目 录

第一章 LINUX 图形用户界面 GUI 介绍 1

1.1 常用 GUI 介绍 1
1.2 关于 Qt 2
1.3 Qt/Embedded 简介 4
1.4 Qtopia 介绍 5

第二章 QT 的安装 6

2.1 Qt X11 的安装 6
2.2 Qt/Embedded 安装 9
2.3 Qtopia 编译 11

第三章 QT 常用工具的介绍 12

4.1 Qt 设计器（Qt Designer） 12
4.2 Tmake 14
4.3 Qvfb 的使用和安装 15
附录. QT 实验环境的建立 17

第四章 QT 的编程 12

实验一 “Helloworld！” Qt 初探 19
实验二 创建一个窗口并添加按钮 24
实验三 对象间通信：Signal 和 Slot 机制 27
实验四 菜单和快捷键 36
实验五 工具条和状态栏 45
实验六 鼠标和键盘事件 55
实验七 对话框 70
实验八 Qt 中的绘图 82
实验九 Qt 中的多线程编程 92
实验十 Qt 中的网络编程 104

第一章 Linux 图形用户界面 GUI 介绍

所谓 GUI(Graphics User Interface)，就是图形用户界面。图形用户界面的广泛流行是当今计算机技术的重大成就之一，它极大地方便了非专业用户的使用，人们不再需要死记硬背大量的命令，而可以通过窗口、菜单方便地操作。它的主要特征有三点：

- ✧ WIMP。其中，W（Windows）指窗口，是用户或系统的一个工作区域。一个屏幕上可以有多个窗口。I（Icons）指图图标，是形象化的图形标志，易于人们隐喻和理解。M（Menu）指菜单，可供用户选择的功能提示。P（Pointing Devices）指鼠标等，便于用户直接对屏幕对象进行操作。
- ✧ 用户模型。GUI 采用了不少 Desktop 桌面办公的隐喻，让使用者共享一个直观的界面框架。由于人们熟悉办公桌的情况，因而对计算机显示的图标的含义容易理解，诸如：文件夹、收件箱、画笔、工作簿、钥匙及时钟等。
- ✧ 直接操作。过去的界面不仅需要记忆大量命令，而且需要指定操作对象的位置，如行号、空格数、X 及 Y 的坐标等。采用 GUI 后，用户可直接对屏幕上的对象进行操作，如拖动、删除、插入以及放大和旋转等。用户执行操作后，屏幕能立即给出反馈信息或结果，称为所见即所得（What You See Is What You Get，WYSIWYG）。用视、点（鼠标）代替了记、击（键盘），给用户带来了方便。

通常所见的 GUI 都是位于 PC 机上的，但是在 PC 上 GUI 并不适合嵌入式系统。嵌入式设备有严格的资源要求（比如十分有限的存储空间）。同时嵌入式系统经常有一些特殊的要求，而普通的 PC 上的图形窗口系统是不能满足这些要求的。比如特殊的外观效果，要提供给用户的函数，提高装载速度，特殊的底层图形或输入设备。由此可见嵌入式系统必定要有自己的 GUI。

综上所述，嵌入式 GUI 就是在嵌入式系统中为特定的硬件设备或环境而设计的图形用户界面系统。所以嵌入式 GUI 不但要具有以上有关 GUI 的特征，而且在实际应用中，嵌入式系统对它来说还有如下的基本要求：

- ✧ 轻型，占用源少；
- ✧ 高性能；
- ✧ 高可靠性；
- ✧ 可配置性。

下面我们就目前市场上常用的嵌入式 GUI 做一个简单的介绍。

1.1 常用 GUI 介绍

1．MiniGUI

由北京飞漫软件技术有限公司开发的 MiniGUI(http://www.minigui.org),是国内为数不多的几大国际知名自由软件之一。MiniGUI 是面向实时嵌入式系统的轻量级图形用户界面支持系统，1999 年初遵循 GPL 条款发布第一个版本以来，已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、彩票机、金融终端等产品和领域。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/uClinux、eCos、uC/OS-II、VxWorks、等操作系统上运行；已验证的硬件平台包括 Intel x86、ARM（ARM7/AMR9/StrongARM/xScale）、PowerPC、MIPS、M68K（DragonBall/ColdFire）等等。

MiniGUI 良好的体系结构及优化的图形接口，可确保最快的图形绘制速度。在设计之

初，就充分考虑到了实时嵌入式系统的特点，针对多窗口环境下的图形绘制开展了大量的研究及开发，优化了 MiniGUI 的图形绘制性能及资源占有。MiniGUI 在大量实际系统中的应用，尤其在工业控制系统的应用，证明 MiniGUI 具有非常好的性能。

2．MicroWindows

MicroWindows（http://microwindows.censoft.com）是一个开放源码的项目，目前由美国 Century Software 公司主持开发。该项目的开发一度非常活跃，国内也有人参与了其中的开发，并编写了 GB2312 等字符集的支持。但在 Qt/Embedded 发布以来，该项目变得不太活跃，并长时间停留在 0.89Pre7 版本。可以说，以开放源码形势发展的 MicroWindows 项目，基本停滞。

MicroWindows 是一个基于典型客户/服务器体系结构的 GUI 系统，基本分为三层。最底层是面向图形输出和键盘、鼠标或触摸屏的驱动程序；中间层提供底层硬件的抽象接口，并进行窗口管理；最高层分别提供兼容于 X Window 和 Windows CE（Win32 子集）的 API。

该项目的主要特色在于提供了类似 X 的客户/服务器体系结构，并提供了相对完善的图形功能，包括一些高级的功能，比如 Alpha 混合，三维支持，TrueType 字体支持等。

MicroWindows 采用 MPL（mozilla public license）条款发布。

3．DinX

DinX 非常适合于在很小的系统上运行多窗口程序，它简单、轻巧，并且快速。DinX 并不是 X，它使用 Linux 核心的 framebuffer 视频驱动，采用 Client/Server 模式。为此，系统提供了两个界面：/dev/dinxsvr 和/dev/dinxwin。

一个服务器程序连接到/dev/dinxsvr，并决定来自各程序窗口的 request 各占有视屏的各个部分。它也负责给各窗口发送像鼠标移动这样的事件消息。Clinet 程序连接到/dev/dinxwin，与 Server 进行消息通信等。Server 进程还负责处理事件、窗口管理、调色板配置等功能。DinX 是一个实验性的窗口系统，它处在发展阶段中，还存在一些缺陷和问题。DinX 的 license 属于 MPL，也可以转化为 GPL。这样，DinX 核心模块可以集成到 Linux 中，DinX 库可以链接到其他的 GPL 程序中。

4．OpenGUI

OpenGUI（http://www.tutok.sk/fastgl/）在 Linux 系统上存在已经很长时间了。最初的名字叫 FastGL，只支持 256 色的线性显存模式，但目前也支持其他显示模式，并且支持多种操作系统平台，比如 MS-DOS、QNX 和 Linux 等等，不过目前只支持 x86 硬件平台。OpenGUI 也分为三层。最低层是由汇编语言编写的快速图形引擎；中间层提供了图形绘制 API，包括线条、矩形、圆弧等，并且兼容于 Borland 的 BGI API。第三层用 C++编写，提供了完整的 GUI 对象集。

OpenGUI 采用 LGPL 条款发布。OpenGUI 比较适合于基于 x86 平台的实时系统，可移植性稍差，目前的发展也基本停滞。

5．Qt/Embedded

Qt/Embedded 是著名的 Qt 库开放商开发的面向嵌入式系统的 Qt 版本。这个版本的主要特点是可移植性较好，许多基于 Qt 的 X Window 程序可以非常方便地移植到嵌入式版本。在下面我们将会详细介绍。

1.2 关于 Qt

Qt是一个跨平台的C++图形用户界面库，由挪威TrollTech公司(www.trolltech.com)出品，它的目的是提供开发应用程序用户界面部分所需要的一切，主要通过汇集C++类的形式来实现这一目的。它提供给应用程序开发者建立艺术级的图形用户界面所需的所用功能。Qt是完

全面向对象的，很容易扩展的，并且允许真正地组件编程的GUI开发工具。

Qt是Trolltech公司的一个标志性产品。Trolltech公司1994年成立于挪威，但是公司的核心开发团队已经在1992 年开始了Qt产品的研发，并于1995年推出了Qt的第一个商业版，直到现在Qt已经被世界各地的跨平台软件开发人员使用，而Qt的功能也得到了不断的完善和提高。

QT/X11和QTE（QT Embedded）是它其中的两个版本。Qt/X11是基于X Windows系统的Qt版本，KDE便是基于它来构建的。为了适用于嵌入式系统，该公司将Qt/X11进行了裁减，发布了QTE（QT Embedded）版本。QTE直接基于Linux中的FrameBuffer设备，删除了Qt/X11中一些对资源要求很高的类实现。所以，基于QTE实现的应用，不作修改重新编译后，就可以在Qt/X11上运行，而反过来便不可以。

QPE（Qt Plamtop Environment）是Trolltech公司所推出的针对PDA软件的整体解决方案，包含了从底层的GUI系统、Window Manager、Soft Keyboard到上层的PIM、浏览器、多媒体等方面。目前QPE的高版本已更名为Qtopia，其包含了更多功能。

Qt是一个支持多操作系统平台的应用程序开发框架，它的开发语言是C++。Qt最初主要是为跨平台的软件开发者提供统一的，精美的图形用户编程接口，但是现在它也提供了统一的网络和数据库操作的编程接口。正如微软当年为操作系统提供了友好，精致的用户界面一样，今天由于Trolltech 的跨平台开发框架Qt的出现，也使得UNIX、LINUX 这些操作系统以更加方便、精美的人机界面走近普通用户。

Qt是以工具开发包的形式提供给开发者的，这些工具开发包包括了图形设计器，字体国际化工具，Makefile 制作工具， Qt的C++类库等等；谈到C++的类库我们自然会想到MFC，是的，Qt的类库也是等价于MFC 的开发库，但是Qt的类库是支持跨平台的类库，也就是说Qt类库封装了适应不同操作系统的访问细节，这正是Qt的魅力所在。

目前 Qt 系列的软件主要包括 Qt，基于 Framebuffer 的 Qt Embedded，快速开发工具 Qt Designer，国际化工具 Qt Linguist 等部分 。

Trolltech 公司在 1994 年成立，但是在 1992 年，成立 Trolltech 公司的那批程序员就已经开始设计 Qt 了，Qt 的第一个商业版本于 1995 年推出然后 Qt 的发展就很快了，下面是 Qt 发展史上的一些里程碑:

1996 Oct KDE 组织成立

1998 Apr 05 Trolltech 的程序员在 5 天之内将 Netscape5.0 从 Motif 移植到 Qt 上

1998 Apr 08 KDE Free Qt 基金会成立

1998 Jul 09 Qt 1.40 发布

1998 Jul 12 KDE 1.0 发布

1999 Mar 04 QPL 1.0 发布

1999 Mar 12 Qt 1.44 发布

1999 Jun 25 Qt 2.0 发布

1999 Sep 13 KDE 1.1.2 发布

2000 Mar 20 嵌入式 Qt 发布

2000 Sep 06 Qt 2.2 发布

2000 Oct 05 Qt 2.2.1 发布

2000 Oct 30 Qt/Embedded 开始使用 GPL 宣言

2000 Sep 04 Qt free edition 开始使用 GPL

基本上，Qt 同 X Window 上的 Motif，Openwin，GTK 等图形界面库和 Windows 平台上的 MFC，OWL，VCL，ATL 是同类型的东西，但是 Qt 具有下列优点:

✧ 优良的跨平台特性: Qt 支持下列操作系统: Microsoft Windows 95/98， Microsoft

Windows NT，Linux，Solaris，SunOS，HP-UX，Digital UNIX (OSF/1，Tru64)，Irix， FreeBSD，BSD/OS，SCO，AIX，OS390，QNX 等等；

- ✧ 面向对象：Qt 的良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。Qt 提供了一种称为 signals/slots 的安全类型来替代 callback，这使得各个元件之间的协同工作变得十分简单；
- ✧ 丰富的 API：Qt 包括多达 250 个以上的 C++ 类,还提供基于模板的 collections，serialization，file，I/O device，directory management，date/time 类。甚至还包括正则表达式的处理功能；
- ✧ 支持 2D/3D 图形渲染，支持 OpenGL；
- ✧ 大量的开发文档；
- ✧ XML 支持；

1.3 Qt/Embedded 简介

Qt/Embedded 是一个为嵌入式设备上的图形用户接口和应用开发而订做的C++工具开发包。它通常可以运行在多种不同的处理器上部署的嵌入式Linux操作系统上。如果不考虑X Window窗口系统的需要，居于Qt/Embedded的应用程序可以直接对缓冲帧进行写操作。除了类库以外，Qt/Embedded还包括了几个提高开发速度的工具，使用标准的Qt API，我们可以非常熟练的在Windows和Unix编程环境里开发应用程序。

Qt/Embedded是一组用于访问嵌入式设备的Qt C++ API；Qt/Embedded的Qt/X11，Qt/Windows和Qt/Mac版本提供的都是相同的API和工具。Qt/Embedded还包括类库以及支持嵌入式开发的工具。

Qt/Embedded提供了一种类型安全的被称之为信号与插槽的真正的组件化编程机制，这种机制和以前的回调函数有所不同。Qt/Embedded还提供了一个通用的widgets类，这个类可以很容易的被子类化为客户自己的组件或是对话框。针对一些通用的任务，Qt还预先为客户定制了像消息框和向导这样的对话框。

运行Qt/Embedded所需的系统资源可以很小，相对X窗口下的嵌入解决方案而言，Qt/Embedded只要求一个较小的存储空间（Flash）和内存。Qt/Embedded可以运行在不同的处理器上部署的Linux系统，只要这个系统有一个线性地址的缓冲帧并支持C＋＋的编译器。你可以选择不编译Qt/Embedded某些你不需要的功能，从而大大减小了它的内存占有量。

Qt/Embedded包括了它自身的窗口系统，并支持多种不同的输入设备。

开发者可以使用他们熟悉的开发环境来编写代码。Qt的图形设计器(designer)可以用来可视化地设计用户接口，设计器中有一个布局系统，它可以使你设计的窗口和组件自动根据屏幕空间的大小而改变布局。开发者可以选择一个预定义的视觉风格，或是建立自已独特的视觉风格。使用UNIX/LINUX操作系统的用户，可以在工作站上通过一个虚拟缓冲帧的应用程序仿真嵌入式系统的显示终端。

Qt/Embedded也提供了许多特定用途的非图形组件，例如国际化，网络和数据库交互组件。

Qt/Embedded是成熟可靠的工具开发包，它在世界各地被广泛使用。除了在商业上的许多应用以外，Qt/Embedded还是为小型设备提供的Qtopia应用环境的基础。Qt/Embedded以简洁的系统，可视化的表单设计和详致的API让编写代码变得愉快和舒畅。

1.4 Qtopia 介绍

Qtopia 是Trolltech为采用嵌入式Linux操作系统的消费电子设备而开发的综合应用平台，Qtopia包含完整的应用层、灵活的用户界面、窗口操作系统、应用程序的启动程序以及开发框架。

Qtopia的特性如下表所示：

窗口操作系统	游戏和多媒体	工作辅助应用程序
同步框架	PIM应用程序	Internet应用程序
开发环境	输入法	Java集成
本地化支持	个性化选项	无线支持

Trolltech提供三大Qtopia 版本：Qtopia手机版、Qtopia PDA 版和Qtopia 消费电子产品平台：

Qtopia 手机版：Trolltech面向嵌入式Linux的Qtopia手机版(Qtopia Phone)是Qtopia的一个自定义版本，用于内存有限的智能手机和功能手机。它的用户界面可自定义内存占用量低，手机制造商使用它可以创建令人赞叹的图形用户界面，从而令手机卓越超群。Qtopia手机版有两个版本的键盘驱动和手写笔驱动。

Qtopia PDA版：Qtopia PDA版Qtopia PDA是一个强大的平台，专用于基于Linux操作系统的PDA，个人数字助理设备等。许多PDA 都已采用了Qtopia，Qtopia PDA版已经成了事实上的Linux 标准，它代表了可行的第三种PDA设计方案--Qtopia PDA版，具有可定制的用户界面，支持多种不同的屏幕尺寸，以及横向和纵向布局。

Qtopia 消费电子产品平台：While Qtopia手机版和Qtopia PDA版是针对移动电话和PDA制造商的统包解决方案，而Qtopia消费电子产品平台(Qtopia CEP) 则是一套高层次开发平台，适用于那些希望自行设计和开发应用套件的制造商。Qtopia CEP使得制造商能够在形形色色的手写笔和键盘驱动的设备上，创建自定义的环境。这些设备包括电视机，Web，Pad，无线联网板，机顶盒，以及许多其它基于Linux的设备等等。

Qtopia产品系列旨在为基于Linux的消费电子设备提供和创建图形用户界面，它为制造商提供了前所未有的灵活性和众多选择。

第二章 Qt 的安装

2.1 Qt X11 的安装

编译Qt/X11的唯一目的就是为编译QTE提供uic（用户接口编译器）以及基于X Windows系统的FrameBuffer设备模拟器qvfb。当然，如果已经有了这两个工具，再编译Qt/X11就多此一举了。uic用来把.ui文件转换成.h和.cpp文件.这里的uic是x86下的文件。

在安装 QT X11 的过程中，可能需要拥有 root 帐号，这取决于你要安装 Qt 的路径的权限。从 TrollTech 公司的官方 ftp 站点获得 qt-x11-2.3.2.tar.gz 软件包。解开和解压缩软件包：

 # tar -vxzf qt-x11-2.3.2.tar.gz

重命名软件包：

 # mv qt-2.3.2 qt-x11-2.3.2

进入解开后的文件，设置一些环境变量

 # cd qt-x11-2.3.2

 # export QTDIR=/2410/qt-x11-2.3.2/

在配置之前，我们先看一下配置的选项：

#./configure –help

主要的配置信息如下：

The defaults (*) are usually acceptable. Here is a short explanation of each option:

* -release ..... Compile and link Qt with debugging turned off.
  -debug ..... Compile and link Qt with debugging turned on.

* -shared ..... Create and use a shared Qt library (libqt.so)
  -static ..... Create and use a static Qt library (libqt.a)

* -no-gif ..... Do not compile in GIF reading support.
  -gif ..... Compile in GIF reading support. See src/kernel/qgif.h

  -no-sm ..... Do not support X Session Management.
* -sm ..... Support X Session Management, links in -lSM -lICE.

* -no-thread ..... Do not compile with Threading Support
  -thread ..... Compile with Threading Support

* -qt-zlib ..... Use the zlib bundled with Qt.
  -system-zlib ..... Use a zlib from the operating system
  http://www.info-zip.org/pub/infozip/zlib

* -qt-libpng ..... Use the libpng bundled with Qt.

-system-libpng ..... Use a libpng from the operating system.
See http://www.libpng.org/pub/png

* -no-mng ..... Do not compile in MNG I/O support.
-system-libmng ..... Use libmng from the operating system.
See http://www.libmng.com

* -no-jpeg ..... Do not compile in JPEG I/O support.
-system-jpeg ..... Use jpeglib from the operating system.
See http://www.ijg.org

* -no-nas-sound ..... Do not compile in NAS sound support.
-system-nas-sound .. Use NAS libaudio from the operating system.
see http://radscan.com/nas.html

-no-<module> ..... Disables a module, where module can
can be one of: opengl table network canvas

-kde ..... Builds the Qt Designer with KDE 2 support, so that
KDE 2 widgets can be used directly in
the Qt Designer. $KDEDIR must be
set to point to a KDE 2 installation.
See http://www.kde.org

-no-g++-exceptions . Disable exceptions on platforms using the GNU C++
compiler by using the -fno-exceptions flag.

-no-xft ..... Disable support for Anti-Aliased fonts through the
Xft extension library (XFree86 4.0.2 and newer).
-xft ..... Enable support for Anti-Aliased fonts.

Xft support is auto-detected, but you may use these
flags to explicitly enable/disable support.

-platform target ... The platform you are building on (linux-g++)
-xplatform target .. The platform when cross-compiling.

See the PLATFORMS file for a list of supported
operating systems and compilers.

-Istring ..... Add an explicit include path.
-Lstring ..... Add an explicit library path.
-Rstring ..... Add an explicit dynamic library runtime search path.
-lstring ..... Add an explicit library.

Qt/Embedded only:

-qconfig local ..... Use src/tools/qconfig-local.h rather than the
default (qconfig.h).

-depths list ..... Comma-separated list of supported bit-per-pixel
depths, from: v, 4, 8, 16, 24, and 32. 'v' is VGA16.

-accel-voodoo3 ..... Enable Voodoo3 acceleration.
-accel-mach64 ..... Enable Mach64 acceleration.
-accel-matrox ..... Enable Matrox MGA acceleration.
-qvfb ..... Enable X11-based Qt Virtual Frame Buffer.
-vnc ..... Enable VNC server (requires network module).

选择我们需要的配置和平台：

#./configure -platform linux-g++ -thread -system-jpeg -gif -no-xft

按照下面的选择进行一步一步，并且会有以下打印信息：

Type 'Q' to view the Q Public License.
Type 'G' to view the GNU General Public License.
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.

Do you accept the terms of the license?
yes

This target is using the GNU C++ compiler (linux-g++).

Recent versions of this compiler automatically include
code for exceptions, which increase both the size of
the Qt library and the amount of memory taken by your
applications.

You may elect to re-run ./configure with the
-no-g++-exceptions
option to compile Qt without exceptions. This is
completely binary compatible, and existing
applications will continue to work (like KDE 2).

Build type: linux-g++-shared

Thread support..... yes
GIF support..... yes
MNG support..... no

JPEG support..... yes
OpenGL support ..... yes
NAS sound support..... no
Session management..... yes
Xft support (Anti-Aliased Fonts) .. no
XKB support ..... yes

Creating makefiles...
Qt is now configured for building. Just run make.
To reconfigure, run make clean and configure.

生成 Makefile 后，就可以进行安装

#make

安装成功后，将会有以下打印信息：

The Qt library is now built in ./lib
The Qt examples are built in the directories in ./examples
The Qt tutorials are built in the directories in ./tutorial

Note: be sure to set $QTDIR to point to here or to wherever
you move these directories.

Enjoy! - the Trolltech team

2.2 Qt/Embedded 安装

先准备好软件包 qt-embedded-2.3.2.tar：
(这里软件包的名字可能不同，CD 中提供的包可能是 qt-embedded-2.3.2.tar.gz 等)
解开软件包：

 # tar –vxf qt-embedded-2.3.2.tar

重命名：

 # mv qt-2.3.2 qt-embedded-2.3.2

进入软件包中并设置一些环境变量：

#export QTDIR=/2410/qt-embedded-2.3.2/

如果是交叉编译，那么请先设置好 configs 目录下的平台文件：

#vi configs/linux-arm-g++-shared

将其中的 arm 编译器设置成 2410 的编译器，保存该文件。
设置编译选项：

#./configure -xplatform linux-arm-g++ -thread

如果不是交叉编译，想在 qvfb 上运行的话，那么则为

#./configure -xplatform linux-x86-g++ -thread –qvfb

根据其打印的信息设置进行如下对话选择：

Type 'G' to view the GNU General Public License.
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.

Do you accept the terms of the license?
Yes

Choose a feature configuration:
1. Minimal (630 kB)
2. Small (960 kB)
3. Medium (1.5 MB)
4. Large (3 MB)
5. Everything (5 MB)
6. Your own local configuration (src/tools/qconfig-local.h)

Sizes are stripped dynamic 80386 build. Static builds are smaller.

Your choice (default 5):
5

Choose pixel-depths to support:

v. VGA-16 - also enables 8bpp
4. 4bpp grayscale - also enables 8bpp
8. 8bpp
16. 16bpp
24. 24bpp - also enables 32bpp
32. 32bpp

Each depth adds around 100Kb on 80386.

Your choices (default 8,16):
16

Enable Qt Virtual Framebuffer support for development on X11 (default yes)
yes

Building on: linux-x86-g++-shared
Building for: linux-arm-g++-shared

Thread support..... yes
GIF support..... no
MNG support..... no
JPEG support..... yes

Creating makefiles...
Qt is now configured for building. Just run make.
To reconfigure, run make clean and configure.

生成 Makefile 之后，就可以进行编译了：

#make

编译成功之后，则会有如下提示：

The Qt library is now built in ./lib
The Qt examples are built in the directories in ./examples
The Qt tutorials are built in the directories in ./tutorial

Note: be sure to set $QTDIR to point to here or to wherever
you move these directories.

Enjoy! - the Trolltech team

2.3 Qtopia 编译

如果已经安装好了 qt-embedded，就可以进行 Qtopia 的安装了。
准备好 Qtopia 的软件包： qtopia-free-1.5.0。
设置环境变量：

#export QTDIR=/2410/qt-embedded-2.3.2
#export QPEDIR=/2410/qtopia-free-1.5.0

接下来就可以配置了：

#./configure -platform linux-arm-g++

如果是在 qvfb 上运行的话，则为：

#./configure -platform linux-g++

生成 Makefile 之后，就可以进行编译了：

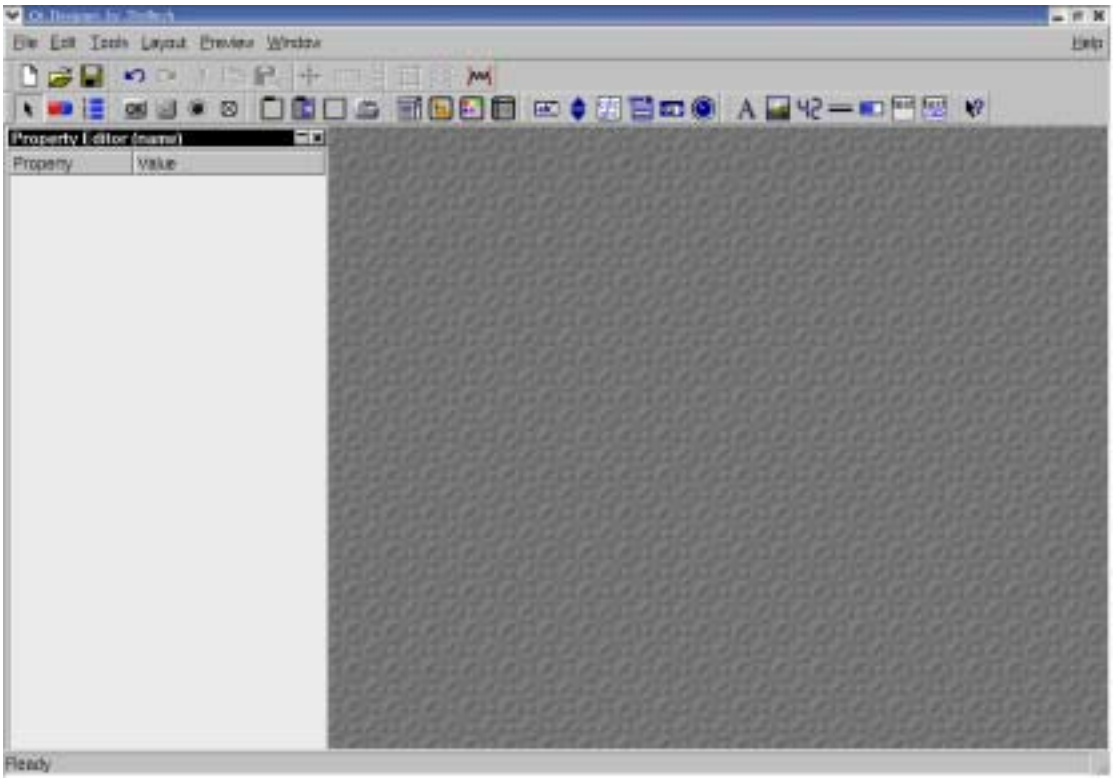
#make

第三章 Qt 常用工具的介绍

4.1 Qt 设计器（Qt Designer）

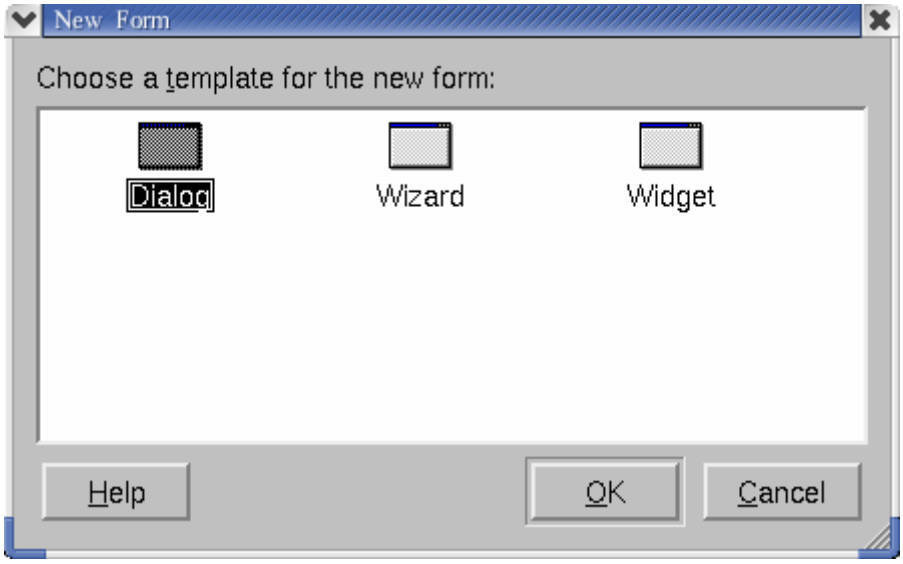
如果用户有 Windows 下 GUI 编程的经验，一定不会反对使用可视化工具辅程序开发工作。如果不使用某个代码生成工具，而采用手动方式编写大量的对话框，其工作量之大。并不是初学者才使用此工具，每一个最求效率的程序员都应该习惯于 GUI 设计工具。

Qt Designer 是 Qt 的可视化图形工具。从前面的实验中，可以发现，要在对话框中对齐控件不是一件容易的事情（需要不断的调整 setGeometry()或者 resize()函数）。Qt 提供了一套管理空间摆放位置的机制，但是使用这些机制对于普通的应用显得有些麻烦。早在 Qt2.0 版本出现之前，一些计算机爱好者就自己动手编写了利用图形化方法设计对话框的工具，Qt 的开发者也认识到这点，因此从 2.2 版本后提供了一个 Qt Designer 这个图形设计工具。下图就是 Qt Designer 的主窗口：

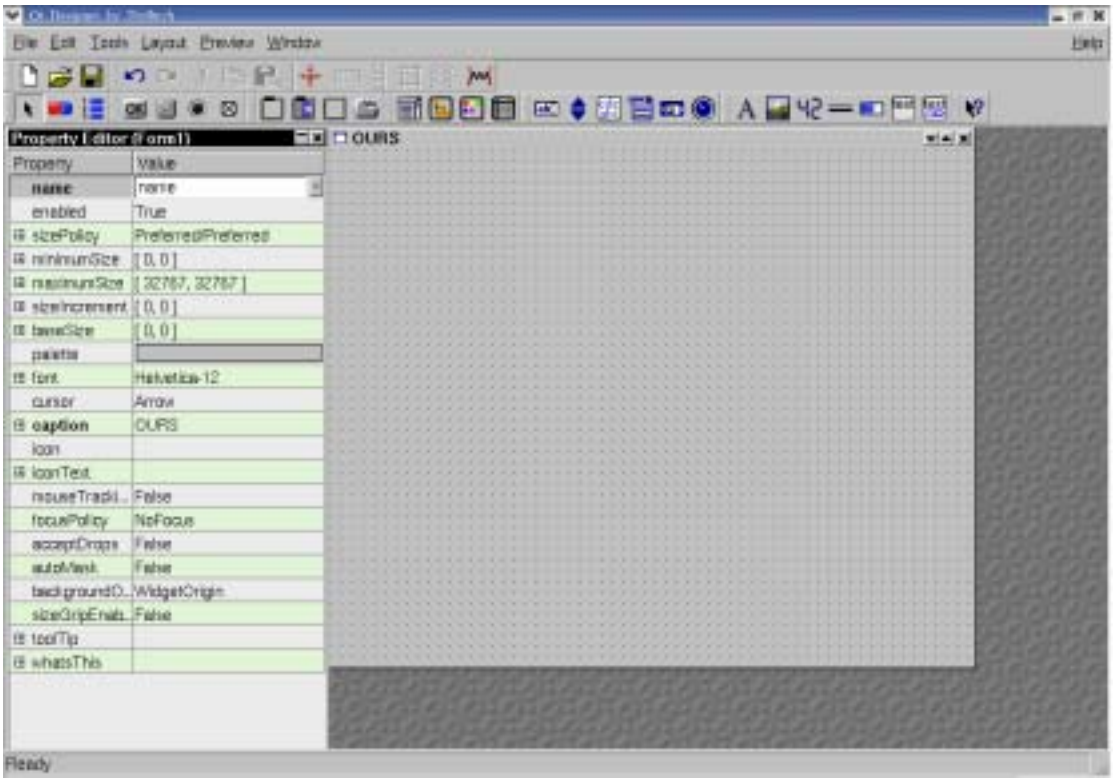


在编译 Qt 的时候会自动编译 Qt Designer。

现在开始创建第一个对话框。从菜单中选择 File|New 命令，或者按快捷键“ Ctrl+N ”，将看到下面这个对话框。



可以在这里选择类型，最常见的可能时 Dialog 或者 Widget。两者的区别在于：Dialog 总是一个单独的窗口，而 Widget 可能被放在其它窗口或者控件中。所有在 Qt Designer 中设计的东西都被称作为表单“ Form ”，既可以是控件、对话框，也可以是其它类型。在这里选择对话框 Dialog，空的对话框看起来如图所示：



主窗口的左边是属性编辑器，可以在这里编辑对话框和控件的各种属性，例如标题、颜色、背景色等，不同的表单有不同的属性域。工具条下面有一些 Qt 的标准控件，可以把它们放入对话框中。先来看看对话框的两个属性。

属性 name 是表示对话框的内部名字，它不是对话框标题。回忆一下，许多 Qt 控件都把 name 作为构造函数的参数之一。可以在属性对话框中直接输入，因为那么是一个字符串变量。

另外一个属性叫 enable，表示该表单的状态，可以选择 true 或者 false，其结果是在代码中生成类似下面的语句：

setEnable(TRUE);

通过前面的学习，应该对这里提到的“ 属性 ”一词并不陌生，它是设置控件行为和外观的基本方法，只不过以前都是直接编写代码。

主窗口的大部分都被表单编辑器占据，可以在这里“ 画 ”出需要的控件。从顶部的控件列表中选择一个喜欢的控件，例如一个复选框。这些控件的外观和程序使用的控件风格有关，所以这里看到的控件可能和最终运行的时候不大一样。可以使用鼠标任意改变控件大小或者移动它的位置。

当手工编写 Qt 应用程序的时候，需要注意控件之间的父子关系。当创建一个非顶层控件的时候，必须指定一个父控件。在 Qt Designer 中，上面的步骤是自动的。当在某一个控件上放置了另外一个控件后，顶层控件自动将其作为父类。可以选择窗口对象查看控件间的层次关系。

保存刚才创建的文件，Qt 把.ui 作为默认的后缀名。Qt Designer 并不直接生成 C＋＋的源代码，原因在于，如果用户修改了文件，那么 Qt Designer 必须修改 C＋＋源程序，而这是一份十分困难的事情。可以打开一个.ui 文件看看，这实际上是 XML 语言格式，就好像是这个样子：

```
<!DOCTYEP UI><UI>
<class>PizzaEntry</class>
<widget>
   <class>QDialog</class>
   <property>
        <name>name</name>
        <cstring>PizzaEntry</cstring>
   </property>
……
```

这个文件是不能被编译的。必须使用另外一个工具 uic，从.ui 文件生成 C++源文件，包括.h 文件和.cpp 文件，uic 表示 UI Compiler。例如：

```
uic –i pizza.h –o pizza.cpp pizza.ui
```

指定从文件 pizza.ui 生成头文件 pizza.h 和实现文件 pizza.cpp，现在可以查看一下生成的文件，它们都是标准的 C++格式。注意相应版本的 uic 要使用对应的 QT。

4.2 Tmake

Tmake 是一个很好用的生成和管理 makefile 的工具，如果不是用 tmake 来管理 makefile 的话，那将会是一件痛苦的事情，虽然有 autoconf 等工具，但毕竟还是十分繁杂的，现在 tmake 将我们完全从繁琐的生成 makefile 的过程中解脱出来，只要很简单的步骤就可以生成 makefile 了。

下面介绍 tmake 的安装

1．下载 tamke 软件包

2．在 linux 上解压 tmake.tar.gz

tar –vxzf tmake.tar.gz

3．设置好 tmake 路径参数（参见下文）

#TMAKEPATH=/tmake/qws/linux-arm-g++

如果不是交叉编译，则为：

#TMAKEPATH=/tmake/qws/linux-g++

#PATH=$PATH:/local/tmake/bin

#export TMAKEPATH PATH

文件夹中可以找到相应的支持。

4．加入 tmake/bin 去你的执行路径。

在/…./qws 路径里面有各种平台支持文件和 tmake 执行文件。tmake 支持的平台有：AIX, Data General, FreeBSD, HPUX, SGI Irix, Linux, NetBSD, OpenBSD, OSF1/DEC, SCO, Solaris, SunOS, Ultrix, Unixware and Win32

用户可以根据自己的应用平台，修改其中平台支持文件。

下面我们开始学习 tmake 的使用：

我假设你有一个小的 qt 程序，他由一个 C++ header 和两个 source file 组成。首先，你要创建一个 tmake 工程文件：

 # progen -n hello -o hello.pro

下面我们来产生 makefile

#tmake hello.pro -o Makefile

最后我们执行 make 命令编译 hello 这个程序。

5．Makefile 模板

Tmake 发行版本中有以下三个模板

- ✧ App.t 用来创建生成发布使用程序的 makefile
- ✧ Lib.t 用来创建生成 libraries 的 makefile
- ✧ Subdirs.t 用来创建目标文档在目录中的 makefile

Tmake.conf 这个 configuration 文件包含了编译选项和各种资源库。在生成的 Makefile 文件里，如果没有达到相应的要求，可以自己手动修改，比如在编写多线程程序的时候，就必须自己手动添加编译选项：-DQT_THREAD_SUPPORT，修改链接库：-lqte-mt。

4.3 Qvfb

虚拟帧缓冲允许在你的桌面机器上开发 Qt/嵌入式程序，而不用在命令台和 X11 之间来回切换。

Qt/Embedded 和 Qt/X11 在 tools 文件夹下都有 qvfb，然而只有 Qt/X11 的中的 qvfb 才是可以执行的二进制代码。

进入 qvfb 所在的文件夹：

 # cd /2410/qt-x11-2.3.2/tools/qvfb

设置一堆环境变量：

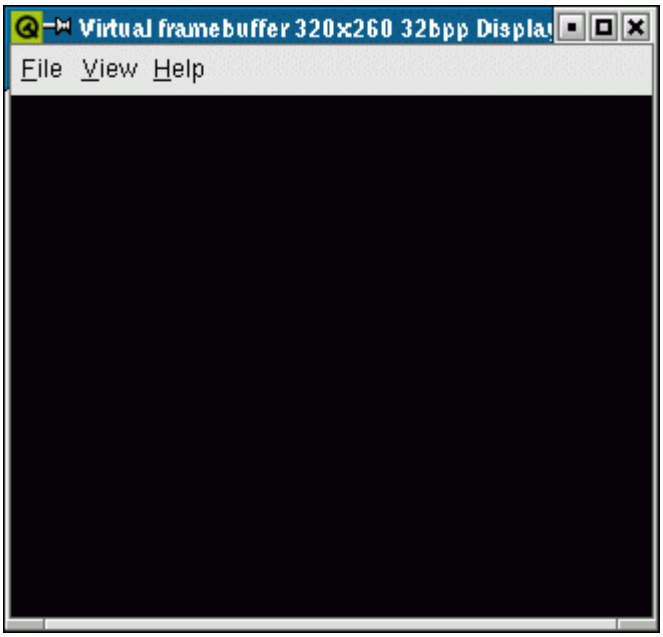
 # export QTDIR=/2410/qt-x11-2.3.2/

 # export PATH=$QTDIR/bin:$PATH

 # export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

运行 qvfb:

 # ./qvfb -width 800 -height 600 -depth 16 &



下面将在 qvfb 上运行编译好的程序。

 # cd /2410/qt-embedded-2.3.2/example/launcher

设置运行的环境变量：

 # export QTDIR=/2410/qt-embedded-2.3.2

 # export QTEDIR=/2410/qt-embedded-2.3.2

#exportLD_LIBRARY_PATH=$QTDIR/lib:$QTEDIR/lib:$LD_LIBRARY_PATH

执行程序：

 # ./launcher –qws

附录. QT 实验环境的建立

qt 环境的建立步骤：

1.所需要的软件包资源:

编译器:arm-linux

QtEmbedded:qt-2.3.2-new

QTopia:qtopia-free-1.5.0

如果不需要在 X Window 环境下运行，而直接在嵌入式 linux 下应用，可以不安装 QT/X11。

2.编译器环境的建立:

将 arm-linux 拷贝到/usr/local/目录下

% cp ~/arm-linux /usr/local -r

设置环境变量

% vi /etc/profile

在# Path manipulation 一行添加下列环境

pathmunge /usr/local/arm-linux/bin

3.QT 环境的建立:

将 qt-2.3.2-new 拷贝到/目录下

% cp ~/qt-2.3.2-new / -r

将 qtopia-free-1.5.0 拷贝到/目录下

% cp ~/qtopia-free-1.5.0 / -r

设置环境变量

修改 QT 环境变量:

% vi /etc/profile.d/qt.sh

写入如下代码

```
# Qt initialization script (sh)
if [ -z "$QTDIR" ] ; then
        QTDIR="/qt-2.3.2-new"
fi
export QTDIR
```

退出保存

% vi /etc/profile.d/qt.csh

写入如下代码

```
# Qt initialization script (csh)
if ( $?QTDIR ) then
        exit
endif
setenv QTDIR /qt-2.3.2-new
```

退出保存

修改 QTopia 环境变量:
仿照上面 QT 环境的设置
% vi /etc/profile.d/qtopia.sh
写入如下代码

```
# QTopia initialization script (sh)
if [ -z "$QPEDIR" ] ; then
        QPEDIR="/qtopia-free-1.5.0"
fi
export QPEDIR
```

退出保存

% vi /etc/profile.d/qtopia.csh
写入如下代码

```
# QTopia initialization script (csh)
if ( $?QPEDIR ) then
        exit
endif
setenv QPEDIR /qtopia-free-1.5.0
```

退出保存

如此,QT 环境基本建立起来了。

4.编译 QT 程序时的问题

qt-2.3.2 本身不带 qmake，所以这里采用 3.1.1 版本里面的 qmake，或者采用前面介绍过的 Tmake 文件来进行 Makefile 文件的修改。

生成后的 Makefile 文件需要修改，修改可以参照实例中的 Makefile 文件进行相应的修改。

另外，QT X11，QTE 和 Qtopia 之间可能存在版本问题，有的版本并不提供相应的工具，需要使用一个叫新版本中的。而且并不是每个版本之间都可以很好的匹配，所以读者在安装 QT 环境的时候一定要注意版本的问题。

第四章 Qt 的编程

实验一 “ Hello word！ ”Qt 初探

实验目的：

本次实验主要任务是创建和显示一个简单的窗口，让用户了解一下 Qt 程序最基本的框架。

实验代码：

helloword.cpp:

```
/****************************************************************************
** $Id: /sample/1/helloworld.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
***/
#include <qapplication.h>
#include <qlabel.h>

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

        QLabel *label = new QLabel( "Hello, world!", 0 );
        label->setAlignment( Qt::AlignVCenter
    | Qt::AlignHCenter );
        label->setGeometry( 10, 10, 200, 80 );
        app.setMainWidget( label );
        label->show();

        int result = app.exec();
        return result;
    }
```

实验原理：

#include <qapplication.h>包含了文件 qapplication.h，此文件总是被包含在同样的源文件中，它里面包含了 main()函数。这个例程因为使用了 QLabel widget 来显示文本，所以也必须包含文件 qlabel.h。

QApplication app(argc, argv) 创建了一个 QApplication 对象，命名为 app。

QApplication 对象是一个容器，包含了应用程序顶层的窗口（或者一组窗口）。顶层窗口是独一无二的，它在应用程序中从来就没有父窗口。因为 QApplication 对象的任务是控制管理你的应用程序，因此在每个应用程序中只能有一个 QApplication 对象。此外，由于创建对象的过程必须初始化 Qt 系统，所以在使用其它任何 Qt 工具之前，QApplication 对象必然已经存在了。

一个 Qt 程序就是一个标准的 C＋＋程序。这就意味着为了启动程序，函数 main()将被操作系统所调用。而且，像所有的 C＋＋程序一样，命令行选项可能会也可能不会传递给 main()函数。命令行选项作为初始化过程的一部分传递给 Qt 系统，也是体现在 **QApplication app(argc, argv)**这条语句中。

两个命令行参数：argc 和 argv，在 app 结构中使用，因为这样可以指定一些特殊的标志和设置。例如，用-geometry 参数启动 Qt 程序，将可以指定窗口显示的位置和大小。通过修改启动程序的外观信息，用户可以自己按照喜好定义程序的外观风格。

QLabel *label = new QLabel("Hello, world!", 0);创建了一个 QLabel 部件。QLabel 部件是一个简单的窗口，能够显示一个字符串。标签指定的父类部件创建为 0,因为这个标签将被作为顶层的窗口，而顶层窗口是没有父类的。Qlabeld 类有三个被定义的构造函数，如下所示，其具体定义参见相关文档

QLabel (QWidget * parent, const char * name = 0, WFlags f = 0)

QLabel(const QString&text, QWidget*parent, const char*name = 0, WFlags f = 0)

QLabel(QWidget*buddy, constQString&text, QWidget*parent, const char*name = 0,WFlags f = 0)

其中 text 参数即为 QLabel 对象所要显示的文本信息，在本程序构造 QLabel 对象时，传入的参数为“ Hello， world ！”。

QLabel 默认的动作是以垂直方向中心对齐的方式显示字符串，以左边为基准。**label->setAlignment(Qt::AlignVCenter | Qt::AlignHCenter)** 调用 setAlignment()函数使得文本在水平和垂直方向上都是位于中心位置。

label->setGeometry(10, 10, 200, 80)决定了标签部件在 QApplication 窗口中的位置、高度和宽度。其中 setGeometry()函数的原形为：setGeometry (int x, int y, int w, int h)。应此 label 在 QApplication 窗口中的坐标是（10， 10）。宽度为 200，高度为 80。

app.setMainWidget(label)语句的作用是把 QLabel 所定义的对象插入到主窗口中。通常插入的主窗口的的部件应该是某种复合部件，是多个部件、文本以及其它应用程序主窗口元件的集合。在本例程中，为了简单起见，插入的对象只是一个简单的标签部件。

label->show()所起的作用是在实现标签在窗口上的显示所必需的。Show()函数并不立刻显示 widget,它只是显示做好准备，以便需要的时候能够显示出来。在这个例程中，父窗口，即 QApplication 窗口，负责显示标签，但它只是在调用标签的 show()方法时才会完成显示。另外一个函数（），叫做 hide()，用于使一个部件从屏幕上消失。

int result = app.exec()；return result 两条语句调用了 exec()函数和返回系统一个值。exec()函数只要当程序停止执行的才返回，它返回一个整型的值，代表程序的完成状态，由于我们并不处理状态代码，这个值只是被简单的返回给系统。

这个例子比较简单，只有一个源文件组成，因此编译它的 Makefile 文件比较简单。但我们不打算自己编写 Makefile，而是采用 tmake 工具生成 Makefile 以减少编写程序的困难，具体 tmake 工具的使用方法参见“ Qt 常用工具的介绍 ”一节。本程序的 Makefile 文件如下：

```
#############################################################################
# Makefile for building helloworld
# Generated by tmake at 11:57, 2005/01/10
```

```
#          Project: helloworld
#        Template: app
#############################################################################

####### Compiler, tools and options

CC   =    gcc
CXX      =    g++
CFLAGS =    -pipe -Wall -W -O2 -fno-default-inline -DNO_DEBUG
CXXFLAGS= -pipe  -DQWS  -fno-exceptions  -fno-rtti  -Wall  -W  -O2  -fno-default-inline
-DNO_DEBUG
INCPATH      =    -I$(QTDIR)/include
LINK     =    g++
LFLAGS =
LIBS     =    $(SUBLIBS) -L$(QTDIR)/lib -lqte
MOC      =    $(QTDIR)/bin/moc
UIC =    $(QTDIR)/bin/uic

TAR=    tar -cf
GZIP     =    gzip -9f

####### Files

HEADERS =
SOURCES =   helloworld.cpp
OBJECTS =   helloworld.o
INTERFACES =
UICDECLS =
UICIMPLS =
SRCMOC      =
OBJMOC      =
DIST      =
TARGET =    helloworld
INTERFACE_DECL_PATH = .

####### Implicit rules

.SUFFIXES: .cpp .cxx .cc .C .c

.cpp.o:
        $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<

.cxx.o:
        $(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<
```

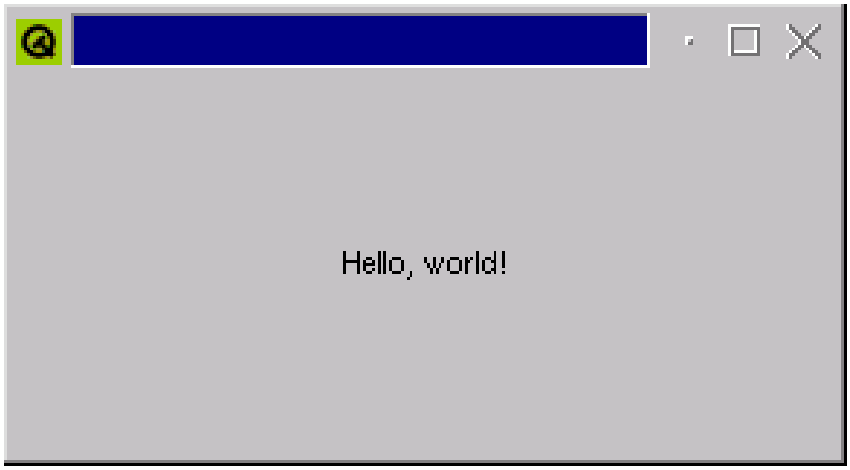
```
.cc.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<

.C.o:
	$(CXX) -c $(CXXFLAGS) $(INCPATH) -o $@ $<

.c.o:
	$(CC) -c $(CFLAGS) $(INCPATH) -o $@ $<

####### Build rules


all: $(TARGET)

$(TARGET): $(UICDECLS) $(OBJECTS) $(OBJMOC)
	$(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(OBJMOC) $(LIBS)

moc: $(SRCMOC)

tmake: Makefile

Makefile: helloworld.pro
	tmake helloworld.pro -o Makefile

dist:
	$(TAR) helloworld.tar helloworld.pro $(SOURCES) $(HEADERS) $(INTERFACES)
$(DIST)
	$(GZIP) helloworld.tar

clean:
	-rm -f $(OBJECTS) $(OBJMOC) $(SRCMOC) $(UICIMPLS) $(UICDECLS)
$(TARGET)
	-rm -f *~ core

####### Sub-libraries


###### Combined headers


####### Compile

helloworld.o: helloworld.cpp
```

在 Makefile 文件中，Makefile 认为环境变量 QTDIR 已经被定义好了，它被用来指明 Qt 开发系统的安装路径。通常，这个环境变量在安装软件的时候就已经配置好它的定义了。因此，在用户编译之前，请务必检查一下环境变量是否正确。

在 Makefile 文件中，定义了一些常用的变量。这些变量有必要解释一下以利于以后的 Makefile 的阅读。如下表所示：

Makefile 中定义的常见变量	
名称	内容
INCL	定义头文件的路径。它们被传递给编译器，告诉编译器到何处去寻找头文件。编译器总是在/usr/include/路径下寻找标准头文件
CFLAGS	定义传给编译器的选择项列表。例如：-pipe 选项指示编译器在两个编译平台间传输数据时使用管道，而不是临时文件。-O2 选项设置了一个相当高的最优化级别。
LFLAGS	定义传给连接器的选择项列表。每个-L 选项所设置得是包含一个或者多个库的路径。
LIBS	定义这个程序需要用到的库的名称列表，将会在 LFLAGS 指明的路径下寻找这些库。这些名称都被扩展称库的名字。例如，-lqte 将改变为 libqte.so，-lm 则成为 libm.so
CC	定义 C 编译器的名字
C××	定义 C＋＋编译器的名字

程序编译完后，在命令提示符下输入“ ./helloword –qws ”选择在 qvfb 中运行。运行结果如图所示：



实验 1 “ Hello world ！”

实验小结：

本实验主要给了一个用户最初的关于 Qt 应用程序的框架，使用户对 Qt 程序的编写有一个初步的印象。在本实验中，用到了 QLabel 部件，用户若需进一步了解其定义，请参照相关文档中的完整定义。

实验二 创建一个窗口并添加按钮

实验目的：

在实验 1 中，我们已经看到了一个 Qt 完整应用程序的基本框架。本实验将继续加深对 Qt 程序整体框架的应用。本质上实验 2 程序与程序 1 无区别。实验将在一个窗口上显示一个按钮，按钮上面添加文字“ Hello,world! ”。

实验代码：

helloword.cpp:

```
/****************************************************************
**
** $Id: /sample/2/helloworld.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************/
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
QApplication a( argc, argv );

QPushButton hello( "Hello world!", 0 );
hello.resize( 100, 30 );

a.setMainWidget( &hello );
hello.show();
int result = a.exec();
return result;
}
```

实验原理：

#include <qapplication.h>由于主程序中必须使用 QApplication 的定义，因此添加此头函数。具体含义如实验 1 所示，这里不加赘述。

#include <qpushbutton.h>这一行包含了 QpushButton 的定义。QpushButton 提供了一个经典的命令按钮。推动按钮或者命令按钮或许是任何图形用户界面中最常用到的窗口部件。推动（点击）按钮来命令计算机执行一些操作，或者回答一个问题。典型的按钮有确定（OK）、应用（Apply）、撤销（Cancel）、关闭（Close）、是（Yes）、否（No）和帮助（Help）。

QPushButton hello("Hello world!", 0)定义了一个按钮。QPushButton 的构造函数在 Qt 中有以下几个定义：

QPushButton (QWidget * parent, const char * name = 0)

QPushButton (const QString & text, QWidget * parent,
const char * name = 0)
QPushButton(const QIconSet & icon, const QString & text,
QWidget * parent, const char * name = 0)

因此本实验中使用的是第二个构造函数，构造了一个命令按钮上面的文字为“ Hello,world! ”。命令按钮通常是矩形并且会用一个文本标签来描述它的操作。标签中有下划线的字母（在文本中它的前面被“ & ”标明）表明快捷键，例如：

QPushButton *pb = new QPushButton("&Download", this);

在这个实例中加速键是 **Alt+D**，并且文本标签将被显示为 **Download**。

按钮可以显示文本标签(Text)或者像素映射(Pixmap)，并且和一个可选的小图标(IconSet)。这些可以通过使用构造函数来设置并且在这之后用 **setText()**、**setPixmap()**和 **setIconSet()**来改变。如果按钮失效，文本或像素映射和图标的外观将被按照图形用户界面的风格来操作表明按钮看起来是失效的。

推动按钮被鼠标、空格键或者键盘快捷键激活，它发射 **clicked()**信号。连接这个信号来执行按钮的操作。推动按钮也提供不太常用的信号，例如，**pressed()**和 **released()**。(关于信号的概念,我们将在下一个实验中重点讲述。)

菜单中的命令按钮默认情况下是自动默认按钮，也就是说当它们接受到键盘焦点时，它们将自动变为默认推动按钮。默认按钮就是一个当用户在对话框中敲击回车键或换行键时被激活的推动按钮。你可以使用 **setAutoDefault()**来改变这一点。注意自动默认按钮会保留一小点额外区域来绘制默认按钮指示器。如果你不想要你的按钮周围的这些空间，调用 **setAutoDefault(FALSE)**。

因为如此的重要，按钮窗口部件在过去的时代中已经发展并提供了大量的变体。Microsoft 风格指南现在显示 Windows 推动按钮大约有 10 种不同状态并且文本暗示有当所有的特种组合都被考虑进去的时候，大约有几十种或更多的情况。

最重要的模式或状态有：

- ✧ 可用或不可用（变灰，失效）:;
- ✧ 标准推动按钮、切换推动按钮或菜单按钮：**void setToggleButton (bool)**、**void setPopup (QPopupMenu * popup)**；
- ✧
- ✧ 开或关（仅对切换推动按钮）：**virtual void setOn (bool)**；
- ✧ 默认或普通。对话框中的默认按钮通常可以被使用回车键或换行键“ 点击 ”：**void QPushButton::setAutoDefault (bool autoDef)**、**void QPushButton::setDefault (bool def)**；
- ✧ 自动重复或者不自动重复：**void QButton::setAutoRepeat (bool)**；
- ✧ 被按下或者没有被按下 **void QButton::setDown (bool)**。

作为一个通用规则，当在用户点击在应用程序或者对话框窗口中点击时（比如应用、撤销、关闭和帮助）并且窗口部件被假设有一个宽的矩形形状的文本标签，应用程序或者对话框窗口要执行一个操作时，使用推动按钮。改变窗口的状态，而不是执行操作的小的、通常正方形的按钮（比如 QFileDialog 右上角的按钮）不是命令按钮，而是工具按钮。Qt 为这些按钮提供了一个特殊类（QToolButton）。

如果你需要切换行为（请参考 **setToggleButton()**）或者当一个按钮被像滚动条那样的箭头按下时，按钮自动重复激活信号（请参考 **setAutoRepeat()**），命令按钮可能不是你想要的。如果拿不准，请使用工具按钮。

命令按钮的一个变体是菜单按钮。它们提供了不仅一个命令，而是几个，因为当它们被按下时，它们弹出一个选项菜单。使用 setPopup()方式来关联一个弹出菜单到一个推动按钮。

其他按钮类是选项按钮（请参考 QRadioButton）和选择框（请参考 QCheckBox）。

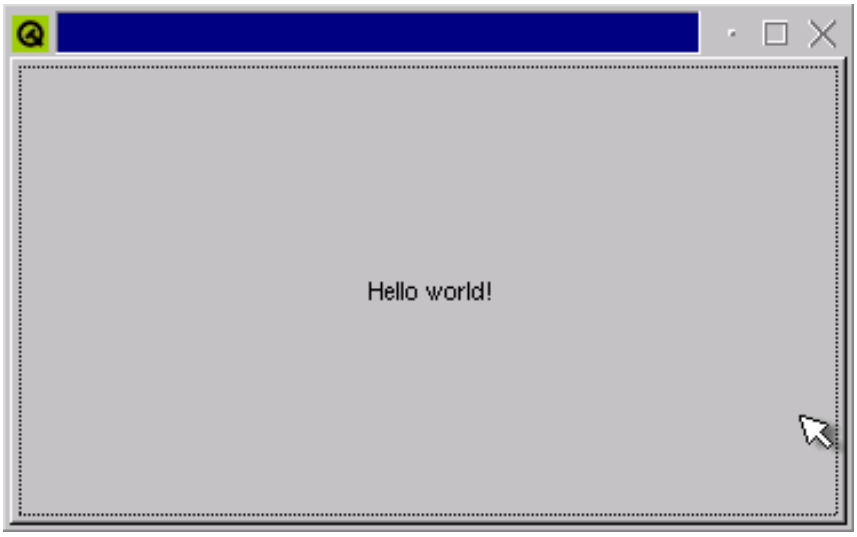
在 Qt 中，QButton 基类提供了绝大多数模式和其它应用编程接口，并且 QPushButton 提供了图形用户界面逻辑。关于应用编程接口的更多信息请参考 QButton。

hello.resize(100, 30) 这个按钮被设置成 100 像素宽，30 像素高（加上窗口系统边框）。在这种情况下，我们不用考虑按钮的位置，并且我们接受默认值。

a.setMainWidget(&hello)这个按钮被选为这个应用程序的主窗口部件。如果用户关闭了主窗口部件，应用程序就退出了。

使用 tmake 工具生成 Makefile 后进行编译，参见光盘中的 Makefile 文件详细内容。

在 Qvfb 中的运行结果如图所示：



实验 2 添加一个按钮

实验小结：

本实验在实验 1 的基础上，进一步使用户加深了对 Qt 应用程序框架的理解。并且重点介绍了一个在用户图形界面中的经典部件：QPushButton。由于用户以后在应用的程序的编写过程中，会经常使用到这个部件。因此在实验完成之后请参照完整的 QPushButton 的定义，加深理解。

实验三 对象间通信：Signal 和 Slot 机制

实验目的：

在 Qt 的众多与众不同的特点中，信号(Signal)/槽(Slot)机制是 Qt 的一个中心特征并且也许是 Qt 与其它工具包的最不相同的部分。信号和槽主要用于对象之间的通讯。在本节的实验中，将重点讲述信号与槽的特点以及其运用。

实验代码：

counter.h:

```
/****************************************************************************
*******
** $Id: /sample/3/counter.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without
limitation.
**
****************************************************************************
*******/
#ifndef COUNTER_H
#define COUNTER_H

#include <qlabel.h>

class Counter: public QWidget
{
    Q_OBJECT
public:
    Counter( QWidget *parent=0, const char *name=0 );
public slots:
    void IncCounter();
    void DecCounter();
private:
    int counter;
    QLabel *label;
};

#endif
```

counter.cpp:

```
/****************************************************************************
*******
```

```
** $Id: /sample/3/counter.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************
***************/
#include <stdio.h>
#include "counter.h"

Counter::Counter( QWidget *parent, const char *name ):
            QWidget( parent, name )
{
      counter = 0;
      label = new QLabel( "0", this );
      label->setAlignment( AlignVCenter | AlignHCenter );
}

void Counter::IncCounter()
{
      char str[30];
      sprintf( str, "%d", ++counter );
      label->setText( str );
}

void Counter::DecCounter()
{
      char str[30];
      sprintf( str, "%d", --counter );
      label->setText( str );
}
```

mainwindow.h:

```
/*****************************************************************
*******
** $Id: /sample/3/mainwindow.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
```

```
**
*****************************************************************************
******/
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <qpushbutton.h>
#include "counter.h"

class MainWindow: public QWidget
{
    Q_OBJECT
public:
    MainWindow( QWidget *parent=0, const char *name=0 );
private:
    QPushButton *AddButton;
    QPushButton *SubButton;
    Counter *counter;
};

#endif
```

mainwindow.cpp

```
/*****************************************************************************
*******
** $Id: /sample/3/mainwindow.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
******/
#include "mainwindow.h"

MainWindow::MainWindow( QWidget *parent, const char *name):
        QWidget( parent, name )
{
    QFont f( "Helvetica", 14, QFont::Bold );
        setFont( f );
    AddButton = new QPushButton( "Add", this );
    AddButton->setGeometry( 50, 15, 90, 40 );
```

```
        SubButton = new QPushButton( "Sub", this );
        SubButton->setGeometry( 50, 70, 90, 40 );

        counter = new Counter( this );
        counter->setGeometry( 50, 125, 90, 40 );

        QObject::connect( AddButton, SIGNAL( clicked() ),
                counter, SLOT( IncCounter() ) );
        QObject::connect( SubButton, SIGNAL( clicked() ),
                counter, SLOT( DecCounter() ) );
}

main.cpp
/****************************************************************************
*******
** $Id: /sample/3/main.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************
      **************/
#include <qapplication.h>
#include "mainwindow.h"

int main( int argc, char **argv )
{
      QApplication app( argc, argv );
      MainWindow *mainwindow = new MainWindow( 0 );
      mainwindow->setGeometry( 50, 20 , 200 ,200 );
      app.setMainWidget( mainwindow );
      mainwindow->show();
      int result = app.exec();
      return result;
}
```

实验原理：

信号与插槽机制提供了对象间的通信机制，它易于理解和使用，并完全被 Qt 图形设计器所支持。

图形用户接口的应用需要对用户的动作做出响应。例如，当用户点击了一个菜单项或是工具栏的按钮时，应用程序会执行某些代码。大部分情况下，我们希望不同类型的对象之间能够进行通信。程序员必须把事件和相关代码联系起来，这样才能对事件做出响应。

以前的工具开发包使用的事件响应机制是易崩溃的，不够健壮的，同时也不是面向对象的。Trolltech 已经创立了一种新的机制，叫做“信号与插槽”。信号与插槽是一种强有力的对象间通信机制，它完全可以取代原始的回调和消息映射机制；信号与插槽是迅速的，类型安全的，健壮的，完全面向对象并用 C++来实现的一种机制。在以前，当我们使用回调函数机制来把某段响应代码和一个按钮的动作相关联时，我们通常把那段响应代码写成一个函数，然后把这个函数的地址指针传给按钮，当那个按钮被按下时，这个函数就会被执行。对于这种方式，以前的开发包不能够确保回调函数被执行时所传递进来的函数参数就是正确的类型，因此容易造成进程崩溃，另外一个问题是，回调这种方式紧紧的绑定了图形用户接口的功能元素，因而很难把开发进行独立的分类。Qt 的信号与插槽机制是不同的。Qt 的窗口在事件发生后会激发信号。例如一个按钮被点击时会激发一个“clicked”信号。程序员通过建立一个函数（称作一个插槽），然后调用 connect()函数把这个插槽和一个信号连接起来，这样就完成了一个事件和响应代码的连接。信号与插槽机制并不要求类之间互相知道细节，这样就可以相对容易的开发出代码可高重用的类。信号与插槽机制是类型安全的，它以警告的方式报告类型错误，而不会使系统产生崩溃。例如，如果一个退出按钮的 clicked() 信号被连接到了一个应用的退出函数－插槽 quit()。那么一个用户点击退出键将使应用程序终止运行。上述的连接过程用代码写出来就是这样

connect(button, SIGNAL(clicked()), qApp, SLOT(quit()))

我们可以在Qt应用程序的执行过程中增加或是减少信号与插槽的连接。信号与插槽的实现扩展了C++的语法，同时也完全利用了C++面向对象的特征。信号与插槽可以被重载或者重新实现，它们可以定义为类的公有，私有或是保护成员。

信号：当对象的内部状态发生改变，信号就被发射，在某些方面对于对象代理或者所有者也许是很有趣的。只有定义了一个信号的类和它的子类才能发射这个信号。

例如，一个列表框同时发射 highlighted()和 activated()这两个信号。绝大多数对象也许只对 activated()这个信号感兴趣，但是有时想知道列表框中的哪个条目在当前是高亮的。如果两个不同的类对同一个信号感兴趣，你可以把这个信号和这两个对象连接起来。当一个信号被发射，它所连接的槽会被立即执行，就像一个普通函数调用一样。信号/槽机制完全不依赖于任何一种图形用户界面的事件回路。当所有的槽都返回后 emit 也将返回。

如果几个槽被连接到一个信号，当信号被发射时，这些槽就会被按任意顺序一个接一个地执行。

槽：当一个和槽连接的信号被发射的时候，这个槽被调用。槽也是普通的 C++函数并且可以像它们一样被调用；它们唯一的特点就是它们可以被信号连接。槽的参数不能含有默认值，并且和信号一样，为了槽的参数而使用自己特定的类型是很不明智的。

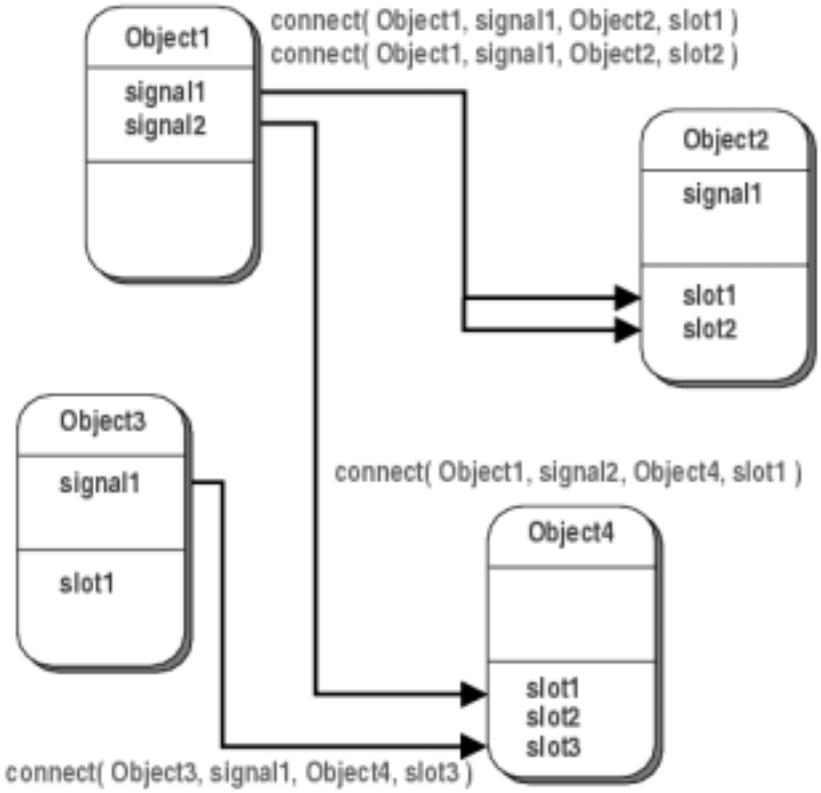
因为槽就是普通成员函数，但却有一点非常有意思的东西，它们也和普通成员函数一样有访问权限。一个槽的访问权限决定了谁可以和它相连：

一个 public slots:区包含了任何信号都可以相连的槽。这对于组件编程来说非常有用：你生成了许多对象，它们互相并不知道，把它们的信号和槽连接起来，这样信息就可以正确地传递，并且就像一个铁路模型，把它打开然后让它跑起来。

一个 protected slots:区包含了之后这个类和它的子类的信号才能连接的槽。这就是说这些槽只是类的实现的一部分，而不是它和外界的接口。

一个 private slots:区包含了之后这个类本身的信号可以连接的槽。这就是说它和这个类是非常紧密的，甚至它的子类都没有获得连接权利这样的信任。

你也可以把槽定义为虚的，这在实践中被发现也是非常有用的。



信号和槽的机制是非常有效的，但是它不像“真正的”回调那样快。信号和槽稍微有些慢，这是因为它们所提供的灵活性，尽管在实际应用中这些不同可以被忽略。通常，发射一个和槽相连的信号，大约只比直接调用那些非虚函数调用的接收器慢十倍。这是定位连接对象所需的开销，可以安全地重复所有的连接（例如在发射期间检查并发接收器是否被破坏）并且可以按一般的方式安排任何参数。当十个非虚函数调用听起来很多时，举个例子来说，时间开销只不过比任何一个“new”或者 “delete”操作要少些。当你执行一个字符串、矢量或者列表操作时，需要“new”或者 “delete”，信号和槽仅仅对一个完整函数调用的时间开销中的一个非常小的部分负责。无论何时你在一个槽中使用一个系统调用和间接地调用超过十个函数的时间是相同的。在一台 i585-500 机器上，你每秒钟可以发射 2，000，000 个左右连接到一个接收器上的信号，或者发射 1，200，000 个左右连接到两个接收器的信号。信号和槽机制的简单性和灵活性对于时间的开销来说是非常值得的，你的用户甚至察觉不出来。

在解释了信号与槽的工作机制后，我们将分析一下实验源程序，以进一步理解。

在 counter.h 文件中，**class Counter: public QWidget** 定义了一个 Counter 类，该类公有继成了 QWidget 类。所以新类是一个窗口部件，并且可以最为一个顶层窗口或者子窗口部件，就像实验 2 所讲述的 QPushButton 那样。除了从 QWidget 类继承的函数之外，该类只有一个公有函数即自己本身的构造函数：**Counter(QWidget *parent=0, const char *name=0)**。第一个参数是它的父窗口部件。为了生成一个顶层窗口，你可以指定一个空指针作为父窗口部件。就像你看到的那样，这个窗口部件总是默认地被认做是一个顶层窗口。第二个参数是这个窗口部件的名称。这个*不是*显示在窗口标题栏或者按钮上的文本。这只是分配给窗口部件的一个名称，以后可以用来查找这个窗口部件，并且这里还有一个方便的调试功能可以完整地列出窗口部件层次。

```
public slots:
    void IncCounter();
    void DecCounter();
```

这两条语句定义了两个公有槽，用于接收外部发出的信号，分别进行加计数和减计数。

```
private:
    int counter;
    QLabel *label;。
```

这两条语句定义了 Counter 类的两个私有成员：计数器（Counter）和标签部件（label），分别用于计数和显示计数值，因为要用到 QLabel 类，所以在包含头文件中必须包含 qlabel.h，即：**#include <qlabel.h>**。

在 Counter.cpp 文件中，主要是实现 Counter 类。

```
Counter::Counter( QWidget *parent, const char *name ):
                    QWidget( parent, name )
{
        counter = 0;
        label = new QLabel( "0", this );
        label->setAlignment( AlignVCenter | AlignHCenter );
}
```

上面一系列即为 Counter 类的有参构造函数，像大多数窗口部件一样，它把 parent 和 name 传递给了 QWidget 的构造函数。在构造函数里面，对两个私有成员进行了初始化：将计数值清零和将标签部件清空。关于 QLabel 类的使用，可以参照实验 1。

```
void Counter::IncCounter()
{
        char str[30];
        sprintf( str, "%d", ++counter );
        label->setText( str );
}
```

上面几条语句是对公有槽 IncCounter()的实现，在每次槽被所连接的信号触发后，计数器加一并且将通过标签部件显示出来。

```
void Counter::DecCounter()
{
        char str[30];
        sprintf( str, "%d", --counter );
        label->setText( str );
}
```

上面几条语句是对公有槽 DecCounter()的实现，在每次槽被所连接的信号触发后，计数器减一并且将通过标签部件显示出来。

由于用到了标准 C 的 springf()函数，因此在头文件中必须包含标准输入输出库：stdio.h 。

到此，Counter 类的定义和实现全部完成，该类主要是定义了两个加计数器和减计数器的共有槽。当在获得外部的信号时，对私有成员 counter 进行加减和显示结果。

mainwindow.h 主要是定义了一个 MainWindow 的类，该类公有的继承了 QWidget 类，成为一个标准的部件。MainWindow 类的作用是用来显示一堆组件而将自己成为较为复杂的组件。MainWindow 类的构造函数与 Counter 的构造函数无异，不加赘述。

```
private:
        QPushButton *AddButton;
        QPushButton *SubButton;
        Counter *counter;
```

在对 MainWindow 类的私有成员声明中，有两个 QPushButton 定义的对象和一个我们刚才定义过的 Counter 类的对象。关于 QPushButton 的使用，可以参照实验二。

在 mainwindow.cpp 中，只有一个 MainWindow 的构造函数。同样的：

```
MainWindow::MainWindow( QWidget *parent, const char *name)
        :QWidget( parent, name )
```

只是简单的将参数给 QWidget 的构造函数。

```
QFont f( "Helvetica", 14, QFont::Bold );
        setFont( f );
```

QFont 是 Qt 中关于字体的类，可以用来设置所要显示文字的字体。QFont 有两个构造函数：

```
QFont()
QFont ( const QString & family, int pointSize = 12,
        int weight = Normal, bool italic = FALSE )
```

在 Qt 中，定义了如下几种字体：

```
enum StyleHint { Helvetica, SansSerif = Helvetica, Times,
        Serif = Times,Courier, TypeWriter = Courier,
    OldEnglish,
        Decorative = OldEnglish, System, AnyStyle }
```

字形有如下几种常用的定义：

```
enum Weight { Light = 25, Normal = 50, DemiBold = 63,
            Bold = 75, Black = 87 }
```

在 MainWindow 的构造函数中，定义了一个字型为：Helvetica，字号为 14，且字形为粗体的字体。**SetFont(f)**将这个字体设为 MainWindow 中的字体，因此 MainWindow 中的文字都为这个字体，且其子部件的字体也为这个字体。

下面四条语句分别定义了两个名为“ Add ”和“ Sub ”的按钮，按钮的宽度均为 90 象素，高度均为 40 象素。一个位于 MainWindow 的（50，15）处，一个位于 MainWindow 的（50，70）处。

```
AddButton = new QPushButton( "Add", this );
AddButton->setGeometry( 50, 15, 90, 40 );
SubButton = new QPushButton( "Sub", this );
SubButton->setGeometry( 50, 70, 90, 40 );
```

MainWindow 还有一个私有成员 counter，为刚才定义过的 Counter 的对象。

```
counter = new Counter( this );
counter->setGeometry( 50, 125, 90, 40 );
```

counter 被放置在 MainWindow 的（50，125）处，其宽度为 90，高度为 40。

```
QObject::connect( AddButton, SIGNAL( clicked() ),
        counter, SLOT( IncCounter() ) );
```

这条语句将“ Add ”按钮的 clicked()信号与 counter 的 IncCounter()公有槽连接在一起。

```
QObject::connect( SubButton, SIGNAL( clicked() ),
        counter, SLOT( DecCounter() ) );
```

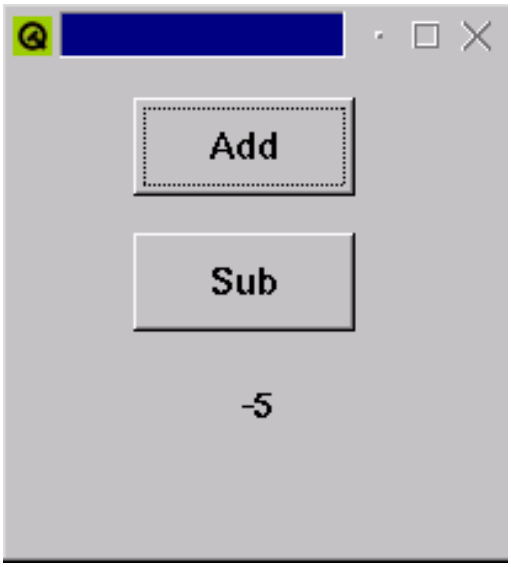
这条语句将“ Sub ”按钮的 clicked()信号与 counter 的 DecCounter()公有槽连接在一起。

当“ Add ”按钮被按下的时候，clicked()信号就被发射。根据信号与槽机制的原理，IncCounter()的代码就被执行，计数器就加一并显示结果。同理，当“ Sub ”按钮被按下时，DecCounter()中的代码就会被执行，计数器减一并显示结果。

在 main.cpp 中，定义了一个 MainWindow 类的对象 mainwindow，并且放在主窗口的（50，20）处，宽为 200 象素，高为 200 象素。

```
MainWindow *mainwindow = new MainWindow( 0 );
    mainwindow->setGeometry( 50, 20 , 200 ,200 );
```

编译之后在 Qvfb 上运行的结果如下所示：



实验小结：

在本实验 3 中，我们主要关注的是 Qt 中的重要特征：信号和槽机制。总体来看，信号和槽构成了一个强有力的组件编程机制。信号与槽的机制在以后的实验中会经常用到，在 Qt 程序编写的时候可以增添很多的灵活性。用户也可以看到，在本此实验中，有多个源文件和头文件。因为 Qt 也是 C＋＋，因此在程序实现的过程中尽量实现模块化，以便于通用性和程序的清晰性。细心的用户不难发现，在定义某些类的时候，不止一次的出现过 Q_OBJECT 宏定义的声明，这个宏必须被包含到所有使用信号和槽的类，在 moc 元编译器编译源文件的时候，遇到 Q_OBJECT 就会自动生成另一个含有源对象的 C++代码，详情可以参照 Qt 的元对象系统：

http://www.qiliang.net/qt/metaobjects.html

在使用信号和槽的机制时，由于其特殊性，因此需要注意的问题有以下几点：

- ✧ signal、slot 可以接受参数，但不能是函数的指针
- ✧ 不支持缺省的参数
- ✧ 所有的 signal 和 slot 不能有返回值（必须是 void）
- ✧ slot 可以为虚函数
- ✧ signal 的访问权限和 protected 相同，即只有本身及其派生类可以 emit signal
- ✧ signal 和 slot 不能是构造函数

实验四 菜单和快捷键

实验目的：

在逐步深入对 Qt 程序的了解之后，虽然已经了解了 Qt 应用程序的框架但是窗口都是简单的几个部件。而在实际应用中，几乎所有的窗口程序都有菜单栏、工具条、状态栏等装饰。在本程序中，我们将加入这些东西，以丰富应用程序的界面和增加其使用功能。

实验代码：

mainwidget.h:

```
/****************************************************************************
*******
** $Id: /sample/4/mainwidget.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
****************************************************************************
*******/

#ifndef _MAINWIDGET_H_
#define _MAINWIDGET_H_

#include <qapplication.h>
#include <qmainwindow.h>
#include <qpopupmenu.h>
#include <qmenubar.h>
#include <qlabel.h>


class MainWidget: public QMainWindow
{
    Q_OBJECT
public:
    MainWidget( QWidget *parent=0, const char *name=0 );
public slots:
    void openFile();
    void saveFile();
    void exitMain();
private:
    QLabel *label;
};
```

```
#endif
```

mainwidget.cpp:

```
/****************************************************************************
*******
** $Id: /sample/4/mainwidget.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#include "mainwidget.h"

MainWidget::MainWidget( QWidget *parent, const char *name )
        :QMainWindow( parent, name )
{
    setCaption( "qt_Example" );
    setBackgroundColor( white );

    QFont f( "Helvetica", 18, QFont::Bold );
    setFont( f );

    label = new QLabel( "", this );
    label->setGeometry( 50, 50, 250, 50 );
    label->setBackgroundColor( white );

    QPopupMenu *file = new QPopupMenu;
    QFont f1( "Helvetica", 14, QFont::Bold );
    setFont( f1 );
    file->setFont( f1 );
    file->insertItem( "&Open", this, SLOT( openFile() ),
CTRL+Key_O );
    file->insertItem( "&Save", this, SLOT( saveFile() ), CTRL+Key_S );

    int id_save = file->insertItem( "&Save", this, SLOT( saveFile() ) );
    file->setItemEnabled( id_save, FALSE );
    file->insertItem( "E&xit", this, SLOT( exitMain() ),
CTRL+Key_X );
```

```
    QMenuBar *menu;
    menu = new QMenuBar( this );
    menu->insertItem( "&File", file );
}


void MainWidget::openFile()
{
    label->setText( "File has been opened!" );
}


void MainWidget::saveFile()
{
    label->setText( "File has been saved!" );
}


void MainWidget::exitMain()
{
    QApplication::exit();
}
```

main.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/4/main.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    **
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #include <qapplication.h>
    #include "mainwidget.h"

    int main( int argc, char **argv )
    {
        QApplication app( argc, argv );
        MainWidget *mainwidget = new MainWidget( 0 );
```

```
    mainwidget->setGeometry(10, 30, 280, 200 );
    app.setMainWidget( mainwidget );
    mainwidget->show();
    int result = app.exec();
    return result;
}
```

实验原理：

在 mainwidget.h 头文件中，定义了一个 MainWidget 的主部件类，用于显示整个主要窗口。该类继承了 QMainWindow 类。并且定义了三个公有的槽，**void openFile()**为打开一个文件，**void saveFile()**为保存一个文件，**void exitMain()**为退出主窗口。MainWidget 类还有一个 QLabel 类型的私有成员。

在 mainwidget.cpp 中，构造函数除了一如既往的将参数传给父类 QMainWindow 之外，还做了一些其它操作。

```
setCaption( "qt_Example" );
setBackgroundColor( white );
```

setCaption()为设置窗口标题的函数，该函数在 QWidget 类中被定义过，MainWidget 共有的继承了 QMainWindow，因此也合乎条理的使用这个函数来设置自己的标题。一般只有顶级窗口设置这个才有效。

SetBackgroundColor()函数为设置当前窗口的背景色。背景色是当绘制不透明文本、点画线和位图时填充的颜色。背景色在透明背景模式（这是默认的）下无效。该函数传递的参数为 QColor 定义的对象。

```
QFont f( "Helvetica", 18, QFont::Bold );
setFont( f );
```

这两条语句设置了当前窗口的字体，实验 3 中曾经介绍过，不加赘述。

```
label = new QLabel( "", this );
label->setGeometry( 50, 50, 250, 50 );
label->setBackgroundColor( white );
```

这三条语句对私有成有 label 进行初始化，设置了 label 标签部件显示的内容、显示的位置和宽高以及其背景色。

```
QPopupMenu *file = new QPopupMenu;
```

该语句定义了一个弹出式菜单 file。

QpopupMenu 类提供了一个弹出式菜单的部件。一个弹出式菜单就是一个选择菜单，它既可以是一个标准的菜单栏中的下拉菜单，也可以是一个独立的（弹出式）菜单。当用户点击菜单项或者按下指定的快捷键就可以在菜单栏上显示下拉菜单，通过调用 QMenuBar::insertItem()函数可以将菜单加入菜单栏中。通过调用 popup()同步地显示独立菜单，而 exec()为异步地显示。

一个弹出菜单应该包含一列菜单项。可以通过使用 insertItem()函数来添加菜单项。一个菜单项可以是字符串(string)，可以是象素映射（pixmap）,也可以是包含了画图功能地自定义项目（参照：QCustomMenuItem）。另外，菜单项在最左边还可以有可供选择地图标和例如“ Ctrl+X ”之类地加速键。

菜单项有三种：分隔符、代表某种操作地菜单项和包含子菜单项的项目。插入分隔符采用 insertSeparator()函数实现。对子菜单，在调用 insetItem()函数时参数传入 QpopupMenu 类型的指针来实现。其它的菜单项都是体现某种操作的条目。

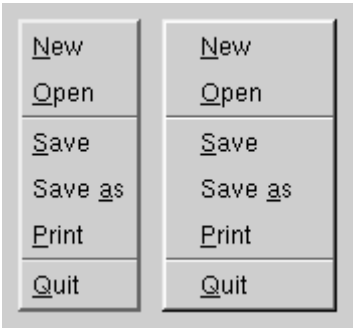
通常在插入具有操作的菜单项目的时候，需要指定一个接受对象和一个槽。接收对象

在菜单项被选中的时候会被告知。另外，QpopupMenu 提供了两个信号：activated()和highligthed()，用于发射各自的标志符。在实际应用中，有时会将多个菜单项和一个槽相连。为了区分它们，可以通过指定槽的入口参数为整型，而通过 setItemParameter（）来设置使每个项目具有独一无二的值。

使用 clear()清除一个菜单，清除一个菜单项使用 removeItem()或 removeItemAt()。

菜单项可以设置为激活或不激活，可以通过 setItemEnabled()来改变它们的状态。在一个菜单项即将可见之前，会发射出一个 aboutToShow()信号。用户可以在用户看见之前通过这个信号设置菜单项的恰当状态：enable/disable。同样的，在菜单项隐藏的时候，相应的 aboutToHide()信号也会被发射。

典型的菜单如下图所示：



上面是对 QpopupMenu 类的一个简单的概述，详细情况请参考 QpopupMenu 类的完整定义。

```
QFont f1( "Helvetica", 14, QFont::Bold );
file->setFont( f1 );
```

在定义完一个弹出式菜单后，上面两条语句对该菜单中显示的字体进行了设置。

```
file->insertItem( "&Open", this, SLOT( openFile() ), CTRL+Key_O );
file->insertItem( "&Save", this, SLOT( saveFile() ), CTRL+Key_S );
```

在介绍 QpopupMenu 类的时候，提到过关于插入菜单项目的函数 insetItem()。在完整定义中，insetItem()函数有如下一些定义：

- ✧ int **insertItem** (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- ✧ int **insertItem** (const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1)
- ✧ int **insertItem** (const QString & text, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QString & text, int id = -1, int index = -1)
- ✧ int **insertItem** (const QString & text, QPopupMenu * popup, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QString & text, QPopupMenu * popup, int id = -1, int index = -1)

- ✧ int **insertItem** (const QPixmap & pixmap, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, int id = -1, int index = -1)
- ✧ int **insertItem** (const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, const QPixmap & pixmap, QPopupMenu * popup, int id = -1, int index = -1)
- ✧ int **insertItem** (QWidget * widget, int id = -1, int index = -1)
- ✧ int **insertItem** (const QIconSet & icon, QCustomMenuItem * custom, int id = -1, int index = -1)
- ✧ int **insertItem** (QCustomMenuItem * custom, int id = -1, int index = -1)

在本实验中，使用的是 int **insertItem** (const QString & text, const QObject * receiver, const char * member, const QKeySequence & accel = 0, int id = -1, int index = -1) 这个函数，第一个参数代表菜单的文本，第二个和第三个参数表示接收菜单事件的类和槽。需要注意的是 insetItem()方法的第 2、3 个参数，这实际上是调用了 connect()方法把菜单被选中这个信号与某个类的槽连接起来，这是很典型的用法。

因此 file->insertItem("&Open", this, SLOT(openFile()), CTRL+Key_O)的含义就是在 file 菜单中插入了一个显示文本名为 Open 的菜单项，并且将这个菜单项的信号与本类定义的槽 openFile()相连，当 Open 菜单项被触发时，就执行 openFile()函数中的代码。并且指定了 Open 菜单项的快捷键为 Ctrl+O。

同样的，file->insertItem("&Save", this, SLOT(saveFile()), CTRL+Key_S); 在 file 菜单中继续插入了一个 Save 菜单项，点击该菜单项时，执行 saveFile()里的代码，并且快捷键为 Ctrl+S。

```
int id_save = file->insertItem( "&Save", this, SLOT( saveFile() ) );
file->setItemEnabled( id_save, FALSE );
```

在 file 菜单中插入一个 save 菜单项时，将 insetItem()函数的返回值赋值给 id_save。InsetItem（）函数返回的是该菜单项的独一无二的菜单标识符。SetItemEnable()函数为设置菜单项的状态，第一个参数为菜单项的菜单标识符，第二个参数为该菜单项的状态，FALSE=Disable，TRUE=Enable。

```
QMenuBar *menu;
menu = new QMenuBar( this );
```

QMenuBar 类为用户提供了一个水平方向的菜单栏。

一个菜单栏是有一列的下拉菜单构成的。通过 insetItem()来为菜单栏添加下拉菜单。例如：假设 menubar 为一个 QmenuBar 类的指针，filemenu 为一个 QpopupMenu 类的指针，下面将阐述如何将菜单加入菜单栏中：

```
menubar->insertItem( "&File", filemenu );
```

“ & ”符号表示着 filemenu 菜单的快捷键为 Alt+F，如果你要使用一个真实的“ & ”，那么就使用两个“ && ”。

菜单也可以设置为激活或不激活，通过使用函数 setItemEnabled()来实现。没有必要为一个菜单栏布置位置，它自己会自动设置自己的位置为父窗口部件的顶端，并且会随着父窗口大小的改变而适当的调整。在大多数的窗口风格的应用程序中，用户将会使用到由 QmainWindw 提供的 menuBar()来往菜单栏上添加菜单和为菜单添加操作。例如：

```
QpopupMenu *file = new QpopupMenu( this );
menuBar()->insertItem( “&File”, file );
```

```
fileNewAction->addTo( file );
```

菜单栏中的条目可以由文字和象素映射（或图标），参照 insetItem()函数的重载，插入分隔符请参照 insertSeparator()。也可以使用自定义的来自 QCustomMenuItem 的菜单元素。菜单栏中的条目可以通过 removeItem()来清除，通过 setItemEnabled()来设置激活与否。典型的菜单栏如下图所示：

因此 menu->insertItem("&File", file)就把刚才定义过的 file 下拉菜单添加到了工具栏上。QMenuBar 的其它内容请参照 QMenuBar 的完整定义。

```
void MainWidget::openFile()
{
    label->setText( "File has been opened!" );
}
```

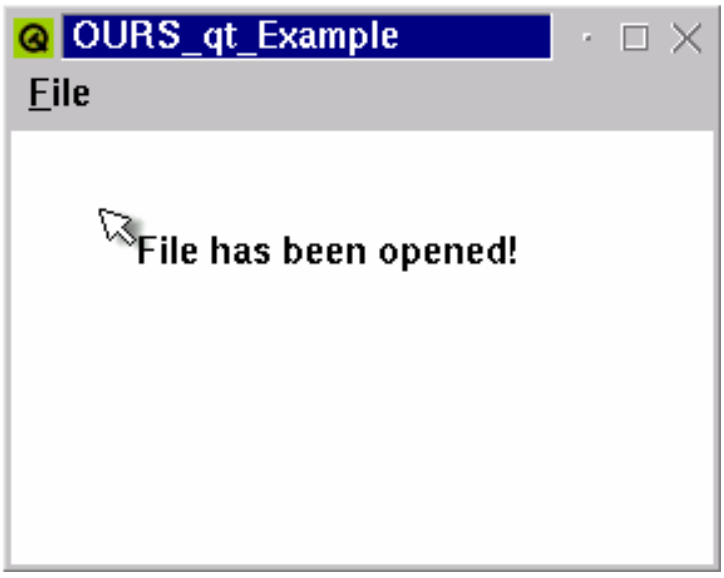
openFile()为 file 下拉菜单中的 Open 菜单项连接的操作，当点击该菜单项时，将标签部件的文字设成相应的状态。

```
void MainWidget::saveFile()
{
    label->setText( "File has been saved!" );
}
```

saveFile()为 file 下拉菜单中的 Save 菜单项连接的操作，当点击该菜单项时，将标签部件的文字设成相应的状态。

```
void MainWidget::exitMain()
{
    QApplication::exit();
}
```

exitMain()为 file 下拉菜单中的 Exit 菜单项连接的操作，当点击该菜单项时，调用了 exit()函数。该退出为应用程序的退出。在 main.cpp 中，所有的语句格式我们均已经在前面的实验中见过，因此这里不加叙述。编译之后，在 Qvfb 中运行的结果如图所示：



实验 4 菜单和快捷键

实验小结：

本实验介绍了窗口应用程序中经常使用的两个重要部件：菜单和菜单栏，几乎所有的较完整的应用程序都会有这两个部件。实验 4 只是添加了一个下拉式菜单，并且菜单项也不多，用户可以试着多添加一些菜单，丰富菜单项。并且可以在菜单项的操作中添加自己的需要的代码。

另外这里补充一下 QMainWindow 的说明，在实验中，我们自定义的类 MainWidget 公有的继承了 QMainWindow 类。QMainWindow 类提供一个有菜单条、锚接窗口（例如工具条）和一个状态条的主应用程序窗口。

主窗口通常用在提供一个大的中央窗口部件（例如文本编辑或者绘制画布）以及周围菜单、工具条和一个状态条。QMainWindow 常常被继承，因为这使得封装中央部件、菜单和工具条以及窗口状态变得更容易。继承使创建当用户点击菜单项或者工具条按钮时被调用的槽成为可能。你也可以使用 *Qt 设计器*来创建主窗口。我们将简要地回顾一下有关添加菜单项和工具条按钮，然后描述 QMainWindow 自己的便捷。

```
QMainWindow *mw = new QMainWindow;
QTextEdit *edit = new QTextEdit( mw, "editor" );
edit->setFocus();
mw->setCaption( "Main Window" );
mw->setCentralWidget( edit );
mw->show();
```

QMainWindow 可以像上面那样显示地来被创建。中央窗口部件是通 setCentralWidget() 设置地。弹出菜单可以被添加到默认工具条，窗口部件可以被添加到状态条，工具条和锚接窗口可以被添加到任何一个锚接区域。

```
ApplicationWindow * mw = new ApplicationWindow();
mw->setCaption( "Qt Example - Application" );
mw->show();
```

上面代码中的 ApplicationWindow 是我们自己写的 QMainWindow 的子类，这是一个使用 QMainWindow 的常用方法。在继承的时候，我们在子类的构造函数中添加菜单项和工具条。如果我们已经直接创建了一个 QMainWindow 实例，我们可以很容易地通过传递 QMainWindow 实例代替作为父对象的 *this* 指针来添加菜单项和工具条。

```
QPopupMenu * help = new QPopupMenu( this );
menuBar()->insertItem( "&Help", help );
    help->insertItem( "&About", this, SLOT(about()), Key_F1 );
```

这里我们添加了有一个菜单项的新菜单。这个菜单已经被插入 QMainWindow 默认提供的并且可以通过 menuBar()函数访问的菜单条。当这个菜单项被点击时，这个槽被调用。

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
QToolButton * fileOpen
= new QToolButton( openIcon, "Open File", QString::null,
        this, SLOT(choose()), fileTools, "open file" );
```

这部分提取显示的是有一个工具条按钮的工具条的创建。QMainWindow 为工具条提供了四个锚接区域。当一个工具条被作为 QMainWindow（或者继承类）实例 的子对象被创建时，它将会被放置到一个锚接区域中（默认是 Top 锚接区域）。当这个工具条按钮被点击时，这个槽被调用。任何锚接窗口可以使用 addDockWindow()，或通过把 QMainWindow 作为父对象来创建的方法来被添加到一个锚接区域中。

```
e = new QTextEdit( this, "editor" );
e->setFocus();
setCentralWidget( e );
statusBar()->message( "Ready", 2000 );
```

创建完菜单和工具条，我们创建一个大的中央窗口部件的实例，给它焦点并且把它设置为主窗口的中央窗口部件。在这个实例中，我们也已经通过 statusBar()函数设置好了状态条，显示初始信息两秒。注意你可以添加其它的窗口部件到状态条重，例如标签，来显示更多的状态信息。详细情况请参考 QStatusBar 文档，特别是 addWidget()函数。

通常我们想让一个工具条按钮和一个菜单项同步。例如，如果用户点击“ 加粗 ”工具条按钮，我们希望“ 加粗 ”菜单项被选中。这种同步可以通过创建操作并且把它们添加到工具条和菜单上来自动实现。

```
QAction * fileOpenAction;
fileOpenAction = new QAction( "Open File", QPixmap( fileopen ),
"&Open", CTRL+Key_O, this, "open" );
connect( fileOpenAction, SIGNAL( activated() ) , this,
SLOT( choose() ) );
```

这里我们创建了一个有图标的操作，这个图标要用在这个操作所被添加到的菜单和工具条中。我们也给定这个操作一个菜单名称“ &Open ”和一个键盘快捷键。我们已经建立的这个连接在用户点击这个菜单项*或者*这个工具条按钮时将会被使用。

```
QPopupMenu * file = new QPopupMenu( this );
menuBar()->insertItem( "&File", file );
fileOpenAction->addTo( file );
```

上面这部分提取显示一个弹出菜单的创建。我们把这个菜单添加到 QMainWindow 的菜单条中并且添加我们的操作。

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
fileOpenAction->addTo( fileTools );
```

这里我们创建一个作为 QMainWindow 的子对象的工具条并且把我们的操作添加到这个工具条中。 典型的 QMainWindow 的图片如下图所示：

更多更详细的说明，请参照相关文档中的关于 QMainWindow 的定义。

实验五 工具条和状态栏

实验目的：

在实验 4 中，我们已经设计出了一个有菜单栏、菜单和快捷键的主窗口部件，这次实验我们将在实验 4 的基础上，进一部完善窗口风格的部件。我们将在实验 5 中添加工具条和状态栏。

实验代码：

mainwidget.h:

```
/****************************************************************************
********
** $Id: /sample/5/mainwidget.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
****************************************************************************
********/

#ifndef _MAINWIDGET_H_
#define _MAINWIDGET_H_

#include <qapplication.h>
#include <qmainwindow.h>
#include <qpopupmenu.h>
#include <qmenubar.h>
#include <qlabel.h>
#include <qtoolbar.h>
#include <qtoolbutton.h>
#include <qstatusbar.h>


class MainWidget: public QMainWindow
{
        Q_OBJECT
public:
        MainWidget( QWidget *parent=0, const char *name=0 );
public slots:
        void openFile();
        void saveFile();
        void exitMain();
private:
```

```
        QLabel *label;
};

#endif
```

mainwidget.cpp:

```
/****************************************************************************
********
** $Id: /sample/5/mainwidget.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
********/

#include "mainwidget.h"

MainWidget::MainWidget( QWidget *parent, const char *name )
            :QMainWindow( parent, name )
{
        setCaption( "qt_Example" );
        setBackgroundColor( white );

        QFont f( "Helvetica", 18, QFont::Bold );
        setFont( f );

        label = new QLabel( "", this );
        label->setGeometry( 50, 50, 250, 50 );
        label->setBackgroundColor( white );

        QPopupMenu *file = new QPopupMenu;
        QFont f1( "Helvetica", 14, QFont::Bold );
         setFont( f1 );
        file->setFont( f1 );
        file->insertItem( "&Open", this, SLOT( openFile() ),
CTRL+Key_O );
        file->insertItem( "&Save", this, SLOT( saveFile() ), CTRL+Key_S );

        int id_save = file->insertItem( "&Save", this, SLOT( saveFile() ) );
        file->setItemEnabled( id_save, FALSE );
```

```
        file->insertItem( "E&xit", this, SLOT( exitMain() ),
CTRL+Key_X );

        QMenuBar *menu;
        menu = new QMenuBar( this );
        menu->insertItem( "&File", file );

        QToolBar *tools = new QToolBar( "example", this );
        QPixmap exitIcon( "exit.xpm" );

        QToolButton *exitmain = new QToolButton( exitIcon, "Exit", 0,
                this, SLOT( exitMain() ),
                tools, "exit" );

        statusBar()->message( "Ready");
}

void MainWidget::openFile()
{
        label->setText( "File has been opened!" );
        statusBar()->clear();
        statusBar()->message( "Opened" );
}

void MainWidget::saveFile()
{
        label->setText( "File has been saved!" );
        statusBar()->clear();
         statusBar()->message( "Saved" );
}

void MainWidget::exitMain()
{
        QApplication::exit();
}
```

main.cpp:

```
/****************************************************************************
********
** $Id: /sample/5/main.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
**
```

```
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************
********/

#include <qapplication.h>
#include "mainwidget.h"

int main( int argc, char **argv )
{
        QApplication app( argc, argv );
        MainWidget *mainwidget = new MainWidget( 0 );
        mainwidget->setGeometry(10, 30, 280, 200 );
        app.setMainWidget( mainwidget );
        mainwidget->show();
        int result = app.exec();
        return result;
}
```

实验原理：

与实验 4 中相比，实验 5 在结构上并没有太大的改变，只是添加了一些内容。因此实验 5 可以看作是实验 4 的升级。头文件 mainwidget.h 与实验 4 的头文件 mainwidget.h 并无太大差别，只是添加了包含 qtoolbar.h、qtoolbutton.h、qstatusbar.h 的声明，其它一切没有改动。因此不加解释。若有疑问，请参照实验 4 关于 mainwidget.h 的说明。

在 mainwidget.cpp 中：

```
setCaption( "qt_Example" );
setBackgroundColor( white );

QFont f( "Helvetica", 18, QFont::Bold );
setFont( f );

label = new QLabel( "", this );
label->setGeometry( 50, 50, 250, 50 );
label->setBackgroundColor( white );

QPopupMenu *file = new QPopupMenu;
QFont f1( "Helvetica", 14, QFont::Bold );
setFont( f1 );
file->setFont( f1 );
file->insertItem( "&Open", this, SLOT( openFile() ), CTRL+Key_O );
file->insertItem( "&Save", this, SLOT( saveFile() ), CTRL+Key_S );

int id_save = file->insertItem( "&Save", this, SLOT( saveFile() ) );
file->setItemEnabled( id_save, FALSE );
```

```
file->insertItem( "E&xit", this, SLOT( exitMain() ), CTRL+Key_X );

QMenuBar *menu;
menu = new QMenuBar( this );
menu->insertItem( "&File", file );
```

上述语句均与实验 4 相同，就是设置了一下窗口的环境，添加了一个菜单和一些菜单项。可参照实验 4。

```
QToolBar *tools = new QToolBar( "example", this );
```

QToolBar 类提供了可以包含工具按钮这类的窗口部件的可移动的面板。

工具条是包含一套控制的面板，通常表现为小图标。它用于提供常用命令或者选项的快速访问。在 QMainWindow 里面，用户可以锚接区域里面或者之间拖动工具条。工具条也可以被拖动出任何锚接区域而作为顶级窗口自由浮动。

QToolBar 是 QDockWindow 的特殊化，并且提供 QDockWindow 的所有功能。

为了使用 QToolBar，你可以简单的把 QToolBar 创建为 QMainWindow 的子对象，从左到右（或者从上到下）创建许多 QToolButton 窗口部件（或者其他窗口部件）并且当你想要一个分隔符，请调用 addSeparator()。当工具条被浮动时，标题会使用在构造函数中给定的标签。这个可以通过 setLabel()来改变。

```
QToolBar * fileTools = new QToolBar( this, "file
operations" );
fileTools->setLabel( "File Operations" );
fileOpenAction->addTo( fileTools );
fileSaveAction->addTo( fileTools );
```

这个实例显示了把新的工具条创建为 QMainWindow 的子对象并且添加两个 QAction。你可以在工具条中使用绝大多数窗口部件，最常用的时 QToolButton 和 QComboBox。注意如果你在一个已经可视的 QToolBar 上创建一个新的窗口部件，这个窗口部件可以在没有明显调用 show()的情况下就会自动变为可视（这和 Qt 中其他窗口部件容器不一样）。总之，请为你放入可视的 QToolBar 的所有窗口部件明显地调用 show()，这个行为也许在未来会被确定下来。

QToolBar，像 QDockWindow 一样，可以被定位在 QDockArea 中或者像顶级窗口一样浮动。QMainWindow 提供了四个 QDockArea（上、下、左、右）。当你创建一个新工具条（像上述实例一样）作为 QMainWindow 的子对象时，这个工具条会被添加到上面的锚接区域。你可以通过调用 QMainWindow::moveDockWindow()把它移动到其它锚接区域（或者浮动起来）。QDockarea 在行中布局它们的窗口。

如果主窗口被重新定义大小，这样工具条所占的区域会变小以至于不能显示所有窗口部件，这时一个小箭头按钮（看起来像指向右的 V 形符号，“ » ”）将会根据工具条的方向显示在工具条的右面或者下面。点击这个按钮将会弹出一个菜单来显示那些“ 过剩的 ”项。

通常工具条将会得到它所需要的空间。可是，通过 setHorizontalStretchable()、setVerticalStretchable()或 setStretchableWidget()，你可以告诉主窗口在指定的方向延伸工具条来填充所有可用的空间。

工具条在水平方向或者垂直方向（详细情况请参考 orientation()）上排列它的按钮。通常 QDockArea 将会为你设置正确的方向，但是你可以使用 setOrientation()来自己设置它并且通过连接到 orientationChanged()信号来跟踪任何变化。你可以使用 clear()方法来移除工具条的所有条目。

典型的 QToolBar 如下图所示：

QtoolBar 的构造函数有如下两个：QToolBar::QToolBar (const QString & label, QMainWindow * mainWindow, QWidget * parent, bool newLine = FALSE, const char * name = 0, WFlags f = 0) 构造一个空的水平工具条。

工具条是 parent 的子对象并且被 mainWindow 管理。label 和 newLine 参数被直接传递给 QMainWindow::addDockWindow()。name 是对象名称并且 *f* 是窗口部件标记。

如果你想创建已经取消锚接正在浮动的工具条或者状态条中的工具条时请使用这个构造函数。

QToolBar::QToolBar (QMainWindow * parent = 0, const char * name = 0)

这是一个重载成员函数，提供了方便。它的行为基本上和上面的函数相同。构造一个在 parent 的上面锚接区域中的父对象为 parent 并且名称为 name 的空工具条。没有任何标签并且不需要一个新行。因此，QToolBar *tools = new QToolBar("example", this)这条语句的主要作用是在本窗口中构造了一个标签为 example 的工具条。

更多关于 QToolBar，请参照相关文档中的完整定义。

QPixmap exitIcon("exit.xpm");

QPixmap 是一个基于象素的脱离屏幕的画图工具类。

QPixmap 是 Qt 提供的处理图形的两个类中的一个，另外一个是 QImage。QPixmap 为画图专门设计和最优化，QImage 为 I/O、象素访问和操作专门设计和最优化。有几个函数用于 QPixmap 和 QImage 之间互相转换（缓慢的），分别是：convertToImage() 和 convertFromImage()。

一个经常使用 QPixmap 类的地方是使能窗口部件的平滑更新，无论何时有画一个复杂的图形，用户都可以如下面一样使用 Pixmap 去获得无闪烁的图画：

1. 定义一个跟窗口部件一般大小的象素映射
2. 用窗口部件的背景色去填充象素映射
3. 画象素映射
4. 在窗口部件之上用 bitBit()象素映射的内容

象素映射中的象素数据是固有的，并且是通过底层的窗口系统来管理的。只有通过 Qpainter 的函数、bitBit()或者将 QPixmap 转换成 QImage 才能访问象素。用户可以使用 QLabel::setPixmap()轻松的在屏幕上显示象素映射。例如：所有的 QButton 的子类都支持使用象素映射。因为 QPixmap 类使用“写时复制”，应此它实际上使通过值来传递 QPixmap 类的对象。

在本实验中，我们用 QPixmap 来定义了一个图标，该图标的内容为保存在文件“exit.xpm”中，该文件是一个可读的文本，如下：

```
/* XPM */
static char *exit_xpm[] = {
/* columns rows colors chars-per-pixel */
"24 24 14 1",
"    c Gray0",
". c Gray3",
"X c #2d2d2d2d2d2d",
"o c #8a8a4d4d4f4f",
```

```
"O c #9e9e57575b5b",
"+ c #a7a75e5e5f5f",
"@ c #afaf66666868",
"# c #b6b66d6d7171",
"$ c #bebe75757979",
"% c #c6c67d7d8181",
"& c #cdcd86868989",
"* c #e4e4e4e4dcdc",
"= c Gray100",
"- c None",
/* pixels */
"------------------------",
"------------------------",
"------------------------",
"------------------------",
"------------------------",
"----------               -----",
"---------- ******    -----",
"---------- ******    -----",
"--------      ******    -----",
"-------- $ ******    -----",
"---           $% **X *    -----",
"--- oO+@#$%& * X*    -----",
"--- oO+@#$%& ****    -----",
"---           $% *****    -----",
"-------- $ ******    -----",
"--------      ******    -----",
"---------- ****       -----",
"---------- **          -----",
"----------      ......-----",
"------------------------",
"------------------------",
"------------------------",
"------------------------",
"------------------------"
};
```

XPM（XPixMap）图形格式在 X11 中是一个标准模式，它的图形保存成 ASCII 文本。这种格式允许用户自己的文本编辑器创建或者修改简单的有颜色的图形。一个 XPM 的定义不仅是 ASCII 格式，它的格式还可以是 C 的源代码，可以直接将它们编辑到自己的程序当中。详细情况参见相关文档关于 XPM 格式的介绍。

```
QToolButton *exitmain = new QToolButton( exitIcon, "Exit", 0,
        this, SLOT( exitMain() ), tools, "exit" );
```

QToolButton 类提供了用于命令或者选项的可以快速访问的按钮，通常可以用在 QToolBar 里面。

工具按钮是提供对特定命令或者选项快速访问的特殊按钮。和普通的命令按钮不同，工具按钮通常不显示文本标签，而是图标。它的经典用法事选择工具，例如在一个绘图程序中的“ 笔 ”工具。这个被 QToolButton 作为切换按钮重新实现。(请参考 setToggleButton())。

QToolButton 支持自动浮起。在自动浮起模式中，按钮只有在鼠标指向它的时候才绘制三维的框架。当按钮被用在 QToolBar 里面的时候，这个特征会自动被启用。可以使用 setAutoRaise()来改变它。

工具按钮的图标是被设置为 QIconSet。这使得它可以为失效和激活状态指定不同的像素映射。当按钮的功能不可用的时候，失效的像素映射被使用。当因为用户用鼠标指向按钮而自动浮起时，激活的像素映射被显示。

按钮的外观和尺寸可以通过 setUsesBigPixmap()和 setUsesTextLabel()来调节。当被用在 QToolBar 里面时，按钮会自动地调节来适合 QMainWindow 的设置（请参考 QMainWindow::setUsesTextLabel()和 QMainWindow::setUsesBigPixmaps()）。

工具按钮可以提供一个弹出菜单的额外选择。这个特征有时对于网页浏览器中的“ 后退 ”按钮是有用的。在按下按钮一段时间之后，一个菜单会弹出来显示所有可以后退浏览的可能页面。你可以使用 setPopup()来为 QToolButton 设置一个弹出菜单。默认延时是 600 毫秒，你可以使用 setPopupDelay()来调节它。

QtoolButton 的三个构造函数如下：

QToolButton::QToolButton (QWidget * parent, const char * name = 0)

构造一个父对象为 parent 并且名称为 name 的空工具按钮。

QToolButton::QToolButton (const QIconSet & iconSet,
const QString & textLabel, const QString & grouptext, QObject * receiver,
const char * slot, QToolBar * parent, const char * name = 0)

构造一个父对象为 parent（必须为 QToolBar）并且名称为 name 的工具按钮。工具按钮将显示 iconSet，它的文本标签和工具提示设置为 textLabel 并且它的状态条信息设置为 grouptext。它将被连接到 receiver 对象的 slot 槽。

QToolButton::QToolButton (ArrowType type, QWidget*parent,const char * name = 0)

把工具按钮构造为箭头按钮。ArrowType type 定义了箭头的方向。可用的值为 LeftArrow、RightArrow、UpArrow 和 DownArrow。箭头按钮的自动重复默认是打开的。parent 和 name 参数被发送给 QWidget 构造函数。因此在本程序中，我们构造了一个标签为“ Exit ”、图标为“ exit.xpm ”中的图标的工具按钮，并且将发射的信号与槽 exitMain()连接在一起，当用户点击该按钮时，执行 exitMain()中的代码。

statusBar()->message("Ready");

statusBar()函数在 QMainWindow 中定义过，它回这个窗口的状态条 QStatusBar。如果没有的话，statusBar()会创建一个空的状态条，并且如果需要也创建一个工具提示组。

QStatusBar 类提供了一个适合呈现状态信息的水平条。每一个状态指示器都会落在下面这三种类别之内：

- ✧ 临时的 - 暂时地占用状态条的大部分。例如，用于解释工具提示文本或者菜单条目。
- ✧ 正常的 - 占用状态条的一部分并且也可能被临时的信息隐藏。例如，用于在字处理器中显示页数和行数。
- ✧ 永久的 - 从不被隐藏。用于重要的模式指示，例如，一些程序把大小写指示器放在状态条中。

QStatusBar 让你能够显示上述所有类型的指示信息。

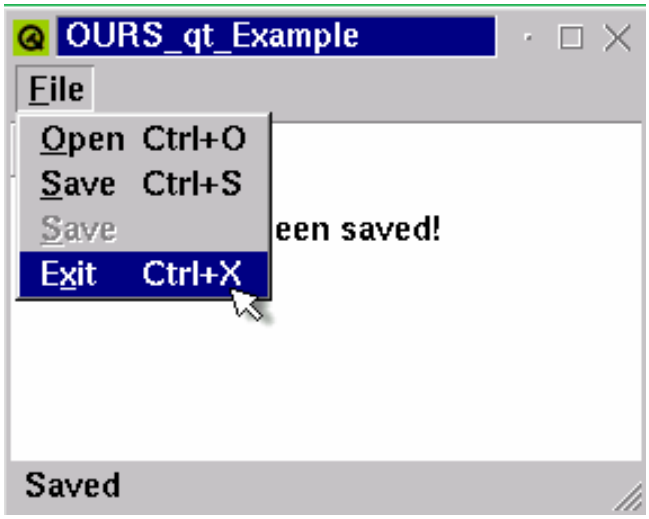
为了显示临时的消息，请调用 message()（可以把一个合适的信号和它连接起来）。如果要移除一个临时的消息，调用 clear()。这里有两类消息：一类消息一直显示到下一个 clear() 或 mesage()被调用才消失，并且另一种是有时间限制的：

```
connect( loader, SIGNAL(progressMessage(const QString&)),
statusBar(), SLOT(message(const QString&)) );
statusBar()->message("Loading...");   // 初始消息
loader.loadStuff();                                    // 发射进程消息
statusBar()->message("Done.", 2000); // 显示 2 秒的最后消息
```

应此该语句其实是使窗口产生了一个临时的状态栏，并且在状态栏上显示了一些信息。

```
void MainWidget::openFile()
{
    label->setText( "File has been opened!" );
    statusBar()->clear();
    statusBar()->message( "Opened" );
}

void MainWidget::saveFile()
{
    label->setText( "File has been saved!" );
    statusBar()->clear();
     statusBar()->message( "Saved" );
}

void MainWidget::exitMain()
{
    QApplication::exit();
}
```

上面四个函数与实验 4 中并无太大区别，这是在函数中添加了相应的状态信息，并在状态栏中显示出来。main.cpp 与实验 4 无区别。程序编译完在 Qvfb 上运行的结果如图所示：



实验 5 工具栏和状态栏

实验小结：

本实验在实验 4 的基础上，添加了两个窗口应用程序常用的元素：工具栏和状态栏。应该来说，几乎所有的复杂一点的应用程序的界面都会设计到这两个元素。本次实验简单起见，只是添加了简单的工具栏和状态栏，但是基本框架已经建立起来，用户可以沿这这个思路继续丰富自己的程序。另外阅读一下实验中介绍的几个类的完整定义也是相当有必要的。详情参加相关文档。

实验六 鼠标和键盘事件

实验目的：

在一般的 GUI 程序中，鼠标和键盘是最主要的输入工具。如果不能很好的处理鼠标事件和键盘事件，应用程序的友好性将会大打折扣。本实验将重点讲述 Qt 是如何提供这一类的解决方法。

实验代码：

mousekeyevent.h:

```
/****************************************************************
***************
** $Id: /sample/6/mousekeyevent.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#ifndef MOUSEKEYEVENT_H
#define MOUSEKEYEVENT_H


#include <qlabel.h>
#include <qevent.h>
#include <qstring.h>

class MouseKeyEvent: public QWidget
{
    Q_OBJECT
public:
    MouseKeyEvent( QWidget *parent=0, const char *name=0 );
protected:
    void mousePressEvent( QMouseEvent* );
    void mouseMoveEvent( QMouseEvent* );
    void keyPressEvent( QKeyEvent* );
private:
    QLabel *label;
};

#endif
```

mousekeyevnet.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/6/mousekeyevent.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #include "mousekeyevent.h"

    MouseKeyEvent::MouseKeyEvent( QWidget *parent, const char *name)
            :QWidget( parent, name )
    {
        setCaption( "qt_Example" );
        setBackgroundColor( white );

        label = new QLabel( "Wellcome!", this );
        label->setBackgroundColor( white );
        QFont f( "Helvetica", 16, QFont::Bold );
        label->setFont( f );
        label->setGeometry( 25, 70, 250, 100 );
        label->setAlignment( AlignVCenter | AlignHCenter );

    }


    void MouseKeyEvent::mousePressEvent( QMouseEvent *e )
    {
        switch( e->button() )
        {
            case LeftButton:
                label->clear();
                label->setText( "Mouse:LeftButton Pressed!" );
                break;
            case RightButton:
                label->clear();
                                label->setText( "Mouse:RightButton Pressed!" );
                break;
            case MidButton:
                label->clear();
                label->setText( "Mouse:MidButton Pressed!" );
```

```
                break;
            default:
                label->clear();
                label->setText( "Mouse:Undefined Pressed!" );
                break;
        }
}


void MouseKeyEvent::mouseMoveEvent( QMouseEvent *e )
{
    QString str = QString( "X:" );
    QString ps = "";
    ps = ps.setNum( e->x() );
    str += ps;
    str += "      Y:";
    ps = "";
    ps = ps.setNum( e->y() );
    str += ps;
    label->clear();
    label->setText( str );
}


void MouseKeyEvent::keyPressEvent( QKeyEvent *e )
{
    switch( e->key() )
    {
        case Key_Escape:
                            label->clear();
                            label->setText( "Key:Esc Pressed!" );
                            break;
        case Key_Tab:
                            label->clear();
                            label->setText( "Key:Tab Pressed!" );
                            break;
        case Key_Backtab:
                            label->clear();
                            label->setText( "Key:BackTab Pressed!" );
                            break;
        case Key_BackSpace:
                            label->clear();
                            label->setText( "Key:BackSpace Pressed!" );
                            break;
```

```
case Key_Return:
                label->clear();
                label->setText( "Key:Return Pressed!" );
                break;
case Key_Enter:
                label->clear();
                label->setText( "Key:Enter Pressed!" );
                break;
case Key_Insert:
                label->clear();
                label->setText( "Key:Insert Pressed!" );
                break;
case Key_Delete:
                label->clear();
                label->setText( "Key:Delete Pressed!" );
                break;
case Key_Pause:
                label->clear();
                label->setText( "Key:Pause Pressed!" );
                break;
case Key_Print:
                label->clear();
                label->setText( "Key:Print Pressed!" );
                break;
case Key_SysReq:
                label->clear();
                label->setText( "Key:SysReq Pressed!" );
                break;
case Key_Home:
                label->clear();
                label->setText( "Key:Home Pressed!" );
                break;
case Key_End:
                label->clear();
                label->setText( "Key:End Pressed!" );
                break;
case Key_Left:
                label->clear();
                label->setText( "Key:Left Pressed!" );
                break;
case Key_Up:
                label->clear();
                label->setText( "Key:Up Pressed!" );
                break;
```

```
case Key_Right:
                label->clear();
                label->setText( "Key:Right Pressed!" );
                break;
case Key_Down:
                label->clear();
                label->setText( "Key:Down Pressed!" );
                break;
case Key_PageUp:
                label->clear();
                label->setText( "Key:PageUp Pressed!" );
                break;
case Key_PageDown:
                label->clear();
                label->setText( "Key:PageDown Pressed!" );
                break;
case Key_Shift:
                label->clear();
                label->setText( "Key:Shift Pressed!" );
                break;
case Key_Control:
                label->clear();
                label->setText( "Key:Control Pressed!" );
                break;
case Key_Meta:
                label->clear();
                label->setText( "Key:Meta Pressed!" );
                break;
case Key_Alt:
                label->clear();
                label->setText( "Key:Alt Pressed!" );
                break;
case Key_CapsLock:
                label->clear();
                label->setText( "Key:CapsLock Pressed!" );
                break;
case Key_NumLock:
                label->clear();
                label->setText( "Key:NumLock Pressed!" );
                break;
case Key_ScrollLock:
                label->clear();
                label->setText( "Key:ScrollLock Pressed!" );
                break;
```

```
        case Key_F1:
                label->clear();
                label->setText( "Key:F1 Pressed!" );
                break;
    case Key_F2:
                label->clear();
                label->setText( "Key:F2 Pressed!" );
                break;
    case Key_F3:
                label->clear();
                label->setText( "Key:F3 Pressed!" );
                break;
    case Key_F4:
                label->clear();
                label->setText( "Key:F4 Pressed!" );
                break;
    case Key_F5:
                label->clear();
                label->setText( "Key:F5 Pressed!" );
                break;
    case Key_F6:
                label->clear();
                label->setText( "Key:F6 Pressed!" );
                break;
    case Key_F7:
                label->clear();
                label->setText( "Key:F7 Pressed!" );
                break;
    case Key_F8:
                label->clear();
                label->setText( "Key:F8 Pressed!" );
                break;
    case Key_F9:
                label->clear();
                label->setText( "Key:F9 Pressed!" );
                break;
        case Key_F10:
                label->clear();
                label->setText( "Key:F10 Pressed!" );
                break;
    case Key_F11:
                label->clear();
                label->setText( "Key:F11 Pressed!" );
                break;
```

```
case Key_F12:
                label->clear();
                label->setText( "Key:F12 Pressed!" );
                break;
case Key_0:
                label->clear();
                label->setText( "Key:0 Pressed!" );
                break;
case Key_1:
                label->clear();
                label->setText( "Key:1 Pressed!" );
                break;
case Key_2:
                label->clear();
                label->setText( "Key:2 Pressed!" );
                break;
case Key_3:
                label->clear();
                label->setText( "Key:3 Pressed!" );
                break;
case Key_4:
                label->clear();
                label->setText( "Key:4 Pressed!" );
                break;
case Key_5:
                label->clear();
                label->setText( "Key:5 Pressed!" );
                break;
case Key_6:
                label->clear();
                label->setText( "Key:6 Pressed!" );
                break;
case Key_7:
                label->clear();
                label->setText( "Key:7 Pressed!" );
                break;
case Key_8:
                label->clear();
                label->setText( "Key:8 Pressed!" );
                break;
case Key_9:
                label->clear();
                label->setText( "Key:9 Pressed!" );
                break;
```

```
case Key_A:
                label->clear();
                label->setText( "Key:A Pressed!" );
                break;
case Key_B:
                label->clear();
                label->setText( "Key:B Pressed!" );
                break;
case Key_C:
                label->clear();
                label->setText( "Key:C Pressed!" );
                break;
case Key_D:
                label->clear();
                label->setText( "Key:D Pressed!" );
                break;
case Key_E:
                label->clear();
                label->setText( "Key:E Pressed!" );
                break;
case Key_F:
                label->clear();
                label->setText( "Key:F Pressed!" );
case Key_G:
                label->clear();
                label->setText( "Key:G Pressed!" );
                break;
case Key_H:
                label->clear();
                label->setText( "Key:H Pressed!" );
                break;
case Key_I:
                label->clear();
                label->setText( "Key:I Pressed!" );
                break;
case Key_J:
                label->clear();
                label->setText( "Key:J Pressed!" );
                break;
case Key_K:
                label->clear();
                label->setText( "Key:K Pressed!" );
                break;
case Key_L:
```

```
                label->clear();
                label->setText( "Key:L Pressed!" );
                break;
case Key_M:
                label->clear();
                label->setText( "Key:M Pressed!" );
                break;
case Key_N:
                label->clear();
                label->setText( "Key:N Pressed!" );
                break;
case Key_O:
                label->clear();
                label->setText( "Key:O Pressed!" );
                break;
case Key_P:
                label->clear();
                label->setText( "Key:P Pressed!" );
                break;
case Key_Q:
                label->clear();
                label->setText( "Key:Q Pressed!" );
                break;
case Key_R:
                label->clear();
                label->setText( "Key:R Pressed!" );
                break;
case Key_S:
                label->clear();
                label->setText( "Key:S Pressed!" );
                break;
case Key_T:
                label->clear();
                label->setText( "Key:T Pressed!" );
                break;
case Key_U:
                label->clear();
                label->setText( "Key:U Pressed!" );
                break;
case Key_V:
                label->clear();
                label->setText( "Key:V Pressed!" );
                break;
case Key_W:
```

```
                                label->clear();
                                label->setText( "Key:W Pressed!" );
                                break;
        case Key_X:
                                label->clear();
                                label->setText( "Key:X Pressed!" );
                                break;
        case Key_Y:
                                label->clear();
                                label->setText( "Key:Y Pressed!" );
                                break;
        case Key_Z:
                                label->clear();
                                label->setText( "Key:Z Pressed!" );
                                break;
        default:
                                label->clear();
                            label->setText( "Key:Undefined key
        Pressed!" );
                                break;
            }
        }

        main.cpp:
/****************************************************************************
*******
        ** $Id: /sample/6/main.cpp      2.3.2      edited 2004-10-12 $
        **
        ** Copyright (C) 2004-2005 AS.    All rights reserved.
        **
        ** This file is part of an example program for Qt.    This example
        ** program may be used, distributed and modified without limitation.
        **
****************************************************************************
*******/

        #include <qapplication.h>
        #include "mousekeyevent.h"

        int main( int argc, char **argv )
        {
            QApplication app( argc, argv );
            MouseKeyEvent *mousekeyevent = new MouseKeyEvent( 0 );
            mousekeyevent->setGeometry( 10, 20, 320, 240 );
```

```
    app.setMainWidget( mousekeyevent );
    mousekeyevent->show();
    int result = app.exec();
    return result;
}
```

实验原理：

我们逐步来分析一下源代码。

在 mousekeyevent.h 中，包含的头文件有 qlabel.h、qevent.h、qstring.h。其中我们比较关注的是 qevent.h。在 qvent.h 中，定义了一个 QEvent 的类。QEvent 类是所有事件类的基类。事件类包含事件参数。

Qt 的主事件回路（QApplication::exec()）从事件队列里取得本地窗口系统事件，并把它们转换为 QEvent 并且把这些转换过的事件发给 QObject。

通常情况下，事件来自于窗口系统（spontaneous()返回真），但是它也可以使用 QApplication::sendEvent()和 QApplication::postEvent()手动发送事件（spontaneous()返回假）。

QObject 通过它们的 QObject::event()函数调用来接收事件。这个函数可以在子类中重新实现来处理自定义的事件和添加额外的事件类型，QWidget::event()就是一个著名的例子。默认情况下，像 QObject::timerEvent()和 QWidget::mouseMoveEvent()这样的事件可以被发送给事件处理函数。QObject::installEventFilter()允许一个对象中途截取发往另一个对象的事件。

基本的 QEvent 只包含了一个事件类型参数。QEvent 的子类包含了额外的描述特定事件的参数。

在 QEvent 中，枚举了所有 Qt 中有效的事件类型，事件类型和其相应的事件类列举如下：

- ✧ QEvent::None - 不是一个事件。
- ✧ QEvent::Accessibility - 可存取性信息被请求。
- ✧ QEvent::Timer - 规则的定时器事件，QTimerEvent。
- ✧ QEvent::MouseButtonPress - 鼠标按下，QMouseEvent。
- ✧ QEvent::MouseButtonRelease - 鼠标抬起，QMouseEvent。
- ✧ QEvent::MouseButtonDblClick - 鼠标再次按下，QMouseEvent。
- ✧ QEvent::MouseMove - 鼠标移动，QMouseEvent。
- ✧ QEvent::KeyPress - 键按下（举例，包括 Shift）QKeyEvent。
- ✧ QEvent::KeyRelease - 键抬起，QKeyEvent。
- ✧ QEvent::IMStart - 输入法写作开始。
- ✧ QEvent::IMCompose - 发生输入法写作。
- ✧ QEvent::IMEnd - 输入法写作结束。
- ✧ QEvent::FocusIn - 窗口部件获得键盘焦点，QFocusEvent。
- ✧ QEvent::FocusOut - 窗口部件失去键盘焦点，QFocusEvent。
- ✧ QEvent::Enter - 鼠标进入窗口部件边缘。
- ✧ QEvent::Leave - 鼠标离开窗口部件边缘。
- ✧ QEvent::Paint - 屏幕更新所需要的，QPaintEvent。
- ✧ QEvent::Move - 窗口部件位置改变了，QMoveEvent。
- ✧ QEvent::Resize - 窗口部件大小改变了，QResizeEvent。
- ✧ QEvent::Show - 窗口部件被显示到屏幕上，QShowEvent。

- QEvent::Hide - 窗口部件被隐藏，QHideEvent。
- QEvent::ShowToParent - 一个子窗口部件被显示。
- QEvent::HideToParent - 一个子窗口部件被隐藏。
- QEvent::Close - 窗口部件被关闭（永久性地） QCloseEvent。
- QEvent::ShowNormal - 窗口部件应该按通常模式显示。
- QEvent::ShowMaximized - 窗口部件应该按最大化模式显示。
- QEvent::ShowMinimized - 窗口部件应该按最小化模式显示。
- QEvent::ShowFullScreen - 窗口部件应该按全屏模式显示。
- QEvent::ShowWindowRequest - 窗口部件窗口应该被显示。**这个类型是废弃的。**它的提供是为了保证旧代码能够工作。我们强烈建议在新代码中不要使用它。
- QEvent::DeferredDelete - 在这个对象被清理干净之后，它将被删除。
- QEvent::Accel -子键按下，用于快捷键处理。QKeyEvent。
- QEvent::Wheel - 鼠标滚轮转动，QWheelEvent。
- QEvent::ContextMenu - 上下文弹出菜单，QContextMenuEvent。
- QEvent::AccelAvailable - 在一些平台上 Qt 使用的内部事件。
- QEvent::AccelOverride - Key press in child, for overriding shortcut key handling, QKeyEvent.
- QEvent::WindowActivate - 窗口被激活了。
- QEvent::WindowDeactivate - 窗口被停用了。
- QEvent::CaptionChange - 窗口部件的标题改变了。
- QEvent::IconChange - 窗口部件的图标改变了。
- QEvent::ParentFontChange - 父窗口部件的字体改变了。
- QEvent::ApplicationFontChange - 默认的应用程序字体改变了。
- QEvent::PaletteChange - 窗口部件的调色板改变了。
- QEvent::ParentPaletteChange - 父窗口部件的调色板改变了。
- QEvent::ApplicationPaletteChange - 默认的应用程序调色板改变了。
- QEvent::Clipboard - 剪贴板内容发生改变，QClipboard。
- QEvent::SockAct - 套接字触发，通常在 QSocketNotifier 中实现。
- QEvent::DragEnter - 一个拖拽进入了一个窗口部件，QDragEnterEvent。
- QEvent::DragMove - 一个拖拽正在进行中，QDragMoveEvent。
- QEvent::DragLeave - 一个拖拽离开了窗口部件，QDragLeaveEvent。
- QEvent::Drop - 一个拖拽完成了，QDropEvent。
- QEvent::DragResponse - 在一些平台上 Qt 使用的内部事件。
- QEvent::ChildInserted - 对象得到了一个子类，QChildEvent。
- QEvent::ChildRemoved - 对象失去了一个子类，QChildEvent。
- QEvent::LayoutHint - 窗口部件子类改变了布局属性。
- QEvent::ActivateControl - 在一些平台上 Qt 使用的内部事件。
- QEvent::DeactivateControl - 在一些平台上 Qt 使用的内部事件。
- QEvent::Quit - 保留的。
- QEvent::Create - 保留的。
- QEvent::Destroy - 保留的。
- QEvent::Reparent - 保留的。
- QEvent::Speech - 为语音输入而保留的。
- QEvent::Tablet - Wacom Tablet 事件。

- ✧ QEvent::User - 用户定义事件。
- ✧ QEvent::MaxUser - 最后用户事件 id。

从上面可以看出，QEvent 类为 Qt 定义了丰富的事件，我们这次实验所要使用到的鼠标和键盘事件也包括其中。在处理鼠标事件的 QMouseEvent 类中，定义了四个不同的关于鼠标的事件，分别为 mouseReleaseEvent、mouseDoubleClickEvent、mousePressEvent、mouseMoveEvent()。它们分别对应着鼠标释放、鼠标双击、鼠标按下、和鼠标移动事件。因此在由 QWidget 类派生的类 MouseKeyEvent 类的定义中， 我们重载了两个关于捕获鼠标事件的函数。

```
void mousePressEvent( QMouseEvent* );
void mouseMoveEvent( QMouseEvent* );
```

从命名上，我们可以直观的看出，我们将可以捕获到鼠标事件中的按下和移动事件。同样的，处理键盘事件的类有 QKeyEvent。在 QKeyEvent 类里，定义了两个关于键盘的事件，为 keyPressEvent 和 keyReleaseEvent。对应的为键盘的按下和释放事件。因此我们在程序中，重载了一个关于捕获键盘事件的函数。

```
void keyPressEvent( QKeyEvent* );
```

于是，键盘每次的按下都会被程序捕获到。

```
QLabel *label;
```

最后，同样的我们定义为类 MouseKeyEvent 定义了一个私有成员——标签部件，用来标示当前事件发生的具体类型。

总结一下事件处理的方法：一个从 QWidget 派生的类只需要重载对应虚函数***Event()就可以捕获相应的事件。当该对象发生了某个事件的时候，该函数就会被执行，并可以通过函数的入口参数来表明具体的事件类型。我们将在 mousekeyevent.cpp 中看到这些函数的实现。

```
setCaption( "qt_Example" );
setBackgroundColor( white );

label = new QLabel( "Wellcome!", this );
label->setBackgroundColor( white );
QFont f( "Helvetica", 16, QFont::Bold );
label->setFont( f );
label->setGeometry( 25, 70, 250, 100 );
label->setAlignment( AlignVCenter | AlignHCenter );
```

在构造函数的实现的时候，我们对窗口的标题、背景、文字字形以及标签的字形、背景色和显示方式位置作出了一些设置。

```
void MouseKeyEvent::mousePressEvent( QMouseEvent *e )
{
    switch( e->button() )
    {
        case LeftButton:
            label->clear();
            label->setText( "Mouse:LeftButton Pressed!" );
            break;
        case RightButton:
            label->clear();
```

```
        label->setText( "Mouse:RightButton Pressed!" );
                break;
            case MidButton:
                label->clear();
                label->setText( "Mouse:MidButton Pressed!" );
                break;
            default:
                label->clear();
                label->setText( "Mouse:Undefined Pressed!" );
                break;
        }
    }
```

在我们重载的 mousePressEvent()函数中，我们针对按下时触发的事件通过入口参数来区分具体时那一种鼠标事件，即通过调用入口参数 QMouseEvent*的 button()方法：

```
ButtonState QMouseEvent::button() const
```

根据不同的鼠标事件，button()方法可能返回下面这些常用的值：

- ✧ LeftButton　　鼠标左键（也可以时左手型鼠标的右键）
- ✧ RightButton　　鼠标右键
- ✧ MidButton　　鼠标中键（三键鼠标）
- ✧ ShiftButton　　Shift＋鼠标键
- ✧ ControlButton　　Ctrl＋鼠标键
- ✧ AltButton　　Alt＋鼠标键

我们在程序中根据鼠标返回的不同值来判断事件的类型，并通过私有成员标签部件 label 来显示相应的状态。

```
    void MouseKeyEvent::mouseMoveEvent( QMouseEvent *e )
{
        QString str = QString( "X:" );
        QString ps = "";
        ps = ps.setNum( e->x() );
        str += ps;
        str += "      Y:";
        ps = "";
        ps = ps.setNum( e->y() );
        str += ps;
        label->clear();
        label->setText( str );
}
```

在另外一个重载的关于鼠标移动事件的函数中，我们通过调用 int QMouseEvent::x()和 int QMouseEvent::y()来得到当前鼠标的坐标值。并通过标签部件显示出来。

在实现的时候，我们使用了 Qstring 类，因为我们有可能会经常使用到这个类，在这里简单的介绍一下。

QString 类提供了一个 Unicode 文本和经典的 C 以零结尾的字符数组的抽象。QString 使用隐含共享，这使它非常有效率并且很容易使用。几乎我们能够想象的关于 string 的操作，在这里都很好的实现了。QString 重载了许多的运算符，因此可以很简单的进行串之间

的操作。例如串赋值、串比较、串连接。QString 还提供了诸如查找、插入、追加、替换等高级操作。而且 QString 还可以进行数字与字符串的相互转换操作。

更多的关于 QString 类的介绍，请参照相关文档。

处理键盘事件的入口函数与处理鼠标事件的类似，所有的 QWidget 的派生类都可以使用 keyPressEvent() 和 keyReleaseEvent() 处理键盘按下与放开事件。我们在重载 keyPressEvent()时，同样的通过入口参数 QKeyEvent 辨别具体的键盘值。并且通过相应的值来改变标签部件 label，显示相应的状态。

调用 QKeyEvent 类的 key()方法以判断具体的按键。与鼠标事件相比，key（）可能返回的值需要对应所用的键盘值。在头文件 qnamespace.h 中定义了所有的键盘键的值：

```
enum Key {
        Key_Escape = 0x1000,
        Key_Tab    = 0x1001,
        Key_Backtab = 0x1002, Key_BackTab = Key_Backtab,
        Key_Backspace = 0x1003, Key_BackSpace = Key_Backspace,
        Key_Return = 0x1004,
        Key_Enter = 0x1005,
        ......
        //有很多，非常长...
    }
```

在实现了几个重载函数后，对于鼠标和键盘事件的响应已经完成。

编译后在 Qvfb 中得到运行结果。

实验小结：

Qt 提供的完整的关于键盘和鼠标事件的解决方法，使得我们在程序的设计时难度大大降低。实际上，Qt 和在 MS-Windows 下进行的程序设计已经十分类似了。本次实验主要关注的是事件处理的框架，通过这次实验，用户应该能够应付自如的处理键盘和鼠标事件。用户可以通过修改重载函数里面的代码来实现自己所需要的功能。另外程序中所涉及到的 QString 类也是以后会经常使用的类，最好能够详细阅读相关文档中关于 QString 类的介绍。

实验七 对话框

实验目的：

在 GUI 编程的时候免不了要使用到对话框。对话框是要求用户输入它们完成某些任务所需要的信息的弹出式窗口。对话框中可以有各种控件：文本框、按钮、图片等等，对话框是窗口的特殊形式。本实验将介绍 Qt 中对话框的操作。

实验代码：

colordialog.h:

```
#include <qpushbutton.h>
#include <qdialog.h>

class ColorDialog: public QDialog
{
    Q_OBJECT
public:
    ColorDialog( QWidget *parent=0, const char *name=0, bool
isModal=TRUE );
    QColor color();
private slots:
    void chooseColor();
private:
    QColor col;
    QPushButton *whiteButton;
    QPushButton *redButton;
    QPushButton *greenButton;
    QPushButton *blueButton;
    QPushButton *cyanButton;
    QPushButton *magentaButton;
    QPushButton *yellowButton;
    QPushButton *grayButton;

    QPushButton *blackButton;
    QPushButton *dredButton;
    QPushButton *dgreenButton;
    QPushButton *dblueButton;
    QPushButton *dcyanButton;
    QPushButton *dmagentaButton;
    QPushButton *dyellowButton;
    QPushButton *dgrayButton;

    QPushButton *lgrayButton;
};
```

```
#endif
```

colordialog.cpp:

```
/****************************************************************************
*******
** $Id: /sample/7/colordialog.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/


#include "colordialog.h"

ColorDialog::ColorDialog( QWidget *parent, const char *name,
                                        bool isModal)
        :QDialog( parent, name, isModal )
{
    setCaption( "Qt Colors" );

    int width = 90;
    int height = 30;
    int x = 1;
    int y = 1;

    col = white;

    whiteButton = new QPushButton( "white", this );
    whiteButton->setFlat( TRUE );
    whiteButton->setBackgroundColor( white );

    redButton = new QPushButton( "red", this );
    redButton->setFlat( TRUE );
    redButton->setBackgroundColor( red );

    greenButton = new QPushButton( "green", this );
    greenButton->setFlat( TRUE );
    greenButton->setBackgroundColor( green );

    blueButton = new QPushButton( "blue", this );
    blueButton->setFlat( TRUE );
```

```
blueButton->setBackgroundColor( blue );

cyanButton = new QPushButton( "cyan", this );
cyanButton->setFlat( "TRUE" );
cyanButton->setBackgroundColor( cyan );

magentaButton = new QPushButton( "magenta", this );
magentaButton->setFlat( TRUE );
magentaButton->setBackgroundColor( magenta );

yellowButton = new QPushButton( "yellow", this );
yellowButton->setFlat( TRUE );
yellowButton->setBackgroundColor( yellow );

grayButton = new QPushButton( "gray", this );
grayButton->setFlat( TRUE );
grayButton->setBackgroundColor( gray );

blackButton = new QPushButton( "black", this );
blackButton->setFlat( TRUE );
blackButton->setBackgroundColor( black );

dredButton = new QPushButton( "darkRed", this );
dredButton->setFlat( TRUE );
dredButton->setBackgroundColor( darkRed );

dgreenButton = new QPushButton( "darkGreen", this );
dgreenButton->setFlat( TRUE );
dgreenButton->setBackgroundColor( darkGreen );

dblueButton = new QPushButton( "darkBlue", this );
dblueButton->setFlat( TRUE );
dblueButton->setBackgroundColor( darkBlue );

dcyanButton = new QPushButton( "darkCyan", this );
dcyanButton->setFlat( "TRUE" );
dcyanButton->setBackgroundColor( darkCyan );

dmagentaButton = new QPushButton( "darkMagenta", this );
dmagentaButton->setFlat( TRUE );
dmagentaButton->setBackgroundColor( darkMagenta );

dyellowButton = new QPushButton( "darkYellow", this );
dyellowButton->setFlat( TRUE );
```

```
        dyellowButton->setBackgroundColor( darkYellow );

    dgrayButton = new QPushButton( "darkGray", this );
        dgrayButton->setFlat( TRUE );
        dgrayButton->setBackgroundColor( darkGray );

        lgrayButton = new QPushButton( "lightGray", this );
        lgrayButton->setFlat( TRUE );
        lgrayButton->setBackgroundColor( lightGray );

        whiteButton->setGeometry( x, y, width, height );
        blackButton->setGeometry( ( x+width ), y, width, height);
        redButton->setGeometry( x, ( y+height ), width, height );
        dredButton->setGeometry( ( x+width ), ( y+height ), width, height );
        greenButton->setGeometry( x, ( y+height*2 ), width, height );
        dgreenButton->setGeometry( ( x+width ), ( y+height*2 ),
    width, height );
        blueButton->setGeometry( x, ( y+height*3 ), width, height );
        dblueButton->setGeometry( ( x+width ), ( y+height*3 ),
    width, height );
        cyanButton->setGeometry( x, ( y+height*4 ), width, height );
        dcyanButton->setGeometry( (x+width), ( y+height*4 ),
    width, height );
        magentaButton->setGeometry( x, ( y+height*5 ), width, height );
        dmagentaButton->setGeometry( ( x+width ), ( y+height*5 ),
    width, height );
        yellowButton->setGeometry( x, ( y+height*6 ), width, height );
        dyellowButton->setGeometry( ( x+width ), ( y+height*6 ),
    width, height );
        grayButton->setGeometry( x, ( y+height*7 ), width, height );
        dgrayButton->setGeometry( ( x+width ), ( y+height*7 ),
    width, height );
        lgrayButton->setGeometry( x, ( y+height*8 ), width*2, height );

        connect( whiteButton,       SIGNAL( pressed() ),
    SLOT( chooseColor() ) );
        connect( blackButton,       SIGNAL( pressed() ),
SLOT( chooseColor() ) );
        connect( redButton,         SIGNAL( pressed() ),
    SLOT( chooseColor() ) );
    connect( dredButton,        SIGNAL( pressed() ),
     SLOT( chooseColor() ) );
        connect( greenButton,       SIGNAL( pressed() ),
    SLOT( chooseColor() ) );
```

```
    connect( dgreenButton,      SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( blueButton,        SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( dblueButton,       SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( cyanButton,        SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( dcyanButton,       SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( magentaButton,     SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( dmagentaButton, SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( yellowButton,      SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( dyellowButton,     SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( grayButton,        SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( dgrayButton,       SIGNAL( pressed() ),
SLOT( chooseColor() ) );
    connect( lgrayButton,       SIGNAL( pressed() ),
SLOT( chooseColor() ) );
}


void ColorDialog::chooseColor()
{
    if ( whiteButton->isDown() ) {
        col = white;
    } else if ( blackButton->isDown() ) {
        col = black;
    } else if ( redButton->isDown() ) {
        col = red;
    } else if ( dredButton->isDown() ) {
        col = darkRed;
    } else if ( greenButton->isDown() ) {
        col = green;
    } else if ( dgreenButton->isDown() ) {
        col = darkGreen;
    } else if ( blueButton->isDown() ) {
        col = blue;
    } else if ( dblueButton->isDown() ) {
```

```
            col = darkBlue;
        } else if ( cyanButton->isDown() ) {
            col = cyan;
        } else if ( dcyanButton->isDown() ) {
            col = darkCyan;
        } else if ( magentaButton->isDown() ) {
            col = magenta;
        } else if ( dmagentaButton->isDown() ) {
            col = darkMagenta;
        } else if ( yellowButton->isDown() ) {
            col = yellow;
        } else if ( dyellowButton->isDown() ) {
            col = darkYellow;
        } else if ( grayButton->isDown() ) {
            col = gray;
        } else if ( dgrayButton->isDown() ) {
            col = darkGray;
        } else if ( lgrayButton->isDown() ) {
            col = lightGray;
        }
        this->close();
    }


    QColor ColorDialog::color()
    {
        return col;
    }
```

mainwindow.h:

```
/****************************************************************************
*******
    ** $Id: /sample/6/main.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    **
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
******/

    #ifndef MAINWINDOW_H
    #define MAINWINDOW_H
```

```
#include <qpopupmenu.h>
#include <qmainwindow.h>
#include <qmenubar.h>
#include "colordialog.h"

class MainWindow: public QMainWindow
{
    Q_OBJECT
public:
    MainWindow( QWidget *parent=0, const char *name=0 );
public slots:
    void chooseBackgroundColor();
};

#endif
```

mainwindow.cpp:

```
/****************************************************************************
*******
** $Id: /sample/7/mainwindow.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#include "mainwindow.h"

MainWindow::MainWindow( QWidget *parent, const char *name )
        :QMainWindow( parent, name )
{
    setCaption( "qt_Example" );
    setBackgroundColor( white );

    QPopupMenu *option = new QPopupMenu;
    option->insertItem( "&Choose background color ",
            this, SLOT( chooseBackgroundColor()));

    QMenuBar *menu = new QMenuBar( this );
    menu->insertItem( "&Option", option );
}
```

```
void MainWindow::chooseBackgroundColor()
{
    ColorDialog *d = new ColorDialog( this, "NULL", TRUE );

    d->exec();

    setBackgroundColor( d->color() );

    delete d;
}
```

main.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/7/main.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #include <qapplication.h>
    #include "mainwindow.h"

    int main( int argc, char **argv )
    {
        QApplication app( argc, argv );
        MainWindow *mainwindow = new MainWindow( 0 );
        app.setMainWidget( mainwindow );
        mainwindow->setGeometry( 5, 5, 400, 320 );
        mainwindow->show();
        int result = app.exec();
        return result;
    }
```

实验原理：

逐步分析一下源代码。

#include <qdialog.h>包含了一个 qdialog.h 的头文件，QDialog 类是对话框窗口的基类。Qt 提供了一些常用的对话框都是由 QDialog 直接派生的。在应用程序中如果进行选择颜色、字体、处理文件、消息等任务都可以使用这些预定义的对话框。Qt 预定义的对话框主要有：

- ✧ QColorDialog
- ✧ QFileDialog
- ✧ QFontDialog
- ✧ QInputDialog
- ✧ QMessageBox
- ✧ QPrintDialog
- ✧ QTabDialog
- ✧ QWizard

使用这些对话框都是直接的。以 QColorDialog 为例，用户只需要调用 QColorDialog 的静态成员函数 getColor()。GetColor()将生成一个颜色选择对话框，并在用户关闭后返回所选作的颜色。

但是在大多数情况下，要使用的对话框都不是标准的，因此必须从 QDialog 类派生自己的对话框。QDialog 类为使用对话框提供额外所需要的成员函数。本实验中的 ColorDialog 正是有 QDialog 类派生出来的，因此在声明该类的时候，声明其共有的继承 QDialog：**class ColorDialog: public QDialog**。

在 ColorDialog 类中，声明了一个共有函数：**QColor color();**该函数将返回给外部调用 ColorDialog 类的用户一个 Qcolor 对象。

```
private slots:
void chooseColor();
```

ColorDialog 为自己定义了一个私有槽。

在私有数据成员定义的时候，ColorDialog 定义了一个 QColor 类的私有成员，用来存放所选择的颜色。还定义了 17 个 QPushButton，每一个 QPushButton 分别代表着 QColor 中枚举的颜色，用来选择不同的颜色。

因此 ColorDialog 其实定义的其实是一个颜色选择的对话框，用户通过点击不同的按钮来选择不同的颜色，而 ColorDialog 将返回其一个所选择的 QColor 对象。

在 colordialog.cpp 中：

```
ColorDialog::ColorDialog( QWidget *parent, const char *name,
                          bool isModal)
    :QDialog( parent, name, isModal )
```

ColorDialog 构造函数直接将参数传递给父类 QDialog。在构造函数里面：

```
whiteButton = new QPushButton( "white", this );
whiteButton->setFlat( TRUE );
whiteButton->setBackgroundColor( white );
……
```

将按钮背景颜色设置成与名字相对应的颜色。并且设置 QPushButton 按钮的外形为平坦的。

```
whiteButton->setGeometry( x, y, width, height );
……
```

这一系列的语句对各个按钮进行了排列。

```
connect( whiteButton,       SIGNAL( pressed() ), SLOT( chooseColor() ) );
……
```

这一系列的语句将各个按钮的 pressed()信号与私有槽 chooseColor()连接在一起。只要有键被按下，私有槽 chooseColor()就会得到相应。在私有槽 chooseColor()里面，开始查询那个键被按下去了。从而设置相应的 QColor 值。

this->close();

调用 close()函数，关闭这个部件，即关闭当前窗口。

Color()函数里面：

return col;

只是简单把值返回给调用者。

在本实验中，我们还构造了另外一个类：MainWindow。MainWindow 类的主要作用是充当本实验的主窗口。该类继承了 QMainWindow 类，因此是一个标准的应用程序的窗口类。

在 mainwindow.h 中，定义了一个共有槽 chooseBackgroundColor()。

在 mainwindow.cpp 中，构造函数把参数直接传递给父类 QMainWindow。

```
setCaption( "qt_Example" );
setBackgroundColor( white );

QPopupMenu *option = new QPopupMenu;
option->insertItem( "&Choose background color ",
        this, SLOT( chooseBackgroundColor()));

QMenuBar *menu = new QMenuBar( this );
menu->insertItem( "&Option", option );
```

构造函数设置窗口的标题、窗口的背景色。

在窗口中添加了一个菜单栏，并且给菜单栏添加了一个下拉菜单。下拉菜单“ Option ”的操作为 chooseBackgroundColor()。

在 chooseBackgroundColor()函数中：

```
ColorDialog *d = new ColorDialog( this, "NULL", TRUE );
d->exec();
setBackgroundColor( d->color() );
delete d;
```

首先定义了一个 ColorDialog 类型的对话框。

d->exec()为执行这个对话框。

在构造对话框的过程中，对话框通过构造函数中的参数 isModal 来区分模式和非模式对话框。

模式对话框就是阻塞同一应用程序中其它可视窗口的输入的对话框：用户必须完成这个对话框中的交互操作并且关闭了它之后才能访问应用程序中的其它任何窗口。模式对话框有它们自己的本地事件循环。用来让用户选择一个文件或者用来设置应用程序参数的对话框通常是模式的。调用 exec()来显示模式对话框。当用户关闭这个对话框，exec()将提供一个可用的返回值并且这时流程控制继续从调用 exec()的地方进行。通常，我们连接默认按钮，例如“ OK ”到 accept()槽并且把“ Cancel ”连接到 reject()槽，来使对话框关闭并且返回适当的值。另外我们也可以连接 done()槽，传递给它 Accepted 或 Rejected。

非模式对话框是和同一个程序中其它窗口操作无关的对话框。在字处理软件中查找和替换对话框通常是非模式的来允许同时与应用程序主窗口和对话框进行交互。调用 show()来显示非模式对话框。show()立即返回，这样调用代码中的控制流将会继续。在实践中你将会经常调用 show()并且在调用 show()的函数最后，控制返回主事件循环。

下面给出模式对话框和非模式对话框的例子：

模式对话框：

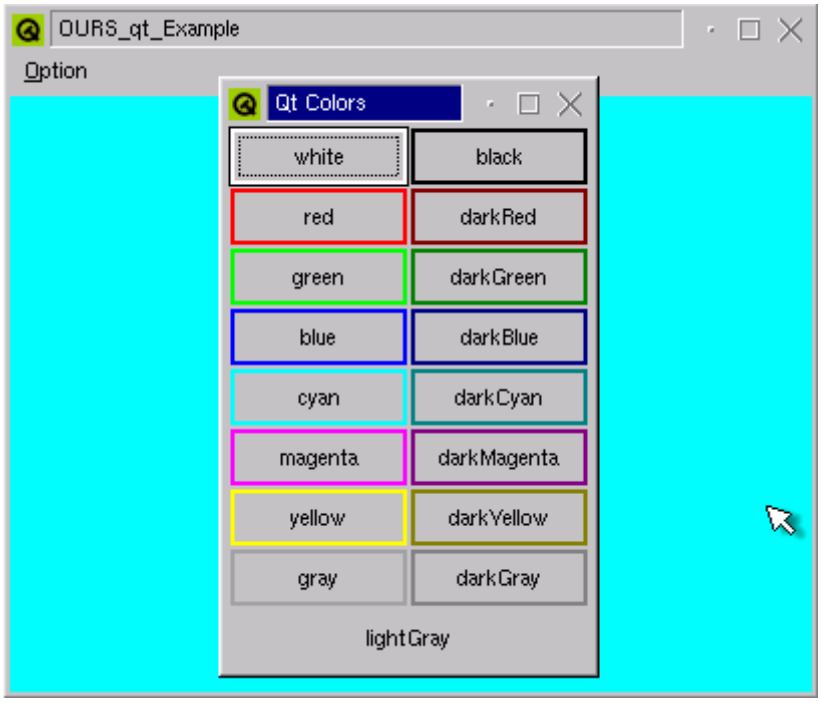
```
QFileDialog *dlg = new QFileDialog( workingDirectory,
QString::null, 0, 0, TRUE );
dlg->setCaption( QFileDialog::tr( "Open" ) );
dlg->setMode( QFileDialog::ExistingFile );
QString result;
if ( dlg->exec() == QDialog::Accepted )
{
    result = dlg->selectedFile();
    workingDirectory = dlg->url();
}
delete dlg;
return result;
```

非模式对话框。在 show()调用之后，控制返回到主事件循环中。

```
int main( int argc, char **argv )
{
  QApplication a( argc, argv );
  int scale = 10;
  LifeDialog *life = new LifeDialog( scale );
  a.setMainWidget( life );
  life->setCaption("Qt Example - Life");
  life->show();
  return a.exec();
}
```

除此之外，对话框还有第三种：半模式对话框。关于半模式对话框，请参照 QProgressDialog。因此在我们的程序中，构造的是一个模式的对话框（isModal=TRUE）。因此使用 exec()来运行。在运行完这个对话框之后，通过获取返回的 QColor，来设置相应的本地窗口的背景色。

最后，**delete d** 用来销毁这个对话框。在 main.cpp 中，我们显示了这个窗口。运行的时候，用户可以通过对话框提供的公共接口来获取参数 QColor 并来设置背景色。编译后在 Qvfb 上运行的结果如图所示：



实验 7 对话框

实验小结：

本实验完成了用户自定义对话框的实现。在 Windows 编程中，使用对话框可以从资源文件创建，而在 Qt 中必须写出每行代码。Windows 下的可视化工具把“画”出的对话框保存为“资源文件”,而在 Qt 中，可视化工具则把它存为 C＋＋源文件。在一点我们在后面学习 Qt Designer 工具时可以看到。有了对话框，用户可以更灵活的设计自己的程序，增加界面的交互性。

实验八 Qt 中的绘图

实验目的：

在前面的几节实验中，并没有涉及到与画图相关的内容，那是因为 Qt 已经为我们做好了。在本此实验中，我们将回到图形上来，考察一下基本的画图方法。在一个窗口画图可以有不同的方法，最简单的方法是直接在窗口中放入一幅位图，这里不讨论这种方式。另外一种是使用的基本的 API 函数进行画线、画点操作。：

实验代码：

drawdemo.h:

```
/****************************************************************************
*******
    ** $Id: /sample/8/drawdemo.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #ifndef DRAWDEMO_H
    #define DRAWDEMO_H


    #include <qwidget.h>
    #include <qcolor.h>
    #include <qpainter.h>
    #include <qtimer.h>
    #include <qframe.h>
    #include <math.h>


    class DrawDemo: public QWidget
    {
    Q_OBJECT
    public:
        DrawDemo( QWidget *parent=0, const char *name=0 );
    protected:
        virtual void paintEvent( QPaintEvent * );
    private slots:
        void flushBuff();
    private:
        int buffer[200];
```

```
        QTimer *timer;
        QFrame *frame;
    };

    #endif
```

drawdemo.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/8/drawdemo.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #define PI 3.1415926

    #include <stdio.h>
    #include "drawdemo.h"

    DrawDemo::DrawDemo( QWidget *parent, const char *name )
            :QWidget( parent, name )
    {
        setCaption( "qt_Example" );

        frame = new QFrame( this, "frame" );
        frame->setBackgroundColor( black );
        frame->setGeometry( QRect( 40, 40, 402, 252 ) );

        for( int i=0; i<200; i++ ) {
            buffer[i] = ( int )( sin( (i*PI) /100 ) * 100 );
        }

        QTimer *timer = new QTimer( this, "timer" );
        connect( timer, SIGNAL( timeout() ), this, SLOT( flushBuff() ) );
        timer->start( 30 );
    }

    void DrawDemo::flushBuff()
    {
        int tmp = buffer[0];
```

```
        int i;
        for( i=0; i<200; i++ ) {
            buffer[i] = buffer[i+1];
        }
        buffer[199] = tmp;
        repaint( 0, 0, 480, 320, TRUE );
    }

    void DrawDemo::paintEvent( QPaintEvent * )
    {
        frame->erase( 0, 0, 400, 320 );
        QPainter painter( frame );
        QPoint beginPoint;
        QPoint endPoint;
        painter.setPen( blue );
        for( int i=0; i<199; i++ ) {
            beginPoint.setX( 2*i );
            beginPoint.setY( buffer[i] +125 );
            endPoint.setX( 2*i+1 );
            endPoint.setY( buffer[i+1] +125 );
            painter.drawLine( beginPoint, endPoint );
        }
    }
```

main.cpp:

```
/****************************************************************************
*******
** $Id: /sample/8/main.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#include <qapplication.h>
#include "drawdemo.h"

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    DrawDemo *drawdemo = new DrawDemo( 0 );
```

```
    drawdemo->setGeometry(10, 20, 480, 320 );
    app.setMainWidget( drawdemo );
    drawdemo->show();
    int result = app.exec();
    return result;
}
```

实验原理：

一行一行的解释，在 drawdemo.h 中，在 protected 成员中：**virtual void paintEvent(QPaintEvent *)**，在前面介绍 QEvent 的时候，我们曾经提及到 QPaintEvent。QPaintEvent 类包含了从画图事件传过来的事件参数。

private slots:

void flushBuff();

定义了一个私有槽，用于刷新缓存区。

int buffer[200]

定义了一个缓存区，大小为 200 字节，用于存储画图数据作为显存。

QTimer *timer;

QTimer 类是 Qt 中关于定时器的一个类，它提供了定时器信号和单触发定时器。

它在内部使用定时器事件来提供更通用的定时器（关于定时器事件，请参照：QTimerEvent）。QTimer 很容易使用：创建一个 QTimer，使用 start()来开始并且把它的 timeout()连接到适当的槽。当这段时间过去了，它将会发射 timeout()信号。

注意当 QTimer 的父对象被销毁时，它也会被自动销毁。实例：

```
QTimer *timer = new QTimer( myObject );
connect( timer, SIGNAL(timeout()), myObject,
    SLOT(timerDone()) );
timer->start( 2000, TRUE ); // 2 秒单触发定时器
```

你也可以使用静态的 singleShot()函数来创建单触发定时器，其详细定义为：

```
void QTimer::singleShot ( int msec, QObject * receiver, const char * member )
```

receiver 是正在接收的对象并且 *member* 是一个槽。时间间隔是 msec。这个静态函数在一个给定时间间隔之后调用一个槽。使用这个函数是非常方便的，因为你不需要被 timerEvent 或创建一个本地 QTimer 对象所困扰。实例：

```
#include <qapplication.h>
#include <qtimer.h>

int main( int argc, char **argv )
{
  QApplication a( argc, argv );
  QTimer::singleShot( 10*60*1000, &a, SLOT(quit()) );
    ... //  创建并且显示你的窗口部件
      return a.exec();
}
```

这个示例程序会自动在 10 分钟之后终止（也就是 600000 毫秒）。作为一个特殊情况，一旦窗口系统事件队列中的所有事件都已经被处理完，一个定时为 0 的 QTimer 就会到时间了。

这也可以用来当提供迅速的用户界面时来做比较繁重的工作。

```
QTimer *t = new QTimer( myObject );
connect( t, SIGNAL(timeout()), SLOT(processOneThing()) );
t->start( 0, FALSE );
```

myObject->processOneThing()将会被重复调用并且应该很快返回（通常在处理一个数据项之后），这样 Qt 可以把事件传送给窗口部件并且一旦它完成这个工作就停止这个定时器。这是在图形用户界面应用程序中实现繁重的工作的一个典型方法，现在多线程可以在越来越多的平台上使用，并且我们希望无效事件最终被线程替代。

注意 QTimer 的精确度依赖于底下的操作系统和硬件。绝大多数平台支持 20 毫秒的精确度，一些平台可以提供更高的。如果 Qt 不能传送定时器触发所要求的数量，它将会默默地抛弃一些。

另一个使用 QTimer 的方法是为你的对象调用 QObject::startTimer()和在你的类中（当然必须继承 QObject）重新实现 QObject::timerEvent()事件处理器。缺点是 timerEvent()不支持像单触发定时器或信号那样的高级水平。

一些操作系统限制可能用到的定时器的数量，Qt 会尽力在限制范围内工作。

QFrame *frame;

QFrame 类是有框架的窗口部件的基类。

它绘制框架并且调用一个虚函数 drawContents()来填充这个框架。这个函数是被子类重新实现的。这里至少还有两个有用的函数：drawFrame()和 frameChanged()。

QPopupMenu 使用这个来把菜单“升高”，高于周围屏幕。QProgressBar 有“凹陷”的外观。QLabel 有平坦的外观。这些有框架的窗口部件可以被改变。

```
QLabel label(...);
label.setFrameStyle( QFrame::Panel | QFrame::Raised );
label.setLineWidth( 2 );

QProgressBar pbar(...);
label.setFrameStyle( QFrame::NoFrame );
```

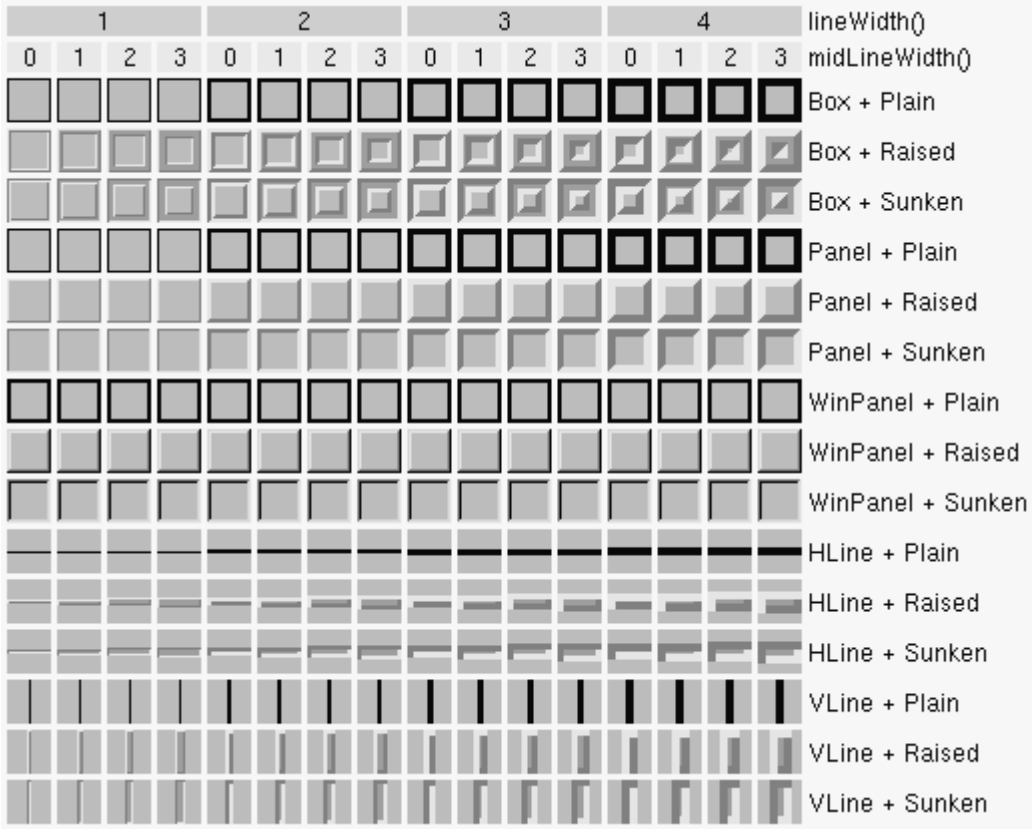
QFrame 类也能够直接被用来创建没有任何内容的简单框架，尽管通常情况下，你要用到 QHBox 或 QVBox，因为它们可以自动的布置你放到框架中的窗口部件。

框架窗口部件有四个属性：frameStyle()、lineWidth()、midLineWidth()和 margin()。

框架风格由框架外形和阴影风格决定。框架外形有 NoFrame、Box、Panel、StyledPanel、PopupPanel、WinPanel、ToolBarPanel、MenuBarPanel、HLine 和 VLine，阴影风格有 Plain、Raised 和 Sunken。

线宽就是框架边界的宽度。中间线宽指定的是在框架中间的另外一条线的宽度，它使用第三种颜色来得到一个三维的效果。注意中间线只有在 Box、HLine 和 VLine 这些凸起和凹陷的框架中才被绘制。边白就是框架和框架内容之间的间隙。

这个表显示的是风格和宽度的绝大多数有用的组合（和一些没有用处的）：



在 drawdemo.cpp 中：**#define PI 3.1415926**，宏定义了 π 的值。

在构造函数中，首先设置了窗口的标题 **setCaption("qt_Example")**。

```
frame = new QFrame( this, "frame" );
frame->setBackgroundColor( black );
frame->setGeometry( QRect( 40, 40, 402, 252 ) );
```

在这里，对 QFrame 定义的对象 frame 进行了设置。

```
for( int i=0; i<200; i++ ) {
        buffer[i] = ( int )( sin( (i*PI) /100 ) * 100 );
}
```

为长为 200 字节的显存填充数据，所填的数据为一个周期的正弦波，并且峰峰值为 200。用来为下面的显示准备好数据。

```
QTimer *timer = new QTimer( this, "timer" );
connect( timer, SIGNAL( timeout() ), this, SLOT( flushBuff() ) );
timer->start( 30 );
```

定义了一个 QTimer——Qt 中的定时器，并且将 timer 的 timeout 信号与私有槽 flushBuff 连接起来。定时器 timer 的定时间隔为 30ms，即每 30ms 就刷新一下显存中的数据。

```
void DrawDemo::flushBuff()
{
    int tmp = buffer[0];
    int i;
    for( i=0; i<200; i++ ) {
        buffer[i] = buffer[i+1];
    }
```

```
        buffer[199] = tmp;
        repaint( 0, 0, 480, 320, TRUE );
    }
```

flushBuff()函数为刷新显存的函数，每刷行一次，就循环移位一次。由于 flushBuff()函数与定时器连接在一起，因此为定时刷新。

```
repaint( 0, 0, 480, 320, TRUE );
```

重新画（0，0，480，320）区域，repaint()方法用于强制窗口立即重画，它有两种调用方式：

```
void QWidget::repaint( int x, int y, int w, int h, bool erase=TRUE ) [slot]
void QWidget::repaint() [slot]
```

两者区别在于，第一个可以指定重画的区域，而第二个简单地全部重画。这两个函数都是 slot，可以和 signal 相连接。

Repaint()会通过立即调用 paintEvent()来直接重新绘制窗口部。如果 erase 为真，Qt 在 paintEvent()调用之前擦除区域*(*x,y,w,h*)*。

当某个窗口或者部件接收到了“ 重画 ”的消息时，它便使用 paintEvent()函数画图，而我们画图的基本方法就是重载 QWidget 的虚方法 paintEvent()。当然，这个只能针对 QWidget 的派生类。

在我们重载的 paintEvent()函数里面：

```
frame->erase( 0, 0, 400, 320 );
```

在窗口部件中擦除指定区域*(x, y, w, h)*，并不产生绘制事件。因此擦除的为 frame 的（0，0，400，300）区域。

```
QPainter painter( frame );
```

Qt 用 Qpainter 来代表图形设备场景。

绘制工具为做到绝大部分绘制图形用户界面程序的需要提供了高度优化的函数。QPainter 可以绘制从简单的直线到像饼图和弦这样的复杂形状。它也可以绘制排列的文本和像素映射。通常，它在一个“ 自然的 ”坐标系统中绘制，但是它也可以在视和世界转换中做到这些。

绘图工具的典型用法是：

- ✧ 构造一个绘图工具。
- ✧ 设置画笔、画刷等等。
- ✧ 绘制。
- ✧ 销毁这个绘图工具。

绝大多数情况下，所有这些是在一个绘制事件中完成的。（实际上，99%的 QPainter 使用是在 QWidget::paintEvent()的重新实现中，并且绘制工具已经为这种用法高度优化了。）这里是一个非常简单的实例：

```
void SimpleExampleWidget::paintEvent()
{
    QPainter paint( this );
    paint.setPen( Qt::blue );
    paint.drawText( rect(), AlignCenter, "The Text" );
}
```

使用方法很简单并且这里有你可以使用的许多设置：font()是当前设置的字体。如果你设置一个不可用的字体，Qt 会找到一个相近的匹配。实际上，font()返回你使用 setFont() 所设置的东西并且 fontInfo()返回你实际使用的字体（这也许是相同的）。

- ✧ brush()是当前设置的画刷，用来填充例如圆的颜色或者调色板。
- ✧ pen()是当前设置的画笔，用来画线或者边缘的颜色或者点画。
- ✧ backgroundMode() 是 Opaque 或者 Transparent，也就是是不是使用 backgroundColor()。仅仅当 backgroundMode()为 Opaque 并且 pen()是一个点画的时候 backgroundColor()才适用，它描述了在点画中背景像素的颜色。
- ✧ rasterOp()是像素绘制和已经存在的像素是如何相互作用的。
- ✧ brushOrigin()是平铺的画刷的原点，通常是窗口的原点。

viewport()、window()、worldMatrix()和很多其它的构成了绘制工具的坐标转换系统。关于这个的解释请参考坐标系统或者参考下面有关这些函数的非常简要的概述。

- ✧ clipping()是指绘制工具是否裁剪。（绘制设备也裁剪。）如果绘制工具裁剪，它裁剪到 clipRegion()。
- ✧ pos()是当前位置，通过 moveTo()设置并且通过 lineTo()使用。

注意这些设置中的一些会镜像到一些绘制设备的设置中，例如 QWidget::font()。QPainter::begin()（或者 QPainter 的构造函数）从绘制设备中复制这些属性。调用，例如 QWidget::setFont()直到绘制工具开始在它上面绘制才会生效。

把所有的这些设置保存到内部栈中，restore()把它们弹出来。

QPainter 的核心功能是绘制，并且这里有最简单的绘制函数：

- ✧ drawPoint():绘制单一的一个点
- ✧ drawPoints()：绘制一组点
- ✧ drawLine()：绘制一条直线
- ✧ drawRect()：绘制一个矩形
- ✧ drawWinFocusRect()：绘制一个窗口焦点矩形
- ✧ drawRoundRect()：绘制一个原形矩形
- ✧ drawEllipse()：绘制一个椭圆
- ✧ drawArc()：绘制一个弧
- ✧ drawPie()：绘制一个饼图
- ✧ drawChord()：绘制一条弦
- ✧ drawLineSegments()：绘制 n 条分隔线
- ✧ drawPolyline()：绘制由 n 个点组成的多边形
- ✧ drawPolygon()：绘制由 n 个点组成的多边形
- ✧ drawConvexPolygon()：绘制由 n 个点组成的凸多边形
- ✧ drawCubicBezier()：绘制三次贝塞尔曲线

所有这些函数使用整数坐标，它们没有浮点数的版本，因为我们想使绘制尽可能快地进行。

这里有绘制像素映射/图像的函数，名为 drawPixmap()、drawImage()和 drawTiledPixmap()。drawPixmap()和 drawImage()产生同样的结果，除了 drawPixmap()在屏幕上更快一些并且 drawImage()在 QPrinter 和 QPicture 上更快并且有时更好。

使用 drawText()可以完成文本绘制，并且当你需要良好的定位，boundingRect()告诉你哪里是给定的 drawText()命令将要绘制的。

这里有一个 drawPicture()函数，用来使用这个绘制工具绘制整个 QPicture 的内容。drawPicture()是唯一忽视所有绘制工具设置的函数：QPicture 有它自己的设置。

通常，QPainter 在设备自己的坐标系统（通常是像素）上操作，但是 QPainter 也很好地支持坐标转换。关于更通用的概述和简单实例请参考坐标系统。

最常用到的函数是 scale()、rotate()、translate()和 shear()，所有这些在 worldMatrix()上

操作。setWorldMatrix()可以替换或者添加到当前设置的 worldMatrix()。

setViewport()设置 QPainter 操作的矩形。默认是整个设备，这通常就很好了，除了在打印机上。setWindow()设置坐标系统，它是被映射到 viewport()的矩形。在 window()中绘制的东西最终会在 viewport()中。窗口的默认就是和视口一样，并且如果你没有使用转换，它们会被优化，赢得一点点速度。

在所有坐标转换完成之后，QPainter 能够把绘制裁剪到一个任意的矩形或者区域。如果 QPainter 裁剪了，hasClipping()为真并且 clipRegion()返回裁剪区域。你可以使用 setClipRegion()或 setClipRect()来设置它。注意裁剪是很慢的。它是完全依赖系统的，但是单凭经验的方法，你可以假设绘制的速度与裁剪区域中的矩形数量成反比。

在 QPainter 的裁剪之后，绘制设备也可以被裁剪。例如，绝大多数窗口部件按子窗口部件的像素裁剪并且绝大多数打印机按接近纸的边缘裁剪。这些另外的裁剪不会受 clipRegion()或 hasClipping()的返回值影响。

QPainter 也包括一些比较少用到的函数，它们在当它们被需要的时候是非常有用的。

isActive()指出绘制工具是否是激活的。begin()（和最常用的构造函数）使它激活。end()（和析构函数）释放它们。如果绘制工具是激活的，device()返回绘制工具在哪个绘制设备上绘制。

有时让其它什么在一个不平常的 QPaintDevice 上绘制是人们想要的。QPainter 支持一个静态函数来做这些，redirect()。我们建议不要使用它，但是对于一些老手这是完美的。

setTabStops()和 setTabArray()可以改变 tab 在哪里停止，但是它们极少被用到。在介绍完了 QPainter 之后，再介绍一下程序中用到的 QPoint。

QPoint 是 Qt 中定义了平面上的一个点的类，由一个 x 坐标和一个 y 坐标确定。标类型是 QCOORD（一个 32 位整数）。QCOORD 的最小值是 QCOORD_MIN（-2147483648），最大值是 QCOORD_MAX（2147483647）。

坐标可以通过函数 x()和 y()来访问，它们可以由 setX()和 setY()来设置并且由 rx()由 ry()来参考。

```
QPoint beginPoint;
QPoint endPoint;
```

定义了两个点，分别代表着画图的起始点和结束点。

```
painter.setPen( blue );
```

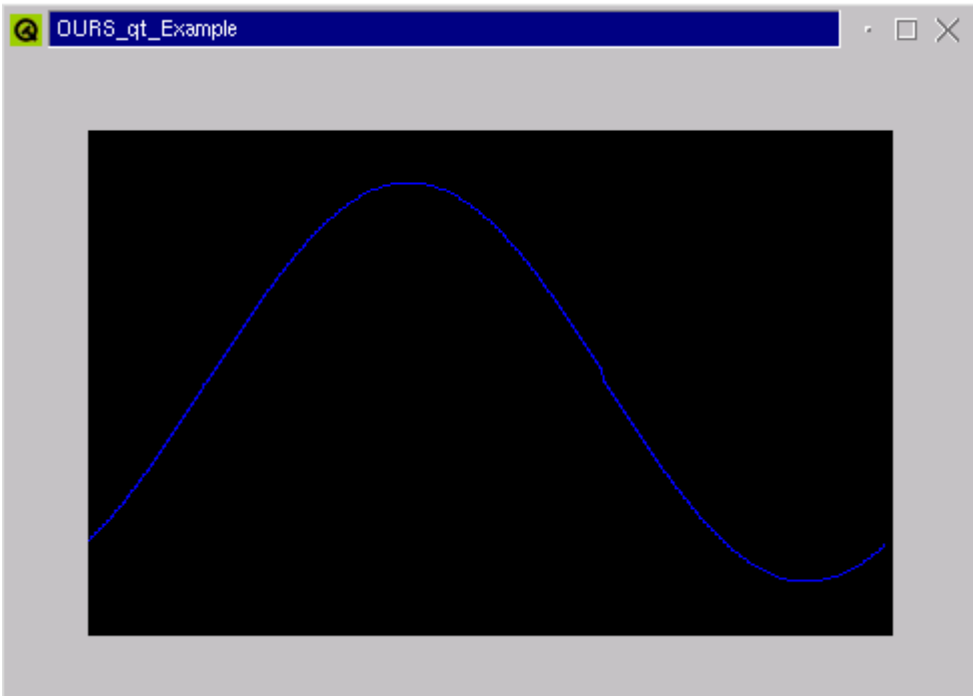
设置画笔的颜色为 blue。

```
for( int i=0; i<199; i++ ) {
        beginPoint.setX( 2*i );
        beginPoint.setY( buffer[i] +125 );
        endPoint.setX( 2*i+1 );
        endPoint.setY( buffer[i+1] +125 );
        painter.drawLine( beginPoint, endPoint );
    }
```

在这个循环里面，画笔用画线的形式将显存的数据画出来，因为显存的数据是正弦波的原始数据，存在着负值，因此加上 125 的偏移量。并且画线的坐标间隔为 2。

在每次程序刷新完显存的数据之后，paintEvent()函数都会被执行一次，即重新画一次正弦波，因此可以在屏幕上看到动态的正弦波。

编译之后在 Qvfb 上运行结果如图所示：



实验 8 Qt 中的绘图

实验小结：

本次实验了解了一下在 Qt 中画图的基本过程和方法。通过 QPainter 方法，它可以提供图形，大约能满足百分之九十的需求。如果需要高度的灵活性，可以使用一个象素接一个象素的方式来准确的获取自己所需要的东西，正如我们实验中画正弦波一样。也可以用能绘图的对象（如 QPen 和 QBrush）。

为了在窗口中画图，必须设置一些基本的图形属性，这就是所谓的图形对象。基本的图形属性有：

- ✧ 笔：QPen
- ✧ 刷子：QBrush
- ✧ 字体：QFont
- ✧ 背景色：setBackgroundColor
- ✧ 背景模式：setBackgroundMode

笔是 Qt 提供的画线工具，刷子是用来填充图形对象。当程序画一个实心圆时，圆的边线时用笔画，圆的内部用刷子画。刷子有颜色和风格属性。另外一种图形对象是字体，当在图形场景中使用文本输出函数时使用的是字体对象。

实验九 Qt 中的多线程编程

实验目的：

本节实验主要考察一下 Qt 对多线程的支持，了解怎样在 Qt 中进行多线程编程以及处理线程的同步。

实验代码：

sinthread.h:

```
/****************************************************************************
*******
** $Id: /sample/9/sinthread.h    2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
****************************************************************************
*******/

#ifndef SINTHREAD_H
#define SINTHREAD_H


#include <qdialog.h>
#include <qwidget.h>
#include <qcolor.h>
#include <qpainter.h>
#include <qtimer.h>
#include <qframe.h>
#include <qapplication.h>
#include <qthread.h>
#include <math.h>


class SinThread: public QWidget, public QThread
{
    Q_OBJECT
public:
    SinThread( QWidget *parent=0, const char *name=0,
QFrame *f=NULL );
    void run();
    void stop();
protected:
    virtual void paintEvent( QPaintEvent * );
```

```
    private slots:
        void flushBuff();
    private:
        int buffer[200];
        QTimer *timer;
        QFrame *frame;
    };

    #endif
```

sinthread.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/8/drawdemo.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005    AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #define PI 3.1415926

    #include <stdio.h>
    #include "sinthread.h"

    SinThread::SinThread( QWidget *parent, const char *name, QFrame *f )
                        :QWidget( parent, name )
    {

        frame = f;
        /*
        frame->setBackgroundColor( black );
        frame->setGeometry( QRect( 40, 40, 402, 252 ) );
        */

        for( int i=0; i<200; i++ ) {
            buffer[i] = ( int )( sin( (i*PI) /100 ) * 100 );
        }

        QTimer *timer = new QTimer( this, "timer" );
        QObject::connect( timer, SIGNAL( timeout() ),
    this, SLOT( flushBuff() ) );
```

```
    timer->start( 500 );
}

void SinThread::flushBuff()
{
    int tmp = buffer[0];
    int i, j=0;

    while( j< 20 ) {
        tmp = buffer[0];
        for( i=0; i<199; i++ ) {
            buffer[i] = buffer[i+1];
        }
        buffer[199] = tmp;
        j++;
    }

    repaint( 0, 0, 480, 320, TRUE );

}

void SinThread::paintEvent( QPaintEvent * )
{
    qApp->lock();
    frame->erase( 0, 0, 400, 320 );
    QPainter painter( frame );
    QPoint beginPoint;
    QPoint endPoint;
    painter.setPen( blue );
    for( int i=0; i<199; i++ ) {
        beginPoint.setX( 2*i );
        beginPoint.setY( buffer[i] +125 );
        endPoint.setX( 2*i+1 );
        endPoint.setY( buffer[i+1] +125 );
        painter.drawLine( beginPoint, endPoint );
    }
    msleep( 50 );
    qApp->unlock();
    msleep( 50 );

}


void SinThread::run()
```

```
{
}

void SinThread::stop()
{
}
```

trithread.h:

```
/****************************************************************************
*******
** $Id: /sample/9/trithread.h    2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
****************************************************************************
*******/

#ifndef TRITHREAD_H
#define TRITHREAD_H


#include <qdialog.h>
#include <qwidget.h>
#include <qcolor.h>
#include <qpainter.h>
#include <qtimer.h>
#include <qframe.h>
#include <qthread.h>
#include <qapplication.h>
#include <math.h>


class TriThread: public QWidget, public QThread
{
    Q_OBJECT
public:
    TriThread( QWidget *parent=0, const char *name=0,
QFrame *f=NULL );
    void run();
    void stop();
protected:
```

```
        virtual void paintEvent( QPaintEvent * );
    private slots:
        void flushBuff();
    private:
        int buffer[200];
        QTimer *timer;
        QFrame *frame;
    };

    #endif
```

tirthread.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/9/trithread.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/

    #define PI 3.1415926

    #include <stdio.h>
    #include "trithread.h"

    TriThread::TriThread( QWidget *parent, const char *name, QFrame *f )
                        :QWidget( parent, name )
    {

        frame = f;
        for( int i=0; i<101; i++ ) {
            buffer[i] = (int) i*2;
        }
        for( int i=101; i<200; i++ ) {
            buffer[i] = (int) (200-i)*2;
        }

        QTimer *timer = new QTimer( this, "timer" );

    QObject::connect( timer, SIGNAL( timeout() ),
    this, SLOT( flushBuff() ) );
```

```
    timer->start( 500 );
}

void TriThread::flushBuff()
{
    int tmp = buffer[0];
    int i , j=0;
    while( j<20 ) {
        tmp = buffer[0];
        for( i=0; i<199; i++ ) {
            buffer[i] = buffer[i+1];
        }
        buffer[199] = tmp;
        j++;
    }
    repaint( 0, 0, 480, 320, TRUE );
    }

void TriThread::paintEvent( QPaintEvent * )
{
    qApp->lock();
    frame->erase( 0, 0, 400, 320 );
    QPainter painter( frame );
    QPoint beginPoint;
    QPoint endPoint;
    painter.setPen( red );
    for( int i=0; i<199; i++ ) {
        beginPoint.setX( 2*i );
        beginPoint.setY( buffer[i] +25 );
        endPoint.setX( 2*i+1 );
        endPoint.setY( buffer[i+1] +25 );
        painter.drawLine( beginPoint, endPoint );
    }
    msleep( 50 );
    qApp->unlock();
    msleep( 50 );
}


void TriThread::run()
{
}

void TriThread::stop()
```

```
    {
    }
```

mainwindow.h:

```
/****************************************************************************
*******
    ** $Id: /sample/9/mainwindow.h      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
*******/


    #ifndef MAINWINDOW_H
    #define MAINWINDOW_H

    #include <qpushbutton.h>
    #include <qframe.h>
    #include "sinthread.h"
    #include "trithread.h"

    class MainWindow: public QWidget
    {
        Q_OBJECT
    public:
        MainWindow( QWidget *parent=0, const char *name=0 );
    private:
        SinThread *sinthread;
        TriThread *trithread;
        QFrame *f;
    };

    #endif
```

mainwindow.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/9/mainwindow.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
```

```
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#include "mainwindow.h"

MainWindow::MainWindow( QWidget *parent, const char *name )
        :QWidget( parent, name )
{
    setCaption( "qt_Example" );

    f = new QFrame( this, "f" );

    f->setBackgroundColor( black );
    f->setGeometry( QRect( 10, 40, 402, 252 ) );
    trithread = new TriThread( this, "trithread", f );
    sinthread = new SinThread( this, "sinthread", f );
}
```

main.cpp:

```
/*****************************************************************************
*******
** $Id: /sample/8/main.cpp      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#include <qapplication.h>
#include <qthread.h>
#include "mainwindow.h"

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    MainWindow *mainwindow =
new MainWindow( 0, "MainWindow" );
    mainwindow->setGeometry( 20, 40, 420, 320 );
    app.setMainWidget( mainwindow );
```

```
    mainwindow->show();
    int result = app.exec();
    return result;
}
```

实验原理：

本实验中，有比较多的文件，这里先说明一下各个文件的作用：

sinthread.h/sinthread.cpp：定义和实现了一个画动态正弦波的线程类

trithread.h/trithread.cpp：定义和实现了一个画动态三角波的线程类

mainwindow.h/mainwindow.cpp：主窗口函数。

main.cpp：主函数。

两个画图的线程，与实验 8 中的画图程序无太大差别，只是对应着不同的画图数据而已，每个类声明中都共有的继承了 Qt 中的线程类：QThread。

QThread 类是最重要的 Qt 线程类，提供了与系统无关的线程。用户可以通过它创建新线程。

QThread 代表在程序中一个单独的线程控制，在多任务操作系统中，它和同一进程中的其它线程共享数据，但运行起来就像一个单独的程序一样。它不是在 main()中开始，QThread 是在 run()中开始运行的。你继承 run()并且在其中包含你的代码。例如：

```
class MyThread : public QThread {
  public:
    virtual void run();
};

void MyThread::run()
{
    for( int count = 0; count < 20; count++ ) {
    sleep( 1 );
    qDebug( "Ping!" );
    }
}

int main()
{
    MyThread a;
    MyThread b;
    a.start();
    b.start();
    a.wait();
    b.wait();
}
```

这将会开始两个线程，每个线程在屏幕上写 20 次“ Ping! ”并且退出。在 main()的结尾调用 wait()是必需的，因为 main()的结束会终结整个程序，它会杀掉所有其它线程。当每个 MyThread 运行到 MyThread::run()结尾时，它就结束运行，就好像一个应用程序离开

main()时所做的一样。

在两个类的实现过程中，基本类似，这里以 TriThread 来进行阐述。

在 TriThread 类的声明中，除了继承了 QWidget 类之外，也继承了 QThread 类。使得画图的同时本身也是一个线程。

```
TriThread( QWidget *parent=0, const char *name=0, QFrame *f=NULL );
```

TriThread 的构造函数除了有两个传给父类 QWidget 的参数之外，自己还定义了一个 QFrame 类型的参数，用于在上面画图。

```
for( int i=0; i<101; i++ ) {
    buffer[i] = (int) i*2;
}
for( int i=101; i<200; i++ ) {
    buffer[i] = (int) (200-i)*2;
}
```

在显存里填充了一个三角波的数据，对应的在 SinThread 类里面应该填充的正弦波数据。在 paintEvent()函数里面，所做的改动只是添加了四条语句：

```
……
qApp->lock();
……
msleep( 50 );
qApp->unlock();
msleep( 50 );
……
```

在 Qt 中，Qmutex 类是用于处理线程同步的类。QMutex 的目的是保护一个对象、数据结构或者代码段，所以同一时间只有一个线程可以访问它。lock()函数是在 QMutex 里面定义过的，如下所示：

```
void QMutex::lock ()
```

试图锁定互斥量。如果另一个线程已经锁定这个互斥量，那么这次调用将阻塞直到那个线程把它解锁。

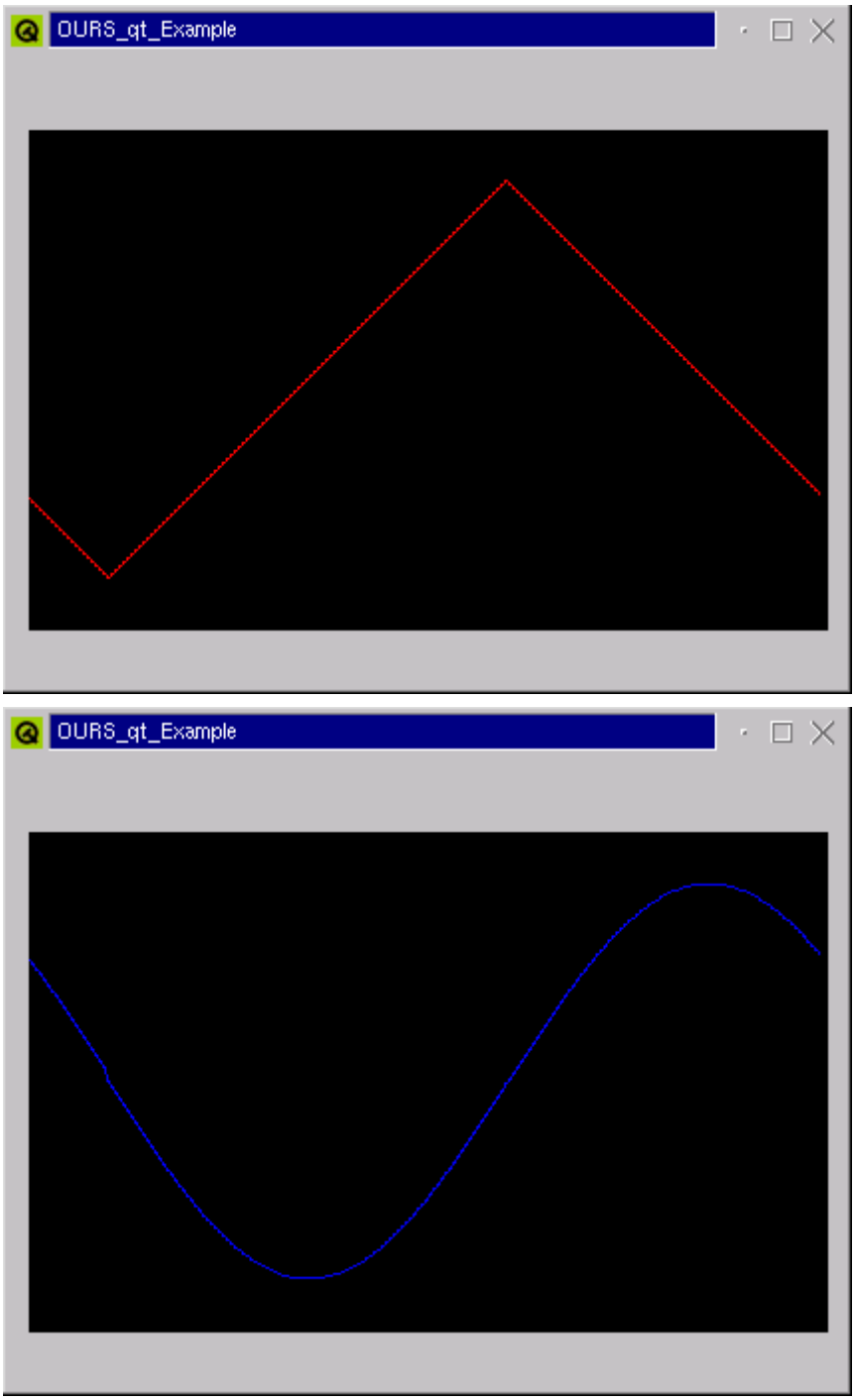
```
void QMutex::unlock ()
```

解锁这个互斥量。试图对不同的线程中锁定的互斥量进行解锁将会返回一个错误。对一个没有锁定的互斥量进行解锁的结果是将导致未定义的行为（不同的操作系统的线程实现是有很大不同的）。

qApp->lock()运行之后，可以确保本只有本类在 frame 上面操作，其它需要对 frame 的操作都会被阻塞，因此不会造成画图时的冲突。

对应着 qApp->unlock()为解锁，解锁之后，其它类就可以访问 frame。msleep()为与系统无关的休眠，这会导致当前线程休眠 n 毫秒。另外还有 sleep()和 usleep()，对应的休眠的事件单位为秒和微秒。

在 mainwindow.cpp 中，定义了两个线程类 trithread 和 sinthread，其中 QFrame 参数相同。这意味着这两个线程都会在 frame 上画图，如果程序中不对进行互斥的操作，将会导致两个线程的冲突。编译之后在 Qvfb 上运行的结果如图所示：



实验 9 Qt 的多线程编程

实验小结：

Qt 对线程提供了支持，基本形式有独立于平台的线程类、线程安全方式的事件传递和一个全局 Qt 库互斥量允许你可以从不同的线程调用 Qt 方法。本实验只是提供了一个自定义的线程的构造方法和简单的调用。读者可以发现，重载的 run()函数和 stop()函数都是空的，读者可以自行添加。

下面是一些使用 Qt 多线程编程时需要注意的问题:

- ✧ 当使用 Qt 库互斥量的时候不要做任何阻塞操作。这将会冻结事件循环。
- ✧ 确认你锁定一个递归 QMutex 的次数和解锁的次数一样，不能多也不能少。
- ✧ 在调用除了 Qt 容器和工具类的任何东西之前锁定 Qt 应用程序互斥量。

- ✧ 谨防隐含地共享类，你应该避免在线程之间使用操作符=()来复制它们。这将会在 Qt 的未来主要的或次要的发行版本中进行改进。
- ✧ 谨防那些没有被设计为线程安全的 Qt 类，例如，QPtrList 的应用程序接口就不是线程安全的并且如果不同的线程需要遍历一个 QPtrList，它们应该在调用 QPtrList::first()之前锁定并且在到达终点之后解锁，而不是在 QPtrList::next()的前后进行锁定和解锁。
- ✧ 确认只在 GUI 线程中创建的继承和使用了 QWidget、QTimer 和 QSocketNotifier 的对象。在一些平台上，在某个不是 GUI 线程的线程中创建这样的对象将永远不会接受到底层窗口系统的事件。
- ✧ 和上面很相似，只在 GUI 线程中使用 QNetwork 类。一个经常被问到的问题是一个 QSocket 是否可以在多线程中使用。这不是必须得，因为所有的 QNetwork 类都是异步的。
- ✧ 不要在不是 GUI 线程的线程中试图调用 processEvents()函数。这也包括 QDialog::exec()、QPopupMenu::exec()、QApplication::processEvents()和其它一些。
- ✧ 在你的应用程序中，不要把普通的 Qt 库和支持线程的 Qt 库混合使用。这也就是说如果你的程序使用了支持线程的 Qt 库，你就不应该连接普通的 Qt 库、动态的载入普通 Qt 库或者动态地连接其它依赖普通 Qt 库的库或者插件。在一些系统上，这样做会导致 Qt 库中使用的静态数据变得不可靠了。

实验十 Qt 中的网络编程

实验目的：

本次实验我们将在 Qt 中利用 Qt 提供的关于网络的类进行网络编程，了解到如何进行网络数据的收发。事实上 Qt 提供的关于网络模块的类可以使用户在编写网络程序的时候更轻松。

实验代码：

在本实验中，有两个完全独立的程序，一个用于客户端 client，一个用于服务器端 server。两个程序单独编译，单独运行。客户端的源代码有：qclient.h、qclient.cpp、main.cpp，服务器端的源代码有：server.cpp。分别如下：

qclient.h:

```
/****************************************************************
***************
** $Id: /sample/10/qclient.h      2.3.2      edited 2004-10-12 $
**
** Copyright (C) 2004-2005 AS.    All rights reserved.
** This file is part of an example program for Qt.    This example
** program may be used, distributed and modified without limitation.
**
*****************************************************************************
*******/

#ifndef QCLIENT_H
#define QCLIENT_H

#include <qsocket.h>
#include <qapplication.h>
#include <qvbox.h>
#include <qhbox.h>
#include <qtextview.h>
#include <qlineedit.h>
#include <qlabel.h>
#include <qpushbutton.h>
#include <qtextstream.h>

class QClient : public QWidget
{
    Q_OBJECT

public:
    QClient(QWidget *parent = 0, const char *name = 0);
private slots:
    void closeConnection();
```

```
        void sendToServer();
        void connectToServer();
        void socketReadyRead();
        void socketConnected();
        void socketConnectionClosed();
        void socketClosed();
        void socketError(int);
    private:
        QSocket     *socket;
        QTextView *infoText;
        QLineEdit *addrText;
        QLineEdit *portText;
        QLineEdit *inputText;
    };

    #endif          //QCLIENT_H
```

qclient.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/10/qclient.h      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
*****************************************************************************
*******/

    #include "qclient.h"

    #include <qsocket.h>
    #include <qapplication.h>
    #include <qvbox.h>
    #include <qhbox.h>
    #include <qtextview.h>
    #include <qlineedit.h>
    #include <qlabel.h>
    #include <qlayout.h>
    #include <qpushbutton.h>
    #include <qtextstream.h>
    #include <qpoint.h>

    QClient::QClient(QWidget *parent, const char *name)
        : QWidget(parent, name)
```

```
{
    infoText    = new QTextView(this);
    QHBox *hb = new QHBox(this);
    inputText = new QLineEdit(hb);

    QHBox *addrBox = new QHBox(this);
    QLabel *ip         = new QLabel("IP:", addrBox, "ip");
    ip->setAlignment(1);
    addrText           = new QLineEdit(addrBox);
    QLabel *port     = new QLabel("PORT:", addrBox, "port");
    port->setAlignment(1);
    portText           = new QLineEdit(addrBox);

    QHBox *buttonBox = new QHBox(this);


    QPushButton *send       = new QPushButton(tr("Send"), hb);
    QPushButton *close      = new QPushButton(tr("Close connection"),
                  buttonBox);
    QPushButton *quit       = new QPushButton(tr("Quit"), buttonBox);
    QPushButton *Connect = new QPushButton(tr("Connect"),
addrBox);

    connect(send, SIGNAL(clicked()),
            SLOT(sendToServer()) );
    connect(close, SIGNAL(clicked()),
            SLOT(closeConnection()) );
    connect(quit, SIGNAL(clicked()),
            qApp, SLOT(quit()) );
    connect(Connect, SIGNAL(clicked()),
            SLOT(connectToServer()) );

    socket = new QSocket(this);
    connect(socket, SIGNAL(connected()),
            SLOT(socketConnected()) );
    connect(socket, SIGNAL(connectionClosed()),
            SLOT(socketConnectionClosed()) );
    connect(socket, SIGNAL(readyRead()),
            SLOT(socketReadyRead()) );
    connect(socket, SIGNAL(error(int)),
            SLOT(socketError(int)) );

    QVBoxLayout *l = new QVBoxLayout(this);
    l->addWidget(infoText, 10);
```

```
    l->addWidget(hb, 1);
    l->addWidget(addrBox, 1);
    l->addWidget(buttonBox, 1);

    //connect to the server
    infoText->append(tr("Tying to connect to the server"));

}


void QClient::closeConnection()
{
    socket->close();
    if (QSocket::Closing == socket->state()) {
        // We have a delayed close
        connect(socket, SIGNAL(delayedCloseFinished()),
                SLOT(socketClosed()));
    } else {
        // The socket is closed
        socketClosed();
    }
}

void QClient::sendToServer()
{
    // write to the server
    if (QSocket::Connection == socket->state()) {
        QTextStream os(socket);
        os << inputText->text() << "\n";
        inputText->setText("");
    } else {
        // The socket is unconnected
        infoText->append(tr("The server is lost\n"));
    }

}

void QClient::connectToServer()
{
    // connect to the server
    socket->connectToHost(addrText->text(), (portText->text()).toInt());
}

void QClient::socketReadyRead()
```

```
{
    // read from the server
    while (socket->canReadLine()) {
        infoText->append(socket->readLine());
    }
}

void QClient::socketConnected()
{
    infoText->append(tr("Connected to server\n"));
}

void QClient::socketConnectionClosed()
{
    infoText->append(tr("Connection closed by the server\n"));
}

void QClient::socketClosed()
{
    infoText->append(tr("Connection closed\n"));
}

void QClient::socketError(int e)
{
    if (e == QSocket::ErrConnectionRefused) {
        infoText->append(tr("Connection Refused\n"));
    } else if (e == QSocket::ErrHostNotFound) {
        infoText->append(tr("Host Not Found\n"));
    } else if (e == QSocket::ErrSocketRead) {
        infoText->append(tr("Socket Read Error\n"));
    }
}
```

main.cpp:

```
/****************************************************************************
*******
    ** $Id: /sample/10/main.cpp      2.3.2      edited 2004-10-12 $
    **
    ** Copyright (C) 2004-2005 AS.    All rights reserved.
    ** This file is part of an example program for Qt.    This example
    ** program may be used, distributed and modified without limitation.
    **
****************************************************************************
```

```
*******/

    #include <qapplication.h>
    #include "qclient.h"

    int main( int argc, char **argv )
    {
        QApplication app( argc, argv );
        QClient *client = new QClient( 0 );
        app.setMainWidget( client );
        client->show();
        int result = app.exec();
        return result;
    }
```

server.cpp:

```
/****************************************************************************
*******
    ** $Id:  qt/server.cpp   3.0.5   edited Oct 12 2001 $
    **
    ** Copyright (C) 1992-2000 Trolltech AS.  All rights reserved.
    **
    ** This file is part of an example program for Qt.  This example
    ** program may be used, distributed and modified without limitation.
    **
*****************************************************************************
*******/

    #include <qsocket.h>
    #include <qserversocket.h>
    #include <qapplication.h>
    #include <qvbox.h>
    #include <qtextview.h>
    #include <qlabel.h>
    #include <qpushbutton.h>
    #include <qtextstream.h>

    #include <stdlib.h>


    /*
      The ClientSocket class provides a socket that is connected with a client.
      For every client that connects to the server, the server creates a new
      instance of this class.
```

```
*/
class ClientSocket : public QSocket
{
    Q_OBJECT
public:
    ClientSocket( int sock, QObject *parent=0, const char *name=0 ) :
        QSocket( parent, name )
    {
        line = 0;
        connect( this, SIGNAL(readyRead()), SLOT(readClient()) );
        connect( this, SIGNAL(connectionClosed()),
SLOT(connectionClosed()) );
        setSocket( sock );
    }

    ~ClientSocket()
    {
    }

private slots:
    void readClient()
    {
        while ( canReadLine() ) {
            QTextStream os( this );
            os << line << ": " << readLine();
            line++;
        }
    }

    void connectionClosed()
    {
        delete this;
    }

private:
    int line;
};


/*
  The SimpleServer class handles new connections to the server. For every
  client that connects, it creates a new ClientSocket -- that instance is now
  responsible for the communication with that client.
*/
```

```
class SimpleServer : public QServerSocket
{
    Q_OBJECT
public:
    SimpleServer( QObject* parent=0 ) :
        QServerSocket( 4242, 1, parent )
    {
        if ( !ok() ) {
            qWarning("Failed to bind to port 4242");
            exit(1);
        }
    }

    ~SimpleServer()
    {
    }

    void newConnection( int socket )
    {
        (void)new ClientSocket( socket, this );
        emit newConnect();
    }

signals:
    void newConnect();
};


/*
  The ServerInfo class provides a small GUI for the server. It also creates the
  SimpleServer and as a result the server.
*/
class ServerInfo : public QVBox
{
    Q_OBJECT
public:
    ServerInfo()
    {
        SimpleServer *server = new SimpleServer( this );

        QString itext = QString(
                "This is a small server example.\n"
                "Connect with the client now."
                );
```

```
            QLabel *lb = new QLabel( itext, this );
            lb->setAlignment( AlignHCenter );
            infoText = new QTextView( this );
            QPushButton *quit = new QPushButton( "Quit" , this );

            connect( server, SIGNAL(newConnect()),
    SLOT(newConnect()) );
            connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
        }

        ~ServerInfo()
        {
        }

    private slots:
        void newConnect()
        {
            infoText->append( "New connection\n" );
        }

    private:
        QTextView *infoText;
    };


    int main( int argc, char** argv )
    {
        QApplication app( argc, argv );
        ServerInfo info;
        app.setMainWidget( &info );
        info.show();
        return app.exec();
    }

    #include "server.moc"
```

实验原理：

在 qclient.h 中，QClient 类继承了 QWidget 类，并且定义了 8 个私有槽，分别如下：

- ✧ closeConnection()：关闭网络连接
- ✧ sendToServer()：发送数据到服务器端
- ✧ connectToServer()：连接服务器
- ✧ socketReadyRead()：有新数据可读
- ✧ socketConnected()：已连接上
- ✧ socketConnectionClosed()：服务器关闭连接关闭
- ✧ socketError(int)：错误

✧ socketClosed()：本地关闭连接

QSocket *socket;

QSocket 类提供了一个有缓冲的 TCP 连接。

它提供一个完全非阻塞的QIODevice，使用套接字特征代码来修改和扩展了 QIODevice 的应用编程接口。你和可能常常调用的 connectToHost()、bytesAvailable()、canReadLine()这些函数并且它们继承了 QIODevice。

connectToHost()是一个最常用的函数。就像它的名字那样，它打开一个被命名的主机的连接。

绝大多数网络协议是基于包的或者基于行的。canReadLine()可以识别一个连接中是否包含一个完全不可读的的行，并且 bytesAvailable()返回可以被读取的字节数量。

信号 error()、connected()、readyRead()和 connectionClosed()通知你连接的进展。还有一些不太常用的信号。当 connectToHost()已经完成它的 DNS 查找并且正在开始它的 TCP 连接时，hostFound()被发射。当 close()成功时，delayedCloseFinished()被发射。当 QSocket 把它的“写”队列中的数据移到 TCP 运行中。

还有几个套接字的访问函数：state()返回这个对象是否空闲，是否正在做 DNS 查找，是否正在连接，还是一个正在操作的连接，等等。address()和 port()返回连接所使用的 IP 地址和端口。peerAddress()和 peerPort()函数返回自身所用到的 IP 地址和端口并且 peerName()返回自身所用的名称（通常是被传送给 connectToHost()的名字）。socket() 返回这个套接字所用到的 QSocketDevice 的指针。

QSocket 继承了 QIODevice 并且重新实现了一些函数。通常你可以把它作为 QIODevice 来写，并且绝大多数情况也可以作为 QIODevice 来读。但匹配的不够完美，因为 QIODevice 应用编程接口是为同一个机器可以控制的设备而设计的，而异步的端对端网络连接和这个不太一样。例如，没有什么可以和 QIODevice::size()确切地匹配。open()、close()、flush()、size()、at()、atEnd()、readBlock()、writeBlock()、getch()、putch()、ungetch()和 readLine()的文档详细地描述了不同点。

QTextView *infoText;

QTextView 提供了一个显示大量文本的类。

QLineEdit *addrText;

QLineEdit *portText;

QLineEdit *inputText;

QLineEdit 部件是一个单行的文本编辑器。

行编辑允许用户通过许多的编辑函数来输入和编辑单行的纯文本，这些函数包括：撤销、重新键入、剪切、复制、拖放等。

通过修改行编辑的 echoMode()，它可以被设置成“只写”的模式，例如输入密码时。

行的长度可以用 maxLength()来限制，并且只可以通过设置 validator()来任意的设置之的合法性。

在程序中，定义了三个 QLineEdit 类对象，分别用来输入服务器地址、服务器端口号和输入文本。

在 qclient.cpp 中，构造函数里面：

```
infoText  = new QTextView(this);
QHBox *hb = new QHBox(this);
inputText = new QLineEdit(hb);
```

QHBox 部件为其子部件提供了水平方向上的几何管理。所有的 QHBox 的子部件都会被互相靠的摆放着，并且会根据它们的 sizeHints()来分配大小。可以使用 setMargin()增加

（子类部件）边缘的空隙，通过 setSpacing()来设置部件之间的空隙。如果用户想要按照比例来摆放不同大小的部件就可以使用 setStretchFactor()。典型的 QHBox 如下图所示：

因此 infotext 将会在 QHBox 所定义的对象 hb 里面水平摆放。

```
QHBox *addrBox = new QHBox(this);
QLabel *ip         = new QLabel("IP:", addrBox, "ip");
ip->setAlignment(1);
addrText           = new QLineEdit(addrBox);
QLabel *port       = new QLabel("PORT:", addrBox, "port");
port->setAlignment(1);
portText           = new QLineEdit(addrBox);
```

分别将 QLabel 对象 ip、port 和 QLineEdit 对象 addrText、portText 放入 addrBox 中，addrBox 将按照声明的先后顺序水平放置它们。

```
QHBox *buttonBox = new QHBox(this);
QPushButton *send       = new QPushButton(tr("Send"), hb);
QPushButton *close      = new QPushButton(tr("Close connection"),
buttonBox);
QPushButton *quit       = new QPushButton(tr("Quit"), buttonBox);
QPushButton *Connect = new QPushButton(tr("Connect"), addrBox);
```

同样的，将定义的一些按钮放入不同的 QHBox 对象中，以得到水平放置的排列。

```
connect(send, SIGNAL(clicked()),
                SLOT(sendToServer()) );
connect(close, SIGNAL(clicked()),
                SLOT(closeConnection()) );
connect(quit, SIGNAL(clicked()),
                qApp, SLOT(quit()) );
connect(Connect, SIGNAL(clicked()),
                SLOT(connectToServer()) );
```

将不同的按钮发出的 clicked()信号与相应的动作槽连接在一起，以便通过按钮来控制一些动作。

```
socket = new QSocket(this);
connect(socket, SIGNAL(connected()),
                SLOT(socketConnected()) );
connect(socket, SIGNAL(connectionClosed()),
                SLOT(socketConnectionClosed()) );
connect(socket, SIGNAL(readyRead()),
                SLOT(socketReadyRead()) );
connect(socket, SIGNAL(error(int)),
                SLOT(socketError(int)) );
```

定义了一个 Qsocket 对象，并且将相应的信号与私有槽相连。

```
QVBoxLayout *l = new QVBoxLayout(this);
l->addWidget(infoText, 10);
l->addWidget(hb, 1);
```

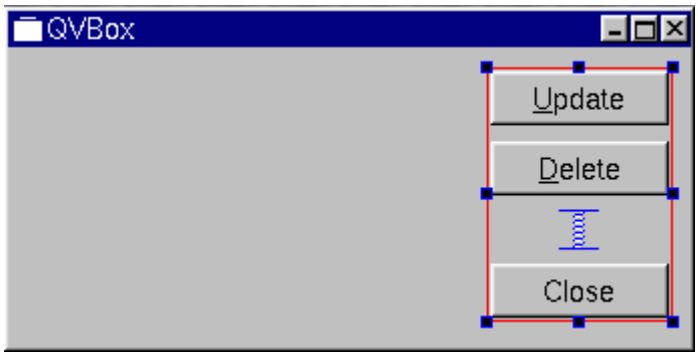
```
l->addWidget(addrBox, 1);
l->addWidget(buttonBox, 1);
```

QVBoxLayout 类使部件垂直的排成一列。

这个类一般用来在垂直方向摆放部件。最简单的使用方式如下：

```
QBoxLayout * l = new QVBoxLayout( widget );
l->addWidget( aWidget );
l->addWidget( anotherWidget );
```

典型的图如下所示：



在 closeConnection()函数中，可以根据 socket->state()函数来获取当前的网络连接状态，判断是否真的关闭连接了，如果使就调用 socketClosed()函数。

在 sendToServer()函数中：

```
QTextStream os(socket);
os << inputText->text() << "\n";
```

QTextStream 类提供了使用 QIODevice 读写文本的基本功能。

文本流类的功能界面和标准的 C++的 iostream 类非常相似。iostream 和 QTextStream 的不同点是我们的流操作在一个很容易被继承的 QIODevice 上，而 iostream 只能操作一个不能被继承的 FILE *指针。

QTextStream 类读写文本，它不适合处理二进制数据（而 QDataStream 是适合的）。通过上面两条语句，inputText 中的文本就被写入到网络中去了。

在 connectToServer()函数中：

```
socket->connectToHost(addrText->text(), (portText->text()).toInt());
```

在 Qt 的库中，connectToHost 是这样定义的：

```
void QSocket::connectToHost ( const QString & host,
        Q_UINT16 port ) [虚]
```

试图连接主机 host 的指定端口 port 并且立即返回。

任何连接或者正在进行的连接被立即关闭，并且 QSocket 进入 HostLookup 状态。当查找成功，它发射 hostFound()，开始一个 TCP 连接并且进入 Connecting 状态。最后，当连接成功时，它发射 connected()并且进入 Connected 状态。如果在任何一个地方出现错误，它发射 error()。

host 可以是一个字符串形式的 IP 地址，也可以是一个 DNS 名称。如果需要 QSocket 将会进行一个普通的 DNS 查找。注意 port 是本地字节顺序，不像其它库那样。

在 socketReadyRead()中：

```
infoText->append(socket->readLine());
```

readLine()返回包含终止新行符（\n）的一行文本。如果 canReadLine()返回假，返回“ ”。

Server.cpp 中，首先定义了一个类 ClientSocket，该类继承了 Qsocket，用于服务器端接

受到一个新连接时所创建的新连接。socket 号由服务器类通过参数传递进来。

```
void readClient()
    {
        while ( canReadLine() ) {
            QTextStream os( this );
            os << line << ": " << readLine();
            line++;
        }
    }
```

ClientSocket 将每次由 clinet 端发送过来的数据加上编号和冒号发送回去。

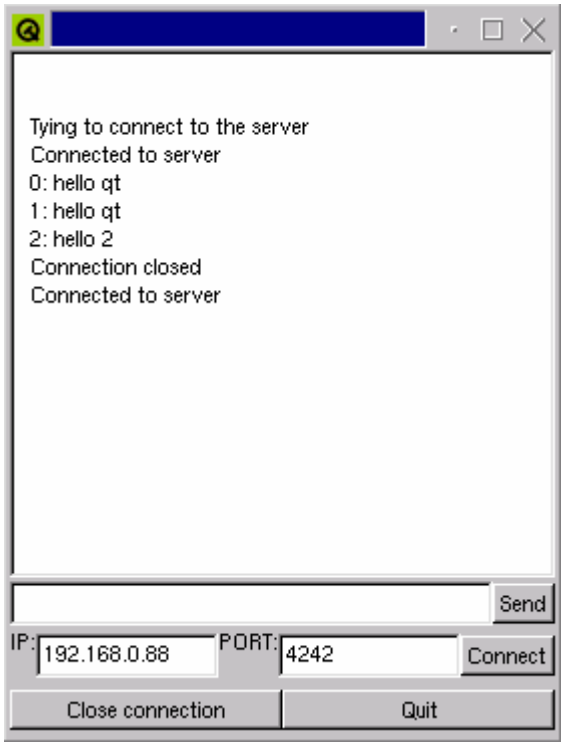
SimpleServer 为一个简单的服务器端的类，该类继承了 QServerSocket 类。

```
void newConnection( int socket )
    {
        (void)new ClientSocket( socket, this );
        emit newConnect();
    }
```

在每次有新连接的时候，QServerSocket 就要构造一个新的 socket，对应着该连接，接收该连接并将数据发回。

ServerInfo 类是一个用来显示服务器端连接信息的类。

SimpleSocket 函数在每次有新连接的时候都会发出 newConnect()信号，SimpleInfo 类接受到后，标示出有新的连接。服务器端和客户端的程序单独编译后，一个在 Qvfb 上运行，一个在 X 上运行，结果如图所示：



实验 10 Qt 中的网络编程

实验小结：

本次实验介绍了如何利用 Qt 中提供的类进行简单的网络编程，并提供了客户端和服务器端的收发实例。这些类使得我们进行 Socket 编程的时候方便轻松一些。用户可以自己与 linux 下 C 的 socket 编程做个比较。其实可以发现，Qt 只不过是提供了一些相对高级的内容。