Lab 1: Programming in C

CS 273 Monday, 8-24-20 Revision 1.2

Preliminary material

- Using manual pages. For example,

```
% man gcc
% man read
% man 2 read
```

  Viewing online manual pages on a browser.

- *Similarities between C++ and C.* C++ language shares most of its same basic syntax with C. Thus, simple variable declarations, assignments, loops, array access, function definitions/declarations/calls, etc. are written the essentially the same way in both languages.

  Much of the basic semantics is shared, also. Thus, a `for` loop behaves as expected, as does an expression statement, a function definition, etc.; also, operators have the same precedence and associativity.

  The two languages both use predefined types `int`, `long`, `float`, `char`, etc. In both languages, ASCII characters are represented as (8-bit) integer values, and simple strings are represented as arrays of characters terminated by nullbytes.

  Preprocessor directives (such as `#include`, `#define`, etc.) are used in both C and C++.

- *Differences between C++ and C.*

  - **C has no classes.** Instead, you must use functions and variables to perform operations. It is still useful to use "interface modules" and "implementation modules" and to organize your program into packages that can be linked separately. However, there are no classes to define per se.

    C does provide `struct` (and `union`) for data structures. Unlike C++, a `struct` is not a class, but instead is a container for variables of various types. There are no methods, no constructors, no destructors, etc.
  - **No overloading of functions or operators in C**. Many conventional meanings of operators in C++ (such as `<<`) are no longer relevant.
  - **No `new` and `delete`.** However, C does provide library functions (instead of these C++ operators) for dynamic memory allocation and deallocation, called `malloc` and `free`. The argument for `malloc` is the number of bytes to be allocated; these operations don't know about types. (However, the `sizeof` operator is available to determine the number of bytes in a data structure.)
  - **No reference types.** If you want to pass a large data structure as an argument to a function, pass that data structure's address using a pointer type for that argument.
  - **Miscellaneous syntax differences,** such as an inability in C to define a loop-control variable within the first part of a `for` loop.
  - **No templates, etc.** Many bells and whistles of C++ are not present in C.

– **Output without `cout`.** C++'s easy output using `cout` and `<<` is certainly not available, since it involves the class `ostream`. Instead, C provides a standard library for input and output called `stdio.h` (accessible through `#include <stdio.h>`). The most common way to perform output using this library is the function `printf`. For example, the C instruction

```
printf("Name:  %s;  age:  %d\n", "John", 19);
```

prints one line of output providing the name and age of a person with labels.

```
Name:  John;  age:  19
```

The first argument of `printf` is a format string, in which sequences of characters beginning with `%` indicate where data values are to be inserted. The two-character sequence `%s` indicates that a string is to be inserted; since this is the first insertion, the first argument after the format string will provide the value to be inserted. The sequence `%d` indicates that an integer should be printed in decimal. The sequence `%o` is for printing octal integers, `%x`— for hexadecimal integers and `%f` for floating point numbers. These sequences within the format string may be modified with additional characters to indicate the number of characters to print, precision for decimal quantities, left justification, etc. The sequence `\n` causes a newline to be output.

The library function `printf` prints to standard output. `fprintf()` may be used to print to other files (but then we'd have to talk about how to open files...); `sprintf()` allows one to place formatted output in a (preallocated) array of characters.

For documentation, the traditional Linux documentation command is `man` (for "manual"):

```
% man 3 printf
```

on a Linux machine that has manual pages installed, or in a browser search engine. (Here, `3` refers to the library manual section; section `2` is for system calls, and section `1` is for user programs.)

Output may also be performed using system calls directly or with other output libraries. Note that `printf` provides *buffered output*, which is collected and performed in groups of bytes for efficiency, whereas system calls such as `write()` are not only less convenient (e.g., you must convert integers to ASCII characters yourself) but also provide no buffering.

– **Input without `cin`.** There is a function in `stdio.h` called `scanf` for obtaining values from standard input and storing them in variables. For example,

```
int n;
char arr[20];
scanf("%d: %s", &n, arr);
```

will attempt to read an integer and a string from standard input. If the standard input holds `22:  Paul Smith` then 22 will be assigned to `n` and the string `Paul` will be assigned to the first five bytes of the array `arr`.

Whitespace characters are ignored in `scanf` format strings, and are skipped in input by `scanf`. Non-whitespace characters in the format string must be matched exactly in

input. `NOTE:` **ADDRESSES** of variables must be passed as the arguments of `scanf` after the format string. Also note that the array `arr` was allocated; `scanf`ing with a pointer variable that doesn't point to an allocated array is a serious memory management error.

There are versions of `scanf` for reading from files (`fscanf`) and from strings (`sscanf`). `scanf` (and `fscanf`) perform buffered input, whereas system calls such as `read` perform unbuffered input.

In practice, `scanf` is often a little messy to work with. It often works better to read a whole line of input into an (**allocated!**) array of `char` using `fgets`, then to parse that array. Such an array is generally called a *buffer*. Example of using `fgets`:

```
#define MAX 100;
int count;
char buff[MAX];
count = fgets(buff, MAX, stdin);
```

This reads up to 99 characters into `buff`, including a newline if it is among the first 99 characters; the input is followed by a nullbyte.

Laboratory exercises

1. Login to a Link CS computer. View some `man` pages for common Linux using a search engine on a browser.
2. Type in the program `reverse.c` below; compile it, debugging it as needed, and get it running. Check the exit status using the command `% echo $status` (assuming you are using `csh` or `tcsh`).

```
/* example of C language programming.  R Brown 1/18 */

#include <stdio.h>
#include <stdlib.h>  /* for exit() */

int main(int argc, char **argv, char **envp)
{
  int i = argc;  /* index into array argv[] */
  while (i--) {
    printf("%s ", argv[i]);
  }
  printf("\n");

  exit(argc);  /* normal exit status would be 0 */
}
```

To compile the program:

```
% gcc -c reverse.c
% gcc -o reverse reverse.o
```

Then, you can run your new program `reverse` with command-line arguments. The program will print all command-line arguments back, but in reverse order. Example:

```
% reverse a b c d e
e d c b a reverse
```

Observe that the program name "reverse" is printed, too. In Linux, the program name is the command-line argument with index 0.

Note: This short program contains a number of features that foreshadow topics that will arise later in the Operating Systems course. Ask now if you're curious.

3. Write a C language program `table.c` that prompts for an input integer, reads one integer N and prints a table of the first N positive integers and their squares, one per line. Label your table with a header; use tab characters (`\t`) or `printf` field width formatting so that the columns will line up; include a vertical bar `|` between columns. Example output:

```
Enter a positive integer:  3

N        | N*N
-----------------
1        | 1
2        | 4
3        | 9
```

Deliverables

All homework and labs and most projects in this course will be submitted using `stogit`, which is St. Olaf CS's `git` server. The document

`http://www.stolaf.edu/people/rab/os/stogit_startup.html`

explains how to set up a working directory `~/OS` to contain your work connected to your `stogit` repository for this course. Carry out those steps, then perform the following:

1. Create a subdirectory `~/OS/lab1` for this assignment, and move your files for this assignment to that subdirectory. (Note that the working directory `~/OS` already exists after your `stogit` setup above.)

```
% mkdir ~/OS/lab1
% mv reverse* table* ~/OS/lab1
```

2. Create a `git commit` ("snapshot" of your files, and upload it to your `stogit` repository.

```
% cd ~/OS/lab1
% git add reverse.c table.c
% git commit -m "Lab 1 complete"
% git pull origin master
% git push origin master
```

Notes:

- The `git add` command stages your source code files `reverse.c` and `table.c` for inclusion in the subsequent `commit` snapshot. **Your git repository should contain only source code files**, not `.o` files or executables.
- The message `"Lab 1 complete"` specified when performing `git commit` should reflect the contents of that `commit` snapshot of files. This informs both you (in case you need to find a prior commit) and the graders. If your lab work is not yet complete, replace that message by an accurate description of what that `commit` contains.
- The `pull` operation not only retrieves the repository contents but also *merges* those contents with your commit, creating a new combined commit snapshot if needed. If `git` creates a new "merge commit," an editor window will appear for entering a message for that new commit snapshot. In that case, exit from that editor to accept the default message. (For the `vi` editor, exit by entering the characters `:wq` to write and quit.)
- You can examine the results of your `push` operation by browsing to `stogit.cs.stolaf.edu`, logging in (with your CS-managed password), clicking on a link for your repository (e.g., `os/f20/rab`), and browsing using the `Files` option in the upper menu bar.