如何学习 Qt

我们假设你已经熟悉 C++了！
请先阅读一下 Qt 白皮书。它包含一个关于 Qt 软件的概述，并且提供了一些用来示范使用 Qt 进行编程的代码的片断。它会给你一个“大的图画”。
如果你想要完全的在 C++中进行编程，不使用任何设计工具的帮助下在代码中设计你的界面，请阅读教程。教程 1 就是被设计成把你带入 Qt 编程的一个教程，它更强调的是编写代码而不是一个特征的漫游。教程 2 是一个更加真实的例子，示范了如何编写菜单、工具条、文件的载入和保存、对话框等地那个。
如果你想使用一个设计工具来设计你的用户界面，那么你至少要先阅读 Qt 设计器手册的前几章。在这之后，学习一下上面提到的纯粹的 C++教程（教程 1 和教程 2）还是很值得的。
到现在为止，如果你已经完成了一些小的可以工作的应用程序并且对 Qt 编程有了一个主要的了解。你可以直接开始你自己的项目了，但我们建议你阅读一些关键的概述来加深你对 Qt 的理解：对象模型和信号和槽。
在这里我们建议你看一下概述并且阅读一些和你的项目相关的文章。你也许会发现浏览和你项目做相同事情的实例的源代码是非常有用的。你也可以阅读 Qt 的源代码，因为它们也被提供。
如果你运行 `demo` 这个应用程序（在`$QTDIR/examples/demo`），你就会看到很多运转中的 Qt 窗口部件
Qt 提供了广泛的文档，完全前后参考的超文本，所以你可以很容易地按你喜欢的方式进行点击。在文档中，你最经常使用的部分可能就是 API 参考。每一个链接都提供了一个不同的方式来导航 API 参考，全都试试，看哪一个更适合你。
你现在应该已经准备好你的伟大工程：祝你好运，玩得开心！

Qt 教程一 —— 共十四步

这个教程介绍了使用 Qt 工具包进行图形用户界面编程。它没有包括所有的东西：强调的是教授一种图形用户界面编程的编程思想，并且介绍 Qt 的特征也是必需的。一些通常情况下使用的特征在这个教程里没有用到。
第一章开始讲述一个十行的 Hello World 程序并且后来的每一章都介绍了一个或几个更多的概念。一直到第十四章，程序已经从第一章的十行变成了六百五十行的游戏。
如果你对 Qt 完全不熟悉，如果你还没有读过如何学习 Qt 的话，请读一下。
教程章节：

1. Hello, World!
2. 调用退出
3. 家庭价值
4. 使用窗口部件
5. 组装积木
6. 组装丰富的积木！
7. 一个事物领导另一个

8. 准备战斗
9. 你可以使用加农炮了
10. 像丝一样滑
11. 给它一个炮弹
12. 悬在空中的砖
13. 游戏结束
14. 面对墙壁

这个小游戏看起来不像一个现代的图形用户界面应用程序。它只使用了有用的少数图形用户界面技术，但是如果你通过它工作之后，我们建议你阅读一下教程二。第二个教程更加正式一些，并且覆盖了包括菜单条、工具条、文件的载入和保存、对话框等典型应用程序的特征。

Qt 教程一 —— 第一章：Hello, World!

第一个程序是一个简单的 Hello World 例子。它只包含你建立和运行 Qt 应用程序所需要的最少的代码。上面的图片是这个程序的快照。

```
/****************************************************************
**
** Qt 教程一 - 2
**
****************************************************************/

#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();
    return a.exec();
}
```

一行一行地解说

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

这一行包含了 QApplication 类的定义。在每一个使用 Qt 的应用程序中都必须使用一个 QApplication 对象。QApplication 管理了各种各样的应用程序的广泛资源，比如默认的字体和光标。

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

这一行包含了 QPushButton 类的定义。参考文档的文件的最上部分提到了使用哪个类就必须包含哪个头文件的说明。

QPushButton 是一个经典的图形用户界面按钮，用户可以按下去，也可以放开。它管理自己的观感，就像其它每一个 QWidget。一个窗口部件就是一个可以处理用户输入和绘制图形的用户界面对象。程序员可以改变它的全部观感和它的许多主要的属性（比如颜色），还有这个窗口部件的内容。一个 QPushButton 可以显示一段文本或者一个 QPixmap。

```
int main( int argc, char **argv )
{
```

main()函数是程序的入口。几乎在使用 Qt 的所有情况下，main()只需要在把控制转交给 Qt 库之前执行一些初始化，然后 Qt 库通过事件来向程序告知用户的行为。argc 是命令行变量的数量，argv 是命令行变量的数组。这是一个 C/C++特征。它不是 Qt 专有的，无论如何 Qt 需要处理这些变量（请看下面）。

```
    QApplication a( argc, argv );
```

a 是这个程序的 QApplication。它在这里被创建并且处理这些命令行变量（比如在 X 窗口下的-display）。请注意，所有被 Qt 识别的命令行参数都会从 argv 中被移除（并且 argc 也因此而减少）。关于细节请看 QApplication::argv()文档。

注意：在任何 Qt 的窗口系统部件被使用之前创建 QApplication 对象是必须的。

```
    QPushButton hello( "Hello world!", 0 );
```

这里，在 QApplication *之后*，接着的是第一个窗口系统代码：一个按钮被创建了。这个按钮被设置成显示“Hello world!”并且它自己构成了一个窗口（因为在构造函数指定 0 为它的父窗口，在这个父窗口中按钮被定位）。

```
    hello.resize( 100, 30 );
```

这个按酒被设置成 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。

你不用必须设置一个主窗口部件，但绝大多数程序都有一个。

```
    hello.show();
```

当你创建一个窗口部件的时候，它是不可见的。你必须调用 show()来使它变为可见的。

```
    return a.exec();
```

这里就是 main()把控制转交给 Qt，并且当应用程序退出的时候 exec()就会返回。在 exec()中，Qt 接受并处理用户和系统的事件并且把它们传递给适当的窗口部件。

```
}
```

你现在可以试着编译和运行这个程序了。

编译

编译一个 C++应用程序，你需要创建一个 makefile。创建一个 Qt 的 makefile 的最容易的方法是使用 Qt 提供的连编工具 qmake。如果你已经把 main.cpp 保存到它自己的目录了，你所要做的就是这些：

```
qmake -project
qmake
```

第一个命令调用 qmake 来生成一个.pro（项目）文件。第二个命令根据这个项目文件来生成一个（系统相关的）makefile。你现在可以输入 make（或者 nmake，如果你使用 Visual Studio），然后运行你的第一个 Qt 应用程序！

行为

当你运行它的时候，你就会看到一个被单一按钮充满的小窗口，在它上面你可以读到著名的词：Hellow World!

练习

试着改变窗口的大小。按下按钮。如果你在 X 窗口下运行，使用-geometry 选项（比如，-geometry 100x200+10+20）来运行这个程序。

现在你可以进行第二章了。

[下一章] [教程一主页]

Qt 教程一 —— 第二章：调用退出

你已经在第一章中创建了一个窗口，我们现在使这个应用程序在用户让它退出的时候退出。

我们也会使用一个比默认字体更好的一个字体。

```
/*****************************************************************
**
** Qt 教程一 - 2
**
*****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
```

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

    QPushButton quit( "Quit", 0 );
    quit.resize( 75, 30 );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );

    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );

    a.setMainWidget( &quit );
    quit.show();
    return a.exec();
}
```

一行一行地解说

```
    #include <qfont.h>
```

因为这个程序使用了 QFont，所以它需要包含 qfont.h。Qt 的字体提取和 X 中提供的可怕的字体提取大为不同，字体的载入和使用都已经被高度优化了。

```
        QPushButton quit( "Quit", 0 );
```

这时，按钮显示“Quit”，确切的说这就是当用户点击这个按钮时程序所要做的。这不是一个巧合。因为这个按钮是一个顶层窗口，我们还是把 0 作为它的父对象。

```
        quit.resize( 75, 30 );
```

我们给这个按钮选择了另外一个大小，因为这个文本比“Hello world!”小一些。我们也可以使用 QFontMetrics 来设置正确的大小。

```
        quit.setFont( QFont( "Times", 18, QFont::Bold ) );
```

这里我们给这个按钮选择了一个新字体，Times 字体中的 18 点加粗字体。注意在这里我们调用了这个字体。
你也可以改变整个应用程序的默认字体（使用 QApplication::setFont()）。

```
        QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
```

connect 也许是 Qt 中*最重要*的特征了。注意 connect()是 QObject 中的一个静态函数。不要把这个函数和 socket 库中的 connect()搞混了。
这一行在两个 Qt 对象（直接或间接继承 QObject 对象的对象）中建立了一种单向的连接。每一个 Qt 对象都有 signals（发送消息）和 slots（接收消息）。所有窗口部件都是 Qt 对象。它们继承 QWidget，而 QWidget 继承 QObject。
这里 *quit* 的 *clicked()*信号和 *a* 的 *quit()*槽连接起来了，所以当这个按钮被按下的时候，这个程序就退出了。
信号和槽文档详细描述了这一主题。

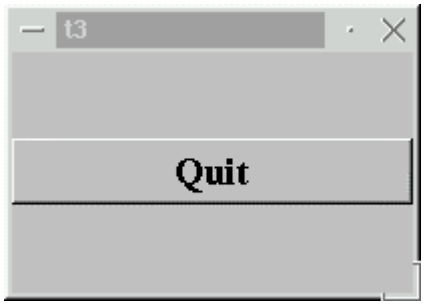
行为

当你运行这个程序的时候，你会看到这个窗口比第一章中的那个小一些，并且被一个更小的按钮充满。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

试着改变窗口的大小。按下按钮。注意！connect()看起来会有一些不同。
是不是在 QPushButton 中还有其它的你可以连接到 quit 的信号？提示：QPushButton 继承了 QButton 的绝大多数行为。
现在你可以进行第三章了。

Qt 教程一 —— 第三章：家庭价值



这个例子演示了如何创建一个父窗口部件和子窗口部件。
我们将会保持这个程序的简单性，并且只使用一个单一的父窗口部件和一个独立的子窗口部件。

```
/*****************************************************************
**
** Qt 教程一 - 3
**
*****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
#include <qvbox.h>

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

    QVBox box;
    box.resize( 200, 120 );

    QPushButton quit( "Quit", &box );
```

```
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );

    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );

    a.setMainWidget( &box );
    box.show();

    return a.exec();
}
```

一行一行地解说

```
    #include <qvbox.h>
```

我们添加了一个头文件 qvbox.h 用来获得我们要使用的布局类。

```
        QVBox box;
```

这里我们简单地创建了一个垂直的盒子容器。QVBox 把它的子窗口部件排成一个垂直的行，一个在其它的上面，根据每一个子窗口部件的 QWidget::sizePolicy() 来安排空间。

```
        box.resize( 200, 120 );
```

我们它的高设置为 120 像素，宽为 200 像素。

```
        QPushButton quit( "Quit", &box );
```

子窗口部件产生了。

QPushButton 通过一个文本（“text”）和一个父窗口部件（box）生成的。子窗口部件总是放在它的父窗口部件的最顶端。当它被显示的时候，它被父窗口部件的边界挡住了一部分。

父窗口部件，QVBox，自动地把这个子窗口部件添加到它的盒子中央。因为没有其它的东西被添加了，这个按钮就获得了父窗口部件的所有空间。

```
        box.show();
```

当父窗口部件被显示的时候，它会调用所有子窗口部件的显示函数（除非在这些子窗口部件中你已经明确地使用 QWidget::hide()）。

行为

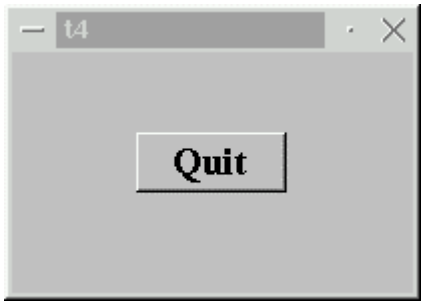
这个按钮不再充满整个窗口部件。相反，它获得了一个“自然的”大小。这是因为现在的这个新的顶层窗口，使用了按钮的大小提示和大小变化策略来设置这个按钮的大小和位置。（请看 QWidget::sizeHint()和 QWidget::setSizePolicy()来获得关于这几个函数的更详细的信息。）

（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

试着改变窗口的大小。按钮是如何变化的？按钮的大小变化策略是什么？如果你运行这个程序的时候使用了一个大一些的字体，按钮的高度发生了什么变化？如果你试图让这个窗口*真的*变小，发生了什么？
现在你可以进行第四章了。

Qt 教程一 —— 第四章：使用窗口部件



这个例子显示了如何创建一个你自己的窗口部件，描述如何控制一个窗口部件的最小大小和最大大小，并且介绍了窗口部件的名称。

```
/****************************************************************
**
** Qt 教程一 - 4
**
****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>


class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};


MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );

    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 62, 40, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
```

```
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}


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

    MyWidget w;
    w.setGeometry( 100, 100, 200, 120 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

一行一行地解说

```
    class MyWidget : public QWidget
    {
    public:
        MyWidget( QWidget *parent=0, const char *name=0 );
    };
```

这里我们创建了一个新类。因为这个类继承了 QWidget，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（像第三章里面的按钮）。
这个类只有一个成员函数，构造函数（加上从 QWidget 继承来的成员函数）。这个构造函数是一个标准的 Qt 窗口部件构造函数，当你创建窗口部件时，你应该总是包含一个相似的构造函数。
第一个参数是它的父窗口部件。为了生成一个顶层窗口，你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。

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

构造函数的实现从这里开始。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。

```
    {
        setMinimumSize( 200, 120 );
        setMaximumSize( 200, 120 );
```

因为这个窗口部件不知道如何处理重新定义大小，我们把它的最小大小和最大大小设置为相等的值，这样我们就确定了它的大小。在下一章，我们将演示窗口部件如何响应用户的重新定义大小事件。

```
        QPushButton *quit = new QPushButton( "Quit", this, "quit" );
```

```
        quit->setGeometry( 62, 40, 75, 30 );
        quit->setFont( QFont( "Times", 18, QFont::Bold ) );
```

这里我们创建并设置了这个窗口部件的一个名称为“quit”的子窗口部件（新窗口部件的父窗口部件是 this）。这个窗口部件名称和按钮文本没有关系，只是在这一情况下碰巧相似。

注意 quit 是这个构造函数中的局部变量。MyWidget 不能跟踪它，但 Qt 可以，当 MyWidget 被删除的时候，默认地它也会被删除。这就是为什么 MyWidget 不需要一个析构函数的原因。（另外一方面，如果你选择删除一个子窗口部件，也没什么坏处，这个子窗口部件会自动告诉 Qt 它即将死亡。）

setGeometry()调用和上一章的 move()和 resize()是一样的。

```
        connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
    }
```

因为 MyWidget 类不知道这个应用程序对象，它不得不连接到 Qt 的指针，qApp。一个窗口部件就是一个软件组件并且它应该尽量少地知道关于它的环境，因为它应该尽可能的通用和可重用。

知道了应用程序的名称将会打破上述原则，所以在一个组件，比如 MyWidget，需要和应用程序对象对话的这种情况下，Qt 提供了一个别名，qApp。

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

        MyWidget w;
        w.setGeometry( 100, 100, 200, 120 );
        a.setMainWidget( &w );
        w.show();
        return a.exec();
    }
```

这里我们举例说明了我们的新子窗口部件，把它设置为主窗口部件，并且执行这个应用程序。

行为

这个程序和上一章的在行为上非常相似。不同点是我们实现的方式。无论如何它的行为还是有一些小差别。试试改变它的大小，你会看到什么？

（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

试着在 main()中创建另一个 MyWidget 对象。发生了什么？

试着添加更多的按钮或者把除了 QPushButton 之外的东西放到窗口部件中。

现在你可以进行第五章了。

Qt 教程一 —— 第五章：组装积木

这个例子显示了创建几个窗口部件并用信号和槽把它们连接起来，和如何处理重新定义大小事件。

```
/****************************************************************
**
** Qt 教程一 - 5
**
****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>

#include <qvbox.h>

class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};


MyWidget::MyWidget( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

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

    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );

    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
```

```
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}

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

    MyWidget w;
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

一行一行地解说

```
    #include <qapplication.h>
    #include <qpushbutton.h>
    #include <qslider.h>
    #include <qlcdnumber.h>
    #include <qfont.h>

    #include <qvbox.h>
```

这里显示的是三个新的被包含的头文件。qslider.h 和 qlcdnumber.h 在这里是因为我们使用了两个新的窗口部件，QSlider 和 QLCDNumber。qvbox.h 在这里是因为我们使用了 Qt 的自动布局支持。

```
    class MyWidget : public QVBox
    {
    public:
        MyWidget( QWidget *parent=0, const char *name=0 );
    };
    MyWidget::MyWidget( QWidget *parent, const char *name )
            : QVBox( parent, name )
    {
```

MyWidget 现在继承了 QVBox，而不是 QWidget。我们通过这种方式来使用 QVBox 的布局（它可以把它的子窗口部件垂直地放在自己里面）。重新定义大小自动地被 QVBox 处理，因此现在也就被 MyWidget 处理了。

```
        QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );
```

lcd 是一个 QLCDNumber，一个可以按像 LCD 的方式显示数字的窗口部件。这个实例被设置为显示两个数字，并且是 *this* 的子窗口部件。它被命名为“lcd”。

```
        QSlider * slider = new QSlider( Horizontal, this, "slider" );
        slider->setRange( 0, 99 );
        slider->setValue( 0 );
```

QSlider 是一个经典的滑块，用户可以通过在拖动一个东西在一定范围内调节一个整数数值的方式来使用这个窗口部件。这里我们创建了一个水平的滑块，设置它的范围是 0~99（包括 0 和 99，参见 QSlider::setRange()文档）并且它的初始值是 0。

```
connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
```

这里我们是用了信号/槽机制把滑块的 valueChanged()信号和 LCD 数字的 display()槽连接起来了。
无论什么时候滑块的值发生了变化，它都会通过发射 valueChanged()信号来广播这个新的值。因为这个信号已经和 LCD 数字的 display()槽连接起来了，当信号被广播的时候，这个槽就被调用了。这两个对象中的任何一个都不知道对方。这就是组件编程的本质。
槽是和普通 C++成员函数的方式不同，但有着普通 C++成员函数的方位规则。

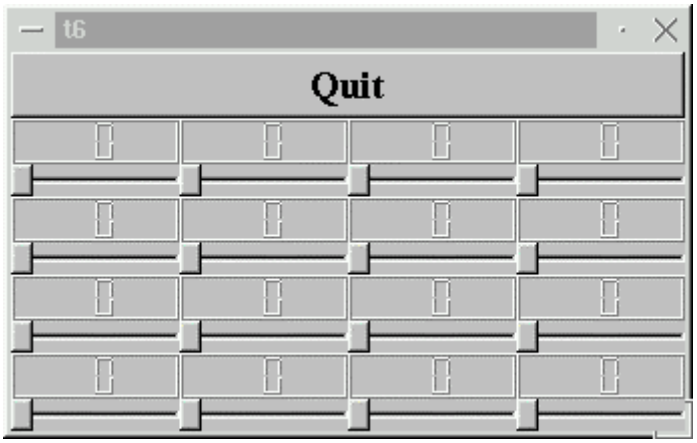
行为

LCD 数字反应了你对滑块做的一切，并且这个窗口部件很好地处理了重新定义大小事件。注意当窗口被重新定义大小（因为它可以）的时候，LDC 数字窗口部件也改变了大小，但是其它的还是和原来一样（因为如果它们变化了，看起来好像很傻）。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

试着改变 LCD 数字，添加更多的数字或者改变模式。你甚至可以添加四个按钮来设置基数。
你也可以改变滑块的范围。
也许使用 QSpinBox 比滑块更好？
试着当 LCD 数字溢出的时候使这个应用程序退出。
现在你可以进行第六章了。

Qt 教程一 —— 第六章：组装丰富的积木！



这个例子显示了如何把两个窗口部件封装成一个新的组件和使用许多窗口部件是多么的容易。首先，我们使用一个自定义的窗口部件作为一个子窗口部件。

```
/****************************************************************
**
** Qt 教程一 - 6
**
****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
#include <qgrid.h>

class LCDRange : public QVBox
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
};

LCDRange::LCDRange( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}

class MyWidget : public QVBox
```

```
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};


MyWidget::MyWidget( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

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

    QGrid *grid = new QGrid( 4, this );

    for( int r = 0 ; r < 4 ; r++ )
        for( int c = 0 ; c < 4 ; c++ )
            (void)new LCDRange( grid );
}

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

    MyWidget w;
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

一行一行地解说

```
    class LCDRange : public QVBox
    {
    public:
        LCDRange( QWidget *parent=0, const char *name=0 );
    };
```

LCDRange 窗口部件是一个没有任何 API 的窗口部件。它只有一个构造函数。这种窗口部件不是很有用，所以我们一会儿会加入一些 API。

```
    LCDRange::LCDRange( QWidget *parent, const char *name )
            : QVBox( parent, name )
    {
        QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );
```

```
        QSlider * slider = new QSlider( Horizontal, this, "slider" );
        slider->setRange( 0, 99 );
        slider->setValue( 0 );
        connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
    }
```

这里直接利用了第五章里面的 MyWidget 的构造函数。唯一的不同是按钮被省略了并且这个类被重新命名了。

```
    class MyWidget : public QVBox
    {
    public:
        MyWidget( QWidget *parent=0, const char *name=0 );
    };
```

MyWidget 也是除了一个构造函数之外没有包含任何 API。

```
    MyWidget::MyWidget( QWidget *parent, const char *name )
            : QVBox( parent, name )
    {
        QPushButton *quit = new QPushButton( "Quit", this, "quit" );
        quit->setFont( QFont( "Times", 18, QFont::Bold ) );

        connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

这个按钮被放在 LCDRange 中，这样我们就有了一个“Quit”按钮和许多 LCDRange 对象。

```
        QGrid *grid = new QGrid( 4, this );
```

我们创建了一个四列的 QGrid 对象。这个 QGrid 窗口部件可以自动地把自己地子窗口部件排列到行列中，你可以指定行和列的数量，并且 QGrid 可以发现它的新子窗口部件并且把它们安放到网格中。

```
        for( int r = 0 ; r < 4 ; r++ )
            for( int c = 0 ; c < 4 ; c++ )
                (void)new LCDRange( grid );
```

四行，四列。

我们创建了一个 4*4 个 LCDRanges，所有这些都是这个 grid 对象的子窗口部件。这个 QGrid 窗口部件会安排它们。

```
    }
```

这就是全部了。

行为

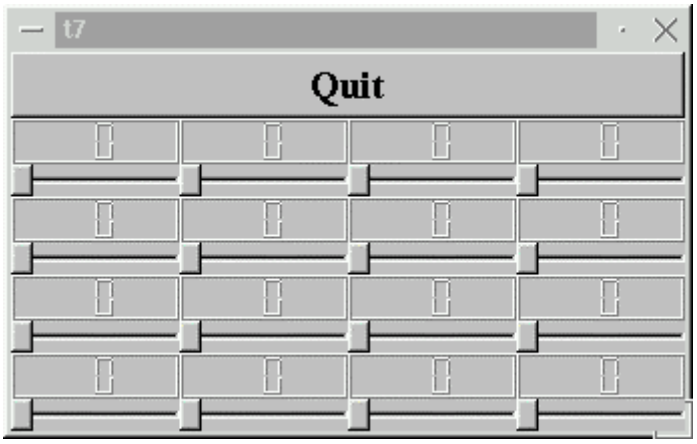
这个程序显示了在同一时间使用许多窗口部件是多么的容易。其中的滑块和 LCD 数字的行为在前一章已经提到过了。还有就是，就是实现的不同。

（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

在开始的时候使用不同的或者随机的值初始化每个滑块。
源代码中的“4”出现了 3 次。如果你改变 QGrid 构造函数中调用的那个，会发生什么？改变另外两个又会发生什么呢？为什么呢？
现在你可以进行第七章了。

Qt 教程一 —— 第七章：一个事物领导另一个



这个例子显示了如何使用信号和槽来创建自定义窗口部件，和如何使用更加复杂的方式把它们连接起来。首先，源文件被我们分成几部分并放在放在 t7 目录下。

- t7/lcdrange.h 包含 LCDRange 类定义。
- t7/lcdrange.cpp 包含 LCDRange 类实现。
- t7/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t7/lcdrange.h

这个文件主要利用了第六章的 main.cpp，在这里只是说明一下改变了哪些。

```
#ifndef LCDRANGE_H
#define LCDRANGE_H
```

这里是一个经典的 C 语句，为了避免出现一个头文件被包含不止一次的情况。如果你没有使用过它，这是开发中的一个很好的习惯。#ifndef 需要把这个头文件的*全部*都包含进去。

```
#include <qvbox.h>
```

qvbox.h 被包含了。LCDRange 继承了 QVBox，所以父类的头文件必须被包含。我们在前几章里面偷了一点懒，我们通过包含其它一些头文件，比如 qpushbutton.h，这样就可以间接地包含 qwidget.h。

```
class QSlider;
```

这里是另外一个小伎俩，但是没有前一个用的多。因为我们在类的*界面*中不需要QSlider，仅仅是在实现中，我们在头文件中使用一个前置的类声明，并且在.cpp文件中包含一个 QSlider 的头文件。
这会使编译一个大的项目变得更快，因为当一个头文件改变的时候，很少的文件需要重新编译。它通常可以给大型编译加速两倍或两倍以上。

```
class LCDRange : public QVBox
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
```

meta object file. 注意 Q_OBJECT。这个宏必须被包含到*所有*使用信号和/或槽的类。如果你很好奇，它定义了在元对象文件中实现的一些函数。

```
    int value() const;
public slots:
    void setValue( int );

signals:
    void valueChanged( int );
```

这三个成员函数构成了这个窗口部件和程序中其它组件的接口。直到现在，LCDRange 根本没有一个真正的接口。
value()是一个可以访问 LCDRange 的值的公共函数。setValue()是我们第一个自定义槽，并且 valueChanged()是我们第一个自定义信号。
槽必须按通常的方式实现（记住槽也是一个 C++成员函数）。信号可以在元对象文件中自动实现。信号也遵守 C++函数的保护法则（比如，一个类只能发射它自己定义的或者继承来的信号）。
当 LCDRange 的值发生变化时，valueChanged()信号就会被使用——你从这个名字中就可以猜到。这将不会是你将会看到的命名为 *something*Changed()的最后一个信号。

t7/lcdrange.cpp

这个文件主要利用了 t6/main.cpp，在这里只是说明一下改变了哪些。

```
    connect( slider, SIGNAL(valueChanged(int)),
             lcd, SLOT(display(int)) );
    connect( slider, SIGNAL(valueChanged(int)),
             SIGNAL(valueChanged(int)) );
```

这个代码来自 LCDRange 的构造函数。
第一个 connect 和你在上一章中看到的一样。第二个是新的，它把滑块的 valueChanged()信号和这个对象的 valueChanged *信号*连接起来了。带有三个参数的 connect()函数连接到 this 对象的信号或槽。
是的，这是正确的。信号可以被连接到其它的信号。当第一个信号被发射时，第二个信号也被发射。

让我们来看看当用户操作这个滑块的时候都发生了些什么。滑块看到自己的值发生了改变，并发射了 valueChanged()信号。这个信号被连接到 QLCDNumber 的 display()槽和 LCDRange 的 valueChanged()信号。
所以，当这个信号被发射的时候，LCDRange 发射它自己的 valueChanged()信号。另外，QLCDNumber::display()被调用并显示新的数字。
注意你并没有保证执行的任何顺序——LCDRange::valueChanged()也许在 QLCDNumber::display()之前或者之后发射，这是完全任意的。

```
int LCDRange::value() const
{
    return slider->value();
}
```

value()的实现是直接了当的，它简单地返回滑块的值。

```
void LCDRange::setValue( int value )
{
    slider->setValue( value );
}
```

setValue()的实现是相当直接了当的。注意因为滑块和 LCD 数字是连接的，设置滑块的值就会自动的改变 LCD 数字的值。另外，如果滑块的值超过了合法范围，它会自动调节。

t7/main.cpp

```
    LCDRange *previous = 0;
    for( int r = 0 ; r < 4 ; r++ ) {
        for( int c = 0 ; c < 4 ; c++ ) {
            LCDRange* lr = new LCDRange( grid );
            if ( previous )
                connect( lr, SIGNAL(valueChanged(int)),
                         previous, SLOT(setValue(int)) );
            previous = lr;
        }
    }
```

main.cpp 中所有的部分都是上一章复制的，除了 MyWidget 的构造函数。当我们创建 16 个 RCDRange 对象时，我们现在使用信号/槽机制连接它们。每一个的 valueChanged()信号都和前一个的 setValue()槽连接起来了。因为当 LCDRange 的值发生改变的时候，发射一个 valueChanged()信号（惊奇！），我们在这里创建了一个信号和槽的“链”。

编译

为一个多文件的应用程序创建一个 makefile 和为一个单文件的应用程序创建一个 makefile 是没有什么不同的。如果你已经把这个例子中的所有文件都保存到它们自己的目录中，你所要做的就是这些：

```
qmake -project
```

qmake
第一个命令调用 qmake 来生成一个.pro（项目）文件。第二个命令根据这个项目文件来生成一个（系统相关的）makefile。你现在可以输入 make（或者 nmake，如果你使用 Visual Studio）。

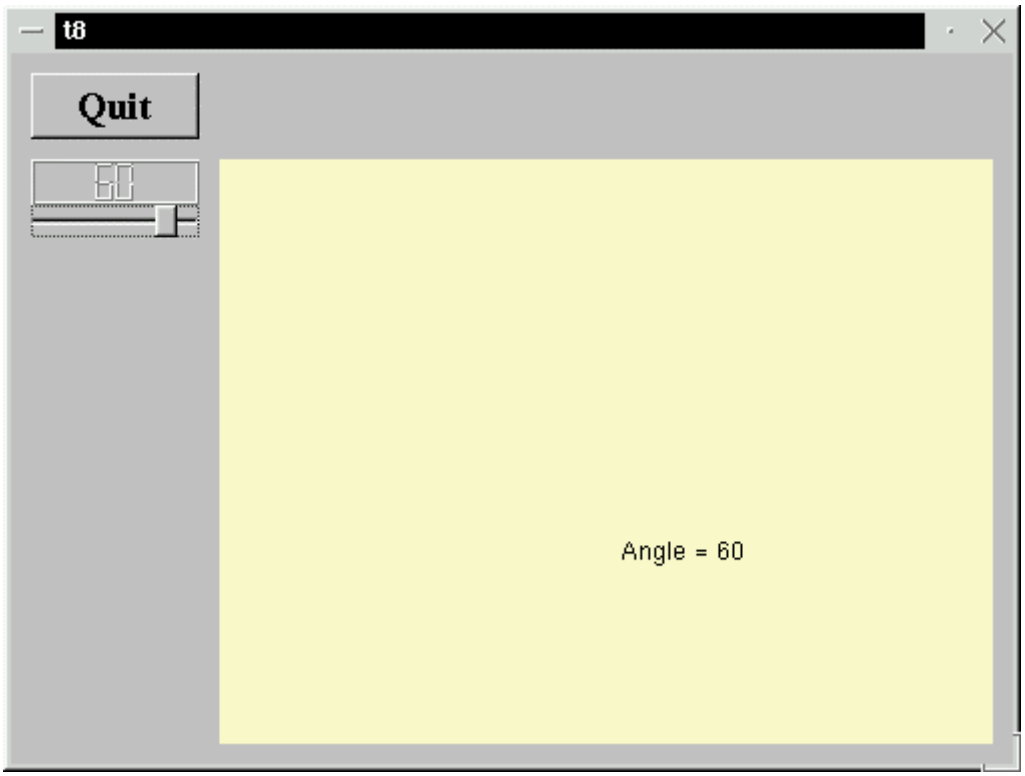
行为

在开始的时候，这个程序看起来和上一章里的一样。试着操作滑块到右下角……

练习

seven LCDs back to 50. 使用右下角的滑块并设置所有的 LCD 到 50。然后设置通过点击这个滑块的左侧把它设置为 40。现在，你可以通过把最后一个调到左边来把前七个 LCD 设置回 50。
点击右下角滑块的滑块的左边。发生了什么？为什么只是正确的行为？
现在你可以进行第八章了。

Qt 教程一 —— 第八章：准备战斗



在这个例子中，我们介绍可以画自己的第一个自定义窗口部件。我们也加入了一个有用的键盘接口（只用了两行代码）。

- t8/lcdrange.h 包含 LCDRange 类定义。

- t8/lcdrange.cpp 包含 LCDRange 类实现。
- t8/cannon.h 包含 CannonField 类定义。
- t8/cannon.cpp 包含 CannonField 类实现。
- t8/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t8/lcdrange.h

这个文件和第七章中的 lcdrange.h 很相似。我们添加了一个槽：setRange()。

```
    void setRange( int minVal, int maxVal );
```

现在我们添加了设置 LCDRange 范围的可能性。直到现在，它就可以被设置为 0～99。

t8/lcdrange.cpp

在构造函数中有一个变化（稍后我们会讨论的）。

```
void LCDRange::setRange( int minVal, int maxVal )
{
    if ( minVal < 0 || maxVal > 99 || minVal > maxVal ) {
      qWarning( "LCDRange::setRange(%d,%d)\n"
                "\tRange must be 0..99\n"
                "\tand minVal must not be greater than maxVal",
                minVal, maxVal );
      return;
    }
    slider->setRange( minVal, maxVal );
}
```

setRange()设置了 LCDRange 中滑块的范围。因为我们已经把 QLCDNumber 设置为只显示两位数字了，我们想通过限制 minVal 和 maxVal 为 0~99 来避免 QLCDNumber 的溢出。（我们可以允许最小值为-9，但是我们没有那样做。）如果参数是非法的，我们使用 Qt 的 qWarning()函数来向用户发出警告并立即返回。qWarning()是一个像 printf 一样的函数，默认情况下它的输出发送到 stderr。如果你想改变的话，你可以使用::qInstallMsgHandler()函数安装自己的处理函数。

t8/cannon.h

CanonField 是一个知道如何显示自己的新的自定义窗口部件。

```
class CannonField : public QWidget
{
    Q_OBJECT
public:
    CannonField( QWidget *parent=0, const char *name=0 );
```

CanonField 继承了 QWidget，我们使用了 LCDRange 中同样的方式。

```
    int angle() const { return ang; }
    QSizePolicy sizePolicy() const;

public slots:
    void setAngle( int degrees );

signals:
    void angleChanged( int );
```

目前，CanonField 只包含一个角度值，我们使用了 LCDRange 中同样的方式。

```
protected:
    void paintEvent( QPaintEvent * );
```

这是我们在 QWidget 中遇到的许多事件处理器中的第二个。只要一个窗口部件需要刷新它自己（比如，画窗口部件表面），这个虚函数就会被 Qt 调用。

t8/cannon.cpp

```
CannonField::CannonField( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
```

我们又一次使用和前一章中的 LCDRange 同样的方式。

```
    ang = 45;
    setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

构造函数把角度值初始化为 45 度并且给这个窗口部件设置了一个自定义调色板。

这个调色板只是说明背景色，并选择了其它合适的颜色。（对于这个窗口部件，只有背景色和文本颜色是要用到的。）

```
void CannonField::setAngle( int degrees )
{
    if ( degrees < 5 )
        degrees = 5;
    if ( degrees > 70 )
        degrees = 70;
    if ( ang == degrees )
        return;
    ang = degrees;
    repaint();
    emit angleChanged( ang );
}
```

这个函数设置角度值。我们选择了一个 5～70 的合法范围，并根据这个范围来调节给定的 degrees 的值。当新的角度值超过了范围，我们选择了不使用警告。

如果新的角度值和旧的一样，我们立即返回。这只对当角度值*真的*发生变化时，发射 angleChanged()信号有重要意义。

然后我们设置新的角度值并重新画我们的窗口部件。QWidget::repaint()函数清空窗口部件（通常用背景色来充满）并向窗口部件发出一个绘画事件。这样的结构就是调用窗口部件的绘画事件函数一次。
最后，我们发射 angleChanged()信号来告诉外面的世界，角度值发生了变化。emit 关键字只是 Qt 中的关键字，而不是标准 C++的语法。实际上，它只是一个宏。

```
void CannonField::paintEvent( QPaintEvent * )
{
    QString s = "Angle = " + QString::number( ang );
    QPainter p( this );
    p.drawText( 200, 200, s );
}
```

这是我们第一次试图写一个绘画事件处理程序。这个事件参数包含一个绘画事件的描述。QPaintEvent 包含一个必须被刷新的窗口部件的区域。现在，我们比较懒惰，并且只是画每一件事。
我们的代码在一个固定位置显示窗口部件的角度值。首先我们创建一个含有一些文本和角度值的 QString，然后我们创建一个操作这个窗口部件的 QPainter 并使用它来画这个字符串。我们一会儿会回到 QPainter，它可以做很多事。

t8/main.cpp

```
#include "cannon.h"
```

我们包含了我们的新类：

```
class MyWidget: public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
```

这一次我们在顶层窗口部件中只使用了一个 LCDRange 和一个 CanonField。

```
    LCDRange *angle = new LCDRange( this, "angle" );
```

在构造函数中，我们创建并设置了我们的 LCDRange。

```
    angle->setRange( 5, 70 );
```

我们设置 LCDRange 能够接受的范围是 5～70 度。

```
    CannonField *cannonField
        = new CannonField( this, "cannonField" );
```

我们创建了我们的 CannonField。

```
    connect( angle, SIGNAL(valueChanged(int)),
             cannonField, SLOT(setAngle(int)) );
    connect( cannonField, SIGNAL(angleChanged(int)),
             angle, SLOT(setValue(int)) );
```

这里我们把 LCDRange 的 valueChanged()信号和 CannonField 的 setAngle()槽连接起来了。只要用户操作 LCDRange，就会刷新 CannonField 的角度值。我们也把它反过来连接了，这样 CannonField 中角度的变化就可以刷新 LCDRange 的值。在我们的例子中，我们从来没有直接改变 CannonField 的角度，但是通过我们的

最后一个 connect()我们就可以确保没有任何变化可以改变这两个值之间的同步关系。
这说明了组件编程和正确封装的能力。
注意只有当角度确实发生变化时，才发射 angleChanged()是多么的重要。如果 LCDRange 和 CanonField 都省略了这个检查，这个程序就会因为第一次数值变化而进入到一个无限循环当中。

```
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
//2×2，10 像素的边界
```

到现在为止，我们没有因为几何管理把 QVBox 和 QGrid 窗口部件集成到一起。现在，无论如何，我们需要对我们的布局加一些控制，所以我们使用了更加强大的 QGridLayout 类。QGridLayout 不是一个窗口部件，它是一个可以管理*任何*窗口部件作为子对象的不同的类。
就像注释中所说的，我们创建了一个以 10 像素为边界的 2*2 的数组。
（QGridLayout 的构造函数有一点神秘，所以最好在这里加入一些注释。）

```
grid->addWidget( quit, 0, 0 );
```

我们在网格的左上的单元格中加入一个 Quit 按钮：0,0。

```
grid->addWidget( angle, 1, 0, Qt::AlignTop );
```

我们把 angle 这个 LCDRange 放到左下的单元格，在单元格内向上对齐。（这只是 QGridLayout 所允许的一种对齐方式，而 QGrid 不允许。）

```
grid->addWidget( cannonField, 1, 1 );
```

我们把 CannonField 对象放到右下的单元格。（右上的单元格是空的。）

```
grid->setColStretch( 1, 10 );
```

我们告诉 QGridLayout 右边的列（列 1）是可拉伸的。因为左边的列不是（它的拉伸因数是 0，这是默认值），QGridLayout 就会在 MyWidget 被重新定义大小的时候试图让左面的窗口部件大小不变，而重新定义 CannonField 的大小。

```
angle->setValue( 60 );
```

我们设置了一个初始角度值。注意这将会引发从 LCDRange 到 CannonField 的连接。

```
angle->setFocus();
```

我们刚才做的是设置 angle 获得键盘焦点，这样默认情况下键盘输入会到达 LCDRange 窗口部件。
LCDRange 没有包含任何 keyPressEvent()，所以这看起来不太可能有用。无论如何，它的构造函数中有了新的一行：

```
setFocusProxy( slider );
```

LCDRange 设置滑块作为它的焦点代理。这就是说当程序或者用户想要给 LCDRange 一个键盘焦点，滑块就会就会注意到它。QSlider 有一个相当好的键盘接口，所以就会出现我们给 LCDRange 添加的这一行。

行为

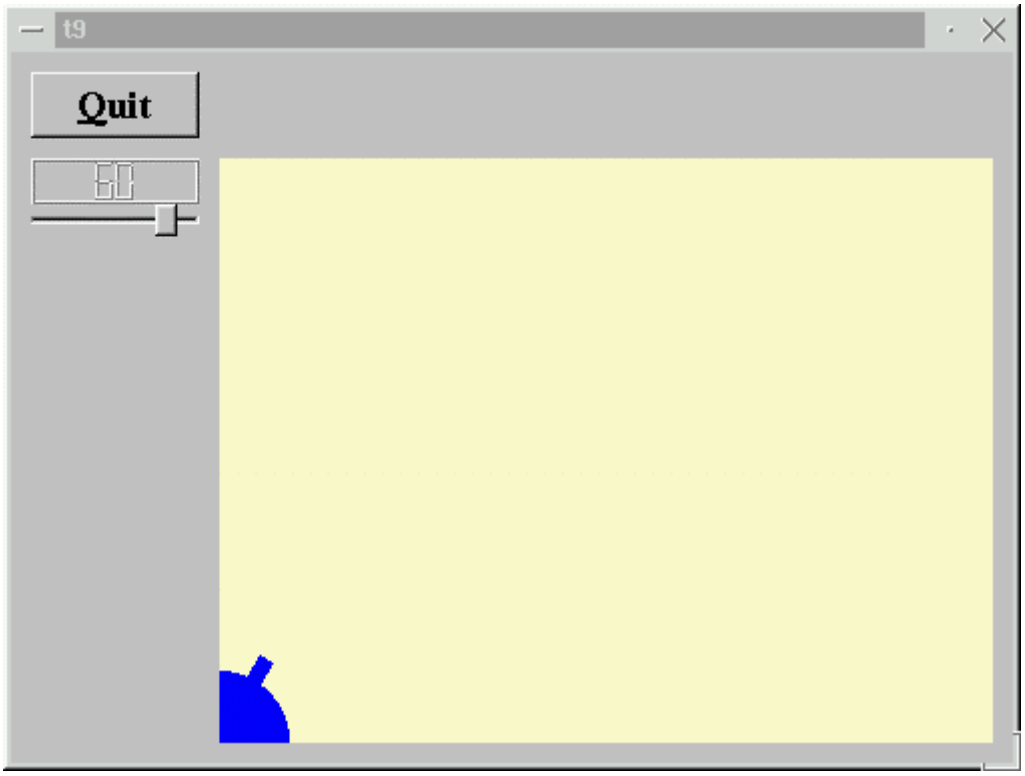
键盘现在可以做一些事了——方向键、Home、End、PageUp 和 PageDown 都可以作一些事情。

当滑块被操作，CannonFiled 会显示新的角度值。如果重新定义大小，CannonField 会得到尽可能多的空间。
在 8 位的 Windows 机器上显示新的颜色会颤动的要命。下一章会处理这些的。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

设置重新定义窗口的大小。如果你把它变窄或者变矮会发生什么？
如果你把 AlignTop 删掉，LCDRange 的位置会发生什么变化？为什么？
如果你给左面的列一个非零的拉伸因数，当你重新定义窗口大小时会发生什么？
不考虑 setFocus()调用。你更喜欢什么样的行为？
试着在 QButton::setText()调用中把“Quit”改为“&Quit”。按钮看起来变成什么样子了？如果你在程序运行的时候按下 Alt+Q 会发生什么？（在少量键盘中时 Meta+Q。）
把 CannonField 的文本放到中间。
现在你可以进行第九章了。

Qt 教程一 —— 第九章：你可以使用加农炮了



在这个例子中我们开始画一个蓝色可爱的小加农炮.只 cannon.cpp 和上一章不同。

- t9/lcdrange.h 包含 LCDRange 类定义。

- t9/lcdrange.cpp 包含 LCDRange 类实现。
- t9/cannon.h 包含 CannonField 类定义。
- t9/cannon.cpp 包含 CannonField 类实现。
- t9/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t9/cannon.cpp

```
void CannonField::paintEvent( QPaintEvent * )
{
    QPainter p( this );
```

我们现在开始认真地使用 QPainter。我们创建一个绘画工具来操作这个窗口部件。

```
    p.setBrush( blue );
```

当一个 QPainter 填满一个矩形、圆或者其它无论什么，它会用它的画刷填满这个图形。这里我们把画刷设置为蓝色。（我们也可以使用一个调色板。）

```
    p.setPen( NoPen );
```

并且 QPainter 使用画笔来画边界。这里我们设置为 NoPen，就是说我们在边界上什么都不画，蓝色画刷会在我们画的东西的边界内画满全部。

```
    p.translate( 0, rect().bottom() );
```

QPainter::translate()函数转化 QPainter 的坐标系统，比如，它通过偏移谅来移动。这里我们设置窗口部件的左下角为(0,0)。x 和 y 的方向没有改变，比如，窗口部件中的所有 y 坐标现在都是负数（请看坐标系统获得有关 Qt 的坐标系统更多的信息。）

```
    p.drawPie( QRect(-35, -35, 70, 70), 0, 90*16 );
```

drawPie()函数使用一个开始角度和弧长在一个指定的矩形内画一个饼型图。角度的度量用的是一度的十六分之一。零度在三点的位置。画的方向是顺时针的。这里我们在窗口部件的左下角画一个四分之一圆。这个饼图被蓝色充满，并且没有边框。

```
    p.rotate( -ang );
```

QPainter::rotate()函数绕 QPainter 坐标系统的初始位置旋转它。旋转的参数是一个按度数给定的浮点数（不是一个像上面那样给的十六分之一的度数）并且是顺时针的。这里我们顺时针旋转 ang 度数。

```
    p.drawRect( QRect(33, -4, 15, 8) );
```

QPainter::drawRect()函数画一个指定的矩形。这里我们画的是加农炮的炮筒。很难想象当坐标系统被转换之后（转化、旋转、缩放或者修剪）的绘画结果。在这种情况下，坐标系统先被转化后被旋转。如果矩形 QRect(33, -4, 15, 8)被画到这个转化后的坐标系统中，它看起来会是这样：

注意矩形被 CannonField 窗口部件的边界省略了一部分。当我们选装坐标系统，以 60 度为例，矩形会以(0,0)为圆心被旋转，也就是左下角，因为我们已经转化了坐标系统。结果会是这样：

我们做完了，除了我们还没有解释为什么 Windows 在这个时候没有发抖。

```
int main( int argc, char **argv )
{
    QApplication::setColorSpec( QApplication::CustomColor );
    QApplication a( argc, argv );
```

我们告诉 Qt 我们在这个程序中想使用一个不同的颜色分配策略。这里没有单一正确的颜色分配策略。因为这个程序使用了不常用的黄色，但不是很多颜色，CustomColor 最好。这里有几个其它的分配策略，你可以在 QApplication::setColorSpec()文档中读到它们。
通常情况下你可以忽略这一点，因为默认的是好的。偶尔一些使用常用颜色的应用程序看起来比较糟糕，因而改变分配策略通常会有所帮助。

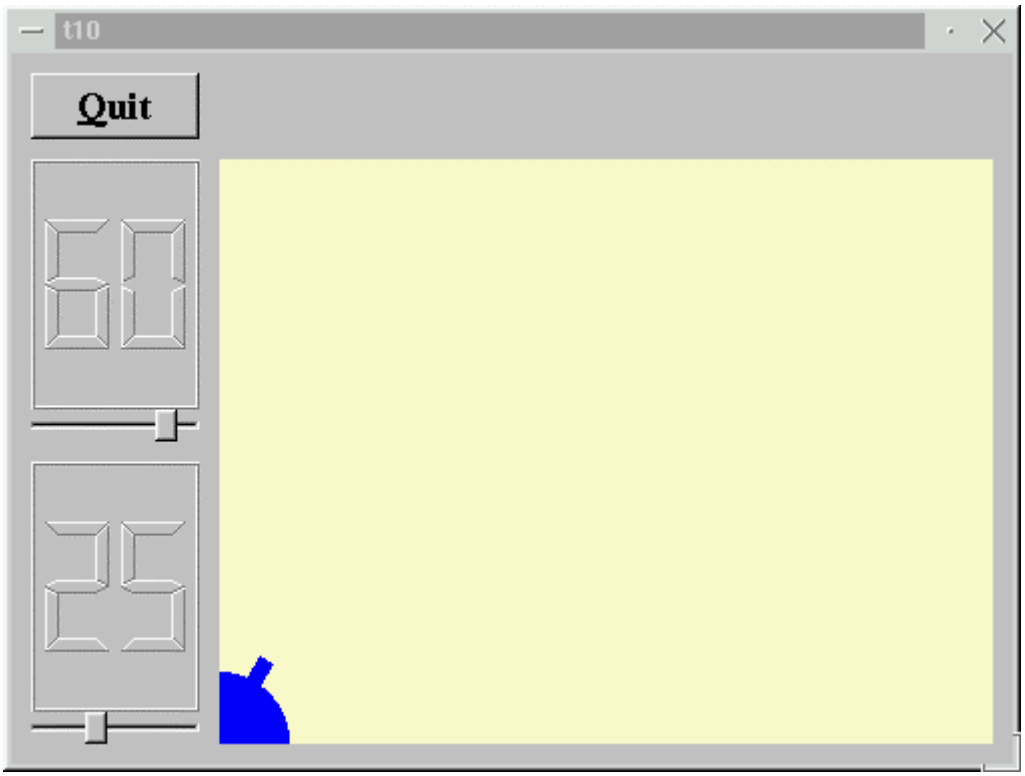
行为

当滑块被操作的时候，所画的加农炮的角度会因此而变化。
Quit 中的字母 Q 现在有下划线，并且 Alt+Q 会实现你所要的。如果你不知道这些，你一定是没有做第八章中的练习。
你也要注意加农炮的闪烁让人很烦，特别是在一个比较慢的机器上。我们将会在下一章修正这一点。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

设置一个不同的画笔代替 NoPen。设置一个调色板的画刷。
试着用“Q&uit”或者“Qu&it”作为按钮的文本来提到“&Quit”。发生了什么？
现在你可以进行第十章了。

Qt 教程一 —— 第十章：像丝一样滑



在这个例子中，我们介绍画一个 pixmap 来除去闪烁。我们也会加入一个力量控制。

- t10/lcdrange.h 包含 LCDRange 类定义。
- t10/lcdrange.cpp 包含 LCDRange 类实现。
- t10/cannon.h 包含 CannonField 类定义。
- t10/cannon.cpp 包含 CannonField 类实现。
- t10/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t10/cannon.h

CannonField 现在除了角度又多了一个力量值。

```
        int   angle() const { return ang; }
        int   force() const { return f; }

    public slots:
        void  setAngle( int degrees );
        void  setForce( int newton );

    signals:
        void  angleChanged( int );
        void  forceChanged( int );
```

力量的接口的实现和角度一样。

```
    private:
        QRect cannonRect() const;
```
我们把加农炮封装的矩形的定义放到了一个单独的函数中。
```
        int ang;
        int f;
    };
```
力量被存储到一个整数 f 中。

t10/cannon.cpp

```
    #include <qpixmap.h>
```
我们包含了 QPixmap 类定义。
```
    CannonField::CannonField( QWidget *parent, const char *name )
            : QWidget( parent, name )
    {
        ang = 45;
        f = 0;
        setPalette( QPalette( QColor( 250, 250, 200) ) );
    }
```
力量（f）被初始化为 0。
```
    void CannonField::setAngle( int degrees )
    {
        if ( degrees < 5 )
            degrees = 5;
        if ( degrees > 70 )
            degrees = 70;
        if ( ang == degrees )
            return;
        ang = degrees;
        repaint( cannonRect(), FALSE );
        emit angleChanged( ang );
    }
```
我们在 setAngle()函数中做了一个小的改变。它只重画窗口部件中含有加农炮的一小部分。FALSE 参数说明在一个绘画事件发送到窗口部件之前指定的矩形将不会被擦去。这将会使绘画过程加速和平滑。
```
    void CannonField::setForce( int newton )
    {
        if ( newton < 0 )
            newton = 0;
        if ( f == newton )
            return;
        f = newton;
        emit forceChanged( f );
    }
```

setForce()的实现和 setAngle()很相似。唯一的不同是因为我们不显示力量值，我们不需要重画窗口部件。

```
void CannonField::paintEvent( QPaintEvent *e )
{
    if ( !e->rect().intersects( cannonRect() ) )
        return;
```

我们现在用只重画需要刷新得部分来优化绘画事件。首先我们检查是否不得不完全重画任何事，我们返回是否不需要。

```
    QRect cr = cannonRect();
    QPixmap pix( cr.size() );
```

然后，我们创建一个临时的 pixmap，我们用来不闪烁地画。所有的绘画操作都在这个 pixmap 中完成，并且之后只用一步操作来把这个 pixmap 画到屏幕上。这是不闪烁绘画的本质：一次准确地在每一个像素上画。更少，你会得到绘画错误。更多，你会得到闪烁。在这个例子中这个并不重要——当代码被写时，仍然是很慢的机器导致闪烁，但以后不会再闪烁了。我们由于教育目的保留了这些代码。

```
    pix.fill( this, cr.topLeft() );
```

我们用这个 pixmap 来充满这个窗口部件的背景。

```
    QPainter p( &pix );
    p.setBrush( blue );
    p.setPen( NoPen );
    p.translate( 0, pix.height() - 1 );
    p.drawPie( QRect( -35,-35, 70, 70 ), 0, 90*16 );
    p.rotate( -ang );
    p.drawRect( QRect(33, -4, 15, 8) );
    p.end();
```

我们就像第九章中一样画，但是现在我们是在 pixmap 上画。
在这一点上，我们有一个绘画工具变量和一个 pixmap 看起来相当正确，但是我们还没有在屏幕上画呢。

```
    p.begin( this );
    p.drawPixmap( cr.topLeft(), pix );
```

所以我们在 CannonField 上面打开绘图工具并在这之后画这个 pixmap。
这就是全部了。在顶部和底部各有一对线，并且这个代码是 100%不闪烁的。

```
QRect CannonField::cannonRect() const
{
    QRect r( 0, 0, 50, 50 );
    r.moveBottomLeft( rect().bottomLeft() );
    return r;
}
```

这个函数返回一个在窗口部件坐标中封装加农炮的矩形。首先我们创建一个 50*50 大小的矩形，然后移动它，使它的左下角和窗口部件自己的左下角相等。QWidget::rect()函数在窗口部件自己的坐标（左上角是 0,0）中返回窗口部件封装的矩形。

t10/main.cpp

```
MyWidget::MyWidget( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
```

构造函数也是一样，但是已经加入了一些东西。

```
    LCDRange *force  = new LCDRange( this, "force" );
    force->setRange( 10, 50 );
```

我们加入了第二个 LCDRange，用来设置力量。

```
    connect( force, SIGNAL(valueChanged(int)),
             cannonField, SLOT(setForce(int)) );
    connect( cannonField, SIGNAL(forceChanged(int)),
             force, SLOT(setValue(int)) );
```

我们把 force 窗口部件和 cannonField 窗口部件连接起来，就和我们对 angle 窗口部件做的一样。

```
    QVBoxLayout *leftBox = new QVBoxLayout;
    grid->addLayout( leftBox, 1, 0 );
    leftBox->addWidget( angle );
    leftBox->addWidget( force );
```

在第九章，我们把 angle 放到了布局的左下单元。现在我们想在这个单元中放入两个窗口部件，所一个我们用了一个垂直的盒子，把这个垂直的盒子放到这个网格单元中，并且把 angle 和 force 放到这个垂直的盒子中。

```
    force->setValue( 25 );
```

我们初始化力量的值为 25。

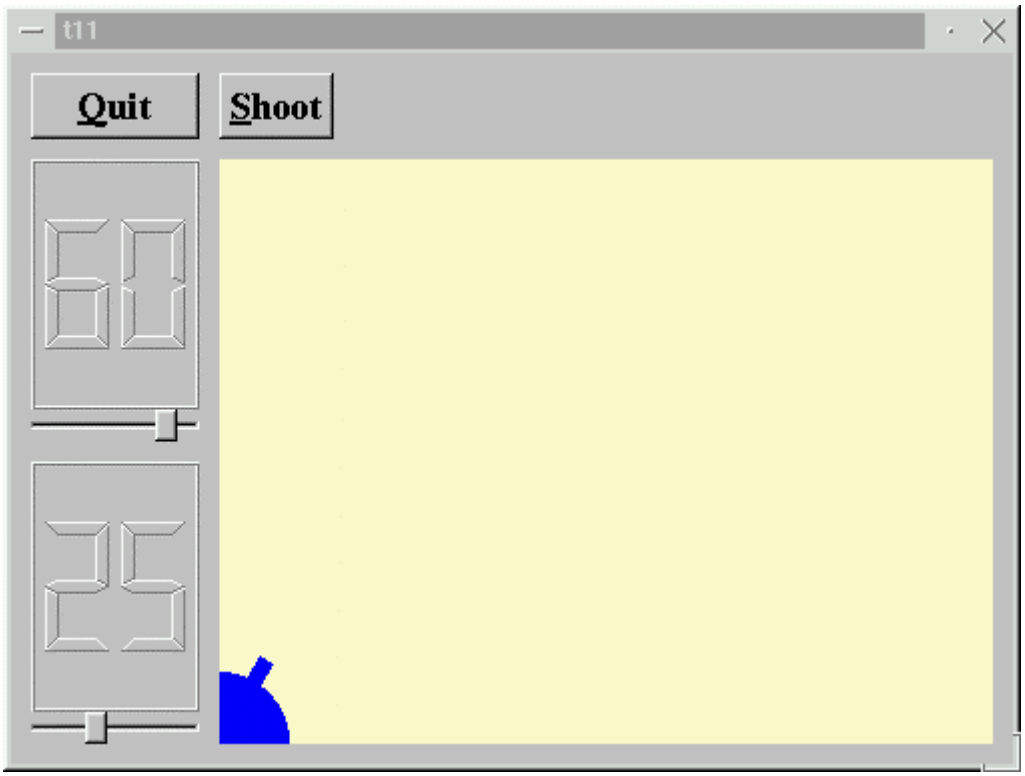
行为

闪烁已经走了，并且我们还有一个力量控制。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

让加农炮的炮筒的大小依赖于力量。
把加农炮放到右下角。
试着加入一个更好的键盘接口。例如，用+和-来增加或者减少力量，用 enter 来发射。提示：QAccel 和在 LCDRange 中新建 addStep()和 subtractStep()，就像 QSlider::addStep()。如果你被左面和右面键所苦恼（我就是！），试着都改变！
现在你可以进行第十一章了。

Qt 教程一 —— 第十一章：给它一个炮弹



在这个例子里我们介绍了一个定时器来实现动画的射击。

- t11/lcdrange.h 包含 LCDRange 类定义。
- t11/lcdrange.cpp 包含 LCDRange 类实现。
- t11/cannon.h 包含 CannonField 类定义。
- t11/cannon.cpp 包含 CannonField 类实现。
- t11/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t11/cannon.h

CannonField 现在就有了射击能力。

```
        void  shoot();
```

当炮弹不在空中中，调用这个槽就会使加农炮射击。

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

当炮弹正在空中时，这个私有槽使用一个定时器来移动射击。

```
    private:
        void  paintShot( QPainter * );
```

这个函数来画射击。

```
        QRect shotRect() const;
```

当炮弹正在空中的时候，这个私有函数返回封装它所占用空间的矩形，否则它就返回一个没有定义的矩形。

```
        int timerCount;
```

```
    QTimer * autoShootTimer;
    float shoot_ang;
    float shoot_f;
};
```

这些私有变量包含了描述射击的信息。timerCount 保留了射击进行后的时间。shoot_ang 是加农炮射击时的角度，shoot_f 是射击时加农炮的力量。

t11/cannon.cpp

```
#include <math.h>
```

我们包含了数学库，因为我们需要使用 sin()和 cos()函数。

```
CannonField::CannonField( QWidget *parent, const char *name )
        : QWidget( parent, name )
{
    ang = 45;
    f = 0;
    timerCount = 0;
    autoShootTimer = new QTimer( this, "movement handler" );
    connect( autoShootTimer, SIGNAL(timeout()),
             this, SLOT(moveShot()) );
    shoot_ang = 0;
    shoot_f = 0;
    setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

我们初始化我们新的私有变量并且把 QTimer::timeout()信号和我们的 moveShot()槽相连。我们会在定时器超时的时候移动射击。

```
void CannonField::shoot()
{
    if ( autoShootTimer->isActive() )
        return;
    timerCount = 0;
    shoot_ang = ang;
    shoot_f = f;
    autoShootTimer->start( 50 );
}
```

只要炮弹不在空中，这个函数就会进行一次射击。timerCount 被重新设置为零。shoot_ang 和 shoot_f 设置为当前加农炮的角度和力量。最后，我们开始这个定时器。

```
void CannonField::moveShot()
{
    QRegion r( shotRect() );
    timerCount++;

    QRect shotR = shotRect();
```

```
        if ( shotR.x() > width() || shotR.y() > height() )
            autoShootTimer->stop();
        else
            r = r.unite( QRegion( shotR ) );
        repaint( r );
    }
```

moveShot()是一个移动射击的槽，当 QTimer 开始的时候，每 50 毫秒被调用一次。它的任务就是计算新的位置，重新画屏幕并把炮弹放到新的位置，并且如果需要的话，停止定时器。

首先我们使用 QRegion 来保留旧的 shotRect()。QRegion 可以保留任何种类的区域，并且我们可以用它来简化绘画过程。shotRect()返回现在炮弹所在的矩形——稍后我们会详细介绍。

然后我们增加 timerCount，用它来实现炮弹在它的轨迹中移动的每一步。

下一步我们算出新的炮弹的矩形。

如果炮弹已经移动到窗口部件的右面或者下面的边界，我们停止定时器或者添加新的 shotRect()到 QRegion。

最后，我们重新绘制 QRegion。这将会发送一个单一的绘画事件，但仅仅有一个到两个举行需要刷新。

```
    void CannonField::paintEvent( QPaintEvent *e )
    {
        QRect updateR = e->rect();
        QPainter p( this );

        if ( updateR.intersects( cannonRect() ) )
            paintCannon( &p );
        if ( autoShootTimer->isActive() &&
             updateR.intersects( shotRect() ) )
            paintShot( &p );
    }
```

绘画事件函数在前一章中已经被分成两部分了。现在我们得到的新的矩形区域需要绘画，检查加农炮和/或炮弹是否相交，并且如果需要的话，调用 paintCannon()和/或 paintShot()。

```
    void CannonField::paintShot( QPainter *p )
    {
        p->setBrush( black );
        p->setPen( NoPen );
        p->drawRect( shotRect() );
    }
```

这个私有函数画一个黑色填充的矩形作为炮弹。

我们把 paintCannon()的实现放到一边，它和前一章中的 paintEvent()一样。

```
    QRect CannonField::shotRect() const
    {
        const double gravity = 4;
```

```
        double time      = timerCount / 4.0;
        double velocity  = shoot_f;
        double radians   = shoot_ang*3.14159265/180;

        double velx      = velocity*cos( radians );
        double vely      = velocity*sin( radians );
        double x0        = ( barrelRect.right()  + 5 )*cos(radians);
        double y0        = ( barrelRect.right()  + 5 )*sin(radians);
        double x         = x0 + velx*time;
        double y         = y0 + vely*time - 0.5*gravity*time*time;

        QRect r = QRect( 0, 0, 6, 6 );
        r.moveCenter( QPoint( qRound(x), height() - 1 - qRound(y) ) );
        return r;
    }
```

这个私有函数计算炮弹的中心点并且返回封装炮弹的矩形。它除了使用自动增加所过去的时间的 timerCount 之外，还使用初始时的加农炮的力量和角度。
运算公式使用的是有重力的环境下光滑运动的经典牛顿公式。简单地说，我们已经选择忽略爱因斯坦理论的结果。
我们在一个 y 坐标向上增加的坐标系统中计算中心点。在我们计算出中心点之后，我们构造一个 6*6 大小的 QRect，并把它的中心移动到我们上面所计算出的中心点。同样的操作我们把这个点移动到窗口部件的坐标系统（请看坐标系统）。
qRound()函数是一个在 qglobal.h 中定义的内嵌函数（被其它所有 Qt 头文件包含）。qRound()把一个双精度实数变为最接近的整数。

t11/main.cpp

```
    class MyWidget: public QWidget
    {
    public:
        MyWidget( QWidget *parent=0, const char *name=0 );
    };
```

唯一的增加是 Shoot 按钮。

```
        QPushButton *shoot = new QPushButton( "&Shoot", this, "shoot" );
        shoot->setFont( QFont( "Times", 18, QFont::Bold ) );
```

在构造函数中我们创建和设置 Shoot 按钮就像我们对 Quit 按钮所做的那样。注意构造函数的第一个参数是按钮的文本，并且第三个是窗口部件的名称。

```
        connect( shoot, SIGNAL(clicked()), cannonField, SLOT(shoot()) );
```

把 Shoot 按钮的 clicked()信号和 CannonField 的 shoot()槽连接起来。

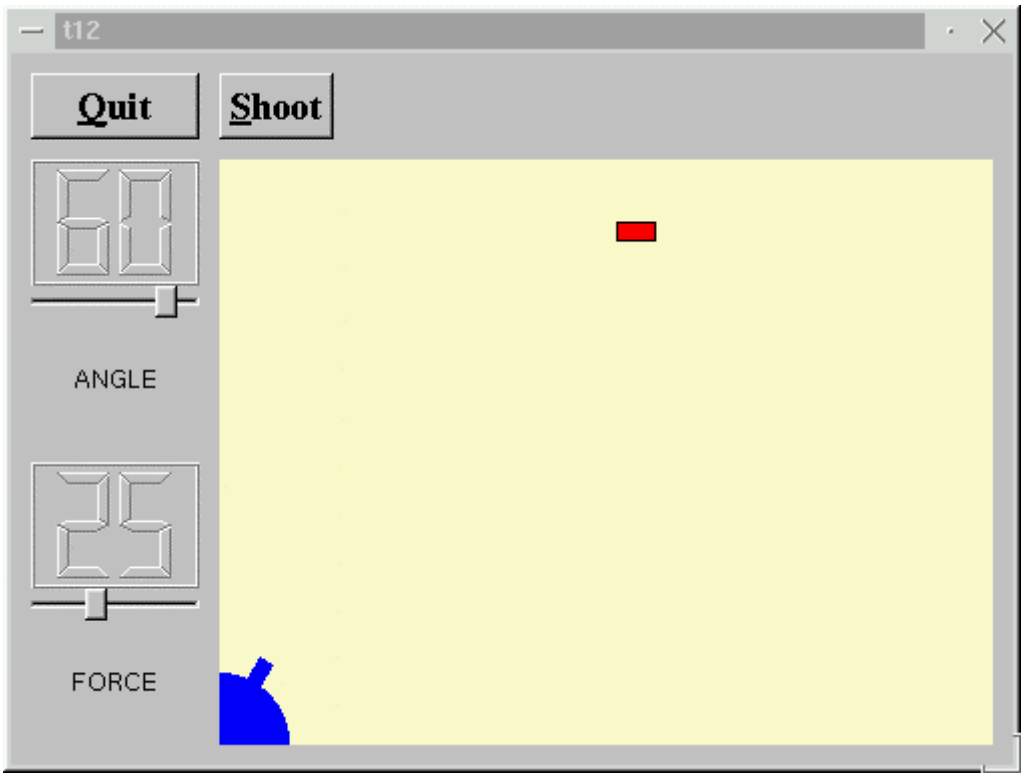
行为

The cannon can shoot, but there's nothing to shoot at.
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

用一个填充的圆来表示炮弹。提示：QPainter::drawEllipse()会对你有所帮助。
当炮弹在空中的时候，改变加农炮的颜色。
现在你可以进行第十二章了。

Qt 教程一 —— 第十一章：悬在空中的砖



在这个例子中，我们扩展我们的 LCDRange 类来包含一个文本标签。我们也会给射击提供一个目标。

- t12/lcdrange.h 包含 LCDRange 类定义。
- t12/lcdrange.cpp 包含 LCDRange 类实现。
- t12/cannon.h 包含 CannonField 类定义。
- t12/cannon.cpp 包含 CannonField 类实现。
- t12/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t12/lcdrange.h

LCDRange 现在有了一个文本标签。

```
class QLabel;
```

我们名称声明 QLabel，因为我们将在这个类声明中使用一个 QLabel 的指针。

```
class LCDRange : public QVBox
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
    LCDRange( const char *s, QWidget *parent=0,
              const char *name=0 );
```

我们添加了一个新的构造函数，这个构造函数在父对象和名称之外还设置了标签文本。

```
    const char *text()  const;
```

这个函数返回标签文本。

```
    void setText( const char * );
```

这个槽设置标签文本。

```
private:
    void init();
```

因为我们现在有了两个构造函数，我们选择把通常的初始化放在一个私有的 init()函数。

```
    QLabel      *label;
```

我们还有一个新的私有变量：一个 QLabel。QLabel 是一个 Qt 标准窗口部件并且可以显示一个有或者没有框架的文本或者 pixmap。

t12/lcdrange.cpp

```
#include <qlabel.h>
```

这里我们包含了 QLabel 类定义。

```
LCDRange::LCDRange( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    init();
}
```

这个构造函数调用了 init()函数，它包括了通常的初始化代码。

```
LCDRange::LCDRange( const char *s, QWidget *parent,
                    const char *name )
        : QVBox( parent, name )
{
    init();
    setText( s );
}
```

这个构造函数首先调用了 init()然后设置标签文本。

```
void LCDRange::init()
{
    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );
```

```
    slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );

    label = new QLabel( " ", this, "label" );
    label->setAlignment( AlignCenter );

    connect( slider, SIGNAL(valueChanged(int)),
             lcd, SLOT(display(int)) );
    connect( slider, SIGNAL(valueChanged(int)),
             SIGNAL(valueChanged(int)) );

    setFocusProxy( slider );
}
```

lcd 和 slider 的设置和上一章一样。接下来我们创建一个 QLabel 并且让它的内容中间对齐（垂直方向和水平方向都是）。connect()语句也来自于上一章。

```
const char *LCDRange::text() const
{
    return label->text();
}
```

这个函数返回标签文本。

```
void LCDRange::setText( const char *s )
{
    label->setText( s );
}
```

这个函数设置标签文本。

t12/cannon.h

CannonField 现在有两个新的信号：hit()和 missed()。另外它还包含一个目标。

```
    void  newTarget();
```

这个槽在新的位置生成一个新的目标。

```
signals:
    void  hit();
    void  missed();
```

hit()信号是当炮弹击中目标的时候被发射的。missed()信号是当炮弹移动超出了窗口部件的右面或者下面的边界时被发射的（例如，当然这种情况下它将不会击中目标）。

```
    void  paintTarget( QPainter * );
```

这个私有函数绘制目标。

```
    QRect targetRect() const;
```

这个私有函数返回一个封装了目标的矩形。

```
    QPoint target;
```

这个私有变量包含目标的中心点。

t12/cannon.cpp

```
#include <qdatetime.h>
```

我们包含了 QDate、QTime 和 QDateTime 类定义。

```
#include <stdlib.h>
```

我们包含了 stdlib 库，因为我们需要 rand()函数。

```
    newTarget();
```

这一行已经被添加到了构造函数中。它为目标创建一个“随机的”位置。实际上，newTarget()函数还试图绘制目标。因为我们在一个构造函数中，CannonField 窗口部件还是不可以见的。Qt 保证在一个隐藏的窗口部件中调用 repaint()是没有害处的。

```
void  CannonField::newTarget()
{
    static bool first_time = TRUE;
    if ( first_time ) {
        first_time = FALSE;
        QTime midnight( 0, 0, 0 );
        srand( midnight.secsTo(QTime::currentTime()) );
    }
    QRegion r( targetRect() );
    target = QPoint( 200 + rand() % 190,
                     10  + rand() % 255 );
    repaint( r.unite( targetRect() ) );
}
```

这个私有函数创建了一个在新的“随机的”位置的目标中心点。

我们使用 rand()函数来获得随机整数。rand()函数通常会在你每次运行这个程序的时候返回同样一组值。这就会使每次运行的时候目标都出现在同样的位置。为了避免这些，我们必须在这个函数第一次被调用的时候设置一个随机种子。为了避免同样一组数据，随机种子也必须是随机的。解决方法就是使用从午夜到现在的秒数作为一个假的随机值。

首先我们创建一个静态布尔型局域变量。静态变量就是在调用函数前后都保证它的值不变。

if 测试会成功，因为只有当这个函数第一次被调用的时候，我们在 if 块中把 first_time 设置为 FALSE。

然后我们创建一个 QTime 对象 midnight，它将会提供时间 00:00:00。接下来我们获得从午夜到现在所过的秒数并且使用它作为一个随机种子。请看 QDate、QTime 和 QDateTime 文档来获得更多的信息。

最后我们计算目标的中心点。我们把它放在一个矩形中（x=200，y=35，width=190，height=255），（例如，可能的 x 和 y 值是 x=200~389 和 y=35~289）在一个我们把窗口边界的下边界作为 y 的零点，并且 y 向上增加，X 轴向通常一样，左边界为零点，并且 x 向右增加的坐标系统中。

通过经验，我们发现这都在炮弹的射程之内。

注意 rand()返回一个>=0 的随机整数。

```
void CannonField::moveShot()
```

```
    {
        QRegion r( shotRect() );
        timerCount++;

        QRect shotR = shotRect();
```

定时器时间这部分和上一章一样。

```
        if ( shotR.intersects( targetRect() ) ) {
            autoShootTimer->stop();
            emit hit();
```

if 语句检查炮弹矩形和目标矩形是否相交。如果是的，炮弹击中了目标（哎哟！）。我们停止射击定时器并且发射 hit()信号来告诉外界目标已经被破坏，并返回。注意，我们可以在这个点上创建一个新的目标，但是因为 CannonField 是一个组件，所以我们要把这样的决定留给组件的使用者。

```
        } else if ( shotR.x() > width() || shotR.y() > height() ) {
            autoShootTimer->stop();
            emit missed();
```

这个 if 语句和上一章一样，除了现在它发射 missed()信号告诉外界这次失败。

```
        } else {
```

函数的其余部分和以前一样。

CannonField::paintEvent() is as before, except that this has been added:

```
        if ( updateR.intersects( targetRect() ) )
            paintTarget( &p );
```

这两行确认在需要的时候目标也被绘制。

```
    void CannonField::paintTarget( QPainter *p )
    {
        p->setBrush( red );
        p->setPen( black );
        p->drawRect( targetRect() );
    }
```

这个私有函数绘制目标，一个由红色填充，有黑色边框的矩形。

```
    QRect CannonField::targetRect() const
    {
        QRect r( 0, 0, 20, 10 );
        r.moveCenter( QPoint(target.x(),height() - 1 - target.y()) );
        return r;
    }
```

这个私有函数返回封装目标的矩形。从 newTarget()中所得的 target 点使用 0 点在窗口部件的下边界的 y。我们在调用 QRect::moveCenter()之前在窗口坐标中计算这个点。

我们选择这个坐标映射的原因是在目标和窗口部件的下边界之间垂直距离。记住这些可以让用户或者程序在任何时候都可以重新定义窗口部件的大小。

t12/main.cpp

在 MyWidget 类中没有新的成员了，但是我们稍微改变了一下构造函数来设置新的 LCDRange 的文本标签。

```
LCDRange *angle  = new LCDRange( "ANGLE", this, "angle" );
```

我们设置角度的文本标签为“ANGLE”。

```
LCDRange *force  = new LCDRange( "FORCE", this, "force" );
```

我们设置力量的文本标签为“FORCE”。

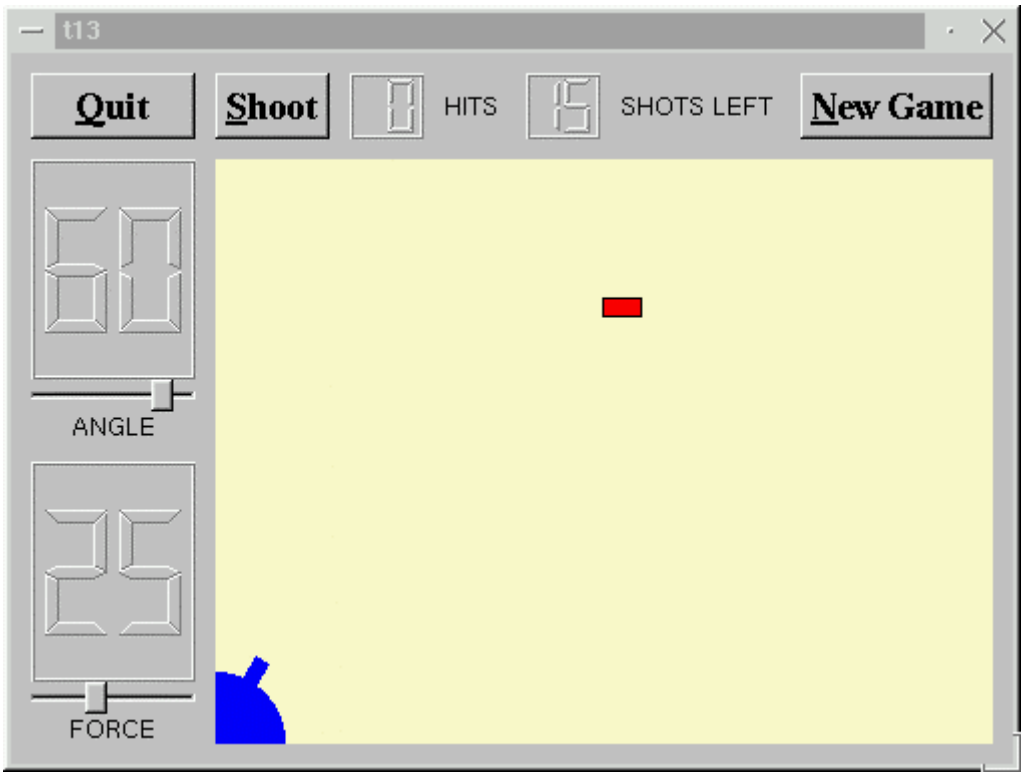
行为

加农炮会向目标射击，当它射中目标的时候，一个新的目标会自动被创建。
LCDRange 窗口部件看起来有一点奇怪——QVBox 中内置的管理给了标签太多的空间而其它的却不够。我们将会在下一章修正这一点。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

创建一个作弊的按钮，当按下它的时候，让 CannonField 画出炮弹在五秒中的轨迹。
如果你在上一章做了“圆形炮弹”的练习，试着改变 shotRect()为可以返回一个 QRegion 的 shotRegion()，这样你就可以真正的做到准确碰撞检测。
做一个移动目标。
确认目标被完全创建在屏幕上。
确认加农炮窗口部件不能被重新定义大小，这样目标不是可见的。提示：QWidget::setMinimumSize()是你的朋友。
不容易的是在同一时刻让几个炮弹在空中成为可能。提示：建立一个炮弹对象。
现在你可以进行第十三章了。

Qt 教程一 —— 第十三章：游戏结束



在这个例子中我们开始研究一个带有记分的真正可玩的游戏。我们给 MyWidget 一个新的名字（GameBoard）并添加一些槽。
我们把定义放在 gamebrd.h 并把实现放在 gamebrd.cpp。
CannonField 现在有了一个游戏结束状态。
在 LCDRange 中的布局问题已经修好了。

- t13/lcdrange.h 包含 LCDRange 类定义。
- t13/lcdrange.cpp 包含 LCDRange 类实现。
- t13/cannon.h 包含 CannonField 类定义。
- t13/cannon.cpp 包含 CannonField 类实现。
- t13/gamebrd.h 包含 GameBoard 类定义。
- t13/gamebrd.cpp 包含 GameBoard 类实现。
- t13/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t13/lcdrange.h

```
#include <qwidget.h>

class QSlider;
class QLabel;

class LCDRange : public QWidget
```

我们继承了 QWidget 而不是 QVBox。QVBox 是非常容易使用的，但是它也显示了它的局域性，所以我们选择使用更加强大和稍微有一些难的 QVBoxLayout。（和你记忆中的一样，QVBoxLayout 不是一个窗口部件，它管理窗口部件。）

t13/lcdrange.cpp

```
#include <qlayout.h>
```

我们现在需要包含 qlayout.h 来获得其它布局管理 API。

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

我们使用一种平常的方式继承 QWidget。

另外一个构造函数作了同样的改动。init()没有变化，除了我们在最后加了几行：

```
    QVBoxLayout * l = new QVBoxLayout( this );
```

我们使用所有默认值创建一个 QVBoxLayout，管理这个窗口部件的子窗口部件。

```
    l->addWidget( lcd, 1 );
```

At the top we add the QLCDNumber with a non-zero stretch.

```
    l->addWidget( slider );
    l->addWidget( label );
```

然后我们添加另外两个，它们都使用默认的零伸展因数。

这个伸展控制是 QVBoxLayout（和 QHBoxLayout，和 QGridLayout）所提供的，而像 QVBox 这样的类却不提供。在这种情况下我们让 QLCDNumber 可以伸展，而其它的不可以。

t13/cannon.h

CannonField 现在有一个游戏结束状态和一些新的函数。

```
    bool  gameOver() const { return gameEnded; }
```

如果游戏结束了，这个函数返回 TRUE，或者如果游戏还在继续，返回 FALSE。

```
    void  setGameOver();
    void  restartGame();
```

这里是两个新槽：setGameOver()和 restartGame()。

```
    void  canShoot( bool );
```

这个新的信号表明 CannonField 使 shoot()槽生效的状态。我们将在下面使用它用来使 Shoot 按钮生效或失效。

```
    bool gameEnded;
```

这个私有变量包含游戏的状态。TRUE 说明游戏结束，FALSE 说明游戏还将继续。

t13/cannon.cpp

```
    gameEnded = FALSE;
```

这一行已经被加入到构造函数中。最开始的时候，游戏没有结束（对于玩家是很幸运的 :-）。

```
void CannonField::shoot()
{
```

```
    if ( isShooting() )
        return;
    timerCount = 0;
    shoot_ang = ang;
    shoot_f = f;
    autoShootTimer->start( 50 );
    emit canShoot( FALSE );
}
```

我们添加一个新的 isShooting()函数，所以 shoot()使用它替代直接的测试。同样，shoot 告诉世界 CannonField 现在不可以射击。

```
void CannonField::setGameOver()
{
    if ( gameEnded )
        return;
    if ( isShooting() )
        autoShootTimer->stop();
    gameEnded = TRUE;
    repaint();
}
```

这个槽终止游戏。它必须被 CannonField 外面的调用，因为这个窗口部件不知道什么时候终止游戏。这是组件编程中一条重要设计原则。我们选择使组件可以尽可能灵活以适应不同的规则（比如，在一个首先射中十次的人胜利的多人游戏版本可能使用不变的 CannonField）。

如果游戏已经被终止，我们立即返回。如果游戏会继续到我们的设计完成，设置游戏结束标志，并且重新绘制整个窗口部件。

```
void CannonField::restartGame()
{
    if ( isShooting() )
        autoShootTimer->stop();
    gameEnded = FALSE;
    repaint();
    emit canShoot( TRUE );
}
```

这个槽开始一个新游戏。如果炮弹还在空中，我们停止设计。然后我们重置 gameEnded 变量并重新绘制窗口部件。

就像 hit()或 miss()一样，moveShot()同时也发射新的 canShoot(TRUE)信号。

CannonField::paintEvent()的修改：

```
void CannonField::paintEvent( QPaintEvent *e )
{
    QRect updateR = e->rect();
    QPainter p( this );

    if ( gameEnded ) {
        p.setPen( black );
```

```
            p.setFont( QFont( "Courier", 48, QFont::Bold ) );
            p.drawText( rect(), AlignCenter, "Game Over" );
        }
```

绘画事件已经通过如果游戏结束，比如 gameEnded 是 TRUE，就显示文本“Game Over”而被增强了。我们在这里不怕麻烦来检查更新矩形，是因为在游戏结束的时候速度不是关键性的。

为了画文本，我们先设置了黑色的画笔，当画文本的时候，画笔颜色会被用到。接下来我们选择 Courier 字体中的 48 号加粗字体。最后我们在窗口部件的矩形中央绘制文本。不幸的是，在一些系统中（特别是使用 Unicode 的 X 服务器）它会用一小段时间来载入如此大的字体。因为 Qt 缓存字体，我们只有第一次使用这个字体的时候才会注意到这一点。

```
        if ( updateR.intersects( cannonRect() ) )
            paintCannon( &p );
        if ( isShooting() && updateR.intersects( shotRect() ) )
            paintShot( &p );
        if ( !gameEnded && updateR.intersects( targetRect() ) )
            paintTarget( &p );
    }
```

我们只有在设计的时候画炮弹，在玩游戏的时候画目标（这也就是说，当游戏没有结束的时候）。

t13/gamebrd.h

这个文件是新的。它包含最后被用来作为 MyWidget 的 GameBoard 类的定义。

```
    class QPushButton;
    class LCDRange;
    class QLCDNumber;
    class CannonField;

    #include "lcdrange.h"
    #include "cannon.h"

    class GameBoard : public QWidget
    {
        Q_OBJECT
    public:
        GameBoard( QWidget *parent=0, const char *name=0 );

    protected slots:
        void  fire();
        void  hit();
        void  missed();
        void  newGame();

    private:
```

```
    QLCDNumber  *hits;
    QLCDNumber  *shotsLeft;
    CannonField *cannonField;
};
```

我们现在已经添加了四个槽。这些槽都是被保护的，只在内部使用。我们也已经加入了两个 QLCDNumbers（hits 和 shotsLeft）用来显示游戏的状态。

t13/gamebrd.cpp

这个文件是新的。它包含最后被用来作为 MyWidget 的 GameBoard 类的实现，我们已经在 GameBoard 的构造函数中做了一些修改。

```
    cannonField = new CannonField( this, "cannonField" );
```

cannonField 现在是一个成员变量，所以我们在使用它的时候要小心地改变它的构造函数。（Trolltech 的*好*程序员从来不会忘记这点，但是我就忘了。告诫程序员一如果“programmor”是拉丁语，至少。无论如何，返回代码。）

```
    connect( cannonField, SIGNAL(hit()),
             this, SLOT(hit()) );
    connect( cannonField, SIGNAL(missed()),
             this, SLOT(missed()) );
```

这次当炮弹射中或者射失目标的时候，我们想做些事情。所以我们把 CannonField 的 hit()和 missed()信号连接到这个类的两个被保护的同名槽。

```
    connect( shoot, SIGNAL(clicked()), SLOT(fire()) );
```

以前我们直接把 Shoot 按钮的 clicked()信号连接到 CannonField 的 shoot()槽。这次我们想跟踪射击的次数，所以我们把它改为连接到这个类里面一个被保护的槽。

注意当你用独立的组件工作的时候，改变程序的行为是多么的容易。

```
    connect( cannonField, SIGNAL(canShoot(bool)),
             shoot, SLOT(setEnabled(bool)) );
```

我们也使用 cannonField 的 canShoot()信号来适当地使 Shoot 按钮生效和失效。

```
    QPushButton *restart
        = new QPushButton( "&New Game", this, "newgame" );
    restart->setFont( QFont( "Times", 18, QFont::Bold ) );


    connect( restart, SIGNAL(clicked()), this, SLOT(newGame()) );
```

我们创建、设置并且连接这个 New Game 按钮就像我们对其它按钮所做的一样。点击这个按钮就会激活这个窗口部件的 newGame()槽。

```
    hits = new QLCDNumber( 2, this, "hits" );
    shotsLeft = new QLCDNumber( 2, this, "shotsleft" );
    QLabel *hitsL = new QLabel( "HITS", this, "hitsLabel" );
    QLabel *shotsLeftL
        = new QLabel( "SHOTS LEFT", this, "shotsleftLabel" );
```

我们创建了四个新的窗口部件。注意我们不怕麻烦的把 QLabel 窗口部件的指针保留到 GameBoard 类中是因为我们不想再对它们做什么了。当 GameBoard 窗口

部件被销毁的时候，Qt 将会删除它们，并且布局类会适当地重新定义它们的大小。

```
        QHBoxLayout *topBox = new QHBoxLayout;
        grid->addLayout( topBox, 0, 1 );
        topBox->addWidget( shoot );
        topBox->addWidget( hits );
        topBox->addWidget( hitsL );
        topBox->addWidget( shotsLeft );
        topBox->addWidget( shotsLeftL );
        topBox->addStretch( 1 );
        topBox->addWidget( restart );
```

右上单元格的窗口部件的数量正在变大。从前它是空的，现在它是完全充足的，我们把它们放到布局中来更好的看到它们。

注意我们让所有的窗口部件获得它们更喜欢的大小，改为在 New Game 按钮的左边加入了一个可以自由伸展的东西。

```
        newGame();
    }
```

我们已经做完了所有关于 GameBoard 的构造，所以我们使用 newGame()来开始。（newGame()是一个槽，但是就像我们所说的，槽也可以像普通的函数一样使用。）

```
    void GameBoard::fire()
    {
        if ( cannonField->gameOver() || cannonField->isShooting() )
            return;
        shotsLeft->display( shotsLeft->intValue() - 1 );
        cannonField->shoot();
    }
```

这个函数进行射击。如果游戏结束了或者还有一个炮弹在空中，我们立即返回。我们减少炮弹的数量并告诉加农炮进行射击。

```
    void GameBoard::hit()
    {
        hits->display( hits->intValue() + 1 );
        if ( shotsLeft->intValue() == 0 )
            cannonField->setGameOver();
        else
            cannonField->newTarget();
    }
```

当炮弹击中目标的时候这个槽被激活。我们增加射中的数量。如果没有炮弹了，游戏就结束了。否则，我们会让 CannonField 生成新的目标。

```
    void GameBoard::missed()
    {
        if ( shotsLeft->intValue() == 0 )
            cannonField->setGameOver();
    }
```

当炮弹射失目标的时候这个槽被激活，如果没有炮弹了，游戏就结束了。

```
void GameBoard::newGame()
{
    shotsLeft->display( 15 );
    hits->display( 0 );
    cannonField->restartGame();
    cannonField->newTarget();
}
```

当用户点击 Restart 按钮的时候这个槽被激活。它也会被构造函数调用。首先它把炮弹的数量设置为 15。注意这里是我们在程序中唯一设置炮弹数量的地方。把它改变为你所想要的游戏规则。接下来我们重置射中的数量，重新开始游戏，并且生成一个新的目标。

t13/main.cpp

这个文件仅仅被删掉了一部分。MyWidget 没了，并且唯一剩下的是 main()函数，除了名称的改变其它都没有改变。

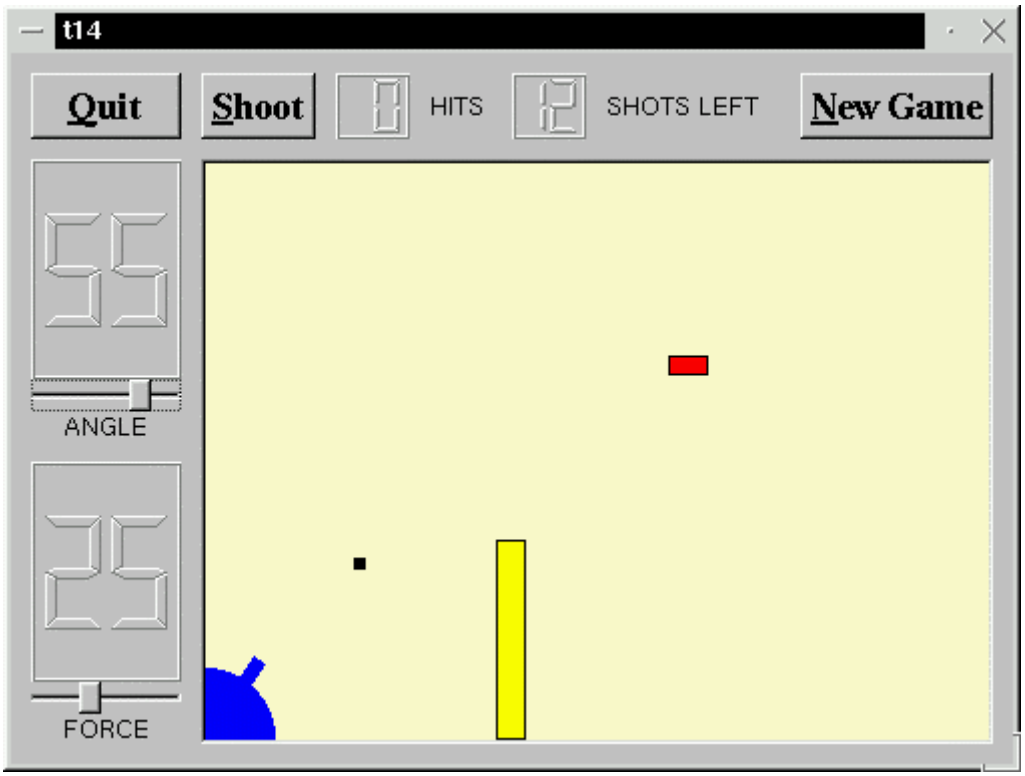
行为

射中的和剩余炮弹的数量被显示并且程序继续跟踪它们。游戏可以结束了，并且还有一个按钮可以开始一个新游戏。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

添加一个随机的风的因素并把它显示给用户看。
当炮弹击中目标的时候做一些飞溅的效果。
实现多个目标。
现在你可以进行第十四章了。

Qt 教程一 —— 第十四章：面对墙壁



这是最后的例子：一个完整的游戏。
我们添加键盘快捷键并引入鼠标事件到 CannonField。我们在 CannonField 周围放一个框架并添加一个障碍物（墙）使这个游戏更富有挑战性。

- t14/lcdrange.h 包含 LCDRange 类定义。
- t14/lcdrange.cpp 包含 LCDRange 类实现。
- t14/cannon.h 包含 CannonField 类定义。
- t14/cannon.cpp 包含 CannonField 类实现。
- t14/gamebrd.h 包含 GameBoard 类定义。
- t14/gamebrd.cpp 包含 GameBoard 类实现。
- t14/main.cpp 包含 MyWidget 和 main。

一行一行地解说

t14/cannon.h

CannonField 现在可以接收鼠标事件，使得用户可以通过点击和拖拽炮筒来瞄准。CannonField 也有一个障碍物的墙。

```
protected:
    void  paintEvent( QPaintEvent * );
    void  mousePressEvent( QMouseEvent * );
    void  mouseMoveEvent( QMouseEvent * );
    void  mouseReleaseEvent( QMouseEvent * );
```

除了常见的事件处理器，CannonField 实现了三个鼠标事件处理器。名称说明了一切。

```
        void  paintBarrier( QPainter * );
```
这个私有函数绘制了障碍物墙。
```
        QRect barrierRect() const;
```
这个私有寒暑返回封装障碍物的矩形。
```
        bool  barrelHit( const QPoint & ) const;
```
这个私有函数检查是否一个点在加农炮炮筒的内部。
```
        bool barrelPressed;
```
当用户在炮筒上点击鼠标并且没有放开的话，这个私有变量为 TRUE。

t14/cannon.cpp

```
        barrelPressed = FALSE;
```
这一行被添加到构造函数中。最开始的时候，鼠标没有在炮筒上点击。
```
        } else if ( shotR.x() > width() || shotR.y() > height() ||
                    shotR.intersects(barrierRect()) ) {
```
现在我们有了一个障碍物，这样就有了三种射失的方法。我们来测试一下第三种。
```
    void CannonField::mousePressEvent( QMouseEvent *e )
    {
        if ( e->button() != LeftButton )
            return;
        if ( barrelHit( e->pos() ) )
            barrelPressed = TRUE;
    }
```
这是一个 Qt 事件处理器。当鼠标指针在窗口部件上，用户按下鼠标的按键时，它被调用。
如果事件不是由鼠标左键产生的，我们立即返回。否则，我们检查鼠标指针是否在加农炮的炮筒内。如果是的，我们设置 barrelPressed 为 TRUE。
注意 pos()函数返回的是窗口部件坐标系统中的点。
```
    void CannonField::mouseMoveEvent( QMouseEvent *e )
    {
        if ( !barrelPressed )
            return;
        QPoint pnt = e->pos();
        if ( pnt.x() <= 0 )
            pnt.setX( 1 );
        if ( pnt.y() >= height() )
            pnt.setY( height() - 1 );
        double rad = atan(((double)rect().bottom()-pnt.y())/pnt.x());
        setAngle( qRound ( rad*180/3.14159265 ) );
    }
```
这是另外一个 Qt 事件处理器。当用户已经在窗口部件中按下了鼠标按键并且移动/拖拽鼠标时，它被调用。（你可以让 Qt 在没有鼠标按键被按下的时候发送鼠标移动事件。请看 QWidget::setMouseTracking()。）
这个处理器根据鼠标指针的位置重新配置加农炮的炮筒。

首先，如果炮筒没有被按下，我们返回。接下来，我们获得鼠标指针的位置。如果鼠标指针到了窗口部件的左面或者下面，我们调整鼠标指针使它返回到窗口部件中。
然后我们计算在鼠标指针和窗口部件的左下角所构成的虚构的线和窗口部件下边界的角度。最后，我们把加农炮的角度设置为我们新算出来的角度。
记住要用 setAngle()来重新绘制加农炮。

```
void CannonField::mouseReleaseEvent( QMouseEvent *e )
{
    if ( e->button() == LeftButton )
        barrelPressed = FALSE;
}
```

只要用户释放鼠标按钮并且它是在窗口部件中按下的时候，这个 Qt 事件处理器就会被调用。
如果鼠标左键被释放，我们就会确认炮筒不再被按下了。
绘画事件包含了下述额外的两行：

```
    if ( updateR.intersects( barrierRect() ) )
        paintBarrier( &p );
```

paintBarrier()做的和 paintShot()、paintTarget()和 paintCannon()是同样的事情。

```
void CannonField::paintBarrier( QPainter *p )
{
    p->setBrush( yellow );
    p->setPen( black );
    p->drawRect( barrierRect() );
}
```

这个私有函数用一个黑色边界黄色填充的矩形作为障碍物。

```
QRect CannonField::barrierRect() const
{
    return QRect( 145, height() - 100, 15, 100 );
}
```

这个私有函数返回障碍物的矩形。我们把障碍物的下边界和窗口部件的下边界放在了一起。

```
bool CannonField::barrelHit( const QPoint &p ) const
{
    QWMatrix mtx;
    mtx.translate( 0, height() - 1 );
    mtx.rotate( -ang );
    mtx = mtx.invert();
    return barrelRect.contains( mtx.map(p) );
}
```

如果点在炮筒内，这个函数返回 TRUE；否则它就返回 FALSE。
这里我们使用 QWMatrix 类。它是在头文件 qwmatrix.h 中定义的，这个头文件被 qpainter.h 包含。
QWMatrix 定义了一个坐标系统映射。它可以执行和 QPainter 中一样的转换。

这里我们实现同样的转换的步骤就和我们在 paintCannon()函数中绘制炮筒的时候所作的一样。首先我们转换坐标系统，然后我们旋转它。
现在我们需要检查点 p（在窗口部件坐标系统中）是否在炮筒内。为了做到这一点，我们倒置这个转换矩阵。倒置的矩阵就执行了我们在绘制炮筒时使用的倒置的转换。我们通过使用倒置矩阵来映射点 p，并且如果它在初始的炮筒矩形内就返回 TRUE。

t14/gamebrd.cpp

```
#include <qaccel.h>
```

我们包含 QAccel 的类定义。

```
QVBox *box = new QVBox( this, "cannonFrame" );
box->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
cannonField = new CannonField( box, "cannonField" );
```

我们创建并设置一个 QVBox，设置它的框架风格，并在之后创建 CannonField 作为这个盒子的子对象。因为没有其它的东西在这个盒子里了，效果就是 QVBox 会在 CannonField 周围生成了一个框架。

```
QAccel *accel = new QAccel( this );
accel->connectItem( accel->insertItem( Key_Enter ),
                    this, SLOT(fire()) );
accel->connectItem( accel->insertItem( Key_Return ),
                    this, SLOT(fire()) );
```

现在我们创建并设置一个加速键。加速键就是在应用程序中截取键盘事件并且如果特定的键被按下的时候调用相应的槽。这种机制也被称为快捷键。注意快捷键是窗口部件的子对象并且当窗口部件被销毁的时候销毁。QAccel *不是*窗口部件，并且在它的父对象中没有任何可见的效果。
我们定义两个快捷键。我们希望在 Enter 键被按下的时候调用 fire()槽，在 Ctrl+Q 键被按下的时候，应用程序退出。因为 Enter 有时又被称为 Return，并且有时键盘中*两个*键都有，所以我们让这两个键都调用 fire()。

```
accel->connectItem( accel->insertItem( CTRL+Key_Q ),
                    qApp, SLOT(quit()) );
```

并且之后我们设置 Ctrl+Q 和 Alt+Q 做同样的事情。一些人通常使用 Ctrl+Q 更多一些（并且无论如何它显示了如果做到它）。
CTRL、Key_Enter、Key_Return 和 Key_Q 都是 Qt 提供的常量。它们实际上就是 Qt::Key_Enter 等等，但是实际上所有的类都继承了 Qt 这个命名空间类。

```
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
grid->addWidget( quit, 0, 0 );
grid->addWidget( box, 1, 1 );
grid->setColStretch( 1, 10 );
```

我们放置 box（QVBox），不是 CannonField，在右下的单元格中。

行为

现在当你按下 Enter 的时候，加农炮就会发射。你也可以用鼠标来确定加农炮的角度。障碍物会使你在玩游戏的时候获得更多一点的挑战。我们还会在 CannnonField 周围看到一个好看的框架。
（请看编译来学习如何创建一个 makefile 和连编应用程序。）

练习

写一个空间入侵者的游戏。
（这个练习首先被 Igor Rafienko 作出来了。你可以下载他的游戏。）
新的练习是：写一个突围游戏。
最后的劝告：现在向前进，创造*编程艺术的杰作*!

Qt 教程二

这个教程会提供一个比第一个教程更加“真实世界”的 Qt 编程实例。它介绍了 Qt 编程的许多方面，介绍了创建菜单（包括最近使用文件列表）、工具条和对话框、载入和保存用户设置，等等。
如果你对 Qt 很陌生，如果你还没有阅读过如何学习 Qt，请阅读一下。

介绍

在这个教程中，我们将会开发一个叫做 `chart` 的单一应用程序，它根据用户输入的数据来显示简单的饼形和条形图表。
这个教程提供了一个应用程序开发的概述，包含了一些代码片断和与之相配的解释。应用程序完整的源程序在 `examples/chart`。

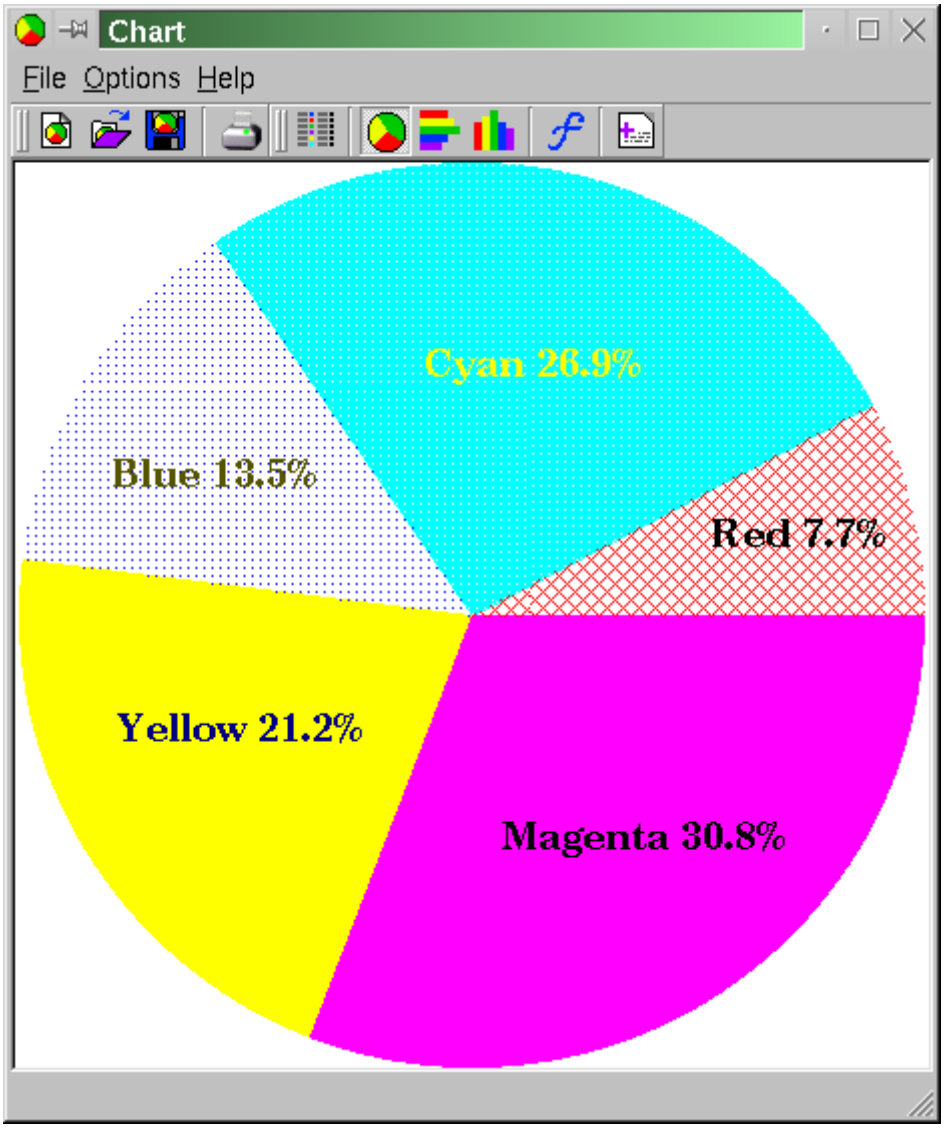


Chart
File Options Help
Cyan 26.9%
Blue 13.5%
Red 7.7%
Yellow 21.2%
Magenta 30.8%

« 目录 | “大图片”

“大图片”

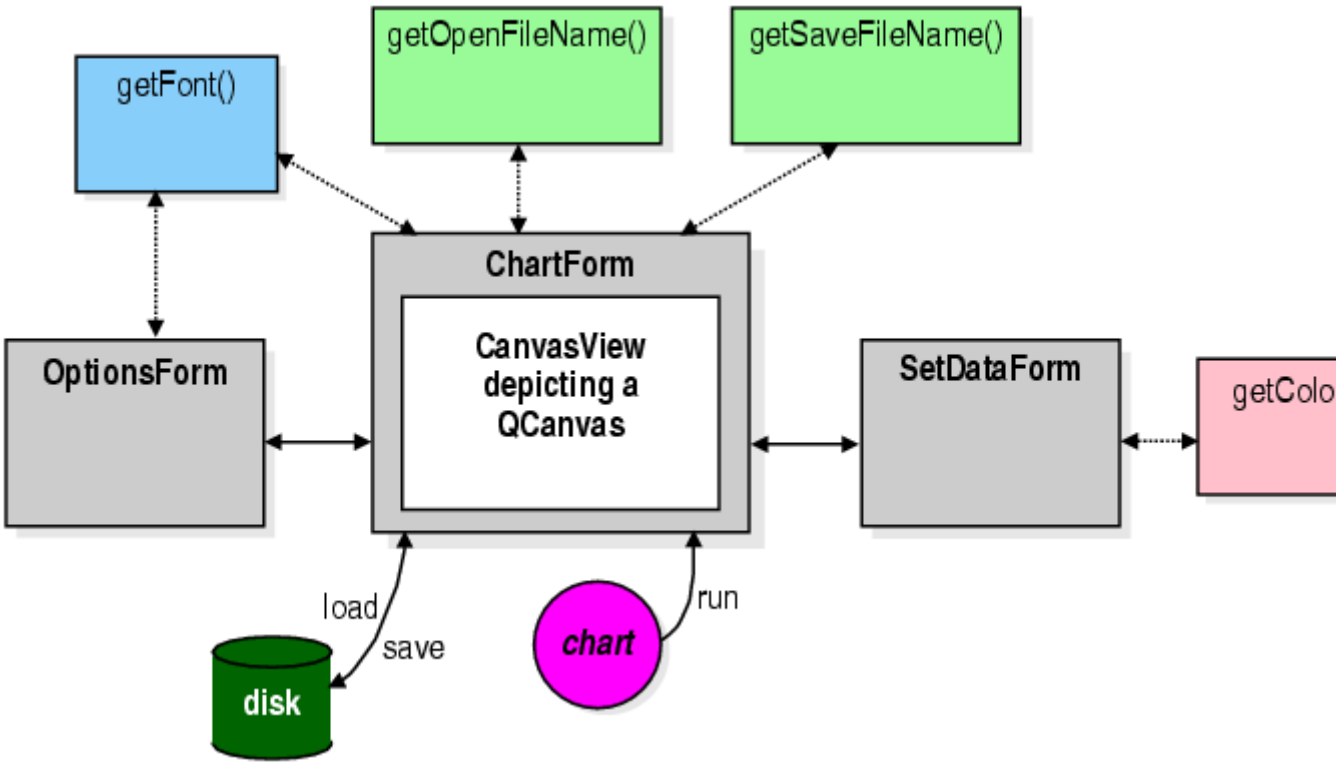


chart 程序允许用户创建、保存、载入和直观化简单的数据组。每一个用户给出的数据元素都可以被给定颜色和饼形块或条形的样式、一些标签文本和文本的位置和颜色。Element 类用来代表数据元素。

程序包含一个调入图表视窗的简单的 main.cpp。这个图表视窗有一个提供访问程序功能的菜单条和工具条。程序还提供了两个对话框，一个设置选项，另一个用来创建和编辑数据组。这两个对话框都是由图表视窗的菜单选项或者工具条按钮调用的。

图表视窗的主窗口部件是 QCanvasView，它显示一个我们用来画饼形图或条形图的 QCanvas。我们继承 QCanvasView 来获得一些特定的行为。同样我们因为需要比标准类提供的稍多一些，所以我们继承了 QCanvasText 类（用来在画布上放置文本条目）。

项目文件，chart.pro，用来创建可以用来连编应用程序的 Makefile。

« 介绍 | 目录 | 数据元素 »

数据元素

我们将使用一个叫 Element 的类来存储和访问数据元素。

（由 element.h 展开。）

```
private:
    double m_value;
    QColor m_valueColor;
    int m_valuePattern;
    QString m_label;
    QColor m_labelColor;
```

```
        double m_propoints[2 * MAX_PROPOINTS];
```

每一个元素都有一个值。每一个值都会被使用一种特定的颜色和填充样式来图形化地显示。值也许会有一个和它们关联的标签，标签会被使用标签的颜色来画，并且对于每一种类型的图表都有一个存储在 m_propoints 数组中的一个（相对）位置。

```
    #include <qcolor.h>
    #include <qnamespace.h>
    #include <qstring.h>
    #include <qvaluevector.h>
```

尽管 Element 是一个纯粹的内部数据类，它包含了四个 Qt 类。Qt 经常被认为是一个纯粹的图形用户界面工具包，但它也提供了一些用来支持应用程序编程的绝大多数方面的非图形用户界面类。我们使用 qcolor.h 可以使我们在 Element 类中控制绘图颜色和文本颜色。qnamespace.h 的用处稍微有些模糊。绝大多数 Qt 类都继承于含有各种各样的枚举的 Qt 这个超级类。Element 类不继承于 Qt，所以我们需要包含 qnamespace.h 来访问这些 Qt 枚举名称。另一个替代的方案就是使 Element 成为 Qt 的一个子类。我们包含 qstring.h 用来使用 Qt 的 Unicode 字符串。为了方便，我们类型定义一个 Element 的矢量容器，这就是我们为什么把 qvaluevector.h 头文件放到这里的原因。

```
    typedef QValueVector<Element> ElementVector;
```

Qt 提供了大量的容器，一些是基于值的，比如 QValueVector，其它一些基于指针。（请看集合类。）这里我们只是类型定义了一个容器类型，我们将在 ElementVector 中保存每一个元素数据组。

```
    const double EPSILON = 0.0000001; // 必须 > INVALID。
```

元素也许只能是正的值。因为我们使用双精度实数来存储值，我们不能很容易地拿它们和零作比较。所以我们指定一个值，EPSILON，它和零非常接近，并且任何比 EPSILON 大的值可以被认为是正的和有效的。

```
    class Element
    {
    public:
        enum { INVALID = -1 };
        enum { NO_PROPORTION = -1 };
        enum { MAX_PROPOINTS = 3 }; // 每个图表类型一个比例值
```

我们给 Element 定义了三个公有的枚举变量。INVALID 被 isValid()函数使用。它是有用的，因为我们将用使用一个固定大小的 Element 矢量，并且可以通过给定 INVALID 值来标明未使用的 Element。NO_PROPORTION 枚举变量用来表明用户还没有定位元素的标签，任何正的比例值都被用来作为与画布大小成比例的文本元素的位置。

如果我们存储每一个标签的实际 x 和 y 的位置，当用户每次重新定义主窗口大小（同样也对画布）的时候，文本会保留它的初始（现在是错的）位置。所以我们不存储绝对(x,y)位置，我们存储*比例*位置，比如 x/width 和 y/height。然后当我们画文本的时候，我们分别用当前的宽度和高度来乘这些位置，这样不管大小如何变化，文本都会被正确定位。比如，如果标签的 x 位置为 300，画布有 400 像素宽，x 的比例值为 300/400 = 0.75。

MAX_PROPOINTS 枚举变量是有些疑问的。我们对于每一个图表类型的文本标签都要存储 x 和 y 的比例。并且我们已经选择把这些比例存储到一个固定大小的数组中。因为我们必须指定所需要的比例对的最大数字。如果我们改变图表类型的数字，这个值就必须被改变，这也就是说 Element 类和由 ChartForm 所提供的图表类型的数字是紧密联系的。在一个更大的应用程序中，我们也许使用一个矢量来存储这些点并且根据所能提供的图表类型的数量来动态改变它的大小。

```
Element( double value = INVALID, QColor valueColor = Qt::gray,
         int valuePattern = Qt::SolidPattern,
         const QString& label = QString::null,
         QColor labelColor = Qt::black ) {
    init( value, valueColor, valuePattern, label, labelColor );
    for ( int i = 0; i < MAX_PROPOINTS * 2; ++i )
        m_propoints[i] = NO_PROPORTION;
}
```

构造函数为 Element 类的所有成员变量提供了默认值。新的元素总是有没有位置的标签文本。我们是用 init()函数是因为我们也提供了一个 set()函数，除了比例位置它做的和构造函数一样。

```
bool isValid() const { return m_value > EPSILON; }
```

因为我们正在把 Element 存储到一个固定大小的矢量中，所以我们需要检测一个特定元素是否有效（比如应该被用来计算和显示）。通过 isValid()函数很容易到达这一目的。

（由 element.cpp 展开。）

```
double Element::proX( int index ) const
{
    Q_ASSERT(index >= 0 && index < MAX_PROPOINTS);
    return m_propoints[2 * index];
}
```

这里对 Element 的所有成员都提供了读取函数和设置函数。proX()和 proY()读取函数和 setProX()和 setProY()设置函数用来读取和设置一个用来确定比例位置所适用的图表的类型索引。这也就是说用户可以为用于竖直条图表、水平条图表和饼形图表的相同数据组设定不同的标签位置。注意我们也使用 Q_ASSERT 宏来提供对图表类型索引的预先情况测试，（请看调试）。

读写数据元素

（由 element.h 展开。）

```
Q_EXPORT QTextStream &operator<<( QTextStream&, const Element& );
Q_EXPORT QTextStream &operator>>( QTextStream&, Element& );
```

为了使我们的 Element 类更加独立，我们提供了<<和>>的操作符重载，这样 Element 就可以被文本流读写。我们也可以很容易地使用二进制流，但是使用文本可以让用户使用文本编辑器来维护他们的数据，可以更容易的使用脚本语言来产生和过滤。

（由 element.cpp 展开。）

```
#include "element.h"
```

```
#include <qstringlist.h>
#include <qtextstream.h>
```

我们对于操作符的实现需要包含 qtextstream.h 和 qstringlist.h。

```
const char FIELD_SEP = ':';
const char PROPOINT_SEP = ';';
const char XY_SEP = ',';
```

我们用来存储数据的格式是用冒号来分隔字段，用换行来分隔记录。比例点用分号间隔，它们的 x 和 y 使用逗号分隔。字段的顺序是值、值的颜色、值的样式、标签颜色、标签点、标签文本。比如：

```
20:#ff0000:14:#000000:0.767033,0.412946;0,0.75;0,0:Red :with colons:!
70:#00ffff:2:#ffff00:0.450549,0.198661;0.198516,0.125954;0,0.198473:Cyan
35:#0000ff:8:#555500:0.10989,0.299107;0.397032,0.562977;0,0.396947:Blue
55:#ffff00:1:#000080:0.0989011,0.625;0.595547,0.312977;0,0.59542:Yellow
80:#ff00ff:1:#000000:0.518681,0.694196;0.794063,0;0,0.793893:Magenta or Violet
```

我们阅读 Element 数据的方式中对于文本标签中的空白符和字段间隔符都没有问题。

```
QTextStream &operator<<( QTextStream &s, const Element &element )
{
    s << element.value() << FIELD_SEP
      << element.valueColor().name() << FIELD_SEP
      << element.valuePattern() << FIELD_SEP
      << element.labelColor().name() << FIELD_SEP;

    for ( int i = 0; i < Element::MAX_PROPOINTS; ++i ) {
        s << element.proX( i ) << XY_SEP << element.proY( i );
        s << ( i == Element::MAX_PROPOINTS - 1 ? FIELD_SEP : PROPOINT_SEP );
    }

    s << element.label() << '\n';

    return s;
}
```

写元素就是一直向前。每一个成员后面都被写一个字段间隔符。点被写成由逗号间隔的（XY_SEP）x 和 y 的组合，每一对由 PROPOINT_SEP 分隔符分隔。最后一个字段是标签和接着的换行符。

```
QTextStream &operator>>( QTextStream &s, Element &element )
{
    QString data = s.readLine();
    element.setValue( Element::INVALID );

    int errors = 0;
    bool ok;
```

```
        QStringList fields = QStringList::split( FIELD_SEP, data );
        if ( fields.count() >= 4 ) {
            double value = fields[0].toDouble( &ok );
            if ( !ok )
                errors++;
            QColor valueColor = QColor( fields[1] );
            if ( !valueColor.isValid() )
                errors++;
            int valuePattern = fields[2].toInt( &ok );
            if ( !ok )
                errors++;
            QColor labelColor = QColor( fields[3] );
            if ( !labelColor.isValid() )
                errors++;
            QStringList propoints = QStringList::split( PROPOINT_SEP, fields[4] );
            QString label = data.section( FIELD_SEP, 5 );

            if ( !errors ) {
                element.set( value, valueColor, valuePattern, label, labelColor );
                int i = 0;
                for ( QStringList::iterator point = propoints.begin();
                    i < Element::MAX_PROPOINTS && point != propoints.end();
                    ++i, ++point ) {
                    errors = 0;
                    QStringList xy = QStringList::split( XY_SEP, *point );
                    double x = xy[0].toDouble( &ok );
                    if ( !ok || x <= 0.0 || x >= 1.0 )
                        errors++;
                    double y = xy[1].toDouble( &ok );
                    if ( !ok || y <= 0.0 || y >= 1.0 )
                        errors++;
                    if ( errors )
                        x = y = Element::NO_PROPORTION;
                    element.setProX( i, x );
                    element.setProY( i, y );
                }
            }
        }

        return s;
    }
```

为了读取一个元素我们读取一条记录（比如一行）。我们使用 QStringList::split() 来把数据分成字段。因为标签中有可能包含 FIELD_SEP 字符，所以我们使用 QString::section()来获得从最后一个字段到这一行结尾的所有文本。如果获得了

足够的字段和值，颜色和样式数据是有效的，我们使用 Element::set()来把这些数据写到元素中，否则我们就会设置这个元素为 INVALID。然后我们对点也是这样。如果 x 和 y 比例是有效的并且在范围内，我们将会为元素设置它们。如果一个或两个比例是无效的，它们将认为值为零，这样是不合适的，所以我们将会改变无效的（和超出范围的）比例点的值为 NO_PROPORTION。
我们的 Element 类现在足够用来存储、维护和读写元素数据了。我们也创建了一个元素矢量类型定义来存储一个元素的集合。
我们现在已经准备好通过我们的用户来生成、编辑和可视化他们的数据组来生成 main.cpp 和用户界面。

如果要获得更多的有关 Qt 的数据流工具请看 QDataStream 操作符格式，和任何一个被提及的和你所要存储的东西相似的 Qt 类的源代码。

« “大图片” | 目录 | 主体很容易 »

主体很容易

（main.cpp。）

```
#include <qapplication.h>
#include "chartform.h"

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

    QString filename;
    if ( app.argc() > 1 ) {
        filename = app.argv()[1];
        if ( !filename.endsWith( ".cht" ) )
            filename = QString::null;
    }

    ChartForm *cf = new ChartForm( filename );
    app.setMainWidget( cf );
    cf->show();
    app.connect( &app, SIGNAL(lastWindowClosed()), cf, SLOT(fileQuit()) );

    return app.exec();
}
```

我们把 main()函数保持得很简单，很小。我们创建一个 QApplication 对象并且传递给它命令行参数。我们也允许用户通过 chart mychart.cht 来调用程序，所以如果他们已经添加了一个文件名，我们就把它传递给构造函数。图表窗口中的大多数行为我们将在下一步进行评论。

« 数据元素 | 目录 | 实现图形用户界面 »

实现图形用户界面

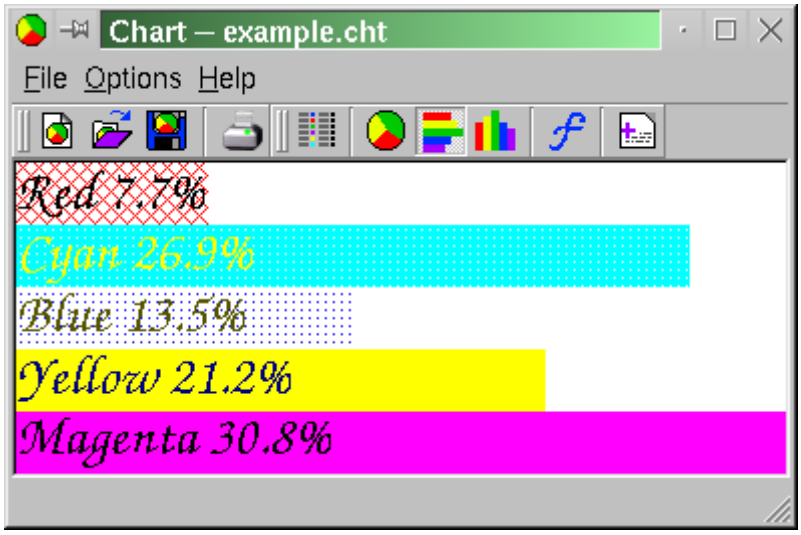


chart 程序提供了通过排列在中央窗口部件周围的菜单和工具条来访问选项，和一个通常的文档在中央的风格的 CanvasView。

（由 chartform.h 展开。）

```
class ChartForm: public QMainWindow
{
    Q_OBJECT
public:
    enum { MAX_ELEMENTS = 100 };
    enum { MAX_RECENTFILES = 9 }; // 必须不超过 9
    enum ChartType { PIE, VERTICAL_BAR, HORIZONTAL_BAR };
    enum AddValuesType { NO, YES, AS_PERCENTAGE };

    ChartForm( const QString& filename );
    ~ChartForm();

    int chartType() { return m_chartType; }
    void setChanged( bool changed = true ) { m_changed = changed; }
    void drawElements();

    QPopupMenu *optionsMenu; // 为什么是公有的？请看 canvasview.cpp。

private slots:
    void fileNew();
    void fileOpen();
    void fileOpenRecent( int index );
    void fileSave();
    void fileSaveAs();
    void fileSaveAsPixmap();
```

```
    void filePrint();
    void fileQuit();
    void optionsSetData();
    void updateChartType( QAction *action );
    void optionsSetFont();
    void optionsSetOptions();
    void helpHelp();
    void helpAbout();
    void helpAboutQt();
    void saveOptions();

private:
    void init();
    void load( const QString& filename );
    bool okToClear();
    void drawPieChart( const double scales[], double total, int count );
    void drawVerticalBarChart( const double scales[], double total, int count );
    void drawHorizontalBarChart( const double scales[], double total, int count );

    QString valueLabel( const QString& label, double value, double total );
    void updateRecentFiles( const QString& filename );
    void updateRecentFilesMenu();
    void setChartType( ChartType chartType );

    QPopupMenu *fileMenu;
    QAction *optionsPieChartAction;
    QAction *optionsHorizontalBarChartAction;
    QAction *optionsVerticalBarChartAction;
    QString m_filename;
    QStringList m_recentFiles;
    QCanvas *m_canvas;
    CanvasView *m_canvasView;
    bool m_changed;
    ElementVector m_elements;
    QPrinter *m_printer;
    ChartType m_chartType;
    AddValuesType m_addValues;
    int m_decimalPlaces;
    QFont m_font;
};
```

我们创建了一个 QMainWindow 的子类 ChartForm。我们的子类使用了 Q_OBJECT 宏来支持 Qt 的信号和槽机制。

公有接口是很少的，被显示的图表类型能够被追溯，图表可以被标记为“changed”（这样用户在退出的时候会被提示保存），并且图表可以要求拖拽自己

（drawElements()）。我们已经把选项菜单设为公有，因为我们也会把这个菜单作为画布视图的关联菜单。

QCanvas 类用来绘制二维矢量图。QCanvasView 类用来在一个应用程序的图形用户界面中实现一个画布的视图。我们所有的绘制操作都发生在画布上，但是事件（比如鼠标点击）却发生在画布视图中。

每一个动作都被一个私有槽实现，比如 fileNew()、optionsSetData()等等。我们也需要相当多的私有函数和数据成员，当我们执行这些实现的时候，我们来看看这些。

为了方便和编译速度的原因，图表视窗的实现被分为三个文件，chartform.cpp 实现图形用户界面，chartform_canvas.cpp 实现画布处理和 chartform_files.cpp 实现文件处理。我们会依次评论每一个。

图表视窗图形用户界面

（由 chartform.cpp 展开。）

```
    #include "images/file_new.xpm"
    #include "images/file_open.xpm"
    #include "images/options_piechart.xpm"
```

chart 中使用的所有图像是我们已经创建好并放在 images 子目录中的.xpm 文件。

构造函数

```
    ChartForm::ChartForm( const QString& filename )
        : QMainWindow( 0, 0, WDestructiveClose )
...
        QAction *fileNewAction;
        QAction *fileOpenAction;
        QAction *fileSaveAction;
```

对于每一个用户动作我们声明了一个 QAction 指针。一些动作在头文件中已经声明，因为它们需要在构造函数外被参考。

大部分用户动作适用于菜单条目和工具条按钮。Qt 允许用户创建一个单一的 QAction 而被添加到菜单和工具条中。这种方法保证了菜单条目和工具条按钮处于同步状态并且可以节省代码。

```
        fileNewAction = new QAction(
                "New Chart", QPixmap( file_new ),
                "&New", CTRL+Key_N, this, "new" );
        connect( fileNewAction, SIGNAL( activated() ), this, SLOT( fileNew() ) );
```

当我们构造一个动作时，我们给它一个名字、一个可选的图标、一个菜单文本和一个加速快捷键（或者 0 如果不需要加速键）。我们也可以使它成为视窗的子对象(通过 this)。当用户点击一个工具条按钮或者点击一个菜单选项时，activated()

信号会被发射。我们把这个信号和这个动作的槽连接起来，就是上面的程序代码中提到的 fileNew()。
图表类型是互斥的：我们可以用一个饼图*或*一个竖直条形图*或*一个水平条形图。这也就是说如果用户选择了饼图菜单选项，饼图工具条按钮也必须被自动地选中，并且其它图表菜单选项和工具条按钮必须被自动地取消选择。这种行为是通过创建一个 QActionGroup 来实现的并且把这些图表类型动作放到这个组中。

```
QActionGroup *chartGroup = new QActionGroup( this ); // Connected later
chartGroup->setExclusive( true );
```

动作组成为了视窗（this）的子对象并且 exlusive 行为通过 setExclusive()调用实现的。

```
optionsPieChartAction = new QAction(
        "Pie Chart", QPixmap( options_piechart ),
        "&Pie Chart", CTRL+Key_I, chartGroup, "pie chart" );
optionsPieChartAction->setToggleAction( true );
```

组中的每一个动作都以和其它动作一样的方式创建，除了动作的父对象是组而不是视窗。因为我们的图表类型动作由开/关状态，我们为它们中的每一个调用 setToggleAction(TRUE)。注意我们没有连接动作，相反，稍后我们会我们会把这个组连接到一个可以使画布重画的槽。

为什么我们不马上连接这个组呢？稍后在构造函数中我们将会读取用户选项，图表类型之一。我们将会直接设置图表类型。但那时我们还没有创建画布或者有任何数据，所以我们想做的一切就是切换画布类型工具条按钮，而不是真正地画（这时还不存在的）画布。在我们设置好画布类型*之后*，我们将会连接这个组。

一旦我们已经创建完所有的用户动作，我们就可以创建工具条和菜单选项来允许用户调用它们。

```
QToolBar* fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
fileNewAction->addTo( fileTools );
fileOpenAction->addTo( fileTools );
fileSaveAction->addTo( fileTools );
...
fileMenu = new QPopupMenu( this );
menuBar()->insertItem( "&File", fileMenu );
fileNewAction->addTo( fileMenu );
fileOpenAction->addTo( fileMenu );
fileSaveAction->addTo( fileMenu );
```

工具条动作和菜单选项可以很容易地由 QAction 生成。
作为一个对我们的用户提供的方便，我们将会重新载入上次窗口的位置和大小并列出最近使用的文件。这是通过在程序退出的时候写出这些设置，在我们构造视窗的时候再把它们都回来实现的。

```
QSettings settings;
settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
int windowWidth = settings.readNumEntry( APP_KEY + "WindowWidth", 460 );
int windowHeight = settings.readNumEntry( APP_KEY + "WindowHeight", 530 );
int windowX = settings.readNumEntry( APP_KEY + "WindowX", 0 );
```

```
    int windowY = settings.readNumEntry( APP_KEY + "WindowY", 0 );
    setChartType( ChartType(
            settings.readNumEntry( APP_KEY + "ChartType", int(PIE) ) ) );
    m_font = QFont( "Helvetica", 18, QFont::Bold );
    m_font.fromString(
            settings.readEntry( APP_KEY + "Font", m_font.toString() ) );
    for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
        QString filename = settings.readEntry( APP_KEY + "File" +
                                               QString::number( i + 1 ) );
        if ( !filename.isEmpty() )
            m_recentFiles.push_back( filename );
    }
    if ( m_recentFiles.count() )
        updateRecentFilesMenu();
```

QSettings 类通过和平台无关的方式来处理用户设置。我们很简单地读写设置，把处理平台依赖性的问题留给 QSettings 来处理。insertSearchPath()调用没有做任何事，除非在 Windows 下被#ifdef 过。

我们使用 readNumEntry()调用来得到图表视窗上次的大小和位置，并且为它的第一次运行提供了默认值。图表类型是以一个整数重新获得并把它扔给 CharType 枚举值。我们创建默认标签字体，然后读取“Font”设置，如果需要的话我们使用刚才生成的默认字体。

尽管 QSettings 可以处理字符串列表，但是我们已经选择把最近使用的每一个文件作为单一的条目来存储，这样就可以更容易地处理和编辑这些设置。我们试着去读每一个可能的文件条目（从“File1”到“File9”），并把每一个非空条目添加到最近使用的文件的列表中。如果有一个或多个最近使用的文件，我们通过调用 updateRecentFilesMenu()来更新 File 菜单，（我们将会在稍后再评论这个）。

```
    connect( chartGroup, SIGNAL( selected(QAction*) ),
             this, SLOT( updateChartType(QAction*) ) );
```

现在我们已经设置图表类型（当我们把它作为一个用户设置读入的时候），把图表组和我们的 updateChartType()槽连接起来是安全的。

```
    resize( windowWidth, windowHeight );
    move( windowX, windowY );
```

并且现在我们已经知道窗口大小和位置，我们就可以根据这些重新定义大小并移动图表视窗窗口。

```
    m_canvas = new QCanvas( this );
    m_canvas->resize( width(), height() );
    m_canvasView = new CanvasView( m_canvas, &m_elements, this );
    setCentralWidget( m_canvasView );
    m_canvasView->show();
```

我们创建一个新的 QCanvas 并且设置它的大小为图表视窗窗口的客户区域。我们也创建一个 CanvasView（我们自己的 QCanvasView 的子类）来显示 QCanvas。我们把这个画布视图作为图表视窗的主窗口部件并显示它。

```
    if ( !filename.isEmpty() )
        load( filename );
```

```
    else {
        init();
        m_elements[0].set( 20, red,     14, "Red" );
        m_elements[1].set( 70, cyan,     2, "Cyan",    darkGreen );
        m_elements[2].set( 35, blue,    11, "Blue" );
        m_elements[3].set( 55, yellow,   1, "Yellow", darkBlue );
        m_elements[4].set( 80, magenta, 1, "Magenta" );
        drawElements();
    }
```

如果我们有一个文件要载入，我们就载入它，否则我们就初始化我们的元素矢量并画一个示例图表。

```
    statusBar()->message( "Ready", 2000 );
```

我们在构造函数中调用 statusBar()是*非常重要的*，因为这个调用保证了我们能够在这个主窗口中创建一个状态条。

init()

```
void ChartForm::init()
{
    setCaption( "Chart" );
    m_filename = QString::null;
    m_changed = false;

    m_elements[0]  = Element( Element::INVALID, red );
    m_elements[1]  = Element( Element::INVALID, cyan );
    m_elements[2]  = Element( Element::INVALID, blue );
...
```

我们使用了 init()函数是因为我们想在视窗被构造的时候和无论用户载入一个存在的数据组或者创建一个新的数据组的时候初始化画布和元素（在 m_elements ElementVector 中）。

我们重新设置标题并设置当前文件名称为 QString::null。我们也用无效的元素来组装元素矢量。这不是必需的，但是给每一个元素一个不同的颜色对于用户来讲是更方便的，因为当他们输入值的时候每一个都会已经有了一个确定的颜色（当然他们可以修改）。

文件处理动作

okToClear()

```
bool ChartForm::okToClear()
{
    if ( m_changed ) {
        QString msg;
        if ( m_filename.isEmpty() )
```

```
            msg = "Unnamed chart ";
        else
            msg = QString( "Chart '%1'\n" ).arg( m_filename );
        msg += "has been changed.";
        switch( QMessageBox::information( this, "Chart -- Unsaved Changes",
                                          msg, "&Save", "Cancel", "&Abandon",
                                          0, 1 ) ) {
            case 0:
                fileSave();
                break;
            case 1:
            default:
                return false;
                break;
            case 2:
                break;
        }
    }

    return true;
}
```

okToClear()函数用来提示用户在有没保存的数据的时候保存它们。它也被其它几个函数使用。

fileNew()

```
void ChartForm::fileNew()
{
    if ( okToClear() ) {
        init();
        drawElements();
    }
}
```

当用户调用 fileNew()动作时，我们调用 okToClear()来给他们一个保存任何为保存的数据的机会。无论他们保存或者放弃或者没有任何为保存的数据，我们都重新初始化元素矢量并绘制默认图表。

我们是不是也应该调用 optionsSetData()来弹出一个对话框，让用户通过它来创建和编辑值、颜色等等呢？你可以运行一下现在的应用程序，然后试着把 optionsSetData()的调用添加进去后再运行并观察它们来决定你更喜欢哪一个。

fileOpen()

```
void ChartForm::fileOpen()
{
```

```
    if ( !okToClear() )
        return;

    QString filename = QFileDialog::getOpenFileName(
                            QString::null, "Charts (*.cht)", this,
                            "file open", "Chart -- File Open" );
    if ( !filename.isEmpty() )
        load( filename );
    else
        statusBar()->message( "File Open abandoned", 2000 );
}
```

我们检查它是否是 okToClear()。如果是的话，我们使用静态的 QFileDialog::getOpenFileName()函数来获得用户想要载入的文件的名称。如果我们得到一个文件名，我们就调用 load()。

fileSaveAs()

```
void ChartForm::fileSaveAs()
{
    QString filename = QFileDialog::getSaveFileName(
                            QString::null, "Charts (*.cht)", this,
                            "file save as", "Chart -- File Save As" );
    if ( !filename.isEmpty() ) {
        int answer = 0;
        if ( QFile::exists( filename ) )
            answer = QMessageBox::warning(
                            this, "Chart -- Overwrite File",
                            QString( "Overwrite\n\'%1\'?" ).
                                arg( filename ),
                            "&Yes", "&No", QString::null, 1, 1 );
        if ( answer == 0 ) {
            m_filename = filename;
            updateRecentFiles( filename );
            fileSave();
            return;
        }
    }
    statusBar()->message( "Saving abandoned", 2000 );
}
```

这个函数调用了静态的 QFileDialog::getSaveFileName()来得到一个要保存数据的文件的明处那个。如果文件存在，我们使用使用一个 QMessageBox::warning()来提醒用户并给他们一个放弃保存的选择。如果文件被保存了我们就更新最近打开的文件列表并调用 fileSave()（在文件处理中）来执行存储。

管理最近打开文件的列表

```
QStringList m_recentFiles;
```

我们用一个字符串列表来处理这个最近打开文件的列表。

```
void ChartForm::updateRecentFilesMenu()
{
    for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
        if ( fileMenu->findItem( i ) )
            fileMenu->removeItem( i );
        if ( i < int(m_recentFiles.count()) )
            fileMenu->insertItem( QString( "&%1 %2" ).
                                  arg( i + 1 ).arg( m_recentFiles[i] ),
                                this, SLOT( fileOpenRecent(int) ),
                                0, i );
    }
}
```

无论用户打开一个存在的文件或者保存一个新文件的时候，这个函数会被调用（通常是通过 updateRecentFiles()）。对于这个字符串列表中的每一个文件我们都插入一个新的菜单条目。我们在每一个文件名的前面都加上一个从 1 到 9 带下划线的数字，这样就可以支持键盘操作（比如，Alt+F，2 就可以打开列表中的第二个文件）。我们给每一个菜单条目一个和它们在字符串列表中的索引位置相同的数值作为 id，并且把每一个菜单条目都和 fileOpenRecent()槽相连。老的文件菜单条目会在每一个最新的文件菜单条目 id 来到的同时被删除。它会工作是因为其它文件菜单条目都有一个由 Qt 生成的 id（它们都是<0 的），然而我们所创建的菜单条目的 id 都是>=0 的。

```
void ChartForm::updateRecentFiles( const QString& filename )
{
    if ( m_recentFiles.find( filename ) != m_recentFiles.end() )
        return;

    m_recentFiles.push_back( filename );
    if ( m_recentFiles.count() > MAX_RECENTFILES )
        m_recentFiles.pop_front();

    updateRecentFilesMenu();
}
```

当用户打开一个存在的文件或者保存一个新文件的时候，它会被调用。如果文件已经存在于列表中，它就会很简单地返回。否则这个文件会被添加到列表的末尾并且如果列表太大（>9 个文件）的话，第一个（最老的）就会被移去。然后 updateRecentFilesMenu()被调用来在 File 菜单中重新创建最近使用的文件列表。

```
void ChartForm::fileOpenRecent( int index )
{
    if ( !okToClear() )
```

```
            return;

        load( m_recentFiles[index] );
    }
```

当用户选择了一个最近打开的文件时，fileOpenRecent()槽会伴随一个用户选择的文件的菜单 id 而被调用。因为我们使文件菜单的 id 和文件在 m_recentFiles 列表中的索引位置相等，我们就可以很简单的通过文件的菜单条目 id 来载入了。

退出

```
    void ChartForm::fileQuit()
    {
        if ( okToClear() ) {
            saveOptions();
            qApp->exit( 0 );
        }
    }
```

当用户退出时，我们给他们保存任何未保存数据的机会（okToClear()），然后在结束之前保存它们的选项，比如窗口的大小和位置、图表类型等等。

```
    void ChartForm::saveOptions()
    {
        QSettings settings;
        settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
        settings.writeEntry( APP_KEY + "WindowWidth", width() );
        settings.writeEntry( APP_KEY + "WindowHeight", height() );
        settings.writeEntry( APP_KEY + "WindowX", x() );
        settings.writeEntry( APP_KEY + "WindowY", y() );
        settings.writeEntry( APP_KEY + "ChartType", int(m_chartType) );
        settings.writeEntry( APP_KEY + "AddValues", int(m_addValues) );
        settings.writeEntry( APP_KEY + "Decimals", m_decimalPlaces );
        settings.writeEntry( APP_KEY + "Font", m_font.toString() );
        for ( int i = 0; i < int(m_recentFiles.count()); ++i )
            settings.writeEntry( APP_KEY + "File" + QString::number( i + 1 ),
                                 m_recentFiles[i] );
    }
```

直接使用 QSettings 来保存用户选项。

自定义对话框

我们想让用户可以手工地设置一些选项并且创建和编辑值、值颜色等等。

```
    void ChartForm::optionsSetOptions()
    {
        OptionsForm *optionsForm = new OptionsForm( this );
```

```
        optionsForm->chartTypeComboBox->setCurrentItem( m_chartType );
        optionsForm->setFont( m_font );
        if ( optionsForm->exec() ) {
            setChartType( ChartType(
                    optionsForm->chartTypeComboBox->currentItem()) );
            m_font = optionsForm->font();
            drawElements();
        }
        delete optionsForm;
    }
```

设置选项的视窗是由我们自定义的 OptionsForm 提供的，在设置选项中。这个选项视窗是一个标准的“哑的”对话框：我们创建一个实例，把所有的图形用户界面元素都和所有相关的设置都组装起来，并且如果用户点击了“OK”（exec()返回一个真值）我们就会从图形用户界面元素中读取设置。

```
    void ChartForm::optionsSetData()
    {
        SetDataForm *setDataForm = new SetDataForm( &m_elements, m_decimalPlaces,
this );
        if ( setDataForm->exec() ) {
            m_changed = true;
            drawElements();
        }
        delete setDataForm;
    }
```

创建和编辑图表数据的视窗由我们自定义的 SetDataForm 提供，在获得数据中。这个视窗是一个“聪明的”对话框。我们传入我们想要使用的数据结构，并且对话框可以自己处理数据机构的表达。如果用户点击“OK”，对话框会更新数据结构并且 exec()会返回一个真值。如果用户改变了数据时我们在 optionsSetData()中所要做的时把图表标记为 changed 并调用 drawElements()来使用新的和更新过的数据来重新绘制图表。

« 主体很容易 | 目录 | 画布控制 »

画布控制

我们在画布上画饼形区域（或者条形图表条），和所有的标签。画布是通过画布视图来呈现给用户的。drawElements()函数被调用从而在需要的时候重新绘制画布。

（由 chartform_canvas.cpp 展开。）

drawElements()

```
    void ChartForm::drawElements()
```

```
{
    QCanvasItemList list = m_canvas->allItems();
    for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
        delete *it;
```

我们在 drawElements()中所作的第一件事是删除所有已经存在的画布条目。

```
    // 360 * 16 为一个饼形,Qt 中使用的是 16 倍的度数(就是它的一个圆周为 360x16)
    int scaleFactor = m_chartType == PIE ? 5760 :
                      m_chartType == VERTICAL_BAR ? m_canvas->height() :
                          m_canvas->width();
```

接下来我们根据要绘制的图表的种类来计算比例因子。

```
    double biggest = 0.0;
    int count = 0;
    double total = 0.0;
    static double scales[MAX_ELEMENTS];

    for ( int i = 0; i < MAX_ELEMENTS; ++i ) {
        if ( m_elements[i].isValid() ) {
            double value = m_elements[i].value();
            count++;
            total += value;
            if ( value > biggest )
                biggest = value;
            scales[i] = m_elements[i].value() * scaleFactor;
        }
    }

    if ( count ) {
            // 第二个循环是因为总量和最大的
        for ( int i = 0; i < MAX_ELEMENTS; ++i )
            if ( m_elements[i].isValid() )
                if ( m_chartType == PIE )
                    scales[i] = (m_elements[i].value() * scaleFactor) / total;
                else
                    scales[i] = (m_elements[i].value() * scaleFactor) / biggest;
```

我们需要知道这里有多少值、最大的值和值的总和，这样我们就可以正确地按比例创建饼形区域或条形了。我们把比例值存放在 scales 数组中。

```
        switch ( m_chartType ) {
            case PIE:
                drawPieChart( scales, total, count );
                break;
            case VERTICAL_BAR:
                drawVerticalBarChart( scales, total, count );
                break;
            case HORIZONTAL_BAR:
```

```
                drawHorizontalBarChart( scales, total, count );
                break;
            }
        }
```

既然我们已经知道了必需的信息，那我们就调用相关绘制函数，传递比例值、总量和计数。

```
        m_canvas->update();
```

最终我们使用 update()更新画布来使所有的变化可视。

drawHorizontalBarChart()

我们来回顾一下刚才的这个绘制函数，看到了画布条目如何被生成并放置到画布上，因为这个教程是关于 Qt 的，而不是关于绘制图表的好的（或者坏的）算法。

```
    void ChartForm::drawHorizontalBarChart(
            const double scales[], double total, int count )
    {
```

画水平条形图我们需要一个比例值的数组、总量（这样我们就可以在需要的时候计算并且画出百分比）和这一组值的计数。

```
        double width = m_canvas->width();
        double height = m_canvas->height();
        int proheight = int(height / count);
        int y = 0;
```

我们重新得到画布的宽度和高度并且计算比例高度（proheight）。我们把初始的 y 位置设为 0。

```
        QPen pen;
        pen.setStyle( NoPen );
```

我们创建一个用来绘制每一个条形（矩形）的画笔，我们把它设置为 NoPen，这样就不会画出边框。

```
        for ( int i = 0; i < MAX_ELEMENTS; ++i ) {
            if ( m_elements[i].isValid() ) {
                int extent = int(scales[i]);
```

我们在元素矢量中迭代每一个元素，忽略无效的元素。每个条的宽度（它的长度）很简单地就是它的比例值。

```
                QCanvasRectangle *rect = new QCanvasRectangle(
                                                0, y, extent, proheight, m_canvas );
                rect->setBrush( QBrush( m_elements[i].valueColor(),

BrushStyle(m_elements[i].valuePattern()) ) );
                rect->setPen( pen );
                rect->setZ( 0 );
                rect->show();
```

我们为每个条形创建一个新的 QCanvasRectangle，它的 x 位置为 0（因为这是一个水平条形图，每个条形都从左边开始），y 值从 0 开始，随着每一个要画的条形的高度增长，一直到我们要画的条形和画布的高度。然后我们设置条形的画刷

为用户为元素指定的颜色和样式，设置画笔为我们先前生成的画笔（比如，设置为 NoPen）并且我们把条形的 Z 轴顺序设置为 0。最后我们调用 show()在画布上绘制条形。

```
                QString label = m_elements[i].label();
                if ( !label.isEmpty() || m_addValues != NO ) {
                    double proX = m_elements[i].proX( HORIZONTAL_BAR );
                    double proY = m_elements[i].proY( HORIZONTAL_BAR );
                    if ( proX < 0 || proY < 0 ) {
                        proX = 0;
                        proY = y / height;
                    }
```

如果用户已经为元素指定了标签或者要求把值（或者百分比）显示出来，我们也要画一个画布文本条目。我们创建我们自己的 CanvasText 类（请看后面），因为我们想存储每一个画布文本条目中对应元素的索引（在元素矢量中）。我们从元素中得出 x 和 y 的比例值。如果其中之一< 0，那么他们还没有被用户定位，所以你必须计算它们的位置。我们标签的 x 值为 0（左）并且 y 值为条形图的顶部（这样标签的左上角就会在 x,y 位置）。

```
                    label = valueLabel( label, m_elements[i].value(), total );
```

然后我们调用一个助手函数 valueLabel()，它可以返回一个包含标签文本的字符串。（如果用户已经设置相应的选项，valueLabel()函数添加值或者百分比到这个文本的标签。）

```
                    CanvasText *text = new CanvasText( i, label, m_font, m_canvas );
                    text->setColor( m_elements[i].labelColor() );
                    text->setX( proX * width );
                    text->setY( proY * height );
                    text->setZ( 1 );
                    text->show();
                    m_elements[i].setProX( HORIZONTAL_BAR, proX );
                    m_elements[i].setProY( HORIZONTAL_BAR, proY );
```

然后我们创建一个 CanvasText 条目，传递给它在元素矢量中这个元素的索引和所要使用的标签、字体和画布。我们设置文本条目的颜色为用户指定的颜色并且设置条目的 x 和 y 位置和画布的宽高成比例。我们设置 Z 轴顺序为 1，这样文本条目总是在条形（Z 轴顺序为 0）的上面（前面）。我们调用 show()函数在画布上绘制文本条目，并且设置元素的相对 x 和 y 位置。

```
                }
                y += proheight;
```

在绘制完条形和可能存在的标签之后，我们给 y 增加一定比例的高度用来准备绘制下一个元素。

```
            }
        }
    }
```

QCanvasText 的子类

（由 canvastext.h 展开。）

```
class CanvasText : public QCanvasText
{
public:
    enum { CANVAS_TEXT = 1100 };

    CanvasText( int index, QCanvas *canvas )
        : QCanvasText( canvas ), m_index( index ) {}
    CanvasText( int index, const QString& text, QCanvas *canvas )
        : QCanvasText( text, canvas ), m_index( index ) {}
    CanvasText( int index, const QString& text, QFont font, QCanvas *canvas )
        : QCanvasText( text, font, canvas ), m_index( index ) {}

    int index() const { return m_index; }
    void setIndex( int index ) { m_index = index; }

    int rtti() const { return CANVAS_TEXT; }

private:
    int m_index;
};
```

我们的 CanvasText 子类是 QCanvasText 的一个非常简单的特化。我们所做的一切只是添加一个私有成员 m_index，它用来保存和这个文本相关的元素的元素矢量索引，并且提供为这个值提供一个读和写函数。

QCanvasView 的子类

（由 canvasview.h 展开。）

```
class CanvasView : public QCanvasView
{
    Q_OBJECT
public:
    CanvasView( QCanvas *canvas, ElementVector *elements,
                QWidget* parent = 0, const char* name = "canvas view",
                WFlags f = 0 )
        : QCanvasView( canvas, parent, name, f ),
          m_elements( elements ) {}

protected:
    void viewportResizeEvent( QResizeEvent *e );
    void contentsMousePressEvent( QMouseEvent *e );
    void contentsMouseMoveEvent( QMouseEvent *e );
    void contentsContextMenuEvent( QContextMenuEvent *e );
```

```
private:
    QCanvasItem *m_movingItem;
    QPoint m_pos;
    ElementVector *m_elements;
};
```

我们需要继承 QCanvasView，这样我们就能处理：

1. 上下文菜单请求。
2. 视窗重定义大小。
3. 用户拖拽标签到任意位置。

为了支持这些，我们存储一个到正在被移动的画布条目的指针和它的最终位置。我们也存储一个到元素矢量的指针。

上下文菜单请求

（由 canvasview.cpp 展开。）

```
void CanvasView::contentsContextMenuEvent( QContextMenuEvent * )
{
    ((ChartForm*)parent())->optionsMenu->exec( QCursor::pos() );
}
```

当用户调用一个上下文菜单（比如在绝大多数平台通过右键点击），我们把画布视图的父对象（是一个 ChartForm）转化为正确的类型，然后用 exec()在光标位置执行选项菜单。

视窗重定义大小

```
void CanvasView::viewportResizeEvent( QResizeEvent *e )
{
    canvas()->resize( e->size().width(), e->size().height() );
    ((ChartForm*)parent())->drawElements();
}
```

为了改变大小我们简单地改变花布的大小，画布视图就会呈现在视窗客户端区域的宽高中，然后调用 drawElements()函数来重新绘制图表。因为 drawElements()画的每一件都和画布的宽高有关，所以图表就会被正确地绘制。

拖拽标签到任意位置

当用户想把标签拖拽到他们点击的位置时，就应该拖拽它并在新的位置释放它。

```
void CanvasView::contentsMousePressEvent( QMouseEvent *e )
{
    QCanvasItemList list = canvas()->collisions( e->pos() );
```

```
        for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
            if ( (*it)->rtti() == CanvasText::CANVAS_TEXT ) {
                m_movingItem = *it;
                m_pos = e->pos();
                return;
            }
        m_movingItem = 0;
    }
```

当用户点击鼠标时，我们创建一个鼠标点击“碰撞”（如果有的话）的画布条目的列表。然后我们迭代这个列表并且如果我们发现一个 CanvasText 条目，我们就把它设置为移动的条目并且记录下它的位置。否则我们设置为不移动条目。

```
    void CanvasView::contentsMouseMoveEvent( QMouseEvent *e )
    {
        if ( m_movingItem ) {
            QPoint offset = e->pos() - m_pos;
            m_movingItem->moveBy( offset.x(), offset.y() );
            m_pos = e->pos();
            ChartForm *form = (ChartForm*)parent();
            form->setChanged( true );
            int chartType = form->chartType();
            CanvasText *item = (CanvasText*)m_movingItem;
            int i = item->index();
            (*m_elements)[i].setProX( chartType, item->x() / canvas()->width() );
            (*m_elements)[i].setProY( chartType, item->y() / canvas()->height() );
            canvas()->update();
        }
    }
```

当用户拖拽鼠标的时候，移动事件就产生了。如果那里是一个可移动条目，我们从鼠标最后的位置和可移动条目原来的位置计算出位移。我们将新的位置记录为最后的位置。因为图表现在改变了，所以我们调用 setChanged()，这样当用户试图退出或者读入已存在的图表时或者创建新的图表，就会被提示是否保存。我们也分别地更新当前图表类型的元素的 x 和 y 的比例位置为当前 x 和 y 与宽和高的比例。我们知道要更新哪个元件因为当我们创建每个画布文本条目的时候，我们传给它一个这个元素所对应的位置索引。我们继承了 QCanvasText，这样我们就可以设置和读取这个索引值。最后我们调用 update()来重绘画布。

QCanvas 没有任何视觉效果。为了看到画布的内容，你必须创建一个 QCanvasView 来呈现画布。如果条目被 show()显示，它们就会出现在画布视图中，然后，只有当 QCanvas::update() 被调用的时候。默认情况下 QCanva 的背景是白色，并且会在画布上绘制默认的形状，比如 QCanvasRectangle、QCanvasEllipse 等等，因为它们被白色填充，所以非常推荐使用一个非白色的画刷颜色！

« 实现图形用户界面 | 目录 | 文件处理 »

文件处理

（从 chartform_files.cpp 展开。）

读图表数据

```
void ChartForm::load( const QString& filename )
{
    QFile file( filename );
    if ( !file.open( IO_ReadOnly ) ) {
        statusBar()->message( QString( "Failed to load \'%1\'" ).
                              arg( filename ), 2000 );
        return;
    }

    init(); // 确保我们拥有颜色
    m_filename = filename;
    QTextStream ts( &file );
    Element element;
    int errors = 0;
    int i = 0;
    while ( !ts.eof() ) {
        ts >> element;
        if ( element.isValid() )
            m_elements[i++] = element;
    file.close();
    setCaption( QString( "Chart -- %1" ).arg( filename ) );
    updateRecentFiles( filename );

    drawElements();
    m_changed = false;
}
```

载入数据组非常容易。我们打开文件并且创建一个文本流。当有数据要读的时候，我们把一个元素读入到 element 并且如果它是有效的，我们就把它插入到 m_elements 矢量。所有的细节都由 Element 类来处理。然后我们关闭文件并且更新标题和最近打开的文件列表。最后我们绘制图表并标明它没有被改变。

写图表数据

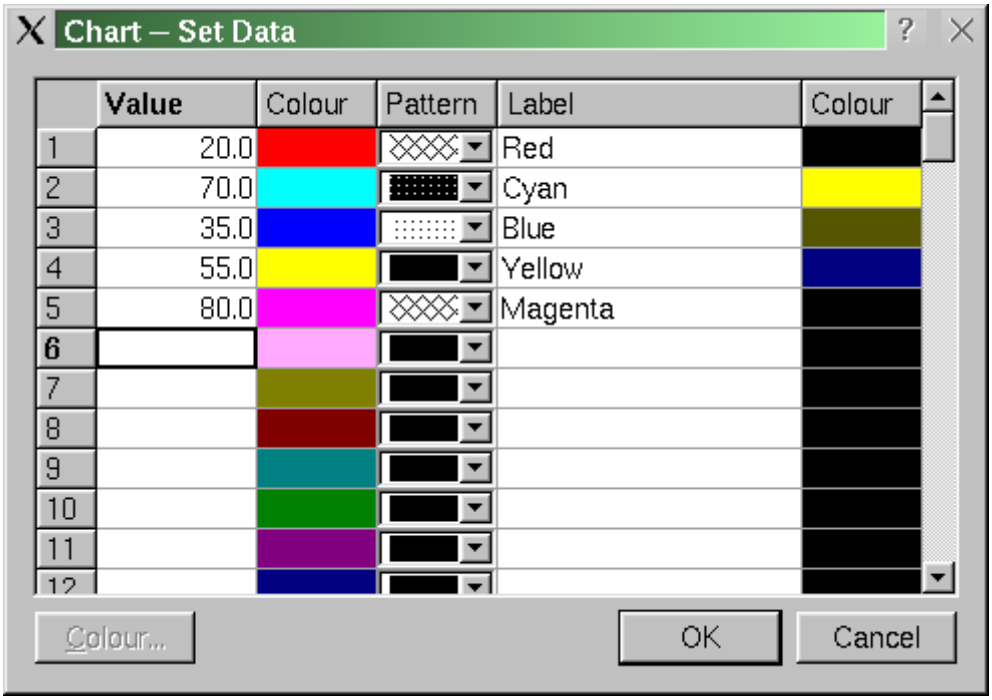
```
void ChartForm::fileSave()
{
    QFile file( m_filename );
    if ( !file.open( IO_WriteOnly ) ) {
        statusBar()->message( QString( "Failed to save \'%1\'" ).
                              arg( m_filename ), 2000 );
```

```
            return;
        }
        QTextStream ts( &file );
        for ( int i = 0; i < MAX_ELEMENTS; ++i )
            if ( m_elements[i].isValid() )
                ts << m_elements[i];

        file.close();

        setCaption( QString( "Chart -- %1" ).arg( m_filename ) );
        statusBar()->message( QString( "Saved \'%1\'" ).arg( m_filename ), 2000 );
        m_changed = false;
    }
```

保存数据一样地容易。我们打开文件并且创建一个文本流。然后我们把每一个有效元素写到文本留中。所有的细节都由 Element 类来处理。

« 画布控制 | 目录 | 获得数据 »

获得数据



设置数据对话框允许用户添加和编辑值，并且可以选择用来显示值的颜色和样式。用户可以可以输入标签文本并为每一个标签选择一个标签颜色。

（由 setdataform.h 展开。）

```
class SetDataForm: public QDialog
{
    Q_OBJECT
public:
```

```
    SetDataForm( ElementVector *elements, int decimalPlaces,
                 QWidget *parent = 0, const char *name = "set data form",
                 bool modal = TRUE, WFlags f = 0 );
    ~SetDataForm() {}

public slots:
    void setColor();
    void setColor( int row, int col );
    void currentChanged( int row, int col );
    void valueChanged( int row, int col );

protected slots:
    void accept();

private:
    QTable *table;
    QPushButton *colorPushButton;
    QPushButton *okPushButton;
    QPushButton *cancelPushButton;

protected:
    QVBoxLayout *tableButtonBox;
    QHBoxLayout *buttonBox;

private:
    ElementVector *m_elements;
    int m_decimalPlaces;
};
```

头文件很简单。构造函数中用一个指针指向元素矢量，这样这个“聪明的”对话框就可以直接显示并且编辑数据。我们将会解释我们在实现中所看到的槽的。（由 setdataform.cpp 展开。）

```
#include "images/pattern01.xpm"
#include "images/pattern02.xpm"
```

我们创建了一个小的.XPM 图片用来显示 Qt 支持的每一种画刷样式。我们将会在样式组合框中使用这些的。

构造函数

```
SetDataForm::SetDataForm( ElementVector *elements, int decimalPlaces,
                          QWidget* parent,  const char* name,
                          bool modal, WFlags f )
    : QDialog( parent, name, modal, f )

{
```

```
    m_elements = elements;
    m_decimalPlaces = decimalPlaces;
```

我们传递了绝大部分参数到 QDialog 超类中。我们把元素矢量指针和所要显示的小数点位数赋给成员变量，这样它们就可以被所有的 SetDataForm 的成员函数访问了。

```
    setCaption( "Chart -- Set Data" );
    resize( 540, 440 );
```

我们为对话框设置一个标题并且重定义它的大小。

```
    tableButtonBox = new QVBoxLayout( this, 11, 6, "table button box layout" );
```

这个视窗的布局相当简单。按钮被组织在一个水平的布局中并且表和这个按钮布局通过使用 tableButtonBox 布局被竖直地组织在一起。

```
    table = new QTable( this, "data table" );
    table->setNumCols( 5 );
    table->setNumRows( ChartForm::MAX_ELEMENTS );
    table->setColumnReadOnly( 1, true );
    table->setColumnReadOnly( 2, true );
    table->setColumnReadOnly( 4, true );
    table->setColumnWidth( 0, 80 );
    table->setColumnWidth( 1, 60 ); // Columns 1 and 4 must be equal
    table->setColumnWidth( 2, 60 );
    table->setColumnWidth( 3, 200 );
    table->setColumnWidth( 4, 60 );
    QHeader *th = table->horizontalHeader();
    th->setLabel( 0, "Value" );
    th->setLabel( 1, "Color" );
    th->setLabel( 2, "Pattern" );
    th->setLabel( 3, "Label" );
    th->setLabel( 4, "Color" );
    tableButtonBox->addWidget( table );
```

我们创建一个有五列的新的 QTable，并且它的行数和元素矢量中的元素个数相同。我们让颜色和样式列只读：这是为了防止用户在这些地方输入。我们将通过让用户在颜色上点击或者定位到颜色上并且点击 Color 按钮时可以修改颜色。样式被放在一个组合框中，很简单地通过用户选择一个不同地样式就可以改变它。接下来我们设置合适地初始宽度，为每一列插入标签并且最后把这个表添加到 tableButtonBox 布局中。

```
    buttonBox = new QHBoxLayout( 0, 0, 6, "button box layout" );
```

我们创建一个水平盒子布局用来保存按钮。

```
    colorPushButton = new QPushButton( this, "color button" );
    colorPushButton->setText( "&Color..." );
    colorPushButton->setEnabled( false );
    buttonBox->addWidget( colorPushButton );
```

我们创建一个 color 按钮并把它添加到 buttonBox 布局中。我们让这个按钮失效，只有当焦点在一个颜色单元格时，我们才会让它有效。

```
    QSpacerItem *spacer = new QSpacerItem( 0, 0, QSizePolicy::Expanding,
```

```
                                                    QSizePolicy::Minimum );
    buttonBox->addItem( spacer );
```

因为我们想把 color 按钮和 OK 以及 Cancel 按钮分开，接下来我们创建一个间隔并把它添加到 buttonBox 布局中。

```
    okPushButton = new QPushButton( this, "ok button" );
    okPushButton->setText( "OK" );
    okPushButton->setDefault( TRUE );
    buttonBox->addWidget( okPushButton );

    cancelPushButton = new QPushButton( this, "cancel button" );
    cancelPushButton->setText( "Cancel" );
    cancelPushButton->setAccel( Key_Escape );
    buttonBox->addWidget( cancelPushButton );
```

OK 和 Cancel 按钮被创建了并被添加到 buttonBox。我们让 OK 按钮为这个对话框的默认按钮，并且我们为 Cancel 按钮提供了一个 Esc 加速键。

```
    tableButtonBox->addLayout( buttonBox );
```

我们把 buttonBox 布局添加到 tableButtonBox 中，并且这个布局也是完整的。

```
    connect( table, SIGNAL( clicked(int,int,int,const QPoint&) ),
             this, SLOT( setColor(int,int) ) );
    connect( table, SIGNAL( currentChanged(int,int) ),
             this, SLOT( currentChanged(int,int) ) );
    connect( table, SIGNAL( valueChanged(int,int) ),
             this, SLOT( valueChanged(int,int) ) );
    connect( colorPushButton, SIGNAL( clicked() ), this, SLOT( setColor() ) );
    connect( okPushButton, SIGNAL( clicked() ), this, SLOT( accept() ) );
    connect( cancelPushButton, SIGNAL( clicked() ), this, SLOT( reject() ) );
```

现在我们来演习一下这个视窗。

- 如果用户点击了一个单元格，我们调用 setColor()槽，它会检查这个单元格是否保存一个颜色，如果是的，将会调用颜色对话框。
- 我们把 QTable 的 currentChanged()信号和我们的 currentChanged()槽连接起来了，举例来说，这将被用在根据用户现在所在的列来决定使 color 按钮有效/失效。
- 我们把表格的 valueChanged()和我们的 valueChanged()槽连接起来了，我们将会用这个来显示带有正确的小数位数的值。
- 如果用户点击 Color 按钮，我们就调用 setColor()槽。
- OK 按钮被连接到 accept()槽，我们将会在这个槽里面更新元素矢量。
- Cancel 按钮被连接到 QDialog 的 reject()槽，并且这部分中不再需要更多的代码和动作。

```
    QPixmap patterns[MAX_PATTERNS];
    patterns[0]  = QPixmap( pattern01 );
    patterns[1]  = QPixmap( pattern02 );
```

我们为每一个画刷样式创建了一个图片并且把它们存储在 patterns 数组中。

```
    QRect rect = table->cellRect( 0, 1 );
    QPixmap pix( rect.width(), rect.height() );
```

我们每一个颜色单元格所占用的矩形并创建一个这样大小的空白图片。

```
        for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
            Element element = (*m_elements)[i];

            if ( element.isValid() )
                table->setText(
                    i, 0,
                    QString( "%1" ).arg( element.value(), 0, 'f',
                                         m_decimalPlaces ) );

            QColor color = element.valueColor();
            pix.fill( color );
            table->setPixmap( i, 1, pix );
            table->setText( i, 1, color.name() );

            QComboBox *combobox = new QComboBox;
            for ( int j = 0; j < MAX_PATTERNS; ++j )
                combobox->insertItem( patterns[j] );
            combobox->setCurrentItem( element.valuePattern() - 1 );
            table->setCellWidget( i, 2, combobox );

            table->setText( i, 3, element.label() );

            color = element.labelColor();
            pix.fill( color );
            table->setPixmap( i, 4, pix );
            table->setText( i, 4, color.name() );
```

对于元素矢量中的每一个元素，我们必须填充表格。

如果元素是有效的，我们把它的值写在第一列（0 列，Value），根据指定的小数点位数进行格式化。

我们读元素的值颜色并用这种颜色填充空白图片，然后我们让颜色单元格显示这个图片。我们需要能够在以后读到这个颜色（比如用户改变了颜色）。一个方法就是测试图片中的一个像素，另一个就是继承 QTableItem（和我们继承 CanvasText 类似）并且在里面存储这个颜色。但是我们用了一个简单的方法：我们设置这个单元格的文本为这个颜色的名字。

接下来我们用样式来填充样式组合框。我们将通过使用被选择的样式在组合框中的位置来决定用户选择了哪一个样式。QTable 可以利用 QComboTableItem 条目，但是只支持文本，所以我们使用 setCellWidget()来代替把 QComboBox 的插入到表中。

接下来我们插入元素的标签。最后我们用我们设置值颜色的方法来设置标签颜色。

槽

```
void SetDataForm::currentChanged( int row, int col )
{
    colorPushButton->setEnabled( col == 1 || col == 4 );
    if ( col == 2 )
        ((QComboBox*)table->cellWidget( row, col ))->popup();
}
```

当用户进行定位时，表的 currentChanged()信号被发射。如果用户进入 1 或 4 列时（值颜色或标签颜色），我们让 colorPushButton 生效，否则让它失效。

为了给键盘用户提供方便，如果用户定位到样式组合框中时，我们弹出它。

```
void SetDataForm::valueChanged( int row, int col )
{
    if ( col == 0 ) {
        bool ok;
        double d = table->text( row, col ).toDouble( &ok );
        if ( ok && d > EPSILON )
            table->setText(
                row, col, QString( "%1" ).arg(
                            d, 0, 'f', m_decimalPlaces ) );
        else
            table->setText( row, col, table->text( row, col ) + "?" );
    }
}
```

如果用户改变值，我们必须使用正确的小数位数对它进行格式化，或者指出它是无效的。

```
void SetDataForm::setColor()
{
    setColor( table->currentRow(), table->currentColumn() );
    table->setFocus();
}
```

如果用户按下 Color 按钮，我们调用另一个 setColor()函数并把焦点返回到表中。

```
void SetDataForm::setColor( int row, int col )
{
    if ( !( col == 1 || col == 4 ) )
        return;

    QColor color = QColorDialog::getColor(
                        QColor( table->text( row, col ) ),
                        this, "color dialog" );
    if ( color.isValid() ) {
        QPixmap pix = table->pixmap( row, col );
        pix.fill( color );
        table->setPixmap( row, col, pix );
        table->setText( row, col, color.name() );
    }
```

```
}
```

如果当焦点在一个颜色单元格中时这个函数被调用，我们调用静态的 QColorDialog::getColor()对话框来获得用户所选择的颜色。如果他们选择了一个颜色，我们就用这种颜色来填充颜色单元格的图片，并且设置单元格的文本为新的颜色的名称。

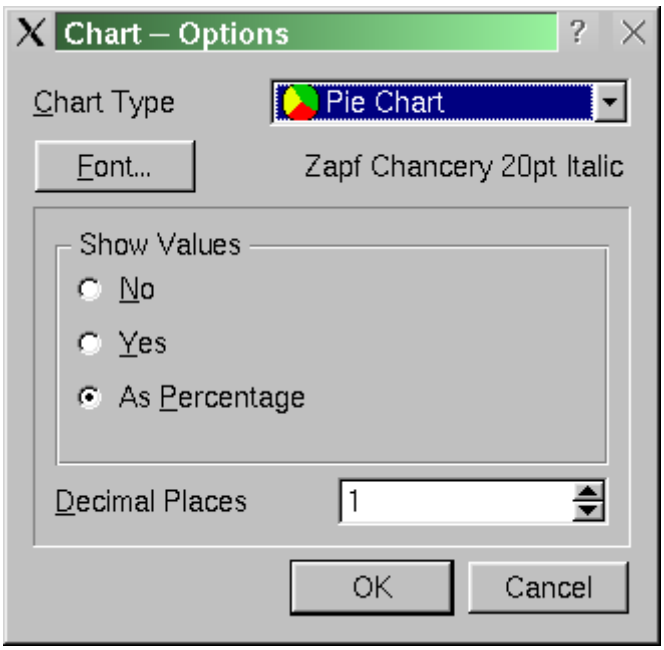
```
void SetDataForm::accept()
{
    bool ok;
    for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
        Element &element = (*m_elements)[i];
        double d = table->text( i, 0 ).toDouble( &ok );
        if ( ok )
            element.setValue( d );
        else
            element.setValue( Element::INVALID );
        element.setValueColor( QColor( table->text( i, 1 ) ) );
        element.setValuePattern(
                ((QComboBox*)table->cellWidget( i, 2 ))->currentItem() + 1 );
        element.setLabel( table->text( i, 3 ) );
        element.setLabelColor( QColor( table->text( i, 4 ) ) );
    }

    QDialog::accept();
}
```

如果用户点击 OK，我们必须更新元素矢量。我们对矢量进行迭代并把每一个元素的值设置为用户输入的值，否则如果值是无效的就设置为 INVALID。我们通过颜色的名称作为参数临时构造一个 QColor 来设置值颜色和标签颜色。样式被设置为样式组合框的当前条目与 1 的偏移量（因为我们的样式数字是从 1 开始的，但是组合框的条目是从 0 开始索引的）。

最后我们调用 QDialog::accept()。

« 文件处理 | 目录 | 设置选项 »

设置选项



我们提供了一个选项对话框，这样用户就可以在一个地方对所有的数据组设置选项。

（由 optionsform.h 展开。）

```
class OptionsForm : public QDialog
{
    Q_OBJECT
public:
    OptionsForm( QWidget* parent = 0, const char* name = "options form",
                 bool modal = FALSE, WFlags f = 0 );
    ~OptionsForm() {}

    QFont font() const { return m_font; }
    void setFont( QFont font );

    QLabel *chartTypeTextLabel;
    QComboBox *chartTypeComboBox;
    QPushButton *fontPushButton;
    QLabel *fontTextLabel;
    QFrame *addValuesFrame;
    QButtonGroup *addValuesButtonGroup;
    QRadioButton *noRadioButton;
    QRadioButton *yesRadioButton;
    QRadioButton *asPercentageRadioButton;
    QLabel *decimalPlacesTextLabel;
    QSpinBox *decimalPlacesSpinBox;
    QPushButton *okPushButton;
    QPushButton *cancelPushButton;
```

```
protected slots:
    void chooseFont();

protected:
    QVBoxLayout *optionsFormLayout;
    QHBoxLayout *chartTypeLayout;
    QHBoxLayout *fontLayout;
    QVBoxLayout *addValuesFrameLayout;
    QVBoxLayout *addValuesButtonGroupLayout;
    QHBoxLayout *decimalPlacesLayout;
    QHBoxLayout *buttonsLayout;

private:
    QFont m_font;
};
```

这个对话框的布局比设置数据视窗要更复杂一些，但是我们只需要一个单一的槽。不像“聪明的”设置数据视窗那样，这是一个“哑的”对话框，它只向窗口部件的调用者提供了读和写。调用者有责任基于用户所作的改变更新事物。

（由 optionsform.cpp 展开。）

```
#include "images/options_horizontalbarchart.xpm"
#include "images/options_piechart.xpm"
#include "images/options_verticalbarchart.xpm"
```

我们包含了一些在图表类型组合框中要使用的图片。

构造函数

```
OptionsForm::OptionsForm( QWidget* parent, const char* name,
                          bool modal, WFlags f )
    : QDialog( parent, name, modal, f )
{
    setCaption( "Chart -- Options" );
    resize( 320, 290 );
```

我们把所有的参数传递给 QDialog 构造函数，设置一个题目并且设置一个初始大小。

视窗的布局将是一个包含图表类型标签和组合框的水平盒子布局，并且对于字体按钮和字体标签、小数点位置标签和微调框也是相似的。按钮也会被放在一个水平布局中，但是还会有一个间隔来把它们移到右边。显示值的单选按钮将会竖直地排列在一个框架中。所有地这些都被放在一个竖直盒子布局中。

```
    optionsFormLayout = new QVBoxLayout( this, 11, 6 );
```

所有的窗口部件都被放在视窗的竖直盒子布局中。

```
    chartTypeLayout = new QHBoxLayout( 0, 0, 6 );
```

图表类型标签和组合框将被并排放置。

```
    chartTypeTextLabel = new QLabel( "&Chart Type", this );
    chartTypeLayout->addWidget( chartTypeTextLabel );
```

```
        chartTypeComboBox = new QComboBox( false, this );
        chartTypeComboBox->insertItem( QPixmap( options_piechart ), "Pie Chart" );
        chartTypeComboBox->insertItem( QPixmap( options_verticalbarchart ),
                                       "Vertical Bar Chart" );
        chartTypeComboBox->insertItem( QPixmap( options_horizontalbarchart ),
                                       "Horizontal Bar Chart" );
        chartTypeLayout->addWidget( chartTypeComboBox );
        optionsFormLayout->addLayout( chartTypeLayout );
```

我们创建图表类型标签（带有一个加速键，稍后我们会把它和图表类型组合框联系起来）。我们也创建一个图表类型组合框，用图片和文本来填充它。我们把它们两个添加到水平布局中，并把水平布局添加到视窗的竖直布局中。

```
        fontLayout = new QHBoxLayout( 0, 0, 6 );

        fontPushButton = new QPushButton( "&Font...", this );
        fontLayout->addWidget( fontPushButton );
        QSpacerItem* spacer = new QSpacerItem( 0, 0,
                                               QSizePolicy::Expanding,
                                               QSizePolicy::Minimum );
        fontLayout->addItem( spacer );

        fontTextLabel = new QLabel( this ); // 必须由调用者通过 setFont()来设置
        fontLayout->addWidget( fontTextLabel );
        optionsFormLayout->addLayout( fontLayout );
```

我们创建一个水平盒子布局用来保存字体按钮和字体标签。字体按钮是被直接加入的。我们添加了一个间隔用来增加效果。字体文本标签被初始化为空（因为我们不知道用户正在使用什么字体）。

```
        addValuesFrame = new QFrame( this );
        addValuesFrame->setFrameShape( QFrame::StyledPanel );
        addValuesFrame->setFrameShadow( QFrame::Sunken );
        addValuesFrameLayout = new QVBoxLayout( addValuesFrame, 11, 6 );

        addValuesButtonGroup = new QButtonGroup( "Show Values", addValuesFrame );
        addValuesButtonGroup->setColumnLayout(0, Qt::Vertical );
        addValuesButtonGroup->layout()->setSpacing( 6 );
        addValuesButtonGroup->layout()->setMargin( 11 );
        addValuesButtonGroupLayout = new QVBoxLayout(
                                         addValuesButtonGroup->layout() );
        addValuesButtonGroupLayout->setAlignment( Qt::AlignTop );

        noRadioButton = new QRadioButton( "&No", addValuesButtonGroup );
        noRadioButton->setChecked( true );
        addValuesButtonGroupLayout->addWidget( noRadioButton );
```

```
yesRadioButton = new QRadioButton( "&Yes", addValuesButtonGroup );
addValuesButtonGroupLayout->addWidget( yesRadioButton );

asPercentageRadioButton = new QRadioButton( "As &Percentage",
                                            addValuesButtonGroup );
addValuesButtonGroupLayout->addWidget( asPercentageRadioButton );
addValuesFrameLayout->addWidget( addValuesButtonGroup );
```

用户也许选择显示它们自己的标签或者在每一个标签的末尾加上值，或者加上百分比。

我们创建一个框架来存放单选按钮并且为它们创建了一个布局。我们创建了一个按钮组（这样 Qt 就可以自动地处理专有的单选按钮行为了）。接下来我们创建单选按钮，并把“No”作为默认值。

小数位标签和微调框被放在另一个水平布局中，并且按钮和设置数据视窗中的按钮的排布方式非常相似。

```
connect( fontPushButton, SIGNAL( clicked() ), this, SLOT( chooseFont() ) );
connect( okPushButton, SIGNAL( clicked() ), this, SLOT( accept() ) );
connect( cancelPushButton, SIGNAL( clicked() ), this, SLOT( reject() ) );
```

我们只需要三个连接：

1. 当用户点击字体按钮时，我们执行我们自己的 chooseFont()槽。
2. 如果用户点击 OK，我们调用 QDialog::accept()，它会让调用者来从对话框的窗口部件中读取数据并且执行任何必要的动作。
3. 如果用户点击 Cancel，我们调用 QDialog::reject()。

```
chartTypeTextLabel->setBuddy( chartTypeComboBox );
decimalPlacesTextLabel->setBuddy( decimalPlacesSpinBox );
```

我们使用 setBuddy()函数来连接窗口部件和标签的加速键。

槽

```
void OptionsForm::chooseFont()
{
    bool ok;
    QFont font = QFontDialog::getFont( &ok, m_font, this );
    if ( ok )
        setFont( font );
}
```

当用户点击 Font 按钮时，这个槽被调用。它简单地调用静态的 QFontDialog::getFont()来获得用户选择的字体。如果他们选择了一个字体，我们调用我们的 setFont()槽在字体标签中提供一个字体的文本描述。

```
void OptionsForm::setFont( QFont font )
{
    QString label = font.family() + " " +
                    QString::number( font.pointSize() ) + "pt";
```

```
        if ( font.bold() )
            label += " Bold";
        if ( font.italic() )
            label += " Italic";
        fontTextLabel->setText( label );
        m_font = font;
    }
```

这个函数在字体标签中显示一个被选字体的文本描述，并且在 m_font 成员中保存一个字体的拷贝。我们需要这个字体为成员，这样我们就会为 chooseFont()提供一个默认字体。

« Taking Data | 目录 | 项目文件 »

项目文件

（chart.pro。）

```
TEMPLATE = app
CONFIG  += warn_on

HEADERS +=  element.h \
            canvastext.h \
            canvasview.h \
            chartform.h \
            optionsform.h \
            setdataform.h
SOURCES +=  element.cpp \
            canvasview.cpp \
            chartform.cpp \
            chartform_canvas.cpp \
            chartform_files.cpp \
            optionsform.cpp \
            setdataform.cpp \
            main.cpp
```

通过使用项目文件，我们能够把我们自己从为我们所要使用的平台创建 Makefile 中脱离出来。为了生成一个 Makefile 我们所要做的一切就是运行 qmake，比如：

```
qmake -o Makefile chart.pro
```

« 设置选项 | 目录 | 完成 »

完成

chart 应用程序显示了用 Qt 创建应用程序和对话框是多么的直接。创建菜单和工具条是很容易的并且 Qt 的信号和槽机制相当地简化了图形用户界面事件处理。

手工创建布局可能会花费一些时间来掌握，但是有一种容易的选择：Qt 设计器。Qt 设计器包括了简单但强大的布局工具和一个代码编辑器。它可以自动地生成 main.cpp 和.pro 项目文件。
chart 应用程序对于进一步开发和实验是成熟的。你也许可以考虑实现下面的一些思路：

- 使用 QValidator 子类来保证只有有效的双精实数被输入到值中。
- 添加更多的图表类型，比如线图、区域图和高低图。
- 允许用户设置上下左右边白。
- 允许用户指定一个可以像标签一样拖拽的标题。
- 提供一个绘制和标签选项的中心线。
- 提供一个选项用键（或者图例）来替换标签。
- 为所有的图表类型添加一个三维的查看选项。