Lab3: Prelab

Man Pages

The man command, short for 'manual', is a unix command to get more information about a particular command or function. There are many sections of man pages, which we can think of as volumes. Each of these volumes contain information about a certain class of commands, like c functions, while another volume will contain information about another set of commands, like bash functions. This is important because sometimes there are functions of the same name in multiple volumes, e.g. printf is a c function and a bash command! In general, you will get the correct page, but it is good to know that there may be more than one version of that same command.

To look up the man page for a given command, you simply type "**`man {command}`**", where {command} is the command you want more information about. There is even a man page for man! These pages are especially useful for you because of the type of information that you can get from one. Let's look at an example.

type "`man puts`" into your terminal.

puts is a rather simple command, so we can look up at that page for a general idea of how a man page is set up.

1. At the very top left, we can see PUTS(3), this tells us that we are in the man page for puts in the third "volume", which is the one that contains information about c functions.

2. If we go down, we see that there is a Name section that has a lot of names that are kind of like puts. This are all the functions in the puts "family", so to speak. They all do roughly similar things, writing a character or string, but there are some that write to a given file instead of the screen, these are the ones that start with an "f", which, probably unsurprisingly, stands for file.

3. Next, we see the Synopsis. This is useful because it will tell you where the function exists, that is to say which header file must be included to use it, as well as the signatures of all the methods in the family, so you know what kind of arguments they each take. You'll notice that this section does not explain *what* the functions do. That is in the next section, since there can sometimes be a lot of members of the family, and their descriptions can get a bit long at times.

4. Now we get to the Description. This is where you'll be told what the function actually do. Example: fputc writes the character c, cast to an unsigned char, to stream. Notice how the parameters are underlined. unsigned char is underlined in this case as the cast can be important since you are passing an integer. You'll notice that some of these functions are basically wrappers around other ones, e.g. putchar is really just putc(c, stdout).

5. One of the most useful sections when you are using the function is the Return Value. We know what type the function returns, it's in the signature, putc returns an int, but what does that int *mean*? You'll see that the int returned by the function is actually the character written, but that's kind of weird, it's just returning what we passed it? Not quite, if we look right after that, we see that if there was an error, it will return EOF. This is a special "character", a defined name for -1, that is returned at the End Of File by many functions, so it can be checked for directly, as is shown in the slides.

We can skip the Attributes and Conforming To sections for right now, as they deal with information that is not yet important, such as how the function deals with threaded programs and the standards that use it.

Also useful is the See Also section, as it can direct you to similar functions, such as write, which also writes things to the screen! or fgets which can help you read from a file.

Exercise

- Take a look at some of the files you used in labs 1 and 2 and look up the man pages for some of the functions(be warned, printf has a very long one, don't worry if it doesn't make sense right now, it will)

ASCII:American Standard Code for Information Interchange

Since, computers only understand numbers, an ASCII code is used to numerically represent characters. You can refer to the ASCII table on slide 54 of class notes. You can also google ASCII table to know more details.
For example, a space is represented by the decimal/number 32. The character 'A' is represented by the decimal 65.

Now, if I write

```
int num = 65;
```

Since the data type here is int, 65 is understood as a number by the computer.

But, if I write

```
char num = 65;
```

Since the data type here is char, 65 is understood as a character by the computer.

The fun part begins when we try to print these.

Try it yourself, see what the output of this is:

```
int num1 = 65;
char num2 = 65;
printf("%d\n", num1);
printf("%c\n", num2);
```

In C, we have a library function called atoi. This function **int atoi(const char *str)** converts the string argument **str** to an integer (type int).

NOTE: Read more about atoi at-
http://www.tutorialspoint.com/c_standard_library/c_function_atoi.htm

But there is no function itoa that could potentially convert an int into a string. So now the problem boils down to how to convert a number into a string in C.
Think about ways of doing that for Lab3.

Exercise

- Now that you know about man pages, you can also look at the man page for atoi.

- You will probably end up using atoi as well, so write a small program that will take a string as a command line argument and, using atoi, increment each letter by 3, wrapping around if it is no longer a letter and print it back to the user.

- You can reference an ascii chart to make sure that the letters are correct, be aware of the wrapping and the change between cases.

Base Conversion

As computer scientists, you should all be aware of binary, or $base_2$, and you are all aware of decimal, or $base_{10}$, that we use in our everyday lives. Knowing about bases is great, but what would be better is to be able to translate numbers between two bases. before we can do that, let's do a quick overview of what a base actually is.

When we use the term "base", we are referring to the fact that at each position there is a digit representing the number of lots of the base number to the power of that position, zero indexed. That's a lot of words, so let's do a bit more of an example.

- In decimal($base_{10}$), we can have the number 13_{10}(meaning "13 in $base_{10}$"). This means that we have 1 lot of 10^1, which is 10, and 3 lots of 10^0, which is 1. You'll notice, or already be aware, that we can only have numbers 0-9 in any position in $base_{10}$, because if we have 10 lots of 10^x, we actually have 1 lot of 10^{x+1}.

- This is the same for binary, let's look at 101_2(meaning "101 in $base_2$"). Starting from the right, we have 1 lot of 2^0, 1, 0 lots of 2^1, and 1 lot of 2^2, 4. So this number is 5 in decimal. We just converted from $base_2$ to $base_{10}$!

This is a really easy middle step for base conversion, because we think in $base_{10}$ anyway, so if we know how to convert any base to $base_{10}$ and from $base_{10}$ to any base, we can go from any base to any base.

You know what would be awesome? Let's do it the other way!

We have 13 in $base_{10}$.

- We can build the binary from right to left. So, we divide by 2, 13/2 = 6 Remainder 1. We want that remainder, so we will have a 1 in the 2^0 column. Now we have 6 left, well 6/2 = 3 R 0, so there will be a zero in the 2^1 column. We divide by 2 again, 3/2 = 1 R 1, so there is a 1 in the 2^2 column. Now we only have 1 left, 1/2 = 0 R 1, R 1 means that the 2^3 column will have a one, and since we got 0, that means we're done. So, 13_{10} in binary is 1101.

- Now we can convert back to $base_{10}$ and make sure we're right. 1 lot of 2^3, 8, 1 lot of 2^2, 4, 1 lot 2^0, 1, if we add that all up, 13. We did it!

NOTE: we can use bases above 10, if we need to put a number greater than 10 in a single place, we use letters, a=11, b=12, and so on. So, we can conceivably have up to base 36!

Exercise

Now try some on your own, there are plenty of online converters you can use to check your answers.

- 10_2 to base 16
- DE_{16} to base 10
- 123_4 to base 12
- $DEADBEEF_{16}$ to base 2

Things to Review:

- This week's lecture slides http://courses.cs.purdue.edu/cs24000:lectures
- Understanding of basic if/else and conditionals
- Understanding of basic loops