Programming Practice Report

of Software Engineering School

SUBJECT	Line Planner
AUTHOR	WEIJU LAN (兰威举)
MAJOR	Software Engineering
CLASS	2013 Class 1
ID	13108115

Table of contents

Chapter 1

Design

1.1 Goals

The `line planner` is designed to be a TUI [Text User Interface] application for `modern Linux`. It's designed to be modern, robust, signal-aware and linux-only. The whole project is written in `GNU C++14`.

The `line planner` should parse the line/stop information from a table-like file and should reparse the file when it is changed by other applications like a text editor. The user should be able to search for the passing stops of a line and the passing lines of a stop. And it should be able to plan routes for the user between 2 stops within 2 changes of lines (i.e. on 3 lines most). The application should respect the use habits of a serious linux user, which means, it should cancel the current blocking operation or exit in a user-friendly way when the user hit `^c` (a.k.a. hold the `control` key and hit the lower–case letter `c`), and it should force quit abnormally when the user smashes `^c`.

Only the plans that have the least line-changes are kept and displayed to the user. The result will be sorted in an ascending manner by the number of stops the lines pass by in the plan. When the user want to see details of a plan, a graph should be generated and displayed on the screen.

1.1.1 The TUI

The TUI should be modern and straightforward like a web browser. It should have only 1 modal interface displaying the line/stop information or the plan list/details.

The TUI should be able to display and input `UTF-8`-encoded Unicode (mostly Chinese) characters. The size/dimension of the UI should not be hard-coded, it should be responsive to the terminal size.

1.1.2 The Planner

The planner is responsible for parsing the line/stop information file, and restructure the data into a planner state. It then should be able to plan the route using that state.

The goal of restructuring the data is to make the later queries easier to do. Thus the data should be restructured into some *associative arrays* and some sort of *adjacent table* of a graph.

The route planning is done by doing an optimized *depth-first search* of the graph in the planner state generated by the data parser.

1.2 Decisions

1.2.1 Only Lines Have Directions

Both the lines and stops provided in the data file have a direction associated. In this application, the direction of a line is *kept*, while stops having the same name but differs in directions are *merged* into one stop.

1.2.2 No Direction Change When Searching

When doing the route planning by searching the graph, the direction won't change if the line is not changed.

1.2.3 Stops Are Visited Only Once

Stops are visited exactly once in each plan.

1.2.4 Only Display Least Changes

Only lines that have least changes are displayed. If there is a direct line, lines with 1 change will not be displayed, even if there is one.

1.2.5 Use `glibc` for `UTF-8` Handling

There is Unicode support since `C++11`, but `libstdc++` (the default `C++` standard library on most linux systems) did **NOT** implement it. There are 2 alternatives: either use the `GNU C` standard library `glibc` or use `libc++` from `clang`. `glibc` is chosen for smaller footprint.

1.2.6 Use `graphviz` for Graph Visualization

`graphviz dot` is used to generate the eye-pleasing graph image of the plan returned by the route planner.

Chapter 2

Implementation

2.1 Overview

- The whole project uses `git` to do the version control
- The building system uses a custom one of mine, which includes a `perl`-written `configure` script that generates `GNU makefile`. Then the actual building is coordinated by `GNU make`, which calls `clang++` to build the whole application.
- The whole project is scanned by `clang static analyzer` in order to discover and eliminate static bugs.
- The application is written in `GNU C++14`, using `ncursesw` to support the TUI. The `w` version of the `ncurses` library supports Unicode `UCS-4` encoding, which then can be converted into `UTF-8`.
- `ANSI Escape Sequences` is used for setting the title of X virtual terminals like `gnome terminal`.
- Designed with `C++`'s multi-paradigm in mind. Using POP Procedure-Oriented Programming, FP Functional Programming, GP Generic Programming and MP Meta-Programming techniques.

```
line-planner
|-----------  main
|-----------  mixin
|-----------  signal
|-----------  utf8
|-----------  watch
|-----------  ui
|             |------  context
|             |------  style
|             |------  widget
|             |------  choose
|             |------  edit
|             |------  list
|             |------  search
|             |------  linestop
|             |------  plan
|             |------  stage
|-----------  planner
              |------  state
              |------  parser
              |------  plan
```

Figure 2.1. Components

2.2 Challenges

2.2.1 Route Planning

This is in fact the simplest part of this program, just do a DFS depth-first search and everything is done. Stops are vertices, and a transition is an edge.

There is some modification in the DFS algorithm. Since only the lines with least changes is needed, lines with change count larger than the already found ones are skipped, and when a line with less changes is found, the already found ones are cleared.

2.2.2 Unicode Handling

In linux, Unicode is encoded in `UTF-8` variable length encoding, which is byte-stream oriented, making it simple to integrate into `char string` but difficult to calculate length/width and edit.

In terminals, texts are displayed in a grid, each English letter taking up one cell of the grid. But some characters like the Chinese ones need 2 horizontally adjacent cells. The width of a `UTF-8 string` is needed to proper align texts in terminals. Because of the length-varying nature of `UTF-8`, it cannot be done by just counting the number of bytes nor codepoints in the string.

The way to do this is tricky: convert `UTF-8 string` to `UCS-4 string`, then calculate its width using `wcswidth(3)` (conforming to the standard `POSIX.1-2001`). Before calling `wcswidth`, `std::setlocale` must be called to setup `LC_CTYPE` to the system's locale.

Editing is done similarly. To remove last character, convert it to `UCS-4 string` and then `pop_back` the last codepoint. To fill gaps, do it the same way.

Other operations like fixing the width of the `string` are done in the same way.

To support the input of Unicode, the `w` version of `ncurses` (i.e. `ncursesw`) is used. The `getch(3)` is replaced by `get_wch(3)` to input `UCS-4 char` (i.e. `wchar_t`, which is a `32-bit fixed-point number`), and then it is converted to `UTF-8 string` for further processing.

2.2.3 The TUI

All the low-level call to `ncursesw` is wrapped in `struct context` to hide the ugly `namespace-less C` interface of the library.

All the rendering is manually done. The TUI is designed to be themable (in `style` module) and composible in a *functional* (as in functional programming) manner. The `std::initializer_list` is exploited in some places (e.g. `choose` and `list`) to make life easier.

```
  widget:  N/A
  choose:  widget
    edit:  N/A
    list:  widget
  search:  widget, list, edit
linestop:  search
    plan:  widget, search
   stage:  widget, choose, linestop
```

Table 2.1. Dependencies

2.2.4 File Changes Monitoring

This application uses `inotify(7)` mechanism provided by the Linux kernel to monitor the change of the data file in the filesystem.

The directory containing the data file is monitored rather than the file itself, bacause the mature and robust way to save a file (like what `vim` has done) is to save the file in another name and then move that file to override the old one. And you can no longer monitor the file when it has been overridden. Thus the directory is monitored with `IN_CREATE | IN_DELETE | IN_MODIFY | IN_MOVE` attributes in order to get notified when the data file is changed.

To actual monitor it, a blocking `read(2)` for `struct inotify_event` (see `inotify(7)`) is called with the `watch descriptor` created from the `inotify descriptor` created by `inotify_init(2)`. When any event happens, the `read(2)` call returns.

Bacause it's a blocking call, in order to co-operate with other part of the program, the monitoring is done in another thread. `C++` provides native threading support by `std::thread`. A `bool` *atomic* variable (`std::atomic_bool`) is used for communication.

But sometimes a blocking wait is needed, thus a `wait_change` function is provided for that. It uses `std::condition_variable` with `std::mutex` and the atomic variable described above to achieve the waiting without wasting CPU cycles.

2.2.5 Signal Handling

Signals, because of its asynchronous nature, is difficult to handle. As a general rule of thumb: do **NOT** change anything (except for `std::sig_atomic_t` variables) or call any mutating function in signal callbacks.

In this application, the signal callback only changes a "quit state" *signal atomic* variable, or throws `force_quit` when appropriate in order to respect the use habits of Linux terminal applications.

2.2.6 Presentation of Plans

The plan (a.k.a. the result of the route planner) ought to be presented in a pleasing manner. Thus, it is displayed as a graph image on the screen.

A `graphviz dot` script is generated for the plan and then fed into a little script called `preview`, which is an `sh(1)` script that calls `dot(1)` to generate a `png` image of the graph, store it in a directory as history, and display the image with `feh(1)`.

Chapter 3
Summary and Thoughts

I thought the route planning algorithm would be the most challenging part of the program, but no, it seems to be the simplest part. Thus it's not focused in this report.

I had only written small TUIs and never tried writing medium-sized or even large-scale TUI application, which is rather complex and daunting, though not difficult. The UI part takes up 64% of all the source code.

Applying various techniques from different paradigms is really important to write *clean*, *simple* yet *readable* code in `modern C++`. OOP Object-Oriented Programming is avoided *deliberately* because it simply **sucks**. `C++` has much much superior resource (memory, files, etc) management method than any other programming language in the world, which made me stick to it.

`C++14` rocks. I switched to `clang` completely, which has better error messages and optimization than `GCC`, and it even comes with a `static analyzer`. It's a pity that I didn't try the `libc++` from `clang` this time, I will switch to it soon.

Software Enginnering School

Class	2013 Class 1
Author	Weiju Lan (兰威举)
Id	13108115
Date	2014/12/06