I/O Streams

Now, we'll delve into the input/output methods provided by the C standard library and see how we can interact with the command-line terminal.

We'll cover the following

The standard C library (linked to using `#include <stdio.h>`) includes a handful of functions for performing input and output in C.

C and UNIX make use of a concept called **streams** in which data can be sent and received between programs, devices, and the operating system. We can distinguish between two types of streams: text and binary.

Text streams are made up of lines, where each line has zero or more characters, and is terminated by a new-line character `\n`. Binary streams are more “raw” and consist of a stream of any number of bytes.

C programs all have three streams “built-in”: **standard input**, **standard output**, and **standard error**. When you interact with a C program in a terminal, and you type values in, you are using standard input.

UNIX has a concept called **pipes** (e.g. see here) whereby you can redirect any stream (e.g. output from some program or device) as input to another process (e.g. your program). When you redirect a stream to your program, which takes it in as input, you are using the standard input stream.

When your program uses `printf()` (see below) to print to the screen, you are using **standard output**.

Files (e.g. data files) are also associated with streams, and we will see below how to read from them and write to them.

We won't talk about **standard error** here. Refer to a UNIX reference, a C book, or Streams and Files for more details.

Character I/O: `getchar()` and `putchar()`

When you want to read data a single character at a time, you can use `getchar()`. The output function `putchar()` will write out one character at a time to standard output.

```
#include <stdio.h>

int main() {
  char c = getchar();
  putchar(c);
  putchar('\n');
  return 0;
}
```

Formatted I/O: `printf()` and `scanf()`

The `printf()` function

We have seen in many of the example code snippets, the use of the `printf()` function to write formatted output to the screen. In general the `printf()` function takes two arguments: first, a string argument that indicates what to write to the screen, including formats, and second, a list of variables that provide the data to the various elements in the formatting string.

Here is an example where we print a sentence that contains a formatted integer, a floating-point number, and a string:

```
#include <stdio.h>

int main() {
  int i = 42;
  char str[] = "the meaning of life";
  double p = 3.14159265;
  printf("pi is about %12.8f and %s is %d\n", p, str, i);
}
```

Let's unpack the `printf()` statement above on line 7. The format string includes three numeric format codes. The first, `%12.8f`, says that we want to print a floating-point number (`f`), we want to print it to 8 decimal places (`.8`), and we want to provide 12 columns of space for it (`12`). The second format argument is `%s` which corresponds to a string. The third format argument is `%d` which corresponds to an integer.

Consult a reference manual (or a website like this for full details).

The `scanf()` function

To read formatted data in from standard input we can use the `fscanf()` function. Just like `printf()`, it takes as a first argument a format string, followed by other arguments specifying the destination of each argument. Unlike `printf()`, the `scanf()` function requires that these destination arguments be **addresses** of the relevant locations in memory (we can feed it a pointer, for example).

Here is a simple example in which we read from standard input a date, which we expect to be in the following format:

```
25 Dec 2012
```

```c
#include <stdio.h>

int main() {
  int day, year;
  char monthname[20];
  scanf("%d %s %d", &day, monthname, &year);
  printf("the date is %s %d, %d\n", monthname, day, year);
  return 0;
}
```

```
25 Dec 2012
the date is Dec 25, 2012
```

Note how on line 6, we pass the **address** of `day` and `year` using the ampersand (`&`) operator. On line 4 we declare `day` and `year` as `int`. Using the ampersand notation, we can write `&day` and `&year`, which correspond to **pointers** to the **address** of `day` and `year`.

The `scanf()` function ignores blanks and tabs in the format string, and it skips over white space (blanks, tabs, newlines, etc) as it looks for input values.

Apart from sending and receiving data from the terminal. We can also read and write from text files.