ECE 206, Fall 2001: Lab 6

C Programming

Learning Objectives

In this lab, you will write a fairly simple C program. This program will give you practice using arrays and I/O library routines.

Background

In this section, we'll give you some information about C and about the lab environment that will help you complete your program. These concepts may seem unrelated when you first read them, but (hopefully) you'll see the relevance when you actually complete the program. It will be helpful for you to read the Lab Exercise first.

Standard C I/O Library

In class, you've seen the `printf` and `scanf` functions, which are part of the standard I/O library. When you use these functions you need to put the following line near the beginning of your program, so that the compiler will include the definitions of the I/O routines (and other useful things) from the standard I/O header file:

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

Another useful function in that library is `getchar`, which reads one character at a time from the standard input stream. (To be more precise, `getchar` is actually a macro, not a function, but that's not important.)

Standard in and *standard out* are the names of the default input and output streams that are opened whenever a C program is executed. Standard in typically gets input from the keyboard, and standard out typically prints results to the screen. In Unix systems, you can "redirect" these streams to/from a file. For standard in, for instance, the program "thinks" you're typing things in at the keyboard, but the characters are actually coming from a file that you prepared earlier.

To redirect standard in, use the '<' character":

```
% prog <infile
```

To redirect standard out to a file, use the '>' character:

```
% prog >outfile
```

You can even do both at the same time:

```
% prog <infile >outfile
```

Now back to `getchar` – this function reads in one character at a time from standard in, and *casts* it to an integer. (See the section on type casting below.) This <u>*does not*</u> mean that it converts the ASCII character ('0') to its integer equivalent (0); it means that the

ASCII character ('0') is assigned an integer type, so that it's now the same as the number 48 (or 0x30).

If `getchar` finds that there are no more characters to be read, because you've reached the "end of the file", a special value called EOF is returned. The standard I/O header file defines the symbol EOF for you to use in your program. (This is one reason that `getchar` returns an integer, rather than a character. There is no EOF *character*; it's just a special *integer value* that many of the I/O routines use.) When typing on the keyboard, you can create the end-of-file condition by typing Control-D. (That's the Control key and the 'D' key at the same time.)

So to read in a sequence of characters, you can do the following:

```
int c;

while ((c = getchar()) != EOF) {
  /* do your character processing here */
}
```

Notice the use of the assignment operator inside the while condition expression. The return value of the assignment operator is the value that was assigned.

You could accomplish the same thing with the scanf function. The return value of the scanf function is the number of input fields that were successfully assigned. If the end-of-file condition was encountered, however, the value EOF is returned. So we could do this:

```
int i;
char c;

while ((i = scanf("%c", &c)) != EOF) {
  /* do your character processing here */
}
```

Question: Why would we not want to do something like `scanf("%d", j)` for this lab exercise?

One other tidbit: To print a percent sign using `printf`, use "%%" in your format string. A single % will be interpreted as the first part of a formatting code, like "%d".

Data Type Conversion

In the text above, we mentioned something about "casting" from a character to an integer. This is the term used for converting a value from one data type to another.

Some type conversions are done automatically (and implicitly) by the compiler when evaluating expressions. For example, when an integer and a floating point number are involved in an expression, the integer is converted to a floating point value before the operation is applied.

```
double x;

x = 10.4 / 6;
/* 10.4 is double, 6 is an int, result is double */
```

There are many such rules about converting types in an expression, and it can get fairly complicated. Here are some examples:

- If either operand is a `double`, the other is converted to `double`, and the result is of type `double`.
- If either operand is of type `char`, it is converted to `int`, and the result is of type `int`.

Sometimes, however, you'll want to explicitly convert one type to another. You can do this by putting the desired type name, in parentheses, before the value (or variable) to be converted. This is known as the *casting* operator.

```
double x;
int i,j;

x = (double) i / j;
/* otherwise, integer (truncating) division
   will be used */
```

The casting operator has a higher precedence than division. So in the above example, it's only the "i" variable that is being cast to a `double`. The automatic conversion rules will then convert "j" to a `double` and create a `double`-typed result. If the cast was not performed, then "i/j" would be performed with integer division, and then the integer result would be converted to a floating point during the assignment.

So to relate this discussion to what we covered earlier, getchar reads a character from standard in and casts it to an integer before returning it. In many cases, integers and characters can be used interchangeably (because a `char` "fits inside" an `int`). But it's good coding practice to explicitly cast from one type to another when necessary, so that your *intent* is captured in the program.

Arrays

An array is a data structure that allows a sequence of same-typed values to be associated with a single variable name. An array is declared as a particular type, followed by the number of elements in the array:

```
int a[32];  /* an array of 32 integers */
```

The elements of the array can then be accessed as `a[0]`, `a[1]`, …, `a[31]`. Note that numbering starts with zero, not with one. The exact number of elements must be specified when the array is declared.

Compiling and Running C Programs

We will be using the GNU C compiler for this lab, which is named "gcc". (On EOS, you could also use "cc", which is the Sun compiler, and there should be no difference.)

On the machines in the lab:

1. Open up a Cygwin shell window. (Your lab TA will tell you how.) This runs a Unix-like shell on top of Windows.
2. Create a temporary directory and go there, as in:
   ```
   mkdir mytmp
   cd mytmp
   ```
3. Use a text editor, like Notepad or LC2Edit, to create your C program, and save it into your temporary directory. (If you know vi, you can run that directly from the shell.)
4. Assuming your program is named "lab6.c", you can compile it as follows:

   ```
   gcc lab6.c
   ```

   This will create an executable called "a.exe". If you want the executable to be named something else (like "lab6"), compile this way:

   ```
   gcc –o lab6 lab6.c
   ```
5. To run the program, type the following at the shell prompt:

   ```
   ./a.exe
   ```

   (If you named the executable something else, use that name instead.)
6. When you're done, copy your program to your floppy disk. (Or you can ftp it to your EOS account on `ftp.ncsu.edu`.) As usual, you must submit your program electronically via Wolfware.

On an EOS workstation:

1. Use a text editor, like emacs, vi, or SlickEdit, to create your C program.
2. Make sure that `/usr/local/bin` is in your search path. (That's where gcc lives.)
3. Assuming your program is named "lab6.c", you can compile it as follows:

   ```
   gcc lab6.c
   ```

   This will create an executable called "a.out". If you want the executable to be named something else (like "lab6"), compile this way:

   ```
   gcc –o lab6 lab6.c
   ```

4. To run the program, type the following at the shell prompt:

   ```
   ./a.out
   ```

   (If you named the executable something else, use that name instead.)

5. As usual, you must submit your program electronically via Wolfware.

Prelab Exercises (30 pts)

There is no prelab as part of this assignment.

If you turn in a solution to the lab exercise, you will receive a free 30 points. (Yay! Hooray!)

If you do not turn in a solution to the lab exercise, you will receive a zero. (Boo! Hiss!)

Laboratory Exercise (70 pts)

Write a C program that does the following:

1. Read in characters from standard in until end-of-file (EOF).
2. Keep track of the number of times each decimal digit is read. (How many times was '0' read? '1'? And so forth.) If a non-digit character is entered, ignore it.
3. Once the EOF character is read, print the number of times each digit was read, along with the percentage of occurrence of that digit. For example, if the input looks like this:

   ```
   012034
   501xyz9
   ```

   then the output will look something like this:

   ```
   0:  3  (30.0%)
   1:  2  (20.0%)
   2:  1  (10.0%)
   3:  1  (10.0%)
   4:  1  (10.0%)
   5:  1  (10.0%)
   6:  0  (0.0%)
   7:  0  (0.0%)
   8:  0  (0.0%)
   9:  1  (10.0%)
   ```

4. Your program *must* use an array to keep track of the counts for each digit.

Submit your program as a file named "lab6.c".

Extra Credit (10 pts)

Submit a separate program (called "lab6-extra.c") that uses a *pointer*, rather than array notation, to access the digit counts (both for setting and for printing). If you don't know how to do this now, you should by the time the lab is due.