C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 8

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

r.kaiser@physics.gla.ac.uk

Summary

- File I/O
- Graphics with C

`http://www.physics.gla.ac.uk/~kaiser/`

Streams

- Input and Output, aka I/O, in C is based on the concept of the stream.
- A stream is a sequence of characters, more precisely, a sequence of bytes of data.
- The advantage of streams is that I/O programming becomes device independent. Programmers don't have to write special I/O functions for each device (keyboard, disk, etc.). The program 'sees' I/O as a continuous stream of bytes, no matter where it's coming from or going to.
- Every C stream is connected to a file, where file does not (only) refer to a disk file. Rather, it's an intermediate step between the stream that your program deals with and the actual physical device used for I/O.
- We will in the following typically not distinguish between streams and files and just use 'file'. And we will deal only with disk files, but keep in mind that a 'file' could also be e.g. a printer.

Standard I/O

- There are three predefined streams, also referred to as the standard I/O files:

Name	Stream	Device
stdin	standard input	keyboard
stdout	standard output	screen
stderror	standard error	screen

- The C library contains a large number of routines for manipulating files. The declarations for the structures and functions used by the file functions are e stored in the standard include file `<stdio.h>`.
- So, before doing anything with files, you must include the line

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

  at the beginning of the program.

FILEs

- Files are handled using the data type `FILE`, which is defined in `<stdio.h>` together with the rest of the I/O facilities.
- The declaration for a file, actually for a file variable is:

```
FILE *file-variable; /* comment */
```

- Example:

```
FILE *in_file;   /* file containing the input data */
```

- To open a file you must create a pointer to type `FILE` and use the `fopen` function. The prototype of this function is located in `stdio.h` and reads

```
FILE *fopen(const char *filename, const char *mode)
```

(ok, have a look, it's a bit more complicated...)

- For example

```
FILE *in_file;
in_file = fopen("data.txt","r");
```

opens the file 'data.txt' in the present working directory for read access.

Opening and Closing Files

- When opening a file with `fopen` the `name` that you use depends on the operating system under which you are working.
- Under Linux, the rules for Linux filenames apply. This means e.g. that you can open the file 'data.txt' in the present working directory or the file '/usr/local/stdio.h' using the absolute path of that file.
- The mode specifies if you want to open the file for reading, writing etc:

Mode	Meaning
`r`	reading
`w`	writing
`a`	appending
`r+`	reading and writing, overwrites from the start
`w+`	reading and writing, if the file exists, it's overwritten
`a+`	reading and appending

Opening and Closing Files cont.

- If you fopen a file for write access and it doesn't yet exist, a file with the specified name will be created. If a file with the same name already exists, it will be overwritten. (That is, if you have the permission to do so.)
- The function `fopen` returns a file handle that will be used in subsequent I/O operations.
- If there is an I/O error, e.g. if you try to open a file for reading that doesn't exist, the value NULL is returned. This can be used in an `if`-statement to check if the file was opened correctly:

```
in_file = fopen("input.txt", "r");
if (in_file == "NULL")
   fprintf(stderr, "Error: unable to open 'input.txt'\n");
   exit(8);
```

- The function `fclose` will close the file, e.g.

```
fclose(in_file);
```

File Access - Example

Count the number of characters in `input.txt` (`copy.c`).

```
#include <stdio.h>
const char FILE_NAME[] = "input.txt";
#include <stdlib.h>

int main(){
    int  count = 0;
    FILE *in_file;
    int ch;
    in_file = fopen(FILE_NAME, "r");
    if (in_file == NULL) {
        printf("Cannot open %s\n", FILE_NAME);
        exit(8);
    }
    while (1) {
        ch = fgetc(in_file);
        if (ch == EOF)
            break;
        ++count;
    }
    printf("Number of characters in
       %s is %d\n", FILE_NAME, count);
    fclose(in_file);
    return (0);
}
```

- open file for read access
- exits if file does not exist or cannot be read
- `fgetc` gets a single character from the file
- more about `fgetc` later

ASCII and Binary Files

- There are two types of files: ASCII and Binary files.
- ASCII, the American Standard Code for Information Interchange, encodes characters as (7-bit) integer numbers. Terminals, keyboards and printers deal with character data.
- Computers, i.e. the CPU and the memory, work on binary data. When reading numbers from an ASCII file, the character data must be processed through a conversion routine like `sscanf`. This takes (some) time. Binary files require no conversion.
- To access a binary file you have to add a 'b' to the mode of `fopen`.

ASCII	Binary
reading requires conversion	no conversion required
human readable	not human readable
portable	(mostly) not portable
small/medium amounts of data	large amounts of data

Different Types of I/O

- There are three different ways to write data to a disk file: formatted output, character output and direct (binary) output. There are of course the corresponding ways of reading them in.

Type of I/O	I/O statements	should be used for
Formatted	`fprintf`	files with text and numerical data
	`fscanf`	to be read in by spreadsheets,
		databases or data analysis
		programs
Character	`fputc`	text files, to be read e.g. by
	`fgetc`	word processors
	`fputs`	
	`fgets`	
Direct	`fwrite`	binary files, best way to save
	`fread`	for later use by a C program

Character I/O

- `int fgetc(FILE *file_ptr)`
  returns a single character from the file, but has return type integer.
- Successive calls get successive characters.
- If no more data exist, `fgetc` returns the constant `EOF`
- EOF defined in `<stdio.h>` as `-1` (which is why the return type has to be `int`).
- `int fputc(int c, FILE *file_ptr)`
  writes a single character to the file.
- `char fgets(char *s, int n, FILE *file_ptr)`
  reads in a string of n characters. We have already met `fgets` before: `file_ptr` can also be `stdin` to read a string from the keyboard.
- `char fputs(const char *s, FILE *file_ptr)`
  writes out a string.

`fprintf` - Formatted File Output

- The function `fprintf` converts data to characters and writes them in a defined format to a file. The general form of `fprintf` is:

```
count = fprintf(file, format, parameter-1,
                parameter-2,...);
```

  where

  - `count` is the return value of `fprintf`: the number of characters sent or -1 if an error occurred,
  - `format` is a format statement of the same type as used with `printf`.

- `fprintf` to `stdout` is identical to `printf`.
- `printf` and `fprintf` have another sister function `sprintf` that does the same for formatted writing to a string.
- Example:

```
fprintf(file_ptr, "The year was %d", year);
```

Formatted File Output - Example

Write the table from `band.c` to file (`band_plot.c`).

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

int main()
{
  int i, j;
  FILE  *f_ptr;    /* file handle */

  f_ptr = fopen("binary_and.txt", "w");

  fprintf(f_ptr, "  &");
  for (j=0; j<16; j++)
        {
          fprintf(f_ptr, "%3d", j);
        }
  fprintf(f_ptr, "\n\n");
  for (i=0; i<16; i++)
    {
      fprintf(f_ptr, "%3d", i);
      for (j=0; j<16; j++)
        {
          fprintf(f_ptr, "%3d", i&j);
        }
      fprintf(f_ptr, "\n");
    }

  fclose(f_ptr);  /* close file */
  return(0);
}
```

The only difference between `band.c` and `band_file.c` is the introduction of the file handle and the `fprintf` instead of `printf`.

Formatted File Input - `fscanf`

- `fscanf` is similar to `scanf`, except that the file (represented by the file handle) has to be specified, while `scanf` assumes `stdin` as input stream.
- The syntax for `fscanf` is

```
fscanf(file, format, &parameter-1, ...);
```

  where the return value of the function `fscanf` is the number of parameters that were read in successfully.
- For example

```
fscanf(f_ptr, "%f %f", &para1, &para2);
```

- Instead of using `fscanf`, we can also use the combination of `fgets` and `sscanf` already introduced in C-lecture 3.

Binary I/0

- Binary I/O is accomplished through the routines `fread` and `fwrite`. The syntax for both commands is similar:

```
  read_size = fread(data_ptr, 1, size, file);
 write_size = fwrite(data_ptr, 1, size, file);
```

  - where `read_size/write_size` is the size of the data that was read/written,
  - `data_ptr` is the pointer to the data to be read/written. This pointer must be cast to a character pointer `(char *)`.
  - size is the size of the data to be read/written in byte,
  - file is the input file (the file pointer).

- `fread` and `fwrite` are originally meant for arrays of objects, so the second parameter (that we set to 1) is the size of an object in bytes and the third parameter is the number of objects in the array.

Binary I/O - Example

Plot a sinus function to screen and postscript file (`sin_plot.c`).

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

#define SIZE 10

int main()
{
  int count, array1[SIZE], array2[SIZE];
  FILE *fp;

  /* Initialise array1 */
  for (count = 0; count < SIZE; count++)
    array1[count] = 2*count;

  /* Save array1 to the binary file */
  fp = fopen("direct.txt", "wb");
  fwrite(array1, sizeof(int), SIZE, fp);
  fclose(fp);

  /* Read data from binary file into array2 */
  fp = fopen("direct.txt", "rb");
  fread(array2, sizeof(int), SIZE, fp);
  fclose(fp);

  /* Display both arrays */
  /* to show they are the same */
  for (count = 0; count < SIZE;
                        count++)
    printf("%d\t%d\n",
     array1[count], array2[count]);

  return(0);
}
```

Output:

```
 0       0
 2       2
 4       4
 6       6
 8       8
10      10
12      12
14      14
16      16
18      18
20      20
```

Graphics with C

- Graphics capabilities are not part of Standard C. Sorry, but that's just the way it is.
- Some versions of C, e.g. Turbo C, have libraries like `graphics.h` that come as part of the distribution, but they are non-standard.
- However, there are libraries available for almost anything, and a lot of quite fancy graphics programs are actually written in C. Just that drawing a line on the screen is not trivial and requires almost the same work in setting up the right infrastructure as more complicated applications.
- Let's have a look at some of the different applications that you actually may want to use graphics for:
  - Windows programming and Graphical User Interfaces,
  - drawing/plotting data,
  - producing picture files as output, e.g. .gif or .pdf files.
- Unfortunately, graphics is mostly beyond the scope of this course.

Windows and GUIs with C

- X Window System
  `http://www.xfree86.org/` and `http://www.x.org/`
  Linux uses the X Window system. You also can find instructions in X/Motif programming under C on the web, but it's a bit outdated.
- Qt
  `http://www.trolltech.com/`
  Actually a C++ development tool for X Window programming, available also for Linux. KDevelop uses Qt.
- GGI - General Graphics Interface
  `http://www.ggi-project.org/`
  A project that aims at a graphics system that works everywhere. The GGI project provides various libraries, the most important ones being LibGGI and LibGII.

Drawing/Plotting Data with `gnuplot`

- `gnuplot` is one of many data analysis and function plot programs.
- It's advantage is that it is available free under Linux and that there is a library (gnuplot_i) that provides an interface for C to gnuplot. http://ndevilla.free.fr/gnuplot/gnuplot_i/
- `gnuplot` website: `http://www.gnuplot.info/` Tutorials can be found at `http://www.cs.uni.edu/Help/gnuplot/` and `http://www.duke.edu/~hpgavin/gnuplot.html`
- One nice feature of `gnuplot_i` is that `gnuplot` commands can be 'piped' from a C-program to `gnuplot`. There are very few additional commands to learn once one is familiar with `gnuplot`.

Basic Functionality of `gnuplot`

- Start `gnuplot` by simply typing `'gnuplot'`. You will get a few lines of text and a command-line interface with the prompt `'gnuplot>'`.
- To quit type `'exit'`.
- To plot a function, type `'plot sin(x)'`. A graphics window will open and the function will be displayed.
- `plot[x1:x2][y1:y2]<function>`
  will plot the specified function within the given x- and y-limits.
- `splot[x1:x2][y1:y2][z1:z2]<function>`
  will plot a function of 2 parameters within the given x-, y- and z-limits.
- `set isosamples x-rate, y-rate`
  sets the number of points in x and y, (`set samples x-rate` for a 1-d function.
- `replot` does just that after a change of parameters

Basic Functionality of `gnuplot` cont.

- `save "work.gnu"`
  saves the present settings and commands to file
- `load 'work.gnu'`
  loads a 'macro' that saves you from typing lots of lines every time. Comments in macros and data files are marked by `#`.
- `plot "<filename>" using x:y`
  Assuming `<filename>` is an ASCII file with data organised in columns separated only by spaces, this command will plot the specified columns (e.g. 1:3) versus one another.
- `help` accesses the built-in online help facility of gnuplot.
- `set output file.ps`
  `set terminal postscript`
  `replot`
  will allow you to plot to a postscript file instead of the screen.
- `set terminal x11`
  to get back to the screen.

Using `gnuplot_i`

- `#include "gnuplot_i.h"` to include the `gnuplot_i` header file
- `gcc -o program program.c gnuplot_i.o`
  to compile, or better, use a `Makefile` and the flag `-I` to add the right directory for `gnuplot_i.h` to the path.
- `gnuplot_ctrl *h1;`
  `h1 = gnuplot_init() ;`
  opens a new gnuplot session, referenced by a handle of type (pointer to) `gnuplot_ctrl` and initialised by calling `gnuplot_init()`.
- `gnuplot_cmd(handle, "gnuplot_command")`
  'pipes' a `gnuplot` command from the C-program to `gnuplot`.
- For some commands there are C versions, see `gnuplot_i.h`.
- `gnuplot_close(h) ;` to close the session.

gnuplot_i - Example

Plot a sinus function to screen and postscript file (sin_plot.c).

```
#include <stdio.h>
#include <stdlib.h>
#include "gnuplot_i.h"

#define SLEEP_LGTH  5
#define NPOINTS     50

int main(int argc, char *argv[])
{
    gnuplot_ctrl   *h1;
    double         x[NPOINTS] ;
    int            i ;

    /* Initialize gnuplot handle */
    h1 = gnuplot_init() ;

    /* Plot sinus to screen */
    printf("sine in points\n") ;
    gnuplot_setstyle(h1, "points") ;
    gnuplot_cmd(h1, "set samples 50");
    gnuplot_cmd(h1, "plot [0:6.2] sin(x)");
    sleep(SLEEP_LGTH) ;

    /* Plot sinus to postscript file */
    gnuplot_cmd(h1, "set terminal postscript");
    gnuplot_cmd(h1, "set output \"sinus.ps\"");
    gnuplot_cmd(h1, "replot");
    gnuplot_cmd(h1, "set terminal x11");

    /* close gnuplot handle */
    printf("\n\n") ;
    printf("*** end of gnuplot example\n") ;
    gnuplot_close(h1) ;
    return(0) ;
}
```