Tip 52: Mark the owner of your data

Ben Klemens

13 January 2012

level: you have many pointers to the same data
purpose: Know when to free

The trick I'll share with you here (which is not new and a common piece of lore) is to add an `owner` element to your strucure. Last time I advised that you have at least four items for every struct that gets used as a basis for data processing: a typedef plus new, copy, and free functions.

Say that you're in a situation where you have one canonical copy of the base data, and then lots of views of the data. When you free the views, they can't free the base data, because everything else will break. However, you may eventually want to free the base data at some point. So your free function has to know if you are dealing with the canonical copy or a view.

Modify the four boilerplate elements as follows:

- The type definition includes an integer named `owner`.
- The `new` function sets `owner = 1`.
- The boilerplate `copy` function could begin as a simple one-liner: `mytype copy = original`. But after making the copy, set `copy.owner = 0`.
- If `owner ==1`, then the `free` function frees all shared data; if `owner==0`, then it doesn't.

I've decided to really give you an example this time. The first block of sample code is a small library that has one structure to speak of, which is intended to read an entire file into a single string. Once you have that single string, you'll want lots and lots of substrings for all sorts of purposes, but instead of copying the potentially very long data string, the views just mark different start and end points. Having all of *Moby Dick* in a single string in memory is not a big deal at all, but we don't want to have a thousand copies of it all over the place.

The header, `fstr.h`. Notice that it requires GLib.

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>
```

```
#define stopifnot(assertion, ...) if (!(assertion)){printf(__VA_ARGS__); exit(1)

typedef struct {
    char *data;
    size_t start, end;
    int owner;
} fstr_t;

fstr_t *fstr_new(char *filename);
fstr_t *fstr_copy(fstr_t *in, size_t start, size_t len);
void fstr_show(fstr_t *fstr);
void fstr_free(fstr_t *in);

typedef struct {
    fstr_t **strings;
    int count;
} fstr_list;

fstr_list fstr_split (fstr_t *in, gchar *start_pattern);
```

The library, `fstr.c`. It is still a little bloated at 57 lines long. It uses GLib to read in the text file and for Perl-compatible regular expression parsing. You can trace the use of the `owner` element to see the steps at the head of this section in practice.

```
#include "fstr.h"

fstr_t *fstr_new(char *filename){
    GError *e=NULL;
    GIOChannel *f  = g_io_channel_new_file(filename, "r", &e);
    stopifnot(f, "failed to open file ’%s’.\n", filename);
    char *str;
    size_t len;
    stopifnot(g_io_channel_read_to_end(f, &str, &len, &e) == G_IO_STATUS_NORMAL,
            "failed to read file.\n")
    fstr_t *out = malloc(sizeof(fstr_t));
    *out = (fstr_t){.data=str, .start=0, .end=len, .owner=1};
    return out;
}

fstr_t *fstr_copy(fstr_t *in, size_t start, size_t len){
    fstr_t *out = malloc(sizeof(fstr_t));
    *out=*in;
    out->start += start;
    if (in->end > out->start + len)
        out->end = out->start + len;
    out->owner=0;
    return out;
}

void fstr_free(fstr_t *in){
    if(in->owner) free(in->data);
    free(in);
}

fstr_list fstr_split (fstr_t *in, gchar *start_pattern){
    GMatchInfo *start_info;
    fstr_t **out=malloc(sizeof(fstr_t*));
    int outlen = 1;
    out[0] = fstr_copy(in, 0, in->end);
    GRegex *start_regex = g_regex_new (start_pattern, 0, 0, NULL);
    gint mstart=0, mend=0;
    fstr_t *remaining = fstr_copy(in, 0, in->end);
    do {
        g_regex_match (start_regex, &remaining->data[remaining->start], 0, &star
        g_match_info_fetch_pos(start_info, 0, &mstart, &mend);
        if (mend > 0 && mend < remaining->end - remaining->start){ //else, no ma
            out = realloc(out, ++outlen * sizeof(fstr_t*));
            out[outlen-1] = fstr_copy(remaining, mend, remaining->end-mend);
            out[outlen-2]->end = remaining->start + mstart;
            remaining->start += mend;
        } else break;
    } while (1);
    g_match_info_free(start_info);
    g_regex_unref(start_regex);
    return (fstr_list){.strings=out, .count=outlen};
}

void fstr_show(fstr_t *fstr){
    printf("%.*s", fstr->end-fstr->start, &fstr->data[fstr->start]);
}
```

And finally, an application. To make this work, you'll need a copy of *Moby Dick, or the Whale*, by Herman Melville:

```
if [ ! -x moby ]
 then curl http://www.gutenberg.org/cache/epub/2701/pg2701.txt  \
     | sed -e '1,/START OF THIS PROJECT GUTENBERG/d' \
     | sed -e '/End of Project Gutenberg/,$d' > moby
fi
```

Great. Now let's split it into chapters, and use the same splitting function to count the uses of the words ('I') and ('whale' or 'whales') in each chapter. Notice that the `fstr` structs can be used as opaque objects at this point, using only the new, copy, free, show, and split functions. The `fstr_list` struct is really just a throwaway, and I use its contents freely.

```
#include "fstr.h"

int main(){
    fstr_t *fstr = fstr_new("moby");
    fstr_list chapters = fstr_split(fstr, "\nCHAPTER");
    for (int i=0; i< chapters.count; i++){
        fstr_show(fstr_split(chapters.strings[i],"\\.").strings[1]);
        fstr_list me     = fstr_split(chapters.strings[i], "\\WI\\W");
        fstr_list whales = fstr_split(chapters.strings[i], "whale(s|)");
        fstr_list words  = fstr_split(chapters.strings[i], "\\W");
        printf("\nch %i, words: %i.\t Is: %i\twhales: %i\n", i, words.count-1,
                me.count-1, whales.count-1);
        fstr_free(chapters.strings[i]);
    }
    fstr_free(fstr);
}
```

I won't reprint the results in detail, but will give some notes (which generally point to how hard it would be for Mr Melville to publish or even blog the book here in the modern day):

- Chapter lengths range by an order of magnitude.
- Whales really don't get discussed all that much until around chapter 30.
- The narrator decidedly has a voice. Even in the famed Cetology chapter, he uses the first person singular 60 times, personalizing what would otherwise be an encyclopædia chapter.
- GLib's regex parser is a little slower than I'd hoped it'd be.