Lab 2: More on C programming

CS 273 Monday, 8-31-20 Revision 1.2

Preliminary material

- `malloc()`, `free()`; `sizeof`
- `enum` types.
- Arrays of pointers to characters/arrays of strings.
- `struct` (see below)
- Interface/implementation modules in C (see below)
- `extern` variable declarations.

- File `structeg.c` . *See* `~cs273/egs` *for this and other examples of C programming.*

```
/* example of struct in C.    RAB  1/2001 */

#include <stdio.h>

struct date {
  int month;
  int day;
  int year;
};

/* array of month names */

char *month_name[12] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

/* get_date
    1 arg:  dp, address of an (allocated) date structure
    state change:  values are obtained interactively for the fields of *dp
    return:  integer 1 on success, 0 on failure */

int get_date(struct date *dp)
{
  printf("Enter day of the month (1-31):  ");
  if (scanf("%d", &dp->day) != 1)
    return 0;
  printf("Enter month of the year (1-12):  ");
  if (scanf("%d", &dp->month) != 1)
    return 0;
  printf("Enter year (e.g., 2001):  ");
  if (scanf("%d", &dp->year) != 1)
    return 0;

  /* successful initialization of *dp */
  return 1;
}

/* main program */

int main()
{
  struct date d;

  if (!get_date(&d))
    printf("Unable to read date successfully\n");
  else
    printf("The date you entered was %s %d, %d.\n",
           month_name[d.month-1], d.day, d.year);

  return 0;  /* normal exit status */
}
```

Notes:

- A struct definition is something like a class definition, except that there are no methods, constructors, `public/private` access specifiers, etc. `day`, `month` and `year` are the *fields* of the struct. The access to fields is always "`public`." As in C++, if `D` is a variable of the type `struct date`, then `D.day` refers to the `day` field for that variable `D`, etc.

  The struct name `date` identifies that struct, but that name must be preceded by the word `struct` in order to be used syntactically as a type (see, e.g., the argument for `get_date()`), unless you use `typedef`.
- `month_name` is an array of twelve strings. The subsequent curly bracket syntax provides initial values for those strings.
- When passing a data structures such as a `date` struct variable as an argument (or return value), you should normally pass the *address* of that data structure, for efficiency. If you want to be sure your function doesn't modify such a data structure, use `const`.
- The syntax `&dp->day` is equivalent to `&((*dp).day)`; its value is the address of the `day` field of the `date` pointed to by `dp`.
- See the `man` page for information about return value from `scanf`.
- `return`ing from `main()` is equivalent to making an `exit` system call with that return value as the argument.

• Here is a rewrite of `structeg.c` as a modular program in C.

1. File `date.h`

```
/* interface module for date data structure
   RAB 2/2018 */

#ifndef _DATE_H_
#define _DATE_H_

struct date {
  int month;
  int day;
  int year;
};

int get_date(struct date *dp);

extern char *month_name[];

#endif
```

2. File `date.c`

```
/* implementation module for date data structure
   RAB 1/2001 */

#include <stdio.h>
#include "date.h"

/* array of month names */

char *month_name[12] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

/* get_date
    1 arg:  dp, address of an (allocated) date structure
    state change:  values are obtained interactively for the fields of *dp
    return:  integer 1 on success, 0 on failure */

int get_date(struct date *dp)
{
  printf("Enter day of the month (1-31):  ");
  if (scanf("%d", &dp->day) != 1)
    return 0;
  printf("Enter month of the year (1-12):  ");
  if (scanf("%d", &dp->month) != 1)
    return 0;
  printf("Enter year (e.g., 2001):  ");
  if (scanf("%d", &dp->year) != 1)
    return 0;

  /* successful initialization of *dp */
  return 1;
}
```

3. Main program file `structeg2.c`

```
/* example of interface/implementation modules in C.    RAB  1/2001 */

#include <stdio.h>
#include "date.h"

/* main program */

int main()
{
  struct date d;

  if (!get_date(&d))
    printf("Unable to read date successfully\n");
  else
    printf("The date you entered was %s %d, %d.\n",
           month_name[d.month-1], d.day, d.year);

  return 0;  /* normal exit status */
}
```

Comments:

– In the interface module `date.h` we give `struct` definitions, `typedef`s, function headers and variable declarations. Without the `extern`, we would be *defining* the variable `month-name` instead of declaring it, which would lead to errors at link time (assuming that `date.h` is `#include`d in more than one file).
– Always `#include` system header files (e.g., `<stdio.h>`) before other header files (e.g., `"date.h"`).
– We *define* and initialize the (global) variable `month_name` in `date.c`, which was merely *declared* in `date.h`.

Laboratory exercises

Our goal is to get familiar with `getline`, `structs` and parsing of input lines.

1. Create a `lab2` directory for work on this lab, and change to that directory.

2. *(Introduction to* `getline()`*.)* Write a brief program `echoline.c` that uses `getline` to read a line of standard input then prints that input line. (Be sure to allocate your buffer, etc.)

   The function `getline()` is declared in `stdio.h`. See the man page to read about its arguments. Its third argument is a pointer to a C language *file stream*; use `stdin` for this argument to read from standard input. **Note:** The program `~cs273/egs/mopen.c` includes example use of `getline`, as discussed in class notes.

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 2 complete:  echoline.c"
   ```

3. *(Validating a* `getline` *call.)* Modify `echoline.c` if necessary to store the return value of `getline` in a variable. Then, make use of that return value to print the number of characters read, **and also** to print an appropriate message if an error or the end of input was encountered. If a nullbyte is encountered in standard input before a newline character, your program should read and count all characters *including nullbytes* before that newline. Test all these features.

   Note: You can enter an end-of-input at a terminal by entering `CTRL/D` at the beginning of a line. You can enter a nullbyte at the terminal (at least on a Link computer) by entering `CTRL/@` (anywhere in a line). Even if your `echoline` program doesn't print characters after a nullbyte, it should print the correct count of characters in the input line including nullbytes.

   If you can figure out a way to cause an input error (not an end-of-input) for this testing, explain how you caused that input error in a file called `README` in your `lab2` directory.

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 3 complete:  echoline.c"
   ```

   Include `README` in the `git add` command above you have something to commit in that file `README`.

4. *(Introduction to* `struct`*; moving I/O to a function.)* Modify `echoline.c` to define a struct `name` with the following fields:

   - An array `tok` of pointers to characters, eventually intended to hold the words ("tokens") in an input line. Use the type `char**` for this field, so we can load it with dynamically allocated memory later.
   - An integer `count`, eventually intended to hold the number of words encountered in an input line.
   - Another integer `status`, eventually intended to hold information about the success of a parsing operation.

   Also define a function `read_name` with (the address of) one `name` struct as its argument, which will ultimately read a line of input and store its tokens in that `name` structure. `read_name` should return an integer (for indicating success or failure).

   For now, move your code for reading, checking and echoing an input line into `read_name`, and change your `main` to allocate a `name` struct then call `read_name` with that struct, as a preliminary step and to get syntax correct. Note that this preliminary verison of `read_name` will not actually use the struct `name` at all in this step. Compile and execute the resulting program to make sure it still operates correctly.

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 4 complete:  echoline.c"
   ```

5. *(Allocating an array of pointers with* `malloc()`*; using an element of an array of pointers.)* Modify your `read_name` function in `echoline.c` to dynamically allocate an array of `MAXTOKS` pointers to strings and assign that newly allocated memory to the `tok` field of the `name` struct

pointed to by the argument of `read_name`. Use `#define MAXTOKS` as a preprocessor constant. Use `sizeof` in your computation of how much space to allocate for your array of pointers.

Then, in `read_name()` after your call to `getline()`, make an assignment causing `tok[0]` (first element in the newly allocated array `tok` within your `name` struct) to hold the address of your buffer array used in your `getline()` call. (Remember you can express the *address* of an array by simply writing that array's name.) Finally, modify your `read_name` function to echo the value in `tok[0]` instead of echoing your input buffer (thus, `getline` still reads into your input buffer but output uses the value of `tok[0]` to print that same buffer). Verify that your program works.

Create a git `commit` containing your work on this problem.

```
$ git add echoline.c
$ git commit -m "Step 5 complete:  echoline.c"
```

6. *(Enumerated type; using a* `status` *field.)* Further modify `read_name` to store information about the success of the input operation in the `status` field of the `name` struct. For example, if end of file (or an error) was encountered before any characters could be read, assign a constant value `EOF_OR_ERROR` to `status`; otherwise, assign `NORMAL` to `status`. Define these constants using an enumerated type, e.g.,

   ```
   enum status_value { NORMAL, EOF_OR_ERROR };
   ```

   Also, have `read_name` return 1 for `NORMAL` status and 0 for other values of `status`, to indicate success or failure.

   In `read_name`, eliminate the printed messages indicating validation that were added in Step 3. Then, in `main`, add similar diagnostics according to the value of the `status` field of your `name` struct. Also, print the return value from `read_name` within `main`. Test your program, including tests for end-of-file, lines containing nullbytes, and the normal case.

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 6 complete:  echoline.c"
   ```

7. *(Copying first token into a dynamically allocated string.)* Now start parsing by storing only one token from the input line in `tok[0]` instead of the entire input line, as follows. In `read_name`, instead of assigning `getline()`'s buffer to `tok[0]`, write a loop that counts the number N of characters in that buffer from the beginning of that buffer and until you encounter a whitespace character (consider the library function `isspace`, which checks for spaces, tabs, newlines, etc.) **or** the end of that buffer. Dynamically allocate a new array of $N + 1$ characters (allowing for added nullbyte), assign that dynamically allocated array to `tok[0]`, then use a second loop to copy the first N characters of `getline()`'s buffer into `tok[0]` and add a nullbyte. In the end, `tok[0]` should hold a dynamically allocated copy of the first token in `getline()`'s buffer, without changing that buffer.

   Be sure that your first (counting) loop stops if `getline()`'s buffer is exhausted before the first newline (you can detect this by comparing the number of characters counted against the return value from `getline()`). Also, be sure to add a nullbyte at the end of your newly-parsed token. Test the results in `main` by printing the value of `tok[0]` in your `name` struct.

Create a git `commit` containing your work on this problem.

```
$ git add echoline.c
$ git commit -m "Step 7 complete:  echoline.c"
```

8. *(Extracting all the tokens.)* Finally, change `read_name` to read all the tokens in a line. This will require you to enclose both the allocation of `tok[i]` and the loop for copying tokens within a larger "tokenizing" loop. If multiple whitespace characters appear between two tokens, be sure to skip all consecutive whitespace characters between such tokens (this will require an additional inner loop). You will know to exit from the outer loop when you encounter the end of your buffer. Store the total number of tokens you encounter in the `count` field of the struct `name`.

   Test the results by printing all the parsed tokens in `main`, one per line, then by printing the last token followed by a comma and the other tokens (so that the input line `Edsger W. Dijkstra` will produce the output `Dijkstra, Edsger W.`).

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 8 complete:  echoline.c"
   ```

9. *(Detecting too many tokens; evolutionary modification of* `status`*.)* Modify `read_name()` to detect when there are more than `MAXTOKS` tokens in input. If this occurs, exit from your "tokenizing" loop, assign a dynamically allocated copy of the remainder of `getline()`'s buffer (containing multiple tokens) to the final location in the array `tok[]`, and assign a value `TOO_MANY_TOKENS` to your `status` field. You will need to add the name `TOO_MANY_TOKENS` to your `enum` type. Add detection of this condition in your `main`. Test this feature by temporarily changing the value of `MAXTOKS` to 3, and entering lines with 2, 3, then 4 tokens.

   Create a git `commit` containing your work on this problem.

   ```
   $ git add echoline.c
   $ git commit -m "Step 9 complete:  echoline.c"
   ```

10. *(More improvements.)* Make your program robust. Test to insure that your program doesn't create any empty tokens, e.g., if someone enters spaces at the beginning or end of an input line. (You will need to skip spaces before extracting the first token.)

    Also, make sure your program behaves correctly when nullbytes are included within an input line. If a nullbyte appears within a token, your program should copy that nullbyte along with the other characters of the token (although `printf()` will only output characters before the first nullbyte), and any future tokens should also be copied into `tok[]`. If a nullbyte appears among consecutive whitespace characters, your program should skip that nullbyte along with those whitespace characters.

    Is there anything else that can go wrong that your program does not address?

    Create a git `commit` containing your work on this problem.

    ```
    $ git add echoline.c
    $ git commit -m "Step 10 complete:  echoline.c"
    ```

Deliverables

Use `stogit` to submit your work for this lab. Assuming you have `committed` your work as above, enter

```
% git commit --amend
% git pull origin master
% git push origin master
```

If your lab is complete, your amended should be `lab2:  complete`. Use an appropriate message for your `commit` if your lab isn't yet complete. (You can make a partial submission at any time, then perform the same `add/commit/pull/push` cycle to resubmit more complete versions later.)

You can examine the results of your `push` operation by browsing to `stogit.cs.stolaf.edu`, logging in (with your St. Olaf password), clicking on a link for your repository (e.g., `OS/F20/rab`), and browsing using the `Files` option in the upper menu bar.