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

November 03, 2017

Databases IIB: DBMS-Implementation
— Exercise Sheet 4 —

Part a) to f) will be discussed in class, you only have to submit Part g) and h). Please send your solutions to the instructor via EMail: `brass@informatik.uni-halle.de` (with "`dbi17`" in the subject line). The official deadline is November 8, 12:00.

Some Feedback on Homework 2

- Please make sure that all submitted homeworks can be compiled. If you need help, clearly mark in the subject line of your email that you have a question (and include the error message of your compiler). I am quite generous with late homeworks, but I do not want to get homework submissions with compilation errors.
- Of course, the difference between returning a value (as the result of a method) and printing a value must be clear. The accessor methods for the attributes must return the value and print nothing.

  Probably most classes in a complex system should not print anything, since this reduces possible usages of the class. Unexpected output might e.g. destroy the format of a message returned from the server to the client. Furthermore, today most programs have a graphical user interface and it is not clear whether output sent to `cout` will be seen by anybody.
- Some students declared the accessor method for `firstname`/`lastname` as returning "`char *`" instead of "`const char *`". Since the method returns only the address of the array in the object, it would be simple for the caller to assign something and destroy the contents of the array. The whole point of having accessor methods and not making the attribute `public` was to protect the array and its contents. Note that if the method is declared as `const` (i.e. not changing the object state), one cannot give non-`const` pointers to object components away.
- Make sure that you understand that it is possible to write the method body already in the class declaration in the `.h`-file. This has the advantage that the compiler can replace the call to the method directly by its body (if it is not recursive and not too long). It has the disadvantage that the class declaration might become less readable because the human reader (in contrast to the compiler) needs only the interface of the method, not its implementation. Since the accessor methods contain only a single statement in their body (the `return` for the attribute value), this disadvantage is very small in this case.

  Note that you are still free to change the implementation of the method later. However, it would require recompilation of all code that used the class. If you only change a method body in a separate `.cpp`-file, it would suffice to recompile that file and link all the object files. But adding an attribute to the class would in the same way require recompilation of all source files that used the class, because more memory is now needed for the objects of the class. Therefore, there are anyway changes of the implementation of the class that need this recompilation.
- Part of the specification was to copy all the characters of the string in the constructor, not only the pointer. E.g. if you declare the the attribute in the object as "`const char *first_name;`" and the constructor parameter with the same name and type, then the assignment

  ```
  this->first_name = first_name;
  ```

  copies only the address of the array that contains the characters of the name, but not the contents. If this array is later reused, for instance when you read the next input line, then the attribute still points to the same array, but a different name is stored in it.
- One option is to declare an array of sufficient size as an attribute of the object:

  ```
  char first_name[21];
  ```

  Note that you need the array length 21 if you want to store names up to length 20, because an additional byte is needed for the null-termination of the string (end-of-string marker `'\0'`).
- The other option is to find out how long the name is and to allocate an array of the right size.

  ```
  int length = strlen(first_name);
  this->first_name = new char[length+1];
  ```

  If you did this, you should free the array in the destructor. Note that this has to be done with

  ```
  delete[] this->first_name;
  ```

  The simple `delete` (without `[]`) cannot be used for arrays.
- Note that the function `strcpy` from the standard C library is inherently unsafe: It copies all characters of the string, even if the array is too small, which means that other variables stored in memory after the array may be overwritten. Whereas in Java, arrays are objects and contain an attribute for their length, arrays in C++ are simply n variables of the base type allocated in consecutive memory locations. E.g. a `char`-array of size 20 will be exactly 20 bytes long (`char` is one byte long, except possible on extremely strange machine architectures). Since `strcpy` gets only the start address of the array, and no information about its length, it is clear that

it cannot check the boundaries. There is another function `strncpy` which has an additional parameter for the array length n, but unfortunately it will simply stop after copying n characters, thus the null-termination of the string might be missing. Then future usages of the string, e.g. when printing the name, will violate the array boundaries because they print bytes until they find a null byte. However, if you first computed the length of the input string with `strlen`, and then allocated an array of the right size, `strcpy` and `strncpy` will be safe (until the contents of the input array changes).

- Note that good testing code in the `main` program would include a name that is longer than 20 charaters. Very few students did that.
- Many students repeated the loop for copying the contents of the input array for `firstname` and `lastname`. Because of the limit of 20 characters, this loop is not extremely simple and short. Therefore, this is a case of Copy&Paste programming which should be avoided. One should write an auxiliary `private` method for copying the string. The method must be declared in the class declaraction in the `.h`-file, but because it is private, it does not change the interface of the class.

In-Class Exercises

a) Download and run again the SQL file that creates the Oracle example tables:

[http://www.informatik.uni-halle.de/~brass/dbi17/empdept.sql]

Execute it while you are logged into SQL*Plus as your own account (not `SYSTEM`) with the following command:

```
@empdept
```

Please note that the home directories on the virtual machine running Oracle are different from your normal home directories. You must download the file on that machine, e.g. with `wget`. You can also use `scp` (Linux) or `pscp` (PuTTY on Windows) to copy files between different machines. Note that the SQL script first drops all the table that are recreated later. When you execute the script for the first time, you will get error messages for the `DROP TABLE` commands, these are normal. SQL*Plus continues the execute the commands in a script, even if a command failed.

If the `CREATE TABBLE` or `CREATE VIEW` commands fail, you probably do not have the needed system privileges. Since you have the password for the DBA account `SYSTEM`, you can correct that.

b) Print a list of tables that you own. Do not worry about tables with names like `"BIN$XRhqbCOPnUjgUDCNZgk7lg==$0"`. These are dropped tables that are still in the recycle bin and can be recovered if necessary. Write a condition that excludes tables starting with “`BIN$`”.

c) Try the SQL*Plus command to list the columns of a table:

```
describe ⟨Table⟩
```

Note that keys and foreign keys are not listed. Try to write an SQL query to the data dictionary that gives similar information as the `describe` command. It is quite a challenging task, but possible, to get exactly the same output as the `describe` command (you have to use `UNION` and the string concatenation `||`). However, you do not have to do that (it suffices if your query contains the same information).

d) In order to repeat SQL, write the following queries:

- For each department (table `DEPT`), print the department number (`DEPTNO`), the department name (`DNAME`), and the number of employees (table `EMP`, it contains the foreign key `DEPTNO`).
- Is there are department without employees?

e) Get the definition of the view `SALES` from the data dictionary (you can compare the result with the definition at the end of the file `empdept.sql`). The required data dictionary table is `USER_VIEWS` (of course, you could find that out by searching for a table in `DICT` that contains the substring `VIEW`). Note that it might be necessary to use the SQL*Plus command

```
SET LONG 1000
```

to increase the number of characters shown for values of type `LONG`. You should also use

```
SET PAGESIZE 100
```

to increase the number of lines shown before the table header is repeated. Note that if you want to re-run the last query (e.g. after changing SQL*Plus settings), this can be done with the SQL*Plus command "run":

```
r
```

It is also possible to replace a string in the current line of the query, e.g.

```
c/abc/xyz/
```

The following command lists the last SQL command:

```
l
```

The current line is marked with "`*`". The change command "`c`" applies only to the current line. One can set the current line e.g. with

```
l2
```

This lists only line 2 and makes it the current line. The "append" command can be used to append text to the current line:

```
a xyz
```

Finally, use `r` to run the modified query. If this simple line-oriented editor that is built into SQL*Plus is not sufficient, you can specify an external editor, e.g. the "`vi`":

```
define _editor = vi
```

Then the command

```
edit
```


can be used to call this editor for the current SQL command. You can use the help command to get more information on a command, e.g.

```
help edit
```

The command

```
help index
```

shows a list of commands.

f) Look at the table `PRICE` (which was also created by `empdept.sql`). The format of the columns `STDPRICE` and `MINPRICE` does not look nice: Prices should be printed with two decimal places after the decimal point, but the default number format prints only decimal digits if needed (i.e. not `0`). Have a look at the Section "Format Models" of the Oracle "Database SQL Language Reference":

[https://docs.oracle.com/cloud/latest/db112/SQLRF/sql_elements004.htm]

Use the following SQL*Plus command to define a format for all columns called "`STDPRICE`":

```
COLUMN STDPRICE FORMAT 9990.00
```

Note that column settings are kept only until the end of the SQL*Plus session. One can write then into an SQL-file that is executed with `@file`. The file `login.sql` is read automatically each time one starts SQL*Plus. More information about this can be found in the "SQL*Plus User's Guide and Reference" (in the Section "Configuring SQL*Plus"):

[https://docs.oracle.com/cd/B28359_01/server.111/b31189/toc.htm]

The "SQL*Plus Quick Reference" is also useful:

[https://docs.oracle.com/cd/B28359_01/server.111/b31190/toc.htm]

Homework Exercises

g) Write an SQL query that shows the original names of all deleted tables in the data dictionary together with the date when they were deleted. Read the Section "Using Flashback Drop and Managing the Recycle Bin" from the Chapter "Managing Tables" from the Oracle "Database Administrator's Guide":

[https://docs.oracle.com/cd/E11882_01/server.112/e25494/tables.htm]

Please answer also the following questions:

- Which command can be used to really delete a table from the recycle bin?
- Which command can be used to recover a table from the recycle bin?

h) Consider again the file:

[http://www.informatik.uni-halle.de/˜brass/dbi17/homeworks.txt]

It contains the data of the example table:

HOMEWORKS			
FIRST_NAME	LAST_NAME	EXERCISE_NO	POINTS
Ann	Smith	1	10
Ann	Smith	2	8
Michael	Jones	1	9
Michael	Jones	2	9
Richard	Turner	1	5

The file contains one line per table row, and the columns are separated with a colon.

Please write a program that prints the sum of points of each student. It should use a linked list of objects of a class `Student`:

- The class should manage the linked list of all its objects. Thus, you declare static variables in the class, e.g. for the start of the linked list and possibly for the last element:

```
static Student *FirstStud;
static Student *LastStud;
```

  Note that this is only a declaration of the static variables, i.e. it tells the compiler that there will be such variables. One has to define the variables in the `cpp`-file in order to reserve memory for them:

```
Student *Student::FirstStud = 0;
Student *Student::LastStud = 0;
```


  The reason for this separation is that the `.h`-file will be seen by the compiler each time it compiles a source file that uses the `Student`-class. But memory should be reserved only once, namely when it compiles `student.cpp`. This ensures that there is a unique address for the variable. Otherwise the linker will complain about "doubly defined symbols". For normal attributes (non `static`), there is no such trouble, because they correspond only to offsets inside the memory for each object. The memory is reserved when the object is created, and there can be any number of objects of the class.

- The constructor has only two parameters for first and last name. The number of points is initialized to 0. The objects also contain a pointer to the next `Student` object on the linked list. This is initialized to the null pointer (if one adds objects at the end of the list). The constructor is responsible for adding the new object to the linked list. No specific order is required. Remember that `this` is a pointer to the current object.
- For simplicity, we assume that all objects exist until the program terminates. This is somewhat dangerous, but it is ok for this homework. If you want, you can write a destructor that removes the object from the linked list. In this case, you need a doubly linked list. But this is optional.
- There should be accessor methods for the four attributes:
  - `get_firstname()`,
  - `get_lastname()`,
  - `get_points()`,
  - `get_next()`.

  In addition, there should be a method `add_points(int n)` that increases the number of points by `n`.
- There must also be a `static` method `get_first` that returns the first student, so that users of the class can loop over all students by following the link returned by `get_next`. A typical loop looks like this:

```
for(Student *s = Student::get_first(); s; s = s->get_next())
```

- Finally, there should be a `static` method

```
Student *find(const char *firstname, const char *lastname)
```

  that returns the Student object for given first and last name, if it exists, or a null pointer otherwise. If you prefer, you can do the searching of a student also in the `main` program.
- An alternative design would be to have a static method that either returns an existing student with the given name, or creates a new `Student` object. In this case, the constructor would be `private` so that it can only be used in this method. This would ensure that there are never two objects with the same name.