Exam Exercises for Systems Programming

Antonio Carzaniga

Faculty of Informatics
USI
(Università della Svizzera italiana)

Edition 3.01
January 2026

WARNING: some solutions might be missing, and many may be incorrect! Please, consider contributing your solutions, including alternative solutions, and please report any error you might find to the author (Antonio Carzaniga <antonio.carzaniga@usi.ch>).

►**Exercise 1.** (20') Write a function with the following declaration:

```
int print_numbers(const char * input);
```

The function print_numbers must print on the standard output all the numbers in the input string, each on an individual line. A *number* is a contiguous sequence of digits (characters '0', ..., '9') possibly preceded by a sign (characters '−' or '+'). The function returns the total count of numbers printed. Write your implementation in a file called *print_numbers.c*.

►**Exercise 2.** The attached header file *bexp.h* (see next page) declares some types and functions intended to specify a system called *bexp* for the evaluation of expressions over binary numbers (i.e., base 2).

Question 1: (60') Implement the two functions `bexp_length` and `bexp_evaluate` declared and documented in *bexp.h*. Write your implementations in a file called *bexp.c*.

A solution is available here.

Question 2: (60') Write a program that uses the *bexp* system to implement a stack-based calculator for binary expressions. The calculator maintains a stack of binary values, as defined in *bexp.h*, and reads binary values and commands separated by space characters (space, new-line, etc.) from the standard input.

When the calculator reads a binary value (a string of `0` and `1` characters), it pushes that value onto the stack. When the calculator reads a command, it performs the action specified by the command, popping the parameters (if any) from the stack, and pushing the result (if any) onto the stack.

Valid commands include all binary and unary operators specified in *bexp.h*. For example, the command selected by the input string "`&`" corresponds to the binary operator `AND`, and therefore evaluates an `AND` expression between the top two values on the stack, and pushes the result back onto the stack.

If the stack does not contain enough parameters for the execution of a command, the program must print an error message.

In addition to the binary and unary operators, the calculator supports the following commands:

- `top` prints the value at the top of the stack without changing the stack.
- `pop` pops a value off the stack and prints it

Write your solution in a file called *bexp_calculator.c*.

Question 3: (60') Write a more compact variant of the *bexp* system in which values are represented as sequences of `int` values, each representing a block of bits, rather than sequences of characters. To do that, modify the header file *bexp.h* and the implementation file *bexp.c* so that this new variant can be selected by defining the pre-processing symbol `USING_COMPACT_VALUES`. Write your solution in two new files *bexp_compact.h* and *bexp_compact.c*.

File *bexp.h* [download this file]

```
#ifndef BEXP_H_INCLUDED
#define BEXP_H_INCLUDED

/* A binary value expressed as a sequence of '0' and '1' characters,
 * the sequence starts at the character pointed to by begin, and ends
 * at the character right before the one pointed to by end. Thus a
 * "null" value can be represented as an empty sequence where
 * begin==end. The first character, pointed to by begin, represents
 * the least-significant bit. All the bits following the most
 * significant one specified in the sequence are assumed to be 0. A
 * valid value must contain only '0' and '1' characters.
 */
typedef struct {
    char * begin;               /* pointer to least significant bit */
    char * end;                 /* pointer to one-past most significant bit */
} value;

/* Identifiers for the operators we can use on binary values
 */
typedef enum {
    /* Binary operators: */
    AND,                        /* & */
    OR,                         /* | */
    XOR,                        /* ^ */
    PLUS,                       /* + */
    GREATER_THAN,               /* > */
    LESS_THAN,                  /* < */
    /* Unary operators: */
    NOT,                        /* ! logical negation: value==0 => 1, otherwise 0 */
    COMPLEMENT,                 /* ~ complement: bitwise not */
} operator_t;

typedef struct {
    operator_t op;
    value value1;               /* first operand */
    value value2;               /* second operand,
                                   ignored if op is a unary operator. */
} expression;

/* Return the length (in characters) of the result of the given
 * expression, without necessarily computing that expression. This
 * can be used to allocate the space necessary to store the result.
 * Return -1 in case of error, for example if the given expression
 * contains an unknown operator.
 */
extern int bexp_length(const expression * ex);

/* Evaluates a given expression. The space in which to store the
 * result must be allocated by the caller. Thus result->begin must
 * point to the beginning of the allocated space while result->end
 * points to one-past the end. Return 1 upon success, or 0 upon
 * failure, for example if the given expression is not valid or if
 * the space provided by the caller is insufficient to store the
 * result. If the allocated space is larged than necessary, this
```

```
 * function must adjust the length of the result sequence.
 */
extern int bexp_evaluate(value * result, expression * ex);

#endif
```

►**Exercise 3.** (30') Write a module in a source file called *strstack.c* that implements a stack of strings (zero-terminated arrays of characters) as defined by the operations declared in the following header file *strstack.h*:

[download this file]

```
#ifndef STRSTACK_H_INCLUDED
#define STRSTACK_H_INCLUDED

#include <stddef.h>             /* declares size_t */

extern void strstack_use_buffer(char * mem, size_t mem_size);
extern void strstack_clear();
extern int strstack_push(const char * s);
extern const char * strstack_pop();

#endif
```

The *strstack* module must not allocate memory to hold the stack of strings. Instead, that memory is provided by the application through the strstack_use_buffer function. The module may of course use other internal meta-data variables.

The stack supports the following three operations:

- strstack_clear initializes the stack, resulting in an empty stack.
- strstack_push pushes a given string on the stack, returning 1 if successful, or 0 in case of overflow.
- strstack_pop pops a string from the stack, returning that string or 0 if the stack is empty. The returned string must be valid only until the next call to any one of the functions of the *strstack* module.

Use the strstack.tgz test package to test your solution.

A solution is available here.

►**Exercise 4.** (45') Write a module in a source file called *path.c* to handle paths (i.e., file names) in a Unix file system. The module must implement what you can think of as a *class* in object-oriented programming. The interface of this class is defined in the following header file *path.h*:

[download this file]

```
#ifndef PATH_H_INCLUDED
#define PATH_H_INCLUDED

struct path;                    /* declaration of the path structure */

struct path * path_new(const char * name); /* constructor */
void path_destroy(struct path * path);  /* destructor */

const char * path_append(struct path * this, const char * name);
const char * path_remove(struct path * this);
const char * path_value(const struct path * this);

#endif
```

The module supports the construction of a path object with a given initial path, which might be a file name, a directory name, or a compound path. A path object must be properly deallocated using the destructor. The module uses the Unix directory separator '/'. The value of a path object can be accessed through the *value* operation.

The module supports an *append* operation that adds a file, directory, or path to an existing path. If successful, the append operation returns the resulting path. For example, starting from "/usr", one could append "bin" to obtain "/usr/bin", and then append "emacs" to obtain "/usr/bin/emacs".

The module also supports a *remove* operation that removes the last (rightmost) component of a path. If successful, the remove operation returns the resulting path. For example, starting from "/usr/bin/emacs", the resulting path would be "/usr/bin".

The module must ensure that a path value is canonical form. A path in canonical form does not contain two consecutive directory separators and does not end with a directory separator. So, for example "/usr/bin" is a canonical path value, but "/usr/bin/" and "/usr//bin" are not. The user may construct or append paths that are not in canonical form, but every path returned by the module must be in canonical form.

Paths must have a fixed maximal length defined by the PATH_MAX macro defined in the *limits.h* header. A append operation that would result in a path exceeding the maximal length must fail by returning a null pointer.

A remove operation that would result in an empty path must fail by returning a null pointer.

The *test_path.c* program you find on-line is a test for the path module. Use it to test your implementation. In order to get full credit for this exercise, your code must pass the test.

►**Exercise 5.** (45') Write a program *file_usage* (source file *file_usage.c*) that prints the total size in bytes of all the files in a given directory that have a given extension (suffix). Notice that your program must consider only *regular files* in the given directory, not other directories or other types of files.

You may use the features and functions of the POSIX library of your system. In particular, you should use the opendir (and then closedir) and readdir functions declared in the *dirent.h* header to read and scan a directory. Also, in order to find whether a directory entry is a regular file, and then to find its size, you should use the stat function.

The documentation you should consult is available through the manual pages of your system through the *man* command. Specifically, read the manual pages for opendir, which also covers readdir and closedir, and stat.

►**Exercise 6.** (60') Write a variant of program *file_usage* described in Exercise 5 that prints the total size in bytes of all the files with the given extension (suffix) in the given directory and all its subdirectories. Write this variant in a source file called *file_usage2.c*. This program must have at most one directory open at any given time. This means that your solution should not use recursion.

Hint: use the *strstack* and *path* modules you developed in exercises 3 and 4.

A solution is available here.

►**Exercise 7.** (40') Write a module in a source file called *strqueue.c* that implements a queue of strings (zero-terminated arrays of characters) as defined by the operations declared in the following header file *strqueue.h*:

[download this file]

```
#ifndef STRQUEUE_H_INCLUDED
#define STRQUEUE_H_INCLUDED

#include <stddef.h>              /* declares size_t */

extern void strqueue_use_buffer(char * mem, size_t mem_size);
extern void strqueue_clear();
extern int strqueue_enqueue(const char * s);
extern const char * strqueue_dequeue();

#endif
```

The *strqueue* module must not allocate memory to hold the queue of strings. Instead, that memory is provided by the application through the strqueue_use_buffer function. The module may of course use other internal meta-data variables.

The queue supports the following three operations:

- strqueue_clear initializes the queue, resulting in an empty queue.
- strqueue_enqueue adds a given string to the queue, returning 1 if successful, or 0 in case there is no space to fit that string in the queue. Notice that it would be incorrect to return 0 when enough space is available.
- strqueue_dequeue extracts and returns the first string from the queue, returning that string (char pointer) or 0 if the queue is empty. The returned string must be valid only until the next call to any one of the functions of the *strqueue* module.

The *test_strqueue.c* program you find on-line system is a test for the *strqueue* module. Use it to test your implementation. In order to get full credit for this exercise, your code must pass the test.

Use the strqueue.tgz test package to test your solution.

A solution is available here.

►**Exercise 8.** (60') Write a program called *print_parts* in a source file called *print_parts.c*. The program must support the following command-line options:

`print_parts` ⟨ `-b` *output-spec* | `-f` *output-spec* [`-d` *delimiters*] ⟩ [*filenames...*]

The program prints selected parts of lines from each input file to standard output. If no input file is given, then the program uses the standard input.

The program may selects *bytes* or *fields* as the parts to be printed, with the `-b` and `-f` command-line options, respectively. A *field* is a portion of a line separated by one or more of the delimiter characters. The default delimiter characters are spaces, as defined by the `isspace` function from the standard C library. Alternatively, the user may specify a set of delimiter characters using the `-d` option.

An output specification (*output-spec* in the command-line above) is a string with the following possible formats and semantics

- n : print the n-th part
- n- : print all the parts from the n-th till the end of the line
- -n : print all the parts from the beginning of the line up to the n-th part
- n_1-n_2 : print all the parts between n_1 and n_2

A solution is available here.

►**Exercise 9.** (20') Write a variant of the *print_parts* program, called *print_parts2*, that accepts more expressive output specifications consisting of a comma-separated list of pairs: file name and part specification:

filename<*parts*[,*filename*<*parts*[...]]

Where each part specification has the same format and semantics of the output specification described above for Exercise 2. Each pair specifies that the given parts must be printed onto the given file.

Example 1:

Imagine a file called *courses.txt* containing lines like the following:

```
Algorithms and Data Structures,Luigi,Samuel,Francesco
Systems Programming,Andrea,Carlo
```

...

Then, the following shell command:

```
./print_parts2 -d , -f courses.txt\<1,students.txt\<2- courses.txt
```

would write a list of course names in a file called *courses.txt* and a list of students in a file called *students.txt*.

Example 2:

The following shell command:

```
./print_parts2 -b one.txt\<1-,two.txt\<1-,three.txt\<1- < /etc/passwd!
```

makes three copies of the file passed as standard input (*/etc/passwd*) called *one.txt*, *two.txt*, and *three.txt*, respectively.

►**Exercise 10.** (30') In a source file *word_reader.c* write a function called read_next_word that reads words out of a file (stream) as declared in the following header file *word_reader.h*:

[download this file]

```
#ifndef WORD_READER_INCLUDED
#define WORD_READER_INCLUDED

#include <stdio.h>

#define WORD_MAX_LEN 1024

extern const char * read_next_word(FILE *);

#endif
```

A word is a contiguous sequence of alphabetic characters as defined by the isalpha function from the C standard library declared in *ctype.h*. The read_next_word function must return the next word in the given file. In particular, read_next_word must return a pointer to a storage allocated and owned by read_next_word, or the null pointer if there are no more words in the file.

A solution is available here.

►**Exercise 11.** (60') Write in a source file called *map.c* a module that implements a map data structure as defined by the declarations in the following header file *map.h*:

[download this file]

```
#ifndef MAP_INCLUDED
#define MAP_INCLUDED

extern int map_init();          /* initializes  the map */
extern void map_shutdown(); /* destroys the map */

extern int map_clear();         /* removes all elements */

extern int map_put(const char * key, void * value); /* associates key with value */
extern void * map_get(const char * key);        /* returns the value associated with key */

extern void map_start_iteration();
extern int map_iterate(const char ** key, void ** value);

#endif
```

These functions must behave as follows:

- map_init and map_shutdown initialize and destroy the map. map_init returns a non-zero value (true) on success, or 0 (false) on error.
- map_clear removes all elements from the map, and also returns a non-zero value (true) on success, or 0 (false) on error.
- map_put associates the given string key with the given value, which is a generic pointer to void. If the same key was already associates with another value, then map_put resets the mapping with the new given value. map_put returns a non-zero value (true) on success, or 0 (false) on error.
- map_get returns the value associated with the given string key, or null if no such key exists in the map.
- map_start_iteration starts an iteration in which all the keys and values in the map can be obtained through map_iterate. In particular, for each mapping map_iterate returns a non-zero value (true) and assigns the key and value referenced by the given pointers, or it returns zero (false) if there are no more mappings. The following code exemplifies a typical iteration pattern:

  [download this file]

  ```
  const char * key;
  void * value;

  map_start_iteration();
  while (map_iterate(&key, &value)) {
      /* do something for this key--value mapping */
  }
  ```

You may either implement your own map module from scratch, or you may complete the partial implementation available with the exam material on-line. If you decide to implement your own module from scratch, then consider using a simple data structure, such as an array or a linked list of elements, each representing a mapping.

A solution is available here.

►**Exercise 12.** (30') Write a program, in a source file *wf.c*, that uses the *read_word* and *map* modules of exercises 10 and 11 to read text files, and for each text file writes a corresponding word-frequency file. For each word in a text file, the corresponding word-frequency file must contain one line with that word followed by the number of occurrences of the word in the text file. The program takes the file names from its command-line parameters, and for each file name f outputs the corresponding word-frequency file named f*.wf* (that is, adding a suffix *.wf*). If no file names are given, then the program reads one text file from the standard input and writes the word frequencies on the standard output.

A solution is available here.

►**Exercise 13.** (40') Write a module that implements a compact set of integers. A set must be created with the cardinality n of its universe $U = \{0, 1, 2, \ldots, n-1\}$, and therefore must be able to represent all subsets of U. For example, `cset * s = cset_new(10)` creates an empty set, and `cset_insert(s,7)` adds element 7, therefore obtaining the set $\{7\}$.

The attached header file *cset.h* (below) defines the interface you must implement, including the complexity requirements for each method. The semantics of each method should be obvious from the method name and parameters. The semantics of `cset_union(a,b)` is $a \leftarrow a \cup b$, and similarly for `cset_intersection(a,b)`. Notice that the definitions in *cset.h* are not merely suggestions: they are *requirements* that your implementation must satisfy precisely. You must write your implementation in a source file called *cset.c* in which you may also define and use other structures.

You should use the test provided on-line (`test_cset`) to test your implementation.

Design hints: use a bitvector or another form of direct-address table. A bitvector represents each element of the universe $\{0, \ldots, n-1\}$ as a bit in one or more integer objects, where the value of the i-th bit tells whether element i is in the set. Individual bits or blocks of bits in a bit vector can be tested or set to 1 or 0 using bitwise operators. Although these operators work on both signed and unsigned integers, their semantics is clearer and less problematic on unsigned integers. For example, a variable `unsigned int b` can typically represent sets from a universe of $n = 32$ elements, and `b |= (1U << 7)` sets the 7th bit in `b`. This assumes that the bit width of an `unsigned int` is 32 on your platform. However, your implementation must be portable to other platforms (see the documentation of the *limits.h* header).

File *cset.h*: [download this file]

```
#ifndef CSET_H_INCLUDED
#define CSET_H_INCLUDED

/* An extremely compact dynamic set capable of representing sets out
 * of the universe of integers {0,1,2,..., n-1}. n is a parameter of
 * the constructor (see below).
 */
struct cset;
typedef struct cset cset;

extern cset * cset_new(unsigned int n);                          /* complexity: O(n) */
extern cset * cset_new_copy(const cset * s);                     /* complexity: O(n) */
extern void cset_destroy(cset * s);                              /* complexity: O(1) */

extern void cset_insert(cset * s, unsigned int elem);            /* complexity: O(1) */
extern void cset_delete(cset * s, unsigned int elem);            /* complexity: O(1) */
extern int cset_find(const cset * s, unsigned int elem);         /* complexity: O(1) */

extern void cset_union(cset * a, const cset * b);                /* complexity: O(n) */
extern void cset_intersection(cset * a, const cset * b);         /* complexity: O(n) */

extern int cset_is_empty(const cset * s);                        /* complexity: O(n) */
extern int cset_is_equal(const cset * a, const cset * b);        /* complexity: O(n) */

#endif
```

A solution is available here.

►**Exercise 14.** (80') Write an accounting module within a group trip manager application. The module must implement a simple accounting systems for the expenses shared by the participants. The system must be first initialized with the names of the participants. Once initialized, the system is used to record shared expenses and the corresponding payments by individuals. At any given time, one can query the system to print an expense report, as well as the payment balance for each participant. Below are the API functions you must implement (declared in the *trip.h* header file attached below). Write a source file called *trip.c*.

`void trip_initialize ()`
Initializes the accounting module, clearing the list of participants and all the expenses.

`void trip_shutdown()`
Deallocates all memory structures managed by the accounting module.

`int trip_add_person(const char * name)`
Registers a Participant in the trip. If successful, the return value is a non-negative, progressive numeric identifier for the new participant. Identifiers are progressive in the sense that they start at 0 and then increase by one for each successfully-registered participant. If an error occurs, for example if there is no more memory to register the new participant, then the result is a negative number.

`int trip_find_person(const char * name)`
Returns the numeric identifier of a previously registered participant. If the given name does not correspond to any previously registered participant, then the result must be negative.

`int trip_add_expense(const char * description, float amount, int payer)`
Records an expense with the given description, amount payed, and the identifier of the payer. The result is 1 if the expense was successfully registered, or 0 upon error. The expense is assumed to be attributable in equal parts to all the registered participants.

Optional alternative: (20') you may implement a more sophisticated variant of this function:

`int trip_add_expense(const char * dscr, float amt, int pyr, const cset * part)`
Records an expense attributable in equal parts to a given set of participants. This variant must be selected by the application using your module by defining the preprocessing macro `WITH_DETAILED_EXPENSE_ATTRIBUTION`.

`void trip_print_expenses()`
Prints on the standard output a well formatted list of all the expenses. For example,

```
     Amount | Description                                | Payer
------------+--------------------------------------------+----------------
     152.50 | Grocery shopping at supermarket            | Antonio
      56.00 | Art museum visit                           | Daniele
     120.00 | Dinner at Mexican restaurant               | Antonio
------------+--------------------------------------------+----------------
```

`void trip_print_balance()` prints on the standard output the balance owed by, or owed to each participant. For example, referring to the expenses above and assuming there is another participant named Aida, the balance would be:

```
Participant     |      Gives |   Receives
----------------+------------+------------
Antonio         |            |     163.00
Aida            |     109.50 |
Daniele         |      53.50 |
----------------+------------+------------
```

File *trip.h*: [download this file]

```
#ifndef TRIP_H_INCLUDED
#define TRIP_H_INCLUDED

#ifdef WITH_DETAILED_EXPENSE_ATTRIBUTION
#include "cset.h"
#endif

/*  A system to manage shared expenses on a trip.
 */

extern void trip_initialize();
extern void trip_shutdown();

extern int trip_add_person(const char * name);
extern int trip_find_person(const char * name);

#ifdef WITH_DETAILED_EXPENSE_ATTRIBUTION
extern int trip_add_expense(const char * descr, float amount, int payer,
                            const cset * participants);
#else
extern int trip_add_expense(const char * descr, float amount, int payer);
#endif

extern void trip_print_expenses();
extern void trip_print_balance();

#endif
```

A solution is available here.

►**Exercise 15.** (60') Write a spell-checking program called *spellcheck*. The program reads an input text file and a dictionary file, and must output a list of words that are found in the input text but not in the dictionary. Words are contiguous sequences of alphabetic characters possibly also containing the apostrophe character ('). So for example, "would've" and "mama's" should be considered as valid words. You may assume that a word in the input text is never longer than 50 characters.

The program may take any combination of the following command-line parameters:

- `in=`*filename* instructs the program to read its input from the given file. If this option is not used, then the program must read from the standard input.
- `out=`*filename* instructs the write its output to the given file. If this option is not used, then the program must write to the standard output.
- `dict=`*filename* instructs the program to read the dictionary from the given file. If this option is not used, then the program must read the dictionary from file */usr/share/dict/words*. The dictionary file contains exactly one word per line, and is sorted in lexicographical order.

The program must be efficient in terms of execution time, as well as in memory usage. It must also correctly deallocate all the memory it allocates, close all the files it opens, and correctly deal with all error conditions.

Design hints: read the whole dictionary directly into memory and use a simple linear index (an array) for the words. Since the input file is already sorted, this would allow a simple and efficient binary search for the dictionary search.

A solution is available here.

►**Exercise 16.** (60') Write a text formatting library called *page* that formats pages of text, and then use the library to implement a program called *makepages* that reads a loosely formatted text file, and outputs it in fixed-width paragraphs and pages.

The formatting library allows a client application to define a page layout in terms of page width and height (number of characters). It then allows the application to incrementally add words and also insert paragraph breaks. Whenever a page is complete, the library prints it out followed by the "form feed" character ('\f'). See the attached header file (below) for a complete specification of the API of the page formatting library.

makepages reads an input containing words and paragraph breaks. A word is anything separated by "space" characters, including the space character itself (' '), as well as tab ('\t'), new line ('\n'), and carriage return ('\r'). Notice that a word in this case may consist of punctuation characters, as well as other graphical characters. A paragraph break is a line that contains no words (an empty line or a line containing only space characters).

The program may take any combination of the following command-line parameters:

- `W=`*text-width* instructs the program to format the output in pages of the given width. The default width is 40.
- `H=`*text-height* instructs the program to format the output in pages of the given height. The default height is 50.

The input file is made of lines of at most 1000 characters.

Implementation hints: you might want to read the input file line by line, and then use the `strtok` function to break a line into words.

File *page.h*: [download this file]

```
#ifndef PAGE_H_INCLUDED
#define PAGE_H_INCLUDED

/*** A "static" page formatting library. ***/

/* Configure the output stream used by the formatting library. The
 * default output stream stdout. The library does not open or close
 * streams, which remains the responsibility of the client
 * application.
 */
extern void page_set_output(FILE * output);

/* Configure the width of the page (number of characters. The
 * default value is 50.
 */
extern void page_set_width(unsigned int w);

/* Configure the width of the page (number of characters. The
 * default value is 50.
 */
extern void page_set_height(unsigned int h);

/* Initialize all internal data structure, allocating memory if
 * necessary. This method must be called before the page is used for
 * formatting. Return 1 when successful, or 0 in case some error
 * occurs, for example if a memory allocation fails.
 */
extern int page_initialize();
```

```
/* Clear all internal data structure, deallocating memory if
 * necessary.
 */
extern void page_clear();

/* Adds a word to the text of the page. This might cause the page to
 * fill completely, and therefore would cause the formatting library
 * to output the whole formatted page.
 */
extern void page_add_word(const char * word);

/* Adds a paragraph break to the text of the page. A paragraph break
 * must appear as an empty line. This might cause the page to fill
 * completely, and therefore would cause the formatting library to
 * output the whole formatted page.
 */
extern void page_paragraph_break();

/* Explicitly prints the current content of the formatted page.
 */
extern void page_print();

#endif
```

A solution is available here: page.c, `makepages.c`.

►**Exercise 17.** (30') Write a variant of the *makepages* program of Exercise 16 that also supports formatting of pages in multiple columns. The columns must be separated horizontally by 2 spaces. This variant should accept the command-line parameters

- `W=`*column-width* instructs the program to format the output in columns of the given width. The default width is 30.
- `C=`*columns* instructs the program to format the output in C columns per page. The default is $C = 2$.
- `H=`*text-height* instructs the program to format the output in pages of the given height. The default height is 50.

A solution is available here: `page2.c`, `makepages2.c`.

►**Exercise 18.** (60') Write three functions in a source file called *format.c* to format a line of ASCII text as defined by the header file below.

File *format.h*: [download this file]

```
#ifndef FORMAT_H_INCLUDED
#define FORMAT_H_INCLUDED

extern void format_flush_left(char * begin, char * end);
extern void format_flush_right(char * begin, char * end);
extern void format_justified(char * begin, char * end);

#endif
```

All three functions take an input line starting at begin and delimited by end (pointer to the first character past the end of the line). All three functions must change the input line directly and within its bounds. The output for each function is as follows.

- format_flush_left changes the input line so as to push all words as far as possible towards the left. So, the first word or punctuation character starts at the beginning of the line, all words are separated by a single space character, and all the characters to the right of the last word, if any, are space characters.

  For example, the input line:

  `'   It ain't over    'til it's over.  '`

  should be changed as follows:

  `'It ain't over 'til it's over.        '`

- format_flush_right changes the input line so as to push all words as far as possible towards the right. So, the last word or punctuation character is at the end of the line, all words are separated by a single space character, and all the remaining characters to the left of the last word, if any, are space characters.

  For example, the input line:

  `'   It ain't over    'til it's over.  '`

  should be changed as follows:

  `'        It ain't over 'til it's over.'`

- format_justified changes the input line so as to justify the text. This means that the first character of the first word (or punctuation mark) is the first character of the line and the last character of the last word (or punctuation mark) is the last character of the line, and words are uniformly spaced, meaning that the space between words is either the same or differs by at most one space in some cases. If there is a single word on the line, then the word must be flush-left.

  For example, the input line:

  `'   It ain't over    'til it's over.  '`

  should be changed as follows:

  `'It   ain't  over  'til   it's   over.'`

A solution is available here.

►**Exercise 19.** (60') Write a program in a source file called *ta.c* to assign teaching assistants to courses. The program takes two command-line parameters. The first parameter is the name of a courses file containing a list of courses each with the required number of teaching assistants. The second parameter is the name of a people file containing a list of persons each with their availability to serve as teaching assistant.

More specifically, each line of the courses file has the following comma-separated fields:

course title, *semester*, *number of assistants*

where *course title* is the title of the course (e.g., "Algorithms and Data Structures"), *semester* is a character, either `F` or `S`, indicating that the course is scheduled in the fall or spring semester, respectively. *number of assistants* is an integer representing the number of assistants that are required for the course.

Each line of the people file has the following comma-separated fields:

name, *fall availability*, *spring availability*

where *name* is the name of the teaching assistant, and *fall availability* and *spring availability* are two integers indicating how many courses the person is willing to assist in the fall and spring semesters, respectively.

With these two files, the program must assign assistants to courses, making sure to respect people's availability. The program must then print a list of courses. For each course, the program must print the course title followed by a list of assistants. If the program could not assign the required number of assistants for a course, the program must also output a warning by printing the word "(INCOMPLETE)" next to the course title.

You may assume that course titles are no more than 200 characters in length, and that people's names are no more than 50 characters. You may not assume any limit on the number of courses or the number of people.

Example: with the following input files:

courses

```
Algorithms and Data Structures,S,2
Systems Programming,S,2
Programming Fundamentals I,F,4
```

people

```
Daniele,1,1
Antonio,1,1
Robert,2,1
Luis,2,0
```

the program should output something like this:

```
Algorithms and Data Structures
Daniele, Antonio

Systems Programming (INCOMPLETE)
Robert

Programming Fundamentals I
Daniele, Antonio, Robert, Luis
```

A solution is available here.

►**Exercise 20.** (60') Write a module that implements a formatting filter for text records. In essence, an application uses a filter by first passing a string of characters representing a *record*, and then by producing an output of that input record according to a format string. A text record consists of one or more textual *fields* separated by a separator string. The module must implement filter *objects* as specified by the attached header file *rec_io.h* (below). An application might create multiple filter objects, which may be used at the same time, and that must be independent of each other.

The *test_rec_io1.c* program you find on-line is a test for the rec_io module. Use it to test your implementation.

File *rec_io.h*: [download this file]

```
#ifndef REC_IO_H_INCLUDED
#define REC_IO_H_INCLUDED

#include <stddef.h>

struct rec_io;

/* This is the maximal size for an input record. The module must
 * support input records up to this size.
 */
#define MAX_RECORD_LENGTH 1000

/* This is the maximal size for a record separator. The module must
 * support separators up to this size.
 */
#define MAX_SEPARATOR_LENGTH 10

/* Constructor
 *
 * Returns a pointer to a valid rec_io object, or NULL in case of
 * error (e.g., insufficient memory)
 */
extern struct rec_io * rec_io_new();

/* Destructor
 */
extern void rec_io_destroy(struct rec_io *);

/* Configure this rec_io filter to use the separator string defined by
 * the begin and end pointers. Notice that a separator can be any
 * sequence of bytes (at most MAX_SEPARATOR_LENGTH). In particular, a
 * separator may contain the NULL character ('\0', one or more).
 *
 * The default separator is the sequence consisting of a single space
 * character. Setting a separator determines how the filter reads any
 * successive input record (with rec_io_read_record).
 */
extern void rec_io_set_separator(struct rec_io * this, const char * begin, const char * end);

/* Passes an input record to this filter for later processing by the
 * filter . An input record is any sequence of characters (at most
 * MAX_RECORD_LENGTH) composed of fields separaterd by separator
 * strings. The beginning of the sequence and the end of the sequence
 * are implicit separators, so an input sequence that does not contain
```

```
 * the separator consists of exactly one field equal to the entire
 * sequence. Fields may be zero–length sequences.
 *
 * This method must return the number of fields read.
 */
extern int rec_io_read_record(struct rec_io * this, const char * begin, const char * end);

/* Outputs the input record previously read with rec_io_read_record
 * using the given format string. The output is given as a sequence
 * of characters written in the output buffer provided by the
 * application.  This method may not write more than maxlen characters
 * into the output buffer. The return value is the actual number of
 * characters written into the output.
 *
 * The format string is a C string, meaning it is terminated by '\0'.
 * The format string may contain field indicator consisting of a
 * single percent character ('%') followed by a decimal number.
 * Fields are numbered starting from 0. So, for example, the format
 * string "%0" should output the first fields. A field indicator that
 * points to a field that was not read must have no output (for
 * example, "%4" when only four or less fields were read).
 *
 * Any other character or sequence of characters in the format string
 * must be copied identically into the output. So, for example, the
 * format string "1: %0\n2: %1\n" should output two lines containing
 * the first and second field, respectively.
 *
 * This method can be called multiple times for the same record,
 * possibly with different format strings.
 */
extern size_t rec_io_write_record(struct rec_io * this,
                                  char * out, size_t maxlen, const char * format);

#endif
```

A solution is available here. A test package is also available here.

►**Exercise 21.** (60') Write a program called *otp* to encrypt files using a one-time pad. A one-time pad is an extremely simple and also very secure encryption scheme that works as follows: each byte x_i of the cleartext file (input) produces a byte $y_i = x_i \oplus k_i$ in the ciphertext file (output) using one byte k_i of the key, where $\oplus$ is the xor operator.

A *one-time* pad is such that a key byte is used only once. This means that you need large keys, and that you need to remember which bytes you used of those keys. So, your encryption program must use a key file f, plus an auxiliary file named f*.counter* that stores the number of bytes already used in f.

The *otp* program must accept the following optional command-line options:

- `in=`*input-file* defines the input cleartext file. By default, the program must read the cleartext file from the standard input (stdin).
- `out=`*output-file* defines the output ciphertext file. By default, the program must write the cleartext file onto the standard output (stdout).
- `key=`*key-file* defines the output ciphertext file. By default, the program must use a key file called *key*. This also determines the name of the counter file to associate with a particular key file. The name of the counter file is then *key-file.counter*. If a counter file does not yet exist, the counter is assumed to be 0.

For example, imagine that you and your friend share a large secret key (say 10MB) that you store in a file called *secretkey*. This is the first time you use this key file, and you start by encrypting a message with the following command:

```
./otp in=private_message1.txt out=encrypted_message1 key=secretkey!
```

This creates a file called *encrypted_message1* and also a file called *secretkey.couter* that contains the number n of characters used from the beginning of the key file, which is also exactly the number of characters in *private_message1.txt* and in *encrypted_message1*. The number n is stored in the counter file in decimal characters.

Then you use again the same key file to encrypt another message:

```
./otp in=message2.txt out=message2 key=secretkey!
```

This second command reads the number from the counter file, and then starts encrypting the second message with the $(n+1)$-th character in the key file, and then updates the counter file.

Hint: in your implementation, you might want to use the following functions from the standard library: fopen, fread, fwrite, fseek, and of course fprintf, etc. Notice that you can open the key counter file in read/write mode ("r+"), and after reading the current counter you can use rewind to go back to the beginning of that file to write the new value of the counter.

A solution is available here.

►**Exercise 22.** (10') In a source file called *avg.c* write a function declared as follows:

```
float average_positives(const int * begin, const int * end);
```

This function takes a sequence of numbers, passed as a pointer to the first element (begin) and a pointer to one-past the last element (end), and returns the average of all the positive numbers in the sequence. When there are no positive numbers in the sequence, the function must return -1. For example, the result for the $(1, -2, 3, 0)$ is 2.0.

The *test_avg.c* program you find on-line is a test for the average_positives function. Use it to test your implementation.

A solution is available here.

►**Exercise 23.** (20') In a source file called *fields.c* fields write a function declared as follows:

```
int equal_fields(const char * begin, const char * end, char separator);
```

This function takes a text defined by the begin and end pointers, and a separator character. The function then interprets the text as a sequence of text fields separated by the given separator character, and returns 1 if there are at least two equal fields, or 0 otherwise. For example, the result with text "ciao,miao,bao" and separator ',' is 0, while with "ciao mamma" and separator 'm', the result is 1. Notice in fact that the beginning and end of the sequence are implicit separators.

The *test_fields.c* program you find on-line is a test for the equal_fields function. Use it to test your implementation.

A solution is available here.

►**Exercise 24.** (90') Implement the abstract class pixmap and two of its concrete sub-classes ascii_pixmap and packed_pixmap defined by the header files below. Write the private header file and implementation of the pixmap class in files named *pixmap_impl.h* and *pixmap_impl.c*, respectively. Also, write the implementation of the ascii_pixmap and packed_pixmap classes in files named *ascii_pixmap.c* and *packed_pixmap.c*, respectively.

The *test_pixmap.c* program you find here is a test for the pixmap class. Use it to test your implementation.

File *pixmap.h*: [download this file]

```
#ifndef PIXMAP_H_INCLUDED
#define PIXMAP_H_INCLUDED

#include <stddef.h> /* for size_t */

/* An abstract 2D pixel map.
 *
 * A 2D pixel matrix. Each pixel is identified with an x,y coordinate
 * pair.  Coordinates are greater or equal to 0. Pixels have binary
 * values: they can be set to a foreground value or to a background
 * value. A concrete implementation of this abstract class will
 * implement the pixel matrix and all the virtual methods defined below.
 */
struct pixmap;

/* Clear the whole pixmap: Every pixel is set to the background value. */
extern void pixmap_clear(struct pixmap * this);

/* Set the value of a pixel in the pixmap.
 *
 * A value != 0 sets the pixel (foreground); a value equals to 0
 * resets the pixel (background). If the given coordinates are
 * outside the dimensions of the given pixmap, this function must have
 * no effect. */
extern void pixmap_set_pixel(struct pixmap * this, unsigned int x, unsigned int y, int value);

/* Get the value of a pixel.  zero means background; non-zero means
 * foreground. If the given coordinates are outside the dimensions of
 * the given pixmap, this function returns 0. */
extern int pixmap_get_pixel(const struct pixmap * this, unsigned int x, unsigned int y);

/* Width of the pixmap. */
extern unsigned int pixmap_get_width(const struct pixmap * this);

/* Height of the pixmap. */
extern unsigned int pixmap_get_height(const struct pixmap * this);

/* Memory size of the implementation of this pixmap. */
extern size_t pixmap_get_memory_size(const struct pixmap * this);

/* Destroy a pixmap.
 */
extern void pixmap_destroy(struct pixmap * this);

#endif
```

File *ascii_pixmap.h*: [download this file]

```
#ifndef ASCII_PIXMAP_H_INCLUDED
#define ASCII_PIXMAP_H_INCLUDED

#include "pixmap.h"

/* A pixmap backed by a matrix of ASCII characters.
 *
 * Each bit corresponds to a character. The background character is
 * the space character (' '); the foreground character is '*'. These
 * are the default values, which may be changed by the user.
 *
 * The pixels are laid out line-by-line. So, the memory buffer starts
 * with the character corresponding to the pixel at position x=0, y=0,
 * followed by the pixel at position x=1, y=0, up to x=width-1, y=0,
 * which is followed by the pixel at position x=0, y=1, and so on.
 */
struct ascii_pixmap;

/* Create a new ascii pixmap backed by the given memory buffer.
 *
 * width is the number of characters in each line; height is the
 * number of lines; map is a pointer to a memory region used to store
 * the pixels.  This region is passed and owned by the user. It is
 * also assumed to be large enough to hold width*height characters.
 */
extern struct pixmap * ascii_pixmap_new(unsigned int width, unsigned int height, char * map);

/* Set the background character for the given ascii pixmap. */
extern void ascii_pixmap_set_background(struct pixmap * this, char bg);

/* Set the foreground character for the given ascii pixmap. */
extern void ascii_pixmap_set_foreground(struct pixmap * this, char fg);

/* Get the background character for the given ascii pixmap. */
extern char ascii_pixmap_get_background(const struct pixmap * this);

/* Get the foreground character for the given ascii pixmap. */
extern char ascii_pixmap_get_foreground(const struct pixmap * this);

#endif
```

File *packed_pixmap.h*: [download this file]

```
#ifndef PACKED_PIXMAP_H_INCLUDED
#define PACKED_PIXMAP_H_INCLUDED

#include <stdint.h>

#include "pixmap.h"

/* A pixmap backed by a packed bit map in memory.
 *
 * A packed bit map is a packed array of bytes (uint8_t) in which each
 * pixel of the image corresponds to a single bit of memory.
 *
 * The pixels are laid out as follows: each line corresponds to a
 * packed sequence of bytes. Let L be a pointer to the first byte
 * representing a line, then the pixel in position x=0 on that line is
 * the least significant bit in the first byte L[0], the pixel in
 * position x=7 corresponds to the most significant in L[0], the pixel
 * in position x=8 then corresponds to the least significant bit in
 * L[1], and so on.
 *
 * If the width of the packed_pixmap image is not a multiple of 8, then some
 * of the bits in the last byte of each line will be ignored.
 *
 * Lines are packed one after the other, with the first line
 * corresponding to the y=0 coordinate.
 *
 * HINT: you can set a bit in position i (i < 8) to 1 in a uint8_t
 * object x with the following code:
 *
 * uint8_t mask = 1; mask <<= i; x |= mask;
 *
 * or you can set it to 0 with the following code:
 *
 * uint8_t mask = 1; mask <<= i; x &= ~mask;
 */

struct packed_pixmap;

/* Create a new packed pixmap backed by the given memory buffer.
 *
 * w is the number of characters in each line; h is the number of
 * lines; mem is a pointer to a memory region used to store the
 * pixels. This region is passed and owned by the user. It is also
 * assumed to be large enough to hold the packed pixel map.
 */
extern struct pixmap * packed_pixmap_new(unsigned int w, unsigned int h, uint8_t * mem);

#endif
```

A solution is available here: `ascii_pixmap.c`, `packed_pixmap.c`.

►**Exercise 25.** (40') In a source file called *comments.c*, write a program that prints the amount of pure comment lines in one or more source files. A *pure comment line* is a line that begins with zero or more space characters followed by a special comment string. The standard library function `isspace()` determines whether a character is a space.

The program takes zero or more command-line parameters. A parameter `comment=`*string* defines the comment string. The default comment string is "//". Any other parameter is interpreted as the name of a file to process. If no file names are specified, the program must process its standard input.

For each input file, the program must print (on standard output) a single line containing the name of the file or `(stdin)` if the program reads from the standard input, then a TAB character ('\t'), then the number of pure comment lines in the file, then another TAB character, and then the total number of lines in the file.

You may assume that the special comment string does not contain any space character. You may also assume that input lines are at most 1000-character long.

You may use the *test_comments.sh* script you find on-line as a test for the *comments* program.

A solution is available here.

►**Exercise 26.** (80') In a source file called *sort_cars.c*, write a program that reads and sorts a file of records representing cars. Each record specifies a maker, a model name, engine power, and price. Your program must read the input records from the standard input, and print the sorted records on the standard output. The input and output records are read and written as a single line with the fields separated by a space character. You may assume that the input format is correct. Maker and model names are strings of up to 20 characters each. Power and price are non-negative integers.

Use the standard library function `qsort` to perform the sorting.

The program takes zero or one command-line parameter that determines the sorting criterion as follows:

- `-model` sorts in decreasing alphabetical order by model name
- `+model` sorts in increasing alphabetical order by model name
- `-maker` sorts in decreasing alphabetical order by maker name
- `+maker` sorts in increasing alphabetical order by maker name
- `-price` sorts in decreasing order by price
- `+price` sorts in increasing order by price
- `-power` sorts in decreasing order by power
- `+power` sorts in increasing order by power

The default sorting criterion is increasing alphabetical order by model name. Your program must be able to handle input files of any size using the proper dynamic memory allocation.

You may use the *test_sort_cars.sh* script you find on-line as a test for the *sort_cars* program.

A solution is available here.

►**Exercise 27.** (60') Write a program called *headlines* that reads a text document specially formatted into sections, and outputs the same document with section numbers. Your program must read the input document from the standard input. The document is structured in sections and subsections up to 8 levels of sub-sectioning depth. Each section begins with a headline, which is a line of text beginning with one or more sectioning characters followed by the section title. By default, the sectioning character is '*'. The text following the headline is the text of the section. For example, the following text is a document with a level-1 section entitled "Introduction".

```
* Introduction
This is the text in the Introduction section...
```

You must allow the user to change the sectioning character using a command-line option '-c' followed by the sectioning character. For example, if invoked with '-c +', your program should read the following text as a document with three sections

```
+ Introduction
We changed the sectioning character, so the following line is
*not* a section headline.
+ Related Work
While this is another section, and now we even get a subsection.
++ Old Stuff
A bunch of old stuff...
++ Recent Stuff
+++ Very Recent Stuff
++ New Stuff
```

Your program must output the input document with section titles introduced by an empty line and with numbers instead of the sectioning characters. A section number is a sequence of numbers for each level up to the level of the current section. The numbers for each level must be separated by a period ('.') without spaces, and the whole section number must be terminated by a close parenthesis (')'). For example, first example above should be output as follows:

```
1) Introduction
This is the text in the Introduction section...
```

While the second example above, with the sectioning character changed with -c +, should be output as follows:

```
1) Introduction
We changed the sectioning character, so the following line is
*not* a section headline.

2) Related Work
While this is another section, and now we even get a subsection.

2.1) Old Stuff
A bunch of old stuff...

2.2) Recent Stuff

2.2.1) Very Recent Stuff

2.3) New Stuff
```

Your program must also support an outline mode selected with the '-o' command-line option. In outline mode, the program must only output the section headlines, omitting all the section text. Again using the first example above, the output in outline mode should be as follows:

```
1) Introduction
```

And the output for the second example, in outline mode, should be as follows:

```
1) Introduction

2) Related Work

2.1) Old Stuff

2.2) Recent Stuff

2.2.1) Very Recent Stuff

2.3) New Stuff
```

Notice that your program might have to define section numbers for sections and subsections that do not exist. For those, the default section number is 0. For example, the following document

```
* Introduction
* Second Section
*** A Sub-Sub-Section Without a Parent Sub-Section
** Here is the Parent SubSection
*** Another Sub-Sub-Section
```

Should produce the following output:

```
1) Introduction

2) Second Section

2.0.1) A Sub-Sub-Section Without a Parent Sub-Section

2.1) Here is the Parent SubSection

2.1.1) Another Sub-Sub-Section
```

You may use the *test_headlines.sh* script you find on-line as a test for the *headlines* program.

A solution is available here. Another one is here.

►**Exercise 28.** (60') Write a library that implements a very simple database of named points, and that supports the computation the total traveled distance in a path connecting points. The library must implement the functions declared in the following header file

File *routing.h*: [download this file]

```
#ifndef ROUTING_H_INCLUDED
#define ROUTING_H_INCLUDED

/* clear the database of points and deallocate all objects within the
   library */
extern void routing_clear();

/* defines or changes the coordinates of a point */
extern int routing_set_point(const char * name, double x, double y);

/* compute the total distance traveled in the given path. A path is a
   comma-separated list of point names. The result must be -1 if any
   one of the points is undefined. */
extern double routing_total_distance(const char * path);

#endif
```

A point is defined by a name and a pair of Cartesian coordinates. The name is a normal C string (zero-terminated). A path is a sequence of one or more points defined by a C string. The format of the path string is that of a comma-separated list. For example, one might define three points with names "A", "XYZ", and "BB", and then compute the distance of the path "XYZ,A,BB,A". Notice that "BB,BB,A,A" is also a valid path, and its total distance is the same as that of path "A,BB". A single point is also a valid path of total distance zero.

You may use the *test_routing.c* program you find on-line as a test for the routing library. A makefile is also available on-line.

A solution is available here.

►**Exercise 29.** (120') Write a program called *keywords* in a single source file called *keywords.c* or *keywords.cc*. You may use either C or C++. The program extracts important words from text files. A *word* is a sequence of one or more consecutive alphabetic characters as defined by the `isalpha` function of the standard C library. The program must read one or more files, or the standard input, and for each file it must print on the standard output the words that occur within a range of frequencies $l \leq f_w \leq h$. The *frequency* f_w of a word w in a file is the number of times that w occurs in the file divided by the total number of words in that file. Frequencies are expressed in percentage values as integers between 0 and 100. The default values are $l = 10$ and $h = 90$, meaning that the program must output words with frequencies between 10% and 90%.

Output

The program must print its output on the standard output. There are two required output formats: *simple* and *fancy*. The simple output (default) is a list of words for each input file. The fancy output is a list of words with their frequencies. More specifically, with simple output, for each input file name x, the program must output one line as follows:

$x \quad w_1 \quad w_2 \ldots$

The output line starts with the file name x, or the empty string for the standard input, and for each output word w_i continues with a space followed by the word w_i.

With "fancy" output, for each input file name x, the program must output one line:

$x \quad c_1\texttt{:}w_1 \quad c_2\texttt{:}w_2 \ldots$

The output line starts with the file name x, or the empty string for the standard input, and for each output word w_i continues with a space followed by the number c_i of occurrences of word w_i, followed by a colon ('`:`'), followed by the word w_i.

For each input file, the output words must be sorted in descending order of frequency. If two words have the same frequency, the order must be lexicographical as defined by the ASCII code for each alphabetic character, so all upper-case letter lexicographically precede all lower-case letters.

Command-line parameters

The names of the input files can be specified with command-line parameters. The default is the standard input. The program must also accept the following command-line options, in any order, before the file names.

- `low=`l specifies that the minimum frequency to be considered is l. The program must output only words with relative frequency $f_w \geq l$. l is given as an integer and represents a percentage value.
- `high=`h specifies that the maximum frequency to be considered is h. The program must output only words with relative frequency $f_w \leq h$. h is given as an integer and represents a percentage value.
- `-f` selects the *fancy* output, so each word must be preceded by its count.
- `-r` inverts the ordering of words by frequency: lower-frequency first. Same-frequency words must still be ordered lexicographically.
- `--` interprets every subsequent command-line option as a filename. This way, the program may also process input file names such as "`-f`" or "`high=23`".

You may use the *test_keywords.sh* script you find on-line as a test for the *keywords* program. A makefile is also available on-line.

A solution is available here.

►**Exercise 30.** (120') Write a library to represent and use a primitive geographic map. Write a single source file called *geomap.c* or *geomap.cc*. You may use either C or C++. The library must implement the interface functions defined in the *geomap.h* header file displayed below.

[download this file]

```
#ifndef GEOMAP_H_INCLUDED
#define GEOMAP_H_INCLUDED

#include <stddef.h>

struct geomap;

/* Create and return a new geomap object. Return 0 on failure. */
struct geomap * geomap_new();

/* Destroy the given geomap object and release all memory allocated by it. */
void geomap_delete(struct geomap * m);

void geomap_clear(struct geomap * m);

/* Add a point with the given coordinates and description.
 * Return 0 on failure, 1 on success. */
int geomap_add_point(struct geomap * m, double x, double y, const char * descr);

/* Add a rectangle with the given coordinates and description.
 * Return 0 on failure, 1 on success */
int geomap_add_rectangle(struct geomap * m,
                         double left_x, double bottom_y, double right_x, double top_y,
                         const char * descr);

/* Remove a point with the given coordinates and description.
 * Return 0 if not found, 1 if found (and removed). */
int geomap_remove_point(struct geomap * m, double x, double y, const char * descr);

/* Remove a rectangle with the given coordinates and description.
 * Return 0 if not found, 1 if found (and removed). */
int geomap_remove_rectangle(struct geomap * m,
                            double left_x, double bottom_y, double right_x, double top_y,
                            const char * descr);

typedef void (*point_callback)(double x, double y, const char * descr);
typedef void (*rectangle_callback)(double left_x, double bottom_y,
                                   double right_x, double top_y,
                                   const char * descr);

/* Iterate over all the elements in the geomap.
 * If the point callback is not null, calls that function with each point.
 * If the rectangle callback is not null, calls that function with each rectangle.
 * Return the number of elements in the iteration. */
size_t geomap_iterate_all(struct geomap * m,
                          point_callback pcb, rectangle_callback rcb);

/* Iterate over the element in the geomap that appear in a given region.
 * If the point callback is not null, calls that function with each point.
 * If the rectangle callback is not null, calls that function with each rectangle.
 * Return the number of elements in the iteration. */
```

```
size_t geomap_iterate_in_region(struct geomap * m,
                                point_callback pcb, rectangle_callback rcb,
                                double left_x, double bottom_y, double right_x, double top_y);

#endif
```

The system must store objects that can be represented as single points on a map, such as a bus stop, as well as objects that can be represented as rectangles, such as a building or a sport field. The library must support the dynamic addition and removal of point and rectangle objects. The library must also support two types of iterations. One is through all the objects in the map; another one is through all the objects that intersect in at least one point a given rectangular region.

Rectangles and regions are represented with the x coordinates of their left and right sides, and the y coordinates of their top and bottom sides.

The *test_geomap.c* program you find on-line is a test for the library. Use it to test your implementation. A makefile is also available on-line.

A solution is available here.

►**Exercise 31.** (120') Write a library that implements a table of string records. Write a single source file called *s_table.c* or *s_table.cc*. You may use either C or C++. The library must implement the interface functions defined in the *s_table.h* header file displayed below.

[download this file]

```
#ifndef S_TABLE_H_INCLUDED
#define S_TABLE_H_INCLUDED
#include <stddef.h>
#include <stdio.h>

struct s_table;

/* Create and return a new s_table object. Return 0 on failure. */
struct s_table * s_table_new();

/* Destroy the given s_table object and release all memory allocated by it. */
void s_table_delete(struct s_table * t);

void s_table_clear(struct s_table * t);

/* Add a record given by the given string
 * Return 0 on failure, 1 on success. */
int s_table_add_record(struct s_table * t, const char * begin, const char * end);

/* Remove a record with the given string
 * Return 0 if not found, 1 if found (and removed). */
int s_table_remove_record(struct s_table * t, const char * begin, const char * end);

/* Record processor: takes a record and returns an int
 */
typedef int (*feature_extractor)(const char * begin, const char * end);

/* Remove all the records that are selected by the given callback
 * function. A record is selected if the selector callback returns an
 * integer value that compares TRUE (i.e., != 0).
 *
 * Return the number of records that were removed. */
size_t s_table_remove_records(struct s_table * t, feature_extractor selector_callback);

/* Find a record with the maximal feature extracted by the given
 * callback function. Copies the corresponding record in the given
 * buffer, defined by the record char pointer and the given max buffer
 * length.  Never copies more than record_max_len characters.
 *
 * Return the number of characters copied in the record buffer, or 0
 * if  there are no records in the table. */
size_t s_table_max_feature(struct s_table * t, feature_extractor feature_callback,
                           char * record, size_t record_max_len);

/* Print the table on the given FILE stream, one record per line, with
 * the records sorted in increasing order of the feature extracted by
 * the given callback function. */
void s_table_print_sorted(struct s_table * t, FILE * f,
                          feature_extractor feature_callback);
#endif
```

The library must implement tables of generic records consisting of character strings. The tables must support the dynamic addition and removal of records. The library must also support access functions controlled by external feature-extraction functions. For example, a table might contain the records: "`carrots 15Kg CHF 40`", "`tomatoes 10Kg CHF 55`", "`peppers 2Kg CHF 6`", and the user might want to find the most expensive item. To do that, the user would provide a feature-extraction function: `int get_price(const char * b, const char * e);` to parse the record and extract the price. The user would then use that function with the library function `s_table_max_int_feature`.

The *test_s_table.c* program you find on-line is a test for the library. Use it to test your implementation. A makefile is also available on-line.

A solution is available here.

►**Exercise 32.** (60') Write a program called *imagenames* in a source file called *imagenames.c*. The program must read a set of file names from the standard input, one file name per line, each not larger than 1000 characters. The program must output the names of image files stripped of directory names and name extensions. The file name without directory names is the suffix of the full file name that follows the last (rightmost) directory-separator character '/', or the full name if there are no directory separators. The extension, if it exists, is the shortest suffix of the file name that includes a period character '.'.

The program must recognize a set of extensions that indicate image files. By default, these extensions are `.jpg`, `.jpeg`, `.png`, `.tiff`, and `.tif`. However, if a command-line parameter is given, then the program must interpret that parameter as the name of a file from which the program must read the set of file extensions. This file contains any number of extentions, one per line including the period character. If for whatever reason the program fails to read the extensions from the given file, the program must fall back to the default extensions.

For example, when running *imagenames* without command-line parameters and with the following input:

```
android/Messaging1550748367499.jpg
books/feyerabend_paul_against_method.pdf
gpu_computing_gems_9780123849892.pdf
strange_fruit.html
android/Messaging1552068751642.jpg
android/smile.gif
lib/images/usi-logo.png
lib/images/usi-logo.pdf
weather/lugano-2019.04.03.png
beyond_vietnam.html
lib/images/pantheon.jpeg
lib/images/ANTO.JPEG
```

The output must be

```
Messaging1550748367499
Messaging1552068751642
usi-logo
lugano-2019.04.03
pantheon
```

Use the imagenames.tgz test package to test your solution.

A solution is available here.

►**Exercise 33.** (60') In a source file called *fwd.c* write a library that implements a basic forwarding table for IPv4. The library must define (i.e., implement) all the declarations in the following header file.

File *fwd.h*: [download this file]

```
#ifndef FWD_H_INCLUDED
#define FWD_H_INCLUDED

struct fwd_table;

/* Constructor: return the null pointer in case of failure. */
struct fwd_table * fwd_new();

/* Destructor: clear all memory allocated for the given table. */
void fwd_delete(struct fwd_table * t);

/* Associate an IPv4 prefix with an output port in the given table.
 * The prefix is a string of the format IPv4-address/prefix-length.
 * For example, the strings "127.0.0.0/8" and "128.138.196.0/19" are
 * valid prefixes.  Return 0 on failure, either because the format of
 * the given prefix is wrong, or because memory is exhausted. Return
 * non-zero on success.
 */
int fwd_add(struct fwd_table * t, const char * prefix, int port);

/* Clear the table. */
void fwd_clear(struct fwd_table * t);

/* Return the output port for the given IPv4 address. The output port
 * is the one associated with the longest prefix that matches the
 * given address. Return -1 if no such prefix is found or if the
 * format of the given address is wrong.
 */
int fwd_forward(const struct fwd_table * t, const char * address);

#endif
```

Recall the basics of IPv4 addressing and forwarding. An *address* is a 32-bit value represented as a string $a_1.a_2.a_3.a_4$, where a_i are numeric values each corresponding to an unsigned 8-bit decimal number, and together they represent the 32-bit address where a_1 contains the most significant bits and a_4 the least significant bits of the address.

A *prefix* is an address plus a *length*. The length is an integer between 0 and 32. An address x matches a prefix p/ℓ of length ℓ if and only if the ℓ most significant bits of p are exactly the same as the ℓ most significant bits of x. For example, the address 127.0.1.1 matches the prefix 127.0.0.0/8, because both the address and the prefix have exactly the same most significant 8 bits, namely 127 (base 10) or 11111111 (base 2).

The forwarding function (fwd_forward) must return the port number associated with the *longest* prefix matching the given address in the given table.

Hint 1: use sscanf (declared in *stdio.h*) to read prefixes and addresses from strings.

Hint 2: you may use the integral types uint8_t and uint32_t declared in *stdint.h* to represent addresses and prefixes. These data types are easy to read as numbers, but then require bit-wise operations to perform the matching. As an alternative, you may simply represent addresses as

arrays of 32 bits each represented with, say, a character. These are perhaps more straightforward for matching, but require a conversion when reading bytes as numbers.

Use the fwd.tgz test package to test your solution.

A solution is available here.

►**Exercise 34.** (80') In a source file called *messaging.cc* write a library that implements a simple publish/subscribe messaging system. You may write the library in C or C++. The library must define (i.e., implement) all the declarations in the following header file.

File *messaging.h*: [download this file]

```
#ifndef MESSAGING_H_INCLUDED
#define MESSAGING_H_INCLUDED

struct server;

struct server * server_new(); /* return 0 on failure */
void server_delete(struct server * s);

struct receiver {
    void (*deliver)(struct receiver * r, const char * message);
};

/* return 0 on failure */
int add_interest(struct server * srv, struct receiver * r, const char * interest);
void remove_interest(struct server * srv, struct receiver * r, const char * interest);

void clear_receiver(struct server * srv, struct receiver * r);
void clear_all(struct server * srv);

void send(const struct server * srv, const char * message);

#endif
```

A messaging *server* records the interests of receivers, and then delivers messages according to those interests. The server processes messages for delivery through the `send` function. A *message* is simply a C string (zero-terminated).

A *receiver* is an object with a *deliver* function that the server invokes to deliver messages to that receiver. Notice that two distinct receiver objects may use the same deliver function.

A receiver can be associated with an *interest* through the `add_interest` function. The function takes a pointer to the receiver object together with an interest string. In this first exercise, an interest is a single *tag*. Exercise 35 extends the notion of an interest to a *set* of tags. Feel free to implement that extension directly.

A tag is a maximal sequence of alphabetic characters possibly starting with one *pound* or *at* character ('#' or '@'). For example, "`ciao`", "`@mamma`", and "`#bellaciao`" are valid tags. An interest (a *tag*) matches a message whenever the message contains one or more tags equal to the interest. The tags contained in a message are separated by non-tag characters. For example, the message "`Ciao @mamma, come stai?`" contains tags "`Ciao`", "`@mamma`", "`come`", and "`stai`", and therefore matches the interest "`@mamma`". However, the message "`Ciao mamma`" does not match that same interest. The message "`Ciao @mammamia`" also does not match the interest "`@mamma`".

The `send` function takes a server and a message, and delivers that message to all the receivers that added one or more interest matching the message. In particular, if server s has an interests t associated with receiver r, then a call `send`(s,m) on server s and a message m matching interest t—meaning that m contains t—results in the invocation of the delivery function of r with r and m as parameters, that is, r–>`deliver`(r,m).

The `remove_interest`(s,r,t) function removes the association between receiver r and interest t within server s; `clear_receiver`(s,r) removes all the interests associated with receiver r within server s; and `clear_all`(s) removes all the interests within server s.

Use the messaging.tgz test package to test your solution.

►**Exercise 35.** (40') In the same file *messaging.cc*, implement a variant of Exercise 34 in which an interest is defined by a *set* of tags. A set of tags T matches a message m when m contains all the tags in T. A set of tags is given as a string interpreted just like a message. Thus a call to add_interest(s,r,"#ciao @mamma") associates receiver r with the set of tags `#ciao` and `@mamma`, and is equivalent to add_interest(s,r,"@mamma #ciao"), add_interest(s,r,"@mamma,#ciao"), etc., and all these equivalent interests would match the message `"@mamma, how are you? (#ciao)."` but not the message `"@mamma, ciao ciao"`.

Notice that the remove_interest function must also treat interests as sets. For example, a call remove_interest(s,r,"@ciao−−#mamma") would remove the association defined by any one of the above three calls to add_interest.

The tests with suffix "s" from the same messaging.tgz test package test this extended functionality of the messaging library.

A solution is available here.

►**Exercise 36.** (40') In a source file called *simplesets.c* write a C library that implements a simple and limited form of sets of objects. The library must define (i.e., implement) all the declarations in the following header file.

File *simplesets.h*: [download this file]

```
#ifndef SIMPLESETS_H_INCLUDED
#define SIMPLESETS_H_INCLUDED

/* Represents a set of objects. */
struct ss;

/* Create a set containing a single object. */
extern struct ss * ss_create_singleton();

/* Destroy all previously created sets. */
extern void ss_destroy_all();

/* Merge two sets into a single set.
 *
 * As a result, all the elements in X and Y belong to the same set,
 * and both X and Y now refer to the same set.
 */
extern void ss_merge(struct ss * X, struct ss * Y);

/* Test whether two sets are disjoint. */
extern int ss_disjoint(struct ss * X, struct ss * Y);

#endif
```

Use the simplesets.tgz test package to test your solution.

A solution is available here.

►**Exercise 37.** (80') In a source file called *cleanup.cc* write a program that takes a list of names of directories as command-line parameters, and explores each directory to list files that could be deleted to save space. You may write this program in C or C++. The program uses a set of rules read from a file called *RULES* to determine which files could be deleted. In general, a rule checks that, if the directory contains a *source* file with extension S, then all the files with the same name and extensions $D_1, D_2, \ldots$ are *derived* files, and therefore could be deleted. The *RULES* file contains one rule per line, with the format: S`.`D_1`.`D_2`. ...`, starting with the source extension and then following with all the derived extensions, with all the extensions starting with a period character, as in the example file below:

```
tex.dvi.aux.bbl.blg.log
c.o.S
cc.o.S
cpp.o.S
java.class
```

You may assume that the lines in the *RULES* file are not more than 1000 characters long. However, you may not assume any limit on the number of lines/rules. You may also assume that file names, excluding the directory name, are not more than 256 characters long.

The program must print on the standard output the full name of each file that could be deleted, including the directory name.

Example: if the directory */home/sysprog/retake* contains the following files:

```
cleanup     cleanup.o  retake.aux  retake.pdf  RULES      simplesets.h  tests
cleanup.cc  Makefile   retake.log  retake.tex  simplesets.c  simplesets.o
```

then, invoking the command `./cleanup /home/sysprog/retake` should produce the following output, although not necessarily in this order:

```
/home/sysprog/retake/cleanup.o
/home/sysprog/retake/simpletest.o
/home/sysprog/retake/retake.aux
/home/sysprog/retake/retake.log
```

Hints: use the functions *opendir*, *readdir*, and *closedir* from the standard POSIX library to open, read, and then close a directory. In particular, *readdir* allows you to read the names of all the files in a directory. See the documentation directly on your computer with the command *man readdir*.

Use the cleanup.tgz test package to test your solution.

A solution is available here.

►**Exercise 38.** (20') In a file called *tank_control.c* write a library that implements a controller for a tank. The library must implement all the declarations in the following header file.

File *tank_control.h*: [download this file]

```
#ifndef TANK_CONTROL_H_INCLUDED
#define TANK_CONTROL_H_INCLUDED

extern void clear();

extern void change_bottle_capacity(unsigned int c);
extern void change_tank(unsigned int c);

extern void add(unsigned int c);
extern unsigned int ship_out_bottles();

extern unsigned int get_wastes();
extern unsigned int get_tank_level();

#endif
```

The tank is used as a temporary storage for liquids (e.g., wine) that arrive from producers, and that need to be bottled and then shipped. The library must keep track of the liquid in the tank, as well as the liquid that might get discarded as waste.

The tank has a capacity of 1000000ml (1000 liters). All capacity values are in milliliters. The function change_tank(c) changes the tank with a new one of capacity c. The current content of the tank is transferred to the new one. If the current content is more than c, then the new tank is filled to its maximum capacity and the rest of the liquid is discarded as waste.

The function add(c) adds c ml of liquid arriving from producers. This amount is added to the tank. If some amount of liquid exceeds the tank capacity, that is discarded as waste.

The function ship_out_bottles() is called to prepare bottles for shipping. The effect is to extract from the tank an amount of liquid that can completely fill a maximal number of bottles, which is the number returned by the function.

By default, bottles have a capacity of 750ml. The capacity of bottles can also be changed dynamically with change_bottle_capacity(c); get_tank_level() returns the current amount of liquid in the tank; get_wastes() returns the total amount of liquid discarded as waste; clear() resets the tank level and the waste counter to zero.

Use the tank_control.tgz test package to test your solution.

A solution is available here.

►**Exercise 39.** (30') In a file called *lists.c* write a library that implements the two functions declared in the following header file. You may not use any external library at all, not even the C standard library.

File *lists.h*: [download this file]

```
#ifndef LISTS_H_INCLUDED
#define LISTS_H_INCLUDED

struct list {
    int value;
    struct list * next;
};

extern struct list * concatenate_all(int count, struct list * lists []);

extern struct list * merge_sorted(struct list * a, struct list * b);

#endif
```

The two functions operate on lists of integers implemented with `struct list` objects (single-link lists). The two functions must not allocate any memory, so they must work exclusively with the objects passed as parameters.

Function `concatenate_all(int n, struct list * L[])` takes an array of n lists (pointers to the first element of a linked list), and returns a single list by concatenating all the elements of the n lists in the given order. Some lists might be empty (null).

Function `merge_sorted(struct list * a, struct list * a)` takes two lists sorted in non-decreasing order, and returns a single, sorted list that contains all the elements of both input lists. No values must be removed, so if there are repeated values, those must be part of the resulting list. Either or both input lists might be empty (null).

Use the lists.tgz test package to test your solution.

A solution is available here.

►**Exercise 40.** (70') In a file called *processes.c* write a library that implements a database of active processes in a system. The library must implement all the declarations in the following header file.

File *processes.h*: [download this file]

```
#ifndef PROCESSS_H_INCLUDED
#define PROCESSS_H_INCLUDED

struct processes;

extern struct processes * new_processes();
extern void delete(struct processes * p);
extern int add_from_file(struct processes * p, const char * filename);
extern void clear(struct processes * p);

struct query {
    int priority;
    long int rss;
    long int size;
    long int vsize;
    float cpu_usage;
};

struct query_result;

struct query_result * search(struct processes *, const struct query *);

extern int get_pid(struct query_result *r);
extern int get_ppid(struct query_result *r);
extern const char * get_user(struct query_result *r);
extern int get_priority(struct query_result *r);
extern float get_cpu_usage(struct query_result *r);
extern long int get_rss(struct query_result *r);
extern long int get_size(struct query_result *r);
extern long int get_vsize(struct query_result *r);
extern const char * get_command(struct query_result *r);

struct query_result * next(struct query_result *);
void terminate_query(struct query_result *);

#endif
```

new_processes() creates a new database object, or null in the case of error. delete(p) destroys a database object and completely deallocates all the memory used by it.

add_from_file(p, filename) reads information on processes from a given input file and returns 0 on error or 1 on success. Each input line contains information about one active process, as in the following example:

```
2331  2264 carzanig  19  2.4 317828 378876 3743140 firefox
```

In particular, each line contains the following fields separated by spaces in this order: *process-id* (integer) *parent-id* (integer) *user* (string up to 8 character long) *priority* (integer) *cpu-usage* (floating-point) *resident-size* (large integer) *size* (large integer) *virtual-size* (large integer) *command* (string up to 15 character long). All numeric fields are greater or equal to zero. (*Hint:* use fscanf to read process information from the input file.)

clear(p) clears the database, deallocating memory as needed.

search(p,q) searches the database using a query object q of type struct query. The query is interpreted as follows: if a field of the query is zero, then the query must match any value for that field. If a field is positive, then the query must match that field value exactly. If a field is negative, then the query must match any field value that is greater than the absolute value of the query field. For example a query initialized with q.priority=19; q.rss=0; q.size=0; q.vsize=0; q.cpu_usage=−10.0; must match all processes with priority 19 and with current CPU usage greater than 10.0.

The search function returns a sequence of results through a struct query_result object. This is an opaque object whose implementation is internal to the library. If the query has no results, or in case of error, search must return the null pointer. Otherwise, search returns a valid pointer to a struct query_result object that can then be used to read the values of the matching process and to iterate through them.

In particular, get_pid(r) returns the process id of the process for the current result pointed by r, get_user(r) returns the username, etc. Then next(r) returns the next query result (or moves the result object to the next result and returns the same result object). If there are no more results, or in case of errors, next(r) returns the null pointer and deallocates the query_result object. A query result iteration can also be terminated immediately with terminate_query. In any case, the results must be returned in the order they were added to the database.

Use the processes.tgz test package to test your solution.

A solution is available here.

►**Exercise 41.** (120') In a source file called *rooms.cc* write a C++ library that implements a room reservation system defined by the following interface:

File *rooms.h*: [download this file]

```
#ifndef ROOMS_H_INCLUDED
#define ROOMS_H_INCLUDED

#include <iostream>

struct room {
    int floor;
    int number;
    int capacity;
};

const int ANY_FLOOR = -1;
const int ANY_ROOM_NUMBER = -1;
const int ANY_CAPACITY = -1;

struct schedule {
    int start;
    int finish;
    int duration;
};

void clear();
void add_rooms(const room * begin, const room * end);
int make_reservation(room * r, schedule * t, const char * event);
int cancel_reservation(int floor, int number, int start);
void print_schedule(std::ostream & output,
                    int floor, int number, int start, int finish);
#endif
```

A *room* is identified uniquely by a floor and a room number, and has a given seating capacity. A *time* is expressed as an integer multiple of some fixed time unit starting from some initial time.

Initially the reservation system does not manage any room.

add_rooms(begin,end) adds rooms from an array of room objects. The begin pointer points to the first room to add, and end points immediately past the last room to add. If a room already exists, then its capacity must be updated with the given new value. Notice that the room objects passed to add_rooms() are owned by the application. Their values might change or they might even be deallocated immediately after the call to add_rooms().

make_reservation(r, t, description) makes a reservation for a given event. The room object r and the schedule object s express *requirements* for the reservation. In particular, r->capacity is the minimum required capacity of the room. However, if r->capacity is equal to ANY_CAPACITY, then any capacity would work. Similarly, r->floor requires a specific floor, or no specific floor if that is equal to ANY_FLOOR, and r->number requires a specific room number, or not if equal to ANY_ROOM_NUMBER. The schedule object s represents a period of a given duration s->duration greater than zero that should be scheduled between a given start time s->start and a given finish time s->finish. A reservation can be made in a room if there is an interval of duration s->duration between times s->start and s->finish (inclusive) that does not overlap with any other event scheduled in the same room. Notice that an interval that finishes at time t does not overlap with another interval that starts at time t.

make_reservation returns 1 or 1 when a room is found or not found, respectively. If a room is found, make_reservation also sets the values of r->floor, r->number, and r->capacity with those of the chosen room, and s->start and s->finish with the chosen time interval in which that room is available. When multiple rooms or schedules are available, make_reservation must select the lowest compatible room capacity, the lowest floor, the lowest room number, and the lowest starting time, in this order.

cancel_reservation(floor, number, start) cancels an existing reservation. If a reservation exists for the given room, as identified by floor and number, and for the given start time, then the result is 1, and that reservation is canceled. Otherwise, the result is 0 and no reservation is canceled.

print_schedule(out, floor, number, start, finish) selects rooms and prints their schedule on the out stream. The floor and room numbers determine which rooms are selected. These are the rooms with number number or with any room number if number is ANY_NUMBER, on floor floor or any floor if floor is ANY_FLOOR. The given start and finish times determine which events are selected. These are all the events in selected rooms that overlap with the given interval, or more specifically the events that start at or before the given finish time or that end at or after the given start time.

The output consist of a set of lines for each room. The first line identifies a room with the format "`room` *room-number* `floor` *floor-number*`:`". Each of the following lines describes an event scheduled in that room with the format "`event:` *start-time finish-time description*". If there are no events scheduled for that room in the requested interval, then no lines follow. The rooms must be sorted first by floor and then by room number. Scheduled events must be sorted by starting time. For example:

```
room 7 floor 1:
event: 10 11 meeting
event: 14 16 final exam
room 10 floor 1:
room 3 floor 2:
event: 17 18 movie screening
event: 18 19 poker game
room 4 floor 2:
event: 12 13 lunch
event: 15 18 study session
```

clear() clears all the data and releases all the resources allocated by the system.

Use the rooms.tgz test package to test your solution.

A solution is available here.

►**Exercise 42.** (40') In a source file called *countchars.c* write a C program that counts the occurrences of a set of target characters in a set of files or in the standard input. The program reads all its parameters from the command line. The first parameter is a string containing the set of target characters that must be counted. If there are no command-line parameters or if the first parameter is an empty string, then the program must output "`No target characters specified.`" the error message in a single output line, and then terminate.

If there are no more command-line parameters after the string of target characters, then the program must count the target characters from the standard input. Otherwise, all the command-line parameters after the first one are interpreted as input file names, which means that the program must count the target characters from all the given files.

For each input file, either named files or the standard input, the program must output a single line in the following format:

$$\textit{filename}\ c_1\texttt{:}n_1\ c_2\texttt{:}n_2\ c_3\texttt{:}n_3\ \ldots$$

Where *filename* is the name of the file, or "`(stdin)`" for the standard input, and c_i is the i-th target character given in the string of target characters passed as command-line argument, and n_i is the number of times that character c_i appears in the input file. If there is any error in opening or reading the file, then the output should be "`Error reading file` *filename*". After an error, the program mut continue normally with the remaining files, if any.

The term *character* here corresponds to an object of type `char` in C. This means that the program interprets the target string (first command-line argument) as a simple C-string representing a sequence of *bytes*. In other words, the program must consider each individual *byte* in that string as a target character, and must therefore count the occurrences of that *byte* in each input file.

Hint: recall that a `char` is an integer numeric type, so a `char` object has a numeric value. Recall also that the range of integer values for a `char` object is implementation-defined and may be negative. You should instead interpret those `char` values as *unsigned* byte values, since those are the values you get from a function line `fgetc` when you read the input. One way to convert a command-line argument as a string of *unsigned* byte values is as follows:

```
const unsigned char * target = (const unsigned char *)argv[1];
```

With that, you can now use the target characters as unsigned byte values.

Use the countchars.tgz test package to test your solution.

A solution is available here.

►**Exercise 43.** (80') In a source file called *stocks.cc* write a C/C++ library to manage stock trades. The library must implement the following interface file *stocks.h*: [download this file]

```
#ifndef STOCKS_H_INCLUDED
#define STOCKS_H_INCLUDED

struct trades_log;

struct trades_log * new_log();          /* constructor */
void delete_log(struct trades_log * l); /* destructor */

int new_trade(struct trades_log * l, double time,
              const char * symbol, double price, unsigned int quantity);
void clear(struct trades_log * l);

void set_time_window(struct trades_log * l, double w);

unsigned int total_quantity(const struct trades_log * l);

double high_price(const struct trades_log * l);
double low_price(const struct trades_log * l);

double stock_high_price(const struct trades_log * l, const char * symbol);
double stock_low_price(const struct trades_log * l, const char * symbol);
double stock_avg_price(const struct trades_log * l, const char * symbol);

#endif
```

A *trade* is a transfer of stocks from a seller to a buyer, and it is characterized by the following information:

- the *time* when the trade is made, represented by a floating-point number representing seconds from a fixed initial date;
- the *symbol* of the company whose stocks are traded, which is a string of at most five characters (ASCII);
- the *quantity* of stocks traded, which is a positive integer;
- the *price* at which the stocks are traded, which is a positive floating-point number.

The library must implement a *log* of the trades made within a certain time window. The time window is 60 seconds by default, and can be set for a log l with set_time_window(l,w). When set_time_window(l,w) *reduces* the window size compared to the previous value, all the trades that fall out of the new window must be removed.

A new trade made at time t for quantity q of stock s at price p can be added to a log l with new_trade(l,t,s,p,q). new_trade must check the validity of the trade, meanining that quantity and price are positive quantities, and that the time is not less than the time of any of the previous trades. new_trade must return 1 on success and 0 on failure or with invalid input.

Notice that adding a new trade at time t for a log with time-window w must result in the effective removal of all the trades made before time $t - w$. A log l can also be explicitly cleared with clear(l).

The library then implements a number of computations over a trade log: total_quantity(l) returns the total quantity of stocks traded within the time window of a log l; low_price(l) and high_price(l) return the lowest and highest price across all stocks, respectively; then stock_high_price(l,s), stock_low_price(l,s), and stock_avg_price(l,s) return the high, low, and average price for a given

stock *s* (symbol). If log *l* is empty, then total_quantity(l), low_price(l), and high_price(l) must all return 0. Similarly, if there are no trades for stock *s*, then stock_high_price(l,s), stock_low_price(l,s), and stock_avg_price(l,s) must all return 0. (Recall that prices and quantities are always positive values, so 0 is an invalid quantity and an invalid price.)

Use the stocks.tgz test package to test your solution.

A solution is available here.

►**Exercise 44.** (80') A text file stores the results of a number of games played by a number of teams. Each line of the file represents a game and contains the names of the two teams playing the game followed by the score of the first and second team, respectively. Team names are strings of up to 20 alphanumeric ASCII characters (without spaces or other characters). Team names and scores are separated by spaces on each input line. Scores are non-negative integers. The team with the highest game score wins that game and gains 3 points in the overall tournament score. Otherwise, if the game is tied (same score for both teams) then both teams gain 1 point in the overall tournament score.

In a source file called *scoreboard.c* write a C program that reads a number of game results from the standard input, and writes on the standard output the tournament scoreboard as defined below. The program takes one optional command-line argument that specifies the maximum number of teams. By default the maximum number of teams is 10.

The scoreboard is a text file in which each line indicates a team name, the tournament score of that team, and the number of games played in the tournament by that team. The scoreboard must be sorted in decreasing order of tournament score, from highest to lowest, and if two or more teams have the same score, in alphabetical order of team names.

As an example, below is a valid input and the corresponding expected output.

Input:

```
programmers coders 3 1
designers programmers 2 5
designers coders 1 1
```

Output:

```
programmers 6 2
coders 1 2
designers 1 2
```

The program must exit with EXIT_FAILURE when the input is incorrect, for example if there are more than the given maximum number of teams, or if there is in error in reading the input or writing the output, or for any other failure such as in memory allocation. Recall that EXIT_FAILURE and EXIT_SUCCESS are defined in the *stdlib.h* standard header file.

Use the scoreboard.tgz test package to test your solution.

A solution is available here.

►**Exercise 45.** (40') In a source file called *bufile.c*, implement the bufile structure and the three functions declared in the *bufile.h* header file [download this file]

```
#ifndef BUFILE_H_INCLUDED
#define BUFILE_H_INCLUDED

struct bufile;

typedef struct bufile BUFILE;

BUFILE * bufopen(char * begin, char * end);
void bufclose(BUFILE * buf);

char *bufgets(char *s, int size, BUFILE * buf);

#endif
```

The BUFILE structure is analogous to the FILE structure of the standard I/O library. In particular, BUFILE represents a stream of bytes whose content is given in a memory buffer consisting of an array of characters.

A BUFILE object is created with bufopen(begin,end). Specifically, begin points to the first character of the content buffer, and end points to the first character past the end of that buffer. On success, bufopen returns a valid pointer to a BUFILE object, or the null pointer on failure. The caller guarantees that the buffer and its content remain valid until the caller closes the BUFILE object with bufclose, which must release any and all resources allocated with bufopen.

The bufgets function is analogous to the fgets function from the standard I/O library.[1] In particular, bufgets(s,size,bf) reads in at most one less than size characters from the buffer stream, and stores them into the buffer pointed to by s. Reading stops when the file buffer stream ends or after a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer. bufgets(s,size,bf) returns s on success, and NULL on error or when the end of stream occurs while no characters have been read.

Use the bufile.tgz test package to test your solution.

A solution is available here.

[1] In fact, this specification is copied from the documentation of the standard function *fgets*.

►**Exercise 46.** (120') In a source file called *media_stats.cc* write a C or C++ module to compute simple statistics on a database of media files (audio, video) and user viewings of those files.

A media file is identified by a *title* given as a C-string, and has a *duration* that is greater than zero and that is expressed in seconds. A user is identified by a *name* that is also given as a C-string. There are no constraints on the characters or the lengths of titles and user names. The module must implement the methods declared in the *media_stats.h* header file (see next page) and documented below.

- add_media(title,duration) adds a media file in the database. The title must be new and the duration must be greater than 0.0. In this case, and if every necessary memory allocation succeeds, this operation must return 1. Otherwise, this operation must fails, leaving the database unchanged and returning 0. If you use C++, you may choose to throw a std::bad_alloc exception when memory allocation fails instead of returning 0.
- add_view(title,user,start,finish) records a viewing of a media file. Users can view a media file (or listen to it if it is an an audio file) starting from any time position and for whatever duration. The given title must exist in the database of media files. Also, the given *start* and *finish* times must be valid and consistent with the duration of the given media file. In particular, *start* and *finish* must indicate time points between zero and the duration of the media file, and they must define a non-zero segment of the file, so *start* must be less than *finish*. If all these conditions are satisfied, and if the necessary memory can be allocated, the operation must succeed and return 1. Otherwise, the operation must fail, leaving the databases unchanged and returning 0. In C++, you may choose to throw a std::bad_alloc exception when memory allocation fails.
- clear_media() clears the database. Removes all the media files and therefore all the viewings.
- clear_views() clears all the viewings but preserves the database of media files.
- unique_viewers(title) returns the number of unique users who viewed any portion of the given media file. Any viewing of any valid (non-zero) duration counts. However, two or more viewings by the same user count as one. Return -1 if the title does not exist in the database.
- total_time_viewed(title,user) returns the total time, in seconds, of the given media file viewed by the given user. Notice that any segment viewed multiple times, possibly in partially overlapping viewings, must count only once. For example, if a user first views the segment starting at time 1.0 and finishing at time 2.0, and then in another viewing the segment starting at time 1.5 and finishing at time 2.5, then the total time is 1.5 seconds. Similarly, if a user views the segments given by the *start–finish* pairs $(40, 50)$, $(15, 25)$, $(20, 30)$, then the total time viewed is 25 seconds. The results must be -1.0 if the given title does not exist and 0.0 if the title exists but the given user never viewed that file.
- complete_views(title) return the number of users who viewed the given media file completely at least once in one or more viewings. The result is -1 if the title does not exist.

Use the media_stats.tgz test package to test your solution.

File *media_stats.h* [download this file]

```
#ifndef MEDIA_STATS_H_INCLUDED
#define MEDIA_STATS_H_INCLUDED

// Add a video or music title with the given total duration. Return 1
// or 0 on success or failure, respectively.
extern int add_media(const char * title, double duration);

// Add a media viewing of the given title by the given user of a
// segment defined by the given start and finish times. Return 1 or 0
// on success or failure, respectively.
extern int add_view(const char * title, const char * user,
                    double start, double finish);

// Clear all media titles and therefore all views.
extern void clear_media();

// Clear all media views but keep the media titles.
extern void clear_views();

// Number of unique users who have viewed any portion of the given
// media. Return -1 if the title does not exist.
extern int unique_viewers(const char * title);

// Total time of the given title that the given user has seen at least
// once. Return -1 if the title does not exist. Return 0 if the user
// does not exist.
extern double total_time_viewed(const char * title, const char * user);

// Number of users who viewed the entire duration of the given media
// file in one or more viewings. Return -1 if the title does not exist.
extern int complete_views(const char * title);

#endif
```

A solution is available here.

►**Exercise 47.** (60') In a source file called *wcmp.cc* write a C or C++ implementation of the function

```
int wcmp(const char * a, const char * b);
```

This function takes two C-strings a and b and returns a value that indicates a word-by-word comparison between a and b. A word-by-word comparison considers strings a and b as two, possibly empty, sequences of words, $W_a = a_1, a_2, \ldots$ and $W_b = b_1, b_2, \ldots$. A *word* is a maximal contiguous sequence of alphabetic characters as defined by the standard function `isalpha` declared in *ctype.h*. So, words are sequences of letters separated by other characters such as spaces, punctuation, and numbers. For example, the string `"don't worry"` consists of a sequence of three words, namely `"don"`, `"t"`, and `"worry"`.

Thus `wcmp(a,b)` must return 1, −1, or 0, if the word-by-word comparison between the sequences W_a and W_b is *greater-than, less-than,* or *equal,* respectively.

The semantics of a word-by-word comparison between two sequences of words $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ is equivalent to a lexicographical comparison in which we compare pairs of corresponding words $(a_1, b_1), (a_2, b_2), \ldots$, similar to a character-by-character comparison. Each comparison of two words is an ordinary lexicographical comparison based on the numeric code of the characters. So, for example, the sequence ("pasta", "al", "pomodoro") compares *greater-than* the sequence ("pasta", "ai", "funghi"), because the two words in the first position are equal ("pasta") but the two words in the second position compare *greater-than*, since "al" is lexicographically greater than "ai", because the character `'l'` has a numeric code greater than that of character `'i'`.

Notice that in a lexicographical comparison, a sequence is always greater than one of its proper prefixes. For example: ("pizza", "margherita") compares *greater-than* ("pizza").

Use the wcmp.tgz test package to test your solution.

A solution is available here.

►**Exercise 48.** (60') In a source file called *blob.cc* write a C or C++ implementation of the functions declared in the following header file *blob.h* [download this file]

```
#ifndef BLOB_H_INCLUDED
#define BLOB_H_INCLUDED

#define CHUNK_MAX_LEN 100

struct chunk {
    char data[CHUNK_MAX_LEN];
    unsigned int length;       /* number of stored chars, possibly 0 */
};

struct chunk_list {
    struct chunk * c;
    struct chunk_list * prev;
    struct chunk_list * next;
};

struct blob {
    struct chunk_list sentinel; /* doubly-linked list with sentinel */
};

extern unsigned int count_char (const struct blob * b, char c);
extern unsigned int copy_to_buffer (const struct blob * b,
                                    char * buf, unsigned int maxlen);
extern unsigned int copy_to_buffer_reverse (const struct blob * b,
                                            char * buf, unsigned int maxlen);
#endif
```

All the functions take a pointer to a *blob* object that represents a potentially large sequence of bytes. The blob structure stores such a sequence within a list of *chunks*, each storing a subsequence of bytes. Each chunk is stored as an array of bytes in a chunk structure. The blob object then simply consists of a doubly-linked list with sentinel of chunk objects. Notice that the list may be empty (no chunks), which represents an empty blob. Individual chunks may also be empty, when their length is 0.

count_char(b,c) must return the number of occurrences of character c in blob b.

copy_to_buffer(b,buf,maxlen) must copy at most maxlen bytes from the blob b into the given buffer buf. If the blob contains less than maxlen bytes or exactly maxlen bytes, then all those bytes are copied into the buffer and the return value is the number bytes actually copied. If the blob contains more than maxlen bytes, then the function must copy the first maxlen bytes in the blob, and the return value must be maxlen +1.

copy_to_buffer_reverse(b,buf,maxlen) must copy at most maxlen bytes from the blob b into the given buffer buf in reverse order, starting from the end of the content of the blob. The order is reversed in the sense that the last byte of the blob must be copied in the first position in the buffer, then the byte right before the last one in the blob must go to the second position in the buffer, and so on. As for copy_to_buffer, the return value is the number of bytes actually copied, or maxlen +1 if the blob contains more than maxlen bytes.

Use the blob.tgz test package to test your solution.

A solution is available here.

►**Exercise 49.** (40') In a source file called *linelen.c* write a C program that reads lines from the standard input, and writes on the standard output the lines that are between a minimum and maximum length $min \le \ell \le max$. A "line" is a maximal sequence of characters (bytes) terminated by the new-line character ('\n'). This means that the program must not print anything past the last new-line character in the input.

The minimum length *min* and the maximum length *max* are passed to the program as non-negative decimal numbers as the optional first and second command-line arguments, respectively. By default, the minimum length is $min = 0$ and the maximum length is $max = \infty$. For example, the command "`linelen 10 100`" must print all the input lines whose length is between 10 and 100 characters. Instead, the command "`linelen 20`" must print all the input lines whose length is at least 20 characters. And the simple command "`linelen`" must print all lines.

When printing a line, regardless of its length, the program must truncate the output to at most 60 characters. If the line is longer than 60 characters, then the program must add three dots at the end of the first 60 characters. For example, in printing the following input lines:

```
0---------1---------2---------3---------4---------5
0---------1---------2---------3---------4---------5---------6
0---------1---------2---------3---------4---------5---------6---------7
```

the program must output

```
0---------1---------2---------3---------4---------5
0---------1---------2---------3---------4---------5---------6
0---------1---------2---------3---------4---------5---------6...
```

Notice that the selection of which lines to print is made on the length of the input, not the length of the truncated output. For example, `linelen 2000` must print all input lines of 2000 or more characters, but it must print only the first 60 characters of each line.

Use the linelen.tgz test package to test your solution.

A solution is available here.

►**Exercise 50.** (80') In a source file called *videos.c*, implement a library that provides statistics for a video serving platform. The library API is given in the *videos.h* header file below:

[download this file]

```
#ifndef VIDEOS_H_INCLUDED
#define VIDEOS_H_INCLUDED

struct platform;

struct platform * create ();
void destroy (struct platform * p);
void clear (struct platform * p);

int add_video (struct platform * p, const char * title, int minutes);
int add_view (struct platform * p, int video_id, int minutes);

int total_view_minutes (struct platform * p);
int total_full_views (struct platform * p);

int video_by_title (struct platform * p, const char * title);

#endif
```

The platform is represented by an object of type `struct platform` defined by the library. The library must support multiple platforms. Each platform can be created with `create` and destroyed with `destroy`. As usual, the destructor must release all allocated resources for that platform. All videos and viewings can be removed from a platform with `clear`. This must also release all allocated resources for the videos and views in the platform, but without destroying the platform object itself, which can be therefore used as if it were newly created.

Videos can be added to a platform with `add_video`, which takes a title of up to 100 characters, and the duration of the video in minutes. `add_video` must return a non-negative unique identifier for the video within the given platform, or -1 in case of error.

The video identifier can then be used with the `add_view` function to add a viewing for that video. The viewing specifies a number of minutes served to the viewer. If the number of minutes is greater or equal to the length of the video, the viewing is considered a *full* viewing. The return value must be 1 on success, or 0 on failure, for example if the given identifier does not exist.

The `total_view_minutes` and `total_full_views` functions return the total view minutes and total full views for all videos, respectively. The `video_by_title` function takes a title and returns the identifier of any video with that title, or -1 if no such title exists.

Hint: focus on correctness rather than complexity; use a simple data structure.

Use the videos.tgz test package to test your solution.

A solution is available here.

►**Exercise 51.** (30') In a source file called *twochars.c* or *twochars.cc*, write a C or C++ program that, for each alphabetic character c read from the input, outputs the most frequent alphabetic character c' that follows c in the input. You may assume that the only characters the program has to consider are the 26 characters of the Latin alphabet (A to Z).

The program must not distinguish between lowercase and uppercase characters. So, for example, the character 'a' must be considered the same as 'A'. The output is an alphabetically ordered list of pairs of uppercase characters. If two or more characters $c'_1, c'_2, \ldots$ follow character c with a maximal frequency, then the program must output c followed by the least of $c'_1, c'_2, \ldots$ in alphabetical order.

For example, with the input "`Abracadabra bonobo`" the program must output:

```
AB
BO
CA
DA
NO
OB
RA
```

Notice that `B` is followed by `R` just as many times (two) as by character `O`, but the program outputs `BO` because `O` precedes `R` in alphabetical order. You may assume that the numeric value of the characters reflects their alphabetical order.

Notice also that the program must consider only the characters that follow other characters *immediately*. So, if the input is "`A b r a c a d a b r a`" then the output is empty.

Hint: the library functions `islower()` and `isupper()` declared in *ctype.h* can be used to check whether a character is a lowercase or uppercase alphabetic character, respectively. Also declared in *ctype.h*, `toupper()` can be used to convert a lowercase letter into the corresponding uppercase letter.

Use the twochars.tgz test package to test your solution.

A solution is available here.

►**Exercise 52.** (90') In a source file called *expand.c* or *expand.cc*, write a C or C++ program that copies its standard input to its standard output replacing some strings in the input as specified by a set of rules. A rule, denoted $p \to t$, is defined by a *pattern* strings p and a replacement *text* t, and specifies that every occurrence of pattern p in the input must be replaced with text t in the output. For example, with the rule "`Hello`"→"`Ciao`" and the input "`Hello World!`", the program must output "`Ciao World!`".

The program reads a sequence of rules from one or more files. The file names are given as command-line arguments. With no arguments, the program reads the default file *RULES*. A rule $p \to t$ is written in a file as a line "p`:`t". You may assume that lines are at most 2000 characters long. A text t that consists of multiple lines continues on multiple lines in the rules file with each of the continuing lines starting with a colon. Lines that do not contain a colon character must be ignored. See the example below:

rules file	**rules**

`poem:Tyger Tyger, burning bright,`
`:In the forests of the night;`
`:What immortal hand or eye,`
`:Could frame thy fearful symmetry?`
`:`
`ciao:hello`
`these two lines will be ignored`
`the following text is empty`
`empty text:`
	"poem" → "Tyger Tyger, burning bright,\n
	In the forests of the night;\n
	What immortal hand or eye,\n
	Could frame thy fearful symmetry?\n"
	"ciao" → "hello"
	"empty text" → ""

Notice that a pattern may not contain any new-line or colon (`:`) character, and may not be empty. Conversely, the replacement text may contain any character, including new-line and colon, and it may also be empty.

Should any error occur while reading a rules file f, the program must print the error message "`invalid rules file` f" and then terminate immediately with the exit status EXIT_FAILURE. The exist status must otherwise be EXIT_SUCCESS. (EXIT_SUCCESS and EXIT_FAILURE are defined in *stdlib.h*.)

The rules are applied in the order given in the rules files. That is, if multiple rules are applicable at any point in the input, the program applies the first applicable rule. For example, with the rules "`Hello`"→"`Ciao`" and "`Hell`"→"`Inferno`" given in this order in a rules file, and with the input "`Hello darkness, my old friend`", the output must be "`Ciao darkness, my old friend`". Notice that the input "`Hello darkness...`" matches the pattern of both rules. Had the rules been given in the inverse order, the output would have been "`Infernoo darkness, my old friend`".

The application of rules is mutually exclusive and non-recursive. This means that, when a rule is applied to some pattern in the input and therefore that pattern is replaced with the corresponding text in the output, neither the pattern nor the replacement text, nor any input before the pattern, are considered further for the application of any rule. For example, given the rules "`A`"→"`ASA`", "`I`"→"`ISI`", and "`SANI`"→"", the input "`ANIMA`" is transformed into the output "`ASANISIMASA`", and the input "`SANITY`" is transformed into "`TY`".

Hint: in C++, the std::ifstream class declared in fstream can be used to read from a file, and a good way to read lines is with std::getline declared in string. In C, use fopen (and fclose) and fgets. In C++, see std::string and its substr feature. In C, strncmp might be useful for pattern matching.

Use the expand.tgz test package to test your solution.

A solution written in C++ is available here.

►**Exercise 53.** (120') In a source file called *snake.c* or *snake.cc*, write a C or C++ module that manages the state of a simple snake video game. The module must implement the interface defined by the following declarations and macros

[download this file]

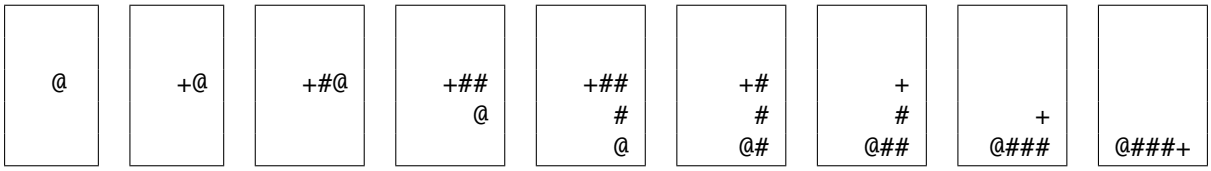
```
#define SNAKE_UP 1
#define SNAKE_DOWN 2
#define SNAKE_LEFT 3
#define SNAKE_RIGHT 4
#define SNAKE_OKAY 1
#define SNAKE_FAIL 2
#define SNAKE_NEW 3

struct snake;

struct snake * snake_new (int, int, int);
void snake_destroy (struct snake *);
int snake_start (struct snake *, int, int, int);
int snake_change_direction (struct snake *, int);
int snake_step (struct snake *);
const char * snake_get_field (const struct snake *);
int snake_get_status (const struct snake *);
```

The player controls a snake that moves across a game field consisting of a $c \times r$ rectangle of cells. That is, the game field consists of r rows of c cells each. The game proceeds in discrete steps. At each step, the head of the snake moves from its current cell to the cell immediately up or down or to the left or to the right. The direction of movement can be set at each step, or otherwise remains unchanged.

The snake starts from a set initial position and direction of movement. Initially, the snake consists of a single cell, which is its head. Each movement of the head also grows the body, so that the rest of the body stays in the same positions. The snake grows up to a maximum length of ℓ cells, at which point the whole snake also starts moving following the head.

The game field must be represented as a matrix of characters, one for each cell. An empty cell is represented by the space character (' '). The '@' character represents the head of the snake. If the length of the snake is greater than one, then the tail of the snake is represented by the '+' character, and if the length is greater than two, then any cell between the head and tail of the snake is represented by the '#' character.

Below is an example of the initial movement of a snake of maximum length $\ell = 5$ starting from the center of a 5×5 game field and moving two steps to the right, then two steps down, and then to the left.



The objective of the game is to move the snake within the rectangular game field so that the snake head will never collide with any other part of the snake (including the tail). If one movement step causes the head to fall off of the game field or to hit another part of the snake, the game is lost and can not progress further.

The preprocessing macros SNAKE_UP, SNAKE_DOWN, SNAKE_LEFT, and SNAKE_RIGHT identify the direction of movement. The macros SNAKE_OKAY, SNAKE_FAIL, and SNAKE_NEW define the status of the game, as we detail below.

Your module must define the snake structure to represents the state of one game, as well as all the related functions specified in detail below. Notice that each snake object is independent and self-contained, so two or more games must coexist, each stored in its snake object. In fact, each function other than the constructor (snake_new) takes and operates on a particular game object, which is always passed as the first parameter.

- struct snake * snake_new (int c, int r, int l) creates a new game over a field of $c \geq 1$ columns and $r \geq 1$ rows, and with a snake that grows up to a maximum size of $\ell \geq 1$ cells. The return value is a valid pointer to a properly constructed object, or the null pointer in case of error or invalid parameters. The status of a newly created game is SNAKE_NEW and the game field and the snake are in an undefined state.
- void snake_destroy (struct snake * s) destroys a game object, releasing all its allocated resources.
- int snake_start (struct snake * s, int c, int r, int d) starts the game with the snake head initially positioned at row r and column c, and moving in direction d. The row and column coordinates start from 0: $(0,0)$ is the upper-left corner of the game field; (c,r) is the lower-right corner of a $c \times r$ game field.

  snake_start initializes the given game object so that the game can start with a first step. If a game was already in progress or it had ended, then snake_start resets the game. The given initial position (c,r) must be contained within the game field, and the direction must be SNAKE_UP, SNAKE_DOWN, SNAKE_LEFT, or SNAKE_RIGHT. If the parameters are valid, the return value is SNAKE_OKAY and the status of the game is SNAKE_OKAY. Otherwise, the return value and the status of the game are SNAKE_FAIL. Notice that d defines the initial direction, but this function does not move the snake head from its initial position at column c and row r.
- void snake_change_direction (struct snake * s, int d) changes direction for the snake head. This function does not itself move the head. It just sets the direction for the next step.
- int snake_step (struct snake * s) If the game status is SNAKE_OKAY, then move the snake by one step. For any other game status, the function has no effect and the return value is SNAKE_FAIL. If the movement is valid, then the return value is SNAKE_OKAY. Otherwise, if the movement causes the snake head to hit itself or to fall off the game field, then the game is lost and the status of the game changes to SNAKE_FAIL. In this case, the return value is also SNAKE_FAIL.
- const char * snake_get_field (const struct snake * s) returns a pointer to a matrix of characters representing the game field. The matrix is stored starting from the character at the top-left corner of the field, continuing with the characters of the first row, left-to-right, then the second row, and so on until the character at the bottom-right corner of the field, which will be in position $c \times r$ for a snake object created with c columns and r rows.
- int snake_get_status (const struct snake * s) returns the status of the game.

Use the snake.tgz test package to test your solution.

A solution written in C++ is available here.

A solution written in C is available here.

►**Exercise 54.** (30') In a source file called *openclose.c* write a program that copies its standard input to its standard output by filtering certain parts of each input line as follows.

The program is given a set of pairs of characters $(o_1, c_1), (o_2, c_2), \ldots, (o_n, c_n)$, where c_i is the "closer" character corresponding to the "opener" character o_i. The pairs of opener and closer characters are given as a single string "$o_1 c_1 o_2 c_2 \ldots o_n c_n$" of length $2n$ passed to the program as its first command-line argument. For example, if the first argument is the string `bdpq`, then that means that opener characters `b` and `p` are associated with closer characters `d` and `q`, respectively. By default, the program uses the string "`()[]{}`" to define the opener/closer character pairs.

The program copies lines from the standard input to the standard output, except that a pair of characters $o_i c_i$, where an opener character o_i is followed by the corresponding closer character c_i, is omitted (not copied to the output) if o_i and c_i are consecutive characters in the input line (one right after the other), or if every other character between o_i and c_i is also omitted according to this rule. For example, with the default opener/closer characters, an input line "`--{-}-{[]}-`" is copied to the output as "`--{-}--`". With `bdpq` as openers/closers, an input line "`Ciao babdba qppqpqqbpd`" is copied to the output as "`Ciao baba qbpd`".

Input lines are properly terminated by the end-of-line character ('`\n`') and do not contain more than 1000 characters.

Hint: an easy way to transform an input line into the corresponding output line is by using a stack of characters to match and therefore remove a closer character together with the corresponding opener character. In fact, you can build the output line itself as a stack of characters.

Use the openclose.tgz test package to test your solution.

A solution is available here.

►**Exercise 55.** (90') In a source file called *tformat.c* write a C function declared as follows:

`int t_format(char * t, struct t_opts * opt)`

that formats the given text *t*, adjusting its spacing in various ways. The spacing is first normalized according to these rules:

- All space characters at the beginning and at the end of a line are removed.
- Space characters are those for which the standard library function `isspace()` returns *true*, except that spaces are considered within each line of the text, not across lines. Therefore the end-of-line character ('\n') is not considered a space
- Empty lines at the beginning and at the end of the text are removed. A line might be empty as a result of the removal of space characters.
- For the remaining space characters, two or more consecutive space characters are replaced by a single space character (' ').

For example, the text on left is reformatted as the text on the right:

```
  These are   three spaces, with
  two at the beginning
```

```
These are three spaces, with
two at the beginning
```

In addition to the basic normalization, `t_format` performs further processing according to some formatting options. Notice that every processing is done directly on the text string passed to the `t_format` function. Options are passed through a structure object declared as follows:

```
struct t_opts {
    int format_paragraphs;
    int format_punctuation;
    int max_line_len;
};
```

The members of `struct t_opts` define the following additional formatting operations:

- If `format_paragraphs` is non-zero (true), then the text is considered as a sequence of *paragraphs* separated by empty lines. In this case, the paragraph separations are also normalized, meaning that consecutive empty lines are replaced by a single empty line. For example, the text on left is reformatted as the text on the right:

```
This is the
 first paragraph


This is the second one
```

```
This is the
first paragraph

This is the second one
```

- If `format_punctuation` is non-zero (true), then the spacing before some punctuation characters are deleted. The punctuation characters are .,;:!? . For example, the text on the left is reformatted as the text on the right:

```
Nice !  !  ! One , two, three .
```

```
Nice!!! One, two, three.
```

- **OPTIONAL:** if `max_line_len` is greater than zero, then paragraphs are formatted into lines of at most `max_line_len` characters. For the purpose of this formatting option, paragraphs consist of all the words in groups of consecutive lines separated by empty lines. A word is a sequence of non-space characters. If a word is longer than `max_line_len`, then that word

will be placed on a single line by itself. Any other line will have at most max_line_len, also counting the spaces that separate words. For example, with max_line_len = 20 the text on left is reformatted as the text on the right:

```
One two three four five six seven eight nine

Ten eleven twelve, thirteen, fourteen, fifteen
```

```
One two three four
five six seven eight
nine

Ten eleven twelve,
thirteen, fourteen,
fifteen
```

Use the tformat.tgz test package to test your solution.

A solution is available here.

►**Exercise 56.** (50') In a source file called *words.c* or *words.cc* write a data structure in C or C++, defined by the following declarations, that can represent a sequence of lines of text and that can match those lines against a set of strings.

```
struct lines;
struct lines * lines_create();
void lines_destroy (struct lines * l);
int add_line (struct lines * l, const char * line);
int remove_line (struct lines * l, int id);
int match (struct lines * l, const char * output[], int max_output,
           const char * words);
```

lines_create() creates a *lines* object that initially contains no lines of text. Returns the null pointer in case of a memory allocation failure.

lines_destroy(l) destroys the given *lines* object, deallocating all its resources.

add_line(l,line) adds the given line of text to the given object l. Returns a unique identifier for that line, or a negative number if memory allocation fails. Notice that the given *line* object belongs to the caller, and therefore its content might change. add_line(l,line) must therefore copy that text in an internally managed data structure.

remove_line(l,id) removes the line with the given *id* from object l. Returns *true* when the given *id* is present and then removed, or *false* when the given *id* does not exist.

match(l,output,maxlen,words) takes a *lines* object l, an array *output* of char pointers of length *maxlen*, and a string *words* containing zero or more words separated by spaces. The *match* function must write into the *output* array a sequence of lines that match the words in *words*. More specifically, a line X matches a set of words $w_1, w_2, \ldots$ if every word w_i is a substring of X. Notice that w_i can appear anywhere in X, and does not have to match a whole word in X. In fact, each line in the l object is considered and therefore should be stored as a single string. For example:

```
add_line(l, "ciao a tutti");
add_line(l, "ciao ciao");
match(l, output, 100, "ti ia"); /* returns "ciao a tutti" */
```

The *match* function must return lines in the same ordered they were added. If a line X is removed and then later an identical line is added, the position of that line in the output must be last. For example:

```
int id1 = add_line(l, "ciao a tutti");
int id2 = add_line(l, "ciao ciao");
remove_line(l, id1);
id1 = add_line(l, "ciao a tutti");
match(l, output, 100, "ciao"); /* returns "ciao ciao", "ciao a tutti" */
```

Notice that the *match* function must return matching lines by copying *pointers* to those lines into the *output* array. Those pointers point to data owned by the l object, which must be valid after the *match* call returns and at least until any other call involving object l. No more than *maxlen* lines must be returned. The *match* function returns the number of lines returned through the *output* vector.

Use the words.tgz test package to test your solution.

A solution written in C++ is available here.

►**Exercise 57.** (20') An application represents a color with its red, green, and blue components (RGB) using a *color* object declared as follows:

```
struct color {
    unsigned int red;
    unsigned int green;
    unsigned int blue;
};
```

In a source file called *colors.c* or *colors.cc* write a C or C++ function declared as

```
void string_to_color (struct color *, const char *);
```

that assigns a *color* object from a string consisting of between zero and six hexadecimal characters representing three bytes, one for each of the RGB components. For example

```
struct color c;
string_to_color(&c, "A00B20"); /* c.red=160; c.green=11; c.blue=32; */
```

When the string has less than six characters, all the other characters are considered to be 0 by default. For example:

```
string_to_color(&c, "A"); /* c.red=160; c.green=0; c.blue=0; */
```

Any character that is not a valid hexadecimal digit (see below) is interpreted as a terminator. The above default rule still applies. For example:

```
string_to_color(&c, "1x"); /* c.red=16; c.green=0; c.blue=0; */
string_to_color(&c, "CIAO"); /* c.red=192; c.green=0; c.blue=0; */
string_to_color(&c, "c1ao"); /* c.red=193; c.green=160; c.blue=0; */
```

As usual, you may assume the ASCII encoding for all characters and strings. No characters outside of the ASCII set need to be considered.

Hint: recall that the hexadecimal digits are `0123456789abcdefABCDEF`, where the numbers have their usual values 0–9 while the alphabetic characters `a–f` and `A–F` have values 10–15, respectively. Each byte value consists of two hexadecimal characters. As usual, the one on the left is the most significant one. Thus `"10"` is 16 while `"01"` is 1. In general, the value of an hexadecimal number made of two digits xy is $16 \times value(x) + value(y)$. For example, the value of the hexadecimal `A1` is 161, since the value of `A` is 10, so $16 \times 10 + 1 = 161$.

Use the colors.tgz test package to test your solution.

A solution written in C is available here.

►**Exercise 58.** (50') In a source file called *styles.c* or *styles.cc* write a C or C++ program that outputs a set of text *styles*, each having a name, a foreground color, a background color, and a font name. Styles are defined by a specification read from one or more files whose names are passed as command-line arguments. When no arguments are given, the specification is read from the standard input. The program must output all the styles specified in the input in alphabetical order of style name. For example, with the following specification

```
style ticino-normal foreground #803333 background blue font fixed
style ticino-bright foreground #ff background bright-blue font italic
color bright-blue 0000ff
color blue 333380
style default
```

the program must then output the following completely specified styles

```
default: fg=0,0,0 bg=255,255,255 ft=fixed
ticino-bright: fg=255,0,0 bg=0,0,255 ft=italic
ticino-normal: fg=128,51,51 bg=51,51,128 ft=fixed
```

The input consists of one-line specifications. A *color* specification defines a symbolic name for a color. The syntax is as follows:

`color` ⟨*color-name*⟩ ⟨*color-spec*⟩

A *color-name* is a string that does not contain any space character. A *color-spec* is a string that defines an RGB color exactly as specified in Exercise 57. (You may well share the same code between this and Exercise 57.) For example, input line "`color bright-blue 0000ff`" defines a color named *bright-blue* whose red, green, and blue components are 0, 0, and 255, respectively. Similarly, an input line "`color bright-red ff`" defines a color with red, green, and blue components 255, 0, 0, respectively.

A *style* specification defines a style with a given name and any combination of *foreground*, *background*, and *font* attributes. The syntax is:

`style` ⟨*style-name*⟩ [⟨*attribute*⟩ ⟨*value*⟩] ...

A *style-name* is a string without spaces. Valid attributes and their values are:

`foreground` (`#`⟨*color-spec*⟩ | ⟨*color-name*⟩)

`background` (`#`⟨*color-spec*⟩ | ⟨*color-name*⟩)

`font` ⟨*font-name*⟩

A *font-name* is a string without spaces. A *color-spec* is preceded by a '#' (but no spaces) and has the same syntax described above. A *color-name* must correspond to the name in a *color* command present anywhere in the input. The default foreground color is `#000000` (black); The default background color is `#FFFFFF` (white); the default font is `fixed`.

The output consists of one line for each style formatted as follows:

⟨*style-name*⟩`: fg=`⟨*byte*⟩`,`⟨*byte*⟩`,`⟨*byte*⟩ `bg=`⟨*byte*⟩`,`⟨*byte*⟩`,`⟨*byte*⟩ `ft=`⟨*font-name*⟩

`fg=`, `bg=`, and `ft=` introduce the foreground color, background color, and font name, respectively. The three *byte* values represent the red, green, and blue components of a color.

If the input is valid, the program must terminate with the `EXIT_SUCCESS` status. Otherwise, with any error in the input, such as bad format or unknown color names, the program must terminate immediately with the `EXIT_FAILURE` status.

Use the styles.tgz test package to test your solution.

A solution written in C++ is available here.

►**Exercise 59.** (60') In a source file called *alpha.c* or *alpha.cc* write a C or C++ program to encode and decode data according to the "alpha" coding scheme defined below. Without command-line arguments, the program works in *encoding mode,* reading data from its standard input and writing the corresponding code onto the standard output. Vice-versa, with the `-d` command-line argument, the program works in *decoding mode,* reading a code from the standard input, and writing the corresponding data to the standard output.

The alpha encoding works as follows. The input is a stream of bytes. The output (or "code") is a stream of the 26 lowercase and 26 uppercase letters of the English language:

```
abcdefghijklmnopqrstuvwxyz    ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

As specified later, the code may contain other characters. However, those must be ignored for the purpose of decoding. The encoding rules are as follows:

- An input byte corresponding to a letter other than the letter `Q` is encoded by an identical output byte. For example, the sequence of characters "`quack`" is encoded as the same sequence of characters "`quack`".
- The input byte representing the letter `Q` is encoded by two identical bytes representing the letter `Q`. For example, the input characters "`Quote`" are encoded as "`QQuote`".
- An input byte b that is not a letter is encoded by the character `Q` followed by two letters c_1c_2 that encode b as follows: c_1 and c_2 encode numbers between 0 and 15 that are the most and least significant four bits of b, respectively. In particular, the values $0, 1, 2, \ldots, 15$ are encoded as `a`,`b`, `c`,...,`p`, respectively. For example a byte $b = 0$ is encoded as `Qaa`; a byte $b = 1$ is encoded as `Qab`; a byte $b = 16$ is encoded as `Qba`; a byte $b = 33$ representing the character '`!`' is encoded as `Qcb`. Recall that, in general, the most and least significant 4 bits of a byte b can be computed (in C/C++) as b `/ 16` and b `% 16`, respectively.
- You may assume the ASCII code. In particular, a letter (`a`...`z`, `A`...`Z`) corresponds to a single byte, and the numeric codes (byte values) of the letters in alphabetical order are consecutive numbers. That is, the code for `b` is one plus the code for `a`, and so on.

In encode mode, the program must output lines of 80 characters. The last line can of course be shorter. For example, an input stream consisting of the following four lines:

```
Mother, mother
There's too many of you crying
Brother, brother, brother
There's far too many of you dying
```

Is encoded as the following two lines:

```
MotherQcmQcamotherQakThereQchsQcatooQcamanyQcaofQcayouQcacryingQakBrotherQcmQcab
rotherQcmQcabrotherQakThereQchsQcafarQcatooQcamanyQcaofQcayouQcadyingQak
```

When running in decode mode, the program must ignore any non-letter character. So, the program must decode the following two lines exactly as the previous ones:

```
MotherQcmQcamotherQakThereQchsQcatooQcamanyQcaofQcayouQcacryingQakBrotherQcmQ--
cabrotherQcmQcabrotherQakThereQchsQcafarQcatooQcamanyQcaofQcayouQcadyingQak!!!!
```

With any error, for example with any command-line arguments other than a single `-d`, or if in decode mode the input contains an invalid code, the program must terminate immediately and return `EXIT_FAILURE`. Recall that `EXIT_FAILURE` is declared in *stdlib.h*.

Use the alpha.tgz test package to test your solution.

A solution written in C is available here.

►**Exercise 60.** (60') In a source file called *rowing.c* or *rowing.cc* write a C or C++ implementation of the structures and functions declared in the following header file:

```
#ifndef ROWING_H_INCLUDED
#define ROWING_H_INCLUDED

struct club;
struct excursion;

#ifdef __cplusplus
extern "C" {
#endif}

struct club * create_club ();
void destroy_club (struct club *);

int add_boat (struct club *, const char *, unsigned int);
int add_person (struct club *, const char *);

struct excursion * create_excursion (struct club *);
int use_boat (struct excursion *, const char *);
int add_crew (struct excursion *, const char *);

int close_excursion (struct excursion *, unsigned int);
void cancel_excursion (struct excursion *);

int get_crew_excursions (struct club *, const char *);
int get_crew_kilometers (struct club *, const char *);

int boat_available (struct club *, const char *);
int get_boat_excursions (struct club *, const char *);
int get_boat_kilometers (struct club *, const char *);

#ifdef __cplusplus
}
#endif}

#endif
```

These structures and functions are intended to be used to manage boats and excursions in a rowing club. The club has a number of boats and a number of members who join the club to participate in excursions. Each excursion must be recorded, so that the system can keep track of which boats are in use at any given time. The system also keeps track of the number of excursions and the total number of kilometers traveled by each member of the club. The detailed specification is as follows:

create_club() creates a club. Return a valid pointer to a new, empty club. Return the null pointer if memory is exhausted and therefore the creation of a new club object fails.

destroy_club(c) destroys the given club object and releases all its allocated resources.

add_boat(c,name,max_crew) adds a boat with the given name and maximum crew size to club *c*. Within a club, boats are uniquely identified by name. A boat with a maximum crew of four can take between one and four persons on an excursion. Return 1 on success. Return 0 on error or if resources are exhausted. It is an error to add two boats with the same name in the same club. It is also an error to have a maximum crew size less than 1.

add_person(c,name) adds a person to club *c*. Persons are also identified uniquely by name within a club. It is therefore an error to add two persons with the same name in the same club. Return 1

on success or 0 on error.

create_excursion(c) create an excursion object to prepare an excursion within club *c*. Return a valid pointer to a new excursion object. Return the null pointer if memory is exhausted and therefore the creation of a new excursion object fails.

use_boat(e,name) assigns the named boat to the excursion *e*. Return 1 on success. In this case, the boat remains assigned to the excursion until the excursion is closed or canceled. Return 0 if the boat does not exist or is not available because it is currently assigned to another excursion.

add_crew(e,name) assigns the named person as a crew member for the excursion *e*. Return 1 on success. In this case, the person is assigned and remains assigned to the given excursion until that excursion is canceled or closed. Return 0 if the excursion does not yet have an assigned boat, or if the maximum crew capacity of the boat has already been reached, or if the named person does not belong to the club, or if the named person has already been added to the crew of another excursion.

close_excursion(e,k) confirms that the given excursion has ended successfully with the given distance *k* traveled (in kilometers). The excursion object *e* is also destroyed and all its allocated resources are properly released. Return 1 on success. Return 0 when the given excursion has not been set up properly, with a valid boat and a crew of at least one person. In this case, the excursion object remains valid.

cancel_excursion(e) the given excursion is canceled. The excursion object *e* is properly destroyed; all its allocated resources are released.

get_crew_excursions(c,name) return the number of excursions made by the given member of the club. Return -1 in case of error, for example if the given person does not belong to the club.

get_crew_kilometers(c,name) return the total number of kilometers traveled in completed excursions made by the given member of club *c*. Return -1 in case of error, for example if the given person does not belong to the club.

boat_available(c,name) return 1 if the boat is available, or 0 if it is not available. Return -1 in case of error, if the given named boat does not exist in club *c*.

get_boat_excursions(c,name) return the number of excursions made with the given boat of club *c*. Return -1 in case of error, if club *c* does not have the given boat.

get_boat_kilometers(c,name) return the number of kilometers traveled with the given boat of club *c*. Return -1 in case of error, if club *c* does not have the given boat.

Use the rowing.tgz test package to test your solution.

A solution written in C++ is available here.

►**Exercise 61.** (70') In a source file *racing.c* write an implementation of a support system for car racing events defined by the following functions:

```
int racing_init(const char *);
void racing_delete();
unsigned count_cars(const char *, const char *, const char *,
                    const unsigned, const unsigned);
unsigned count_drivers(const char *, const char *);
unsigned get_teams();
```

The system stores cars, drivers, and teams. The data are read from a text as follows. The file is divided into sections. A section represents a car, a driver, or a team. A section starts with a line "`#car`", "`#driver`", or "`#team`", which indicates the section type. A section consists of zero or more lines, and is terminated by an "`#end`" line. Each line in each section consists of the name of an attribute, followed by the character "=", then followed by the value of that attribute. Lines are at most 1000 characters long. There might be empty lines that must be skipped. For example, the data file might look like so:

```
#car
maker=Ferrari
model=250 GTO
hp=300
#end
#driver
name=Mario Andretti
team=Lotus
#end
#driver
team=McLaren
name=Niki Lauda
#end
```

The attributes for a car are three strings, *model*, *maker*, and *driver*, and two positive integers, *hp* (horse power) and *price*. The attributes for a driver are *name* and *team*, both strings. The only attribute for a team is *name*, a string. Attributes can be given in any order and some attributes may be left undefined, as in the example above.

racing_init(f) initializes the system and reads data from file f (file name). If successful, return the total number of cars, drivers, and teams read into the system. Otherwise, return -1 in case of error, if the file is not readable, or the format is invalid, or memory is exhausted. In case of error, the system must be completely cleared.

racing_delete() clears the system, releasing all previously allocated resources.

count_cars(model,maker,driver,hp,price) returns the number of cars in the system that match the given *model*, *maker*, *driver*, *hp*, and *price* values. The null pointer is considered a wildcard value for *model*, *maker*, and *driver*; zero is considered a wildcard value for *hp* and *price*. A wildcard value matches any value in the system, including the undefined values. For example, count_cars(0,0,0,300,0) should return the number of cars with 300 hp, regardless of maker, model, driver, and price.

count_drivers(name,team) counts the number of drivers in the system with the given *name* and *team*. As for count_cars, a null pointer for *name* and *team* would match any string for name and team in the system, respectively.

get_teams() counts the number of teams in the system.

Use the racing.tgz test package to test your solution.

A solution is available here.

►**Exercise 62.** (50') In a source file *iterator.c* write an implementation of the structures and functions declared in the following header file:

```
struct iterator;

struct sequence {
    const char * begin;
    const char * end;
};

struct iterator * create_iterator();
void destroy_iterator(struct iterator *);
int set_text(struct iterator *, const struct sequence *);
int set_separators(struct iterator *, const struct sequence *);
int get_next(struct iterator *, struct sequence *);
```

A struct iterator object represents an iteration over a sequence of strings defined by a sequence of characters T ("text") and a sequence of characters S ("separators"). All the characters in S are considered *separators*, while all other characters are considered *string characters*. Together, T and S define a sequence of strings $x_1, x_2, \ldots$, where each string x_i is a maximal contiguous subsequence of T made of string characters (i.e., characters not in S). For example, if T = `abcdefghijk` and S = `ciao`, then the sequence consists of the three strings x_1 = `b`, x_2 = `defgh`, and x_3 = `jk`. Notice that the sequence might be empty, for example if T is itself empty or if T contains only separator characters. For example, the sequence is empty with T = `ciaoaoao` and S = `ciao`.

A struct sequence object represents a sequence of characters. As usual, *begin* points to the start of the sequence and *end* points one past the end of the sequence.

create_iterator() creates an iterator object. Return a valid pointer to a new iterator. Return the null pointer if memory is exhausted and therefore the creation of a new iterator object fails. For the new iterator, the default value of T is an empty sequence, and the default value of S is "`,;`" (comma and semicolon).

destroy_iterator(i) destroys the given iterator object and releases all its allocated resources.

set_text(i,T) uses the given sequence T of characters as text to start a new iteration for iterator i. This cancels any previous iteration on i. The set of strings is determined by T together with the current set of separators set by the most recent call to set_separators on i, or with the default ones if set_separators was never called on i. The given text T is owned by the caller. The iterator must not rely on the given text after this function returns. Return 1 on success, or 0 on failure.

set_separators(i,S) assigns the given set S as separators for iterator i. This setting will take effect as soon as a new sequence is defined using set_text. The given sequence S is owned by the caller. Therefore, the iterator i must not rely on that sequence after this function returns. Return 1 on success, or 0 on failure.

get_next(i,X) return the next string in i's current iteration. The string is returned through the sequence object X, in which case the return value is 1. Otherwise, if the iteration has ended or was anyway empty, the return value is 0.

Use the iterator.tgz test package to test your solution.

A solution is available here.

►**Exercise 63.** (120') In a source file *bigletters.c* or *bigletters.cc* write a C or C++ program that prints text in big letters. The program first reads a font file that defines the appearance of big-letter characters, and then writes in big letters on the standard output the text it reads from the standard input. The name of the font file can be given as a command-line argument. The default name is *FONT*. The exit value of the program is 0 on success, or 1 in case of errors (e.g., incorrect font file format, memory exhaustion, etc.).

Each line of the font file specifies the appearance in big letters of an input character as a matrix of output characters. For example, the following line specifies the appearance of the input character "A". Notice that space characters are explicitly marked as "␣":

```
A␣4␣5␣0:␣**␣*␣␣******␣␣**␣␣*
```

With this specification, an input character "A" will appear on the output as follows:

```
 **
*  *
****
*  *
*  *
```

Specifically, each line of the font file starts with a single character (A in the example above) followed by a space, then followed by three positive decimal numbers, w, h, and d, separated by a space and immediately followed by a colon character (":"). w and h are the width and total height of the matrix, respectively; d is the depth of the character (explained later). Immediately after the colon character, the line contains exactly $w \times h$ characters representing the matrix in a row-by-row layout starting from the top row. In the example above, the line contains the characters "␣**␣*␣␣******␣␣**␣␣*" (without double quotes, and again with explicitly marked spaces) that represent the five rows of the big-letter A: "␣**␣", "*␣␣*", "****", and so on. Notice that the matrix starts right after the colon character, so the space character that follows is in fact part of the matrix.

All big-letter characters of the same input line must be aligned vertically with respect to their *baseline*. The depth d specifies where the baseline is. Specifically, d is the number of rows of a big-letter character that are *below* the baseline. So, it must be $d \leq h$. In the example above, the depth of the big-letter A is 0, which means that the baseline is right under the bottom row of the matrix. Here is another example:

```
p␣4␣4␣1:***␣*␣␣****␣*␣␣␣
```

The input character "p" must then appear in the output as:

```
***
*  *
***
*
```

The "p" character has a depth $d = 1$ and a total height $h = 4$. This means that the character has $h - d = 3$ rows above the baseline, and $d = 1$ row below the baseline. So, with the given definitions of the characters A and p, the big-letter output of an input string "App" must be:

```
 **
*  *
**** ***  ***
*  * *  * *  *
*  * ***  ***        baseline (virtual line, does not appear in the output)
      *    *
```

The font file may specify any number of characters. In fact, it might even be empty. When a character c is not specified in the font file, its default big-letter appearance is a single character c, with width $w = 1$, height $h = 1$, and depth $d = 0$. In other words, the default appearance of a character c is equivalent to a line "c 1 1 0:c" in the font file. So, for example, if the font file contains the only two specification lines given in the examples above (for A and p), then the output for the input string "Appropriate" must be:

```
 **
*  *
**** ***  ***      ***
*  * *  * *  *     *  *
*  * ***  ***  r o ***  r i a t e
     *    *        *
```

When printing an input line, which you may assume is no more than 1000 characters long, the corresponding output characters in big letters must be separated horizontally by one space, as in the examples above. The total height of the big-letter output corresponding to one input line is a fixed number of lines that does not depend on the content of the line, and instead depends on the font, as we detail below. Furthermore, there is always an empty line separating the big-letter output of two consecutive input lines.

The height of the output of a single input line is the maximal height above the baseline of any character, plus the maximal depth of any character. More specifically, for each big-letter character with height h and depth d, including the default ones, let $h^* = h - d$ be the height of the character above the baseline, and let H^* and D be the maximal values of h^* and d over all characters. Then, a single non-empty input line results in an output of exactly $H^* + D$ lines. So, again with a font file consisting of the two examples above, the two consecutive input lines "`Appropriate`" and "`power to the people`" must be printed as follows. The symbol "|" explicitly indicates the end of each output line.

```
 ** |
*  *|
**** ***  ***      *** |                          ← 6 lines for the first input line
*  * *  * *  *     *  *|
*  * ***  ***  r o ***  r i a t e|
     *    *        *   |
|                                                 ← empty, separation line
|
|
***                         ***      *** |        ← 6 lines for the second input line
*  *                        *  *     *  *|
***  o w e r   t o   t h e  ***  e o ***  l e|
*                           *        *   |
```

Notice that each input line results in six lines of output. This is because the maximal height above the baseline is 5 (character "`A`") and the maximal depth is 1 (character "`p`"). Then, an additional empty line separates those two groups of six output lines. Notice also that all output lines must terminate at the end of the right-most big-letter character matrix that occupies that line.

Use the bigletters.tgz test package to test your solution.

A solution is available here.

►**Exercise 64.** (120') In a source file *ochecker.c* or *ochecker.cc* write a C or C++ implementation of an *Output Checker* system specified as follows. The system is intended to be used to check that a program produces a specific given output. At a high level, the program writes its output using a struct ochecker object as it would use a FILE object of the C and C++ standard input/output library (declared in *stdio.h*). For example, the checker object provides the oc_putc function, to output a single character, that is equivalent to the fputc function of the standard library. The checker output functions (oc_putc, oc_puts, oc_write, oc_close) are equivalent to their standard I/O counterparts (fputc, fputs, fwrite, fclose) except for their return values. The standard I/O functions return whether or not the output operation was successful. Instead, the return value of the checker functions indicates whether the output operation is *correct.* The output is considered correct if it is consistent with a reference output (a sequence of bytes) that is preset within a checker object with the oc_open_file, oc_open_str, and oc_open_mem functions.

A newly created checker object is in the *closed* state, defined by the OC_CLOSED constant. In this state, the checker provides no valid information. The checker can then be opened by setting the reference output from either a file, a C string, or a memory buffer. After a successful open operation, the checker is in the *open* state (OC_OPEN). In the open state, the checker can take and therefore check output data from the program with oc_putc, oc_puts, oc_write, and oc_close. The checker remains in the *open* state as long as the output is consistent with the reference output. As soon as there output differs, the checker goes into the *error* state (OC_ERROR), where every output operation will return *false*. The checker can always be reopened.

In the *open* state, the checker provides the current position in the output stream with the functions oc_position, oc_line, and oc_column. Positions start from 1. In the *error* state, oc_position, oc_line, and oc_column indicate the position of the first byte where the output of the program differs from the reference output.

The following declarations define the interface of the output checker:

```
struct ochecker;

struct ochecker * oc_create ();
void oc_destroy (struct ochecker * oc);

int oc_open_file (struct ochecker * oc, const char * fname);
int oc_open_mem (struct ochecker * oc, const char * begin, const char * end);
int oc_open_str (struct ochecker * oc, const char * s);

int oc_putc (struct ochecker * oc, char c);
int oc_puts (struct ochecker * oc, const char * s);
int oc_write (struct ochecker * oc, const char * buf, size_t len);
int oc_close (struct ochecker * oc);

typedef enum { OC_CLOSED = 1, OC_OPEN = 2, OC_ERROR = 3 } oc_status_t;

oc_status_t oc_status (const struct ochecker * oc);

size_t oc_position (const struct ochecker * oc);
size_t oc_line (const struct ochecker * oc);
size_t oc_column (const struct ochecker * oc);
```

Thus you must implement the struct ochecker type, with its constructor and destructor functions, oc_create and oc_destroy, as well as all its access functions detailed below:

- *Functions to open the checker and set the reference output.* oc_open_file(x, name) opens the x checker and sets the reference output to the content of the file with the given name; with oc_open_mem(x, begin, end) the reference output is the content of the memory buffer starting at *begin* and ending right before *end*; and with oc_open_str(x, s), the reference output is the zero-terminated string s. oc_open_file may fail if the given file does not exist or is not readable. oc_open_str and oc_open_mem may fail if there is insufficient memory to store the

reference output. In case of failure, the status must be OC_CLOSED, otherwise the status is OC_OPEN. In any case, any previously set reference output is discarded.

- *Functions to check an output operation against the reference output.* The three functions oc_putc(x,c), oc_puts(x,s), and oc_write(x,buf,len) check that the output of the single character *c*, the string *s*, and *len* bytes pointed by *buf*, respectively, is consistent with the reference output of checker *x*. That is, check that the single character *c*, the string *s*, and *len* bytes pointed by *buf*, respectively, are at the current position in the reference output. The function oc_close() checks that closing the output would be consistent with the reference output, meaning that the current position of the reference output is at the end of the output. All these functions return *true* if their output operation is correct. If the output is incorrect, then the *oc* checker switches to the *error* state and the return value is *false.*

- *Function to obtain the status of an output checker.* oc_status(x) returns the state of the *x* checker. The output is OC_CLOSED, OC_OPEN, or OC_ERROR.

- *Functions to obtain the current position in the reference output.* oc_position(x), oc_line(x), and oc_column(x) return the byte position, the line number, and the column number, respectively, in the reference output for checker *x*. All positions start from 1. So, for example, after a call oc_open_str(x,"ciao\namici"), the positions you oc_position(x), oc_line(x), and oc_column(x) functions must return 1. Then, after a call oc_puts(x,"ciao\n"), oc_position(x) returns 6, oc_line(x) returns 2, and oc_column(x) returns 1.

Use the ochecker.tgz test package to test your solution.

A solution is available here.

►**Exercise 65.** (60') Write a C program in a source file named *tabops.c* that reads data from one or more files or from standard input if no files are specified. The program interprets each input line as a sequence of data items based on a given set of delimiter characters. The program then writes on standard output those data items based on a given format string.

The delimiters, the format string, and the input files are specified through command-line arguments. Specifically, `format=`*format-string* sets the format string; `delim=`*delimiters* sets the delimiters; and any other command-line argument is interpreted as an input file name, which the program reads and processes according to the current format and delimiters.

By default, the format string is "`@a`" and the delimiters are "` \t`". If no input file names are given, the program reads from standard input. For example, consider the following command line:

```
./tabops f1 format=@b,@a f2 delim=/ format=@c f3 f4
```

Here, the program reads and processes file *f1* with the default format and delimiters, then processes *f2* with the format "`@b,@a`" and default delimiters, and then processes *f3* and *f4* with format "`@c`" and delimiters "`/`".

The input is read line by line. A line, at most 1024 characters, is interpreted as a sequence of data items separated by one or more delimiter characters. The end-of-line character is not considered part of the line and therefore of any data item. For example, with delimiters "`,.;`", the line "`Apples,;5;3`" is parsed into three items: `Apples`, `5`, and `3`.

For each line, the program outputs the format string where placeholders like `@a`, `@b`, etc., are replaced with corresponding data items from the line based on alphabetical order (`@a` for the first item, `@b` for the second, etc.). For example, given the line `Apples,5,3` with delimiters `","` and the format string "`@b,@a`", the program outputs `5,Apples`.

A placeholder `@`*x* referring to a non-existent data item or where *x* is any character other than a lowercase letter (`a`–`z`) is ignored. For instance, if there are three items `Apples`, `5`, `3` and the format string is "`@b@,@a--@x`", the output will be `5Apples--`.

Use the tabops.tgz test package to test your solution.

A solution is available here.

►**Exercise 66.** (60') Write an extension of the program of Exercise 65 in a source file named *tabops2.c* that can process data items as numbers. The format string can now contain arithmetic expressions enclosed in curly braces `{}` that are evaluated using a simple stack-based calculator.

The calculator processes each lowercase letter (`a`–`z`) as a reference to an input data item, pushing its integer value onto the stack. If the item does not exist or is not a valid integer, it is treated as 0. For instance, given the items `Apples`, `5`, and `3`, the expression `{bcxa}` pushes 5, 3, and 0 onto the stack.

The characters `+`, `-`, `*`, and `/` pop two values from the stack, apply the corresponding arithmetic operation to them, and push the result onto the stack. If there are less than two values on the stack, the characters `+`, `-`, `*`, and `/` are ignored.

Non-negative integer literals push their value onto the stack. Any other character is ignored. The maximal stack size is 100. Any operation that would overflow the stack is ignored.

The value of an expression is the value of the top of the stack. An expression that terminates with an empty stack is ignored, resulting in an empty string.

Examples: below are two examples, both with items `Apples`, `5`, and `3`. For each example, we show the format string, followed by an arrow (⟶) and the corresponding output:

```
{b},{bc/},{2b*},{10c*b2*-} ⟶ 5,1,10,20
{ac-}@a{2 2*}, {b+c-}bc{10:+:20:+:}::{-123} ⟶ -3Apples4, 2bc30::123
```

Use the tabops2.tgz test package to test your solution.

A solution is available here.

►**Exercise 67.** (40') In a source file *tdecode.c* or *tdecode.cc*, implement a C or C++ program that decodes a file encoded as follows.

For our purposes, a *text file* is a file consisting of ASCII characters, meaning bytes between 0 and 127 interpreted as characters according to the ASCII code. The lines of a text file are at most 1000 characters long. We define a compression encoding for such text files. Since the original file ("plaintext") does not contain bytes between 128 and 255, the idea is to use those byte values to represent entire words. A *word* is a maximal contiguous sequence of lowercase or uppercase letters as defined by the *isalpha* function of the standard library. A *code* defines which byte maps to which word, and vice-versa. In essence, a code consists of *byte–word* pairs, which are listed at the beginning of the encoded file.

Specifically, if a code exists, the first pair (b_1, w_1) is stored in the encoded file with b_1 as the first byte of the file, immediately followed by word w_1 in plaintext. A second pair (b_2, w_2) would immediately follow the first one, and so on. The list is terminated by an end-of-line character ('\n'). For example, if the code consists of the two pairs (128, `hello`) and (129, `world`), then the beginning of the encoded file would contain the following bytes:

128	104	101	108	108	111	129	119	111	114	108	100	10
	'h'	'e'	'l'	'l'	'o'		'w'	'o'	'r'	'l'	'd'	'\n'

The rest of the encoded file then consists of the plaintext content, except that every word in the given code is replaced by the corresponding byte. The example below shows a plaintext file (left-hand side) and the corresponding encoded file (right-hand side). In the example, we show bytes above 127 as boxed numbers.

original ("plaintext") file	*encoded file*
`hello, my friends!`	128`hello`129`world`
`hello, world!!`	128`, my friends!`
`hello, world!!!`	128`,` 129`!!`
	128`,` 129`!!!`

Notice that, if the first byte of the encoded file is less than 128 (it is an ASCII character) then that means that no code is given. In this case, the encoded file is identical to the plaintext.

The `tdecode` program must read the encoded file from standard input and write the decoded plaintext file on standard output. In case of error, the program must print an error message on standard error, and immediately exit with status `EXIT_FAILURE`.

Use the tdecode.tgz test package to test your solution.

A solution is available here.

►**Exercise 68.** (80') In a source file *tencode.c* or *tencode.cc*, implement a C or C++ program that encodes a text file using a *byte-word* code described in Exercise 67. tencode must use a *minimal* code, meaning that the code must produce an encoded file of minimal size compared to other codes. Therefore, a word is in the code only when that leads to a reduction of the encoded file. Furthermore, since there are only 128 available byte values (from 128 to 255), tencode must prioritize and therefore choose the words that would lead to the maximal size reduction for the encoded file.

In order to define the code, tencode computes the *gain* of the encoding of a particular word as follows. Let l_w be the length of word w, and c_w be the number of times that w appears in the plaintext. Therefore, w uses $c_w l_w$ bytes of the input, and encoding w with some single byte value would use $l_w + 1 + c_w$ bytes of the output, $l_w + 1$ for the initial mapping plus c_w for the content. The gain g_w of encoding word w is therefore:

$$g_w = c_w l_w - l_w - 1 - c_w$$

Thus a valid code includes only words with a *positive* gain, and among those, the ones with higher gains. Specifically, tencode must assign the available code bytes (128–255) to words in decreasing order of gain. And to make the encoding deterministic, words that have the same gain must be included in the code in alphabetical order.

The program reads the plaintext from standard input, and writes the encoded file on standard output. In case of error, the program must print an error message on standard error, and immediately exit with status EXIT_FAILURE.

Hints:

To define the minimal code, tencode must compute the gain of every word in the file. This means that the encoding—where you actually use the code—can start only after you have read the whole file. However, since you are reading the file from standard input, you can't go back to the beginning of the file. In practice, this means that you have to store the whole content of the file in memory.

You must then rank the words by the potential gain of their encoding. This is very easy to do in C++. For example, if you store each word together with its gain in a structure:

```
struct word_gain {
    std::string word;
    unsigned gain;
};
```

then you can store all potentially useful words in, say, a vector W, and use the standard *sort* function with the desired ordering:

```
std::sort (W.begin(), W.end(), wg_less_than);
```

where a function bool wg_less_than(const word_gain & a, const word_gain & b) defines the ranking criterion. The function is analogous to a *less-than* comparison, so the function must determine whether a precedes b in the desired sorted order.

Use the tencode.tgz test package to test your solution.

A solution is available here.