Department of Computer Science
National University of Singapore

# CS3249 User Interface Development

AY2013-14 Semester 2

# Lab 1: Ubuntu Linux and MyEditor

## Objectives

- Learn to use Linux command in a terrminal.
- Learn to edit, compile and run a Qt program.
- Learn to add menu items and tool bar items to a GUI program.

## Part 0. First Taste of Linux Commands

This part illustrates several frequently used Linux command in command-line mode.

1. Look for the Launcher located on the left of Ubuntu desktop (Figure 1).
2. Launch the GNOME terminal by single clicking its icon in the launcher.
3. In the terminal, you should see a command prompt something like

```
stu001-~$
```

4. The GNOME terminal runs a program called bash (Bourne-Again Shell) that interprets the commands that you type and executes the corresponding programs. Examine the following commands by typing them one at a  time followed by the `Enter` key:

```
pwd
ls
ls -l
ls -a
ls Pictures
ls .
man ls
```

- `pwd` displays the name of the current directory.
- `ls` lists the files and sub-directories in the current directory. The `ls` command can be followed by options or arguments. For example, the option `-l` means list details of files and sub-directories, and `-a` means list hidden files, which are files that begin with the character '`.`'. The argument `Pictures` refers to the subdirectory called Pictures and . refers to the current directory.
- `man` stands for manual. It is used to display manual information about commands. For example, "`man ls`" displays manual of `ls`.

5. Enter the following command at the command prompt `$` (you don't need to type the first $, but you do need to type the second `$` just before `PATH`):

   ```
   $ echo $PATH
   ```

   It displays the value of the environment variable `PATH`, which contains a list of directories that bash searches for the program that corresponds to the command that you type. By default, the current directory `.` is not listed in `PATH`. This is inconvenient because to run a program in the current directory, say `test`, you need to type `./test` instead of just `test`.
6. To add the current directory to `PATH`, do the following:
   a. Look for Home Folder in the Launcher and launch it.
   b. Click the Home icon on the left panel to display the Home folder (i.e., directory).
   c. In Home Folder, select the menu item View → Show Hidden Files.
   d. Look for the file called `.bashrc` and open it by double-clicking its icon.
   e. At the end of `.bashrc`, enter the following line

      ```
      PATH=$PATH:.
      ```

      This will append the current directory to `PATH`.
   f. If you prefer a shorter command prompt, you can also change it in `.bashrc`. For example, the following line

      ```
      PS1="\u-\w\$ "
      ```

      sets the command prompt `PS1` as the string that contains your login name (`\u`) followed by `-`, the current working director (`\w`), and `$`.
   g. Save `.bashrc` and exit the editor.
   h. Close the GNOME terminal and re-launch it.
   i. Now, if you enter the command `echo $PATH`, you should see the current directory listed in `PATH`.
   j. In Home Folder, deselect View → Show Hidden Files to hide the hidden files.

## Part 1. First Taste of Ubuntu

This part gives you a first taste of using Ubuntu.

1. In Home Folder, click the `Home` icon on the left panel or in the menu bar to go to your home directory.
2. With the cursor in the Home Folder, press the right mouse button and create a new folder with your name, e.g., `myname`.
3. Create a folder called `myeditor` under `myname`. This is your working folder.
4. Double click the icon for `myeditor` to open your working folder.
5. Download `myeditor.zip` from the course website into your working folder.
6. Double click `myeditor.zip` to open it. You should see the following files: `example.txt`, `main.cpp`, `myeditor.pdf`, `myeditor.qrc`, `MyEditor.h`, `MyEditor.cpp`. You should also see a folder called `images` with several images in it.

Drag the files and folders into your working folder.

7. Double click `myeditor.qrc` to open it in the default editor. Two editors have been installed: `medit` and `gedit`. `medit` has block indent and block comment which are convenient for indenting and commenting a block of codes, whereas `gedit`'s search feature is more convenient to use. You can change the default editor by right-clicking on the file icon, selecting the `Properties` menu item in the pop-up menu, and change the default editor in the `Open With` tab.
8. Similarly, examine `MyEditor.h`, `MyEditor.cpp`, and `main.cpp`.
9. If you find that your desktop is getting cluttered, no worry. Ubuntu comes with not just one but four workspaces! Click the workspace icon in the Launcher (Figure 1) and you should see the four workspaces. You can move any application from one workspace to another by clicking and dragging the application. Double click any one of the workspaces will bring you to that workspace. Now, you can organise different applications in different workspaces and keep your workspaces neat and tidy.

## Part 2. Program Compilation

This part illustrates how to compile Qt programs.

1. Launch the GNOME terminal by clicking its icon in the launcher.
2. Change directory to your working directory, e.g.,
   ```
   $ cd myname/myeditor
   ```
3. Compile the Qt programs in three steps. First, create project file as follows:
   ```
   $ qmake -project
   ```
   This command creates the project file `myeditor.pro`. You can double click on it to open it and examine its content. This step is needed only when the Qt resource file `myeditor.qrc` is revised or new program files are added.
4. Second, create `Makefile` as follows:
   ```
   $ qmake
   ```
   This command creates the `Makefile` which indicates how to compile the program. You can double click its icon to open it and examine its content.
5. Finally, compile the programs as follows:
   ```
   $ make
   ```
   This command executes the instructions in `Makefile`, which include the following:
   - Create and compile the resource file `qrc_myeditor.cpp`. This file encodes the icon images that will be embedded into the executable file. This way, the executable file can be run else where.
   - Create and compile `moc_MyEditor.cpp`. This file contains the meta object codes for `MyEditor`.
   - Compile `MyEditor.cpp` and `main.cpp`.
   - Link all the `.o` files and the Qt library files into an executable file.
6. By default, the name of the directory is used as the name of the executable file. To

specify a different name for the executable file, open `myeditor.pro` and fill in the `TARGET` entry, for example,

```
TARGET = simple_editor
```

Next, run `qmake` to generate appropriate `Makefile`. Finally, compile the program with `make`.

## Part 3. Program Execution

This part illustrates running the program `myeditor`.

1. There are two ways to run `myeditor`: (1) In your working folder in Home Folder, double click `myeditor` icon. (2) At the GNOME terminal's command prompt, type `myeditor` followed by the `Enter` key.
2. Try each of the menu items in `myeditor` using the mouse. Load the sample file `example.txt` into the editor to test the editor's features.
3. Try each of the menu items using the keyboard. For example, press `ALT+F` to show the File menu, then press `O` to open a file.
4. Try each of the menu items using keyboard short cuts, e..g, `CTRL+O` to open a file.
5. Try each of the tool bar items by clicking the icons in the tool bar.

## Part 4. Adding Menu and Tool Bar Items

In this part, you will learn to add three new actions to the editor application.

1. Add the following icon image file names to `myeditor.qrc`: `copy.png`, `new.png` and `saveas.png`. The icon image files are already supplied in the `images` directory.
2. Add the following `QActions` to the `MyEditor.h`: `newAction`, `saveAsAction`, `copyAction`.
3. Add the following slot functions to `MyEditor.h`:
    - `void newFile()`
    - `bool saveAs()`
4. Add to `MyEditor::createActions()` in `MyEditor.cpp` the following codes:
    - Statements for creating `newAction`. Choose appropriate icon image file, keyboard short cut, and status tip for the action. Connect the action's `triggered()` signal to the `newFile()` slot.
    - Repeat for `saveAsAction` and `copyAction`.
    - The `QPlainTextEdit` widget used in `MyEditor` has a `copy()` slot which can be connected to the `copyAction`. So, it is not necessary to reimplement `copy()` slot.
5. Add the new actions to the appropriate menu bars and tool bars in `MyEditor::createMenus()` and `MyEditor::createToolBars()`.
6. Define the following functions in `MyEditor.cpp` but leave the function bodies empty for the time being:

    - `void MyEditor::newFile()`
    - `bool MyEditor::saveAs()`

7. Compile and run the program. You should see the new actions included in the menu bars and the tool bars. Try them out. As the `newFile` and `saveAs` function bodies are empty, activating their menu items and tool bar items performs nothing.

## Part 5. Writing Slot Functions

In this part, you will learn to write slot functions. You may need to refer to Qt Assistant for the arguments and return objects of the functions. There are two ways to run Qt Assistant: (1) Click Dash home icon in the Launcher, type Qt in the search box and press enter, then click the Qt Assistant icon. (2) In GNOME terminal's command prompt, enter

```
assistant &
```

The `&` option informs bash to run Qt Assistant in a background process. This way, you get your command prompt back.

1. In `newFile` slot, call `okToContinue()` to check whether it is ok to continue. If yes, use `textEdit->clear()` to clear the text editor's content, and then call `setCurrentFile(””)` to set the file name to an empty string.
2. In `saveAs` slot, call `QFileDialog::getSaveFileName` to get a user-specified file name. If the file name is not empty, then call `saveFile` to save the content.
3. Change the `save` function to call `saveAs` if `currFile` is empty.
4. Compile and run the program.

(Optional) As this is a simple exercise, you should not need a debugger to debug your program. Nevertheless, it's good to learn how to use a debugger to debug complex programs. Refer to the Appendix for how to use a debugger in Ubuntu Linux.

## Submission

After completing the lab exercise, run the revised editor and show it to the Lab TA for verification. Then, print and submit the following to the Lab TA:

- Revised `MyEditor.h` and `MyEditor.cpp`. Remember to write your name, matriculation number and lab group as comments in `MyEditor.h` and `MyEditor.cpp`.
- A screen shot of the revised `MyEditor`. To save a screen shot into a file, run the program and press `ALT+PRTSC`. By default, the screen shot will be saved into the Pictures directory under Home. You can save it into another directory if you wish.

## Important Note

Save `myeditor` folder and everything in it into a flash drive for Lab 2.

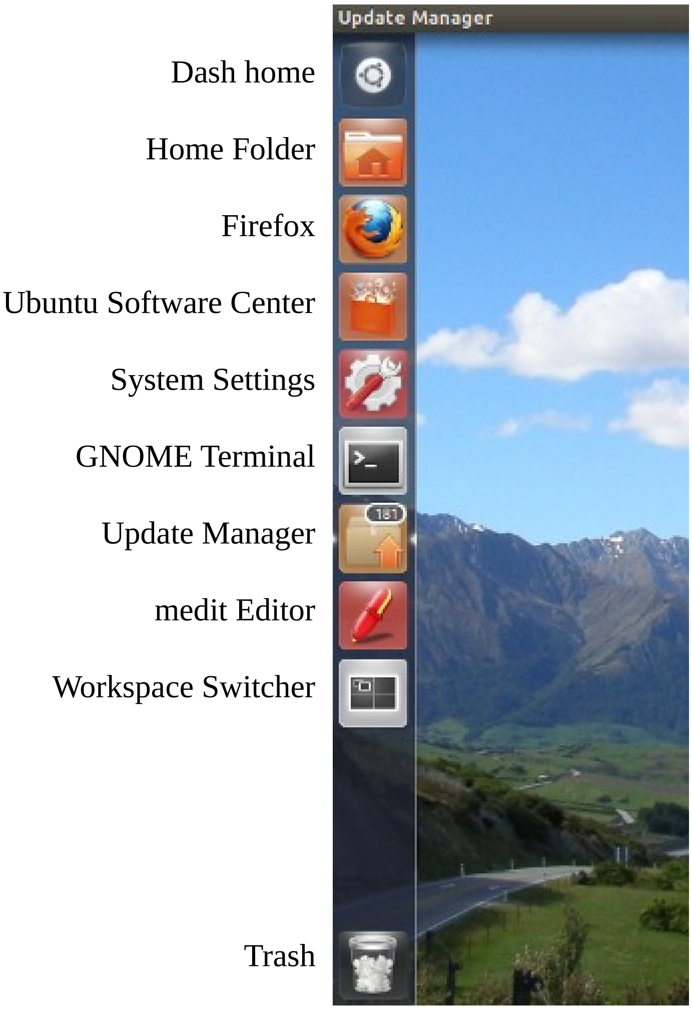


Figure 1. Launcher in Ubuntu desktop. The icons in your default launcher may not be the same as those shown here. You can keep or remove any of them by right-clicking the icons and selecting or deselecting “Keep in launcher” option.

# Appendix. Debugging Qt Programs

## 1. Compile Programs for Debugging

To debug a program, it has to be compiled with the `-g` option. This option can be included in one of the following ways:

1. Include in `Makefile`

   In the `Makefile`, there is a line that begins with `CXXFLAGS`:

   ```
   CXXFLAGS = -pipe -O2 ...
   ```

   These are the options for compiling C++ programs. Add `-g` anywhere after =. As the `Makefile` is generated by `qmake`, you need to add `-g` option to the `Makefile` after you run `qmake`.

2. Include in Qt project file

   In the Qt project file, e.g., `myeditor.pro`, add the line

   ```
   QMAKE_CXXFLAGS += -g
   ```

   before the `HEADERS` line. Then, run `qmake`, and `qmake` will generate a `Makefile` file with the `-g` option included in `CXXFLAGS`. This is more convenient if you need to re-run `qmake` to generate `Makefile`. As the project file is generated by running

   ```
   qmake -project
   ```

   you need to add the line to the project file after it is generated by `qmake`.

3. Include in an auxiliary project file

   Create an auxiliary project file, e.g., `extra.pro`, and add the line

   ```
   QMAKE_CXXFLAGS += -g
   ```

   to `extra.pro`. Then, add the line

   ```
   include(extra.pro)
   ```

   in the project file before the `HEADERS` line. Then, `qmake` will include the additional statements in `extra.pro` when it is run. This is the same as for the second method, but is convenient when you have more than one line of additional statements to include into the project file.

After compiling the programs with the -g option, you can run a debugger to debug your program. There are two convenient debuggers that come with Ubuntu: GDB and Nemiver.

## 2. GDB

GDB is a command-line debugger developed by GNU. It is very light-weight and quite easy to use despite being a command-line debugger. To run GDB to debug an executable program compiled with `-g` option, for example `myeditor`, open a terminal and type

```
gdb myeditor
```

At the `(gdb)` prompt, you can enter various GDB commands to debug your program. Frequently used commands can be entered with a single letter, for example, `h` for `help`, `l` for `list`, `c` for `continue`, etc. The following is a list of frequently used commands:

| | |
|---|---|
| ↑ | Display the previously entered GDB command |
| `help` | List categories of GDB commands |
| `help` <category> | List commands in a category |
| `help` <command> | Display information about a command |
| `list` | List current function around current break point |
| `list` <line number> | List current function around specified line number |
| `list` <file>`:`<line number> | List file content around specified line number |
| `list` <function> | List first few lines of specified function |
| `break` <location> | Set breakpoint at specified location (line number or function) |
| `clear` <location> | Clear breakpoint at specified location |
| `delete` <breakpoint> | Delete breakpoint specified by number |
| `delete` <display> | Cancel display expressions |
| `run` | Start running program |
| `step` | Step through one source line |
| `step` <n> | Step through n source lines |
| `next` | Step over one source line without entering a function |
| `next` <n> | Step over n source lines |
| `advance` <location> | Continue the program up to the given location |
| `continue` | Continue the program up to the next breakpoint |
| `print` <expr> | Display value of variable or expression |
| `display` <expr> | Display value of variable or expression when program stops |
| `info breakpoint` | Show information about breakpoints |
| `info display` | Show information about auto-display expressions |

For more information about GDB, refer to the user guide *Debugging with GDB* located in /usr/local/doc. The root directory / is accessible as File System sub-folder in the Home Folder.

## 3. Nemiver C/C++ Debugger

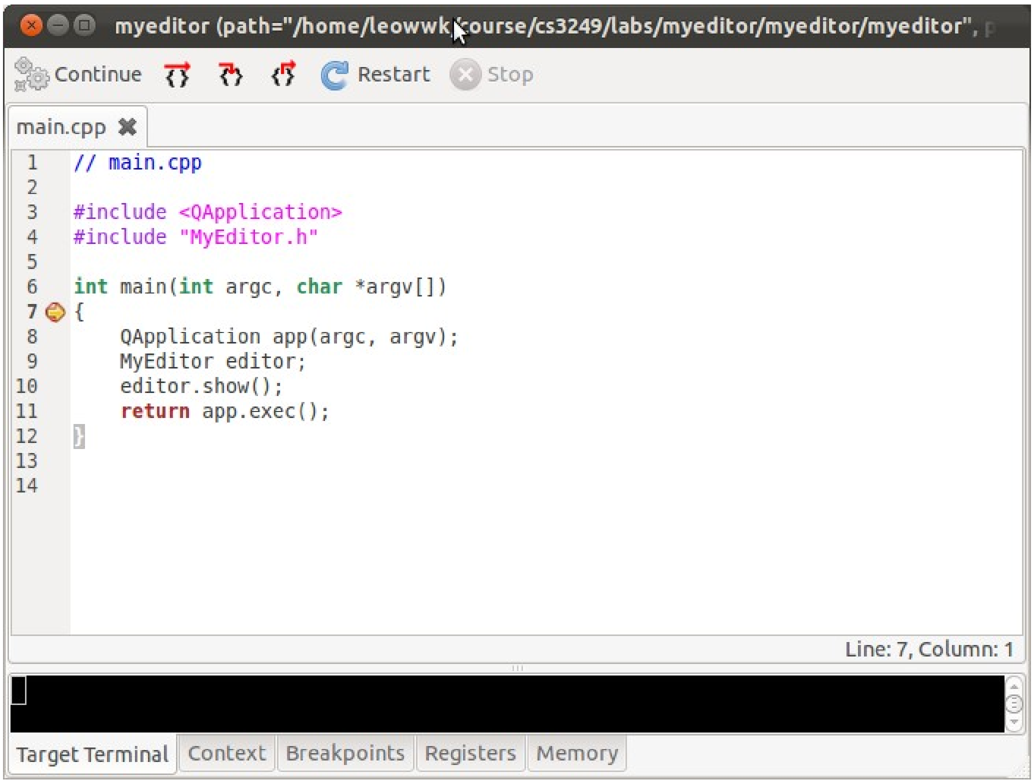
Nemiver C/C++ Debugger is a GUI tool for debugging C/C++ programs. There are two ways to run Nemiver:

1. Click the Dash home icon and type Nemiver or debug in the search box. Then, click the Nemiver icon. After Nemiver is launched, you need to load the executable by selecting the menu item File → Load Executable, followed by filling in the popup form.

2. At a terminal's command prompt, change directory to the working directory and type nemiver with the executable file as the argument, e.g.

```
cd myname/myeditor
nemiver myeditor
```

Nemiver will be launched with the source file loaded and ready to go.



```
// main.cpp

#include <QApplication>
#include "MyEditor.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyEditor editor;
    editor.show();
    return app.exec();
}
```

Figure 2. Nemiver C/C++ debugger.