

A Rudimentary Intro to C programming

Wayne Goddard

School of Computing, Clemson University, 2008

Part 5: Odds and Ends

23	Files and I/O	E1
24	Command Line Arguments and Header Files	E3
25	Structs	E6
26	BitOps	E9

Files and I/O

23.1 Streams and Redirection

A *stream* is a source of input or a destination of output. One automatically has the standard streams `stdin` and `stdout`; we have accessed these with `scanf` and `printf` etc. One can redirect these. For example:

```
a.out < in.txt > out.txt
```

will cause the program to read from `in.txt` (instead of the keyboard) and write to `out.txt` (instead of console).

23.2 Files

In C, a `FILE` provides access to a stream. It is declared with

```
FILE *myfile;
```

You just need to know the basics. The four common operations are:

- `fopen(fileName, accessString)` — *open a stream*:
This creates access to the file on disk with the specified name and returns a *handle* (so one writes `myfile = fopen...`). The access string specifies the *mode*: "r" for reading and "w" for writing. (Can also specify other modes.) Note that the handle is returned as `NULL` (0) if there was a failure; one should check this.
- `fgets(string, maxSize, myfile)` — *read from a stream*:
This function reads a line from the stream into the `string` and appends the `'\0'`-terminator. The number of characters read had better be at most `maxSize - 1`. The function returns `NULL` if end-of-file prevented reading.
- `fprintf(myfile, string, vals)` — *write to a stream*:
This is like `printf`, but to the stream.
- `fclose(myfile)` — *close a stream*:
This is called at the end. It helps the operating system by releasing resource.

The formal signatures are:

```
FILE *fopen( const char *name, const char *mode );
int fclose( FILE *f );
char *fgets( char *buffer, int buffer_size, FILE *stream);
int fprintf( FILE *stream, const char *format, ... );
```

There are also: `feof`, `fputs`, `fputc`, `fgetc`. Beware: do not use `open()` or `close()`; these are low-level commands.

23.3 Sample Code

For copying from one file to another.

```
// filer.c copies from one file to another
// not robust
#include <stdio.h>

int main(void) {
    FILE *in, *out;
    in = fopen("from.txt", "r");
    out = fopen("to.txt", "w");
    if( !in || !out ) {
        printf("Error in opening files\n");
        return 1;
    }
    char s[100];
    while( fgets(s, 100, in)!=NULL )
        fprintf(out,s);
    fclose( in );
    fclose( out );
    return 0;
}
```

Practice. Write a program that will take the input, and write it to a file adding line-numbers.

Command Line Arguments and Header Files

24.1 Command Line Arguments

A C program can access information you type on the *command line*. This is useful, for example, if you want to specify the name of an input file at run time.

For example we could do the following:

```
a.out myDict.txt
```

Using a different signature, the main function has access to the command line as an array of strings. The array is called `argv` and the count is given by `argc`. In the above case, it would get an array with two strings: “a.out” and “myDict.txt”.

Here is code, taken from howstuffworks.com, which echoes the command line to the screen. Note that `char *argv[]` is an array of strings.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int x;
    printf("argc is %d\n", argc);
    for (x=0; x<argc; x++)
        printf("arg[%d] is %s\n", x, argv[x]);
    return 0;
}
```

Practice. Write a program that will add up all the values in a file whose name is supplied by the user on the command line.

24.2 Header Files

You have included several files ending in “.h”. These are *header* files. They do not have the source code, only the prototypes. You can do the same. We illustrate this with an example.

Suppose one wanted to write a statistics library, and a program that uses it. The library would contain functions for computing and displaying statistics, while the program itself might have functions for reading an array, printing an array, and the `main` function.

The steps are:

- The header file is `statistics.h`. It contains the prototypes. Since header files should be included only once per file, they should also contain guards (commands to the compiler)—see the example code.
- The library file is `statistics.c`. It just has the implementations.
- The main program must `#include` the header file. Your own header files are included with `"` rather than with `< >`. You should include the standard files before you include your header files.
- `gcc` must be told to compile the library file and join it to the main program. This is done by adding the name before the main program:

```
gcc statistics.c testStats.c -lm
```

24.3 Example Code

Here are the three files.

```
/* statistics.h */
#ifndef STATISTICS_H
#define STATISTICS_H 1
#endif
```

```
float mean(float[], int);
float sdev(float[], int);
```

```
/* statistics.c */
#include <math.h>
#include "statistics.h"

float mean(float data[], int size) {
    double sum=0.0;
    int i;
    for(i=0; i<size; i++)
        sum += data[i];
    return sum / size;
}
```

```
float sdev(float data[], int size) {
    double sumDevs=0.0;
    float mu = mean( data, size);
```

```
int i;
for(i=0; i<size; i++)
    sumDevs += (data[i] - mu)*(data[i]-mu);
return sqrt( sumDevs/ (size-1) );
}
```

```
/* testStats.c */
#include <stdio.h>
#include "statistics.h"

int main(void) {
    float array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int arraySize = 10;
    printf("Mean is %.2f\n", mean(array, arraySize));
    printf("Standard deviation is %.2f\n", sdev(array, arraySize));
    return 0;
}
```

Structs

Structs are used for grouping. Sometimes called a *record*, they are the precursor to objects, the cornerstone of C++.

25.1 Defining a Struct

A **struct** is a collection of *members*. There are a couple of ways to set up a struct. One way is to use **typedef**, which is used in general to create a user-defined type. The following example creates a `Date` type which can be used in a program.

```
typedef struct {
    int day;
    int month;
    int year;
} Date;
```

25.2 Accessing a Struct

One has in general direct member access. This uses DOT notation:

```
Date D;
D.day = 1;
D.month = 1;
D.year = 2008;
```

You can have an array of structs. (Indeed you can have an array of anything.) So a program might care about a sequence of events, stored as `Date dates[100]`

25.3 Example Code: pointing.c

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

typedef struct {
    int xpos;
    int ypos;
} Point;

const int MAX_PTS=100;
```

```

const int DIM=20;
const char BLANK=' '; const char POINT='X';

int loadPoints(char * fileName,Point list[]); // returns number loaded
void plotPoints(Point list[], int count);

int main(int argc, char * argv[]) {
    Point listing[MAX_PTS];
    int count = loadPoints(argv[1],listing);
    plotPoints( listing, count);
    return 0;
}

// loads points from file
// assumes number of points given at start of file
int loadPoints(char* fileName, Point list[]) {
    FILE* inFile;
    inFile = fopen(fileName, "r");
    if( inFile==NULL ) {
        printf("Couldn't find file %s\n", fileName);
        return 0;
    }

    int p, limit;
    fscanf(inFile, "%d", &limit);
    if(limit>MAX_PTS) {
        printf("Only taking first %d points\n", MAX_PTS);
        limit = MAX_PTS;
    }
    for(p=0; p<limit; p++)
        fscanf(inFile, "%d%d", &list[p].xpos, &list[p].ypos);

    fclose(inFile);
    return limit;
}

```

```

// plots points on a grid of DIM x DIM
// assumes coordinates in range 0..DIM-1
void plotPoints(Point list[], int count) {
    char screen[DIM][DIM];
    int i,j,p;
    for(i=0; i<DIM; i++)
        for(j=0; j<DIM; j++)
            screen[i][j] = BLANK;
    for(p=0; p<count; p++)
        if(list[p].xpos>=0 && list[p].xpos<DIM && list[p].ypos>=00 && list[p].ypos<DIM )
            screen[ list[p].xpos ][ list[p].ypos ] = POINT;

    for(i=0; i<DIM; i++) {
        for(j=0; j<DIM; j++)
            printf("%c", screen[i][j] );
        printf("\n");
    }
}

```

BitOps

26.1 Bitwise Operators

A fundamental C approach is that one uses ints to store things other than numbers. An integer has a representation in **binary** as a series of **bits**: 1s and 0s. Sometimes a program uses an integer where each bit is a piece of information about the system, such as an error code: the first bit might say what type.

To access these bits you have bit operators. The most common are the *shift* operators. Shift right is equivalent to divide by 2, and shift left is equivalent to multiply by 2 (at least for small positive numbers). For example, the following code rounds a number down to the nearest even number:

```
x = x >> 1;  
x = x << 1;
```