Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Lab 4, February 9/10

In this lab, you will first get some experience in using the job control mechanism of the Bash shell. Next, you will first learn how to use `man` to get help on C library functions. You will then learn more about conversion specifications used in the `printf` function in C. During this process, we will also learn how to divide the screen of `emacs` vertically. After that, you will see an example that will help you understand how `scanf` works. Finally, you will be asked to use `printf` and `scanf` to write a program.

Be sure to get help from teaching assistants whenever you have any questions.

1. First, perform the following steps to get started:

   (i) Login to server bluenose.cs.dal.ca via SSH from a CS Teaching Lab computer or from your own computer.

   (ii) Change your current working directory to the `csci2132` directory created in Lab 1.

   (iii) Create a subdirectory named lab4.

   (iv) Change your current working directory to this new directory.

   (v) Copy the `hello.c` file we created in Lab 3 from directory `lab3` to the current directory.

2. In Lab 3, we learned the `emacs` hot key that allows us to run any command without exiting `emacs`. Now, instead of invoking shell command inside `emacs`, we learn how to send `emacs` to background and then bring it back to the foreground. This is useful if you wish to stop programming to do a list of tasks, and then resume your work later.

   To get started, use `emacs` to open `hello.c`. Now, perform the following steps (reviewing lectures regarding job control might be helpful):

   (i) Enter Control-z. What does this do?

   (ii) We are now using the shell. Now, enter commands to compile and run this program.

   (iii) Use what you learned in class about job control to bring `emacs` to foreground again. Do NOT simply enter `emacs`! Hint: These two commands will be helpful: `jobs` and `fg`.

3. The UNIX `man` command can be used to display information about C library functions, such as `printf` and `scanf`. However, when you type the command `man printf`, you will see the manual page of a UNIX utility called `printf`, not the C function with the same name. Enter the above command, and take a look at the information displayed on the screen.

   Why would this happen? This is because the entries in the UNIX manual are grouped into sections. Section 1 is for commands and application programs, while Section 3 is for library functions. When we use the `man` command to look for a word, by default, `man` displays the first entry that it finds, starting from Section 1. That's why the manual page for the command `printf` is displayed.

   How do we retrieve the entry for the C `printf` function? To do so, we need specify the section number. Press `q` to quit the previous manual page if you have not done so yet. Then, enter the command `man 3 printf`. This will bring up the manual page on the C function `printf`. Use space bar and page-up / page-down keys to scroll up and down the entry. You can even see an example. Thus, the `man` command is very useful when we program in C in UNIX.

   Try this for the C `scanf` function as well.

4. In class, we learned the `printf` function. We have seen the following statement:

   ```
   printf("Profit: %.2f\n", profit);
   ```

   In this example, the first parameter of `printf` is a string literal `"Profit: %.2f\n"`. This string literal is used as the *format string* of the `printf` function in this statement, as it specifies the output format. In this string, `%.2f` is a place holder indicating where the value of `profit` is to be filled in during printing, and in what format this value will be printed. The official name for this place holder is *conversion specification*. A conversion specification is of the form `%m.px` or `%-m.px`.

   There is a set of rules regarding the conversion specifications for `printf`, which are easy to follow. We have seen some examples during the lectures, and now let's learn more about these rules in this lab.

   First, the character `m` in the syntax for conversion specifications is called the *minimum field width*. It specifies the minimum number of characters to print. There are two cases that we need pay attention to:

   (i) When the value to print requires fewer than m characters, additional space characters will be printed so that m characters will be printed. Again, there are two sub-cases: if "-" is not used in the conversion specification, the value is right-justified within the field when it is printed. Otherwise, it is left-justified.

   (ii) When the value requires more than m characters to print, the field width will automatically expand so that no digit is lost when printing.

Let's run an example to see what exactly the above two cases mean. Write a program called `testprintf.c` with the following as its content:

```
#include <stdio.h>

int main(void) {
  int value1 = 123, value2 = 12345;

  printf("[%4d]\n", value1);
  printf("[%-4d]\n", value1);
  printf("[%4d]\n", value2);
  printf("[%-4d]\n", value2);

  return 0;
}
```

Compile and run this program. During this process, make use of what we learned in lab 3, i.e. do not exit `emacs` to compile, but do it slightly differently: Before we compile, maximize your terminal and press Control-x and then 3. This will divide the emacs screen into two windows vertically (Control-x 1 can be used to go back to one-window mode). Then compile your code. This way the output of `gcc` will be displayed on the right window, so that you will see about the same number of lines of your original code. As these days wide-screen monitors are popular, this is often more convenient than dividing the screen horizontally. Maximize your terminal window to fully take advantage of this.

Check the output. Explain why each line has been printed as shown on your screen by making use of what you learned above.

5. The `p` in the syntax for conversion specifications is called *precision*. What it means depends on `x`. We already learned what it means when `x` is `f` (What does it mean?), so now let's learn what it means when `x` is `d`.

   In this case, the precision specifies the minimum number of digits to display. Note that here it uses the word "digits", not characters. When p is larger than the number of digits in the int value, extra 0's are added to the beginning of the number.

   Add the following statement to the `testprintf.c` program, before the return statement.

   ```
   printf("[%6.4d]\n", value1);
   ```

   Explain what you see on the terminal.

6. In class, we learned in details about how `scanf` works. Let's now run a program and see if you can explain why its output is what you see on the monitor. Name your source program file `testscanf.c`.

```
#include <stdio.h>

int main(void) {
  int a;
  float x, y;
  int b;

  scanf("%d%f%f%d", &a, &x, &y, &b);
  printf("%d %.1f %.1f %d\n", a, x, y, b);

  return 0;
}
```

   Compile and run your program. When the program waits for your input, enter `12.5 3e6 2.2 4`. Separate the numbers by spaces.

   Explain why the output is what you see on the screen.

7. Let's make use of what we learned today to write a program. This is project 4 on page 50 of the textbook. It asks you to write a program that prompts the user to enter a phone number in form (xxx)xxx-xxxx, and then prints the same number in the form xxx.xxx.xxxx to stdout.

   An example of running this program is:

```
Enter a telephone number [(xxx)xxx-xxxx]: (902)494-9999
The number is 902.494.9999
```

   Test your program thoroughly. One test case you need consider is (902)494-0001.

8. By now, you have finished the required work of this lab. Assignment 3 is due at 3:00PM next Wednesday. Work on it if you have not finished it yet. Also work on the practice programming questions (see the practice question page on the course website).