Curtin University — Department of Computing

**UNIX and C Programming 120 / 500** (10163 / 10225)

Semester 2, 2013

# Assignment

**Due:** 8th November 2013, 12:00 noon.

**Weight:** 20% of the unit mark.

Your task is to create a calendar application, using a graphical user interface (GUI).

## 1 Documentation

As you write your code, you must thoroughly document it using C comments (`/* ... */`).

For each function you define and type (struct/typedef) you declare, you must place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. (These comments are worth substantial marks – see Section 8.)

## 2 Compiling

The use of a GUI requires a slightly more complex gcc command-line. You will need to add this to the end of your compile and link commands:

```
`pkg-config --cflags --libs gtk+-2.0`
```

Each ` character is a *backtick*, not a single quote. For example:

```
gcc -c gui.c `pkg-config --cflags --libs gtk+-2.0`
gcc -c example.c `pkg-config --cflags --libs gtk+-2.0`
gcc gui.o example.o -o program `pkg-config --cflags --libs gtk+-2.0`
```

(Use `-ansi -pedantic -Wall` too, of course.)

You are required to submit a working Makefile. Hopefully you will also realise that *using* a Makefile will save you a lot of time!

In order to compile and run your program, the machine you use must have "GTK+" installed. This is true of the Dept. of Computing lab machines.

# 3 Program Outline

Your program will work with one "calendar" at a time, where a calendar consists of a set of events. Each event will have a date, a time, a duration (in minutes), a name and a location. The last two are strings.

(For simplicity, we will not consider repeating events, and we will not attempt to *draw* a calendar in a day-by-day, week-by-week or month-by-month fashion.)

Your program must do the following:

- The user may optionally supply a command-line parameter. If they do, this represents a calendar file, and must be loaded. (Section 4 explains the file format.)
- You must store the loaded calendar, if any, in an appropriate linked list. If there are no command-line parameters, the list will initially be empty.
- Display the entire calendar, in the text area of the GUI window. (Section 5 explains the on-screen format.)
- Using the GUI functions described in Section 6, present the user with the following options (as buttons):

  **Load a calendar from file** – An existing, possibly different calendar to replace the one currently loaded.
  **Save the current calendar to file** – In the format described in Section 4, so that it can be re-loaded by your application.
  **Add a calendar event.**
  **Edit a calendar event.** (Consider how the user should identify an event to edit.)
  **Delete a calendar event.**

  These options will all need to work with the linked list, and (except for saving) will need to modify it. Use the GUI functions for all user input/output.
- Where appropriate, you must validate the contents of a calendar event, to ensure it contains a valid date, time and duration, and that it has a non-empty name. (The location may legitimately be empty.)

# 4 Calendar Files

Calendar files contain several events. The file *does not* state the number of events – this can only be known by reading the file to the end.

Each event consists of *one or two* lines, and events are separated by empty lines. For simplicity, the very last line of the file is also empty. For each event:

- The first line contains the date, time, duration and name, in that order, separated by spaces.
  - The date is in YEAR-MONTH-DAY format (e.g. "2013-05-17").

  - The time is in 24-hour HOURS:MINUTES format (e.g. "18:30").
  - The duration is a single non-negative integer (a time in minutes).
  - The name takes up the rest of the line, and may contain any valid string characters except for a new-line.

- If the entry takes up two lines, the second line contains the location. Otherwise, the location is empty. Like the name, the location may contain any set of characters, except for a new-line.

Here is a simple example file:

```
2013-11-08 10:00 75 Veg out
Home

2013-11-08 11:15 45 Work on UCP Assignment
The Labs

2013-12-03 12:30 5 Make Huge Foreign Currency Transactions

2014-06-30 23:59 25920 Armageddon -- Contact Bruce Willis
Earth
```

# 5 Calendar Display

Whenever your application loads or modifies the calendar, it must update the text shown to users (see the `setText` function in Section 6). The calendar events *do not* have to be sorted by date/time; they may appear in any order. The on-screen calendar format is *different* to the file format. Here's an example (based on the same data as in the file format example above):

```
Veg out @ Home (1 hour, 15 minutes)
8 November 2013, 10am
---

Work on UCP Assignment @ The Labs (45 minutes)
8 November 2013, 11:15am
---

Make Huge Foreign Currency Transactions (5 minutes)
12 December 2013, 12:30pm
---

Armageddon -- Contact Bruce Willis @ Earth (432 hours)
30 June 2014, 12:59:59pm
---
```

Some points to note, for each event:

- The event name comes first.
- The location comes next, preceded by a "" character (but only if there is a location; i.e. if the location string is not empty).
- The event duration comes next, in brackets. If the duration is greater than 60 minutes, it is expressed in hours and minutes – otherwise, just minutes.
- The date and time appear on the second line. The date is written in DAY MONTH-NAME YEAR format. The time is written in 12-hour format, with "am" or "pm" as appropriate. If the minutes are zero, omit them.
- The event ends with "`---`".

# 6 Graphical User Interface

GUIs can be very complex, but you have been provided with a greatly-simplified set of functions. These are declared in `gui.h` and defined in `gui.c` (available on Blackboard):

```
Window *createWindow(char *title);
```

Creates and returns a new GUI window. This window will have space for a set of buttons on the left, and an area to display text on the right. You must specify the window title.

(The returned `Window*` pointer is a handle that you must supply to the other functions listed below. You do not need to access its fields yourself.)

```
void addButton(Window *window, char *label,
               void (*callback)(void*), void *data);
```

Adds a button to the window. You must specify:

- `window` – as returned by `createWindow`.
- `label` – a piece of text to appear on the button.
- `callback` – a function to be called later, if/when the button is pressed. This function will take a void pointer.
- `data` – A pointer to a set of data (which can be whatever you like) to be passed as a parameter to the callback function, if/when it is called. This is only means by which you can provide data to your callback function. (You *will* lose marks for using global variables.)

Note: if `callback == NULL`, the button will cause the `runGUI` function to end (in which case the `data` parameter will be ignored).

```
void runGUI(Window *window);
```

Once you have set up the window, using `createWindow` and `addButton`, call `runGUI` to hand over control to the GUI system. This will display the window and wait for user input (in the form of button presses).

When a button press occurs, the corresponding callback function you specified before will be called. The callback function itself must complete the task the user wants to accomplish (using its data parameter).

```
void setText(Window *window, char *newText);
```

Sets the text displayed by the window. Specifically, this function will erase whatever text is currently displayed, and *copy* your new text into the window.

```
int dialogBox(Window *window, char *dialogTitle, int nInputs,
              InputProperties *properties, char **inputs);
```

Displays an extra window, called a *dialog box*, with "Cancel" and "Ok" buttons and one or more spaces for user input. The parameters are as follows:

- `window` – as returned by `createWindow`.
- `dialogTitle` – the title of the dialog box.
- `nInputs` – the number of different input strings you want the user to enter.
- `properties` – an array of structs containing information on each input space. See the `InputProperties` struct below. The length of the array must correspond to `nInputs`.
- `inputs` – an array of strings to store the user input. The number of strings must correspond to `nInputs`. Each string must have enough space for `maxLength + 1` bytes (where `maxLength` is a field in the `InputProperties` struct). Any pre-existing values will used as the initial values, as displayed when the dialog box is opened. If you want the inputs to be initially blank, you must pass empty strings.

The function will return when the user presses a button – `TRUE` for "Ok" or `FALSE` otherwise. The function will not modify `inputs` unless the user presses "Ok".

```
typedef struct {
    char *label;
    int maxLength;
    int isMultiLine;
} InputProperties;
```

An array of these structs, one for each input space, is passed to the `dialogBox` function. The fields are as follows:

- `label` – a string label (prompt) to be displayed next to the input space.
- `maxLength` – the maximum number of characters the user can enter.
- `isMultiLine` – `TRUE` if the user is allowed to enter multiple lines of text; `FALSE` if only a single line is required.

```
void messageBox(Window *window, char *message);
```

Displays a simple "message box" window, with a message and a "Close" button.

```
void freeWindow(Window *window);
```

Frees a window. This should be done at the end of the program, once `runGUI` has finished.

# 7 Submission

Submit your entire assignment electronically, via Blackboard, before the deadline (i.e. 12:00 noon on Friday 8th November, 2013).

Submit one .zip or .tar.gz (*please* not .rar) file containing your Makefile and all .c and .h files.

After submitting, please verify that your submission worked by downloading your submitted files and comparing them to the originals. In the event that you encounter problems, email your assignment to the lecturer instead – david.cooper@curtin.edu.au; use the subject "`UCP120 Assignment`" – but do so *before* the deadline.

See the UCP120 Unit Outline for the policy on late submissions.

# 8 Mark Allocation

Here is a rough breakdown of how marks will be awarded. The percentages are of the total possible assignment mark:

- 20% – using code comments, you have provided good, meaningful explanations of all the files, functions and data structures needed for your *complete* implementation. (If your implementation is incomplete, you may still be awarded some marks for comments relating to functions you haven't yet written.)
- 20% – for each required piece of functionality, your code is well structured and adheres to practices emphasised in the lectures and practicals. Your code should also be separated into an appropriate set of .c and .h files.
- 30% – you have correctly implementated the required functionality, according to a visual inspection of your code by the marker.
- 30% – your program compiles (with a Makefile), runs and performs the required tasks. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data yourself.
- $-10\%$ – for each practical signoff mark of zero (pro-rata for fractional marks; e.g. $-7.5\%$ for each signoff mark of $\frac{1}{4}$).

# 9 Academic Misconduct – Plagiarism and Collusion

Copying material (from other students, websites or other sources) and presenting it as your own work is **plagiarism**. Even with your own (possibly extensive) modifications, it is still plagiarism.

Exchanging assignment solutions, or parts thereof, with other students is **collusion**.

Engaging in such activities may lead to a grade of ANN (Result Annulled Due to Academic Misconduct) being awarded for the unit, or other penalties. Serious or repeated offences may result in termination or expulsion.

You are expected to understand this at all times, across all your university studies, *with or without* warnings like this.

## — End of Assignment —