W28 In-Chapter Exercise Solutions

W28.1 On our Wayland reference systems, Ubuntu 17.10 and 18.04 , and Linux Mint 19 Cinnamon (with the notable exception of RebeccaBlackOS for obvious reasons), we got the following tabularized output for the two commands-

Ubuntu 17.10 $DESKTOP\_SESSION $XDG\_SESSION\_TYPE

login choice ubuntu-wayland(default) ubuntu-wayland wayland

ubuntu ubuntu x11

weston terminal ubuntu-xorg x11

Weston login weston wayland

Ubuntu 18.04

login choice ubuntu(default) ubuntu x11

ubuntu-wayland ubuntu-wayland wayland

weston terminal ubuntu x11

Weston login weston wayland

Linux Mint 19

login choice Cinnamon (default) cinnamon x11

weston terminal cinnamon x11

Weston login weston wayland

Why? That’s the great Caveat Emptor of this chapter. These Debian-family systems are running a “Wayland/Weston lookalike” so to speak, under an X.org desktop system. In all three reference systems, when you have logged in with Weston, and it is the only GUI compositor running, you get the desktop session as Wayland and compositor as Weston. When will this change, so that all desktop systems are running under Wayland? In the immediate future, coming to your Linux system (perhaps even CentOS)! We leave it to you to execute the code in all of the sections of this chapter using the various login choices shown, and note the results. For example, if you develop GTK3+ and Qt programs as shown in the GTK3+ and Qt sections of the chapter, and try to run them in a Weston login session, what happens?

W28.2 gtk0.c analysis

A point-by-point explanation of the gtk0.c code is as follows-

1. Generally, all GTK+ applications include gtk/gtk.h, which declares functions, types and macros required by GTK+ client applications.

2. If GTK+ installs multiple header files, only the top-level gtk/gtk.h header can be directly included by third party code. The compiler will abort with an error if any other header is directly included. Also, for CentOS 7.5 systems, the pathname to the GTK+ libraries needs to be set properly, or else you won’t be able to compile the program code we provide in this chapter. The header gtk/gtk.h will not be found.

3. The objective of the main() function is to create a GtkApplication object and run it. In this example a GtkApplication pointer named app is called and then initialized using gtk\_application\_new().

4. When creating a GtkApplication you need to name it, and input that name to the function gtk\_application\_new() as an argument. In gtk0.c, org.gtk.example is used for this purpose. gtk\_application\_new() takes a GApplicationFlags as input for your application.

5. The “activation” signal is connected to the activate() function above the main() functions. The activate() signal will be sent when the client application is launched with g\_application\_run().

6. The function gtk\_application\_run() takes as arguments the pointers to the command line arguments counter and string array; this allows GTK+ to parse specific command line arguments that control the behavior of GTK+ itself. The parsed arguments will be removed from the array, leaving the unrecognized ones for your application to parse.

7. The function g\_application\_run the activate() signal is propagated, and then then program execution proceeds to the activate() function of the client application. Inside the activate() function, the Gtk window is actually constructed, and can be displayed when the client application is launched. The function gtk\_application\_window\_new() creates a new GtkWindow, and store it inside the window pointer. The window will have a frame, a title bar, and window controls depending on the platform.

8. A window title is instanced using the function gtk\_window\_set\_title(). This function takes a GtkWindow\* pointer and a string as input. In the example, the window pointer is a GtkWidget pointer, and it has to be instaced in GtkWindow\*.

9. The window size is set using gtk\_window\_set\_default\_size, and the window is then displayed by GTK, using the function gtk\_widget\_show\_all().

10. When you exit the window, by for example pressing the X, the g\_application\_run() in the main loop returns with a number which is saved inside an integer named "status". The GtkApplication object is freed from memory with g\_object\_unref(). The status integer is returned and the GTK client application exits.

11. Conclusion: While the program is running, GTK+ is receiving event signals. These are input events caused by the user interacting with your program, but they can also be things like messages from the window manager or other applications. GTK+ processes these and as a result, signals may be emitted on your widgets. Connecting handlers for these signals is how you make your program do something in response to user input.

W28.3 gtk1.c analysis

A point-by-point explanation of the gtk1.c code is as follows-

1. gtk1.c extends gtk0.c by adding a button to the window, with the button labeled "GTK+ in Weston". Two new GtkWidget pointer objects are used to accomplish this addition, button and button\_box. The button\_box variable stores a GtkButtonBox, which controls the size and layout of buttons. The GtkButtonBox is created and assigned to gtk\_button\_box\_new() which takes a GtkOrientation enum data type as an argument. The buttons in a box can either be positioned horizontally or vertically. After initializing button\_box horizontally, the button\_box widget is added to the window widget using the function gtk\_container\_add().

2. The button variable is initialized. The function gtk\_button\_new\_with\_label().

3. button is then added to our button\_box. “Wiring” via g\_signal\_connect, the button is connected to a function print\_hello(); when the button is clicked, GTK calls this function. As the print\_hello() function does not use any data as input, the NULL pointer is passed to it. print\_hello() calls g\_print() with the string "GTK+ in Weston" which will print GTK+ in Weston in the terminal the client application was started in.

4. After wiring print\_hello(), another signal is “wired” to the "clicked" state of the button using the function g\_signal\_connect\_swapped(). This functions is similar to a g\_signal\_connect() except for how the callback function is treated. The function g\_signal\_connect\_swapped() allows specification of what the callback function should take as an argument, by letting you pass it as data. The function being “called back” is gtk\_widget\_destroy() and the window pointer is passed to it. When the button is clicked, the GTK window is destroyed. If g\_signal\_connect() were used to connect the "clicked" signal with gtk\_widget\_destroy(), then the button would have been destroyed, not the parent window frame.

W28.4 Qt Exercise W28.2

Dialog of explanation and description:

Lines 1 and 2 include the header files for QApplication and QTextEdit, which are the two classes that Qt uses. Qt is an object-oriented programming language that uses C++ class and object descriptions and functionality. All Qt classes have a header file named after them.

Line 4 declares the variables and opens the program.

Line 6 creates a QApplication object. This object manages application-wide resources and is necessary to run any Qt program that has a GUI. It needs argv and args because Qt accepts a few command line arguments for this object.

Line 8 creates a QTextEdit object. A *text edit* is a visual element in the GUI. In Qt, we call such elements *widgets*, short for *window gadgets*. Examples of other Qt widgets are scroll bars, labels, spin boxes, sliders, and radio buttons. A widget can also be a container for other widgets, a dialog area, or a main application window.

Line 9 shows the text edit on the screen in its own window frame. Since widgets also function as containers (for instance QMainWindow, which has toolbars, menus, a status bar, and a few other widgets), it is possible to show a single widget in its own window. Widgets are not visible by default; the function show() makes the widget visible.

Line 11 makes the QApplication object enter its event loop, similar to XCB and Xlib client application programs. When a Qt client application is running, events are generated and sent to the widgets of the application. Examples of events, as seen in XCB and Xlib, are mouse button presses, mouse cursor movements, and key strokes pressed on the keyboard. When you type text in the text edit widget, it receives key press events and responds by drawing the text that was typed.

W28.5 Qt Exercise W28.3

Line 1 includes QtGui, which contains all of Qt’s GUI classes.

Lines 7 and 8 create two pointer objects to be used to reference the classes of objects below.

The next line illustrates probably the most important Qt call.

Line 10 uses Qt’s s*ignals and slots* mechanism to make the application exit when the Quit button is pushed. A *slot* is a function that can be invoked at run time using its name (as a literal string). A *signal* is a function that when called will invoke slots registered with it; we call that to connect the slot to the signal and to emit the signal. So, quit() is a *slot* of QApplication that exits the application; clicked() is a signal that QPushButton emits when it is pushed.

As a programming reminder for C++, **::** is called the (binary) *scope resolution operator*. By using the scope resolution operator, you can address member functions outside of a class. Also remember that the scope resolution operator specifies that the identifier which is on the right belongs to the data type or class on the left.

The static QObject::connect() function takes care of connecting the slot to the signal. SIGNAL() and SLOT() are two macros that take the function signatures of the signal and slot to connect. We also need to give pointers to the objects that should send and receive the signal.

Line 12 creates a QVBoxLayout. As mentioned, widgets can contain other widgets. It is possible to set the bounds (the location and size) of child widgets directly, but it is usually easier to use a layout. A layout manages the bounds of a widget’s children. QVBoxLayout places the children in a vertical row.

Line 13 and 14 adds the text edit and button to the layout.

Line 17 sets the layout on a widget.

Line 19 uncovers the window.

Line 21 starts the event loop.

W28.5 No answer required. But please realize that when we had installed Qt before installing gnuplot via the Software Manager in Linux Mint 18.3, the Qt terminal type was available, useable, and set as the default. This may not be true in your Linux system.

Following is a listing of a few library-dev packages which must be installed first before setting terminal types for gnuplot in not only Ubuntu, but in Linux Mint as well:

For the x11 terminal- libx11-dev

For the pdfcairo, pngcairo, epscairo terminals- libcairo2-dev, libpango1.0-dev

For the interactive wxt terminal- libwxgtk2.8-dev or libwxgtk3.0-dev, libgtk2.0-dev

For libgd-based png, gif, jpeg terminals- libgd-dev, and by default these will also be installed: libxpm-dev, libjpeg-turbo8-dev, libtiff5-dev

For the interactive qt terminal - qtbase5-dev, libqt5svg5-dev

For qt4 terminals- libqt4-dev

\*\*Note that later versions of the above library-dev packages will be available for later releases of Linux systems that have gnuplot available.

W28.6 No answer required.

W28.7 No answer required.

W28.8 No answer required.

W28.9 No answer required.

W28.10 No answer required.