10. Input and Output

Table of Contents

Command-line arguments

When you run a C program from the command line, you can pass it input arguments.

I/O Streams

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;
}
```

```
x
x
```

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);
}
```

```
pi is about   3.14159265 and the meaning of life is 42
```

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
```

```
#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.

Input and Output with Files

Opening and Closing files with `fopen()` and `fclose()`

Before a file can be read or written to, it has to be **opened** using the `fopen()` function, which takes as arguments a string corresponding to the filename, and a second argument (also a string) corresponding to the **mode**. The mode is read ("r"), write ("w") or append ("a"). The `fopen()` function then returns a pointer to the (open) file. After reading and/or writing to your file, you will need to **close** it using the `fclose()` function.

Reading and Writing to files

There are many functions in `stdio.h` for reading from and writing to files. There is a collection of functions for reading and writing ascii (text) data, and there are functions for dealing with binary data.

Ascii Files (plain text)

There are functions to read single characters at a time (`getc()`and `putc()`), there are functions to read and write formatted output (`fscanf()` and `fprintf()`), and there are functions to read and write single lines at a time (`fgets()` and `fputs()`).

Here is an example program that outputs a table of temperature values in Fahrenheit and Celsius to an ascii file.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  double tmpC[11] = {-10.0, -8.0, -6.0,
                      -4.0, -2.0,  0.0,  2.0,
                       4.0,  6.0,  8.0, 10.0};
  double tmpF;
  int i;

  fp = fopen("outfile.txt", "w");
  if (fp == NULL) {
    printf("sorry can't open outfile.txt\n");
    return 1;
  }
  else {
    // print a table header
    fprintf(fp, "%10s %10s\n", "Celsius", "Fahrenheit");
    for (i=0; i<11; i++) {
      tmpF = ((tmpC[i] * (9.0/5.0)) + 32.0);
      fprintf(fp, "%10.2f %10.2f\n", tmpC[i], tmpF);
    }
    fclose(fp);
  }

  return 0;
}
```

```
plg@wildebeest:~/Desktop$ more outfile.txt
   Celsius Fahrenheit
    -10.00      14.00
     -8.00      17.60
     -6.00      21.20
     -4.00      24.80
     -2.00      28.40
      0.00      32.00
      2.00      35.60
      4.00      39.20
      6.00      42.80
      8.00      46.40
     10.00      50.00
```

A couple of things are worth noting about the code above. On line 13, we check the value of the file pointer `fp`, and if it is equal to `NULL` (which means there was an error opening the file), we write a message to the screen and we `return 1` (which exits the `main()` function and thus exits our program). A convention in UNIX is that programs which execute successfully return `0`and non-zero values are returned when there was an error encountered.

On lines 19 and 22 we use the `fprintf()` function to write to the file. This is just like the `printf()` function that we have seen before, to write formatted output to standard output. This time we're writing to a file instead.

To illustrate reading from ascii files, here's an example program that will read in the file produced by the previous code example, and do some arithmetic on them.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  char buffer[256];
  double tempC, tempF;
  double sumC = 0.0;
  double sumF = 0.0;
  int numread = 0;

  fp = fopen("outfile.txt", "r");
  if (fp == NULL) {
    printf("there was an error opening outfile.txt\n");
    return 1;
  }
  else {
    // read in the header line first
    fgets(buffer, 256, fp);
    while (!feof(fp)) {
      fscanf(fp, "%lf %lf\n", &tempC, &tempF);
      printf("tempC=%.2f, tempF=%.2f\n", tempC, tempF);
      sumC += tempC;
      sumF += tempF;
      numread++;
    }
    fclose(fp);
    printf("%d values read, sumC=%.2f and sumF=%.2f\n", numread, sumC, sumF);
  }

  return 0;
}
```

```
tempC=-10.00, tempF=14.00
tempC=-8.00, tempF=17.60
tempC=-6.00, tempF=21.20
tempC=-4.00, tempF=24.80
tempC=-2.00, tempF=28.40
tempC=0.00, tempF=32.00
tempC=2.00, tempF=35.60
tempC=4.00, tempF=39.20
tempC=6.00, tempF=42.80
tempC=8.00, tempF=46.40
tempC=10.00, tempF=50.00
11 values read, sumC=0.00 and sumF=352.00
```

Some comments about the above code example: on line 19 we use the `fgets()` function to read in the first line of the file to a character string (`buffer`) that we declared above. The `fgets()`function requires as its second argument the maximum number of characters to read. Since we know we don't expect many here, we indicate a maximum of 256. After reading in the first line, we now enter a **while loop**, using `fscanf()` to read in each pair of floating-point values. The while loop terminates when `!feof(fp)` is false. The `feof()` function returns TRUE if we are at the end of the file, and FALSE otherwise.

Binary Files (raw bytes)

There are many circumstances in which you may want to read from and write to binary files. Binary files are not plain text (ascii) files where each chunk of bytes represents an ascii character. In binary files, you store raw bytes, in whatever format you want. For example Optotrak stores its data files as binary files: a header of a given length (number of bytes) followed by data, in a specific byte format.

Advantages of binary files over ascii files is that they are typically smaller in size, and they can be read from and written to faster (no need to convert between raw bytes and ascii characters). Disadvantages of binary files are that they are not human readable (you can't open in them in a text editor and "look" at them).

The `fread()` and `fwrite()` functions are used to read and write binary data (raw bytes) from and to binary files. Here is an example of writing some data to a binary file. We first write a 16 byte header containing the date (4 + 4 + 4 = 12 bytes) and the number of data points (4 bytes). We then write out the data array, 4 bytes per element. In this example the data are integer values.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  int year = 2012;
  int month = 8;
  int day = 26;
  int mydata[5] = {2, 4, 6, 8, 10};

  fp = fopen("data.bin", "w");
  if (fp == NULL) {
    printf("error opening data.bin\n");
    return 1;
  }
  else {
    // write out the header
    int bytesout;
    bytesout = fwrite(&year, sizeof(year), 1, fp);
    bytesout = fwrite(&month, sizeof(month), 1, fp);
    bytesout = fwrite(&day, sizeof(day), 1, fp);
    // write the data
    bytesout = fwrite(mydata, sizeof(int), 5, fp);
    fclose(fp);
  }

  return 0;
}
```

Here is an example program to read from the binary data file:

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *fp;
  int bytesread;
  int yy, mm, dd;
  int thedata[5];

  fp = fopen("data.bin", "r");
  if (fp == NULL) {
    printf("error opening data.bin\n");
    return 1;
  }
  else {
    // read the header
    bytesread = fread(&yy, sizeof(int), 1, fp);
    bytesread = fread(&mm, sizeof(int), 1, fp);
    bytesread = fread(&dd, sizeof(int), 1, fp);
    printf("year=%d, month=%d, day=%d\n", yy, mm, dd);
    // read the data
    bytesread = fread(thedata, sizeof(int), 5, fp);
    printf("data = [%d,%d,%d,%d,%d]\n",
           thedata[0], thedata[1],thedata[2],thedata[3],thedata[4]);
    fclose(fp);
  }

  return 0;
}
```

```
year=2012, month=8, day=26
data = [2,4,6,8,10]
```

The bottom line is, as long as you know what the binary **format**is (that is, how many bytes represent each value) then you can read and write them in "raw" binary using `fread()` and `fwrite()`.

Links

- The C Library Reference Guide
- C file input/output

Exercises

- 1 Write a program that asks the user to enter three strings. After they have entered all three strings, print the strings out using all uppercase letters.
- 2 Alter the program so that it prints out the all-caps strings in reverse.
- 3 Alter the program again so that it writes the all-caps reversed strings to a plaintext file.
- 4 Write a program that reads three strings from a plaintext file, reverses each string, and prints them out to the screen.

Solutions

- x
- x
- x
- x