

第二章 Hello World

作为一个初学者，安装完 Qt 后第一件事，当然 Hello World 一下，通常介绍编程的教科书都是从 Hello World 开始的，我不知道如果打破这个传统会带来什么后果，我现在还没有勇气去做第一个吃螃蟹的人。如果你不是第一次接触 Qt，可以跳过本章节。下面用两个经典的示例来写讲述 Hello World。

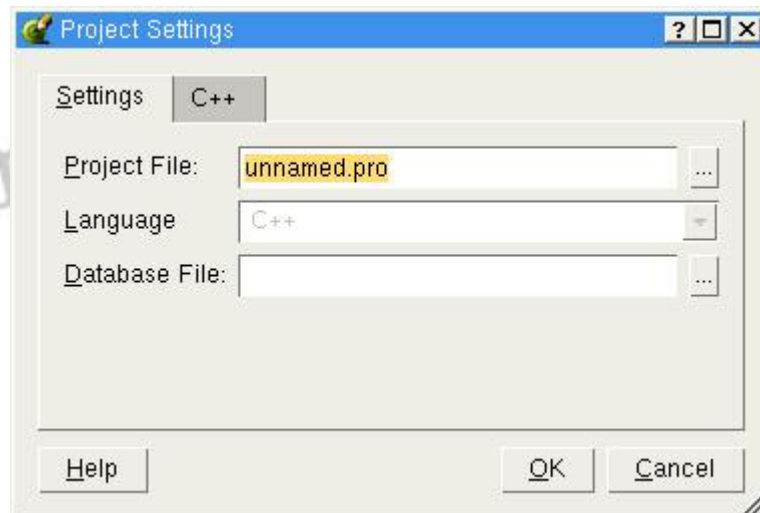
2.1 从两个例子开始

示例一：

运行 Qt Designer，点击菜单 File->new，新建一个项目，



这里选择 C++ Project，确定。接下来会提示项目保存位置，



选择保存路径和文件名，确定，然后，点击菜单 File->New, 选择 C++ Source File, 确定，录入以下内容：

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

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

    QPushButton hellobtn( "Hello world!", 0 );
    hellobtn.resize(100, 20 );
    a.setMainWidget( &hellobtn );
    hellobtn.show();
    return a.exec();
}
```

代码说明：

```
#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 hellobtn( "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）。

```
    hellobtn.resize( 100, 20 );
```

这个按钮被设置成 100 像素宽，20 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( &hellobtn );
```

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。

```
hellobtn.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 hello.pro
```

第一个命令调用 `qmake` 来生成一个 `.pro`（项目）文件，运行后会生成一个 `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！

```
qmake hello.pro
make
./hello
```

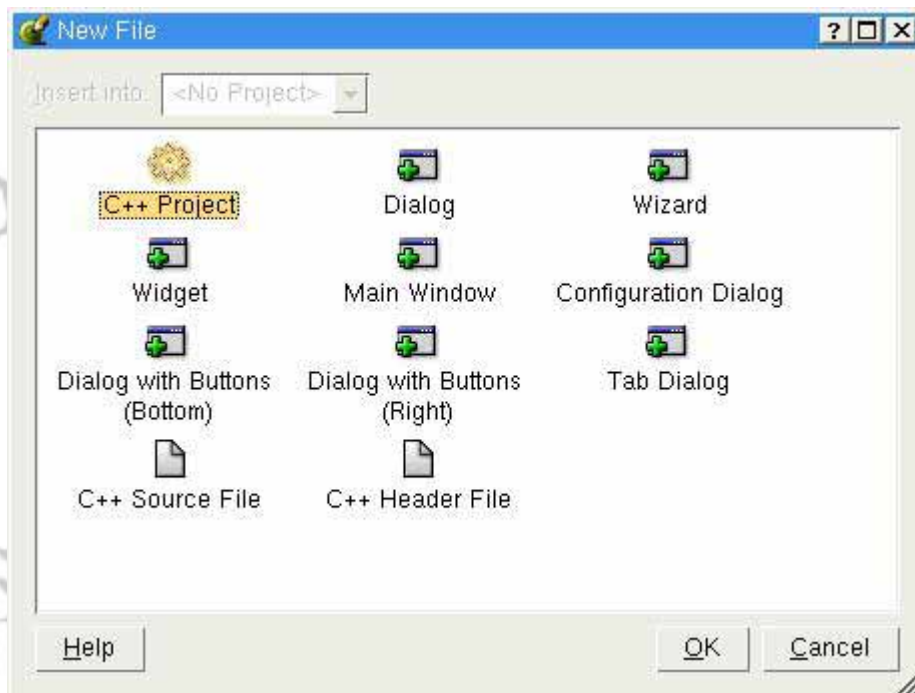
运行后显示窗口



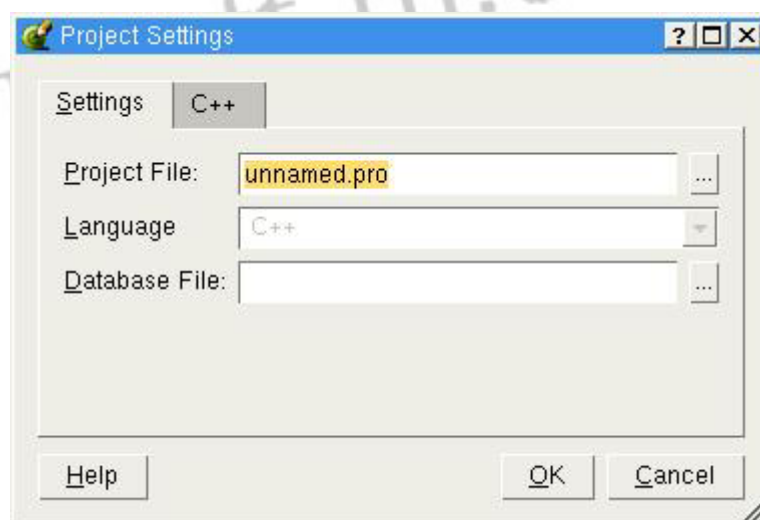
其实使用 Qt Designer 完全可以生成上面的代码，下面我们再看一个例子，

示例二：

运行 Qt Designer，点击菜单 `File->new`，新建一个项目，

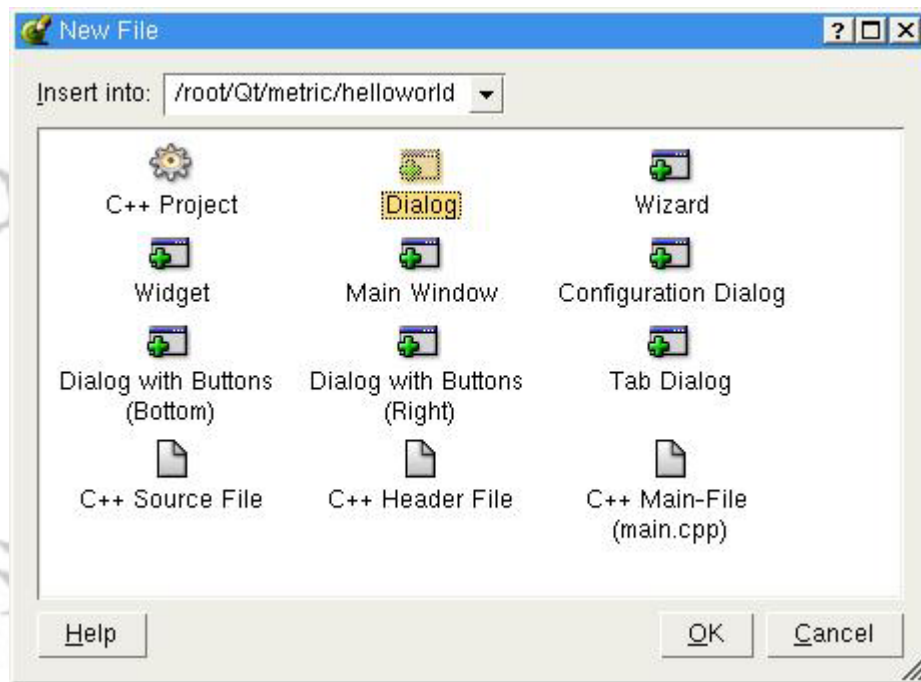


这里选择 C++ Project，确定。接下来会提示项目保存位置，



选择保存路径和文件名，确定。

选择菜单 File->new，新建一个窗口，选择 Dialog，确定



设置 Form1 的 Caption 为“我的程序”，在 Property Editor 设置窗口属性，如果你的 IDE 上看不到 Property Editor，请通过菜单 Windows->Views，将 Property Editor/Signal Handlers 选上。

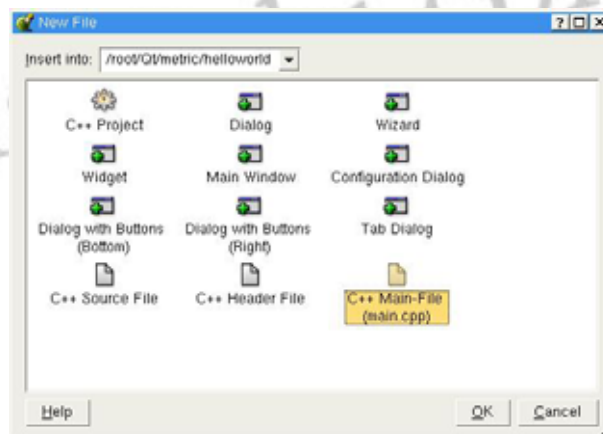
接着在窗口上放一个 TextLabel，选择 Toolbox 上的 Common Widgets 上的 TextLabel，再添加一个按钮，PushButton，



将 textLabel1 的 Text 设为“Hello World”，pushButton1 的 Text 设为“关闭”。接下来为关闭按钮添加事件，选择菜单 Edit->Connections...，弹出 View and Edit Connections 窗口，



点击按钮“New”在新增项中，Sender 选择 pushButton1，Signal 选择 Clicked()，Receiver 选择 Form1，Slot 选择 Close()，确定然后保存，



到这里，基本就快完成我们的 Hello World 了，我们还需要一个 main.cpp 文件，点击菜单 File->new，选择 C++ main file(main.cpp)



文件名 filename:main.cpp，程序主窗 Main-Form:Form，确定，Qt 会自动生成 main.cpp 文件代码，

```
#include <qapplication.h>
#include "form1.h"

int main( int argc, char ** argv )
{
    QApplication a( argc, argv );
    Form1 w;
    w.show();
    a.connect( &a, SIGNAL( lastWindowClosed() ), &a, SLOT( quit() ) );
    return a.exec();
}
```

好了，到这里我们的第二个 Hello World 程序设计完了。保存项目，然后编译程序：

```
qmake helloworld.pro
make
./helloworld
```

如果您保存的项目名称是 helloworld.pro 就可以用上面的命令编译了。Qmake helloworld.pro 生成 Makefile，make 开始编译程序，编译完后我们就可以运行我们的程序了。./helloworld 运行，



点击“关闭”按钮，关闭程序。

2.2 本章小节

本章通过两个简单例子带你走入 Qt 世界，如果你已经是位 C/C++ 程序员，你会发现 Qt 将会非常容易上手，本章的二个简单的例子，讲述了如何使用 Qt Designer 来快速的建立应用程序。其中第二个例子中介绍了如何新建窗口，添加组件，并如何设置它们。上面的两个例子，主要使用了 Qt 以下两个类(Class)：

QApplication 类

QApplication 类管理图形用户界面应用程序的控制流和主要设置，

公有成员

```
QApplication ( int & argc, char ** argv )
QApplication ( int & argc, char ** argv, bool GULenabled )
enum Type { Tty, GuiClient, GuiServer }
QApplication ( int & argc, char ** argv, Type type )
QApplication ( Display * dpy, HANDLE visual = 0, HANDLE colormap = 0 )
QApplication ( Display * dpy, int argc, char ** argv, HANDLE visual = 0,
HANDLE colormap = 0 )
virtual ~QApplication ()
int argc () const
char ** argv () const
Type type () const
enum ColorSpec { NormalColor = 0, CustomColor = 1, ManyColor = 2 }
QWidget * mainWindow () const
virtual void setMainWindow ( QWidget * mainWindow )
virtual void polish ( QWidget * w )
QWidget * focusWidget () const
QWidget * activeWindow () const
int exec ()
void processEvents ()
void processEvents ( int maxtime )
void processOneEvent ()
bool hasPendingEvents ()
int enter_loop ()
```

```
void exit_loop ()
int loopLevel () const
virtual bool notify ( QObject * receiver, QEvent * e )
void setDefaultCodec ( QTextCodec * codec )
QTextCodec * defaultCodec () const
void installTranslator ( QTranslator * mf )
void removeTranslator ( QTranslator * mf )
enum Encoding { DefaultCodec, UnicodeUTF8 }
QString translate ( const char * context, const char * sourceText, const
char * comment = 0, Encoding encoding = DefaultCodec ) const
virtual bool macEventFilter ( EventRef )
virtual bool winEventFilter ( MSG * )
virtual bool x11EventFilter ( XEvent * )
int x11ProcessEvent ( XEvent * event )
virtual bool qwsEventFilter ( QWSEvent * )
void qwsSetCustomColors ( QRgb * colorTable, int start, int numColors )
void winFocus ( QWidget * widget, bool gotFocus )
bool isSessionRestored () const
QString sessionId () const
virtual void commitData ( QSessionManager & sm )
virtual void saveState ( QSessionManager & sm )
void wakeUpGuiThread ()
void lock ()
void unlock ( bool wakeUpGui = TRUE )
bool locked ()
bool tryLock ()
```

公有槽

```
void quit ()
void closeAllWindows ()
```

相关函数

```
void qAddPostRoutine ( QtCleanupFunction p )
const char * qVersion ()
bool qSysInfo ( int * wordSize, bool * bigEndian )
void qDebug ( const char * msg, ... )
void qWarning ( const char * msg, ... )
void qFatal ( const char * msg, ... )
void qSystemWarning ( const char * msg, int code )
void Q_ASSERT ( bool test )
void Q_CHECK_PTR ( void * p )
QtMsgHandler qInstallMsgHandler ( QtMsgHandler h )
```

它包含主事件循环,在其中来自窗口系统和其它资源的所有事件被处理和调度。它也处理应用程序的初始化和结束,并且提供对话管理。它也处理绝大多数系统范围和应用程序范围的设置。

对于任何一个使用 Qt 的图形用户界面应用程序，都正好存在一个 QApplication 对象，而不论这个应用程序在同一时间内是不是有 0、1、2 或更多个窗口。

QApplication 对象是可以通过全局变量 qApp 访问。它的负责的主要范围有：

它使用用户的桌面设置，例如 palette()、font() 和 doubleClickInterval() 来初始化应用程序。如果用户改变全局桌面，例如通过一些控制面板，它会对这些属性保持跟踪。

它执行事件处理，也就是说它从低下的窗口系统接收事件并且把它们分派给相关的窗口部件。通过使用 sendEvent() 和 postEvent()，你可以发送你自己的事件到窗口部件。

它分析命令行参数并且根据它们设置内部状态。

它定义了由 QStyle 对象封装的应用程序的观感。在运行状态下，可以通过 setStyle() 来改变。

它指定了应用程序如何分配颜色。参考 setColorSpec()。

它定义了默认文本编码（请参考 setDefaultCodec()）并且提供了通过 translate() 用户可见的本地化字符串。

它提供了一些像 desktop() 和 clipboard() 这样的魔术般的对象。

它知道应用程序的窗口。你可以使用 widgetAt() 来询问在一个确定点上存在哪个窗口部件，得到一个 topLevelWidgets()（顶级窗口部件）的列表和通过 closeAllWindows() 来关闭所有窗口，等等。

它管理应用程序的鼠标光标处理，参考 setOverrideCursor() 和 setGlobalMouseTracking()。

在 X 窗口系统上，它提供刷新和同步通讯流的函数，可以参考 flushX() 和 syncX()。

它提供复杂的对话管理支持。这使得当用户注销时，它可以让应用程序很好地结束，如果无法终止，撤消关闭进程并且甚至为未来的对话保留整个应用程序的状态。

QPushButton 类

QPushButton 主要用于命令按钮，

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

继承了 QButton。

公有成员

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

- ✧ QPushButton (const QString & text, QWidget * parent, const char * name = 0)
- ✧ QPushButton (const QIconSet & icon, const QString & text, QWidget * parent, const char * name = 0)
- ✧ ~QPushButton ()
- ✧ void setToggleButton (bool)
- ✧ bool autoDefault () const
- ✧ virtual void setAutoDefault (bool autoDef)
- ✧ bool isDefault () const
- ✧ virtual void setDefault (bool def)
- ✧ virtual void setIsMenuButton (bool enable) (废弃)
- ✧ bool isMenuButton () const (废弃)
- ✧ void setPopup (QPopupMenu * popup)
- ✧ QPopupMenu * popup () const
- ✧ void setIconSet (const QIconSet &)
- ✧ QIconSet * iconSet () const
- ✧ void setFlat (bool)
- ✧ bool isFlat () const

公有槽

virtual void setOn (bool)

属性

- bool autoDefault - 推动按钮是否是自动默认按钮
- bool autoMask - 按钮中自动面具特征是否有效 (只读)
- bool default - 推动按钮是否是默认按钮
- bool flat - 边缘是否失效
- QIconSet iconSet - 推动按钮上的图标
- bool menuButton - 推动按钮是否有一个菜单按钮在上面 (废弃)
- bool on - 推动按钮是否被切换
- bool toggleButton - 按钮是不是切换按钮

推动按钮或者命令按钮或许是任何图形用户界面中最常用到的窗口部件。推动(点击)按钮来命令计算机执行一些操作, 或者回答一个问题。典型的按钮有确定 (OK)、应用 (Apply)、撤销 (Cancel)、关闭 (Close)、是 (Yes)、否 (No) 和帮助 (Help)。

命令按钮是矩形的并且通常显示一个文本标签来描述它的操作。标签中有下划线的字母 (在文本中它的前面被 “&” 标明) 表明快捷键, 例如:

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

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