4. Text parsing

Ján Dugáček

November 10, 2017

Table of Contents

Overview

- You know quite a bit about this already
- We'll study how to open files, read them and write new ones
- We'll learn a few useful tricks and about parsing files that are not human-readable
- This is mostly an exercise for pointer usage
- C++ offers many tricks to make this easier, but they are now always applicable (or grossly inefficient in the situation)

Input with getchar()

```
char buffer[30];
char got;
int i = 0;
for ( ; i < 29 && (got = getchar()) != '\n' && got; i++)
        buffer[i] = got;
buffer[i] = 0;
```

- Here, we first define an array for storing text `buffer`, a variable to store read characters `got` and a position iterator `i`
- Then we use the function `getchar()` to read from input (what is written into the command line after the program is started)
- It is read until it finds a newline symbol or the input ends or the array is full
- After the cycle, a terminating character is set
- `getchar()` reads the line after a newline is pressed, until that, the program sleeps
- `stdio.h` needs to be included for this

scanf()

```
char buffer[30];
scanf("%s\n", buffer);
```

- `scanf()` is a function that parses text in most cases, returns the number of variables parsed
- It's much like a reverse `printf()`
- If the input is too long, the function writes behind the array (*buffer overflow*), which is one of the primary ways how systems get hacked (reading integers or floats in this way is fine)
- To prevent it, use function `scanf_s()` that accepts an additional argument after each string that contains the maximum size allowed (Microsoft-only, unfortunately)

Exercises

1. Write a program that reads one line of the user's input, writes it back and exits (assuming maximum text size 100)
2. Write a program that reads several lines of numbers (until an empty line is inserted) and writes back their average
3. Write a program that reads one line of the user's input, replaces letters with capitals, writes the result and exits (assuming maximum text size 100)
4. Write a program that reads a 3x3 matrix (space separates numbers on one line, newline separates lines) and outputs its determinant (`https://en.wikipedia.org/wiki/Rule_of_Sarrus`)

Advanced:

1. Write *Angry internet poster simulator* that reads input of any size, accepts a command line argument determining the percentage of words that will be capitalised and writes the result back
2. Write a program that reads a matrix of any size (space separates numbers on one line, newline separates lines) and outputs its determinant (`https://en.wikipedia.org/wiki/Laplace_expansion`)

Files

```
FILE* file = fopen("file.txt", "r");
int number;
fscanf(file, "%i\n", &number);
char character = fgetc(file);
fclose(file);
```

- `fopen()` opens a file, file name (if it is in the same folder, otherwise there can be the path to the file) is the first argument, if reading, second argument is `"r"`
- `fscanf()` is a version of `scanf()` for reading files, the only difference is that the file is placed before its first argument
- Use `feof()` to check if there's still something to read
- `fscanf()` is also potentially insecure and `fscanf_s()` might be useful if using a Microsoft compiler
- Also, `fgetc()` is analogical to `getc()` for reading from files, it returns `EOF` if there is nothing left to be read
- The file has a position, so multiple reads read parts that follow one after another (you can use `rewind()` to get to the beginning of the file)
- File should be closed with `fclose()`

Files

```
FILE* file = fopen("file.txt", "w")
fprintf(file, "Hello world!\n");
```

- `fopen()` opens a file, if the second argument is `"w"`, the file is created (if it exists, it is cleared) and ready to be written into, if it is "a", new text will be appended to its end
- `fprintf()` is a version of `printf()` for writing into files, the only difference is that the file is placed before its first argument
- `fprintf()` is faster than `printf`, so it can be useful to write into files if a lot of information needs to be printed (in the order of tens of megabytes)

Exercises

1. Write a program that reads numbers from a file (one number per line) and writes the largest one
2. Write a program that writes a table of the sine function into a file (x and $\sin(x)$ separated by tab on each line)
3. Write a program that reads one file, replaces all letters by capitals and writes that into another file

Advanced:

1. Write a parser of simple commands that reads files and executes them, supports 26 variables (from a to z), the file name is given as command line argument and result printed, should be able to execute this:

```
a=3
b=2+a
c=b*a
d=c-a
return d
```

Custom variable type: struct

```
struct someStuff {
        short int index;
        char stuffType;
        float value;
};
struct someStuff a = {1, 'a', 12.5};
```

- struct is a custom variable type, composed of other variable types (not necessarily primitive ones)
- In memory, they are saved similarly to arrays, but the elements are named and may be of different types (with different sizes)
- They may be initialised like arrays, but unlike arrays they are copied when given as function arguments

struct

```
struct someStuff {
        short int index;
        char stuffType;
        float value;
};
struct someStuff a = {1, 'a', 12.5};
```

- It is important to know that variables smaller than word size (that is 8 bytes on 64-bit architectures) are stored only on addresses divisible by their size (larger ones must be on addresses divisible by word size), so `index` (`short int`, 2 bytes) will always be saved on an address divisible by 2, `value` (`float`, 4 bytes) on an address divisible by 4 and `stuffType`, `char`, `size 1` can be saved anywhere
- Although `index` and `stuffType` occupy only bytes 0, 1 and 2, byte 3 cannot be occupied by `value` because it's not divisible by 4 and thus it will be saved at bytes 4-7
- The size of `someStuff` is 8

struct #2

```
struct parsing {
        char first[4];
        char end;
        char second[4];
        char end2;
};

char unparsed[] = "0245 3245";
struct parsing parsed = *((struct parsing*)unparsed);
parsed.end = 0;
parsed.end2 = 0;
printf("%i, %i\n", atoi(parsed.first), atoi(parsed.second));
```

- Structure `parsing` is a custom variable type, composed of 10 variables of type `char`
- Because it is identical to an array with 10 elements, we can covert an array to it
- Named members are accessed using the `.` operator, if we have a pointer to the struct, we use `->` instead

struct #3

- `struct` may be used only in parsing of files where everything has a fixed position on its line
- However, programs often store data in formats that are not human readable, often in the form of `struct` directly saved into a file, with values mostly set over 4 bytes in IEEE 754, meaning they may contain anywhere the 0 character that ends strings; in that case, functions like `fscanf` are useless
- To read or write files like this, use `"rb"`, `"wb"` or `"ab"` (depending if you read, write or append) as arguments to `fopen`
- `struct` is incredibly useful for many other things as we shall see later

union

```
union someStuff {
        char raw[8];
        double number;
};
union someStuff a;
for (int i = 0; i < 8; i++)
        a.raw[i] = fgetc(file);
printf("Read %f\n", a.number);
```

- union is similar to struct, but its contents are saved on the same place instead of one after another, allowing to access the same data as different types comfortably

enum

```
enum logic {
        no = 0,
        maybe = 1,
        yes = 2
};

enum logic logic_and(enum logic a, enum logic b) {
        if (a == yes && b == yes) return yes;
        if (a == no || b == no) return no;
        return maybe;
}
```

- enum is a custom variable type whose values are named
- Using numbers instead is less readable and makes adding new values in the middle very, very troublesome

enum #2

```
enum coordinate {
        x,
        y,
        z,
        coordCount
};

float position[coordCount];
for (int i = 0; i < coordCount; i++)
        position[i] = 0;
position[x] = 12;
```

- enum is very useful for naming elements in an array
- An element after the last valid one is the quantity of valid ones and can be used in iteration or array sizes

enum #3

```
enum flag {
        ignoreProtection = 1,
        truncate = 2,
        backup = 4
};
void changeFile(char* name, unsigned int flags) {
        if (flags&ignoreProtection) {/*...*/}
        if (flags&truncate) {/*...*/}
        //...
}
//...
changeFile("file.txt", ignoreProtection | truncate);
```

- 1 is 1 in binary, 2 is 10 in binary, 4 is 100 in binary
- Operator & is bitwise and, so 110 & 011 yields 010, 011 & 001 yields 001 etc.
- Operator | is bitwise or, so 110 | 011 yields 111, 010 | 001 yields 011 etc.
- Together with `enum`, it can be used to pack 32 options into an `int`

Combinations

```
union entry {
        char raw[24];
        struct {
                int index;
                int age;
                float importance;
                float coordinates[3];
        };
};
union entry a;
for (int i = 0; i < 8; i++)
        a.raw[i] = fgetc(file);
printf("Read %i of age %i\n", a.index, a.age);
```

- `union` and `struct` can be contained freely in each other
- If the internal ones are not named, the elements are accessed as if they weren't deeper

Combinations #2

```
enum storedType {
        maybeObject = 0,
        nil = 1,
        character = 2,
        shortString = 3,
        integer = 4,
        floating = 5
};
union value {
        struct object* obj;
        struct {
                union {
                        char asChar;
                        char asString[4];
                        int asInt;
                        float asFloat;
                };
                enum storedType type;
        };
};
```

- It may be used to store more types in one (but the program will need clues to know which type is actually stored there)
- This can be used to implement dynamically typed languages like Python
- As long as `struct object` is at least 8 bytes large, its address will be a multiple of 8, which does not include cases where the type field is 1-5

Shortening

```
struct thisIsATerriblyLongName { int a; int b; };
typedef struct thisIsATerriblyLongName longName;
longName a;
```

- Some types are annoying long (this is particularly bad in C++), sometimes it is annoying to write the `struct` prefix everywhere, so `typedef` can be used to shorten it

```
typedef struct {
        int a;
        int b;
} someStruct;
```

- It is not necessary to name the `struct` before renaming it

Exercises

1. Write a program that reads a file that contains floats (in IEEE 754), preceded by the an unsigned int that contains the quantity of these numbers (example file is available)
2. Write a program that reads pairs of numbers from a file (a pair is always on one line) and writes their sums into another file (file names are given as command line arguments), all numbers in the read file will be formatted as follows:

```
0012 3242
2421 0921
9212 0424
```

Advanced:

1. Write a parser of a simple nested markup language (of your design or choice) and save it in a recursive structure as follows:

```
struct block {
        char* text; // The contents
        struct block* children; // An array
        struct block* parent; // In whose children it is
}
```

Homework

- Read the file with table of function $f(x) = (\sin(x - c))^{a-x} \frac{b}{x+a}$, always x and $f(x)$ on one line, and determine the coefficients a, b and c
- Advanced homework: write a parser that can parse and execute this program, supporting 26 variables `a` - `z` that can contain both string and number (the result should be 14)

```
a = 2
b = "2"
c = "3" + b
if (c == 32)
        d = (c / 2) - a
return d
```

- Challenge: support also some sort of `while` or `for`