Exercises of programming v. 0.4

Andrea Marin

April 7, 2015

1 Exercises of cycles

This section contains some exercises that aim at improving the skills of the students on the usage of cycles. The exercises must be solved without using any C library but the `stdio.h` and adopting the cycle which is mostly suitable to the proposed solution. In order to achieve this, keep in mind what follows:

- You use the `for` cycle when, once the values of the variables are know, you can predict the number of iterations just by reading the heading of the cycle;
- You use the `do .. while` cycle when the `for` is inappropriate and you are sure that the cycle's block must be executed at least once;
- You use the `while` cycle in all the other cases.

Do not use the *jumping* instructions as *break*, *continue* and *return* to abort a cycle.

1.1 Co-prime numbers

Write a C program that reads two integers from the standard input and decides if they are coprime. The program must write the answer on the standard output. Two numbers are coprime if their only common positive divisor is 1.

Solution. First let us understand the meaning of coprimality. 5 and 12 are coprime because they do not share any divisor but 1 and -1; indeed the same holds for -5 and 12. If we consider 21 and 27 they are not coprime because they share 3 as a common divisor. Notice that 1 is coprime with every number (included 0), but 0 is coprime only with 1.

The easiest solution consists in using the *universal property* pattern. Indeed, if the two input numbers (say n_1 and n_2) are positive, we can reformulate the coprime condition as: *all the numbers in the interval* $[2, \min(n_1, n_2)]$ *are not simultaneously divisors of* n_1 *and* n_2. However, we must consider the case of negative numbers which is easy because we can just work with the absolute values. Finally, the case of $n_1 = 0$ or $n_2 = 0$ can be treated separately; however observe that if we consider the divisors in the interval $[2, \max(n_1, n_2)]$, then we immediately solve the problem when one of the two numbers is 0 and the other is not. So the case $n_1 = 0$ and $n_2 = 0$ must be considered separately anyway. The solution is shown in Table 1. A more efficient solution relies on the Euclide's algorithm for the computation of the largest common divisor. Also in this case pay attention to the case when one or both the input numbers are 0.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n1, n2;    /*the two input numbers*/
   int coprime;   /*Are they coprime?*/
   int div;       /*Candidate common divisor*/


   printf("Give the first number: ");
   scanf("%d", &n1);
   printf("Give the second number: ");
   scanf("%d", &n2);
   /* compute the absolute value */
   if (n1 < 0)
      n1 = -n1;
   if (n2 < 0)
      n2 = -n2;

   /*Universal property: first assume it is true!*/
   coprime = 1;

   /*Start from divisor  2*/
   div = 2;
   while ( (div <= n1 || div <= n2) && coprime) {
      if(n1 % div == 0 && n2 % div == 0)
         coprime = 0;    /*Counterexample has been found!*/
      div = div + 1;
   }

   /*case of both number equal to zero*/
   if (n1 == 0 && n2 == 0)
      coprime = 0;

   /*output*/
   if (coprime)
      printf("The numbers are coprime.\n");
   else
      printf("The numbers are not coprime.\n");

   return 0;
}
```

Table 1: Decision of coprimality between two integers based on testing a universal property.

1.2 Exercise on sequence, give the sum and the quantity of numbers

Write a C program that reads from the standard input a sequence of numbers; the program must stop reading from the input sequence when it finds two equal consecutive numbers. Compute the sum and the quantity of all the read numbers.

Solution. The easiest way to address the problem consists in using the *sequence pattern*, i.e., we use variable *prec* to store the last read number and variable *cur* to store the current input. At each iteration of the cycle the value of *cur* is stored in *prec* and *cur* is newly read. The other patterns to apply is that of the *accumulator* to store the partial sum of the input sequence and of the *counter* to store the partial counting of the read numbers. Notice that we must do at least two read operations from the standard input, therefore, we do one reading before the cycle block and then we choose the `do .. while` cycle to ensure a second reading.

1.3 Printing the odd numbers in $[1, 1000]$

Write on standard output all the odd numbers included in the interval [1, 1000] with a newline every 10 printed numbers.

Solution. The exercise is very simple, the key-point here is counting the number of written numbers and adding a new line printing every ten. The solution is shown in Table 3. Try to rewrite the solution avoiding the use of variable *printed*.

1.4 Compute the integer $r-$th root of a positive number

Write a C program that reads from standard input two positive integers, n and r, and compute the floor approximation of the $r-$root of n:

$$x = \lfloor \sqrt[r]{n} \rfloor \qquad r, n \in \mathbb{N}^+$$

Solution. The problem can be a little difficult because we do not want to use the C Mathematics library. Let us imagine we are able to compute in some way x^r. In this case the exercise can be reformulated as follows: *find the smallest x such that $x^r > n$*, i.e., we can see the solution as an application of the *find the edge* pattern: so we start from $x = 1$ and we try all the successive values until we find the first for which $x^r > n$. Then the solution is $x - 1$. However computing x^r is easy because it is sufficient to multiply x by itself r times. The solution is depicted by Table 4, observe carefully where the initialisation of the variables are put.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int sum, count; /*accumulator and counter*/
   int cur, prev; /*current and previous values of the seqeunce*/

   scanf("%d", &cur);
   sum = cur;
   count = 1;
   do {
      prev = cur;
      scanf("%d", &cur);
      count = count + 1; /*counter*/
      sum = sum + cur; /*accumulator*/
   } while (prev != cur);
   printf("Read numbers: %d\nSum: %d\n", count, sum);

   return 0;
}
```

Table 2: Read a sequence of integers and stop when two consecutive numbers are equal. Compute the sum and count the quantity or read input.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
        int i; /*odd number to be printed*/
        int printed; /*counting the printed numbers*/

        printed = 0;
        for (i = 1; i < 1000; i = i + 2) {
                printf("%d ", i);
                printed = printed + 1;

                if (printed % 10 == 0)
                        printf("\n");
        }
        return 0;
}
```

Table 3: Printing the odd numbers in $[1, 1000]$; new line every 10.

```
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n, r;  /*input variables*/
   int root, pow;
   int i;

   printf("Write n: ");
   scanf("%d", &n);
   printf("Write r: ");
   scanf("%d", &r);

   if (n > 0 && r > 0) {
      root = 1;
      pow = 1;
      while (pow <= n) {
         root = root + 1;
         pow = 1;
         for (i = 0; i < r; i++)
            pow = pow * radice;
      }

      printf("The floor approximation of
        the %d-root of %d is %d.\n", r, n, radice - 1);
   } else
      printf("Invalid input.\n");

   return 0;
}
```

Table 4: Compute $\lfloor \sqrt[r]{n} \rfloor$.

2 Recursion

2.1 From iterative to recursive

We desire to compute the square root of a float number $n > 1$ according to the following algorithm:

1. `inf` takes value 1
2. `sup` takes value n
3. `x` takes value $(inf + sup)/2$
4. If $|n - x^2| < t$ then x is the approximating square root of n
5. If $x^2 > n$ then `sup` takes value x and jump to step 3
6. If $x^2 < n$ then `inf` takes value x and jump to step 3

where t denotes the tolerance. Write the recursive function `float mysqrt(float n, float inf, float sup, float t)` and a function call that computes the square root of 14.3 with a precision of $t = 0.00001$.

Solution The only difficulty in the exercise is understanding the signature of function `mysqrt`. We may interpret it as: *Find the square root inside the interval* $[inf, sup]$ *with tolerance* t. Therefore the solution is that shown in Table 5.

3 Arrays

3.1 Given an ordered array eliminate duplicated elements

Given an ordered array of integers and its size, eliminate the duplicated elements. The resulting array must be instantiated dynamically with the minimal size. The function returns the instantiated array and its size.

Solution The solution is rather simple and consists of two phases: first we count the number of different elements in the input vector, the we instantiate the resulting vector in the heap and copy the elements. We also provide a `main` in Table 6.

3.2 Eliminate duplicated elements from a non-ordered array

Write the function:
`int different(int input[], int size, int **vetout);`
while given an array `input` of integers with dimension `size` creates a vector in dynamic memory that contains all the elements of `input` repeated once. Function `different` returns the size of the modified array.

Solution We use array `sup` to spot the repetitions of the elements in `input`. `sup` is allocated in the dynamic memory with the same size of `input`. An element 1 denotes a fresh element, 0 a repeated one. The solution is shown in Table 7.

```c
#include<math.h>
#include<stdio.h>

float mysqrt(float n, float inf, float sup, float t) {
  float x = (inf+sup)/2.0;
  if (fabs(n-x*x) < t)
    return x;
  else
    if (x*x>n)
      return mysqrt(n, inf, x, t);
    else
      return mysqrt(n, x, sup, t);
}

int main() {
  float res;
  res = mysqrt(14.3, 1.0, 14.3, 0.00001);
  printf("The square root of 14.3 is approximatively %f\n", res);
  return 0;
}
```

Table 5: Recursive computation of the square root of a float number.

```
/*assume size>0*/
int deleteduplicated(int* vect, int size, int** pres, int *psize) {
  int i;
  *psize = 1;
  for (i=1; i<size; i++)
    if (vect[i] != vect[i-1])
      (*psize)++;
  *pres = (int*) malloc (sizeof(int)*(*psize));
  if (*pres) {
    int j=1;
    (*pres)[0]=vect[0];
    for (i=1; i<size; i++) {
      if (vect[i] != vect[i-1]){
        (*pres)[j] = vect[i];
        j++;
      }
    }
    return 1;
  }
  else
    return 0;
}

int main() {
  int vector[] = {5, 5, 9, 9, 9, 10, 14, 15, 15, 20};
  int* result;
  int dimensione;

  if (deleteduplicated(vector, 10, &result, &dimensione)) {
    int i;
    for (i=0; i<dimensione; i++)
      printf(" %d ", result[i]);
  }
  free(result);
  return 0;
}
```

Table 6: Eliminates duplicated elements from an ordered vector.

```
int different(int input[], int size, int **vetout) {
   int *sup = (int*)malloc(sizeof(int) * size);
   int i,j;
   int unequal = 0;

   /*initialisation of the vector of duplicates*/
   for (i=0; i<size; i++)
      sup[i] = 1;

   /*marking of duplicates*/
   for (i=0; i< size; i++)
      if (sup[i]==1) {
         unequal++;
         for (j=i+1; j<size; j++)
            if (input[i]==input[j])
               sup[j] = 0;
      }

    /*output array*/
    *vetout = (int*)malloc(sizeof(int)*unequal);

    /*copy of the elements in the output array*/
    j=0;
    for (i=0; i<size; i++) {
       if (sup[i] == 1) {
          (*vetout)[j]=input[i];
          j++;
       }
    }

    free(sup);
    return unequal;
}

int main() {

   int vect[] = {3,4,2,3,6,7,3,4};
   int *ris;
   int sizeris;


   sizeris = different(vect, 8, &ris);

   for (i=0; i<sizeris; i++)
      printf("\n %d ",ris[i]);

   free(ris);
   return 0;
}
```

Table 7: Eliminate duplicated from a non-ordered array

4 Recursion on arrays

4.1 Recursive upcase

Write a C subroutine that upcases a string recursively.

Solution. The key-points to provide a simple solution are:

- if `str` is a non-empty string then `str + 1` is a valid string
- when the subroutine is called the parameters are passed by value. However, in this case what is copied is address of the first element of the string; therefore, any change to the referenced object (a char) represents a side effect of the function call.

The solution is displayed in Table 8.

4.2 String anagrams

Given a string composed of different chars, write a recursive function that prints on standard output all its anagrams.

Solution Suppose we have a string of length n and that we are able to generate all the anagrams of strings of length $n-1$, who can we solve our problem? The answer is rather simple. Indeed, we can swap the first string char with all the remaining ones and then generate the anagrams of a shorter string. Suppose we wish to generate the anagrams of the string `''ABCD''`. These are:

- `'A'` + anagrams of `'BCD'`
- `'B'` + anagrams of `'ACD'`
- `'C'` + anagrams of `'BAD'`
- `'D'` + anagrams of `'BCA'`

The solution is shown in Table 9

4.3 Creation of a maze

In this exercise we want to create a maze of dimension $N \times M$, where N and M are taken as input values. The starting point is the left-bottom corner and the arrival point is the top-right corner. All the cells must be reachable from the starting point. The program must store the maze in a data structure and display it on the standard output.

Solution We first discuss the encoding of the maze. Ideally, we would like to have a $N \times M$ bi-dimensional array allocated in the dynamic memory. We will linearize the data structure in order to simplify the memory management operations. Each cell contains an encoding of the surrounding walls: *Left*, *Up*, *Right*, *Down*. Moreover, we would like to have a marking of a cell, i.e., *Visited*. A cell can have any combination of these attributes, e.g., it may have the left, up, and right walls and be visited. We exploit the binary encoding shown in the following table:

```
void upcaseric(char* str) {
   if (*str!='\0') {
      if (*str >= 'a' && *str <='z')
         *str = *str - 'a' + 'A';
      upcaseric(str+1);
   }
}
```

Table 8: Recursive upcase of a string.

```
void swap(char* c1, char* c2) {
   char c = *c1;
   *c1 = *c2;
   *c2 = c;

}

void permute(char* str, char* from) {
   if (!*from) {
      printf("%s\n", str);
   }
   else {
      int l = strlen(from) - 1;
      int i;
      for (i=0; i<=l; i++) {
         swap(from, from+i);
         permute(str, from+1);
         swap(from, from+i);
      }
   }
}
```

Table 9: Recursive generation of all the anagrams of a string.

Attribute	Binary enc.	Base 10
Left	00001	1
Up	00010	2
Right	00100	4
Down	01000	8
Visited	10000	16

Following this approach we can combine the attributes by using simple binary operations: |, & and $\sim$. For instance the visited cell with left, up and right border has value: `Left|Up|Right|Visited`.

We adopt the following recursive algorithm to construct the maze from a starting point r, c:

1. Mark cell r, c as visited
2. Let $\mathcal{R}$ be the set of unvisited neighbours of r, c
3. If $\mathcal{R} = \emptyset$ terminate
4. Pick a random element (r', c') in $\mathcal{R}$
5. Create the maze from (r', c')
6. Return to step 2

Table 10 shows the definition of the constants and the subroutine which allocates the data structure. Moreover we use a function to linearize the coordinates. The recursive algorithm is implemented by function `createroutes` shown in Table 11. The array `positions` contains the candidate positions for the next visit. The North, South, West and East cells are denoted by codes 0, 1, 2, 3, respectively. In principle there are 4 available cells but some of them are not valid because they are outside the field or because they have already been visited. The cycle `while (i<available)` counts the number of usable cells to keep the caving and ensure that they are all contained in the first positions of the array `positions`. Then, we choose a valid direction via `u`. According to the chosen direction, we remove the maze walls. The recursive call follows.

4.4 Longest sequence of identical chars

Implement the following *recursive* function:
`char* consec(char str[], int *lung);` that returns the address of the string char where the first longest sequence of identical chars start. In `*lung` store the length of longest sequence found. If the string is empty, then return `NULL`. Examples:

- In the string *ippodromo* return the address of the first 'p' and `*lung` is 2
- In the string *aabbbcdddbbe* return the address of the first 'b' and `*lung` is 3

Solution The solution is based on the following recursive scheme:

- If the string is the empty string, then return NULL and `*lung` is 0
- Otherwise count the length of the identical chars in the string prefix and compare the length with the result of the recursive call on the remaining string. Set the result according to the outcome of the comparison.

The code is shown in Table 12.

```
#define EMPTY 0
#define LEFT 1
#define UP 2
#define RIGHT 4
#define DOWN 8
#define VISITED 16

int coord(int r, int c, int dimc) {
   return r*dimc+c;
}

void createmaze(int** maze, int dimr, int dimc) {
   int i, r, c;

   *maze = (int*) malloc((dimr * dimc) * sizeof(int));

   for (i=0; i<dimr*dimc; i++)
      (*maze)[i] = LEFT | UP | RIGHT | DOWN;

   /*select the bottom left corner as starting point*/

   r = dimr-1;
   c = 0;

   (*maze)[coord(r, c, dimc)] |= VISITED;
   (*maze)[coord(r, c, dimc)] &= (~DOWN);

   srand ( time(NULL) );

   createroutes(*maze, dimr, dimc, r, c);

   (*maze)[coord(0,dimc-1,dimc)] &= (~UP);

   /*remove visited information*/
   for (i=0; i<dimr*dimc; i++)
      (*maze)[i] &= (~VISITED);

}
```

Table 10: First part of the maze creation problem.

```
void createroutes(int *maze, int dimr, int dimc, int r, int c) {
   int i, u;
   /* N=0, S=1, W=2, E=3*/
   int positions[4][3] = {{0, r-1, c}, {1, r+1, c}, {2, r,c-1}, {3, r,c+1}};
   int nr, nc;
   int available = 4;
   while (available>0) {
      i = 0;
      while (i<available) {
         if (positions[i][1]<0 || positions[i][1]>=dimr ||
               positions[i][2]<0 || positions[i][2]>=dimc ||
               (maze[coord(positions[i][1],positions[i][2],dimc)]&VISITED)) {

            positions[i][0] = positions[available-1][0];
            positions[i][1] = positions[available-1][1];
            positions[i][2] = positions[available-1][2];

            available--;
         }
         else
            i++;
      }

      if (available) {
         u = rand()%available; /*random number between 0 and available-1*/
         maze[coord(positions[u][1],positions[u][2],dimc)] |= VISITED;
         /*fix the walls*/
         switch(positions[u][0]) {
            case 0:  /*Going North*/
               maze[coord(r, c, dimc)] &= (~UP);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~DOWN);
               break;
            case 1: /*Going South*/
               maze[coord(r, c, dimc)] &= (~DOWN);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~UP);
               break;
            case 2: /*Going West*/
               maze[coord(r, c, dimc)] &= (~LEFT);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~RIGHT);
               break;
            case 3: /*Going East*/
               maze[coord(r, c, dimc)] &= (~RIGHT);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~LEFT);
         }

         createroutes(maze, dimr, dimc, positions[u][1], positions[u][2]);

         positions[u][0] = positions[available-1][0];
         positions[u][1] = positions[available-1][1];
         positions[u][2] = positions[available-1][2];

         available--;
      }

   }
}
```

Table 11: Recursive algorithm for the maze creation.

```c
char* consec(char str[], int *lung) {
   char *pc = str;

   if (*str == '\0') {
      *lung = 0;
      return NULL;
   }
   else{
      int cl = 1, ric;
      char* pstr;
      while (*str == *(str+1)) {
         cl++;
         str++;
      }
      pstr = consec(str+1, &ric);
      if (ric>cl) {
         *lung = ric;
         return pstr;
      }
      else {
         *lung = cl;
         return pc;
      }
   }
}

int main(){
   char str1[] = "ippodromo";
   char str2[] = "aabbbcdddbbe";
   char *pc;
   int l;

   pc = consec(str1, &l);
   printf("Sequenza di lunghezza %d: %s\n", l, pc);


   pc = consec(str2, &l);
   printf("Sequenza di lunghezza %d: %s\n", l, pc);
   return 0;
}
```

Table 12: Longest sequence of consecutive chars.

```
typedef struct cell{
        int digit;
        struct cell * next;
} t_cell;

typedef t_cell* t_list;
```

Table 13: Types for managing the numbers as lists.

5 Exercises on simple lists

5.1 Numbers as lists

Write a subroutine that transforms a positive integer number given as parameter into a list of integers. Each node of the list contains a decimal digit of the number, the empty list represents the number 0. The order of the digit must be defined such that the original number (if different from 0) can be read by printing the list elements from the last to the first.

Solution. The creation of the list from the number is a simple application of appending a new cell to the end of the list. The definition of the types is given in Table 13 and the solution is given in Table 14 (iterative) and Table 15 (recursive).

5.2 Sum the lists

Given the number encoding defined by Exercise 5.1, define a subroutine that sums two numbers represented as lists. Do not convert the lists into the standard int type.

Solution. We propose a recursive solution that basically implements the algorithm for the sum of decimal numbers. The function requires a third parameter which is the carry, so when called to sum two lists this should be set to 0. The recursion takes advantage of the number representation that puts the less significant digit in the head of the list. The solution is shown in Table 16 and the data types are those defined in Table 13.

5.3 Remove instances of an element

Given a list of `char`, remove all the cells that contain the value `'a'` of `'A'`.

Solution The problem can be easily addressed with the recursive function illustrate in Table 17.

5.4 Difference between list and vector

Given a list of integers ℓ, a vector of integers v and its dimension, write a function that returns the elements which are contained by ℓ but not by v.

Solution We propose a recursive solution of the problem. For each cell of the list, we check if its element is contained in the vector; if this case the function returns the list generated by the tail of ℓ and the same vector v. Otherwise, it generates a new cell which is the head of a list whose tail is the given by the same recursive call. The solution is shown in Table 18.

```
t_list create_list(unsigned int number) {
   t_list result=NULL;
   t_list pc = NULL;

   int digit;

   while (number > 0) {
      digit = number % 10;
      number = number / 10;

      t_list newcell = (t_list) malloc(sizeof (t_cell));
      newcell->digit = digit;
      newcell->next = NULL;
      if (pc) {
         pc->next = newcell;
         pc = pc->next;
      }
      else { /*first element*/
         pc = newcell;
         result = pc;
      }
   }
   return result;
}
```

Table 14: Creation of list from a positive number (iterative solution).

```
t_list create_list(unsigned int number) {
   if (number == 0)
      return NULL;
   else {
      t_list newcell = (t_list) malloc(sizeof(t_cell));
      newcell->digit = number % 10;
      newcell->next = create_list(number/10);
      return newcell;
   }
}
```

Table 15: Creation of list from a positive number (recursive solution).

```
t_list sum_list(t_list l1, t_list l2, int carry) {

   if (l1==NULL && l2==NULL && carry ==0)
      return NULL;
   else {

      t_list next1 = NULL;    /*next digits*/
      t_list next2 = NULL;
      int next_carry;

      t_list result = (t_list)malloc(sizeof(t_cell));
      result->digit = carry;

      if (l1) {
         result->digit += l1->digit;
         next1 = l1->next;
      }

      if (l2) {
         result->digit += l2->digit;
         next2 = l2->next;
      }

      next_carry = result->digit/10;
      result->digit = result->digit%10;

      result->next = sum_list(next1, next2, next_carry);

      return result;
   }
}
```

Table 16: Number as lists, implementing the sum.

```
struct listchar{
  char info;
  struct listchar* next;
};

typedef struct listchar* Listchar;

void remove_a_rec(Listchar *plist) {
  if (*plist) {
    /*remove 'a' and 'A' from the tail*/
    remove_a_rec(&((*plist)->next));
    if (((*plist)->info == 'a')||(*plist)->info == 'A') {
      Listchar pc = *plist;
      *plist = (*plist)->next;
      free(pc);
    }
  }
}
```

Table 17: Remove values `'a'` and 'A' from a list of `char`.

```
struct listint{
  int info;
  struct listint *next;
};

typedef struct listint *Listint;

Listint difference_list_vect(Listint l, int vect[], int dim){
  if (l) {
    int i;
    int present;
    i = 0;
    present = 0;
    while (!present && i<dim) {
      if (vect[i] == l->info)
        present = 1;
      i++;
    }
    if (!present) {
      Listint newcell = (Listint) malloc(sizeof(struct listint));
      newcell->info = l->info;
      newcell->next = difference_list_vect(l->next, vect, dim);
      return newcell;
    }
    else
      return difference_list_vect(l->next, vect, dim);

  }
  else
    return NULL;
}
```

Table 18: Compute the difference between a list of integers and a vector of integers.

```
struct stringlist{
  char *info;
  struct stringlist *next;
};

typedef struct stringlist *Stringlist;

int mystrlen(char *str) {
  int n=0;
  while (*str!='\0' && *str!=' ') {
    str++;
    n++;
  }
  return n;
}


Stringlist find_words(char *par) {
  if (*par == '\0')
    return NULL;
  else
    if (*par == ' ')
      return find_words(par+1);
    else {
      int len, i;
      Stringlist newcell =
         (Stringlist) malloc(sizeof (struct stringlist));
      len = mystrlen(par);
      newcell->info = (char*)malloc((len + 1)*sizeof(char));
      for (i=0; i<len; i++)
        (newcell->info)[i] = par[i];
      (newcell->info)[len] = '\0';
      newcell->next = find_words(par+len);
      return newcell;
    }
}
```

Table 19: Given a paragraph, find a list in which each cell contains a word of the paragraph.

5.5 Identify words in a paragraph

Given a string containing a paragraph, write a function that returns a list of strings in which each cell contains a word of the paragraph. Assume that the every word is separated from the other by a sequence of at least one *blank* `' '`. Do not use any function from library `string.h`.

Solution We propose a recursive solution and we use an auxiliary function `mysrtlen` that computes the length of the first word in the paragraph. At each recursive call a word is processed. The solution is shown in Table 19.

5.6 Compare lists of string

Given two lists of strings, write a function that decides if they are equals. The comparison between the strings must be done char by char.

```
struct liststring{
  char *info;
  struct liststring *next;
};

typedef struct liststring *Liststring;

int equals(Liststring l1, Liststring l2) {
  if (!l1 && !l2)
    return 1;
  else {
    if (!l1 || !l2)
      return 0;
    else
      return (!strcmp(l1->info, l2->info) && equals(l1->next, l2->next));
  }
}
```

Table 20: Comparison of two lists of strings.

Solution The solution is simply based on the recursive comparison of two lists and is shown in Table 20.

5.7 Substring in list

Write a function that decides if, given a list of `char` and a string, the string is contained in the list.

Solution The solution is shown in Table 21. Observe that we need two nested cycles to deal with the cases such as the following: consider the list containing the chard `a-a-a-c-t` and the string ‘‘`aac`’’; the string is contained in the list, but we we must rollback after the failure in recognising the sequence `a-a-a` to the second char.

6 Advanced lists

6.1 Circular or plain list?

In a program the list of integers may be either circular or linear. Write a function that return 1 if a non-empty list is circular, 0 otherwise.

Solution The main idea is to leave a pointer to the head of the list and then analyse all the cells. We terminate in two cases: either we find the null pointer or we find the pointer to the head of the list. The solution is shown in Table 22.

```c
struct cell{
  char c;
  struct cell *next;
};

typedef struct cell *listchar;

int substring(listchar l, char* str) {
  listchar pc;

  int found = 0;
  int i;

  while (l && !found) {
    i = 0;
    pc = l;
    while(pc && str[i] && pc->c == str[i]) {
      pc = pc->next;
      i++;
    }
    found = (str[i]=='\0');
    l = l->next;
  }

  return found;
}
```

Table 21: Is a string contained in a list of char?

```c
int cicular-linear(listint l) {
  listint pc = l;
  while (pc->next!=NULL && pc->next!=l)
     pc = pc->next;
  return (pc->next == l);
```

Table 22: In the list circular or linear?

7 Exercises on binary trees

7.1 Delete a tree

Write a function that removes from the heap a binary tree of integer.

Solution We propose the recursive solution to the problem in Table 23. Notice that, similarly to the library function `free`, the pointer specified as actual parameter is not changed since it is passed by value.

8 Tests

8.1 Questions

1. Given the following declaration: `int *a;` what can be said about `&a`?

   (a) It is an expression of type `int`

   (b) It is a variable of type `int`

   (c) It is an expression of type `int**`

   (d) It is a variable of type `int**`

2. Given the following function:

   ```
   void swap(int *a, int *b) {
      int c = *a;
      *a = *b;
      *b = c;
   }
   ```

   and the following call:

   ```
   int *k1, *k2;
   swap(k1, k2);
   ```

   Which is the correct choice?

   (a) The listing does not compile

   (b) The listing compiles but the usage of pointers is wrong

   (c) The listing exchanges the values of pointers `k1` e `k2`

   (d) The listing exchanges the values of the variables pointed by `k1` e `k2`

3. Given the following declarations `int a; int *b;` the writing `*(b+a)` denotes:

   (a) It is a variable with type `int`

   (b) It is not correct

   (c) It is an expression with type `int*` but not a variable

   (d) It is a variable with type `int*`

4. Given the declarations `int a; int *b;` the writing `(*b)+a` denotes:

(a) A variable of type `int`

(b) An expression of type `int` but not a variable

(c) An expression with type `int*` but not a variable

(d) A variable with type `int*`

5. Given the following declaration `int myvect[100];`, then the writing `myvect` in an assignment is:

   (a) A variable with type `int*`

   (b) An expression with type `int*` equivalent to `&myvect[0]`

   (c) A variable of type `int` equivalent to `myvect[0]`

   (d) Is wrong

6. Given the following listing:

```
char* mystring() {
   char news[100];
   news[0]='\0';
   return news;
}
```

   Which of he following is correct?

   (a) It does not compile

   (b) It compiles the it does not work properly because it accesses to position 0 of an array

   (c) It compiles but the instruction `return` causes a type casting

   (d) It compiles but the instruction `return` gives the address of a memory location that will be freed when the execution of the `mystring` terminates

7. Given the following function:

```
int foo(int a, int b) {
   if (a == b)
      return a;
   else
      return (a>b) ? foo(a-b, b) : foo(a, b-a);
 }
```

   What is the value of the expression `foo(18, 60)`?

   (a) 6

   (b) 60

   (c) 18

   (d) The recursion does not terminate

8. Given the following function:

```
int* mine(int a) {
   return &a;
}
```

and the following call:

```
int k1=8;
int *pt;
pt = mine(k1);
```

Which of the following propositions is correct?

(a) `pt` points to variable `k1`

(b) `pt` points to a memory location which is not associated with a variable

(c) The listing does not compile because of a type error

(d) The value of `pt` is 8

9. Given the following declarations `float *a; float *b;` the writing `b-a` denotes:

(a) An expression with type `float*` but not a variable

(b) An expression with type `float` but not a variable

(c) An expression with type `int` but not a variable

(d) A variable with type `float*`

10. The C language supports the parameter passing:

(a) by value

(b) by reference

(c) both by value and by reference

(d) not by value nor by reference

11. Given the following declaration `double* myvect[100];`, how can you interpret this data structure?

(a) A vector with 100 cells of type `double`

(b) A vector with 100 pointers to `double` memory locations

(c) A pointer to a vector of cells with type `double`

(d) A pointer to a vector of pointers to `double` memory locations

12. Given the following declarations `float a[100]; int b=20;`, the writing `*(a+b)` is equivalent to:

(a) It is not correct

(b) `&a[0]+b`

(c) `*a[b]`

(d) `a[b]`

13. Given the following code:

```
int i, a = 1;
for (i=0; i<10; i++)
  a = a*2;
```

Which is the value of variable `a` after the cycle termination?

(a) 512
(b) 20
(c) 1024
(d) 2

14. Given the following function:

```
void foo(int a, int *b) {
  *b = a;
}
```

and the follwoing function call:

```
int k1=10, k2=20;
foo(k1, &k2);
```

Which is the value of variables `k1` and k2?

(a) 10 and 20
(b) 10 and 10
(c) 20 and 10
(d) The code is wrong because pointers are used uncorecctly

15. Given the following declarations: `float a; float *b; float **c=&b;`. We want to assign value 10.2 to variable `a` with the following instruction `**c = 10.2`. Before doing this, which other instruction must be executed?

(a) `*a = b;`
(b) `b = a;`
(c) `&b = a;`
(d) `b = &a;`

16. Given the array declaration `int a[100];` which of the following writings is equivalent to `a[3]`?

(a) `a+3`
(b) `&(a+3)`
(c) `*(a+3)`
(d) `**(a+3)`

17. Given the following code:

```
float* myfuction(float x){
  float var = 12.0 + x;
  return &var;
}
```

Which of the following propositions is true?

(a) The code does not compile due to an error on types
(b) The code does not compile due to a syntax error
(c) The code compiles but the return statement returns the address of a local variable that will be deallocated just after the termination of the function call
(d) The function assigns the value 12.0 to the argument

8.2 Solutions

1. c
2. b - function `swap` uses the operator * on pointers `a, b` which contains the same values of `k1, k2` which are not initialised.
3. a
4. b
5. b
6. d - local parameters are automatically deallocated when the function execution terminates
7. a - the function computed the greatest common divisor of two positive integers
8. b - in C the parameters are passed by copy
9. c
10. a
11. b
12. d
13. c
14. b
15. d
16. c
17. c

```c
struct bintree{
  int info;
  struct bintree *left;
  struct bintree *right;
};

typedef struct bintree *Bintree;

void delete_tree(Bintree bt) {
  if (bt) {
    delete_tree(bt->left);
    delete_tree(bt->right);
    free(bt);
  }
}
```

Table 23: Free the memory occupied by a binary tree.