Programming KDE - A Primer

Sirtaj Singh Kang <taj@kde.org>

Conference of Australian Linux Users, Melbourne, Australia

July 1999

What is KDE?

- A framework for Graphical User Applications for UNIX and the X Window System
- Brings consistency to look and feel and functionality
- Why do I want to program for KDE?
  - Libraries provide excellent functionality and are well documented. All KDE libraries are supplied under the GNU Library GPL.
  - Fills in many feature-holes in available application frameworks
    - All applications respond to look and feel changes immediately with almost zero programmer effort
    - Drag and drop
    - HTML-based documentation system
    - Internationalization support - KDE already supports over 30 languages
    - Session management built in to the system - applications are notified when the user wishes to log out, and can save its state. State is restored on next login
    - Easy non-blocking network file transfers, without threads
    - Easy-to-use three-level (Global/User/Application) configuration system
    - High-level C++ access to all library features allows quick development of applications
    - GNU Autoconf/Automake frameworks for source portability to other UNIXes.

What do I need to know to program for KDE?

- You need to know a little C++ to begin, but more C++ experience will help with more advanced features (templates and exceptions, for example)
- A general idea of Object Oriented (OO) programming, eg classes, inheritance and programming to interfaces
- You should know how to program in the UNIX environment - using text editors and compilers (and sometimes debuggers!)

What tools do I need?

- A system running UNIX and X11
- An ANSI-compliant C++ compiler - all KDE development is done with the EGCS compiler
- The Qt GUI toolkit
- KDE libraries
- The KDE Software Development Kit (optional)
    - Example applications
    - Application framework generator - generates a generic KDE application which you can customize and add your specific features.
    - Documentation tools
        - KDOC to generate documentation for your source.
        - LinuxDoc SGML template for generation of user-level documentation.
- KLyX - for easy authoring of user documentation in the LinuxDoc SGML format.

What is Qt? Why do we use it in KDE?

- Qt is a high-level, cross-platform toolkit for writing GUI applications with C++
- Very good, consistent design with many very useful features
  - Callbacks using the signal/slot system for Event/Observer programming
  - Integrated printing support for PostScript output
  - Easy development of applications that use OpenGL (or Mesa)
  - Many generally useful classes for C++ programming - for example, efficient containers and string class
  - Complete documentation, tutorials and examples available in HTML and printable formats
  - Very reliable and now in second generation (version 2 was released a few weeks ago)
  - Available under an open source license for open source development under the X Window system, but a license must be purchased for commercial development and for the Windows version.

Qt Programming Basics

- All Qt-based programs have a single QApplication object, which handles application-wide information and the event queue.
- On-screen user interface elements are known as “Widgets” (derived from the QWidget class) and can be nested inside each other.
- The QPainter class allows drawing inside a widget.
- Processing of events (eg resize, redraw, or mouse events) can be done by overriding virtual functions in Qwidget.
- Compiling Qt applications that use signals and slots (most useful programs do) requires the use of a simple preprocessor called moc. More about this later.

Hello, World - Our first Qt Program

Here is the canonical “first program”. It demonstrates the basic style and ideas of a Qt-based application.

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


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

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

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

    hello.show();
    return a.exec();
}
```

Listing 1 - hello.cpp

Here is what we get when we run this program:

Figure 1-Output of hello.cpp

- What do we see here?
  - Qt header files are the named after the classes they declare, but in lower case.
  - We have created a QApplication object.
  - We have created a push-button widget. Since it has no parent, it appears as a top-level window with window-manager decorations.
  - We resize the push-button to a fixed size, and connect it's clicked() signal (as published in the class interface) to the quit() slot of the QApplication object. When the clicked signal is emitted by the push-button, all connected slots (which are functions) are called.
  - We make the push-button the main widget of the application. This ensures that the application exits if the push-button is destroyed.
  - The push-button is displayed. This does not actually appear on screen till the next line is executed, where...
  - ... we enter the application's main event loop. This will be exited when the push-button is closed (perhaps by clicking the close button in the title bar).

When there is more than one top-level widget in an application, we cannot set any one to be the main widget in the QApplication, it is more handy to call the QApplication::quit() method, which closes all windows and exits the event loop.

Hello again, World

In **Listing 2**, we see the next version of our hello world program. This one creates a custom button that has it's own drawing function, which uses QPainter to draw the button text in a resize-able ellipse.

Note that the drawButtonLabel function is virtual, and is called from within Qt whenever the button needs to be redrawn. In this way, we can customize any Qt widget simply by overriding the virtual functions declared in QWidget.

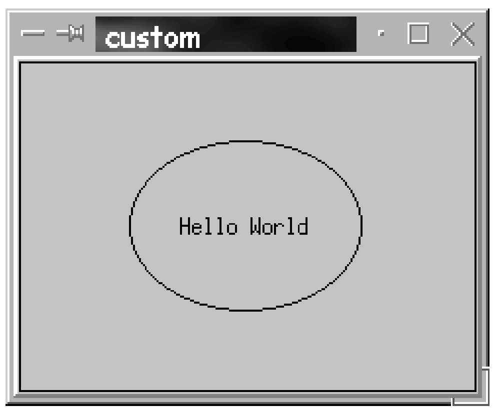

Figure 2 – Custom Button

The KDE Libraries

The classes that comprise the KDE API are split into various libraries based on the type of functionality they offer. The libraries in KDE 1.x are:

- **KDECORE - Core Library**
  - Classes that provide basic application functionality, including
  - **KApplication** - derived from QApplication, this provides additional information and features, including session management, and desktop-wide configuration change notification (style changes etc)
  - **KConfig family** - allows applications to load and save settings in human-readable text files
  - **KURL** - provides easy manipulation of WWW urls
  - **KProcess** - allows initiation and control of processes
  - **KWM** - provides communication with and control of KDE-compliant window managers.
  - Various other classes, including socket IO and keybinding configuration.
- **KDEUI** - User Interface Library
  - Classes that are used to build user interfaces for KDE applications, including **KTMainWindow**, which provides easy management of windows with menubars, toolbars and statusbars.
- **LIBKFM** - File Management Library
  - Communicates with the KDE file manager to provide file asynchronous file management and network file transfers.
- **KHTMLW** - HTML Widget Library
  - Provides a fast HTML widget, which is used by the KDE help

system and the KDE file manager/browser.

- **KIMGIO** - Image IO Library
  - Extends KDE's image manipulation abilities with reading and writing of several image formats including JPEG, PNG and TIFF.
- **KFILE** - Network-aware File Dialog Library
  - Provides a file dialog that can accept and provide URLs, allowing browsing of remote file repositories via HTTP and FTP (and even inside tar.gz files!).

Kpad - A text editor

Kpad (**Listing 3**) is a simple text editor that demonstrates the KDE libraries and more of Qt. It demonstrates:

- Use of the KTMainWindow class
- Declaration of slots
- Use of the editor widget QMultiLineEdit and menu classes.
- Making your code ready for internationalization with the i18n() function.
- Use of the KFileDialog and KMsgBox classes
- Use of the QFile and QString utility classes.

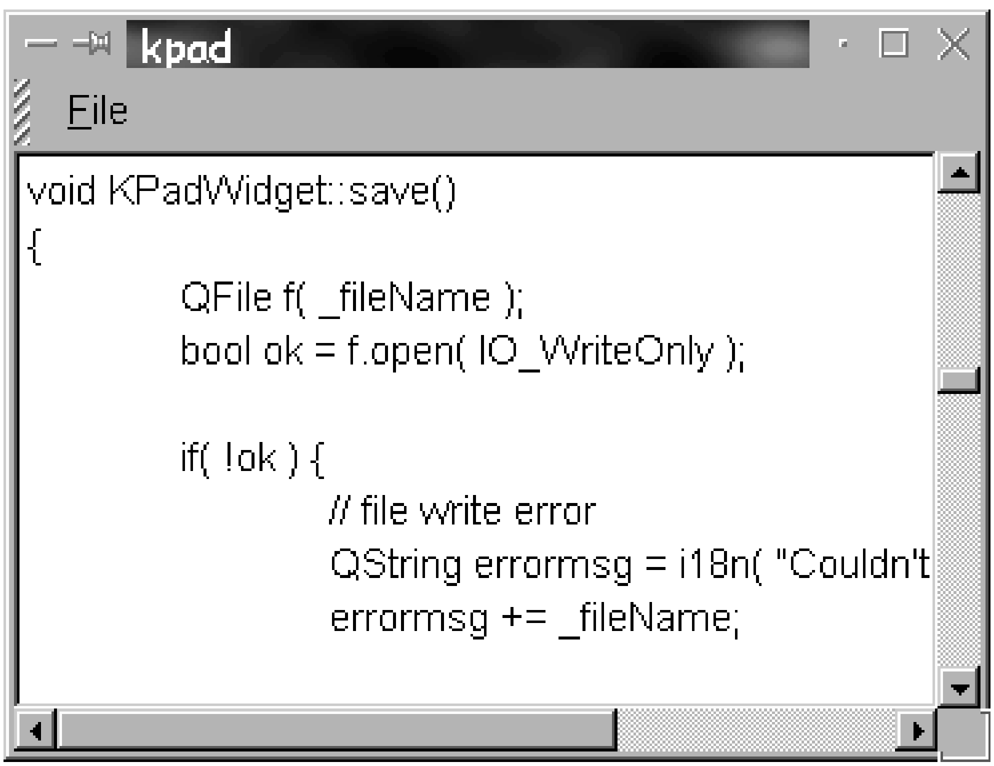

Figure 3 - Kpad editing its own source

Listing 2 - custom.cpp

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


/**
     A little custom button
*/
class MyHelloButton : public QPushButton
{
     // This is required to use signals and slots
     Q_OBJECT

public:
     // constructor and destructor
     MyHelloButton( QWidget *parent = 0 ) : QPushButton ( parent
) {}
     virtual ~MyHelloButton() {}

     virtual void drawButtonLabel( QPainter * );
};

void MyHelloButton::drawButtonLabel( QPainter *painter )
{
     QRect r;

     r.setX( width() / 4 );
     r.setY( height() / 4 );
     r.setWidth( width() / 2 );
     r.setHeight( height() / 2 );

     painter->drawEllipse( r );
     painter->drawText( r, AlignCenter, text() );
}


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

    MyHelloButton hello;
    hello.setText( "Hello World" );

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

```
    hello.show();
    return a.exec();
}

#include"custom.moc"
```

Listing 3 - kpad.cpp

```
#include<ktmainwindow.h>      // For KTMainWindow
#include<kapp.h>              // For KApplication
#include<kmsgbox.h>           // For KMsgBox
#include<kfiledialog.h>            // For KFileDialog

#include<qfile.h>             // For QFile
#include<qstring.h>           // For QString
#include<qmultilinedit.h>          // For QMultiLineEdit
#include<qkeycode.h>               // For key codes

class KPadWidget : public KTMainWindow
{
     Q_OBJECT

public:
     // constructor and destructor
     KPadWidget();
     virtual ~KPadWidget() {}

public slots:
     void save();
     void load();
     void quit();

protected slots:
     void setDirty();

private:
     QPopupMenu     *_fileMenu;
     QMultiLineEdit *_editor;

     QString   _fileName;
     bool _textChanged;
};

KPadWidget::KPadWidget()
     : KTMainWindow(),
     _editor( new QMultiLineEdit( this ) ),
     _fileName( "Untitled" ),
     _textChanged( false )
{
     // set up the editor widget
     setView( _editor );
     connect( _editor, SIGNAL(textChanged()), this,
          SLOT(setDirty()) );

     // set up the popup menu
```

```
    _fileMenu = new QPopupMenu;
    _fileMenu->insertItem( i18n( "&Open" ), this, SLOT(load()),
         CTRL + Key_O );
    _fileMenu->insertItem( i18n( "&Save" ), this, SLOT(save()),
         CTRL + Key_S );
    _fileMenu->insertSeparator();
    _fileMenu->insertItem( i18n( "E&xit" ), this, SLOT(quit()),
         CTRL + Key_X );


    // set up the menubar and insert the popup menu
    KMenuBar *menubar = menuBar();
    menubar->insertItem( i18n( "&File" ), _fileMenu );
}

void KPadWidget::save()
{
    QFile f( _fileName );
    bool ok = f.open( IO_WriteOnly );

    if( !ok ) {
         // file write error
         QString errormsg = i18n( "Couldn't write to File:" );
         errormsg += _fileName;

         KMsgBox::message( 0, i18n( "File Write Error" ),
errormsg );

         return;
    }

    QString text = _editor->text();
    f.writeBlock( text, text.length() );
    f.close();

    _textChanged = false;
}

void KPadWidget::load()
{
    QString name = KFileDialog::getOpenFileName();

    if( name.isEmpty() ) {
         // no file was selected.
         return;
    }

    QFile f( name );
    bool ok = f.open( IO_ReadOnly );

    if( !ok ) {
         QString errormsg = i18n( "Couldn't Read File:" );
```

```
          errormsg += name;

          KMsgBox::message( 0, i18n( "File Read Error" ),
errormsg );

          return;
     }

     QString buffer( 1024 );

     _editor->setText( "" );

     while( !f.atEnd() ) {
          f.readBlock( buffer.data(), 1024 );
          _editor->append( buffer );
     }

     f.close();
     _fileName = name;
     _textChanged   = false;
}

void KPadWidget::quit()
{
     if( _textChanged ) {
          int result = KMsgBox::yesNoCancel( 0,
               i18n( "File has changed" ),
               i18n( "Save file before exit?" ) );

          switch ( result ) {
               case 1:        save();        // Yes
                         break;

               case 2:        break;         // No

               case 3:        return;        // Cancel

               default:  break;
          }
     }

     kapp->quit();
}

void KPadWidget::setDirty()
{
     _textChanged = true;
}


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

```
    KApplication app( argc, argv );

    KPadWidget *editor = new KPadWidget;

    app.setMainWidget( editor );

    editor->show();

    return app.exec();
}

#include"kpad.moc"
```

Listing 4 - Project Makefile

This Makefile displays the use of the moc preprocessor, and is customized for a system running Debian GNU/Linux.

```
CFLAGS = -I /usr/lib/qt1g/include -fno-rtti -Wall -W -ansi \
     -pedantic -L/usr/lib/qt1g/lib -I /usr/include/kde \
     -L /usr/X11R6/lib

all: hello custom kpad

hello: hello.cpp
     g++ $(CFLAGS) -o $@ $< -lqt

custom: custom.cpp custom.moc
     g++ $(CFLAGS) -o $@ $< -lqt

custom.moc: custom.cpp
     moc custom.cpp -o custom.moc

kpad:     kpad.cpp  kpad.moc
     g++ $(CFLAGS) -o $@ $< -lkfile -lkfm -lkdeui -lkdecore -lqt

kpad.moc: kpad.cpp
     moc kpad.cpp -o kpad.moc
```

Appendix A - Sources of Information

Qt Programming

- Official Troll Tech web site: http://www.troll.no/
- The Qt distribution comes with full documentation in HTML and PostScript formats.
- From the Troll Tech web page, you can obtain information about the qt-interest mailing list, which is frequented by many seasoned Qt developers and Trolls.
- Kalle Dalheimer <kalle@kde.org> has written a book on Qt programming, published by O'Reilly.

KDE Programming

- The official KDE web site: http://www.kde.org/
- Main KDE developer site: http://developer.kde.org/
- My pages of information for KDE developers, including KDE API documentation:

http://www.ph.unimelb.edu.au/~ssk/kde/devel/

- The kde-devel mailing list is used by developers of KDE applications.
- The code presented in this tutorial will be made available from: http://www.ph.unimelb.edu.au/~ssk/kde/CALU/