某种意义上说，dpkg 是一个功能强大的工具，但大多数情况下，我们只是借助 apt-get 而间接用到它。dpkg -L *packageName* 是其中一个很有用的选项，这个命令会列出从这个软件包中提取的全部文件并给出它们在系统中的当前位置。

apt-get build-dep *packageName*

有时，在编译大的软件包（如 Qt）时，一些库（或者它们的开发包）的缺失会造成构建的失败。当我还在以自己的方式研究我的第一个基于 Debian 系统时，我是通过以下这种暴力性质的重复过程来构建应用程序和库的：配置（configure），遇到并检查每一个错误信息，试着找出遗漏了什么库，安装它，继续重复直到没有错误为止。在我得知 apt-get build-dep 之后，即刻意识到我原本可以节约那么多的时间，这是多么微不足道的一件事情啊！

你已经知道 apt 系统可以知道哪个软件包会依赖另外哪一个软件包，但使用 apt-get build-dep *packageName* 命令，apt 就会自动下载所有那些依赖的库及其开发包，以便可以拥有构建 *packageName* 所需的全部东西。换句话说，再也不需要去跟踪是哪些头文件被遗漏了！

为了能够看到各个软件包之间的依赖关系，可以试一下命令 apt-cache showpkg *packageName*。

aptitude：在 apt 系统无法使用时

有时，在试图安装东西时遇到了错误，但此时 apt 也已经处于一种无法使用的状态了。在运行非稳定版时，这非常常见，在使用测试版时会少一些。或许，你会打算试一下 apt-get -f install，但试验后，还是需要试着移除那些让人不舒服的软件包，但你还得继续忍着。

仔细阅读错误信息很重要：绝大多数情况下，你会看到一些指向造成这一问题的具体软件包的名称。通过在同一行内使用 apt-get remove *pkg1 pkg2 pkg3* 移除它们，有时可以把系统重新带回可用的状态。

修复这一问题的另一种方法，是对每个软件包都使用 aptitude remove pkgN。aptitude 可以帮助你找到并移除造成问题的相关软件包集。

D.2 update-alternatives

因为在 Debian 系统上可同时存在同一个程序的多个替代版本(尤其是诸如 qmake 和 java 这样的程序)，会有一些从/usr/bin 到这些程序所需版本的符号链接。为了管理这些链接并将其修改指向所需的版本，root 可以使用 update-alternatives 命令。

例如，为了选择要运行的 qmake 的默认版本，root 可以使用

```
update-alternatives --config qmake
```

为了查看每个已安装的有可替换版本的程序清单，root 可以键入命令

```
update-alternatives --all
```

并在所有的替换版本中选中默认版本。

附录 E　C++/Qt 配置

为充分用好这本书，需要把 Qt 4.6 或更新的 Qt 版本以及 C++编译器安装在你的计算机上。以下各节介绍了我们所推荐的在三大桌面平台上安装 Qt 的流程。

E.1　C++/Qt 配置：开源平台

在大多数 UNIX 的派生平台上，都自带了开源开发工具(ssh, bash, gcc)[①]。

在讨论那些基于 UNIX 派生的平台(Linux, BSD, Solaris 等)时，我们会以*nix 来简称“大致类似于 UNIX 的操作系统”。

另外一个重要的缩略语是 POSIX，表示 Portable Operating System Interface for UNIX，即 UNIX 可移植操作系统接口。这个应用程序接口(API)系列标准是由 IEEE(Institute of Electrical and Electronics Engineers，美国电气与电子工程师学会)倡导的。IEEE 是一个面向工程师、科学家和学生的组织，以制定计算机和电子工业的开发标准而著称[②]。

这一节主要面向在计算机上安装并使用*nix 操作系统的读者。

针对本书设置计算机的第一步是要确认系统中完全安装了 Qt。这不仅包括源代码和编译库的代码，还包括 Qt 帮助文档系统、编程示例以及 QtCreator 程序。

> **Qt SDK**
>
> Qt SDK 包括 Qt 库、帮助程序、设计师以及 QtCreator，可用特殊的编译器进行构建。建议在选定的开发平台上使用快速安装的方式。

要查看在计算机系统中安装的是哪一个版本的 Qt(如果有的话)，可以用命令

```
which qmake
qmake -v
```

第一条命令的输出会显示可执行文件 qmake 在系统中的位置。如果输出类似于“bash: qmake: command not found”，则可能的情况是：

1. Qt 没有“完全”安装(包括开发工具)。
2. 安装了 Qt，但 PATH 环境变量中没有包含/path/to/qt/bin 路径。
3. 是通过诸如 qmake-qt4 这样的安装包管理器安装的，以避免与 Qt 3 中的可执行文件名冲突。

如果该命令可以运行，那么 qmake -v 会提供版本信息。如果报告信息是“Using Qt version 4.x.y”，那么就可以使用类似的命令分别查看是否其他的 Qt 工具也是可用的：moc, uic, assistant, designer 和 qtcreator。

---

① 在 Mac OS X 上，为了获得 C++编译器和 make 工具，需要安装 Xcode。

② 如果打算写一个 POSIX 正则表达式(参见 14.3 节)，对于*nix 来说，看起来会是这个样子的：(lin|mac-os|un|solar|ir|ultr|ai|hp)[iu]?[sx]。

如果所有这些可执行命令都可以找到并且版本匹配，那么就说明已安装了 Qt，并且可以使用了。

如果这些测试表明你安装了早前的版本，或者根本没有安装 Qt，再或者在安装 Qt 未安装某些组件，那么无论是哪种情况，都需要构建或者安装 Qt 的最新发行版，并把 Qt 的可执行文件放到默认路径中。

从安装包安装 Qt

使用*nix 系统的安装包管理器(例如，apt，zypper，aptitude，kpackage，synaptic，yum 等)，可以轻松快捷地从包含 Qt 的安装包进行安装。只是需要记住的是，Qt 有可能会被分隔成许多小块，而需要的则可能是它们所有。以下是可用于 2011 年 Debian wheezy 系统的 Qt 4.7 的安装包清单：

```
[ROOT@lazarus]# apt-cache search qt4
libqt4-assistant - transitional package for Qt 4 assistant module
libqt4-core - trans pkg for Qt 4 core non-GUI runtime libraries
libqt4-dbg - Qt 4 library debugging symbols
libqt4-dbus - Qt 4 D-Bus module
libqt4-declarative - Qt 4 Declarative module
libqt4-declarative-folderlistmodel - Qt 4 folderlistmodel QML plugin
libqt4-declarative-gestures - Qt 4 gestures QML plugin
libqt4-declarative-particles - Qt 4 particles QML plugin
libqt4-designer - Qt 4 designer module
libqt4-dev - Qt 4 development files
libqt4-gui - transitional package for Qt 4 GUI runtime libraries
libqt4-help - Qt 4 help module
libqt4-network - Qt 4 network module
libqt4-opengl - Qt 4 OpenGL module
libqt4-opengl-dev - Qt 4 OpenGL library development files
libqt4-phonon - Qt 4 Phonon module
libqt4-qt3support - Qt 3 compatibility library for Qt 4
libqt4-script - Qt 4 script module
libqt4-scripttools - Qt 4 script tools module
libqt4-sql - Qt 4 SQL module
libqt4-sql-ibase - Qt 4 InterBase/FireBird database driver
libqt4-sql-mysql - Qt 4 MySQL database driver
libqt4-sql-odbc - Qt 4 ODBC database driver
libqt4-sql-psql - Qt 4 PostgreSQL database driver
libqt4-sql-sqlite - Qt 4 SQLite 3 database driver
libqt4-sql-sqlite2 - Qt 4 SQLite 2 database driver
libqt4-sql-tds - Qt 4 FreeTDS database driver
libqt4-svg - Qt 4 SVG module
libqt4-test - Qt 4 test module
libqt4-webkit - transitional package for Qt 4 WebKit module
libqt4-webkit-dbg - trans pkg for Qt 4 WebKit debugging symbols
libqt4-xml - Qt 4 XML module
libqt4-xmlpatterns - Qt 4 XML patterns module
libqt4-xmlpatterns-dbg - Qt 4 XML patterns library debugging symbols
qt4-demos - Qt 4 examples and demos
```

```
qt4-demos-dbg - Qt 4 examples and demos debugging symbols
qt4-designer - graphical designer for Qt 4 applications
qt4-dev-tools - Qt 4 development tools
qt4-doc - Qt 4 API documentation
qt4-doc-html - Qt 4 API documentation (HTML format)
qt4-qmake - Qt 4 qmake Makefile generator tool
qt4-qmlviewer - Qt 4 QML viewer
qt4-qtconfig - Qt 4 configuration tool
[ROOT@lazarus]#
```

正如上面所示，在 Debian 中，Qt 4 被划分成多个独立的安装包，这样开发者可以在配置时获得更好的灵活性。在开发时，需要全部的 Qt 4 安装包，如果打算使用 Qt 发行版进行程序调试，就尤其需要-core，-dev，-doc，-dev-tools，-designer 和用于调试的-dbg 符号。

从源代码安装

可以从 qt.nokia.com[①]下载最新的源文件 tarball，然后进行解压缩和编译。如果有 Qt SDK，可以运行 Updater 并选择 Qt Sources 来将其下载到 Qt SDK 目录。要确保也选择了 demos 和 examples。

提示

在 Debian 中，在另外编译一个 Debian 安装包后，也是有可能用一个简单的命令即可自动安装所有的工具和库。当打算从源代码编译任何流行的开源工具时，可以充分利用这一点。更多详细信息，可以参阅附录 D。这个命令为：

```
apt-get build-dep libqt4-dev
```

注意

tarball 是利用 tar 命令(tape archive，磁带归档的缩写)生成的一个文件，为了便于存储和传送，一般使用 tar 命令把多个文件结合生成一个文件(扩展名通常是.tar)。得到的文件往往再使用诸如 gzip 或者 bzip2 等压缩工具进行压缩，形成一个扩展名为.gz 或者.bz 的压缩文件。

tarball 的解压命令取决于其建立的方式。一般可以通过文件的扩展名来进行判断：

```
tar -vxf  whatever.tar                  //使用verbose开关
tar -zxf  whatever.tar.gz               //使用gzip压缩
tar -zxf  whatever.tgz                  //同样也是使用gzip压缩
tar -jxf  whatever.tar.bz2              //使用bzip2压缩
```

tar 文件可以保留目录结构，同时也有很多选择和开关，可以通过输入下面的命令来阅读在线文档：

```
info tar
info gzip
info bzip
```

① 参见http://qt.nokia.com/downloads。

Qt 的源文件 tarball 包含完整的 Qt 库源文件，外加无数的例子、教程和完整的参考文献。该 tarball 包含简单的安装命令(在 README 和 INSTALL 文件中)以及一个 configure -help 帮助信息。从源文件 tarball 安装软件时，一定要仔细阅读 README 文件。

从源文件进行编译可能要花费 2~3 小时的时间(根据系统的速度)，但花费这个时间是值得的。示例 E.1 给出了配置 Qt 4.7 时的选项。

示例 E.1 /bin/qtconfi gure

```
#!/bin/sh
# specify -phonon if you want to build the audiojukebox exercise or
any of the Phonon examples.
# replace username with your username, and Qt473 with your version of
Qt
./configure -phonon -fast -prefix /home/username/Trolltech/Qt473
```

在 Qt 配置后，键入 make 编译，然后再安装它。

提示

如果有一颗四核处理器，可以试着键入

```
make -j 4
```

这样编译器即可同时运行四个编译进程，可以充分利用额外的其他处理器核。

在最后一步中，make install 会把可执行文件和头文件从 tarball 解包后的目录复制到另外的一个位置。如果在一个公共位置进行安装，则必须以 root 身份来执行这一步。

检查 Qt 的安装

提示

在安装之后，键入 qmake -v 命令来确定 shell 中能够找到的是哪个版本的 qmake。对于安装有多个版本的系统，这是务必要做的一步。

```
[ezust@stan] /home/ezust> which qmake
 which qmake
/usr/local/Trolltech/QtSDK/Desktop/Qt/473/gcc/bin/qmake
[ezust@cerberus] /home/ezust> qmake -version
QMake version 2.01a
Using Qt version 4.7.3
   in /usr/local/Trolltech/QtSDK/Desktop/Qt/473/gcc/lib
```

环境变量

安装完成后，检查一下环境变量，确保 PATH 中含有指向已安装 Qt 的合适的引用。

提示

使用 bash 命令 env 可以显示目前系统中的所有环境变量。7.2 节讨论并使用过环境变量。

示例 E.2 中的脚本文件给出了如何利用 bash 命令设置环境变量，但实际值需取决于系统中文件的具体位置。

示例 E.2　/bin/qt.sh

```
# Using the Qt SDK 1.1
# None of the variables below are required by Qt
# I just like having variables pointing to these locations for easy access:
export QTSDK=/opt/QtSDK
export QTCREATOR=$QTSDK/QtCreator
export QTDIR=$QTSDK/Desktop/Qt/473/gcc
export QTSRC=$QTSDK/QtSources/4.7.3

# make sure SDK's qmake and qtcreator are found first in the path:
export PATH=$QTDIR/bin:$QTCREATOR/bin:$PATH

# Location of your shared libraries:
export CPPLIBS=~/cs331/projects/libs

# Where to search for shared object libraries at runtime:
export LD_LIBRARY_PATH=$CPPLIBS
```

E.2　C++/Qt 配置：Win32

有两个 Qt 版本可用于 Win32 平台：

1. mingw[①]版，编译需使用 MinGW（Minimalist Gnu for Windows，Windows Gnu 最小包）中的 g++，可以下载并可自由用于开源项目中。
2. Microsoft Visual C++版，编译需使用诸如 Microsoft Visual Studio 2008 这样的 Microsoft 编译器。

无论安装的是哪个版本的 Qt，都很容易：Win32 安装程序会引导你完成包括注册扩展名和设置环境变量在内的整个过程。

提示

如果还没有编译器或者安装 IDE，那么可以安装 Qt SDK，在安装包中含有各个库、mingw 编译器和 QtCreator。

Qt 安装后，还需要单击 Start->Programs->Qt by Nokia->Build debug symbols。这可能需要花上几个小时。

然后，通过单击 Start->Programs->Qt by Nokia->Command Prompt 打开一个 shell 窗口。

现在，可以在这个命令提示符控制台中运行 qmake -v 来查看当前安装的 Qt 的版本。此时，qmake, assistant, designer, qtdemo, qtconfig, g++和 make 都应当在系统的搜索路径中。也可以试一下“开始”菜单中的 qtdemo。

① 参见http://www.mingw.org/。

E.3 C++/Qt 配置：Mac OS X

1. 从 Developer.apple.com 安装 xcode[①]。
2. 从 qt.nokia.com 安装 qtsdk[②]。
3. 从之前安装的 qtsdk 那里运行 qtcreator。

qmake 的特别之处

- qmake 默认创建 xcode 工程而不是利用 makefile。
- 使用 qmake -spec macx-g++生成一个命令行型的 Makefile。
- 在.pro 文件中放一个 CONFIG -= app_bundle，以免在 Mac OS 的应用程序目录结果中生成可执行文件。
- 如果使用 fink 安装的 make，请阅读帮助文档中有关如何适当重新编译/设置 Qt 环境部分的内容。

① 参见 developer.apple.com。
② 参见 qt.nokia.com。

参 考 文 献

C++参考文献

[Stroustrup97] *The C++ Programming* Language.Special Edition. Bjarne Stroustrup 1997. Addison Wesley. 0-201-70073-5

[Meyers] *Effective* C++. Scott Meyers. 1999-2005. Addison Wesley Professional Software Series. 0321334876.

[Morris06] "The C++ Interpreter Pattern for Grammar Management" Stephen Morris. 2006. informit.com.

Qt 参考文献

[Blanchette08] *C++ GUI Programming with Qt 4*. Jasmin Blanchette and Mark Summerfield. Second Edition. 2008 Prentice Hall.

[Summer11] *Advanced Qt Programming*. Mark Summerfield. 2011. Prentice Hall.

[Thelin07] *Foundations of Qt Development*. Johan Thelin. 2007. Apress.com.

[Molkentin06] *The Book of Qt 4*. Daniel Molkentin 2006. No Starch Press.

[kdestyle] "KDE Style Guidelines." Coding Style.kde.org①.

[qtapistyle] "Designing Qt-Style C++ APIs." Matthias Ettrich. 2005. Trolltech②.

[qtapistyle] "Qt Coding Style" Community Project. 2010. Trolltech③.

[qttestlib] "Writing Unittests for Qt 4 and KDE4 with QtTestLib." Brad Hards. 2005. developer.kde.org④.

OOP 参考文献

[Buschmann96] *Pattern-Oriented Software Architecture*. First Edition. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. John Wiley & Sons. 0-471-95869-7.

[Fowler04] *UML Distilled*. Third Edition. Martin Fowler. 2004. Addison Wesley. 0-321-19368-7.

[Gamma95] *Design Patterns： Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Addison-Wesley 0-201-63361-2.

[Koenig95] *Patterns and Antipatterns*. Andrew Koenig. 1995. Journal of Object-Oriented Programming 8(1) pgs 46-48.

[Martin98] *Pattern Languages of Program Design 3* . Robert C. Martin, Dirk Riehle, and Frank Buschmann. 1998. Addison-Wesley. pgs 293-312. 0-201-31011-2.

---

① http://techbase.kde.org/Policies/Kdelibs_Coding_Style.

② http://doc.trolltech.com/qq/qq13-apis.html.

③ http://qt.gitorious.org/qt/pages/QtCodingStyle.

④ http://developer.kde.org/documentation/tutorials/writingunittests/writingunittests.html.

Docbook 参考文献

[docbook] *Docbook: The Definitive Guide*. Norman Walsh. 2006. O'reilly Associates①.

[docbookxsl] *Docbook XSL: The Complete Guide*. Bob Stayton. 2005. SageHill Enterprises②.

其他参考文献

[xhtmlw3c] "w3c Recommendation: XHTML 1.0 The Extensible HyperText Markup Language."2005. W3C (World Wide Web Consortium)③.

[dist] The web accessible site from which you can download example source code and other useful resources: http://informit.com/title/9780132826457.

[Friedl98] *Mastering Regular Expressions*. Second Edition. Jeffrey Friedl. 1998. O'Reilly. 1-56592-257-3.

[Rehman03] *The Linux Development Platform* . Rafeeq Ur Rehman. Christopher Paul. 2003. Prentice Hall. 0-13-009115-4.

[Guzdial07] *Introduction to Computing & Programming with Java, A Multimedia Approach*. Mark Guzdial and Barbara Ericson. 2007. Prentice Hall. 0-13-149698-0.

① http://www.docbook.org/tdg/en/html/docbook.html.

② http://www.sagehill.net/docbookxsl/

③ http://www.w3.org/TR/xhtml1/