Introduction to Software Systems (308-206B)
Assignment 1: a hashed string pool

Winter Term 2001

Practical information

- Due date: Thursday 8 February, before 23:59. Submissions after the due date will be discarded.
- Submission medium: VisualCM. Submit *one* file: `hashpool.c`, not your input/output. Beware: your submission is for 206B, *not* for 206A !
- *Individual* solutions and submissions. I will (pseudo-)randomly check whether you can explain your "own" assignment.
- You may develop your code on any platform. Your code will however be evaluated on a SOCS UNIX machine. Make certain it runs correctly on a SOCS UNIX machine. Compile your code with

```
gcc -Wall -g -ansi -o hashpool hashpool.c
```

- Your code will be evaluated by comparing (using the UNIX diff) its output with correct output. Be certain to generate *exactly* the same output (including blank lines) as in the examples below !

Objectives

- Thorough understanding of C *arrays* and *strings.*
- An exercise in top-down and/or bottom-up *design.* A good design will be elegant and include all special cases (*e.g.*, no input, too much input).
- Thorough *testing* of code: print out and inspect internal data structures, test all special cases.
- An exercise in *coding style.* As no functions are used, it is necessary to comment and indent properly to tackle complexity. Make the code re-usable by including requirements and design information in the comments as well as by making use of named constants (rather than inline constants).

Assignment

A program `hashpool` needs to be built which keeps asking the user to enter a string until the empty string (immediate return pressed) or the string "halt" is entered. You may assume the maximum number of characters entered by a user will be 100 (and must reserve enough memory accordingly). It is left unspecified what will happen if more than 100 characters are input. You should specify (in comments) how your program will handle this case and implement this specification. Any "reasonable" behaviour is allowed.

Each new string entered should be added to a string pool, if it is not already present in the string pool. The string pool is a character vector of length 150. If the string is already present in the string pool, this is mentioned to the user.

To avoid having to scan the whole string pool to check if an entered string is already present, a hash table or string index table (of integers) with 5 buckets (rows) and chain length 3 (columns) is used. The hash table is initialized to all -1 elements. Non-negative values are used as indices in the string pool, indicating the start of a string. An entered string is "hashed" into an integer. This integer indicates a row (a "bucket") of the hash table. Each element of the row is

- either a non-negative integer, indicating the start of a previously stored string in the string pool,

- or a negative integer (-1) indicating that the string is not present in the string pool.

The entered string is compared with strings in the string pool pointed at by the non-negative entries in the row. If an equal string is found, the user is told so and the entered string is not added to the string pool. If none of the non-negative entries in the row (if any) point to a string equal to the entered one, the entered string is added to the string pool (behind the last added one; adding starts from position 0).

Hashing is done based on the ASCII values of the characters in the string as demonstrated below:

```
int hash_value=0;
int index;
for (index=0; index<strlen(string); ++index)
 hash_value = (hash_value+string[index]) % 5;
```

After the user finishes entering strings, the two main data structures (string pool and hash table) must be output in *exactly* the format as given in the examples. Note how a `'\t'` is used to separate elements in the hash table. Also, the formatting directive `%3d` is used to print positions.

Some extra requirements:

- The solution should be coded as *one* large `main()` function. The problem should *not* be split up into smaller functions (though that should in the future *always* be done).
- It should be easy to change the size of the string pool and hashing table.
- Array overflow of the string pool and the hashing table must be detected. If either of these occur, the program should print a message and exit. The exiting program must return a `1` to the caller (the shell) in case of string pool overflow and a `2` in case of hashing table overflow. Make sure to test for overflow *before* it occurs.

Input/output behaviour

Ending the input

1. Ending input by means of empty string (hashpool.end.empty.txt)

```
hv@lookfar 136% hashpool
Enter a string:

String pool of length 150 contains:


End of string pool


5 x 3 string index table:

  row 0:        -1      -1      -1
  row 1:        -1      -1      -1
  row 2:        -1      -1      -1
  row 3:        -1      -1      -1
  row 4:        -1      -1      -1

End of string index table
```

2. Ending input by means of “halt” string (hashpool.end.halt.txt)

```
hv@lookfar 137% hashpool
Enter a string: halt
```

```
String pool of length 150 contains:


End of string pool


5 x 3 string index table:

  row 0:        -1      -1      -1
  row 1:        -1      -1      -1
  row 2:        -1      -1      -1
  row 3:        -1      -1      -1
  row 4:        -1      -1      -1

End of string index table
```

Normal operation

1. Normal operation without duplicate strings (hashpool.normal.txt)

```
hv@lookfar 138% hashpool
Enter a string: abc
bucket number for "abc" is 4
Enter a string: 1
bucket number for "1" is 4
Enter a string: 2
bucket number for "2" is 0
Enter a string: 3
bucket number for "3" is 1
Enter a string: 4
bucket number for "4" is 2
Enter a string: 5
bucket number for "5" is 3
Enter a string: 6
bucket number for "6" is 4
Enter a string: 7
bucket number for "7" is 0
Enter a string: 8
bucket number for "8" is 1
Enter a string:

String pool of length 150 contains:

 At position   0, string "abc"
 At position   4, string "1"
 At position   6, string "2"
 At position   8, string "3"
 At position  10, string "4"
 At position  12, string "5"
 At position  14, string "6"
 At position  16, string "7"
 At position  18, string "8"

End of string pool


5 x 3 string index table:
```

```
  row 0:        6       16      -1
  row 1:        8       18      -1
  row 2:        10      -1      -1
  row 3:        12      -1      -1
  row 4:        0       4       14

End of string index table
```

2. Normal operation with duplicate strings (hashpool.normal.duplicate.txt)

```
hv@lookfar 141% hashpool
Enter a string: a complete sentence
bucket number for "a complete sentence" is 1
Enter a string: abcd
bucket number for "abcd" is 4
Enter a string: if
bucket number for "if" is 2
Enter a string: then
bucket number for "then " is 3
Enter a string: a complete sentence
bucket number for "a complete sentence" is 1
String already present in string pool
Enter a string: halt

String pool of length 150 contains:

 At position   0, string "a complete sentence"
 At position  20, string "abcd"
 At position  25, string "if"
 At position  28, string "then "

End of string pool


5 x 3 string index table:

  row 0:        -1      -1      -1
  row 1:        0       -1      -1
  row 2:        25      -1      -1
  row 3:        28      -1      -1
  row 4:        20      -1      -1

End of string index table
```

Data structure overflow

1. String pool overflow (hashpool.stringpool.overflow.txt)

```
hv@lookfar 148% hashpool
Enter a string: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bucket number for "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" is 0
Enter a string: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bucket number for "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb" is 0
Enter a string: abcabc                          abcabc
bucket number for "abcabc                          abcabc" is 3
Enter a string: def                             def
```

```
bucket number for "def                              def" is 4
Enter a string: a very long ...                 ... sentence
bucket number for "a very long ...                  ... sentence" is 2

>>> Error: string pool overflow
```

2. Bucket overflow (hashpool.bucket.overflow.txt)

```
hv@lookfar 144% hashpool
Enter a string: 1
bucket number for "1" is 4
Enter a string: 2
bucket number for "2" is 0
Enter a string: 6
bucket number for "6" is 4
Enter a string: 10
bucket number for "10" is 2
Enter a string: 13
bucket number for "13" is 0
Enter a string: 14
bucket number for "14" is 1
Enter a string: 17
bucket number for "17" is 4
Enter a string: 23
bucket number for "23" is 1
Enter a string: abc
bucket number for "abc" is 4

>>> Error: bucket 4 overflow
```

Resources

- The template to use for your C code: template.c
- Be consistent in your indentation. Using the GNU tool “indent” can help. Below are some reasonable command-line options (which you may wish to put in your .cshrc file if you use csh).

```
#         indent (GNU C beautifier)
#
#          -ts1: tab stop width = 1
#          -bli0: block indent ({}) 0
#          -c28: try to start code-line comments in column 28
#          -cd28: try to start declaration-line comments in column 28
#          -npcs: no space after function call names
#          -l72: maximum non-comment-line length
#          -lc72: maximum comment-line length
#
alias indent  'indent -ts1 -bli0 -c28 -cd28 -npcs -l72 -lc72'
```