1 Programming the User Interface II

1.1 Creating Graphical Interfaces

- When it comes to creating GUIs, Linux programmers have more options available than they do for creating TUIs. Probably the most popular and certainly the best known toolkits used to create graphical applications are Qt and GTK+.
  - Qt is the C++ application framework that powers KDE, the K Desktop Environment.
  - GTK+ is the toolkit underneath GNOME, the GNU Network Object Model Environment. GTK+ is written largely in C, but it has language bindings available for many other programming languages, such as Perl, C++, and Python, so you can use GTK+ features in many programming environments.
- There are many other toolkits, frameworks, and libraries that you can use to develop GUIbased applications for Linux. The following list, arranged alphabetically, describes some of the most common ones. Most of these toolkits and frameworks describe widget sets, which are implemented in one or more programming libraries. Widget is the term applied to a user interface abstraction, such as a scrollbar or a button, created using the toolkit.
  - *Athena*; The Athena library was one of the earliest widget libraries available for the X Window System. It was a thin layer of abstraction on top of raw Xlib calls that made it slightly less painful to create scrollbars, text entry boxes, and other typical GUI elements. It is part of the standard X11 distribution.
  - *3-D Athena Toolkit*; The 3D Athena Toolkit was a 3D version of the original Athena toolkit. It gave Athena a 3D look and was a considerable visual improvement over plain vanilla Athena. The 3D Athena toolkit, although no longer widely used.
  - *FLTK*; FLTK, which is pronounced "full tick" is an acronym for the Fast Light Toolkit. FLTK is a GUI for X, Mac OS X, and Microsoft Windows. Written in C++, FLTK makes it possible to write GUIs that look almost identical regardless of the platform on which the GUI runs. FLTK also supports OpenGL graphics.
  - *XForms*; XForms is a GUI toolkit based on Xlib. It isn't highly configurable like the other GUI toolkits discussed in this section,

but its simplicity makes XForms easier to use than the other graphical toolkits. It comes with a GUI builder that makes it fast and easy to get working application up and running.

- *OpenGL*; OpenGL is the industry-standard 3D graphics toolkit. It provides the most realistic and lifelike graphics currently available for the X Window System. It is generally available as part of XFree86.
- *Motif*; Motif was one of the first widget or interface toolkits available for the X Window System that combined both an interface toolkit and a window manager. Originally available only as a commercial product, it is now available in an open source version.
- *Xlib*; Xlib is shorthand for the X library, a low-level, C-based interface to the raw X Window System protocol. If you want to write as close to the X graphics core as possible, you write Xlib-based programs. Indeed, most window managers, widget libraries, and GUI toolkits are written using Xlib function. While using straight Xlib gives you the best performance, it is extremely code intensive. Xlib is an essential ingredient of the standard X distribution.
- *Xt*; Xt Intrinsics are a very thin layer of functions and data structures on top of Xlib. Xt Intrinsics create an object-oriented interface that C programs can use to create graphical elements. Without other widget sets, the Intrinsics are not especially useful. Xt, like Xlib, is a part of the standard X distribution and is not available separately.

1.2 KDE and Qt

- The name of the software is the K Desktop Environment, called KDE for short. It is a graphical user interface that is popular on Linux and other flavors of the UNIX family of operating systems. Virtually all graphical interfaces in the UNIX family are built on top of the X Windowing System.
- The X Windowing System gives the graphics its portability across many systems; the Qt library of graphics objects provides the basic building blocks of an application; and the KDE library provides a standard look and feel.
- When you write a KDE application, you are writing code that will rest on top of a lot of other code. Most of the detailed work of getting your

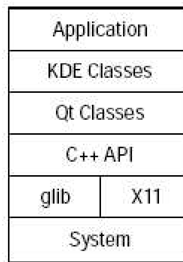

Figure 1: The levels of software for a KDE application in Linux.

application written has already been done, and that work resides in the libraries of code that will link to your application to do the things you would like for it to do. The diagram in Figure 1 should give you some idea of the levels of software that make up a KDE application.

- The way the diagram is drawn makes it appear that the levels are completely separate, but that's not the case. For example, perfectly valid calls are made from KDE classes to glib functions, and there is nothing to prevent your application from making calls directly to, say, glib or the system calls.
- An application typically uses classes from both KDE and Qt. However, the calls are only downward – for example, no part of the Qt API makes use of anything in KDE.

The Software Levels

- *System*; This is the lowest layer of software available to every Linux application. A set of low-level system calls provides direct access into the operation system, and its drivers, to do things like open files and create directories. Because the Linux kernel is written in C, these are all C function calls.
- *glib*; This is a set of C functions, macros, and structures that are used by all the layers above it; and, quite often, it is also used by applications. The glib library contains functions for memory allocation, string formatting, date and time, I/O, and timers. It also has utility functions for linked lists, arrays, hash tables, trees, quarks, and caches. One of the crucial functions handled by glib

is the main loop, which enables KDE to handle multiple resources while it simultaneously executes the code of an application.

- *X11*; This is the graphics layer that handles the low-level functions used to control the display. All the fundamental windowing functions are included – these are the functions that display windows and respond to the mouse and keyboard. This library has become very stable over the years and the version numbers have rarely changed. Currently, it is version 11 (as indicated by its name). And, because version 11 is in release 6, it is also known as X11R6. Its original name was without the version number, so it is often simply called X.
- *C++ API*; Everything above this layer is written using C++, so the C++ run-time system is called on for things such as creating new objects and handling I/O streams.
- *Qt Classes*; This set of C++ classes implements the various widgets (buttons, window frames, and so on) that can be used to create an application. It has the capability of combing windows to together to create complicated graphics dialogs. At the same time that it displays these widgets, it can respond to the mouse and keyboard for more input, and dispatch information from the input window to the correct part of the program.
- *KDE Classes*; These classes modify and add functionality to the Qt classes. There is a large number of KDE classes, but the majority of them extend directly from one or more of the Qt classes. This layer is what gives KDE its unique appearance, and standardizes the way the window, mouse, and keyboard all interact with one another.
- *Applications*; There are two basic flavors of applications. You can create either a Qt application or a KDE application. A Qt application is one that creates a QApplication object to initialize itself, while a KDE application initializes itself by creating a KApplication object. The KApplication class extends the QApplication class by adding the things that are necessary for the standard appearance and capabilities of a KDE application.

About Qt

- Qt is a library of C++ GUI application development software. Its purpose is to provide everything needed to develop the user

interface portion of applications. It does this primarily in the form of a collection of C++ classes. The Norwegian company Troll Tech (http://www.trolltech.com) first introduced Qt as a commercial product in 1995.

- The set of Qt classes is quite robust. The Qt classes include everything from basic window controls, drag and drop, and internationalization to network programming.
- *The QObject Class*; All but about a dozen of the Qt classes inherit from the base class QObject. This means that virtually every class in the Qt library contains the same basic set of methods. The constructor for this class can optionally accept the address of a parent object, and a character string that assigns the object a name:

  ```
  QObject(QObject *parent = 0, const char *name = 0);
  ```

- *The MOC Compiler*; One feature used by developers is the Meta Object Compiler (also called the MOC compiler).
  * The MOC compiler reads your source code and generates special C++ source files for you to compile and link along with your application.
  * These special files contain the code necessary for one object to emit a "signal" that is received by a "slot" in one or more other objects. This is the method used to asynchronously transmit information from one object to another within an application.
  * The MOC compiler is triggered by the presence of the Q_OBJECT macro within a class definition to determine whether to generate code, and what code is generated.
  * The resulting source code can be either compiled separately and linked, or simply included in your code with the #include directive.
  * Using the MOC compiler not only activates the signals and slots, but also generates code that enables some special methods that are defined in every Qt class (and thus, by inheritance, in every object in your program). These special methods are defined in the QtObject class.

About KDE

- KDE is an open source development project of a graphical desktop environment. Other than being the first letter of the acronym, the K doesn't stand for anything. It is just a name.
- The KDE software is constructed using Qt. The project began in 1996, the year after the first version of Qt was released. Since then, the project has grown to become a very complete desktop environment with a large collection of applications.
- From the software developer's point of view, KDE is quite simple. While most of the software written as part of the KDE project is used as an integral part of the desktop environment, a large number of classes have also been developed; and they are included as part of a core KDE API. These classes are meant to help give KDE applications a standard look and feel.
- Most of these classes inherit from one or more classes of the Qt library, and some of the KDE classes add capabilities beyond that of Qt, but most of them are simply for the sake of maintaining the standard appearance of KDE. It would be easy enough to write your entire application using only the classes of Qt, but if you use the KDE classes, your application is more likely to appear integrated with the rest of the desktop.

Events Happen

- An application that runs in the K Desktop Environment is an *event-driven* program. This means that when a program starts running, it displays its window (or windows) and waits for input from the mouse or keyboard. This input comes wrapped inside objects called *events*.
- An event can also tell the program that a window has been closed, or that the window has been exposed after being hidden behind another window. The application's entire purpose is to respond intelligently to the keyboard and mouse.
- An application has one main top-level window. It can also have other windows. These windows can exist for the entire life of the application, or they can appear and disappear as the application responds to events.

- Each window is encapsulated in a *widget*. The top-level window of an application is a widget. Each pop-up window is also a widget. In fact, the entire display is made up of widgets. Because one widget is capable of containing and displaying other widgets, every button, label, and menu item is its own individual widget.
- Programming the graphical display portion of your application is a matter of creating and combining widgets, and then writing the code that activates the widgets and responds to the events received by the widgets.
- A widget is any class that inherits from the Qt class named QWidget. A QWidget object contains and manages its own displayable window. It can also be set to respond to events issued by the mouse and keyboard (and whatever else you have for input) that are sent to the window inside the widget. It knows things about its current visibility, its size, its background color, its foreground color, its position on the display, and so on. You can use the widgets defined in either Qt or KDE, or you can create your own by using QWidget as a base class.

The Names of Things

- The Qt class names begin with the letter Q and the KDE class names begin with the letter K. That way, when you read the source code of a program, you can determine where a class is defined. If you find two classes that have the same name except for the first letter, it means that one is an extension of the other.
- For example, the KDE class KPixmap uses the Qt class QPixmap as its base class. Every class in Qt and KDE is defined in a header file. In every case (well, almost every case), the header file derives its name from the name of the class.
- For example, the header file for the QPopupMenu class is named qpopupmenu.h, and the class KFontDialog is defined in kfontdialog.h. However, this naming convention is not universally true because more than one class can be defined in a header.
- For example, the class KFontChooser is also defined in kfontdialog.h. Also, some source filenames are abbreviated. For example, the header for KColorDialog is named kcolordlg.h.

```
1 /* helloworld.cpp */
2 #include <qapplication.h>
3 #include <qlabel.h>
4 #include <qstring.h>
5
6 int main(int argc,char **argv)
7 {
8   QApplication app(argc,argv);
9   QLabel *label = new QLabel(NULL);
10  QString string(``Hello, world'');
11  label->setText(string);
12  label->setAlignment(
13  Qt::AlignVCenter | Qt::AlignHCenter);
14  label->setGeometry(0,0,180,75);
15  label->show();
16  app.setMainWidget(label);
17  return(app.exec());
18 }
```

Figure 2: A simple Qt program displaying text.

1.3 Creating and Displaying a Window

The first example is a minimal Qt application, and the second is a minimal KDE application.

Hello Qt

- The following example program creates and displays a simple window. It doesn't do anything other than display a line of text, but it gives you an idea of the fundamental requirements of a Qt program. The window is shown in Figure 2.
  - The file qapplication.h included on line 2 is almost always included in the same source file that contains the main() function.
  - This example uses a QLabel widget to display text, so it is necessary to also include qlabel.h. And a QString object is required to specify the text displayed by the QLabel object, so qstring.h is included on line 4.
  - Line 8 creates a QApplication object named app. The QApplication object is a container that will hold the top-level window (or set of windows) of an application. A top-level window is unique

in that it never has a parent window in the application. Because the QApplication object takes over things and manages your application, there can only be one of these per program. Also, the creation of a QApplication object initializes the Qt system, so it must exist before any of the other Qt facilities are available.

- A Qt program is a C++ program. This means that in order to start the program, a function named main() will be called by the operating system. And, like all C++ programs, command-line options may or may not be passed to the main() function. The command-line options are passed on to the Qt software as part of the initialization process, as shown on line 8.
- The two command-line arguments, argc and argv, are used in the construction of app because some special flags and settings can be specified. For example, starting a Qt program with -geometry will specify the size and location of the window it displays. By altering the profile information that starts a program, a user can personalize a program's appearance.
- A QLabel widget is created on line 9. A QLabel widget is simply a window that is capable of displaying a string of characters. The label is created with its specified parent widget as NULL because this label is to be the top-level window, and top-level windows have no parents. As it is created, the label contains no text, but it is provided text by being passed the QString object created on line 10.
- The QString object is inserted into the QLabel with the call to setText() on line 11. The default action for a QLabel is to display the character string centered vertically and justified to the left, so the call to setAlignment() is made on line 12 to center the text both vertically and horizontally.
- The call to setGeometry() on line 14 determines the location, height, and width of the label widget inside the QApplication window. For this example, the label is positioned at location (0,0), which is the upper-left corner of the main window. It is also instructed to be 180 pixels wide by 75 pixels high. Before anything is displayed, the main window will query the label to find out its size, and then the main window will set its own size to contain the label.
- The call to show() on line 16 is necessary in order for the label to actually appear on the window. The show() function does not

immediately display the widget, it only configures it so that it will be displayed when the time comes. The parent window – in this case, the QApplication window – assumes the task of displaying the label, but will only do so if there has been a call to the label's show() method. Another function, named hide(), can be used to cause a widget to disappear from the display.

  - The call to setMainWidget() on line 11 inserts the label into the main window. To keep this example simple, the QLabel object is used, but normally the widget will be some sort of compound widget that contains the collection of widgets, text, and other elements of the main window of an application.
  - Finally, a call is made to exec() on line 17. This function does not return until it is time for the program to cease execution. It returns an int value representing its completion status; and because we are not processing status codes, the value is simply returned to the system.

- Because the program is simple and consists of only one source file, the makefile that compiles it is quite simple:

```
 INCL= -I$(QTDIR)/include -I$(KDEDIR)/include
CFLAGS= -pipe -O2 -fno-strength-reduce
LFLAGS= -L$(QTDIR)/lib -L$(KDEDIR)/lib -L/usr/X11R6/lib
LIBS= -lqt-mt -lX11 -lXext
CC=g++
helloworld: helloworld.o
        $(CC) $(LFLAGS) -o helloworld helloworld.o $(LIBS)
helloworld.o: helloworld.cpp
clean:
        rm -f helloworld
        rm -f helloworld.o
.SUFFIXES: .cpp
.cpp.o:
        $(CC) -c $(CFLAGS) $(INCL) -o $@ $<
```

- The makefile assumes that the environment variables QTDIR and KDEDIR are defined as the name of the installation directory of the Qt and KDE development systems. Normally, these two environment variables have their definitions configured when you install the software.

```
1 /* hellokde.cpp */
2 #include <kapp.h>
3 #include <qlabel.h>
4 #include <qstring.h>
5
6 int main(int argc,char **argv)
7 {
8   KApplication app(argc,argv,``hellokde'');
9   QLabel *label = new QLabel(NULL);
10 QString string(``Hello, KDE``);
11 label->setText(string);
12 label->setAlignment(
13 Qt::AlignVCenter | Qt::AlignHCenter);
14 label->setGeometry(0,0,180,75);
15 label->show();
16 app.setMainWidget(label);
17 return(app.exec());
18 }
```

Figure 3: A simple KDE program displaying text.

Hello KDE

- This example, shown in Figure 3, is the same as the previous one except it is based on a KApplication object, rather than a QApplication object. Because the KApplication class is based on QApplication, there are no fundamental differences other than the addition of KDE facilities such as styles and themes, the capability to use KDE widgets, access to the standard KDE configuration, access to session management information, and the capability to launch the user's Web browser and e-mail client.
  - The KApplication object is defined in the header file kapp.h included on line 2. The kapp.h file includes the qapplication.h file, so every facility available to a Qt program is also available to a KDE program. The header files included on lines 3 and 4 hold the definitions of the QLabel and QString classes.
  - The KApplication object is created on line 8 by being passed the command-line arguments and a name for the application. This name can be used for such application-specific tasks as locating icons, receiving messages, and reading configuration information.

- Because a KDE object is being used in this program, it is necessary to include the KDE library that holds the object. There are some specialized KDE libraries, but the main two libraries are libkdecore and libkdeui.

```
INCL= -I$(QTDIR)/include -I$(KDEDIR)/include/kde
CFLAGS= -O2 -fno-strength-reduce
LFLAGS= -L$(QTDIR)/lib -L$(KDEDIR)/lib -L/usr/X11R6/lib
LIBS= -lkdecore -lkdeui -lqt-mt -lX11 -lXext -ldl
CC=g++
hellokde: hellokde.o
        $(CC) $(LFLAGS) -o hellokde hellokde.o $(LIBS)
hellokde.o: hellokde.cpp
clean:
        rm -f hellokde
        rm -f hellokde.o
.SUFFIXES: .cpp
.cpp.o:
        $(CC) -c $(CFLAGS) $(INCL) -o $@ $<
```

- The LIBS definition shows the inclusion of the libraries libkdecore.a, which contains the core functionality of KDE; and libkdeui.a, which contains all of the KDE widgets. KDE internally implements ODBC (Open Database Connectivity) by dynamically loading ODBC drivers, so it is also necessary to include the library libdl.a. The installation of KDE places these libraries in the default directory, so there is no need to add a new search path to LFLAGS.