Qt 教程一 —— 第一章：Hello, World!

第一个程序只包含你建立和运行 Qt 应用程序所需要的最少的代码。

```
#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);

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

a.setMainWidget(&hello);

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

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。

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：

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！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。

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。

```
#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 Application::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。

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。

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。

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。

```
#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()）。

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。）

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？

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。

```
#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。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？

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。

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）。

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。

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。

```
#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。

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。

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？

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 somethingChanged()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”。

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）。

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？

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。

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。

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。

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。

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？

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。

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!

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.

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。

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。

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。

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－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
                = newQPushButton( "&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
                = newQLabel( "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。

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。

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。

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。

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。

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。