Supplement: Basic Calculator

Leonard König, April 12, 2021

ISO C File Streams

In the previous exercise we've dealt with file descriptors (which are just `int`s) and the `read` and `write` functions which provide only a thin layer of abstraction over the UNIX/Linux OS below. While we will come back to these in later exercises when we really need to deal with OS specific functionality, for regular reading and writing from and to 'real' files (including `stdin` and `stdout`) the C standard library provides another set of functions. These also work on non-UNIX systems and are called 'file streams'.

Let's start to rewrite the `main` function from last time using this higher level interface. As we can see in **Listing 1**, the first thing we change is to replace the `0` and `1` for the standard input/output file descriptors with the constants `stdin` and `stdout`. In order to be able to use these, we need to include the C standard I/O header `stdio.h`. Further, we replace `open` by `fopen` with the option `O_RDONLY` being replaced by the option string `"r"`. This function doesn't return an integer anymore, but a *pointer*, specifically a pointer to some memory holding a value of type `FILE`. This `FILE*` represents our opened file, if we could successfully open it; however, if the requested file doesn't exist or we don't have the permissions to open it, `fopen` will return a special pointer value, called `NULL`. Such a `NULL` pointer doesn't point to anything, in fact, trying to access the memory 'behind' it will likely crash your program. It is thus crucial to check, whether `fopen` succeeded, e.g.:

```
if (src == NULL) {
        perror(argv[i]);
        // ...
}
```

In the given code, we contracted this `NULL` check to `!src` which is equivalent and can be read as 'if `src` does *not* exist'.

Finally, we replace `close` by `fclose`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc == 1) {
                // copy would need to be adapted
                // as well ...
                int err = copy(stdin, stdout);
                return (err);
        }

        int status = 0;
        for (int i = 1; i < argc; i++) {
                FILE *src = fopen(argv[i], "r");
                if (!src) {
                          perror(argv[i]);
                          status = 1;
                          continue;
                }

                int err = copy(src, stdout);
                if (err) {
                          status = 1;
                          fclose(src);
                          break;
                }

                fclose(src);
        }
        return (status);
}
```

Listing 1 `cat(3)` `main` with file streams

We would now need modify the `copy` function accordingly, however this would diverge too much from this document. I want to end this section with a small note though: Although the streams interface is (arguably) more simple, it supports not all use-cases the file descriptor based I/O does. That's why the `cat` program should indeed be

written as we have done. For our next use case, streams are completely fine though.

Parsing Strings

In this exercise you should implement a *simple* calculator. Again, we write a specific function for this which will not be called `copy` though (this would be rather unintuitive) and only take on argument, e.g.:

```
void process_file(FILE *input);
```

In order to not completely give off the solution, we will here solve a slightly modified exercise to the original. We start with an accumulator variable that's initialized to the value $a = 0$. The user can now input an operator such as `+` or `-`, `*` or `/` followed by an integer x. Afterwards we calculate a op x and store this result in a. The user may then enter another operand and integer.

Assume that the input thus comes in the format: operator, integer, operator integer, and so on. Between these there may be arbitrary 'white space', i.e., new lines, tabulator, space, etc. We terminate if no (valid) input was given.

However, the actual input is simply a string of bytes, e.g., `"+ 42"`. We need to *parse* it, turning the the decimal representation of the integer into an actual number. Fortunately, the C standard library provides method to do just that. All we need to do is to provide a so-called 'format string' which contains place holders. When the input is read, it will be matched against those place holders and a conversion is done:

```
while (1) {
        int a = 0;
        int x;
        char op;
        int n = fscanf(input, " %c %d", &op, &x);
        // ...
}
```

This is done using the `scanf` family of functions. The format string in our case uses the place holders `%c` for a single character and `%d` for decimal integers. Any space matches any number of white space inbetween, thus the given format string matches inputs like (`\n` denoting a new line):

```
+ 42
    +      42
+\n    42
```

Following the format string we pass pointers to memory locations (using `&` to take the pointer from the original variable) which allow the function to store the extracted and converted data, `char op` for `%c` and `int x` for `%d`.

But what if we have invalid input? The `scanf` functions return an integer which denotes the number of *successful* conversions and assignments. That is, we expect this number to be two, since we have two conversion specifiers—if it is less, at least one match has failed. We thus can continue with doing the actual math as seen in **Listing 2**.

First we to distinguish the four different valid cases for our operator and the fifth, invalid, case (e.g., input `"= 42"`). This can be done with a switch-case statement which allows to do simple equality comparison of one expression (`op`) against a list of values. It's important to remember to end each case with a `break` statement, as otherwise it would 'fall through', i.e., in case of the `+`, first `acc += x` would be executed, then `acc -= x`, then the multiplication and so forth.

The final step within the loop is to print the current value of the accumulator, which is done with a function practically opposite: It takes a format string with place holders, but instead of matching string input, it replaces these with the values we provide, thus constructing a string from, e.g., decimals. In this case, we ask the function to replace the `%d` with the decimal representation of the number stored in `acc`. Note the new line character `\n` at the end which is essential for the output actually being printed.

```
if (n != 2) {
        // exit our function
        return;
}
switch (op) {
case '+':
        acc += x;
        break;
case '-':
        acc -= x;
        break;
case '*':
        acc *= x;
        break;
case '/':
        acc /= x;
        break;
default:
        // none of the above matched
        return;
}
printf("acc = %d\n", acc);
```

Listing 2 Processing the parsed information