ICT1002 – LAB – WEEK 12

Files

1 OBJECTIVES

To understand the structure of files, and to open, read from, write to, and close files in C.

2 PRE-READING

For some of the exercises in this lab, it may be useful to know a little bit more about how C programs interact with the operating system.

You may know that, when starting a program from the command line, you can specify *command line arguments* that tell the program more about what you want it to do. When compiling a C program, for example, you might type '`cl hello.c`' or '`gcc hello.c`', which tell the `cl` and `gcc` programs, respectively, to read from a file called `hello.c`.

In C, the command line arguments are passed to the program as arguments of the `main()` function. The complete function protoype for `main()` is:

```
int main(int argc, char **argv)
```

The `argc` argument contains the number of arguments on the command line, including the name of the program itself. The `argv` argument is an array of strings containing the arguments in order. So, `argv[0]` is the name of the program, `argv[1]` is the first argument, and so on. Note that these are always strings, so you need to use `atoi()` and friends if you want to convert them to numbers.

Most compilers allow you to omit the arguments for `main()`, which is why we've been able to use just `int main()` up until now. You only need them if your program uses the arguments.

The return value of `main()` is an integer. This integer is returned to the program that invoked your program, and is usually used to indicate whether or not your program succeeded. By convention, a return value of zero indicates that a program completed without any errors, and a non-zero value indicates that something went wrong. Some programs return specific non-zero values to indicate different kinds of errors. Shell scripts (in Unix) and batch files (in Windows) can use the return values from program to make decisions about what they should do.

Here is a link with examples: https://stackoverflow.com/questions/16869467/command-line-arguments-reading-a-file/16869591

3 EXERCISES FOR WEEK 12 LAB

WEEK_12_LAB_EXE_1: BINARY FILES

The `tar` program (for "**t**ape **ar**chive") creates an uncompressed file archive by joining a collection of files end to end. At the start of each file in the archive is a fixed-length header that stores information about the file, represented as a C structure as follows:

```
/*
 * Standard Archive Format - Standard TAR - USTAR
 * from https://www.fileformat.info/format/tar/corion.htm
 */

#define  RECORDSIZE  512
#define  NAMSIZ      100
#define  TUNMLEN      32
#define  TGNMLEN      32

struct header {
    char    name[NAMSIZ];
    char    mode[8];
    char    uid[8];
    char    gid[8];
    char    size[12];
    char    mtime[12];
    char    chksum[8];
    char    linkflag;
    char    linkname[NAMSIZ];
    char    magic[8];
    char    uname[TUNMLEN];
    char    gname[TGNMLEN];
    char    devmajor[8];
    char    devminor[8];
};
```

For the purposes of this exercise, we only need to worry about the *`name`* and *`size`* fields. The `name` field contains the name of the file, while the `size` field contains the length of the file in bytes. Note that the `size` field is a string, not an integer, so it needs to be converted using `atoi()` in order to perform arithmetic on it. We can ignore all of the other other fields in our program.

A `tar` archive thus has the form:

`struct header`	*file1 data*	`struct header`	*file2 data*	`struct header`	*file3 data*	…

The header sections always have the same length, `sizeof(struct header)`, but the length of each file is given by the `size` field in the header.

Write a program called `minitar` that accepts the two given file names ("File1.txt" and "File2.txt") on the command line (`minitar File1.txt File2.txt`) and generate an archive

file ("Result.tar") containing all of these files, with the `name` and `size` fields filled in as described above.

Hints:

- See http://www.cplusplus.com/reference/cstdio/fread/ for an example of how to find the length of a file and read it into memory. Note that reading the whole file into memory at once may require quite a lot of memory; you might like to try finding a more efficient method of copying data from the input file to the output file.

Submit your tested source code to repl.it by 11:30PM on 30 Nov 2021 (Tue).