Prof. Dr. Stefan Brass
Institut für Informatik
MLU Halle-Wittenberg

November 21, 2019

Databases IIB: DBMS-Implementation
— Exercise Sheet 5 —

Please upload your solution into the StudIP file folder called "Hausaufgabe_5" in the StudIP entry of the lecture. The deadline is November 26 (the day before the next lecture).

It is permitted to form groups of up two members, but please make sure that both members can fully explain all homeworks submitted by the group.

Please upload only one file per group. I can unpack archives in `zip`, `tar` and `rar` formats. Please make sure that unpacking the archive yields a directory with your name(s).

Homework Exercise 5

This exercise continues Homework 2. If you did not submit Homework 2, you have two options: You can use my solution available at

[http://www.informatik.uni-halle.de/~brass/dbi19/h2/h2.zip]

Alternatively, if you develop your own solution, you will get the point for Homework 2.

Download the file

[http://www.informatik.uni-halle.de/~brass/dbi19/homework/homeworks.txt]

It contains the data of the example table from Homework 2 (homework points with columns `FIRST_NAME`, `LAST_NAME`, `EXERCISE_NO`, `POINTS`). The file contains one line per table row, and the columns are separated with a vertical bar "`|`", e.g. the first line is

```
Ann|Smith|1|10
```

Please change your `main` program from Homework 2 such that it

- opens the file,
- reads all lines in the file,
- parses each line and creates an object of your homeworks class,
- stores the object in an array (store at most 100 objects).
- Then write a small query method that prints students (first and last name) with the maximal number of points for Homework 1.

Of course, you can define auxiliary functions or even a class (e.g. for the relation). Note that if you use a class only in one source file, you do not need separate `.h` and `.cpp`-files for the class. Simply define the class in the source file where it is needed.

Working with Files in C++

Here is some information about accessing files:

- There are different ways to access files. The high level C++ class for file input streams is `std::ifstream`. You must include `<iostream>` and `<fstream>` for the declaration of this class. You can find the documentation of this class e.g. here:

  [http://www.cplusplus.com/reference/fstream/ifstream/]

  An alternative documentation source is

  [http://en.cppreference.com/w/cpp/io/basic_ifstream]

  (This really documents a template, of which `ifstream` is an instance.)

- You open the file by calling the `open` method of the `ifstream` object.

  [http://www.cplusplus.com/reference/fstream/ifstream/open/]

  Its first parameter is the file name, e.g. `homeworks.txt`. The optional second parameter are mode bits. If the file is a binary file, i.e. the system should not replace Windows line ends (CR-LF: `"\r\n"`) by the UNIX version (LF: `'\n'`), one can open it as follows (with an object `File` of class `ifstream`):

  ```
  File.open(filename, std::ios::in|std::ios::binary);
  ```

  However, this is a text file. Note that if you use an IDE, it might execute the program in a subdirectory of your project directory (such as `Debug` for the debug configuration in Visual C++). One solution would be to specify the full path name (do not forget to escape "\" as "\\" in the string constant).

- The method `open` has no return value and does not throw exceptions (unless specifically requested). Actually, one can program in C++ without using exceptions at all. One can check `File.fail()` or `File.is_open()` after the call to `open` in order to check whether the file was successfully opened.

  If you want to access the error message of the operating system, the numeric code of the last error is stored in the global variable `errno` declared by including `<errno.h>`. This can be translated to a string with the function `strerror` declared in `<string.h>`, i.e. `strerror(errno)` should work (at least under Linux). If there are problems, you can also look for the functions `strerror_s` and `perror`.

- One method to read a line from the file into a character array is

```
getline(char *buf, std::streamsize bufsize).
```

The type `std::streamsize` is system-dependent, it can e.g. be `int` or a type for 64-bit integers (`long long` or `int64_t` defined in `stdint.h`). An `int` value for the second parameter will be ok (it is automatically converted to a larger type if needed). E.g. one declares the array "`char buf[80];`" and calls "`File.getline(buf, 80);`".

The method reads characters from the input stream until (1) the end of file was reached, or (2) the end of line was reached, or (3) no more characters fit into the buffer, i.e. `bufsize-1` characters were read. In each case, the string in the buffer will be terminated with a null byte. The newline `'\n'` is not stored in the buffer.

The function returns the stream object itself (which might be helpful for sequences of `getline`-operations). The number of characters read in the last such operation can be accessed with the method `gcount()`. If the end of file was reached during the operation, the method `eof()` will return `true`. If the line was too long for the buffer, `fail()` will return `true`. If one wants to continue reading, one must reset the state bits of the stream object with the method `clear()`. (If `bad()` should be true, the stream is corrupted, and one cannot expect that one can continue reading.)

There is also a version of `getline` with an additional argument for the delimiter character (actually, there is only one version, but `'\n'` is the default value for the delimiter). If you prefer, you can use this with the delimiter `'|'` for the first fields. The delimiter will be read from the file, but not stored in the array.

- Streams can be automatically converted to boolean values, and return the negation of the `fail()` function in this case. I.e. "`while(File.getline(buf, 80))`" is possible, although the function returns a reference to the stream object `File`. C++ permits that the programmer defines type conversion functions for own classes. (In this case, the conversion was actually to `void *`, but the language treats a null pointer as false, and all other pointers as true.) Also the operator `!` is defined for streams, thus `if(!File.getline(buf, 80))` can be used to print an error message.

- If you want to convert a string to an integer, you can e.g. use the library functions `strtol` ("string to long integer") for standard character pointers or `stoi` for string objects. Other alternatives include `sscanf` (formatted input in C) and streams reading from strings: `istringstream`.

  The function `strtol` declared in <cstdlib> has three parameters: The buffer containing the string to be converted (`const char*`), an address of a pointer which will be set to point to the next character after the number (`char **`), and the base of the number representation (`10` for decimal numbers). If you declare a variable "`char *end;`" you can write `&end` for the second parameter. By using this pointer, one can continue reading further data in the string after the number. The function returns the converted value as a `long`. It might be necessary to explicitly write a cast to `int`. The function returns `0` if the input did not start with a digit (possibly after skipping whitespace). Since this is a valid value, one really should check whether `*end == '\0'` afterwards (i.e. nothing remains in the string buffer that was not converted). It is also possible to check for overflows, see e.g.

  [https://stackoverflow.com/questions/14176123/correct-usage-of-strtol]

  The function `sscanf` (declared in <cstdio>) can convert several data items based on a format string. If one wants to read a single integer, the call would be

```
int success = sscanf(buf, "%d", &n);
```

  Here `buf` contains the input, it has type `const char *`, and one can pass e.g. the name of an `char[]` array. The variable `n` must have type `int`, and is set to the conversion result. The function returns the number of successfully converted format elements, i.e. `success` will be `1` in the positive case and `0` if the buffer did not contain a valid number. Note that the conversion is successfull if the buffer starts with a number, there might be additional characters not used, e.g. `"123abc"` as first argument would set `n` to `123` and return `1`.

  If you want to check whether a character `c` is a digit, you can use `isdigit(c)` defined in <cctype>.

- At the end, you should close the file with `close()`. This would give you the option to check for errors (the method returns no value, but one can call `fail()` afterwards). At least for output files, it would be possible that there is still output in the internal buffer of the stream object, and that an error occurs while this is written to the file.

  However, you do not have to close the file, because the destructor of the `ifstream` class automatically closes the file if it is still open. This technique to manage the allocation of a resource with a local variable is called RAII ("resource acquisition is initialization").

- It is also possible to use lower level interfaces for working with files. C had file pointers (type `FILE*`) defined in <cstdio>.

- There is also a direct interface to the UNIX system calls `open`, `read`, `write`, and `close`. These functions work with file descriptors (small integers) to identify files. Similar functions are available under Windows (there the numbers are called file handles), but there might be differences in the details and the necessary include files are different.

  On Linux/UNIX, `read` and `write` are declared in in `unistd.h`:

  [http://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html]

  `open` is declared in `fcntl.h`:

  [http://pubs.opengroup.org/onlinepubs/7908799/xsh/fcntl.h.html]

  It might be necessary to include also `sys/types.h`, `sys/stat.h` and `sys/uio.h`.

  For Visual Studio on Windows, use the include file `io.h` and read e.g.

  [https://msdn.microsoft.com/en-us/library/z0kc8e3z(v=vs.140).aspx]