DATA STRUCTURES AND ALGORITHMS
Department of Electrical and Computer Engineering
Faculty of Science and Technology, University of Coimbra

WORKSHEET 1
CONSOLIDATION OF PROGRAMMING CONCEPTS IN C

1.1 Objectives

This assignment sheet aims at reviewing important concepts in C language, which were taught in the Computer Programming course unit:

- Review main concepts related with variables, flow control, data structures, and functions;
- Review main concepts related with the manipulation of basic data structures – arrays, structures, and files;
- Make the student aware of the importance of source code indentation, and source code documentation through the use of comments embedded on it, making the code more readable, optimizing the development effort, and reducing the chance of coding errors (i.e. bugs);
- Recall basic concepts about variables and pointers to variables;
- Review the different mechanisms available in C for argument passing in functions, namely by value and by pointer.

1.2 Flow control – Loops

A student with basic programming skills in C language should be able to use different types of loops depending on the purpose in hands, namely `for / while / do` loops, to perform repetitive processing tasks, such as processing strings and arrays.

Consider the following example:

```
char names[20][60];
int i;
. . .
for (i=0; i<20; i++) {    /*  display an array containing 20 names  */
      printf("The name of student %d is %s\n", i+1, names[i]);
}
```

The `for` loop may be subject to an unusual writing with respect to its initialization statement, condition statement, and increment/decrement statement, with the purpose of obtaining more compact source code. However, this practice should be avoided so as to avoid the occurrence of unnecessary programming errors.

Any of the different types of loops available in C, `for/do/while`, may be readily converted in one of the others. Note the examples below.

`for` loop converted in a `while` loop:

`for (A; B; C) {`
`  D`
`}`
	`A;`
	`while (B) {`
	`  D`
	`  C;`
	`}`

The program excerpts on the left and on the right do exactly the same regardless of what A, B, C or D (statements or subprograms) are. However, the source code clarity is different. Note that the left-subprogram is not necessarily simpler than the right-subprogram; it all depends on the complexity of A, B, C and D.

`while` loop converted in a `for` loop:

`while (E) {`
`  F`
`}`
	`for (;E;) {`
	`  F`
	`}`

In this case, the left-subprogram makes more sense than the right-subprogram because it is more readable.

Often, the programmer needs to opt for a `do/while` loop, instead of a more usual `while` loop with the condition statement on the top of the loop. The `do/while` loop is used when the loop should execute at least one time regardless the condition. Note that the condition is evaluated only in the end of each iteration in the `do/while` loop. However, the `do/while` loop could be converted in a `while` loop at the expense of a less readable code:

`do {`
`  G`
`} while (H);`
	`while (1) {`
	`  G`
	`  if (!H) break;`
	`}`

Sometimes the programming algorithms require the use of nested loops, i.e. the use of loops that contain other loops inside. This loop nesting tends to confuse less experienced programmers who are not adequately familiar with `for/do/while` loop statements. The reasoning behind the use of two nested cycles is similar to the specification of mathematical sums. For instance, if we want to compute the sum of every element in a bi-dimensional matrix (`m×n`),

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_{ij} \ ,$$

two nested loops are needed:

```
#include <stdio.h>

#define LINES  4   /* number of lines */
#define COLS  3   /* mumber of columns */

int main() {
  int a[LINES][COLS] = {{1,2,1},{2,1,2},{1,2,1},{2,1,2}};
  /* sum every element of the matrix */
  int s = 0;
  for (int i = 0; i < LINES; i++)
       for(int j = 0; j < COLS; j++)
          s += a[i][j] ;
  printf("The sum is equal to %d\n", s);
}
```

Two nested loops are also needed to compute the sum of two bi-dimensional matrices `m×n`:

$$s_{ij} = a_{ij} + b_{ij} \ \ (i=0..m\text{-}1,\ j=0..n\text{-}1).$$

Three nested loops are needed to compute the product of a bi-dimensional matrix a(`m×n`) by a matrix b(`n×p`), resulting in a matrix s(`m×p`):

$$s_{ik} = \sum_{j=0}^{n-1} a_{ij} b_{jk} \ , (i=0..m\text{-}1,\ k=0..p\text{-}1).$$

Suggestion: Write a C program to compute the sum of two matrices. Afterwards, write a C program to compute the product of two matrices.

1.3 Structures

Structures (reserved word `struct` in C) are a convenient way to group a data set associated with a given object or entity. There are many real-life examples where data records are used to represent, e.g., a citizen card with several personal data that identify a person, car record, driver license, bank debit/credit card, student, etc.

Example of the definition of a structure with three members:

```
struct Arc { /* node of a graph diagram */
  int source, destination;
  float cost;
};
```

Example of the declaration of variables of type `struct Arc` (note that the reserved word `struct` is not optional in the declaration):

```
struct Arc a;
struct Arc b;
```

The manipulation of a structure is slightly different from manipulating a basic variable (e.g. of type `int`). Besides specifying the name of the structure variable to be accessed, the name of the data field (usually denoted as member) inside the structure must also be specified after a period.

Example of a simple assignment statement:

```
i = 5;
```

In the example below, a structure is declared and the access to its data fields is exemplified.

```
struct TDate{
  int day, month, year;
};

struct TDate birthDate;

birthDate.day = 19;
birthDate.month = 10;
birthDate.year = 1998;
```

If we want to copy the data contained in a structure to another structure, we simply write an assignment statement at the level of the structure:

```
TDate birthDateA, birthDateB;

birthDateA.day = 19; birthDateA.month = 10; birthDateA.year = 1998;

birthDateB = birthDateA;
```

1.4 Arrays

Arrays are a convenient way to store a (possible large) set of data of the same type. The access to each one of its elements is done by indexation. Example: `a[i]` represents the *(i+1)*-th value stored in array `a[]`.

A common programming error with arrays arises when the program tries to access an element beyond the limits of the array. The C compiler *does not* check the correctness of the index. This type of error is even more likely if complex arithmetic expressions are used to index an array, e.g:

```
a[i+(j-1)*6*(y+7)] = 3;
```

Errors in array indexation may result in severe program failures, depending on the value of variables `i`, `j` and `y`, and depending on the size of the array. It is absolutely necessary that the programmer uses carefully the array indexation limits so that indexation errors, which are very difficult to debug, are avoided. The minimum value of the index is zero and it can never be higher than the number of elements of the array minus 1.

To store a 3×4 bi-dimensional matrix, a unidimensional array of size 12 could be used:

```
int matrix[3*4];  /* unidimensional array with 12 integers */
...
for (int line = 0, i = 0; line < 3; line++)
  for (int col = 0; col < 4; col++, i++)
     matrix[line *4 + col] = i;  /* equivalent to the assignment: matrix[i] = i */
```

Special care must be given to the indexation of the array so as to avoid programming errors. The programmer must have the required abstraction skills to correctly understand the layout of the data in the computer memory so as to avoid such problems. Note, however, that it would be much more practical to define a bi-dimensional array rather than using an unidimensional array:

```
int matrix[3][4];       /* declaration of a bi-dimensional array 3x4 */
```

and use two indexes instead of one to access the elements of the matrix in the previously presented example.

The severe consequences of the occurrence of programming errors in array indexation can be easily understood through an example:

```
int a, b[10], c;        /* declaration of 2 integers and a 10-element array */
char d;
```

What happens if the assignment `b[-1] = 0` (clearly a programming error) is written in the program?

The result will depend on the compiler being used, but most likely the program will write "on top" of the variable `a`, because this variable is physically stored in memory at an address located just before the array `b`.

And what happens with the instruction `b[10] = 3` ? Most likely (it depends on the compiler), the program will write "on top" of the variable `c`, thus resulting in a *bug*! Note that `b[10]` is a position beyond the last element of array `b`.

And what if the instruction `b[14]=7` is written in the program? Here the error can me much more severe because the out-of-range index may refer to a block of data located beyond the memory area reserved to the program variables. It can be even a block of data within the program instructions! Therefore, in this case, the program execution may yield a failure that can only be overcome by terminating (abnormally) the program and re-executing it; and the situation will be repeated whenever that apparently "harmless" part of the code is executed.

1.5 Files

Reading and writing data to files are a common task in almost all programs. Data stored *persistently* as a file in the computer hard disk, or in an external disk, may represent image, text, video, databases, music, games, etc. Files are very important because often data processed by programs must persist beyond the lifetime of the program execution. Therefore, programs manipulate files very often to store, retrieve, modify, and delete data stored in files.

Besides the basic steps of file manipulation – open, read/write, close – the programmer should care about additional aspects that are inherent to common operations with files. Contrarily to other data structures, files are prone to several exception situations that should be dealt with by the program (exception handling). For instance, the program may try to open a file that does not exist, or try to store data in a file for which the user does not have write permission, or try to access a file that no longer exists physically in the computer (e.g. the external disk or the CD/DVD disk where it was stored is unplugged/removed from the computer by the user during the program execution), or try to read data from a file beyond the end of the file, etc. Ideally, the program should handle properly all these exceptions.

Example regarding opening a file:

```
#include <stdio.h>
...
char filename[80]; /* filename containing up to 80 characters (including '\0') */
FILE *my_file;     /* variable associated with the file to be read */
...
printf("Insert the filename to be opened->";
scanf("%s", filename);
my_file = fopen(filename, "r");  /* open file to read data */
if (my_file == NULL) { /* Was there any error when opening the file? */
  printf("ERROR when opening the file %s\n", filename);
} else { /* file was successfully opened */
    ...  /* all source code related with reading data from the file goes here */
  fclose(my_file);    /* close the file when the program does not need it anymore */
}
```

In this example, the program checks whether the file is successfully opened. This kind of exception handling is important in programs that manipulate files.

Another important feature of file manipulation is random data access in binary files. In binary files, every data record occupies the same number of bytes in the file, therefore the program can access directly to *any* one of the (possibly many) data records stored in the file rather than accessing sequentially to data records. In this way, the program can run much more efficiently, especially with large files.

Imagine, for example, that a program needs to read the 1 000 000-th element of an array of integers stored in the computer memory. The code required for this is very simple:

```
  int result, array[2000000];
  ...
  result = array[999999]; /* result becomes equal to the 1000000th array element */
```

By using random file data access, a similar behavior can be obtained in a program that manipulates a binary file containing integer values:

```
int result;
FILE *my_file;
...
my_file = fopen("my_integers.dat", "rb"); /* r for reading, b for binary data */
...
fseek(my_file, 999999 * sizeof(int), 0); /* seek the pointer position */
fread(&result, sizeof(result), 1, my_file);  /* read the 4 bytes of the int */
...
fclose(my_file);  /* do not forget to close the file! */
```

For the sake of simplicity, exception handling related with opening and accessing data from the file was omitted in the example. Besides exception handling, which should be included in a good-quality

program, the example shows that accessing directly a given record in a binary data file is not much more complicated than accessing data in an array.

1.6 Indentation: a "lifeguard" programming habit

One of the possible sources of bugs created unintentionally by inexperienced programmers is the lack of visual organization of the source code. When the programmer is trying to learn the syntax of instructions, trying to create his/her first algorithms, or simply trying to make a program that fulfills certain objectives, he/she rarely pays attention to texts, like this one, that try to alert him to the fact it is vital to indent the code. In fact, it is vital to indent, comment and document internally the source code that is produced.

Indentation consists of writing the code in order to visually show the hierarchy and structure of the source code. The indentation must be applied in the definition of variables, in the structuring of a function or in the structuring of a loop or in any flow control block (e.g., `if`, `if-else`, `switch`, etc.).

Consider the following two examples regarding the declaration of variables:

```
int a,b,c,d,e,f,g,h;
float f1,f2,a3,f14,e15,c13,v20,c40;
char xy,de,ef;
```

```
int
  a, b, c, d, e, f, g, h;
float
  f1, f2, a3, f14, e15, c13, v20, c40;
char
  xy, de, ef, df;
```

Note that the source code on the left and on the right are equivalent. But, which one is clearer and more readable? What lesson should be extracted from the example?

Consider now two examples regarding the definition of a new structure type:

```
typedef struct {
int num, marital_status;
char name[20], date[8];
}tCitizen;
```

```
typedef struct {
  int  num;
  char name[20];
  char date[8];
  int  marital_status;
} tCitizen;
```

Which one of these two excerpts of source code is more readable? What have you learnt from this second example?

Example with `for` loops:

```
for (a=0;a<20;a++) {
for (b=10;b<30;b+=2)
for (z=5;z>-3;z--)
d=3+z+b*a;
s+=d;
for (x=3.5;x<7.0;x+=1.5){
s+=x;}
}
```

```
for (a = 0; a < 20; a++) {
  for (b = 10; b < 30; b += 2) {
    for (z = 5; z >- 3; z--) {
      d = 3 + z + b * a;
    }
  }
  s += d;
  for (x = 3.5; x < 7.0; x += 1.5) {
    s += x;
  }
}
```

Do you believe that the code on the right does exactly the same as the code on the left? Which of the two texts allows you to better visualize how the cycles are embedded? Note that non-strictly necessary braces have been added to the code on the right to improve clarity.

Note that all examples of indented codes typically have a better "look" when it comes to structuring. Furthermore, the use of additional lines of code, as well as the use of more braces to highlight the nesting hierarchy improves the code readability, thus reducing the probability of the programmer making errors when writing the code (e.g. missing closing braces). The way of programming shown on the left in the previous examples is very conducive to the appearance of programming errors.

Each person is free to indent or structure the source in his/her own style. However, a few decades after the emergence of the first programming languages, some rules or recommendations have been followed by thousands of programmers around the world. For example, whenever you move down a level in a program hierarchy, the text is moved two spaces (or a tab) to the right and whenever you move up the hierarchy, the text is moved two spaces (or a tab) to the left (see the previous examples).

This part of the text is far from being able to teach a whole set of rules that programmers have spent years perfecting to avoid bugs or improve code clarity. Just to give a few examples of a whole set of rules that is nowadays assumed to produce good-quality, readable code:

- There are rules for naming variables and the use of lowercase and uppercase characters;
- There are rules for numbering code versions;
- There are rules on how to validate the code;
- There are several rules for indenting the source code;
- There are rules for writing portable C code for different types of computer architectures without involving any type of code changes (many restrictions!);
- There are rules on how comments should be placed and how comments change lines when they are too long.

It is hoped that after reading this section on indentation you become aware of the need to properly organize source code, so as not to risk spending most of your time finding and correcting less obvious, but avoidable, bugs, sometimes almost "inexplicable", or understand "that piece of code that I made a month ago and I don't understand how it was coded and how it works", just because the code was not commented nor written in a clear or organized way.

1.7 Pointers

The declaration of a variable of type pointer has the following syntax: `<type> *pointer_name;`

A pointer variable "does not point at anything" until it contains the address of a memory location reserved for program data. When we want to indicate that a pointer does not contain useful information (i.e. that it does not point at anything), we can make the following assignment:

```
int *pointer;
pointer = NULL;
```

Analyze the following code to recall the concepts previously learned about pointers in the Computer Programming course unit:

```
    int a, b;
    int *p_1, *p_2;

    /* variables a and b have two values... */
    a = 10; b = 20;

    /*...and there are pointers pointing at them */
    /* (What does this mean?) */
    p_1 = &a; p_2 = &b;

    /* What will happen when executing the following lines?... */
    printf("%p, %p\n", p_1, p_2);
    printf("%d, %d\n", *p_1, *p_2);
    p_1 = p_2; p_2 = &a;
    printf("%d, %d\n", *p_1, *p_2);
```

C compilers store arrays line by line (in Fortran matrices are stored column by column!). Like this:

```
#include <stdio.h>
#include <stdbool.h>

  int matrix[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}}; /* declaration, initializ. */
  int *p = &matrix[0][0];        /* pointer to the 1st matrix entry */

  bool line_by_line = true;      /* confirm (or not) line-by-line storage */

  for (int line = 0; line < 3 && line_by_line; line++)
    for (int col = 0; col < 4 && line_by_line; col++)
      line_by_line = (&matrix[line][col] == &p[line*4+col]);

  if (line_by_line) printf("\nThe matrix IS stored line by line\n");
  else printf("\nThe matrix IS NOT stored line by line\n");
```

When running this code excerpt, the following message will be displayed in the console:

```
The matrix IS stored line by line
```

However, there is no need to traverse a bi-dimensional array using a pointer, when it is much more intuitive and less susceptible to bugs to use the array name and the appropriate indexes.

1.8 Passing arguments to a function by value or by pointer

Arguments can be passed to functions by value, i.e. by passing a copy of the variable to the function. This implies that the changes made to the argument inside the function are not reflected in the original variable (outside the called function) since they were made on a copy of it.

In practice, it is quite common to have the need to return multiple values. Arguments passed by reference or by pointer can be used with this purpose.

Consider the example of a function that should swap the value of two integer variables:

```
void swap(int num1, int num2) { /* does the function swap values as intended? */
  int temp;
  temp = num2;
  num2 = num1;
  num1 = temp;
}
/*-------------------------------------------------------*/
int main() {
  int x = 4, y = 2;

  printf("Before calling swap(), x=%d and y=%d\n", x, y);
  swap(x, y);
  printf(" After calling swap(), x=%d and y=%d\n", x, y);
}
```

Try this code. Does it run as it was intended?... No! And why not?

This function does not run properly because the two arguments are passed by value. This means that the function receives a local copy of the two parameters. Any changes that are made to these local copies are not reflected in the original variables in the function `main()`.

In C there is a technique that allows solving this problem. It is the use of pointers in the argument list: instead of passing a copy of the variable, a pointer to the variable is passed instead. This pointer can be manipulated in order to change the value of the variable it points to (variable external to the function). It is curious to point out that the pointer is passed by value. The function cannot change the value of the pointer (i.e. the address), as it receives a local copy of the pointer. However, the function can change the contents of the memory pointed to by the pointer, i.e. the variable associated with that pointer. Passing arguments by pointer has the advantage that changes made to the function are reflected in the value of the variables external to the function and in the fact that we can change several arguments within the same function simultaneously. The main disadvantage in using pointers is the greater complexity associated with the use of pointers (especially for less experienced programmers). A corrected version of the previous non-working example is presented below, which uses arguments passed by pointer:

```
void swap(int *num1, int *num2) {
  int temp;
  temp = *num2;
  *num2 = *num1;
  *num1 = temp;
}
/*-------------------------------------------------------*/
int main() {
    int x = 4, y = 2;

  printf("Before calling swap(), x=%d and y=%d\n", x, y);
  swap(&x, &y);
  printf(" After calling swap(), x=%d and y=%d\n", x, y);
}
```

Arrays are usually passed to functions as arguments passed by value. As the value associated with the name of an array always contains the address of the first element of the array, i.e. a pointer to the first element of the array, the function is actually given an address (passing by pointer):

```
/* zeroes the dim elements of an array a of integers */
void zeroes(int *a, int dim) {  /* or  void zeroes(int a[], int dim) */
  for (int i = 0; i < dim; i++)
       a[i] = 0;
}
/*-------------------------------------------------------------*/
int main() {
  int array[12]={1,2,3,4,5,6,7,8,9,10,11,12}; /* an array... */
  zeroes(array, 12);
  return 0;
}
```

If the array to be passed to the function is multi-dimensional, it is necessary to pass as argument the number of columns, because the values are stored line by line, and it is necessary to know the number of columns to move to the next line.

Consider the following example:

```
/* zeroes the elements of a bi-dimensional array with m lines and 4 columns */
void zeroes(int a[][4], int m) {
   for (int i = 0; i < m; i++)
     for (int j = 0 ; j < 4; j++)
       a[i][j] = 0; /* distance to the 1st element of the array: i*4+j */
}
/*-------------------------------------------------------*/
int main() {
  int matrix[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
  zeroes(matrix, 3);
  return 0;
}
```

1.9 The `#include` directive and multiple source code files

In programming, there are many ways to organize the source code to make life easier for the programmer:

- comments (helps to read the programs more easily);
- documentation (documenting code helps to discover bugs; improves clarity);
- indentation (helps to understand the code and its structure, avoiding many bugs);
- break complex functions into several simpler functions (decrease complexity);
- use function libraries (saves the programmer work, using `#include`);
- distribute the code among several files (helps to separate tasks, use of `#include`);
- etc.

The programmer has at his/her disposal the use of the `#include` compilation directive that allows him/her to include either code made by others or code made by himself/herself. In general, each file with extension .c (or .cpp for C++ programs) contains the definition of functions related to a specific task/common purpose, which are grouped in the file (e.g. input/output, character manipulation, array manipulation, etc). To make the functions in each .c file easily accessible to the functions in other c files, a header file usually having the same name but with extension .h, is also created, which contains the prototypes of all functions defined in the file with extension .c.

The `#include` directive, as you already know, allows you to include one file in another. Thus, when a file `main.c` includes a file `arrays.h`, by adding the line `#include "arrays.h"`, it then contains all the prototypes contained in `arrays.h`, and it would be possible to compile the `main.c` source code file and create the object file `main.obj`. To create an executable program, it would be necessary to link the object code `main.obj` with the object code `arrays.obj` (obtained from the compilation of `arrays.c`).

The action of grouping functions with a specific theme in a given file with c extension and the creation of the respective file with the list of prototypes make the programs modular and reusable. Imagine that we wanted to group all the functions associated with swapping values in arrays in one file. At the moment, we only have two functions:

Then file `arrays.c` would be:

```
#include "arrays.h"
/*-------------------------------------------------------------*/
void swap(int *num1, int *num2) {
  int temp;
  temp = *num2;
  *num2 = *num1;
  *num1 = temp;
}
/*-------------------------------------------------------------*/
void swap2(int num1, int num2) { /* does the function swap values as intended? */
  int temp;
  temp = num2;
  num2 = num1;
  num1 = temp;
}
/*-------------------------------------------------------------*/
```

The file `arrays.h` would be:

```
#ifndef ARRAYS_H
#define ARRAYS_H
void swap(int *num1, int *num2);
void swap2(int num1, int num2);
#endif
```

The directives `#ifndef`, `#endif`, and `#define` are used in the example to avoid including more than once the declarations contained in `arrays.h`, which would raise a compilation error caused by the multiple declaration of the same functions.

Now, any program can reuse the functions defined in `arrays.cpp`. For instance, the file `main.cpp`:

```
#include <stdio.h>
#include "arrays.h"
int main() {
  int x = 4, y = 2;

  printf("Before calling swap(), x=%d and y=%d\n", x, y);
  swap(&x, &y);
  printf("After calling swap() by pointer, x=%d and y=%d\n", x, y);
  swap2(x, y);
  printf("After calling swap() not by pointer, x=%d and y=%d\n", x, y);
}
```

When a software development team writes code in C (or in C++), it uses `#include` to integrate code made by several people. It is common nowadays for teams to integrate more than 100 independent code contributions in a single program! This is equivalent to a program containing a few thousand functions or tens of thousands of variables. Imagine what it would be like for someone to try to integrate all the code in a single file!

The purpose of this section is to alert you to the fact that in the near future you may have to resort to partitioning a program into several files in order to better organize your source code and allow the reuse of some functions.

1.10 Suggested Exercises

With the set of exercises presented below, it is intended that the student recalls important concepts taught in the Computer Programming course unit. To allow for a better orientation of the study, each exercise was classified according to its level of difficulty. The student is advised to try to solve at home the exercises that could not be solved/completed during practical classes.

Exercise 1.1 – FOR – Easy

Write a program that computes the result of the following mathematical expression: $\prod_{n=5}^{19}\left(\sum_{i=1}^{n} 2n\right)$.

Exercise 1.2 – FOR – Easy

Write a function that returns `true` if its input argument (integer with up to 9 significant decimal digits) is a prime number. You have to include `stdbool.h` to be able to use the `bool` type in your C program.

Exercise 1.3 – FOR – Medium

Write a function that, given an uppercase character between `'A'` a `'J'` as input argument, draws a diamond in the console as large as the distance of the character to `'A'`. Example of a diamond obtained when the chosen character is `'D'`:

```
   A
  B-B
 C---C
D-----D
 C---C
  B-B
   A
```

Exercise 1.4 – STRUCTS – Easy

Define a data structure that allows you to store 2 integers (member number and movie number), 2 dates (loan and delivery) and an actual amount corresponding to an amount paid by the client. Suppose that this structure is used in a DVD lending database of a video club. Use within the structure another date structure defined by you, which allows you to calculate how many days a client had a film in his/her possession (to simplify, consider that all months have 30 days).

Exercise 1.5 – STRUCTS – Easy

Define an appropriate data structure to store all the data present on a Portuguese Citizen Card (`PCC`). Choose carefully the type and data size you assign to the different fields of the data structure, and choose the appropriate structure to save the date. To test your structure, declare and initialize an example of variable of type `PCC`. Implement a function that, given the current date and an argument of type `PCC`, computes the age (the number of years) that the person is at the moment.

Exercise 1.6 – ARRAYS & FUNCTIONS – Medium

Define an array that allows memorizing 10000 records. Each record must contain a string (the device serial number, with up to 20 characters) and a floating-point value corresponding to a measurement performed by that device. Also implement a function that allows, using this data structure, to return the number of elements in an array of that data type containing a value equal to or greater than a given value (measurement threshold) received as input argument.

Exercise 1.7 – ARRAYS – Medium

Declare and initialize with various constants chosen by you a bi-dimensional array containing 20 × 20 floating-point values. Implement a program that asks the user for a real number and prints on the console, in text mode, a square made with 20 × 20 characters of the type `'-'`, `'='`, or `'+'`. Each character forming the square must relate to each value in the bi-dimensional array entry as if the value in the array is, respectively, lower, equal, or higher that the input real number.

Exercise 1.8 – ARRAYS – Medium

Define a data structure that allows storing 20 names of people, each name containing up to 20 characters. To use this structure, make a program that, given the key pressed by the user, from `'A'` to `'Z'` (either lowercase or uppercase), print on the console all the names starting with that character.

Exercise 1.9 – FILES, STRINGS – Medium

Implement a program that, given a text file, prints on the console all lines of the file that contain the word `"word"`.

Exercise 1.10 – FILES – Medium

Implement 2 programs:

a) program 1 – define an array with 20 integers and save the array in a binary file.
b) program 2 – read the binary file containing the array with 20 integers and print them on the console.

Exercise 1.11 – INDENTATION / CLARITY – Easy

Try to explain what the result of the program presented below is. Is it an easy task?

Make the code more readable by using a basic indentation rule. Simplify the code and change what you think is less good, until you get a program as readable as possible.

```
int dif=0; for (dif=0,int vz=2;vz<12;dif=complic){complic=0;for (int xpty10o_my = 0;xpty10o_my<10;xpty10o_my=xpty10o_my+332/166){complic =(xpty10o_my+1)*(vz-1)+complic;}++vz;complic=dif+complic;}printf("%d",dif);
```

Exercise 1.12 – FUNCTIONS – Easy

Implement a function that computes the area and perimeter of a circle, given the value of its radius. Implement a program that asks the user to input the radius of a circle and, after calling the previously defined function, writes the area and the perimeter of the circle on the console.

Exercise 1.13 – FUNCTIONS – Easy

Consider the following variables:

```
struct address{
  char street[40];
  char town[40];
  int postalCode1;
   int postalCode2;
} house;

int volumeCube;
```

Knowing that you have to create two functions, one that asks for information from a user to fill in an address, returning the information received properly, and another that asks the length of a cube edge from a user, returning the value of its volume, how would you solve your problem in an expeditious way, by using the advice provided in the previous sections of this document? Justify the answer.

Exercise 1.14 – FUNCTIONS – Medium

Solve the exercise **1.9** in a more modular and organized way by partitioning the functionality of the program in different functions that are called in the function `main()`:

a) Implement a function that asks the user to input a string and returns this string.
b) Implement a function that receives a word and a string, searches for the word within the string, and writes on the console the number of occurrences of the word in the string.
c) Implement a new version of the main() function that calls twice the function implemented in point a) (to get the input of the filename and of the word to be searched) and once the function implemented in point b).

Exercise 1.15 – FUNCTIONS – Medium

Implement a function that accepts the name of a text file as a parameter and returns the number of uppercase and lowercase characters that exist in that file. Then implement a program that calls this function. Do not forget to improve the presentation of the code through adequate indentation, a good choice of names for the variables, and comments that makes the code more readable and clearer by presenting useful information (information not evident at first sight).

Exercise 1.16 – ARRAYS, FUNCTIONS – Medium

Write the function `void invert(int array[], int dim)` that swaps the position of the elements of the array of size `dim` (e.g. the first element becomes the last one, the second becomes the penultimate, etc.). Design its code so that it calls an auxiliary function `swap()`that swaps the value of the integer variables passed by pointer as arguments to it.

Exercise 1.17 – ARRAYS, FUNCTIONS – Easy (appeared in the test held on 12/03/2014)

Consider an array of structures (i.e. each element is a `struct`) containing data about stores of a company: name (string with 50 useful characters), store number (integer) and sells amount (real number, in Euro).

a) Declare the `struct`.
b) Implement the function `testSells()` that counts (and returns) the number of stores with sells greater than `sellsThreshold` (a real number). The arguments of the function are an array (of the type defined in a)), the size of the array, and `sellsThreshold`.
c) Implement a `main()` function to test the function implemented in b) and display on the console the number of stores that sold more than 100.000 Euro, for an array containing 5 stores.

Exercise 1.18 – ARRAYS, FUNCTIONS – Medium (appeared in the test held on 29/03/2017)

Consider a computer application that aims to record and monitor clinical data of users of a health establishment. In this application, there is a form defined with the reserved word struct that stores information about a user suffering from hypertension: `name` (C string with size 60), `age` (`int`), `weight` (`float`), `height` (`float`), `systolic` (`float`) and `diastolic` (`float`).

a) Declare the `struct`.
b) Implement the function `updateBloodPressure()` that returns void and receives as arguments a structure by pointer, the systolic blood pressure value by copy (`float`), and the diastolic blood pressure value by copy (`float`). The function updates the fields `systolic` and `diastolic` of the structure (1st argument).

c) Implement the function `updateArray()` that receives as arguments an array of structures, an array of systolic blood pressure values, an array of diastolic blood pressure values, and the size of these arrays (equal size for the three arrays). The function updates the fields `systolic` and `diastolic` of every element of the 1st array passed as argument, by using the values contained in the two other arrays passed as arguments. The function must call the function of point b) for every element of the first array.

Exercise 1.19 – ARRAYS, FUNCTIONS – Medium/Difficult

Define an array containing 100,000 real numbers represented in floating-point notation with double precision and `nelems`, an unsigned integer variable that contains the number of elements of the array.

a) Fill in the array with data computed through the mathematical formula $(-1)^i \times i/1000$, being `i` the iterating variable used in the loop required to do the processing.
b) Write a function to print the array on the console and show the average of its elements.
c) Write a function to remove from the array every negative number, while keeping the array compact, i.e. by keeping the remaining values as near to the beginning of the array as possible, and without changing the relative order of its elements. Use an auxiliary function that eliminates element `i` from the array.
d) Test the functions implemented in previous points by writing a program that calls those functions: first the function of point a), then the function of point b), then the function of elimination, and again the function of point b). The program should display the processing time for the delete function (function of point c)).
e) Repeat point c) by implementing a new, more efficient, version of the function that goes only once through the entire array. With this purpose, use two auxiliary indices: `i` that stores the (reading) index of the next element to check whether it is negative; `j` that stores the (writing) index in the table where the next non-negative element found in the array will be written. Extend the test program implemented in d) to compare the run time of the two versions of the elimination function.

FICHA 2
RECURSIVIDADE

2.1 Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de recursividade de funções;
- Saber aplicar de forma correta esse conceito, tendo nomeadamente muito cuidado com a escolha das condições de paragem.
- Adquirir alguma sensibilidade sobre quais as situações em que é mais eficaz a aplicação de recursividade.

2.2 Conceito básico da recursividade

Uma função diz-se recursiva quando se chama a si própria, direta ou indiretamente (por intermédio de outras funções). A recursividade permite implementar algoritmos que decompõem o problema a resolver numa sucessão de problemas cada vez mais simples com a mesma estrutura do problema original.

Como há o risco de uma função recursiva poder chamar-se a si mesma um número infinito de vezes, é essencial que exista uma condição que determine o fim do processo recursivo, caso contrário ocorrerá um erro do tipo *stack overflow*. Esta condição determina o momento em que a função deverá parar de se chamar a si mesma. Para se conseguir implementar uma solução recursiva, estrutura-se normalmente a função correspondente em duas partes fundamentais:

1. **Caso Elementar, Caso Base ou Condição de Paragem**. O caso elementar é resolvido sem utilização de recursividade, sendo este ponto geralmente um limite superior ou inferior da regra geral. Dado que não existe a necessidade de se chamar uma nova instância da função, diz-se que este caso corresponde ao ponto de paragem da recursividade.

2. **Regra Geral**. A regra geral da recursividade simplifica a resolução do problema através da utilização recursiva de casos sucessivamente mais simples do problema. O processo repete-se até se atingir o caso elementar que determina o ponto de paragem de todo o processo.

É importante entender que quando uma função se chama a si própria, passa a ser executada uma nova cópia da função, o que implica que as suas variáveis locais são independentes das variáveis locais da primeira cópia, e não podem relacionar-se diretamente.

Existem algumas vantagens e desvantagens na utilização de métodos baseados na recursividade. Algumas delas são sumariadas a seguir.

Vantagens:

- A utilização de uma função recursiva pode simplificar a resolução de alguns problemas;
- Pode-se obter um código mais conciso e elegante nessas situações;
- Uma solução recursiva pode, por outro lado, eliminar a necessidade de manter o controlo manual sobre uma série de variáveis normalmente associadas aos métodos alternativos à recursividade.

Desvantagens:

- As funções recursivas são geralmente mais lentas e necessitam de mais memória na sua execução do que as funções iterativas equivalentes, uma vez que são feitas muitas chamadas consecutivas a funções que implicam a alocação de memória e passagem de parâmetros a cada chamada;
- Um erro de implementação pode levar ao esgotamento dos recursos associados à pilha que gere a chamada a funções. Ou seja, caso não seja indicada nenhuma condição de paragem, ou se essa condição for definida de forma errada e nunca for satisfeita, então o processo recursivo não terá fim.

2.3 Exemplo clássico de recursividade

Para ajudar a compreender o funcionamento de uma função recursiva, utiliza-se geralmente o exemplo do cálculo do fatorial de um número inteiro positivo. O fatorial pode ser definido em função do fatorial do número anterior o que torna a sua implementação um caso típico da utilização de métodos recursivos. Para um dado número inteiro positivo *n*, podemos definir o fatorial como:

$$n! = n.(n-1).(n-2)….2 \,.\, 1 \text{ ou seja } n! = n.(n-1)!$$

Esta função pode ser definida recursivamente em função de um caso elementar e de uma forma geral:

$0! = 1$ (por definição, caso mais simples)

$n! = n \,.\, (n - 1)!$ para $n>0$ (formula recursiva)

Uma possível implementação em C++ de uma função que calcula o fatorial de um dado número *n* é apresentada a seguir.

```
long fatorial(int n) {
  if (n == 0)    /* Caso base que define a condição de paragem */
    return 1;
  else
    return ( n * fatorial ( n-1 ) );
}
```

Note que, enquanto *n* não for igual a 0, a função `fatorial()` chama-se a si mesma com um valor cada vez menor até chegar a 0. A condição `(n==0)` representa o critério de paragem.

Há certos algoritmos que são mais eficientes quando implementados de maneira recursiva. Todavia, a utilização da recursividade deve ser evitada sempre que possível. A utilização de métodos recursivos tende

a consumir mais memória e a ser mais lenta. Lembre-se que é alocado espaço na pilha de valores (é ocupada mais memória) cada vez que o computador faz uma chamada a uma função. A utilização de funções recursivas sem controlo pode esgotar rapidamente a memória do computador.

2.4 Cuidados a ter com funções recursivas.

Se num determinado método recursivo não tiver sido definida uma condição de paragem adequada, poderão surgir alguns problemas. Na função definida anteriormente para o cálculo do fatorial, o que aconteceria se por engano fosse passado como parâmetro o valor -1? A função entraria num ciclo recursivo infinito pois nunca mais a condição de paragem seria válida. O mais provável seria o programa originar um erro de falta de memória da pilha que gere as chamadas a funções (*stack overflow*). Para evitar este tipo de problemas, teremos que garantir que a condição de paragem utilizada contempla todas as situações não desejadas. Uma possível solução para o exemplo do cálculo do fatorial seria a reformulação da condição de paragem:

```
long fatorial(int n) {
  if (n > 1)
    return ( n * fatorial ( n-1 ) );
  else
    return 1;
}
```

Nesta situação seria assumido que apesar de não fazer sentido existir o fatorial de números negativos a função devolveria 1 para esses casos. Se este pressuposto não for considerado aceitável então será necessário encontrar outra solução. Tente alterar esta função de forma a que devolva **-1** quando o número introduzido é negativo.

2.5 Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à recursividade. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Recomenda-se vivamente que o aluno tente resolver em casa os exercícios que não for possível resolver durante as aulas práticas.

Problema 2.1 – Fácil

Escreva uma função recursiva `imprimeAte(int n)` que imprima no ecrã os números inteiros de `0` até `n` (por esta ordem, separados por espaços ou outros caracteres).

Problema 2.2 – Fácil

Escreva uma função que permita escrever no ecrã `n` asteriscos na mesma linha, em que `n` é um parâmetro dessa função, sem utilizar nenhum ciclo (nem `for`, nem `while` e muito menos o `do...while`). O protótipo da função é:

```
void escreveAsteriscos(int n);
```

Problema 2.3 – Fácil

Escreva uma função recursiva que, dados um número real **x** e um número inteiro **n**, retorne $\mathbf{x^n}$. Comece por escrever uma definição recursiva da função $\mathbf{x^n}$.

Problema 2.4 – Fácil

Escreva uma função que devolva o número de *Fibonacci* associado a um determinado inteiro positivo. A sequência de *Fibonacci* **a(1)**, **a(2)**, **a(3)**, ..., **a(n)**, é definida da seguinte forma:

$$a(1) = 1,\ a(2) = 1 \text{ e } a(n) = a(n\text{-}1) + a(n\text{-}2), \text{ para todo o número } n > 2$$

A função assim definida devolve a seguinte sequência de valores:

1, 1, 2, 3, 5, 8, 13, 21, ... , para valores de n=1, 2, 3, 4, 5, 6, 7, 8, …

Problema 2.5 – Difícil

Escreva uma função que converta recursivamente um dado inteiro positivo numa *string* de dígitos.

Problema 2.6 – Médio

Escreva uma função recursiva que converta uma *string* de dígitos no número inteiro por ela representado.

Problema 2.7 – Médio

Escreva uma função recursiva que calcule a soma de uma tabela de inteiros. O protótipo da função é:

```
int soma(int tab[ ], int numElems);
```

Problema 2.8 – Médio

Escreva uma função recursiva que devolva o maior número de uma tabela de inteiros. O protótipo da função é:

```
int maior(int tab[ ], int numElems);
```

Problema 2.9 – Difícil

Escreva uma função recursiva que inverta uma tabela de inteiros de comprimento **n**, utilizando apenas uma variável auxiliar.

Problema 2.10 – Difícil

Escreva uma função recursiva que determine se um número inteiro é ou não uma capicua. Um número é uma capicua se a primeira parte do número é uma imagem espelhada da segunda. Dois exemplos de capicuas são os seguintes números: 1234321 e 56788765. O protótipo da função é:

```
bool capicua(int num);
```

Sugestão: Pode ir comparando recursivamente os dois dígitos extremos do número. Para tal, crie uma função auxiliar que permita eliminar esses dois dígitos de um dado número.

Problema 2.11 – Médio

Implemente uma função recursiva que calcule a função matemática F(n) cuja expressão é indicada a seguir. A variável n representa um número inteiro. Para n negativo, a função que vai implementar tem de devolver -1. Implemente também uma função de teste (função `main()`).

$$F(n) = \begin{cases} 10\,, & \text{para } n = 0 \\ 20\,, & \text{para } n = 1 \\ F(n-1) + 2 \times F(n-2), & \text{para } n > 1 \end{cases}$$

Problema 2.12 – Fácil (saiu no teste de frequência de 12/03/2014)

Implemente a função recursiva F(n) indicada a seguir, assumindo que n é inteiro. Para n negativo, a função retorna 0. Implemente também uma função de teste (função `main()`).

$$F(n) = \begin{cases} 1\,, & \text{para } n = 0 \\ 3 \times F(n-1) + 3, & \text{para } n > 0 \end{cases}$$

Problema 2.13 – Médio (saiu no teste de frequência de 25/03/2015)

Escreva a função recursiva `void decimalToOctal(int num)` que imprime no ecrã a conversão em octal (base 8) do número `num` em base decimal (base 10) passado como parâmetro da função.
Nota: Recorde que a conversão consiste em realizar sucessivamente divisões inteiras por 8 do número passado à função. Os restos de cada divisão inteira formam os dígitos do número codificado em octal.
Exemplo: $156_{10}=234_8$, ou seja: 156/8(4)=19 /8(3)=2 /8(2)=0
Os números entre parêntesis correspondem ao resto da divisão.

Problema 2.14 – Fácil (saiu no exame de recurso de 03/07/2017)

Utilizando uma função recursiva, implemente a função indicada a seguir:

$$\begin{cases} T(n \leq 0) = 0 \\ T(1) = 1 \\ T(n \geq 2) = (T(n-1) + T(n-2))/2 \end{cases}$$

Problema 2.15 – Médio (adaptado do teste de frequência de 13/04/2018)

Escreva o código da função recursiva `unsigned int octalParaDecimal(unsigned int num)` que devolve a conversão em decimal (base 10) do número `num` em octal (base 8) passado como parâmetro da função. Assuma que `num` é sempre um número representado corretamente em octal, *i.e.* que só contém dígitos de 0 a 7. Recorde que a conversão consiste em realizar a soma pesada dos dígitos octais do número passado à função, pesados com potências de 8 de expoente dependente da posição dos dígitos. A função não deve escrever nada no ecrã. Implemente também uma função de teste (função `main()`).
Nota: O resto da divisão inteira de `num` por 10 permite extrair o dígito mais à direita. O quociente desta divisão inteira permite obter os restantes dígitos. A conversão recursiva consiste em somar o dígito mais à direita com a conversão do número constituído pelos restantes dígitos (problema mais pequeno) multiplicada por 8.
Exemplo: $273_8 = 187_{10} = 3 * 8^0 + 7 * 8^1 + 2 * 8^2 = (3) + 8 * (7 + 2 * 8^1) = (3) + 8 * [\ (7) + 8 * (2)\ ]$.

WORKSHEET 3
PROCEDURAL PROGRAMMING IN C++

3.1 Objectives

This assignment aims at practicing procedural programming in C++, with a special emphasis on the aspects mentioned explicitly in the next section.

3.2 Introduction

Although many of the concepts involved in this assignment were already learnt by the students in the Computer Programming subject by using the C language, there are a few aspects that are differently, or slightly differently implemented in C++, namely:

- Comments within the source code;
- Input from keyboard and output to console;
- String manipulation;
- Input/Output from/to Files;
- Dynamic memory allocation.

There are also a few concepts that are entirely new in C++:

- Reference variable;
- Function returning a reference;
- Function overloading and default arguments;
- Namespace.

While in C, only `/*` and `*/` delimiters can be used to insert either single- or multi-line comments in the source code, the `//` delimiter can also be used in C++ to insert single-line comments.

The input from keyboard and output to console, as well as to files, appears differently in C++, compared with the C language, by using streams provided by `iostream` and `fstream` libraries. Also, the `iomanip` library provides a set of convenient stream manipulators that help a lot in formatting conveniently the text output, either on console or in text files.

The manipulation of C++ strings is also more intuitive and powerful than with C strings, by using the `string` data type provided by the library `string`.

Dynamic memory allocation is native in C++ language through the reserved words `new` and `delete`.

The C++ language allows defining function overloading, i.e. implementing multiple functions with the same name, provided that they differ in the list of arguments. The definition of default arguments is also possible in C++, providing a means to omit one or more arguments when calling a function with default arguments.

3.3 Passing arguments to a function by reference

Reference variables were introduced in C++ (they do not exist in C) and can be seen as alias to variables. They are important to pass arguments to functions by reference and to implement functions that return a reference.

A method (available in C++ but not available in C) to pass arguments to a function and allowing them to be changed within the function is to use argument passing by reference. When a reference is passed into a function, any change to that reference undergoes as if that change were made directly on the original variable.

The previous example seen in Worksheet 1, of a function that should swap the value of two integer variables, can be easily adapted to arguments passed by reference:

```
void swap(int &num1, int &num2) {
  int temp;
  temp = num2;
  num2 = num1;
  num1 = temp;
}
//------------------------------------------------------
int main() {
  int x = 4, y = 2;

  printf("Before calling swap(), x=%d and y=%d\n", x, y);
  swap(x, y);
  printf(" After calling swap(), x=%d and y=%d\n", x, y);
}
```

3.4 Suggested Exercises

In the set of exercises listed below, you cannot use C functions whenever the programming feature in hands can be implemented differently in C++ (most often, in a more powerful way). For instance, you cannot use `scanf()` and `printf()` for input/output in this assignment; you have to use alternatively `cin` and `cout`.

Exercise 3.1 – `for` loop, `cout` – Medium

Write a function that, given an integer number as input argument, draws a diamond in the console whose side contains that number of asterisks. Write a small program (i.e. the `main()` function) to test your function. You can only use C++ methods for input/output (i.e. `cin` and `cout`). Check the usefulness of adding the line `using namespace std;` in your program. Example of the diamond obtained when the given number is 4:

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

Exercise 3.2 – Input/Output, Output Formatting – Easy

Consider a function that multiplies a bi-dimensional matrix, containing real numbers represented in floating-point notation, by a real number (also a number represented in floating-point notation). Assume that the matrix has `COLUMNS =3` (a global constant) columns.

a) Choose an adequate prototype for the function and implement the function. Note that the function needs to receive the number of lines of the matrix as argument.

b) Write a program to test the function through the following example of a matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 1 & 0.5 \end{bmatrix}$$, with `COLUMNS =3`.

The test program must ask the user to input a real number to be multiplied by the matrix and print the matrix before and after the function call. With this purpose, implement an auxiliary function to format adequately the output of the matrix on the console by using manipulators available in the `iomanip` library. Each matrix entry must be presented aligned to the right, in fixed-point notation with 3 digits after the decimal point. The matrix entries in the same column must appear properly aligned on the console. Assume that each matrix entry has up to 3 digits before the decimal point, i.e. it is limited to the interval]-1000.000; +1000.000[.

Exercise 3.3 – Functions, Text Files – Difficult (appeared in the test held on 25/03/2015)

Consider a structure (defined with the reserved word `struct`) that stores data about a customer of a dietetics center: `name` (C string, size 50), `female` (boolean flag that is `true` if the customer is a woman, `false` otherwise), `weight` (`float`), `height` (`float`) and `bmi` (`float` that stores the BMI – Body Mass Index).

a) Declare the `struct`.

b) Implement the function `computeBMI()` that receives as argument a structure passed by reference, computes the BMI and stores the result in the `bmi` field of the structure. Assume that the structure already contains valid values in `weight` and `height` fields.
Note: `bmi = weight / (height * height).`

c) Implement the function `updateBMI()` that receives as arguments a pointer to the first element of an array of structures and the size of the array, and updates the field `bmi` of every element in the array. Note: this function must call the function `computeBMI()`.

d) Write the function `listCustomers()` that receives as arguments an array of customers, its size, a BMI threshold, and a filename. The function writes a text file that lists every customer in the array whose BMI is greater or equal to the threshold. The function returns the number of customers written to the file (zero if none); the function returns -1 if some error occurs when opening the file. If the file already exists, data is added at the end of the file without destroying the pre-existent data. The data written in each line of the text file (name, weight and BMI of a customer) must be properly formatted by using manipulators of the `iomanip` library: name aligned to the left in column 1; weight and BMI aligned to the right in columns 60 and 70, using fixed-point notation with 2 digits after the decimal point.

e) Write a test program to test all the functions implemented in the previous points.

Exercise 3.4 – Functions, Binary Files, Dynamic Memory Allocation – Difficult

Consider again the `struct` datatype defined in exercise 3.3.

a) Implement a function that saves to a binary file all the data stored in an array of structures.
b) Implement a function that opens a binary file and retrieves the data of a customer stored in the binary file at the index `i`; assume that the index of the first customer is zero. If the function retrieves the data successfully, it returns `true`, otherwise it returns `false`.
c) Implement a function that adds data of a new customer at the end of a binary file.
d) Implement a function that reads all structures stored in a binary file and return them in an array. The function allocates dynamically memory for the array. The function returns a pointer to the first element of the created array and the size of the array.
e) Implement a `main()` function to test the functions implemented in points a), b), c), and d). The program should show a menu with the following options: 1-Input data of customers from keyboard to an array (implement an auxiliary function to input data of one customer from the keyboard and return a structure); 2-Save customers data to file; 3-Retrieve data of a customer from file; 4-Add new customer at end of file; 5-List all customers stored in file.

Exercise 3.5 – Strings – Medium

Implement a program that builds the list of the authors of a publication. The list of authors should be stored in memory as a C++ string. Program a loop to ask the user, in each iteration, the name of each author, with the format "`<first_name_author< <last_name_author>`". The loop ends when the user inputs an empty string, i.e. an empty name. In the end of the loop execution, the result of the processing should be a string containing the list of authors with the format "`<first_name_author1> <last_name_author1>, ..., <first_name_authorN> <last_name_authorN>`". Extend the program to abbreviate the first names in the string containing the list of authors, e.g. "`Winston Churchill, Franklin Roosevelt`" becomes "`W. Churchill, F. Roosevelt`". Hint: The method `find_first_of()` allows searching for the first occurrence of a character at or after a given position in a string.

Exercise 3.6 – Function Overloading, Functions with Default Arguments – Easy

Consider a family of functions `<return_type> sum(…)` that compute the sum of a set of values; each function has a different argument list. Implement the functions indicated below and build a small program to call and test each one of them.

a) `int sum(int a, int b, int c = 0);  // sum 2 or 3 integers (3rd argument optional)`
b) `int sum(int array[], int size);  // sum the elements of an array of integers`
c) `void sum(int array1[], int size1, const int array2[], int size2);  // sum 2 arrays`
d) `float sum(float a, float b, float c = 0.0);  // sum 2 or 3 floats`
e) `float sum(float array[], int size);  // sum the elements of an array of floats`
f) `void sum(float array1[], int size1, const float array2[], int size2);// sum 2 arrays`

Exercise 3.7 – Reference Variable, Function – Medium

Explain why the following functions are not correct:

```
int &badIdea1() {
   int i;
   return i;
}
//---------------------------
int &badIdea2(int arg) {
   return arg;
}
```

If you do not know how to answer, create a small program that calls both functions, check the warnings provided by the compiler, and try to interpret them...

Exercise 3.8 – Arguments Passed by Reference to Functions – Easy

Write the function `void invert(int array[], int dim)` that swaps the position of the elements of the array of size `dim` (e.g. the first element becomes the last one, the second becomes the penultimate, etc.). Design its code so that it calls an auxiliary function `swapRef()` that swaps the value of the integer variables passed by reference as arguments to it.

FICHA 4
ALGORITMIA E ALGORITMOS DE ORDENAÇÃO

4.1 Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:

- Entender o funcionamento de dois algoritmos simples de ordenação, o *selection sort* e o *bubble sort*;
- Saber implementar estes dois algoritmos em C/C++ para ordenar uma tabela de inteiros, uma tabela de *strings* e uma tabela de *structs*, tanto por ordem crescente, como por ordem decrescente;
- Saber utilizar a função `qsort()` disponível no ANSI C para ordenar uma tabela de dados;
- Saber ordenar um ficheiro de dados desordenados, lendo a totalidade do ficheiro para memória, ordenando os dados em memória e finalmente escrevendo os dados ordenados de volta para o ficheiro.
- Saber calcular a complexidade de um algoritmo, expressando-a na notação O-grande.

4.2 Ordenação

A ordenação de informação sempre foi uma necessidade nas mais variadas aplicações:

- Ordenação de livros em bibliotecas, por título, por autor ou por outros campos;
- Ordenação dos contribuintes de uma determinada repartição de finanças;
- Ordenação de nomes numa lista telefónica ou numa simples agenda telefónica pessoal;
- Ordenação de uma lista de clientes de uma empresa;
- Ordenação de peças contidas num armazém.

Facilmente se conclui que a ordenação de dados surge com alguma frequência como um problema a resolver em programação. É normal que o programador recorra a funções/processos já implementados para efetuar a ordenação pretendida, não tendo portanto que implementar um algoritmo de ordenação. Todavia, é objetivo desta ficha exercitar o aluno na implementação de alguns algoritmos simples de ordenação.

Quando usar ordenação de dados?

É difícil estar a indicar todas as situações em que se deve ou não utilizar a ordenação de dados. Pode-se no entanto utilizar a seguinte regra orientadora: um conjunto de dados deve estar ordenado sempre que as tarefas executadas sobre ele se tornem consideravelmente mais fáceis/rápidas de executar pelo simples facto de os dados estarem ordenados.

Imagine que a base de dados em papel do registo civil dos cidadãos portugueses (a base de dados dos cartões de cidadão) não estava ordenada pelo número de identificação. Se fosse pedido a um funcionário para pesquisar fichas de cidadãos portugueses pelo seu número de identificação, ele teria que percorrer em média, e por cada pesquisa, metade das fichas de todos os portugueses (5 milhões de fichas). Isto seria obviamente inviável! Contudo, se os ficheiros em papel estiverem ordenados pelo número de identificação, ele só teria que ir à secção onde está o número respetivo e ler os dados de uma única ficha.

A base de dados dos cartões de cidadão tem também que estar ordenada por nome: imagine que se pede a um funcionário que procure a ficha de um português com um determinado nome, sem se conhecer o seu número de identificação. Se a base de dados não estiver ordenada também pelo nome, ele teria que ver as fichas de quase todos os portugueses, comparando o nome até encontrar o pretendido. Se a base de dados estiver também ordenada pelo nome, então a tarefa será bastante mais fácil.

A conclusão a tirar deste exemplo é que a existência de dados ordenados permite, na grande maioria das vezes, pesquisas muito mais rápidas.

Que algoritmo de ordenação devo usar?

Conhece-se atualmente muitos algoritmos de ordenação. Uns são muito rápidos a ordenar, mas bastante complexos, outros são mais lentos, mas mais simples de implementar e de entender.

Um programador que necessite de ordenar os seus dados recorre habitualmente a algoritmos já implementados ou que sabe implementar rapidamente. Ele só precisa de recorrer aos algoritmos mais eficientes e complexos se estiver a lidar com conjuntos de dados muito grandes, algures entre centenas de milhar a milhões de dados. Para ordenar conjuntos de dados de menor dimensão, um dos algoritmos mais simples cumpre quase de certeza a tarefa de ordenação de forma eficiente. O próprio C tem definida a função `qsort()` que implementa um algoritmo de ordenação bastante eficiente: o *quick sort*. De seguida, são apresentados alguns dos algoritmos de ordenação mais simples.

4.3 Ordenação por seleção (*selection sort*)

4.3.1 O algoritmo de seleção

Um dos algoritmos mais simples para ordenar um conjunto de dados é o algoritmo de seleção (*selection sort*). O princípio de funcionamento do algoritmo é o seguinte:

Imagine-se que se quer colocar por ordem crescente uma tabela de números inteiros. Percorre-se a tabela do início ao fim para encontrar o menor dos números, depois troca-se esse número com o existente na 1ª posição. Volta-se a repetir este processo, mas agora só a partir da 2ª posição. Depois o menor elemento encontrado é trocado com o elemento na 2ª posição. Repetindo este procedimento partindo sempre de uma posição mais à frente que a anterior, ficaremos no final do processo com a tabela ordenada por ordem crescente.

Vejamos um exemplo:

tab =	98	42	41	**23**	37	58
índice	0	1	2	**3**	4	5

Percorrendo a tabela do índice 0 ao 5 verificamos que é no índice 3 que se encontra o menor número da tabela. Fazemos pois uma troca entre o número no índice 0 (primeiro) e o número no índice 3.

tab =	23	42	41	98	**37**	58
índice	0	1	2	3	**4**	5

(A cinzento encontram-se os números ordenados e a branco os que ainda estão por ordenar)

Percorrendo a tabela do índice 1 ao 5, selecionamos o menor dos números, neste caso na posição 4. Trocamos pois os elementos nas posições 1 e 4.

tab =	23	37	**41**	98	42	58
Índice	0	1	**2**	3	4	5

Percorrendo do índice 2 ao 5, selecionamos o menor dos números, neste caso na posição 2. Nesta situação o número até já se encontra na posição correta.

tab =	23	37	41	98	**42**	58
Índice	0	1	2	3	**4**	5

Percorrendo do índice 3 ao 5, selecionamos o menor dos números, neste caso na posição 4. Trocamos pois os elementos nas posições 3 e 4.

tab =	23	37	41	42	98	**58**
índice	0	1	2	3	4	**5**

Percorrendo do índice 4 ao 5, selecionamos o menor dos números, neste caso na posição 5. Trocamos pois os elementos nas posições 4 e 5.

tab =	23	37	41	42	58	98
índice	0	1	2	3	4	5

Neste ponto, podemos concluir que o elemento na posição 5 é o maior de todos e que a tabela já se encontra ordenada.

Algumas notas:

- Este algoritmo pode ser implementado quer para ordenar por ordem crescente (menor número no início) quer por ordem decrescente (maior elemento no início).

- O algoritmo também pode ser implementado partindo do fim para o início, retendo os maiores valores no fim (supondo uma ordenação crescente).

4.3.2 Implementação em C/C++ do algoritmo de seleção

No programa abaixo, mostra-se uma possível implementação do algoritmo de seleção para ordenar uma tabela de inteiros por ordem crescente.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela

//-----------------------------------------------------------------------------
int indiceMin(int tab[ ],int inicio,int fim) // Devolve o índice do menor elemento da
{                                            // Tabela entre os índices inicio e fim
  int i, idxMenor;
```

```
  idxMenor=inicio;                // Assume que o primeiro elemento é o menor
  for(i=inicio+1; i<=fim; i++)    // Percorre a tabela entre inicio+1 e fim
     if(tab[idxMenor]>tab[i])     // Se o elemento atual é menor do que aquele que
        idxMenor=i;               // considerávamos menor então o elemento actual passa
                                  // a ser considerado o menor.
  return(idxMenor);
}
//-----------------------------------------------------------------------------------
// Troca o elemento com índice i pelo elemento com índice j na tabela tab
void troca(int tab[ ], int i, int j) {
  int aux = tab[i];
  tab[i] = tab[j];
  tab[j] = aux;
}
//-----------------------------------------------------------------------------------
// Ordena uma tabela tab com comp elementos
void ordenacaoSelecao(int tab[ ],int comp) {
  for (int i = 0; i < comp-1; i++)
    troca(tab,i,indiceMin(tab,i,comp-1));  // Troca o elemento actual com o menor
}                                          // elemento dos restantes valores da tabela
//-----------------------------------------------------------------------------------
// Imprime os primeiros numElem elementos da tabela
void imprimeTabela(const char *nomeTab, int tab[], int numElem) {
  cout << nomeTab << " = ";        // Imprime nome da tabela =
  for (int i=0; i<numElem; i++) {  // percorre todos os elementos da tabela
    cout << tab[i] << " ";
  }
  cout << endl;   // muda de linha no final
}
//-----------------------------------------------------------------------------------
int main() {
  // Inicializa a tabela como constante
  int tabela[N] = {98, 42, 41, 23, 37, 58};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoSelecao(tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
```

No programa anterior a função de ordenação `ordenacaoSelecao()` recorre a duas funções auxiliares: a primeira para encontrar o menor valor na tabela de uma dada posição até ao fim da tabela e a segunda para trocar dois elementos da tabela. Foi também criada uma função auxiliar `imprimeTabela()` que permite visualizar os elementos da tabela, antes e depois da tabela ser ordenada.

Note que as alterações a fazer à função `ordenacaoSelecao()` para que esta ordene por ordem decrescente, em vez de o fazer por ordem crescente, são mínimas. Como exercício, adapte esta função para permitir a ordenação por ordem decrescente.

4.4 Ordenação por bolha (*Bubble sort*)

4.4.1 O algoritmo de ordenação por bolha

O algoritmo de ordenação por bolha (*bubble-sort*) é um outro algoritmo simples de ordenação que consiste em "deslocar" repetidas vezes ao longo da tabela uma "bolha ordenadora de 2 elementos". A bolha corresponde a dois elementos sucessivos de uma tabela. Verifica-se se esses dois elementos já estão ou não na ordem certa. Se não estiverem, então são trocados. Se este procedimento for repetido um número de vezes suficiente, a tabela fica ordenada.

Voltando à tabela do exemplo anterior, coloca-se uma bolha ordenadora nos primeiros 2 elementos:

tab =	98	42	41	23	37	58
índice	0	1	2	3	4	5

Uma vez que estes dois elementos estão desordenados (42 devia estar antes do 98), são trocados. A bolha desloca-se de seguida para a direita. O resultado após a troca é:

tab =	42	98	41	23	37	58
índice	0	1	2	3	4	5

O processo repete-se: neste caso o 41 é menor do que o 98. Tem que ocorrer mais uma troca dentro da bolha para a tabela ficar ordenada.

tab =	42	41	98	23	37	58
índice	0	1	2	3	4	5

O procedimento é repetido até a bolha chegar ao final da tabela.

tab =	42	41	23	37	58	**98**
índice	0	1	2	3	4	5

Como se pode observar, a tabela ficou com o maior elemento na posição correta, ou seja na última posição da tabela. De seguida, volta-se a colocar a bolha no início da tabela.

tab =	42	41	23	37	58	**98**
índice	0	1	2	3	4	5

Repete-se este procedimento até que a bolha chegue ao penúltimo elemento (o último elemento já está ordenado).

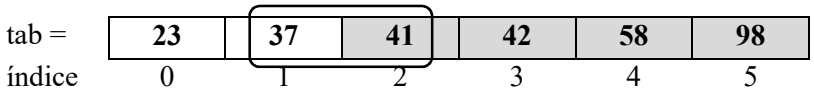
tab =	41	23	37	42	**58**	**98**
Índice	0	1	2	3	4	5

De seguida, volta-se a colocar a bolha no início e faz-se percorrer a tabela até ao antepenúltimo elemento:

tab =	23	37	41	**42**	**58**	**98**
índice	0	1	2	3	4	5

Neste momento, tem-se a certeza que, seja qual for a tabela, o número na posição 3 já está na posição correta. Neste exemplo em concreto, a tabela até já está toda ordenada, mas como o algoritmo tem de ser genérico e dar o resultado pretendido seja qual for a tabela, o processo tem que continuar.

Volta-se portanto a fazer percorrer a bolha ao longo da parte desordenada da tabela.



Quando a bolha de ordenação chega ao fim, sabe-se que o número na posição 2 também já está ordenado. Mais, sabe-se que, uma vez que não ocorreu qualquer troca de elementos, então a tabela já está ordenada, pois todos os pares consecutivos da tabela encontram-se na posição certa. Esta é uma particularidade do algoritmo de ordenação por bolha que o pode tornar ligeiramente mais rápido que o algoritmo de seleção.

Algumas notas:

- O algoritmo de ordenação por bolha pode ser implementado, quer para ordenar por ordem crescente (menor número no início) quer por ordem decrescente (maior número no início).

- O algoritmo pode também ser utilizado deslocando a "bolha" do fim para o início. Nesse caso, a bolha deixa de parar no primeiro valor da tabela para parar sucessivamente numa posição mais à frente.

4.4.2 Implementação em C/C++ do algoritmo de ordenação por bolha

No programa abaixo, apresenta-se uma possível implementação do algoritmo de ordenação por bolha para ordenar uma tabela do tipo `double` por ordem crescente. Mais uma vez, recorre-se a uma função auxiliar `troca()` que executa a tarefa de trocar dois elementos caso estes estejam desordenados.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
// Troca o elemento com índice i pelo elemento com índice j na tabela tab
void troca(double tab[ ], int i, int j) {
  double aux = tab[i];
  tab[i] = tab[j];
  tab[j] = aux;
}
//-----------------------------------------------------------------------------
void ordenacaoBolha(double tab[],int comp) {
  bool houveTroca;

  for (int i = 0; i < comp-1; i++) {
    houveTroca = false;
    for (int j = 1; j<comp-i; j++)
      if(tab[j]<tab[j-1]) {
        troca( tab, j-1, j);
        houveTroca = true;
      }
    if (!houveTroca) break;
  }
}
//-----------------------------------------------------------------------------
int main() {
  // inicializa a tabela como constante
  double tabela[N] = {9.8, 4.2, 4.1, 2.3, 3.7, 5.8};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoBolha (tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Mais uma vez, as alterações a fazer à função `ordenacaoBolha()` para que esta ordene por ordem decrescente em vez de o fazer por ordem crescente são mínimas (na realidade basta alterar a condição que determina a troca dos valores).

A variável `houveTroca` foi acrescentada para tornar o algoritmo mais rápido e permitir parar a ordenação a partir do momento que se saiba que a tabela se encontra ordenada (não houve trocas na interação anterior). Caso se retire todas as referências a esta variável o algoritmo funciona à mesma.

4.5 Ordenação por inserção (*insertion sort*)

4.5.1 O algoritmo de ordenação por inserção

O algoritmo de ordenação por inserção (*insertion sort*) é outro algoritmo simples de ordenação que consiste em ir ordenando os elementos de uma tabela como se faz com um baralho de cartas: inserindo na ordem correta os elementos a partir do segundo (o primeiro elemento está naturalmente ordenado). Considera-se que a tabela está dividida em duas partes. A parte esquerda está ordenada (começa com um único elemento, o primeiro elemento da tabela). A parte direita está desordenada e vai-se inserindo, ordenadamente, o primeiro elemento da parte desordenada na parte ordenada. Este processo é repetido até a parte desordenada desaparecer. Neste processo iterativo é usada uma variável temporária que guarda o elemento a inserir enquanto os elementos da parte esquerda (a já ordenada) vão sendo deslocados para a direita até se chegar à posição de inserção, inserindo-se aí o elemento guardado na variável temporária. Dos algoritmos de complexidade quadrática, este é o mais eficiente. Quando se pretende inserir um novo elemento numa tabela já ordenada, deve ser usado um algoritmo baseado neste.

Voltando à tabela do exemplo anterior, começa-se por considerar a parte já ordenada (a cinzento) apenas com o primeiro elemento da tabela, estando os restantes elementos desordenados. A seguir copia-se o elemento a inserir, o de índice 1, na variável temporária `temp`.

Tab =	98	**42**	41	23	37	58
índice	0	**1**	2	3	4	5

`temp` = 42

Percorrendo a tabela desde o índice 1 até ao índice 0, vai-se deslocando os elementos para a direita até encontrarmos um elemento menor que o elemento a inserir. Neste caso, só um elemento é deslocado e chega-se ao índice 0, ficando:

	↓					
Tab =	98	**98**	41	23	37	58
índice	0	**1**	2	3	4	5

`temp` = 42

Copia-se o valor da variável temporária para a tabela e prossegue-se da mesma forma. Agora a tabela já tem dois elementos ordenados e o novo elemento a inserir (o 3º) foi copiado para a variável temporária:

Tab =	42	98	**41**	23	37	58
índice	0	1	**2**	3	4	5

`temp` = 41

Percorrendo agora a tabela desde o índice 2 até ao índice 0, vai-se deslocando os elementos para a direita até encontrarmos um elemento menor que o a inserir. Neste caso dois elementos são deslocados e chega-se ao índice 0, ficando:

			↓			
Tab =	42	42	**98**	23	37	58
índice	0	1	**2**	3	4	5

temp = 41

Depois de se inserir ordenadamente o 3ºelemento da tabela e quando se vai inserir o 4º elemento, fica:

Tab =	41	42	98	**23**	37	58
índice	0	1	2	**3**	4	5

temp = 23

Depois de se deslocarem os elementos para a direita, fica:

		↓				
Tab =	41	41	42	**98**	37	58
índice	0	1	2	**3**	4	5

temp = 23

Depois de se inserir o 4ºelemento da tabela e quando se vai inserir o 5º elemento, temos:

Tab =	23	41	42	98	**37**	58
índice	0	1	2	3	**4**	5

temp = 37

Depois de se deslocarem os elementos para a direita, fica:

		↓				
Tab =	23	41	41	42	**98**	58
índice	0	1	2	**3**	4	5

temp = 37

Depois de se inserir o 5ºelemento da tabela e quando se vai inserir o 6º elemento, temos:

Tab =	23	37	41	42	98	**58**
índice	0	1	2	3	4	**5**

temp = 58

Depois de se deslocarem os elementos para a direita, fica:

					↓	
Tab =	23	37	41	42	98	**98**
índice	0	1	2	3	4	**5**

temp = 58

Finalmente, depois de se inserir o último elemento, obtemos a tabela ordenada:

Tab =	23	37	41	42	58	98
índice	0	1	2	3	4	5

4.5.2 Implementação em C/C++ do algoritmo de inserção

No programa abaixo, mostra-se uma possível implementação do algoritmo de inserção para ordenar uma tabela de inteiros por ordem crescente.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
// ordena por inserção vetor v com n elementos
void ordenacaoInsercao(int v[], int n) {
   for (int i = 1; i < n; i++) {  // n-1 iterações
      int aux = v[i];            // elemento a inserir copiado para var temporaria
      int j;                     // vai conter índice onde inserir
      for (j = i; (j > 0) && (aux < v[j-1]); j--)  // desloca elementos para direita
         v[j] = v[j-1];                            // ate' chegar à posição de insercao
      v[j] = aux;                                 // insere elemento no índice j
   }
}
//-----------------------------------------------------------------------------
// Imprime os primeiros numElem elementos da tabela
void imprimeTabela(const char *nomeTab, int tab[], int numElem) {
  cout << nomeTab << " = ";        // Imprime nome da tabela =
  for (int i=0; i<numElem; i++) {  // percorre todos os elementos da tabela
    cout << tab[i] << " ";
  }
  cout << endl;   // muda de linha no final
}
//-----------------------------------------------------------------------------
int main() {
  // inicializa a tabela como constante
  double tabela[N] = {9.8, 4.2, 4.1, 2.3, 3.7, 5.8};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoInsercao(tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Mais uma vez, as alterações a fazer à função `ordenacaoInsercao()` para que esta ordene por ordem decrescente em vez de o fazer por ordem crescente são mínimas (na realidade basta alterar a condição que determina a deslocação dos elementos).

4.6 Como usar a função qsort (ordenação «rápida» ou *quick sort*)

O C possui uma função de ordenação denominada `qsort()` definida na biblioteca `<cstdlib>`. Esta função implementa o algoritmo de ordenação «rápida», conhecido por *Quick Sort* e tem a capacidade de ordenar dados de diferentes tipos. Para tornar esta função genérica, é necessário fornecer uma função que permita comparar o tipo de dados a ordenar. A definição da função é a seguinte:

```
void qsort(void *base, size_t nelem, size_t width,
           int (*fcmp)(const void *, const void *));
```

`base` = ponteiro para o 1º elemento da tabela;

`nelem` = número de elementos dessa tabela;

`width` = número de bytes ocupado por cada elemento (use o `sizeof()` para o ajudar nesta tarefa);

`fcmp` = função de comparação entre dois elementos. Esta função tem que ser implementada pelo programador de acordo com o tipo de dados a utilizar como critério de ordenação. Deve devolver um número menor do que zero se o primeiro elemento for menor do que o segundo, zero se os dois elementos forem iguais, e um número maior do que zero se o primeiro elemento for maior que o segundo.

Vejamos o potencial da função `qsort()`**:** Imagine que se pretende ordenar uma tabela de fichas que contêm dois dados relativos a uma pessoa: nome e idade. Pretende-se ordenar esta tabela pelos nomes das pessoas. Para conseguir este propósito, poderemos utilizar a função `qsort()` da seguinte forma:

```
#include <iostream>   // cout
#include <iomanip>    // setw()
#include <cstring>    // strcmp()
#include <cstdlib>    // qsort()
using namespace std;
//-----------------------------------------------------------------------------
const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
struct TFicha {  // definição de 1 ficha
  int idade;     // idade de cada pessoa
  char nome[21]; // nome da pessoa (até 20 caracteres + '\0')
};
//-----------------------------------------------------------------------------
int comparaNome(const void *ficha1, const void *ficha2) { // compara 2 nomes da ficha
  return strcmp( ((TFicha *)ficha1)->nome, ((TFicha *)ficha2)->nome);
  // usou-se a função strcmp que devolve um resultado igual ao pretendido
}
//-----------------------------------------------------------------------------
void imprimeTabela(const char *nomeTab, TFicha tab[], int numElem) {
  cout << nomeTab << " = " << endl;  // Escreve nomeTab =
  for (int i=0; i < numElem; i++) {  // percorre todos os elementos da tabela
    cout << "Nome: "<< setw(20)<< tab[i].nome << "   Idade: " << tab[i].idade << endl;
  }
}
//-----------------------------------------------------------------------------
int main() {
  // inicia-se a tabela no momento da definição
  TFicha tabela[N] = {35, "Rui" , 21, "Manuel", 20, "Jorge",
                      16, "Luis", 19, "Pedro" , 25, "Artur"};

  imprimeTabela("Tabela original", tabela, N);
  qsort((void *)tabela, N, sizeof(TFicha), comparaNome);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Notas:

- O uso de `qsort()` obriga à inclusão da biblioteca `<cstdlib>` ou `<search.h>`;
- Quando se pretende armazenar nomes com 20 caracteres, por exemplo, deve-se somar mais um caráter para a terminação da *string*: o caráter `'\0'` (caráter com código ASCII 0);
- A função `compara_nome()` usa alguns *typecasts* (adaptação de tipos) necessários a uma correta utilização do `qsort()`. Na função `compara_nome()`, usou-se uma função da biblioteca `<cstring>` do C++, a função `strcmp()`, que tem um comportamento igual ao desejado para a função de comparação do

`qsort()` – devolve zero se as duas *strings* são iguais, um valor menor do que zero se a primeira *string* é menor do que a segunda, e um valor maior do que zero se a primeira *string* for maior do que a segunda;
- Esta secção pretendeu mostrar como fazer a ordenação de uma tabela sem haver a necessidade de implementar explicitamente um dos algoritmos estudados na secção anterior.

4.7 Geração de números aleatórios

Muitas vezes existe a necessidade de se recorrer à geração de números aleatórios, para os mais variados fins: nos jogos de computador estes são utilizados muitas vezes para criar um comportamento imprevisível para o jogador; nos simuladores para dar um comportamento mais próximo da realidade variável dos sistemas reais, etc.

Nesta ficha, a geração de números aleatórios é útil para testar os algoritmos de ordenação. Em vez de se definir explicitamente uma tabela de números inteiros, podemos recorrer a uma função geradora de números aleatórios para preencher uma tabela com números aleatórios. Imagine que se pretende preencher uma tabela `tab[]` de 20 inteiros, com números entre 0 e 100, escolhidos ao acaso. O código a seguir apresentado permite-nos precisamente isso:

```
for (int i=0; i<20; i++) {  // percorre os 20 elementos da tabela
  tab[i] = rand() % 101;    // atribui um valor ao "calhas" entre 0 e 100
}
```

A função `rand()` devolve um número inteiro aleatório compreendido entre `0` e `RAND_MAX` (constante definida em `<cstdlib>`). Para se gerar um inteiro compreendido entre 0 e 100, basta utilizar o módulo da divisão inteira (`%`). O resto da divisão de um número qualquer por 101 será sempre um número entre 0 e 100.

Imagine agora que se pretende preencher a mesma tabela `tab[]` de 20 inteiros escolhidos aleatoriamente, mas com números entre `valor_inicial` e `valor_final`. O código poderia ser:

```
for (int i=0; i<20; i++) {  // percorre os 20 elementos da tabela
  tab[i] = valor_inicial + rand() % ( valor_final - valor_inicial + 1 );
}   // atribui um valor ao "calhas" entre valor_inicial e valor_final (inclusive)
```

Para gerar números reais com uma casa decimal podemos gerar inteiros correspondendo às décimas desses números reais e depois fazer a divisão real por 10.0. No nosso caso ficaria:

```
// atribui um valor real ao "calhas",
// entre valor_inicial/10 e valor_final/10, inclusive, com "precisão" às décimas
   tab[i] = ( valor_inicial + rand() % ( valor_final – valor_inicial + 1) ) / 10.0;
```

Se experimentar um destes códigos, facilmente constatará que a sequência de números aleatórios gerada é sempre a mesma. Isto deve-se ao facto da função `rand()` basear a geração dos números aleatórios na utilização de um determinado número inteiro (designado por *semente*) como base para a sequência aleatória. Se esse número for sempre o mesmo, então a sequência de números aleatórios também será sempre a mesma. Para alterar este tipo de comportamento existe a função `srand(int seed)` que permite definir a semente a utilizar pelo `rand()`. Se alteramos o valor da semente a cada utilização do programa, garantimos que a sequência de números gerados vai ser sempre diferente. A função `time()` (definida em `time.h`) que devolve o número de segundos desde a meia-noite do dia 1 de Janeiro de 1970, pode ser utilizada para garantir que a semente dos números aleatórios é sempre diferente a cada execução do

programa. O código que permite inicializar a semente com valores diferentes a cada execução do programa será o seguinte:

```
#include <time.h>

...
srand((unsigned)time(NULL));
...
```

Sugere-se uma pesquisa às funções `srand()` e `rand()` para aprofundar os seus conhecimentos sobre geração de números aleatórios.

4.8 Ordenar um ficheiro de texto

Até agora apenas se considerou a ordenação de dados numa tabela em memória: uma tabela de inteiros, uma tabela do tipo `double` ou uma tabela de fichas. Em aplicações reais, os dados estão armazenados em ficheiros, o que determina que por vezes as operações de ordenação tenham que ser efetuadas em ficheiros.

A ordenação de um ficheiro de texto pode ser feita recorrendo temporariamente a uma tabela de *strings* em memória. Um procedimento simples para ordenar um ficheiro de texto pode resumir-se ao seguinte:

- Ler todo o ficheiro de texto para uma tabela de *strings* (cada *string* corresponde a uma linha);
- Aplicar um qualquer algoritmo de ordenação à tabela de *strings* em memória;
- Escrever a tabela de *strings* já ordenada para um ficheiro de texto em disco.

4.9 Ordenar um ficheiro binário

Um esquema simples de ordenação de um ficheiro binário obedece aos mesmos três passos apresentados na secção anterior: ler ficheiro, ordenar em memória, gravar para ficheiro. Ao contrário de um ficheiro de texto, um ficheiro binário pode conter uma enorme complexidade de dados. Um ficheiro binário tanto pode guardar os tipos mais usuais do C/C++, como `int`, `float` ou `double`, como pode guardar fichas/estruturas mais ou menos complexas.

Esta secção apresenta um exemplo de um programa para ordenar um ficheiro de fichas contendo dados sobre pessoas (nome e altura). Pretende-se reordenar o ficheiro de modo a que ele passe a estar ordenado pela altura das pessoas. São definidas duas funções auxiliares: para iniciar de forma aleatória o ficheiro e para mostrar os dados armazenados em ficheiro.

```
// Este programa demonstra a forma como ordenar um ficheiro binário contendo fichas

#include <iostream>   // cout, cin
#include <iomanip>    // setw(), setprecision()
#include <cstdio>     // EOF
#include <cstring>    // strcmp(), strcpy(), strcat()
#include <search.h>   // qsort()
#include <cstdlib>    // random()
#include <fstream>    // open(), write(), read(), close()
#include <time.h>     // time()
using namespace std;
//---------------------------------------------------------------------------------
const int NumFichas=22;  // número de fichas que o ficheiro possui
const char nomeFicheiro[] = "Dados.dat"; // nome do ficheiro binário
const char nomes[6][50] = { "Artur", "Bruno", "Carlos",   // nomes para preencher
                            "Diogo", "Eduardo", "Felix"}; // ficheiro
//---------------------------------------------------------------------------------
typedef struct{   // definição de 1 ficha contendo nome e altura de uma pessoa
  char nome[21];  // nome da pessoa (até 20 caracteres + '\0')
  double altura;  // altura de cada pessoa
}TRec;
//---------------------------------------------------------------------------------
// criaFicheiro
//---------------------------------------------------------------------------------
// Esta função cria um ficheiro de fichas do tipo TRec com n fichas geradas
// aleatoriamente. Devolve 1 se o ficheiro foi inicializado sem problemas, ou 0 se
// ocorreram problemas durante a geração do ficheiro.
//
int criaFicheiro(int n) {
  fstream fic; // variável que se vai associar ao ficheiro a abrir
  char c;      // variável auxiliar
  TRec aux;    // variável auxiliar

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::in); // verifica se ficheiro já existe
  if (fic.is_open()) { // aparentemente o ficheiro já existe. será que o utilizador
                       // quer escrever por cima?
    cout << "O ficheiro: " << nomeFicheiro << " já existe. Quer reescrever?" << endl;
    cout << "[S-SIM, qualquer outra tecla = NÃO]" << endl;
    c = cin.get();
    cin.ignore(); // limpa lixo do buffer de leitura
    if((c != 'S') && (c!='s')) { // não quer reescrever o ficheiro existente
      fic.close();              // fecha o ficheiro que acabou de abrir
      return 0;        // termina função sem ter criado um novo ficheiro aleatório
    }
    // chegou aqui porque o utilizador carregou na tecla S de SIM.
    fic.close();  // fecha o ficheiro que abriu para leitura
  }
  // a função continua na pág. seguinte
```

```
  // Se chegou aqui é porque vamos mesmo criar um novo ficheiro com dados aleatórios
  // o ficheiro a abrir é binário, para escrita e apaga o conteúdo anterior.
  fic.open(nomeFicheiro, ios::out | ios::trunc | ios::binary);
  if (!fic.is_open()) {
    cout << "Houve problemas a criar o ficheiro!!" << endl << "Prima uma tecla"<<endl;
    cin.get();   // espera que o utilizador prima uma tecla
    return 0;
  }
  for (int i = 0; i < n; i++) {
    strcpy( aux.nome, nomes[rand() % 6] );  // Gera um nome aleatório
    strcat( aux.nome, " ");                 // Acrescenta um espaço
    strcat( aux.nome, nomes[rand() % 6] );  // Acrescenta outro nome próprio
    aux.altura = (rand() % 80)/100.0+1.30;  // gera altura de 1.30 m a 2.10 m
    fic.write((char *)&aux, sizeof(aux));   // grava ficha no ficheiro
  }
  fic.close(); // fecha o ficheiro
  return 1;   // chegou aqui, o que quer dizer que tudo correu bem
} // fim de criaFicheiro
//-----------------------------------------------------------------------------
void imprimeFicheiro(void) {
  fstream fic;  // variável que vai ser associada ao ficheiro a ler
  int n;        // variável auxiliar para contar o número de fichas do ficheiro
  TRec aux;

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::in | ios::binary);
  if (!fic.is_open()) {
    cout << "houve problemas a abrir o ficheiro "<< nomeFicheiro << " para leitura\n";
    cout << "Prima qualquer tecla";
    cin.get();
    return;
  }
  // chegou aqui porque o ficheiro abriu corretamente
  n = 0;  // vai ler ficha na posição 0 do ficheiro
  while (fic.peek() !=.EOF) {  // le ficheiro enquanto houver fichas para ler
    fic.read( (char *)&aux, sizeof(aux) );
    cout << "(" << setw(2) << n << ") = " << setw(20) << aux.nome;
    cout << ", " << setprecision(2) << aux.altura << endl;
    n++;
  }
  fic.close();  // termina leitura do ficheiro
  cout << "Prima qualquer tecla" << endl;
  cin.get();
}
//-----------------------------------------------------------------------------

// o programa continua na pág. seguinte
```

```
//--------------------------------------------------------------------------------
// Função para comparar as alturas de 2 fichas (para o qsort)
//
int comparaAltura(const void *ficha1, const void *ficha2) {
  return ((TRec *)ficha1)->altura < ((TRec *)ficha2)->altura ? -1 : 1;
  // devolve -1 se altura 1 for menor que a altura 2, senão devolve 1
}
//--------------------------------------------------------------------------------
void ordenaFicheiro(void) {
  fstream fic;
  TRec *tab;

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::binary | ios::in );
  if (!fic.is_open()) {
    cout<<"Houve problemas a abrir o ficheiro "<< nomeFicheiro<<" para leitura"<<endl;
    cout << "Prima qualquer tecla para continuar" << endl;
    cin.get();
    return;
  }

  tab = new TRec[NumFichas];  // Aloca memória para a tabela
  fic.read((char *)tab, NumFichas * sizeof(TRec)); // Lê o ficheiro todo para memória
  fic.close();  // termina a leitura do ficheiro

  qsort((char *)tab, NumFichas, sizeof(TRec), comparaAltura); // ordena tabela

  fic.open(nomeFicheiro, ios::binary | ios::out | ios:: trunc);
  fic.write((char *)tab, NumFichas * sizeof(TRec)); // escreve tabela toda no ficheiro
  fic.close(); // termina a escrita no ficheiro

  delete [] tab;  // liberta toda a memória pedida para a tabela

  cout << "Ficheiro ordenado com sucesso..." << endl << endl;
}
//--------------------------------------------------------------------------------
int main() {
  char c; // variável auxiliar para opção do utilizador

  srand((unsigned)time(NULL));
  do {
    cout << "\n\n MENU PRINCIPAL:\n";
    cout << "1 - Gerar um novo ficheiro com dados aleatórios\n";
    cout << "2 - Ver conteúdo do ficheiro\n";
    cout << "3 - Ordenar ficheiro\n";
    cout << "4 - Sair do programa\n";
    cin.get(c);     // le opção
    cin.ignore();   // ignora lixo que fica no buffer de leitura
    switch (c) {
      case '1':
        criaFicheiro(NumFichas);
      break;
      case '2':
        imprimeFicheiro();
      break;
      case '3':
        ordenaFicheiro();
      break;
    }
  } while(c != '4'); // não termina programa enquanto o utilizador não carregar em '4'
}
//--------------------------------------------------------------------------------
// fim de programa
//--------------------------------------------------------------------------------
```

Notas:
- A maior parte do código é apenas auxiliar. As únicas funções que fazem mesmo a ordenação do ficheiro são a `comparaAltura()` e a `ordenaFicheiro()`;
- Procure entender o código apresentado: a maior parte dele representa tarefas elementares de programação;
- Foi utilizada a função `qsort()` para ordenar os dados. Recomenda-se que tente alterar o código para utilizar um dos algoritmos de ordenação apresentados nas aulas teóricas.

4.10 Análise da complexidade de algoritmos – notação O-grande

Em Ciências da Computação, faz-se a análise da complexidade dos algoritmos para se ter uma ideia da ordem de grandeza do tempo de execução dos algoritmos para um determinado nº de dados a processar. Os algoritmos são então classificados usando a notação O-grande (*Big-O notation*). Ao analisar a complexidade de um algoritmo em termos de tempo de processamento, vai ter-se em conta o nº de vezes que o conjunto de instruções mais frequentes é executado, em função do nº de dados N a processar. Assim, um algoritmo em que só existe um ciclo (for, while ou do…while) com N iterações tem complexidade linear, sendo classificado como O(N). O cálculo do valor máximo de uma tabela é um exemplo de um algoritmo com complexidade linear. Se houver dois ciclos imbricados a complexidade é quadrática, sendo a classificação $O(N^2)$. Um exemplo de um algoritmo com complexidade quadrática é o algoritmo de ordenação por seleção. Uma pesquisa binária tem complexidade O(log N). Na análise da complexidade de algoritmos não se tem em conta constantes simples (não dependentes do nº de dados a processar). Assim, um algoritmo de determinação simultânea do maior e do menor valor de uma tabela também tem complexidade linear, O(N).

4.11 Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à ordenação de dados e complexidade de algoritmos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas.

Problema 4.1 – Ordenação por bolha (*Bubble-sort*) de inteiros + `random()` – Fácil

Faça um programa que permita ordenar por ordem crescente uma tabela com 20 inteiros (idades de eleitores) usando o algoritmo de ordenação por bolha (*bubble-sort*). Inicie a tabela no início do programa de forma aleatória, com números no intervalo de 18 a 90. Implemente também uma função que imprima os números da tabela. Utilize esta função para imprimir a tabela antes e depois de ter sido ordenada.

Problema 4.2 – Ordenação por seleção (*Selection-sort*) de reais – Fácil

Implemente uma função que permita ordenar por ordem decrescente uma tabela de reais (tipo `float`) utilizando obrigatoriamente o algoritmo de ordenação por seleção (*selection-sort*). Teste a função num programa feito por si e com os valores contidos na tabela a serem inicializados como constantes (esses valores serão temperaturas em 6 cidades europeias). A função deverá receber como parâmetro o número de elementos que a tabela possui.

Problema 4.3 – Ordenação de uma tabela de estruturas – Fácil

Implemente uma função que permita ordenar por ordem alfabética (do nome) uma tabela com 10 fichas (defina uma nova estrutura de dados usando `struct`). Cada ficha contém o nome, a idade e o número de estudante de um aluno. Teste a função num programa feito por si e com valores na tabela inicializados como constantes (use valores verosímeis).

Problema 4.4 – `qsort` de *ints* – Fácil /Médio

Resolva o problema anterior utilizando a função `qsort()`, disponível no ANSI C. Ordene por ordem crescente da idade.

Problema 4.5 – Ordenação de um *array* de *strings* – Médio

Faça uma função que permita ordenar uma tabela de nomes de países (*strings*). A função deverá receber como parâmetro o ponteiro para a tabela e o número de países na tabela. Teste essa função com um programa feito por si. Inicialize a tabela explicitamente com nomes de teste não ordenados e crie uma nova função para imprimir a tabela. Utilize esta função para mostrar a tabela antes e depois de ser ordenada.

Problema 4.6 – Ordenação de um ficheiro de texto – Médio

Faça um programa que ordene por ordem decrescente de idades as pessoas listadas num ficheiro de texto. O programa deve ler os dados do ficheiro para memória, ordená-los e voltar a escrevê-los no disco. O formato de cada linha do ficheiro é: *data_de_nascimento nome*. Exemplo: `19880501 José Silva`.

Para verificar o resultado do programa utilize uma tabela de *strings* com pelo menos 100 caracteres. Assuma que o ficheiro não tem mais do que 100 linhas de texto. Crie um ficheiro de texto (editado à mão num qualquer processador de texto) com nome `dados_desord.txt` e grave o resultado da ordenação em `dados_ord.txt`.

Problema 4.7 – Ordenação de um ficheiro de texto – `qsort()` – Médio

Resolva o problema anterior usando a função `qsort()`, disponível no ANSI C. Se não resolveu o problema anterior, faça a ordenação de uma tabela de *strings* iniciada na declaração.

Problema 4.8 – Ordenação de um ficheiro de texto – `qsort()` – Difícil

Resolva o problema anterior usando a função `qsort()` mas peça ao utilizador mais um parâmetro: a partir de que coluna do texto é que quer começar a ordenar o texto. Veja o resultado no exemplo seguinte:

conteúdo do ficheiro `nomes.txt`:	ordenado a partir da coluna 1	ordenado a partir da coluna 10:

`19880521 Carlos`
`19781021 Artur`
`20000501 Bruno`
	`19781021 Artur`
	`19880521 Carlos`
	`20000501 Bruno`
		`19781021 Artur`
		`20000501 Bruno`
		`19880521 Carlos`

Problema 4.9 – Ordenação de um *array* de *structs* - Difícil

Faça um programa que ordene uma tabela com 12 fichas criadas por si explicitamente. Cada ficha deve ter 3 campos: uma *string* de 60 caracteres (nome), um número inteiro sem sinal (idade) e um `float` (altura em metros). Deverá iniciar a tabela usando uma função que coloque dados desordenados nas 12 fichas e em todos os campos. Deve depois mostrar um menu de opções para o utilizador com o seguinte aspeto:

```
MENU:
1 – Iniciar com os dados de arranque (desordenados) e mostrar no monitor
2 – Ordenar por nome e mostrar no monitor
3 – Ordenar pela idade e mostrar no monitor
4 – Ordenar pela altura e mostrar no monitor
5 – Sair do programa (sem confirmação)
```

Problema 4.10 – `qsort()` de uma tabela de `structs` - Difícil

Use a função `qsort()` para resolver o problema anterior.

Problema 4.11 – Q*uick-sort* versus *bubble-sort* - Difícil

Faça um programa que permita comparar o desempenho em termos de tempo de execução dos algoritmos *quick-sort* (usando a função `qsort()`, disponível no ANSI C) e *bubble-sort*. Esse programa deve ter 2 opções: testar *quick-sort* e testar *bubble-sort*. Para fazer cada um dos testes, deve fazer o seguinte: iniciar uma tabela com 100000 elementos do tipo `double` gerados aleatoriamente; depois, use a função `time()` da biblioteca `<time.h>` para guardar o tempo em segundos, antes de chamar o algoritmo de ordenação e depois deste terminar; calcular e mostrar no monitor a diferença entre os 2 instantes, em segundos.

Para efeitos de verificação, teste primeiro os dois algoritmos com uma tabela com apenas 10 elementos e imprima-os no monitor para confirmar que o programa está a funcionar bem. Calcule os tempos para a tabela com 100000 elementos do tipo `double` e para uma tabela com 200000 elementos do tipo `double`. Conclua como evolui o tempo de execução de cada um dos algoritmos quando aumenta a tabela para o dobro.

Problema 4.12 – *Selection sort* versus `qsort()` versus *bubble-sort* - Difícil

Altere o programa anterior de forma a permitir comparar o desempenho em termos de tempo de execução dos algoritmos *selection sort*, *quick-sort* (usando a função `qsort()` disponível no ANSI C) e *bubble-sort*. Esse programa deve ter 3 opções: testar *selection sort,* testar *quick-sort* e testar *bubble-sort*.

Para efeitos de verificação, teste primeiro os 3 algoritmos com uma tabela com apenas 10 elementos e imprima-os no monitor para confirmar que o programa está a funcionar bem. Calcule os tempos para a tabela com 100000 elementos do tipo `double` e para uma tabela com 200000 elementos do tipo `double`. Conclua como evolui o tempo de execução de cada um dos algoritmos quando aumenta o tamanho da tabela para o dobro.

Problema 4.13 – Ordenação de fichas com o Quick-Sort – Difícil (adaptado do teste de frequência de 16/03/2016)

Considere uma tabela de fichas (`struct`) com informação sobre ratos de computador à venda numa determinada loja. Cada ficha tem os seguintes campos:

- `sem_fios` (booleano; `true` se rato sem fios)
- `marca` (*string* à C, até 40 caracteres úteis)
- `ref` (referência; *string* à C, até 20 caracteres úteis)
- `em_stock` (indica quantos ratos com esta referência existem para venda na loja).

Considere que existe uma tabela de fichas de tipos de rato definida e inicializada na função `main()`.

a) Declare a estrutura e chame-lhe rato.

b) Escreva uma função para indicar quantos ratos com fios existem para venda. A função tem 2 parâmetros: uma tabela de fichas e um inteiro com o nº de fichas existentes na tabela. Vai retornar o nº de ratos com fios existentes.

c) Implemente uma função que permita ordenar uma tabela de fichas de ratos pela sua marca, usando o algoritmo Quick-Sort (`qsort`).

Nenhuma destas funções escreve nada no ecrã. Como trabalho extra, implemente uma função `main()` mínima para ir testando as funções implementadas.

Problema 4.14 – Inserção ordenada por *Bubble-sort* – Difícil (adaptado do teste de frequência de 25/03/2015)

Considere uma `struct` para gerir uma tabela ordenada de números reais (amostras de temperaturas). Os campos são a tabela propriamente dita (`tab`), a sua dimensão (`max`) e um inteiro (`n`) com o nº de elementos na tabela. A tabela deve estar sempre ordenada por ordem decrescente, com os dados agrupados no início da tabela.

a) Declare a `struct` e defina uma função para inicializar uma tabela vazia com dimensão dada pelo 2.º parâmetro dessa função. A `struct` da tabela a inicializar é o 1.º parâmetro da função.

b) Implemente uma função `insereTab()` que acrescenta um novo elemento, ordenadamente, a uma tabela, devolvendo `false` se a inserção falhar por a tabela estar cheia e `true` caso contrário. Os dois parâmetros da função são: a `struct` com a tabela e o novo elemento a inserir. Utilize o princípio do algoritmo de ordenação da bolha, assumindo que a tabela está ordenada antes da inserção do novo elemento.

Problema 4.15 – Ordenação por inserção – Difícil (adaptado do Exame de Recurso de 03/07/2017)

Considere uma tabela de fichas (`struct`) com informação sobre ratos de computador à venda numa determinada loja. Cada ficha tem os seguintes campos:

- `pvp` (preço em euros, real de precisão dupla)
- `marca` (*string* à C++)
- `ref` (referência, *string* à C, até 20 caracteres úteis)
- `stock` (indica quantos ratos destes existem para venda na loja).

a) Declare a estrutura e chame-lhe rato.

b) Implemente uma função que permita ordenar por ordem decrescente uma tabela de fichas de ratos pelo seu preço, usando o algoritmo de ordenação por inserção. Esta função não escreve nada no ecrã.

Problema 4.16 – Inserção ordenada – Difícil (adaptado do teste de frequência de 13/03/2018)

Considere uma tabela de fichas (tabela de `struct`) com informação sobre monitores de computador à venda numa dada loja. Cada ficha tem os seguintes campos:

. `marca` (*string* à C, até 50 caracteres úteis)
. `ref` (referência; *string* à C, até 30 caracteres úteis)
. `fullHD` (booleano, `true` se formato `fullHD`)
. `em_stock` (indica quantos monitores destes existem para venda na loja, máx. até cerca de 700).

a) Declare a estrutura e chame-lhe monitor.
b) Implemente uma função (devolve `void`) que permita inserir ordenadamente um novo tipo de monitor (ainda não existente) numa tabela de fichas de monitores, ordenada pela marca, usando o princípio do algoritmo da ordenação por inserção. Parta do princípio de que nunca se atinge a dimensão máxima da tabela. A função não escreve nada no ecrã.
c) Declare e inicialize na função `main()` uma tabela (ordenada pela marca) de tipos de monitores existentes e uma variável inteira com o nº de elementos existentes nessa tabela.
d) Chame a função da alínea b) para inserir ordenadamente uma nova ficha (novo tipo de monitor), declarada e inicializada na função `main()`.
e) Imprima a lista de marcas existentes.

Problema 4.17 – Complexidade de algoritmos – Fácil

Considere uma tabela com `N` números inteiros ordenada por ordem crescente. Indique exemplos de operações sobre essa tabela que tenham complexidade: a) constante; b) linear; c) logarítmica; d) quadrática.

Problema 4.18 – Complexidade de algoritmos – Fácil

Considere uma função `g()` de complexidade O(N) e outra função `f()` de complexidade $O(N^2)$ que quando manipulam `N` números reais demoram cerca de 10 ms cada uma a serem executadas. Se essas funções passarem a manipular 10×N números reais qual o tempo expectável de processamento?

Problema 4.19 – Complexidade de algoritmos – Fácil

Considere uma tabela com `N` nomes de pessoas (strings C++) ordenada por ordem alfabética. Usando a notação O-grande, indique a complexidade dos seguintes algoritmos:

a) ordenação da tabela (por ordem alfabética) usando um algoritmo da bolha standard;
b) ordenação da tabela (por ordem alfabética inversa) usando um algoritmo da bolha standard;
c) pesquisa sequencial de um nome;
d) pesquisa binária de um nome.

ESTRUTURAS DE DADOS E ALGORITMOS

Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 5
ALGORITMOS BÁSICOS DE PESQUISA

OBJETIVOS

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o método de pesquisa sequencial de dados;
- Compreender o algoritmo de pesquisa binária;
- Conhecer a diferença, em termos de desempenho, entre uma pesquisa binária e uma sequencial;
- Saber implementar em C++ o algoritmo de pesquisa sequencial numa tabela e num ficheiro;
- Saber implementar em C++ o algoritmo de pesquisa binária numa tabela e num ficheiro.

PESQUISA SEQUENCIAL

Uma das operações mais frequentes de manipulação de dados é a pesquisa. Atualmente, e graças à massificação e maior poder de cálculo dos computadores pessoais, é possível fazer pesquisas rápidas de informação, pesquisas essas que outrora eram complexas ou demoradas de fazer.

A pesquisa sequencial (ou linear) é muitas vezes o único método de pesquisa disponível quando os dados a procurar não se encontram ordenados. Pesquisar sequencialmente um conjunto de dados corresponde a comparar um termo de pesquisa com cada um dos elementos do conjunto de dados, até se encontrar uma correspondência entre o termo de pesquisa e um dos dados do conjunto. A principal desvantagem da pesquisa sequencial é que esta obriga a percorrer todo o conjunto de dados para se encontrar todas as ocorrências do termo de pesquisa, por exemplo, todas as pessoas cujo primeiro nome é Carlos. Estes dados podem estar armazenados em tabelas ou em ficheiros. Numa pesquisa sequencial é necessário percorrer todo o conjunto de dados para verificar que o elemento a pesquisar não existe.

Vejamos um caso concreto: imaginemos que queremos procurar uma ficha numa tabela desordenada de fichas (*array* de *structs*), que tenha o campo nome igual a um determinado nome introduzido pelo utilizador:

```
#include <cstring>  // strcmpi()
// strcmpi(S1,S2); -> compara S1 com S2 ignorando se as letras
//                    estão em maiúsculas ou minúsculas.

const int MAX_FICHAS=10000;

typedef struct {   // definição da estrutura de cada ficha na tabela
  long int numBI;  // pode ter outros campos, não relevante para o exemplo
  char nome[151];
} TBI;
// continua na página seguinte
```

```
TBI Tab[MAX_FICHAS]; // tabela pode ter até MAX_FICHAS fichas do tipo TBI
                     // (não deve ser var global)
//-------------------------------------------------------------------------
// Função que devolve a posição, na tabela, da ficha que contém o nome igual
// ao nome na string de pesquisa 'nomePesquisar'. Procura desde a posição
// 'inicio' da tabela até à posição 'fim'.
// Devolve -1 se não encontrar nenhuma ficha.
int procuraNome(TBI tab[], int inicio, int fim, char *nomePesquisar) {
  for (int i=inicio; i<= fim; i++) {      // Um ciclo desde inicio até fim
    if ( strcmpi(nomePesquisar, tab[i].nome) == 0) // se encontrou a ficha,
      return i;                                    // então devolve índice
  }
  return -1;  // não encontrou nenhuma ficha com o mesmo nome, devolve -1
} // fim da função
//-------------------------------------------------------------------------
```

Como foi referido anteriormente a desvantagem da pesquisa sequencial reside no facto de ser necessário percorrer *toda* a tabela para nos certificarmos de que o valor que andamos à procura não existe.

A pesquisa do exemplo anterior podia ser feita em alternativa através do número do BI ou por qualquer outro dado que pertença à estrutura de dados. O exemplo seguinte ilustra uma pesquisa sequencial pelo número do BI:

```
//-------------------------------------------------------------------------
// Função que devolve a posição (índice) da ficha com número de
// BI igual a 'numBI' ou -1 caso não exista uma ficha com esse número de BI
int ProcuraBI(TBI tab[], int inicio, int fim, long int numBI) {
      for (int i=inicio; i<= fim; i++) {  // Um ciclo desde inicio até fim
            if ( numBI == tab[i].numBI )  // se encontrou a ficha, então
                  return i;               // devolve índice
      }
      return -1; //não encontrou nenhuma ficha com o número numBI: devolve -1
} // fim da função
//-------------------------------------------------------------------------
```

Como estimar o tempo de pesquisa sequencial a uma tabela de dados?

Supondo que se quer pesquisar um ficheiro (por exemplo) com **N** fichas e que demoramos um tempo de **t** segundos a "ler e verificar" cada ficha, então será de esperar que o tempo para achar um elemento no ficheiro (supondo que ele existe) seja, em média, igual a **t×N/2** segundos, supondo que a ficha a procurar pode estar numa qualquer posição do ficheiro. Diz-se então que a pesquisa sequencial ou linear tem um peso computacional proporcional ao número de dados a pesquisar, logo tem complexidade de ordem **N**, ou **O(N)**.

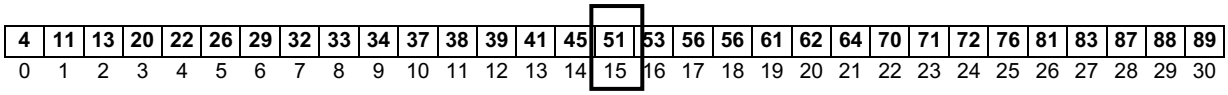
PESQUISA BINÁRIA

Quando o conjunto dos dados a pesquisar está ordenado segundo um determinado critério então podemos utilizar um método de pesquisa mais eficiente do que a pesquisa sequencial: a pesquisa binária, também conhecida por pesquisa dicotómica. Esta forma de pesquisa é bastante rápida e consiste no seguinte:

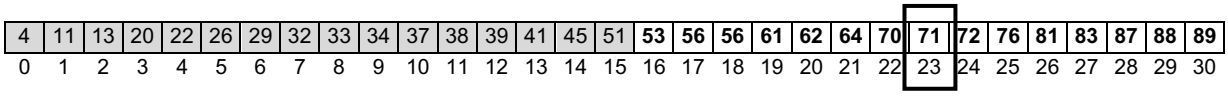
1. Toma-se o conjunto de dados ordenados a pesquisar;

2. Compara-se o elemento no meio do conjunto com o termo de pesquisa. Se forem iguais terminou a pesquisa;
3. Se o termo de pesquisa for anterior ao elemento do meio então reduz-se o conjunto a pesquisar apenas aos elementos anteriores ao elemento do meio. Se o termo de pesquisa for posterior ao elemento do meio então reduz-se o conjunto de dados a pesquisar à metade posterior ao elemento do meio.
4. Enquanto o conjunto resultante não for nulo, volta-se ao passo 2, agora com metade dos elementos para procurar.

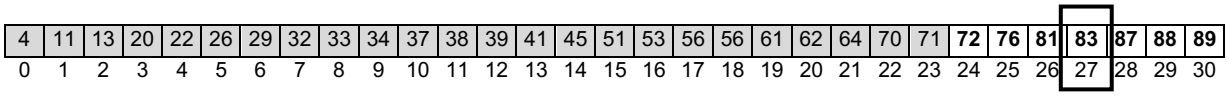
Vejamos um exemplo do funcionamento da pesquisa binária: Consideremos uma tabela de inteiros **tab** com 31 valores ordenados. Queremos saber em que posição da tabela está o número 81.



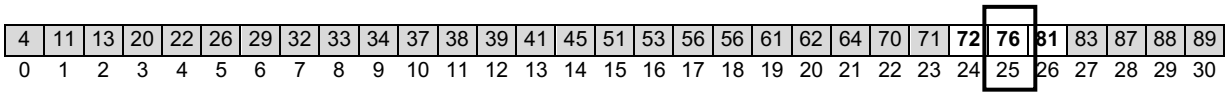
Começa-se a pesquisa pelo elemento do meio da tabela. Define-se o elemento do meio através da seguinte expressão: **(indiceFinal + indiceInicial)/2**. Neste caso concreto: **meio = (0 + 30)/2 = 15**. Como **tab[15] = 51 < 81**, então o elemento a pesquisar está entre a posição 16 e a posição 30 (ver abaixo):



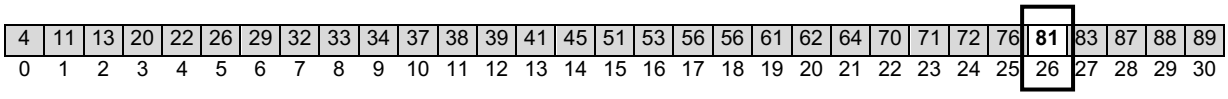
A pesquisa continua agora apenas na sub-tabela entre os índices 16 e 30. O novo elemento do meio é agora: **meio = (16+30)/2 = 23**. Como **tab[23] = 71 < 81**, o elemento a procurar está entre a posição 24 e 30.



A pesquisa continua agora na sub-tabela entre os índices 24 e 30. Mais uma vez é calculado o elemento do meio: **meio = (24+30)/2 = 27**. Como **tab[27] = 83 > 81** então o elemento a procurar está abaixo do elemento do meio, logo da posição 24 à posição 26.



A pesquisa continua agora entre os elementos com índices 24 a 26. O novo elemento do meio passa a ser: **meio = (24+26)/2 = 25**. Sendo t**ab[25] = 76 < 81**, o elemento a procurar está acima do meio, logo apenas poderá estar no elemento com índice 26.



Finalmente analisa-se o único elemento que falta: **meio= (26+26)/2 = 26**. Como **tab[26] = 81** então encontrou-se na posição 26 o elemento que se procurava. Foram apenas necessárias 5 iterações até se encontrar este valor. Se tivesse sido utilizada a pesquisa sequencial neste mesmo exemplo, teriam sido necessárias 27 iterações.

Resumindo, a pesquisa binária vai dividindo ao meio o conjunto a procurar até se chegar a um conjunto com apenas 1 elemento.

Como estimar o tempo de pesquisa binária de uma base de dados?

Supondo que se quer pesquisar um ficheiro (por exemplo) com **N** fichas e que demoramos um tempo de **t** segundos a ler e verificar cada ficha, então podemos esperar que o tempo necessário para encontrar um elemento no ficheiro (caso exista ou não) seja, em média, $\mathbf{t \times log_2(N+1)}$ segundos (supondo que o ficheiro se encontra ordenado e que a ficha a procurar se pode encontrar numa posição qualquer do ficheiro). Diz-se então que a pesquisa binária tem um peso computacional proporcional ao logaritmo dos dados a pesquisar, logo uma complexidade de ordem **log(N)**, ou simplesmente **O(log(N))**.

Na tabela abaixo apresenta-se a correspondência entre o número de elementos de uma tabela ordenada e o número de leituras necessárias até se encontrar o elemento utilizando pesquisa binária:

Número de elementos na tabela	Número de leituras (pesquisa binária)
16...31	5
1.000	10
100.000	17
1.000.000	20
100.000.000	27

Isto quer dizer que se tivermos um ficheiro com 100 milhões de dados ordenados, apenas temos que ler 27 desses dados até chegar à posição do dado procurado. Numa pesquisa sequencial teríamos que ler em média 50 milhões de fichas nessa mesma pesquisa. Podemos concluir que o facto de termos uma tabela ordenada permite reduzir em muito o tempo de pesquisa caso seja utilizada a pesquisa binária.

Implementação da pesquisa binária em C++

Podemos implementar a pesquisa binária utilizando dois tipos de abordagem: uma abordagem iterativa ou uma abordagem recursiva. A título de exemplo, consideremos uma tabela de **n** inteiros ordenados por ordem crescente. Consideremos em primeiro lugar a solução iterativa. O algoritmo de pesquisa binária pode ser resumido da seguinte forma:

Algoritmo de pesquisa binária:

Dados: **`inicio=0`**, o índice do primeiro elemento da tabela, **`fim = n-1`** o índice do último elemento da tabela e **`pesq`** o elemento de pesquisa;

Passo 1: Determina o elemento no meio da tabela **`meio = (inicio+fim)/2`**, arredondado para baixo;

Passo 2: Determina o valor da tabela no índice **`meio: valor=tab[meio]`**;

Passo 3: Se o valor a pesquisar **`pesq`** for igual a **`valor`** então foi encontrado o valor de pesquisa. Termina o algoritmo devolvendo o índice do **`valor`** (na posição cujo índice é **`meio`**).

Passo 4: Se **`pesq`** for maior do que **`valor`** então procura na sub-tabela da direita, ou seja o novo **`inicio=meio+1`**. Caso contrário procura na sub-tabela da esquerda, ou seja o novo **`fim=meio-1`**;

Passo 5: Se **`inicio <= fim`** então repete de novo a partir do passo 1;

Passo 6: Como **`inicio > fim`** ==> Insucesso! Não foi encontrado qualquer elemento igual na tabela.

Com base neste algoritmo, podemos chegar à seguinte implementação da pesquisa binária:

```
//Função que devolve o índice da tabela de inteiros 'tab' que contém o número
//'valor'. A tabela tem 'comp' elementos.
// No caso de não existir, a função devolve -1
int pesquisaBinaria(int tab[ ], int comp, int valor) {
  int inicio = 0;       // inicio tem o índice do primeiro elemento da tabela
  int fim = comp - 1;   // e fim tem o índice do último elemento
  while (inicio <= fim){  // Enquanto a tabela a pesquisar tiver elementos
    int meio = (inicio+fim)/2;  // determina o índice do elemento do meio
    if (valor == tab[meio]) return meio; // O valor pretendido foi encontrado
    else
      if (valor > tab[meio])  // O valor é maior?
        inicio = meio + 1;    // Então procura na sub-tabela da direita
      else                    // Senão (o valor é menor e)
        fim = meio - 1;       // procura na sub-tabela da esquerda.
  }
  return -1;  // O valor pretendido não foi encontrado
}  // Fim da função
```

A solução recursiva é em tudo idêntica à resolução iterativa. A condição de paragem é a tabela ser vazia. Neste caso, não é possível encontrar o valor e a função devolve -1. No caso geral, compara-se o valor a procurar com o valor no meio da tabela. Se o valor for maior então devolve-se (de uma forma recursiva) o resultado da pesquisa na sub-tabela da direita. Se o valor for menor então devolve-se (também de uma forma recursiva) o resultado da pesquisa na sub-tabela da esquerda. No caso de o valor ser igual ao elemento do meio então devolve-se o índice do elemento do meio. A função de pesquisa passa a ser:

```
// Função de pesquisa recursiva que devolve o índice da tabela 'tab'
// que contém o número 'valor'.
// A tabela é definida pelo índice inicial 'inicio' e pelo índice
// final 'fim'. No caso de não existir o valor a função devolve -1.
int pesquisaBinariaRec(int tab[ ], int inicio, int fim, int valor) {
  if(inicio>fim) return -1;   //A tabela está vazia - valor não encontrado
  int meio = (inicio+fim)/2;    // Calcula o índice do elemento do meio
  if(valor>tab[meio]) //O valor é maior então procura na sub-tabela à direita
    return(PesquisaBinariaRec(tab, meio+1, fim, valor));
  if(valor<tab[meio]) //O valor é menor então procura na sub-tabela à esq.
    return(PesquisaBinariaRec(tab, inicio, meio-1, valor));
  return meio;    // Se não maior nem menor então só pode ser igual…
} // Fim da função
```

De notar que o protótipo da função passa a ser diferente uma vez que necessitamos de indicar nas sucessivas chamadas à função os limites de cada uma das sub-tabelas. Nesse sentido, deixamos de passar o comprimento da tabela para passar a indicar o índice inicial e o índice final.

PESQUISAS NUM FICHEIRO DE FICHAS

Escrever um programa para fazer pesquisa sequencial num ficheiro é relativamente simples. O procedimento é relativamente básico: abrir o ficheiro; ler e comparar cada ficha com o termo de pesquisa, até se encontrar uma ficha com um campo de pesquisa igual ao termo de pesquisa, ou até se atingir o fim

do ficheiro. Neste último caso, deve ser dada uma indicação especial para assinalar que a ficha pretendida não foi encontrada. Finalmente dever-se-á sempre fechar o ficheiro. **Deixa-se ao cuidado do aluno implementar um código de exemplo.**

Pesquisa binária num ficheiro binário

Para se fazer uma pesquisa binária (dicotómica) num ficheiro binário contendo fichas, o procedimento é análogo ao utilizado na pesquisa binária numa tabela. Há no entanto algumas diferenças resultantes das particularidades dos ficheiros: é preciso analisar quantas fichas estão guardadas em ficheiro antes de se iniciar a pesquisa e é necessário ler uma ficha do ficheiro antes de ela poder ser analisada.

A seguir apresenta-se um exemplo de uma função para fazer uma pesquisa binária num ficheiro de fichas com informação sobre o BI. Estas fichas estão ordenadas pelo campo altura.

```
typedef struct {      // definição da estrutura de cada ficha no ficheiro
      long int numBI; // pode ter outros campos, não relevante para o exemplo
      char nome[151];
      double altura;
} TBI;
//-----------------------------------------------------------------------
// Função que devolve a posição da ficha com altura igual ao
// parâmetro 'altura' ou -1 no caso de não existir nenhuma ficha
// com a altura procurada. Esta função implementa a pesquisa
// binária no ficheiro dado por 'nomeFicheiro'. Para além disso,
// a função devolve no parâmetro 'ficha' a ficha retirada do ficheiro.
int pesquisaBinariaFicheiro(char *nomeFicheiro, float altura, TBI *ficha) {
  int inicio, meio, fim, n; // variáveis auxiliares
  // abre ficheiro apenas se ele já existir
  ifstream fic(nomeFicheiro, ios::binary);
  if (!fic.is_open() ) return -1; // Não foi possível abrir o ficheiro, sai!
  // Se chegou aqui então foi porque conseguiu abrir o ficheiro para leitura
  fic.seekg(0, ios::end);   // Aponta para o elemento 0 a partir da última
                            // posição, ou seja aponta para o último elemento
  n = fic.tellg()/sizeof(TBI);
  // O método tellg() devolve a posição em bytes do próximo elemento que se
  // vai ler do ficheiro. Como o ponteiro de leitura está a apontar para o
  // fim do ficheiro, este devolve o número de bytes que o ficheiro possui.
  // n passa a ter o número de fichas existentes no ficheiro
  inicio = 0;  // define o inicio do conjunto de pesquisa
  fim = n-1;   // define o fim do conjunto de pesquisa
  while ( inicio <= fim ) {
    meio = (inicio+fim)/2;  // meio <- posição do elemento no meio da tabela
    fic.seekg(meio*sizeof(TBI)); // Aponta para a posição do ficheiro onde
                                 // está a ficha no meio do conjunto de pesquisa


// continua na próxima página…
```

```
    // lê a ficha para memória. Uma ficha ocupa exactamente sizof(TBI) bytes
    fic.read( (unsigned char *)ficha, sizeof(TBI));
    if (altura == ficha->altura) {  // encontrada ficha com altura procurada
      fic.close(); // Fecha o ficheiro, pois terminaram as leituras.
      return meio; // A estrutura ficha tem os dados da ficha e a função deve
    }              // devolver a posição do ficheiro onde estava a ficha.
    if (altura > ficha->altura) // Se altura > altura no meio da tabela
      inicio = meio + 1;        // Então procura na 2ª metade do sub-ficheiro
    else                        // Senão (altura é menor e)
      fim = meio - 1;           // procura na 1ª metade do sub-ficheiro
  } // fim do while
  fic.close();    // Fecha ficheiro pois terminaram as leituras
  return -1;      // Não encontrou qualquer ficha com a altura pretendida
}  // Fim da função
```

PESQUISA EM FICHEIRO DE TEXTO

Fazer uma pesquisa sequencial num ficheiro de texto é relativamente fácil. O processo consiste em ler o ficheiro linha a linha, comparando cada linha com a *string* a procurar. **Deixa-se ao cuidado do aluno implementar um exemplo.**

Uma pesquisa binária num ficheiro de texto torna-se numa tarefa mais complexa, pois uma linha de texto, ao contrário de uma ficha (`struct`) não ocupa um número fixo de bytes. Ora, apontar para o início de uma linha no meio de um ficheiro obriga normalmente a uma procura sequencial do sítio onde essa linha começa e acaba. Apesar deste detalhe, a pesquisa binária pode também ser aplicável a ficheiros de texto, embora a forma de o conseguir não seja o objetivo desta unidade curricular.

EXERCÍCIOS SUGERIDOS

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à pesquisa de dados. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas.

Problema 5.1 – Pesquisa sequencial de números – Fácil

Escreva uma função que permita procurar, numa tabela contendo **N** números inteiros, a primeira ocorrência de um determinado número. A função deverá receber como parâmetros um ponteiro para a tabela, o número de elementos da tabela e o número a procurar. Deverá devolver o índice do elemento da tabela onde se encontra o valor ou –1 caso este não exista na tabela.

Teste a função com um programa feito por si e com os valores da tabela a serem inicializados aleatoriamente (considere N=100 e que cada elemento pode variar entre –10 e 42).

Problema 5.2 – Pesquisa sequencial de uma substring – Fácil

Escreva uma função que permita procurar, numa tabela com **N** *strings* (com até 100 caracteres), a primeira ocorrência de uma determinada *substring*. A função deverá receber como parâmetros um ponteiro para a tabela de *strings*, o número de elementos que contém a tabela e um ponteiro para a *substring* que se pretende procurar. A função deverá devolver o índice do 1º elemento da tabela onde aparece a *substring* ou então –1, se não foi encontrada a *substring* na tabela. Abaixo está um possível protótipo da função:

```
int procuraSubString(char tab[][101], int numElemsTab, char *subString);
```

Sugestão: Utilize a função `strstr()` para o ajudar na resolução deste problema.
Teste a função num programa feito por si e com 6 valores na tabela inicializados como constantes.

Problema 5.3 – Pesquisa binária de número – Médio

Implemente uma função que devolva quantos valores de uma tabela ordenada de *doubles* (temperaturas máximas anuais) estão acima de um dado valor **`min`**. A função deverá usar pesquisa binária para descobrir a posição na tabela do valor a seguir ao procurado. Os parâmetros de entrada da função deverão ser: um ponteiro para a tabela, o número de elementos na tabela e o valor **`min`**. A função deverá devolver o número de valores na tabela com valor acima do valor **`min`**.

Nota: ao contrário das funções normais de pesquisa binária, esta função não devolve –1 quando o valor não existe. O que se quer devolver é o número de valores na tabela acima do valor **`min`** e portanto, mesmo que o valor não exista na tabela a função pode devolver valores de 0 até n. Também pode suceder existirem vários elementos na tabela com o valor procurado.

Teste a função com um programa feito por si e com os n valores da tabela a serem inicializados aleatoriamente (considere n=100 e que cada elemento pode variar entre –12.5 e 45.5).

Problema 5.4 – Pesquisa binária (e inserção eventual) de inteiro – Médio

Implemente uma função que insira um número inteiro (n.º de sócio) numa tabela ordenada de inteiros, se este não existir na tabela. A função deverá usar pesquisa binária para procurar na tabela o número procurado. Se este não existir, insere-o na posição correta de forma a manter a tabela ordenada.

Os parâmetros de entrada da função deverão ser: um ponteiro para a tabela, o número de elementos na tabela, o número máximo de elementos da tabela e o número inteiro a pesquisar (e a inserir eventualmente). A função deverá devolver zero se o número já existir, 1 no caso de ele ser inserido e –1 se o número não existir mas a tabela estiver cheia.

Teste a função com um programa feito por si e com 10 valores da tabela a serem inicializados aleatoriamente. Considere que o nº máximo de elementos na tabela é 15 e que cada elemento pode variar entre 1 e 100.

Problema 5.5 – Pesquisa binária de tabela indexada – Médio/Demorado

Faça um programa que permita fazer uma pesquisa binária a uma tabela indexada de *structs*. Para facilitar a tarefa defina uma tabela desordenada de BI's com 15 fichas (nome, número, altura, naturalidade) com 3 tabelas de indexação, uma que possui a ordenação por nome, outra por número e outra por altura. O programa deverá ter várias opções: 1-procurar por nome, 2-procurar por número 3-procurar por altura, 4-nome seguinte, 5-número seguinte, 6-altura seguinte. Todas as opções de procura

devem implementar pesquisa binária à tabela indexada. Inicialmente, antes da primeira pesquisa, a "ficha seguinte" será a primeira da sua ordem.

Problema 5.6 – Pesquisa binária de nome – Difícil/Demorado

Faça uma função que imprima no monitor todas as fichas de uma tabela de *structs* ordenada por nome, que possuam o nome de uma pessoa começado por uma substring passada como parâmetro. A função deverá usar pesquisa binária. Um possível protótipo para esta função será:

```
void imprimeInicio(TBI tab[], int nelems, char *subString);
```

A estrutura de cada ficha e a tabela poderão ser:

```
typedef struct {  // local onde está definido o tipo TBI, uma ficha com os
  char nome[30];  // dados de um bilhete de identidade
  long int numBI;
  float altura;
} TBI;

TBI tabBI[100];   // sem ser variável global
```

Um possível resultado para `imprimeInicio(tabBI, 50, "Carlos");` seria:

```
[20]: 8123457,               Carlos Alberto, 1.70 m
[21]:11828399,          Carlos Luis Antunes, 1.80 m
[22]: 9327681,                 Carlos Silva, 1.74 m
```

Notas:

- Utilize a função **`strcasecmp()`** ou **`strcmpi()`** para o ajudar na resolução do problema. Estas funções fazem o mesmo que a função **`strcmp()`** mas não fazem distinção entre letras minúsculas e letras maiúsculas.
- Uma pesquisa binária não acha obrigatoriamente a primeira ocorrência de uma dada igualdade, mas apenas uma delas. Na resolução deste problema, é preciso ter isto em conta.

Problema 5.7 – Comparação entre pesquisa sequencial e binária – Médio/Demorado

Faça um programa que permita comparar o desempenho entre uma pesquisa sequencial (ou linear) e uma binária (ou dicotómica) a um ficheiro com 1 milhão de inteiros.

O programa deverá ter 3 opções num menu:

1 – Criar o ficheiro: deverá criar o ficheiro com 1 milhão de inteiros múltiplos de 10, a começar no 10 e a terminar no 10.000.000 (pretende-se criar um ficheiro conhecido);

2 – Procurar a posição de um inteiro por pesquisa sequencial. Pede-se um número e mostra-se a sua posição (índice) no ficheiro. Por exemplo 10000000 daria 999999, 7 dará –1 (nº inexistente), e 10 dará 0. Esta opção deverá também imprimir no ecrã quanto tempo demorou a encontrar a posição.

3 – Procurar a posição de um inteiro por pesquisa binária. O comportamento deverá ser análogo ao da opção 2. Esta opção deverá mostrar no ecrã o tempo que demorou a pesquisa.

Problema 5.8 – Pesquisa sequencial – Médio (saiu no exame da época de recurso de 05/07/2016)

Considere uma tabela de fichas (`structs`) de filiais de uma empresa, sendo cada ficha constituída por: nome (55 caracteres úteis), nº de trabalhadores (inteiro) e volume de negócios (real de precisão dupla, em euros). A tabela pode conter até 200 fichas, havendo uma variável (`nfiliais`) com o nº de fichas existentes na tabela.

a) Declare a `struct`.

b) Implemente uma função para contar (e devolver) o nº de sucursais com volume de negócios superior a `valorInf` e inferior a `valorSup`. A função, chamada `contaFiliais()`, tem como parâmetros uma tabela de elementos do tipo declarado na alínea a), o nº de elementos (filiais) na tabela, `valorInf` e `valorSup`. Esta função não escreve nada no ecrã.

c) Implemente uma função `main()` de teste para mostrar no ecrã o nº de filiais com volume de negócios superior a 100.000 euros e inferior a 300.000 euros, para uma tabela inicializada por si com 4 filiais.

Problema 5.9 – Pesquisa sequencial – Médio (adaptado do 1º teste de frequência de 20/03/2012)

Considere uma tabela de fichas (`structs`) de alunos de uma disciplina, sendo cada ficha constituída por nome (50 caracteres úteis), nº de aluno e ano de nascimento.

a) Declare a `struct`.

b) Faça uma função para eliminar de uma tabela de fichas do tipo definido em a), passada como parâmetro da função, uma ficha dada pelo seu índice (também passado como parâmetro da função). O nº de elementos da tabela é passado por referência. A função não precisa de verificar a consistência dos parâmetros.

c) Faça uma função para eliminar de uma tabela destas, passada como parâmetro da função, a ficha da pessoa mais velha (se houver várias pessoas mais velhas com a mesma idade pode ser eliminada uma qualquer).

d) Implemente uma função `main()` de teste, para uma tabela inicializada com 5 alunos.

Obs.: Aconselha-se a elaborar previamente em papel um esboço ou desenho da tabela, de modo a perceber o problema e as suas implicações. Comente o código.

Problema 5.10 – Pesquisa binária – Médio (adaptado do 1º teste de frequência de 29/03/2017)

Considere a estrutura de dados: `struct ficha{ int id; float campo_1; float campo_2; };`
Implemente a função recursiva `int pesquisaBinaria(ifstream &ficheiro, int inicio, int fim, float valor)` para realizar uma pesquisa binária de `valor` no campo `campo_2` das fichas armazenadas no ficheiro em modo binário. Pode assumir que o ficheiro já se encontra aberto e que as fichas estão ordenadas por ordem crescente segundo os valores guardados em `campo_2`. A função deve devolver a posição no ficheiro da primeira ficha encontrada contendo `valor` no campo `campo_2`, entre os índices `inicio` e `fim` do ficheiro (posições absolutas a partir do início do ficheiro), ou -1 se não for encontrada a ficha pretendida.

Recorde que `ficheiro.seekg(pos)` ajusta o ponteiro para leitura na posição absoluta `pos` a partir do início do ficheiro e que a função `ficheiro.read(unsigned char *buffer, int n)` lê em modo binário `n` bytes de informação do ficheiro, guardando a informação lida na tabela `buffer`.

Problema 5.11 – Pesquisa sequencial – Fácil (adaptado do 1º teste de frequência de 12/03/2014)

Considere uma tabela de fichas (`structs`) de sucursais de uma empresa, sendo cada ficha constituída por: `nome` (60 carateres úteis), `empregados` (inteiro) e `area` (real, em m^2).

a) Declare a `struct`.
b) Implemente uma função para retornar o número de sucursais com área superior a `areaDada`. A função, chamada `contaDim()`, tem como parâmetros uma tabela de elementos do tipo declarado na alínea a), o n.º de elementos (ou sucursais) na tabela e `areaDada`. Considere que a tabela está ordenada por área crescente, não precisando por esse motivo de fazer uma pesquisa sequencial completa da tabela.
c) Implemente uma função `main()` de teste para contar e mostrar no ecrã o número de sucursais com área superior a 100 m^2, para uma tabela inicializada com 5 sucursais.

   Obs.: Comente o código. Os 100 m^2 devem ser definidos numa constante global.

Problema 5.12 – Pesquisa binária de tabela indexada – Médio

Faça um programa que permita fazer uma pesquisa binária a uma tabela indexada de fichas (`structs`). Para facilitar a tarefa defina uma tabela ordenada por ordem alfabética de cartões de cidadão (CC) com 15 fichas (nome, número, altura, naturalidade) e uma tabela de indexação para a ordenação por número. O programa deverá previamente efetuar a ordenação da tabela de índices, por número crescente do CC. O programa deve solicitar repetidamente ao utilizador um número de CC a procurar, mostrando os vários campos da ficha no caso de o número existir e uma mensagem adequada no caso contrário. O programa termina quando o utilizador introduzir o número 0 (zero).

ESTRUTURAS DE DADOS E ALGORITMOS
Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 6
INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

6.1. Objetivos

Objetivos que o aluno deve atingir com esta ficha:
- Compreender o conceito de classe e objeto;
- Compreender o conceito de construtor e destrutor;
- Aplicar exemplos de ponteiros para classes.

6.2. Classes: declaração e implementação

A programação orientada a objetos tem como principais objetivos reduzir a complexidade no desenvolvimento de software e aumentar a sua produtividade.

Um objeto é uma abstração de software que pretende representar um determinado tipo de dados (informação) e também o conjunto de operações ou funções de manipulação desses dados. Um objeto é formado por um conjunto de atributos (variáveis) e métodos (procedimentos, funções ou rotinas). Os atributos possuem um tipo de dados que define os possíveis valores que o atributo pode representar, como um número inteiro, um número real ou uma *string*. Os métodos são rotinas que, quando executadas, realizam alguma tarefa específica, como alterar o valor de um atributo do objeto.

Uma classe é uma descrição ou um modelo para um objeto. Uma classe define de forma abstrata os atributos e os métodos que representam características de um conjunto de objetos semelhantes. O conjunto de todos os atributos e métodos de uma classe designa-se por membros da classe.

O conceito de classe é um dos pilares da programação orientada a objetos, por permitir a reutilização efetiva de código. Um objeto é definido como uma instância (ou concretização) de uma determinada classe. Para melhor compreender as vantagens da utilização de objetos em programação, tomemos o seguinte exemplo:

```
// Estrutura de dados que define um retângulo
typedef struct {
   // Atributos do Retângulo
   int altura;
   int comprimento;
} TRetangulo;
```

Se pretendêssemos implementar uma série de funções para manipular retângulos definidos através desta estrutura, ter-se-ia de criar funções com um parâmetro deste tipo, como a função que se apresenta a seguir:

```
// Desenha um retângulo com as caraterísticas guardadas numa variável
// do tipo estrutura TRetangulo
void desenhaRetangulo(TRetangulo &ret) {

   // O que será que fazem as linhas que se seguem?
   cout << endl;
   for (int i=0; i < ret.altura; i++) {
      for (int j=0; j < ret.comprimento; j++) {
          if((i>0) && (i<ret.altura-1) && (j>0) && (j<ret.comprimento-1))
             cout << " ";
          else
             cout << "*";
      }
      cout << endl; // Para que serve esta instrução?
   }
   cout << endl;
}
//------------------------------------------------------------------------------
int main() {
   TRetangulo umRetangulo;

   umRetangulo.altura  = 5;
   umRetangulo.comprimento   = 7;

   desenhaRetangulo(umRetangulo);
}
```

O C++ permite resolver o mesmo problema de uma forma muito mais expedita, utilizando uma classe de objetos para representar um retângulo. Neste caso, seria possível combinar os atributos do retângulo e a função que o desenha numa *única* ***declaração*** e ***definição*** (ou ***implementação***). A declaração da classe de objetos do tipo retângulo poderia ser algo semelhante ao seguinte:

```
// Classe de retângulos
class CRetangulo {
  private:
     // Atributos do retângulo
     int altura;
     int comprimento;

  public:
     // Método que desenha o retângulo
     void desenha(void);
}; // O «;» é essencial: não esquecer ou ocorrerão erros de
   //  compilação noutras partes do programa !!!
```

A implementação do método `desenha()` definido no âmbito da classe `CRetangulo` poderá ser a seguinte:

```
// Definição/implementação do método que desenha o retângulo
// guardado pela classe
// Veja-se o nome da classe seguida do "scope resolution operator" ::
// é essencial para o compilador saber a que classe pertence o método...
void CRetangulo::desenha(void) {
   for (int i=0; i < altura; i++) {
      for (int j=0; j < comprimento; j++) {
          if ((i>0) && (i<altura-1) && (j>0) && (j<comprimento-1))
             cout << " ";
          else
             cout << "*";
      }
      cout << endl;
   }
   cout << endl;
}
```

Notas:

- O nome de uma classe obedece normalmente a uma convenção que determina que este seja precedido por um prefixo C (a indicar a definição de uma classe de objetos).
- As palavras reservadas `private`, `public` e `protected` determinam/controlam a forma como é permitido o acesso aos membros da classe.
  - `private`: Os membros privados de uma classe só são acessíveis a partir de métodos internos da classe ou então por objetos de classes declaradas como «amigas».
  - `protected`: Os membros protegidos de uma classe são visíveis a partir de métodos internos da classe, por objetos de classes declaradas como «amigas», e também por objetos de todas as suas classes derivadas.
  - `public`: Os membros públicos de uma classe são visíveis em qualquer ponto onde a classe seja visível.
  - Se declararmos membros de uma classe sem especificarmos qualquer tipo de visibilidade, então esses membros são por omissão do tipo `private`.

6.3. Objetos: construtores e destrutores, instanciação e membros estáticos

Uma classe não faz sentido se não houver maneira de a instanciar. Assim, analogamente aos tipos normais que se instanciam através de variáveis, os **objetos** permitem instanciar as classes. Desta forma, num programa em C++ existirão quase sempre vários objetos da mesma classe, cada um correspondendo a uma instanciação diferente, ocupando regiões de memória diferentes e representando, provavelmente, dados diferentes.

Em cada classe existe um método obrigatório que convém ser declarado e definido e que determina a forma como se podem fazer instanciações de uma classe: o **construtor por omissão ou por defeito**. Este método declara-se sempre da mesma forma: tem o nome da classe, não tem argumentos (parâmetros) e é

sempre `public`. Na sua definição, é esperado que se inicializem os atributos da classe, mas não é obrigatório fazê-lo. O construtor é *sempre* chamado automaticamente quando um novo objeto é criado (instanciado).

O código definido a seguir complementa a classe anterior definindo um construtor por omissão:

```
...
// Classe de retângulos
class CRetangulo {
  private:
     // atributos do retângulo
     int altura;
     int comprimento;

  public:
     // Construtor por defeito ou por omissão
     CRetangulo() {
          altura = comprimento = 0;
     }
     // Método que desenha o retângulo
     void desenha(void);
};
```

Como se pode observar, foi definido um construtor por defeito que permite inicializar os atributos do retângulo a zero. Neste exemplo, podemos também verificar que é possível definir métodos dentro da declaração da classe: chama-se a isto **definição *inline***. Normalmente, apenas se utiliza esta forma de definir os métodos em situações em que estes não são extensos, para não prejudicar a legibilidade do código e em particular da definição da classe.

O C++ permite também definir membros de uma classe que são partilhados por todos os objetos dessa classe: os chamados membros estáticos ou `static`. Estes membros ficam disponíveis para todos os objetos, uma espécie de variável global da classe. Uma utilidade para este tipo de métodos é apresentada no exemplo que se segue: imagine-se que se quer contabilizar o número de objetos de uma classe; não existe maneira de o fazer automaticamente, a não ser recorrendo a um membro `static` que guarde essa informação. Para manter a boa prática de encapsulamento, pode-se usar um membro `static` mas `private` e aceder a ele a partir de métodos. Vejamos um exemplo utilizando mais uma vez a classe `CRetangulo`:

```
...
// Classe de retângulos
class CRetangulo {
  private:
     // Atributos do retângulo
     int altura;
     int comprimento;

     // Membro acessível a todos os objetos desta classe,
     // ocupando apenas uma zona de memória
     static int numRet;
```

```
public:
   // Construtor por omissão ou defeito
   CRetangulo() {
        altura = comprimento = 0;
        numRet++;
   }

   // Construtor com parâmetros
   CRetangulo(int a, int c) {
       altura = a;
        comprimento = c;
        numRet++;
   }

  // Construtor por cópia
   CRetangulo(const CRetangulo &r) {
       altura = r.altura;
        comprimento = r.comprimento;
        numRet++;
   }

   // Destrutor
   ~CRetangulo() {
        numRet--;
   }

   // Método que inicializa os atributos do retângulo
   void inicializaTamanho(int a, int c) {
     altura = a;
     comprimento = c;
   }

   // Método que devolve os atributos do retângulo (por referência)
   void devolveTamanho(int &a, int &c) {
     a = altura;
     c = comprimento;
   }

   // Método que utiliza o atributo "static" para imprimir o número
   // total de objetos deste tipo.
   void imprimeNumRetangulos(void){
     cout << "O numero atual de objetos-retângulo e': " << numRet;
     cout << endl;
   }


   // Método que utiliza o atributo "static" para devolver o número
   // total de objetos deste tipo
```

```
      int devolveNumRetangulos(void){
        // A utilização do ponteiro "this" era desnecessária, mas mostra
        // como funciona...
        return this->numRet;
      }

      // Método que desenha o retângulo
      void desenha(void);

};  // fim da classe CRetangulo
//-----------------------------------------------------------------
int CRetangulo::numRet = 0;  // Definição do membro estático
//-----------------------------------------------------------------
...
```

O programa principal que usaria a classe poderia ser o seguinte:

```
...
int main(void) {
   // Cria retângulo e chama construtor por omissão
   CRetangulo retangulo;

   // Cria retângulo utilizando construtor por enumeração
   CRetangulo outroRetangulo (7, 5);

   // Desenha o retângulo
   outroRetangulo.desenha();

   retangulo.inicializaTamanho(5, 7);

   // Desenha outro retângulo
   retangulo.desenha();

   // Cria outro objeto retângulo e usa a sobrecarga alternativa
   // para o construtor
   CRetangulo ultimoRetangulo(4, 5);

   // Imprime o número de objetos-retângulo criados até ao momento
   ultimoRetangulo.imprimeNumRetangulos();

   // Criação de uma tabela estática de objetos-retângulo
   CRetangulo variosRets[5];


   // Criação de uma tabela dinâmica de objetos-retângulo
   CRetangulo *eMaisEstesRets = new CRetangulo[10];

   // Quantos objetos-retângulo existem neste momento?...
```

```
    eMaisEstesRets[0].imprimeNumRetangulos();

    // Vamos guardar este número para uso futuro...
    int antNumRets = eMaisEstesRets[9].devolveNumRetangulos();

    // Vamos agora apagar a tabela dinâmica...
    delete [] eMaisEstesRets;

    // Quantos objetos-retângulo há agora?...
    variosRets[0].imprimeNumRetangulos();

    // Mas...
    cout << "Mas já foram " << antNumRets << "..." << endl;
    return 0;
}
```

Neste exemplo, além do que foi referido anteriormente, foi acrescentada uma série de métodos úteis e algumas novidades:

- Em primeiro lugar, foram adicionados mais dois construtores alternativos (por sobrecarga de função) que permitem definir implicitamente os parâmetros do retângulo e copiar os dados de um retângulo já existente. Em todos os construtores, é incrementado o membro estático `numRet`.
- Pode ser também definido um destrutor que é chamado sempre que um objeto é destruído. Supostamente, será nesse método que se faz a "limpeza da casa", como libertar a memória ocupada dinamicamente, *etc*. No nosso caso concreto, decrementa-se apenas o contador de objetos.
- O ponteiro `this` nada mais faz que conter o endereço do objeto na memória e é sempre implicitamente usado quando se faz referência a um membro a partir de um método da classe...
- A criação e destruição de ponteiros/tabelas dinâmicos/as de objetos utilizando os operadores `new` e `delete`: podem usar-se quaisquer tipos de dados com estes operadores, sejam classes ou não. Apenas se pode criar e destruir dinamicamente um objeto com um ponteiro, ou uma tabela de objetos se forem usados os parênteses retos.

6.4. Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à programação orientada a objetos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios que não for possível realizar durante as aulas práticas.

Problema 6.1 – Fácil

Acrescente métodos à classe base `CRetangulo` que permitam calcular a área e o perímetro do retângulo. Estes métodos não devem aceitar qualquer parâmetro. Implemente também um método que verifique se "este" retângulo está contido noutro retângulo passado como parâmetro por referência, ou seja se o primeiro retângulo tem uma largura e um comprimento menores ou iguais aos do segundo retângulo.

Problema 6.2 – Fácil

Implemente uma classe que permita representar a ficha de um aluno, com nome do aluno, *username* e nome do curso (e.g., MiEEC), e que armazene também as suas classificações em 9 disciplinas diferentes: PdC, ALG, AM1, LSD, AM2, LEC, EDA, MO e SMP. Implemente três métodos: (1) um que possibilite calcular e imprimir a média das suas classificações; (2) outro que devolva a nota máxima; (3) e um último que imprima o nome da disciplina a que o aluno obteve a nota mínima.

Problema 6.3 – Médio

Defina uma classe que permita representar um dispositivo elétrico. Deve permitir definir o nome/referência do dispositivo (`string` de 10 carateres), a potência de consumo, uma `struct` para definir a data de validade (no formato dd/mm/aaa) e um membro (de que tipo?) que permita definir se se encontra dentro da garantia (`true` ou `false`).

Problema 6.4 – Médio

Implemente uma classe denominada `CCash` que permite definir um preço em Euros. A classe deve armazenar um valor em dois membros inteiros diferentes: `euros` e `centimos`. Implemente um construtor por defeito (que coloca o preço em 1.99€) e um construtor por enumeração (que recebe o preço em Euros e em Cêntimos). Implemente também o método `troco()` e o método `imprime()`. O primeiro recebe como parâmetro passado por referência um objeto `CCash` com o valor pago em dinheiro e devolve um objeto `CCash` com o montante a devolver como troco, assumindo que o preço a pagar está guardado "neste objeto" (i.e. no objeto para o qual é chamado o método). O segundo mostra no ecrã o conteúdo do objeto (por ex. no formato "5,02 Eur"), formatando corretamente a sua apresentação na consola com os manipuladores da biblioteca `iomanip`.

Problema 6.5 – Médio

Dada a seguinte definição da classe `CData` que permite representar datas, implemente o construtor e os métodos indicados. Implemente também um novo método privado que permita verificar se uma dada data é ou não válida.

```
class CData {
    int dia;
```

```
    int mes;
    int ano;
    int horas;
    int minutos;
  public:
    CData();                      // Construtor por defeito (00:00 de 01.01.1970)
    CData(int d, int m, int a, int h, int min);     // Construtor por enumeração
    int devolveDia();             // Devolve o dia da data atual
    int devolveMes();             // Devolve o mês da data atual
    int devolveAno();             // Devolve o ano da data atual
    void mudaHoraDoDia(int, int);// Permite alterar a hora do dia
    void mudaData(int, int, int);// Permite introduzir uma nova data
     void escreveData();          /* Escreve hora e data no ecrã no
                                        formato hh:mm de dd/mm/aaaa */
    int numeroFDS();              /* Devolve o número de fins de semana decorridos
                                        desde o início do ano */
};
```

Problema 6.6 – Médio

Defina uma classe que permita representar frações inteiras com `numerador` e `denominador`. Implemente os três construtores e também um método que permita reduzir uma fração à sua forma irredutível (por exemplo transformar 4/8 em 1/2). Implemente também um método que permita apresentar a fração no ecrã. Preveja o caso em que o denominador seja igual a 1, passando a fração a ser representada sem denominador; por exemplo, a fração 2/1 deverá ser apresentada apenas como 2.

Problema 6.7 – Difícil

Defina uma classe que permita representar *passwords* recorrendo a alocação dinâmica de memória. Implemente um construtor por enumeração (aceita uma `string` que defina uma *password*) e por cópia. Implemente ainda os seguintes métodos:

`comprimentoPassword()` – Devolve o comprimento da *password* em número de carateres;

`baralhaPassoword()` – Troca a segunda com a primeira metade da *password* (por exemplo, "tpb3u8as" passa a "u8astpb3");

`apagaMetadePassword()` – Elimina a primeira metade da *password* formando uma nova *password* com metade do tamanho dos carateres da original. Se uma *password* contiver N carateres, a nova *password* será formada pelos N/2 últimos caracteres da original (por exemplo, "latina" passa a "ina"). Analise o que acontece a *passwords* com número ímpar de carateres;

`converteMinusculas()` – Converte a *password* para letras minúsculas.

Problema 6.8 – Médio (adaptado do exame de recurso de 05/07/2012)

Declare uma classe denominada `CDiametro` que permite definir um diâmetro em pés e polegadas (ambos inteiros), tendo em conta as alíneas seguintes.

a) Implemente um construtor por omissão (por defeito), que coloca o diâmetro em 10' 0" (10 pés e 0 polegadas) e um construtor por enumeração (que recebe o comprimento em pés e polegadas, sendo as polegadas facultativas, tendo o valor 0 por omissão).

b) Nesta classe, as polegadas são sempre inferiores a 12 (1 pé = 12 polegadas). Crie um método chamado `corrige_polegadas()` para garantir esta restrição, corrigindo o valor (das polegadas e pés) no caso de as polegadas serem superiores a 12 (as polegadas podem ser > 24 ou qualquer outro múltiplo de 12).

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 6.9 – Médio **(adaptado do teste de frequência de 25/03/2015)**

Considere uma classe para gerir uma tabela ordenada de números reais (e.g. amostras de temperaturas). A classe aloca dinamicamente memória para guardar a tabela. Os atributos (privados) são a tabela propriamente dita (o primeiro elemento é apontado por `tab`), a sua dimensão (`max`) e um inteiro (`n`) com o número de elementos na tabela. A tabela deve estar sempre ordenada por ordem decrescente, com os dados agrupados no início da tabela.

a) Declare a classe com os métodos (públicos) seguintes: construtor por omissão – cria uma tabela vazia com dimensão 100 (aloca memória e atribui zero a `n`; construtor com um parâmetro – cria uma tabela vazia com dimensão genérica indicada pelo parâmetro; destrutor; construtor por cópia – cria uma tabela que é uma cópia de outra recebida como parâmetro; `apaga()` – apaga (esvazia) a tabela, i.e. atribui zero a `n`; `vazia()` – verifica se a tabela está vazia; `insere()` – acrescenta um elemento ordenadamente, devolvendo `false` se a inserção falhar por a tabela estar cheia (`true` caso contrário). A tabela está cheia se `n == max`.

b) Implemente os construtores e o destrutor.

c) Implemente os restantes métodos. Na implementação do método `insere()`, utilize o princípio do algoritmo de ordenação por bolha, assumindo que a tabela está ordenada antes da inserção do novo elemento (dado como parâmetro deste método).

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 6.10 – Médio **(saiu no teste de frequência de 29/03/2017)**

A classe de objetos `CDuracao` permite definir uma duração temporal em horas e minutos. Possui os atributos `horas` e `minutos` (tipo `int`) e os métodos seguintes: i) um construtor por defeito/omissão (objeto com uma duração por defeito igual a 1h30m); ii) um construtor por enumeração (recebe dois inteiros com uma duração qualquer em horas e minutos); iii) um construtor por cópia; iv) o método `compara()` que recebe como parâmetro, por referência, outro objeto da classe `CDuracao` e devolve um n.º inteiro negativo, zero ou um n.º positivo, consoante o objeto seja menor (em duração temporal), igual ou superior ao parâmetro, respetivamente.

a) Escreva a declaração da classe de objetos `CDuracao`. Nesta alínea, não deve implementar qualquer método.
b) Implemente os construtores por defeito, por enumeração e por cópia da classe `CDuracao`.
c) Implemente o método `compara()` da classe `CDuracao`.

Implemente uma função main() para ir testando a implementação da classe.

Problema 6.11 – Médio **(saiu no teste de frequência de 13/04/2018)**

A classe de objetos `CNumComplexo` permite representar números complexos, i.e. números com parte real e parte imaginária e realizar operações com os mesmos. Possui os atributos `preal` e `pimag` (tipo `double`) e os métodos seguintes: i) construtor por defeito/omissão (número real 1+0j); ii) construtor por enumeração (recebe dois números reais com os valores da parte real e parte imaginária); iii) o método `compara()` que compara o objeto com outro objeto passado como parâmetro e devolve um booleano consoante dois números complexos sejam ou não iguais.

a) Escreva a declaração da classe de objetos `CNumComplexo`. Nesta alínea, não deve definir/implementar qualquer método.
b) Defina os construtores por defeito e por enumeração da classe `CNumComplexo`.
c) Defina o método `compara()` da classe `CNumComplexo`.

Implemente uma função main() para ir testando a implementação da classe.

Problema 6.12 – Fácil (saiu no teste de frequência de 20/03/2019)

A classe de objetos `Odometria` representa dados da odometria de um robô que se move num plano, i.e. da sua pose (posição e orientação) num dado instante de tempo. Possui os atributos `x`, `y`, e `theta` (todos números reais de dupla precisão). Possui os métodos públicos: i) construtor por defeito/omissão (todos os atributos com valor `0.0`); ii) construtor por enumeração (recebe três números reais correspondentes a `x`, `y`, e `theta`, respetivamente); iii) `void simplifOrient(void)` que converte o valor do atributo `theta` de forma a reduzir o ângulo de orientação do robô a um ângulo pertencente ao intervalo]-π; π].

a) Escreva a declaração da classe de objetos `Odometria`. Nesta alínea, não deve definir/implementar qq. método.
b) Defina os construtores por defeito e por enumeração da classe `Odometria`.
c) Defina o método `simplifOrient()`. Defina a constante `PI` (constante matemática π). O algoritmo consiste em somar ou subtrair `2*PI` ao ângulo `theta` até que este se situe no intervalo pretendido.

Implemente uma função main() para ir testando a implementação da classe.

ESTRUTURAS DE DADOS E ALGORITMOS
Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 7
TÓPICOS AVANÇADOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

7.1 Objetivos

Objetivos que o aluno deve atingir com esta ficha:
- Compreender o conceito de membros constantes;
- Compreender o conceito de derivação de classes, heranças e hierarquia de classes;
- Compreender o conceito de membro virtual e polimorfismo;
- Compreender o conceito de sobrecarga de operadores.

7.2 Membros constantes

Quando um objeto de uma classe é declarado como constante (`const MyClass myObject;`) os seus atributos não podem ser alterados (só podem ser lidos). No entanto, o construtor ainda é chamado e é-lhe permitido inicializar e modificar os atributos.

Os objetos constantes só podem invocar métodos declarados como constantes. Um método declarado como constante não pode alterar os atributos não estáticos dos objetos da classe (pode alterar atributos estáticos). Também não pode chamar métodos não constantes. Para declarar um método como constante usa-se a palavra-chave `const` depois do parêntesis da direita da lista de parâmetros, antes da chaveta de início do corpo do método:

```
int getLargura() const {return largura;}
```

Os atributos declarados como constantes não podem ser alterados depois da criação do objeto. Podem ser inicializados após a lista de parâmetros, antes do corpo do construtor:

```
class CPoligono {
        int largura, altura;
        const int numLados; // atributo constante
  public:
        CPoligono(int larg, int alt, int nLad) : numLados(nLad)
                    { largura = larg; altura = alt; }
        int getLargura(void) const;      // método constante
  //    void setLargura(int larg) const { largura = larg; } // Erro!!!
}; // Fim da classe
//-----------------------------------------------------------------
int CPoligono::getLargura(void) const { return largura; }
//-----------------------------------------------------------------
```

7.3 Derivação de classes, membros virtuais e polimorfismo

Uma das grandes vantagens da programação orientada a objetos reside na possibilidade de facilmente podermos reutilizar código escrito anteriormente. Uma noção que contribui para esse facto é a noção de ***herança*** de classes. É possível criar novas *classes derivadas* que sejam descendentes de uma *classe base* já existente. Uma classe derivada inclui os novos membros que nela sejam definidos e também todos os membros (públicos, protegidos e privados) da classe base, com a exceção dos construtores e do destrutor, da sobrecarga de operadores e da definição das classes e funções amigas. Embora os construtores e os destrutores não sejam herdados, a classe derivada mantém os construtores e o destrutor da classe base, que são sempre chamados antes de ser chamado um construtor ou o destrutor da classe derivada.

Imagine que pretendia criar uma nova classe que representasse retângulos a cheio ou blocos. Em vez de criarmos uma classe completamente nova, onde se teria de redefinir todos os atributos e métodos que definem um retângulo, poderíamos aproveitar a definição da classe `CRetangulo` anteriormente definida (ver ficha anterior):

```
class CRetangulo {
  protected:    // estes membros são visíveis pelas classes derivadas
     // Atributos do retângulo
     int altura;
     int comprimento;

  public:
     CRetangulo();                        // Construtor por defeito
     CRetangulo(int a, int l);            // Construtor por parâmetros
     CRetangulo(const CRetangulo &r);     // Construtor por cópia
     void desenha(void) const;            // Método que desenha o retângulo
}; // fim da classe
//------------------------------------------------------------------------
```

Podemos aproveitar a definição desta classe para definir uma nova classe `CBloco` como uma *classe derivada* de `CRetangulo`. Note que como pretendemos também aceder nos métodos da classe derivada aos membros `altura` e `comprimento` herdados da classe `CRetangulo`, tivemos que alterar a visibilidade destes membros na classe `CRetangulo` de privados para protegidos já que os membros privados não são visíveis por classes derivadas (exceto se forem amigas).

```
class CBloco : public CRetangulo {
  protected:
    bool cheio;

  public:
     /* Temos de definir os construtores desta nova classe, uma vez que
         eles não são herdados da classe base */

     CBloco() {
        altura = comprimento = 0;
        cheio = true;
```

```
        }

        CBloco(int a, int c, bool ch = true) {
           altura = a;
           comprimento = C;
           cheio = ch;
        }

        void inicializaCheio(bool ch) {
           cheio = ch;
        }

        void desenha(void) const;

    }; // fim da declaração da classe
    //-----------------------------------------------------------------
    void CBloco::desenha(void) const {
        cout << endl;
        if (cheio) {
           for (int i=0; i < altura; i++) {
              for (int j=0; j < comprimento; j++)  cout << "*";
              cout << endl;
           }
           cout << endl;
        } else
        CRetangulo::desenha();  // Chama o método da classe base
    }
    //-----------------------------------------------------------------
```

Repare que nesta nova classe redefinimos o comportamento do método `desenha()`. Neste caso, podemos desenhar o retângulo a cheio ou não, dependendo do valor do membro `cheio`. No caso de `cheio` ser `false` então o desenho do retângulo é igual ao da classe base e chamamos o método `desenha()` da classe `CRetangulo`.

Para testarmos esta nova classe, podemos definir o programa principal apresentado a seguir:

```
    int main() {
       CBloco *umBloco = new CBloco(7,5, true);

       cout << "Agora e' cheio..." << endl;
       umBloco->desenha();

       umBloco->inicializaCheio(false);
       cout << "Agora ja' nao..." << endl;
       umBloco->desenha();

       delete umBloco;
       return 0;
```

```
}
```

Por forma a compatibilizar a utilização de ponteiros do tipo da classe base para referenciar as suas classes derivadas, por vezes é necessário declarar na classe base métodos que apenas vão ser definidos nas classes derivadas. Esses membros são definidos recorrendo à palavra reservada `virtual`.

Os métodos virtuais devem ser redefinidos nas classes derivadas. Se tal não acontecer é aplicado o comportamento definido na classe base. Esta propriedade designa-se por polimorfismo porque cada classe derivada pode alterar a forma de funcionamento dos métodos virtuais.

Para ilustrar este conceito tomemos o seguinte exemplo: consideremos uma nova classe, derivada da classe `CBloco`, e que produz blocos "redondos" cujos cantos do perímetro não são preenchidos.

```
class CRetangulo{
   ...
   virtual void desenha(void) const; /* Permite referenciar este
   ...                                  método nas classes derivadas */
}; // fim da declaração da classe
//---------------------------------------------------------------------
class CBlocoRedondo : public CBloco {
public:
   void desenha(void) const;
   CBlocoRedondo(){ }
   CBlocoRedondo(int a, int c, bool ch = true) {
     altura=a;
     comprimento=c;
     cheio=ch;
   }
   ~CBlocoRedondo(){}
}; // fim da declaração da classe
//---------------------------------------------------------------------
void CBlocoRedondo::desenha() const {
   cout << endl;
   for (int i = 0; i < altura; i++) {
     for (int j = 0; j < comprimento; j++) {
       if ((i > 0 && i < altura - 1 && j > 0 &&
            j < comprimento - 1 && !cheio) ||
           (i == 0 && j == 0) || (i == altura - 1 && j == 0) ||
           (i == 0 && j == comprimento - 1) ||
           (i == altura - 1 && j == comprimento - 1))
         cout << " ";
       else
         cout << "*";
     }
     cout << endl;
   }
   cout << endl;
}
//---------------------------------------------------------------------
```

Uma vez que a figura é diferente das anteriores, o método `desenha()` foi alterado para permitir desenhar esta nova figura. As três classes podem ser manipuladas da seguinte forma:

```
int main(void) {

   CBloco *umBlocoRedondo = new CBlocoRedondo(5, 5, true);
   cout << "Agora e' cheio..." << endl;
   umBlocoRedondo->desenha();

   umBlocoRedondo->inicializaCheio(false);
   cout << "Agora ja' nao..." << endl;
   umBlocoRedondo->desenha();
   delete umBlocoRedondo;

   CRetangulo *pFigura = new CBloco(3, 4);
   // Será que o programa sabe qual dos métodos desenha() chamar?
   pFigura->desenha();
   delete pFigura;

   // E aqui?
   pFigura = new CBlocoRedondo(4, 4);
   pFigura->desenha();
   delete pFigura;

   pFigura = new CRetangulo(3, 4);
   // E desta, será mais fácil de ver o que desenha?
   pFigura->desenha();
   delete pFigura;
   return 0;
}
```

Consegue "ver" as vantagens na utilização deste conceito em funções que lidam com parâmetros que são retângulos em geral?

7.4 Sobrecarga de operadores e construtores por cópia e conversão – funções amigas

Um dos conceitos mais úteis em programação orientada a objetos é o conceito de *sobrecarga de operadores*. Para além da sobrecarga de funções, é também possível modificar o comportamento de grande parte dos operadores, permitindo que estes se adaptem ao contexto de cada classe. Os operadores passíveis de serem sobrecarregados para classes definidas pelo programador (e não para os tipos convencionais) são:

```
+     -     *     /     %     ^     &     |     ~     !     =     <     >
+=    -=    *=    /=    %=    ^=    &=    |=    <<    >>    >>=   <<=   = =
!=    <=    >=    &&    ||    ++    --    ,     ->*   ->    ()    []
new   delete
```

Para ilustrar o conceito de sobrecarga de operadores, iremos de seguida sobrecarregar o operador binário + da nossa classe `CRetangulo`. O objetivo é somar dois retângulos, somando os lados, ou então somar um retângulo com uma constante, adicionando essa constante aos lados. Temos de ter presente que, em termos de classes, qualquer expressão do tipo `objeto1 + objeto2` é substituída pelo método `objeto1.operator + (objeto2)`. A partir daqui, fica claro que o primeiro operando é o próprio objeto onde se efetua a sobrecarga da operação e o segundo operando é passado como parâmetro do método. As duas formas de sobrecarga do operador + referidas anteriormente podem ser definidas da seguinte forma:

```
public:
   CRetangulo operator+(const CRetangulo &ret);
   CRetangulo operator+(const int aumento);
   ...
}; // fim da classe
//-----------------------------------------------------------------
/* Implementação da sobrecarga do operador + por forma a permitir a
   soma de dois objetos do tipo CRetangulo */
CRetangulo CRetangulo::operator + (const CRetangulo &ret) {
    return CRetangulo (altura+ret.altura, comprimento+ret.comprimento);
}
//-----------------------------------------------------------------
/* Implementação da sobrecarga do operador + por forma a
   permitir adicionar um retângulo a um inteiro */
CRetangulo CRetangulo::operator + (const int aumento) {
    return CRetangulo(altura + aumento, comprimento + aumento);
}
```

Repare na utilização da declaração `const` nos parâmetros de entrada dos dois métodos apresentados. A ideia é impedir que um parâmetro, mesmo que passado por referência, seja alterado quando passado à função. E porquê a passagem por referência neste caso e não a passagem de parâmetros por valor? A resposta tem a ver com uma questão de eficiência: é menos eficiente fazer uma cópia desse objeto, principalmente de um objeto que ocupe muita memória, do que utilizar um "pseudónimo", i.e. uma referência para o designar.

E se quisermos que a última sobrecarga, que permite adicionar uma constante inteira a ambos os lados do retângulo, possa, além do caso `CRetangulo + int`, também estender-se ao caso `int + CRetangulo`? Teremos de recorrer a uma estratégia diferente, uma vez que a sobrecarga de operadores numa classe assume sempre que o objeto atual é o primeiro parâmetro. A solução é definir uma função *global* da seguinte forma:

```
/* Esta função global funciona da seguinte forma:
   CRetangulo operator+(int operando1, CRetangulo operando2)
   operando1 (aumento) é o operando da esquerda e é do tipo inteiro e
   operando2 (ret) é o da direita e é do tipo CRetangulo. */
CRetangulo operator + (const int aumento, const CRetangulo &ret) {
   return CRetangulo (ret.altura + aumento, ret.comprimento + aumento);
}
```

Esta função tem, porém, um problema: ela tenta aceder a membros privados de `CRetangulo` (`altura` e `comprimento`). Como sabe, não é possível aceder a membros privados fora do âmbito da classe...

Para resolver esta questão foi criado um tipo de declaração especial: a declaração `friend`. Esta declaração define o conceito de funções amigas de uma classe. Uma *função amiga* de uma classe tem permissão para aceder aos membros privados dessa classe. No exemplo apresentado acima teremos de adicionar à declaração da classe `CRetangulo` a linha que se segue:

```
    public:
       ...
       friend CRetangulo operator+(int aumento, const CRetangulo &ret);
```

A partir do momento que esta função global é declarada "amiga", ela passa a poder aceder sem reservas a todos os membros da classe `CRetangulo`.

A seguir apresenta-se um exemplo de utilização das sobrecargas do operador +:

```
    int main() {
       cout << "Agora demonstremos o operador +..." << endl;
       CRetangulo ret1(1,1);
       ret1 = ret1 + CRetangulo(1,2);
       ret1.desenha();
       ret1 = ret1 + 1;
       ret1.desenha();
       ret1 = 2 + Ret1;
       ret1.desenha();
    }
```

Para saber mais sobre os operadores em C++, ver os links:
http://www.cplusplus.com/doc/tutorial/operators/
http://en.wikipedia.org/wiki/Operators_in_C_and_C++
http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

7.5 Exercícios sugeridos

Com o conjunto de exercícios apresentados a seguir, pretende-se que o aluno consolide os seus conhecimentos sobre tópicos avançados de programação orientada a objetos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios que não seja possível realizar durante as aulas práticas.

Problema 7.1 – Fácil

Utilizando a classe `CData` definida na ficha anterior, derive uma nova classe `CFeriado` que permita representar os feriados no calendário. Esta nova classe, para além da data, deve também armazenar a descrição do feriado. Altere o método `escreveData()` na nova classe por forma a permitir escrever a data seguida da descrição do feriado (por exemplo: 17:53 de 10/06/2019 – Dia de Portugal).

```
class CData{
  private:     // deverá ser "protected" para ser visível da classe derivada
    int dia;
```

```
    int mes;
    int ano;
    int horas;
    int minutos;
  public:
    CData();                      // Construtor por defeito (00:00 de 01.01.1970)
    CData(int d, int m, int a, int h, int min);     // Construtor por enumeração
    CData(const CData &data);     // Construtor por cópia
    int devolveDia()const;        // Devolve o dia da data atual
    int devolveMes()const;        // Devolve o mês da data atual
    int devolveAno()const;        // Devolve o ano da data atual
    void mudaHoraDoDia(int, int);// Permite alterar a hora do dia
    void mudaData(int, int, int);// Permite introduzir uma nova data
     void escreveData()const;     // Escreve hora e data no ecrã no
                                  //  formato hh:mm de dd/mm/aaaa
    int numeroFDS()const;         // Devolve o número de fins de semana decorridos
                                  //  desde o início do ano
}; // fim da declaração da classe
```

Implemente a sobrecarga do operador – por forma a devolver o número de dias que separam dois feriados diferentes.

Problema 7.2 – Fácil

Dada a classe `CCash` definida na ficha anterior, que permite definir um preço em euros, implemente as seguintes sobrecargas de operadores:

- Operador `+` – permite somar um objeto do tipo `CCash` a uma constante do mesmo tipo;
- Operador `!=` – permite verificar se dois preços são diferentes);
- Operador `%` – devolve um `int` que é a soma de apenas as componentes cêntimos de um objeto do tipo `CCash` com uma constante do mesmo tipo.

Problema 7.3 – Médio

Considere a classe `CPolinomio` que pretende representar polinómios do segundo grau do tipo $a_2x^2+a_1x+a_0$:

```
class CPolinomio{
   private:
     int grau;          // Contém sempre o grau do polinómio.
     double coef[3];    // Coeficientes do polinómio: [0]=a0, [1]=a1,…
   public:
     CPolinomio ();    // Inicialização por defeito (polinómio = 1)
     CPolinomio(double a2, double a1, double a0); /* construtor por parâmetros:
                                                     atualiza grau */
     CPolinomio(const CPolinomio &p);           // Construtor por cópia
     ~CPolinomio( );
}; // fim da classe
```

Implemente um método `escreve()` que permita escrever o polinómio representado pelo objeto. Implemente também a sobrecarga dos seguintes operadores: `-` (subtração de dois polinómios), `*`

(multiplicação dos coeficientes por uma constante inteira), e finalmente o operador `[]` que permita devolver o coeficiente correspondente ao índice passado como parâmetro (por exemplo, `polinomio[2]` deverá devolver o coeficiente a_2).

Problema 7.4 – Médio

Implemente uma classe `CVetor` que defina vetores tridimensionais $[x\ y\ z]^T$ através das suas coordenadas. Implemente os seguintes métodos:

- `modulo()` – devolve o módulo do vetor;
- `inverte()` – vetor passa a ter sentido contrário (simétrico);
- Sobrecarga do operador `+` – implementa a soma de dois vetores);
- Sobrecarga do operador `!=` – neste caso, considera-se que dois vetores são diferentes se têm módulos diferentes).

Problema 7.5 – Médio/Trabalhoso

Implemente a seguinte classe denominada `CEsfera`:

```
class CEsfera{
   double raio; // Raio da Esfera
   char cor[15];// String que indica a cor da Esfera

public:
   CEsfera();                // Construtor por defeito (Raio = 1.0, Cor = Branco)
   CEsfera(double r, const char* c);      // Construtor por parâmetros
   CEsfera(const CEsfera& e);// Construtor por cópia
   ~CEsfera();               // Destrutor
   double volume() const;    // Devolve o volume da Esfera(4./3*pi*Raio^3)

  // Sobrecarga de operadores:
   CEsfera operator+(const CEsfera&) const; // Sobrecarga do operador +
                      // somam-se os raios e a cor é a da primeira Esfera
   CEsfera& operator=(const CEsfera&);    // Atribuição de uma Esfera a outra
                                          //  Esfera
   bool operator==(const CEsfera& e) const; //Testa igualdade entre esferas
   bool operator==(double r) const; // Idem em que r é o raio da segunda
}; // fim da classe
```

Problema 7.6 – Médio/Trabalhoso

Implemente a seguinte classe denominada `CStreamVideo`:

```
class CStreamVideo {
  private:
    char stream[1024/8];  // Armazena os até 1024 bits da sequência de video
    int numBits;          // nº de bits da sequência (múltiplo de 8)
    bool erro;            // Este valor, que deve ser sempre atualizado, é igual a
       // true se o resultado da operação XOR entre todos os bits for 1
    bool verificaErro();  // permite atualizar o valor do membro Erro

  public:
    CStreamVideo();        // Inicialização por defeito com tudo a zero
    CStreamVideo(const char *seq, int nBits); /* Inicialização através de...
                   ... uma tabela de 0's e 1's (se não for múltiplo de 8 trunca) */
    CStreamVideo(const CStreamVideo &sv);     // Inicialização por cópia
    ~CStreamVideo( );                         //Destrutor
    bool Erro() const;                        // retorna o valor do membro Erro
    CStreamVideo operator + (const CStreamVideo &sv); // Concatena Streams
}; // fim da declaração da classe
```

Problema 7.7 – Difícil/Trabalhoso

Implemente uma classe denominada `CString` com a seguinte definição:

```
    class CString{
       int Comprimento;          // Comprimento da String
       char* PonString;          /* Ponteiro para a zona de memória onde a String
                                     está armazenada */
    public:
       CString();                // Construtor por defeito -> String vazia
       CString(const char *);    // Inicializa string a partir de um conjunto
                                 // de caracteres
       CString(const String&);   // Construtor por cópia
       ~CString();               // Destrutor
       GetLine();          // Pede uma linha ao utilizador e guarda-a na string
       int Length();// Devolve o comprimento da String

      // Sobrecarga de operadores:
       CString& operator=(const char*); // Atribuicao de um conjunto de
                                        //  carateres a uma String
       CString& operator=(const CString&);    // Atribuicao de uma String a outra
                                              // String

// continua...
```

```
    CString operator()(int,int);     // Devolve uma Substring, dada a posição
                                     //  inicial e o comprimento
    char& operator[](int);           // Devolve o caracter numa dada posicao
    CString& operator+=(const CString&);   // concatenacao de Strings, por
                                           // auto-incremento
    bool operator==(const CString&);       // Testa igualdade entre Strings
    bool operator==(const char*);          // Testa igualdade entre uma String
                                           //  e uma cadeia de carateres
    bool operator!();                // Testa se String nula?
    bool operator<(CString &);// Operador <, para comparação de Strings
    bool operator>(CString &);// Operador >, para comparação de Strings
  }; // fim da declaração da classe
```

Problema 7.8 – Fácil (adaptado do teste de frequência de 23/06/2015)

Considere a classe de objetos `CEsfera` para representar esferas.

```
class CEsfera{            // representa uma esfera de centro (x0,y0,z0) e raio r
  protected:
      double centro[3];
      double raio;
  public:
      CEsfera() { ... }   // construtor por omissão
      CEsfera(double x, double y, double z, double r) { ... } // por enumeração
      void mostra() { for (int i=0; i<3; i++) cout << centro[i] << ", ";
                  cout << raio; }
};
```

a) Declare uma classe derivada `CBola` para incluir o atributo `peso` (do tipo `double`).
b) Na classe derivada (re)defina o método `mostra()` para que apresente no ecrã todos os atributos, incluindo os herdados da classe base.
c) Defina um método que devolva o valor da densidade de uma bola (=peso/volume, volume=4/3 π r^3). Este método não deve escrever nada no ecrã.

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.9 – Médio (adaptado do teste de frequência de 06/05/2015)

Considere a classe de objetos `CTempo` para representar tempos.

```
class CTempo {
      int horas, minutos, segundos;
public:
      CTempo() { horas = minutos = segundos = 0; }
      CTempo(float t) { set(t); }
      void set(float t);  // ex: se t = 2.56, então horas=2, minutos=33, segundos=36
      float get(void) const; // ex: se horas=2, minutos=33, segundos=36, devolve 2.56
      void mostra(void) const { cout << setfill('0') << setw(2) << horas << ':' <<
                          setw(2) << minutos<<':'<< setw(2) << segundos << endl; }
      CTempo operator -(const CTempo& t2);
};
```

a) Defina o método constante `float CTempo::get(void) const`.
Sugestão: use a fórmula `t = horas + minutos/60.0 + segundos/3600.0`.
b) Defina o método `void CTempo::set(float t)` com base no algoritmo descrito a seguir em pseudo-código.

```
horas = parte inteira de t
t = (t – horas) * 60
minutos = parte inteira de t
segundos = arredonda ((t – minutos) * 60) para o inteiro mais próximo
```

c) Defina o método `CTempo CTempo::operator -(const CTempo& t2)`.
Sugestão: chame o método `get()` para ambos os objetos e use o método `set()` no novo objeto.

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 7.10 – Médio (adaptado do exame de recurso de 10/07/2015)

Considere a classe de objetos `CPiramide` para representar pirâmides regulares. Uma pirâmide é regular se a base for um polígono regular, ou seja um polígono cujos lados tenham todos o mesmo comprimento. Um apótema da base é um segmento de reta que liga o centro do polígono da base ao ponto médio de um dos seus lados.

```
class CPiramide{
  protected:
      int numLadosBase;          // número de lados da base regular
      double altura, lado;       // altura da pirâmide e comprimento dos lados da base
      double apotema;            // comprimento do apótema da base
  public:
      CPiramide() { numLadosBase = 3; altura = 1.0; lado = 1.0; setApotema(); }
      CPiramide(int n, double a, double l) { ... }
      void mostra()const{ cout << "Pirâmide regular de altura " << altura <<
                          " e base de lado " << lado << endl; }
      void setApotema() { apotema = lado / (2.0 * tan(3.14159265 / numLadosBase)); }
      double areaBase() const { return (0.5 * apotema * numLadosBase * lado); }
      virtual double volume() = 0;     // método para calcular o volume da pirâmide
};
```

a) Declare uma classe derivada `CPiramideHexagonal` para representar pirâmides retas regulares cuja base é um hexágono regular. Uma pirâmide é reta se a projeção do vértice da pirâmide na base coincide com o centro geométrico da base. Declare e implemente nesta classe derivada um construtor que aceita dois parâmetros: a altura da pirâmide e o comprimento dos lados da base. Este construtor inicializa todos os atributos do objeto.
b) Defina na classe derivada o método virtual `volume()`.
Nota: O volume da pirâmide é dado pela expressão volume = 1/3 x Área da Base x Altura.
Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.11 – Médio (adaptado do teste de frequência de 26/04/2017)

Considere uma aplicação informática que tem por objetivo gerir dados de pessoas. Nesta aplicação, existe uma classe de objetos `CPessoas` que permite definir um conjunto de pessoas. Possui os atributos (privados) `npessoas` (`int`) e uma tabela de nomes (`string`) e outra de datas de nascimento (`int`) em que a data é expressa no formato aaaammdd. As tabelas têm dimensão `MAX` (definida como constante inteira no início

do programa e igual a 100). Existem os métodos (públicos) seguintes: i) um construtor por defeito/omissão (tabelas vazias); ii) um construtor por cópia; iii) um método para inserir uma pessoa (dados o nome e a data de nascimento como parâmetros); iv) um método `ordena()` que ordena as pessoas por data de nascimento (altera ambas as tabelas).

a) Escreva a declaração da classe de objetos `CPessoas`. Nesta alínea, não deve implementar qq. método.
b) Implemente os construtores por defeito, por cópia e o método para inserir.
c) Implemente o método `ordena()`.

Considere agora que nessa aplicação informática foi criada uma classe derivada da anterior, `CAdultos`, que inclui mais um atributo (privado) inteiro (`nadultos`, o nº de pessoas com mais de 18 anos). Esta classe deve ter acesso a todos os atributos da classe base. Existem os métodos (públicos) da classe base e ainda os métodos `getNumAdultos()` (devolve o valor do atributo `nadultos`) e o método `atualiza()` que ordena as tabelas e atualiza o atributo `nadultos`, usando a data de hoje como referência (considere por exemplo que a data de hoje é 20200125 ou outra qualquer à sua escolha).

d) Re-escreva a declaração da classe de objetos `CPessoas` tendo em conta a nova classe derivada.
e) Escreva a declaração da classe de objetos `CAdultos`. Nesta alínea, não deve implementar qq. método.
f) Implemente o método `atualiza()`. Deve ter em conta que as tabelas são primeiro ordenadas (chamando o método herdado `ordena()`) antes de contar o nº de adultos (i.e., não deve pesquisar toda a tabela).

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.12 – Médio (adaptado do exame de recurso de 03/07/2017)

A classe de objetos `CAngulo` permite definir um ângulo em graus, minutos e segundos. Possui os atributos privados `graus`, `minutos` e `segundos` (todos do tipo `int`) e os métodos seguintes: i) um construtor por defeito (objeto com 10º5'15''); ii) um construtor por enumeração (recebe 3 inteiros, correspondentes aos graus, minutos e segundos); iii) um construtor por cópia; iv) a sobrecarga do operador `<`.

a) Escreva a declaração da classe de objetos `CAngulo`. Nesta alínea, não deve implementar qualquer método.
b) Implemente os construtores por defeito, por enumeração e por cópia da classe `CAngulo`.
c) Implemente a sobrecarga do operador `<`.

Considere agora que foi criada uma classe derivada da classe `CAngulo`, `CSetor`, que inclui mais um atributo privado inteiro: `raio`, o raio do setor circular. A classe derivada deve ter acesso a todos os atributos da classe base. Existem os métodos públicos da classe mãe e ainda o método `menorTamanhoQue()` – um setor circular é menor que outro se simultaneamente tiver menor ângulo e menor raio.

d) Re-escreva a declaração da classe de objetos `CAngulo` tendo em conta a nova classe derivada.
e) Escreva a declaração da classe de objetos `CSetor`. Nesta alínea, não deve implementar qualquer método.
f) Implemente o método `menorTamanhoQue()`. Deve chamar o operador herdado `<`.

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.13 – Médio (saiu no teste de frequência de 16/05/2018)

A classe de objetos `ArrayDinamico` permite manipular uma tabela alocada dinamicamente na *heap*, do tipo `float`, de tamanho máximo igual a `tamanho` (tipo `int`), que é um múltiplo de 100 que pode ser aumentado dinamicamente se houver necessidade de inserir mais elementos do que o tamanho inicialmente previsto. Para além do atributo `tamanho`, a classe possui o atributo `array` (ponteiro para `float`) que guarda o endereço do 1.º elemento da tabela e o atributo `n` (tipo `int`) que guarda o número de elementos realmente ocupados na tabela (é sempre ≤ que tamanho). Possui pelo menos os métodos seguintes: i) construtor por defeito/omissão (tabela com `tamanho` = 100 e `n` = 0); ii) construtor que aceita um inteiro `tam` (tabela com tamanho igual ao menor múltiplo de 100 maior ou igual a tam); iii) destrutor; iv) operador `<<` que insere um novo número na posição `n` da tabela e a seguir incrementa `n` – se antes da inserção tiver sido atingido o tamanho máximo (i.e. `n==tamanho`), a tabela é redimensionada para aumentar em 100 elementos `tamanho` (aloca novo bloco de memória com `tamanho=tamanho+100` elementos, copia para o novo bloco a informação atual, atualiza o ponteiro `array` e liberta da *heap* o bloco anteriormente utilizado); v) operador `+` que adiciona dois objetos da classe, realizando a adição elemento a elemento das tabelas, sem alterar as tabelas, e devolve um objeto da classe com o resultado, de tamanho igual ao menor dos tamanhos das duas tabelas a adicionar (sendo `n` também igual ao menor `n` das duas tabelas).

a) Escreva a declaração da classe de objetos `ArrayDinamico`. Nesta alínea, não deve definir/implementar qualquer método *inline*; os métodos devem ser apenas declarados nesta alínea.
b) Defina/implemente o operador `<<`. O operador deve permitir executar a operação em cascata, ex. `a << 1.5 << 2.5`, em que a é um objeto da classe `ArrayDinamico`.
c) Defina/implemente o operador `+` da classe `ArrayDinamico`, bem como quaisquer outros métodos da classe que considere necessários para o operador poder executar corretamente.

Implemente uma função `main()` para ir testando a implementação da classe.

ESTRUTURAS DE DADOS E ALGORITMOS

Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 8
LISTAS LIGADAS

8.1. Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de lista ligada;
- Perceber as vantagens da utilização das listas ligadas em relação às tabelas;
- Saber manipular de forma adequada o conceito de lista ligada de forma a implementar/alterar funções que manipulam essas listas ligadas.

8.2. Introdução às listas ligadas

As listas ligadas (*linked list*) são utilizadas para armazenar conjuntos de dados do mesmo tipo. Como vimos anteriormente, podemos também utilizar tabelas para esse efeito. Contudo, uma tabela é uma estrutura sequencial constituída por elementos que ocupam zonas de memória contíguas. O tamanho máximo de uma tabela é fixo, o que implica que mesmo quando vazias estas podem ocupar um grande espaço na memória. As inserções ou remoções de novos valores no meio de uma tabela ordenada implicam normalmente a movimentação de um grande número dos seus elementos.

Quando o número de elementos a armazenar não é conhecido *a priori*, ou quando os elementos necessitam de ser inseridos ou eliminados com frequência, é preferível utilizar listas ligadas. A razão para isso prende-se com o facto de cada elemento de uma lista ligada poder ser armazenado em qualquer lugar da memória sem que exista nenhum tipo de restrição entre elementos consecutivos.

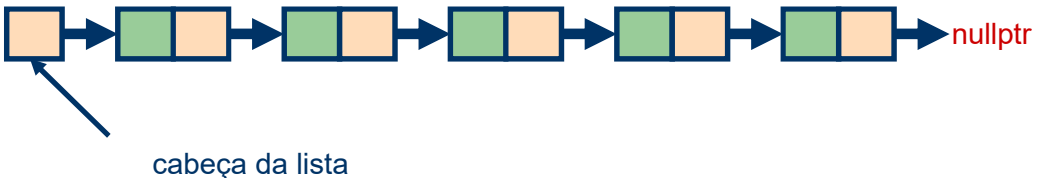
Cada elemento numa lista ligada é designado por **nó**. Numa lista ligada simples cada nó armazena dois tipos de informação:

- O **Dado** a armazenar. Este pode ser de qualquer tipo (*standard* ou definido pelo utilizador).
- O **Ponteiro** (ou ligação) para o próximo nó.

Podemos definir a seguinte classe `CNoLista` para representar um nó de uma lista ligada de inteiros:

```
class CNoLista{
  public:
       int dados;
       CNoLista *proximo;
};
```

Esta classe possui o atributo `dados` que representa a informação a armazenar (neste caso um inteiro) e o atributo `proximo` do tipo ponteiro que permite aceder ao próximo elemento na lista. Uma lista ligada é consequentemente definida como um conjunto de vários nós ligados entre si.



Para definir a lista, apenas necessitamos de armazenar um ponteiro para o primeiro nó da lista. Este ponteiro é normalmente designado por **cabeça da lista**. Como se pode observar, o último nó da lista contém, no atributo `proximo`, um ponteiro vazio (`nullptr`) para indicar o fim da lista.

Podemos representar uma **lista** de inteiros através da definição da seguinte **classe**:

```
class CListaInteiros{
      CNoLista *cabeca;   // ponteiro para 1º nó da lista
  public:
      CListaInteiros(void);
      ~CListaInteiros(void);

      void insereItem(int);
      void insereItemOrdenado(int);
      void apagaItem(int);
      bool procuraItem(int) const;
      void escreveLista(void) const;
      ...
};
```

A classe `CListaInteiros` apenas armazena, como já foi referido, um ponteiro para o primeiro elemento da lista (**atributo** `cabeca`). Para além disso, são definidos alguns **métodos** que permitem manipular a lista. O primeiro dos métodos indicados é o **construtor** que apenas inicializa o membro `cabeca` com o valor `nullptr`, indicando que a lista está vazia inicialmente.

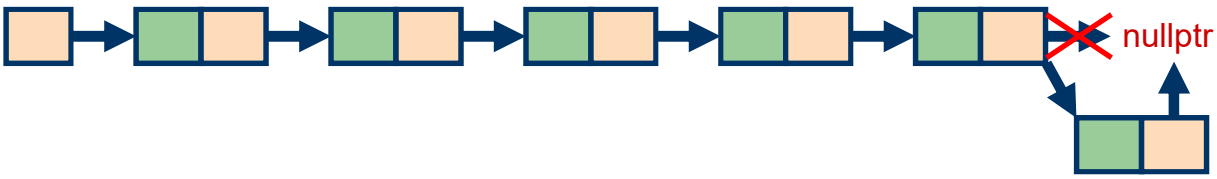
```
CListaInteiros :: CListaInteiros() {
      cabeca = nullptr;
}
```

É importante também definir um **destrutor**, uma vez que antes de se destruir o objeto `CListaInteiros` é necessário libertar a memória associada a cada nó. Para tal, percorre-se a lista utilizando dois ponteiros auxiliares que apontam, em cada instante, para o elemento atual e para o próximo elemento. Desta forma, é possível destruir o elemento atual sem perder uma ligação aos elementos seguintes da lista (apontados por `proximo`).

```
CListaInteiros :: ~CListaInteiros() {
	CNoLista *atual, *proximo;

	atual = cabeca;
	while (atual != nullptr){
		proximo = atual->proximo;
		delete atual;
		atual = proximo;
	}
}
```

Para **inserir** um novo elemento na lista, temos duas possibilidades: ou assumimos que a lista não está ordenada e inserimos o novo elemento no final da lista (ou no início), ou então assumimos que a lista está previamente ordenada e nesse caso introduz-se o novo elemento na sua ordem correta.

Comecemos pelo primeiro caso. A figura a seguir representa a inserção de um novo elemento no final da lista. Em primeiro lugar, é necessário percorrer a lista até se chegar ao último elemento. Depois basta atribuir ao ponteiro do último nó (que tem o valor `nullptr`) o valor do ponteiro para o novo nó.



O algoritmo para inserir um novo elemento no final da lista pode ser descrito da seguinte forma, em pseudocódigo:

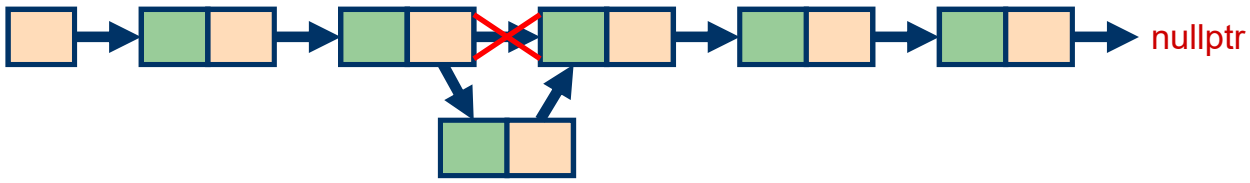
__1__. Criar o novo nó.
__2__. Guardar os dados no novo nó, inicializando o ponteiro «proximo» a nullptr.
__3__. Se a lista estiver vazia
Colocar a cabeça da lista a apontar para o novo nó.
Caso contrário
Percorrer a lista até se atingir o último nó.
Colocar o último nó a apontar para o novo nó.
Fim Se.

A implementação do método `void insereItem(int)`, utilizado para inserir um novo inteiro no fim da lista, é a seguinte:

```
void CListaInteiros :: insereItem(int item) {  // insere no final da lista
	CNoLista *novo=new CNoLista;
	CNoLista *atual;

	novo->dados = item;
	novo->proximo = nullptr;

	if (cabeca == nullptr) cabeca = novo;
	else{
		atual = cabeca;
		while (atual->proximo != nullptr) atual = atual->proximo;
		atual->proximo = novo;
	}
}
```

Dado que a inserção no final da lista é pouco eficiente com listas longas, é preferível, nesse caso, inserir à cabeça, se isso for possível no problema concreto a resolver.

Para inserir um novo elemento respeitando o critério de ordenação previamente definido numa lista ordenada, podemos implementar o método ilustrado na figura seguinte.



Neste caso, e após se ter encontrado a posição correta para a inserção, é necessário alterar os ponteiros do nó anterior para passar a apontar para o novo nó e também o ponteiro do novo nó para passar a apontar para o nó seguinte. Esta regra é alterada se a posição de inserção for no primeiro nó da lista. Neste caso, atualiza-se o ponteiro para a cabeça da lista com o ponteiro para o novo elemento e faz-se o novo elemento apontar para o antigo primeiro nó.

A implementação do algoritmo de inserção ordenada é a seguinte:

```
void CListaInteiros :: insereItemOrdenado(int item) {
	CNoLista *novo = new CNoLista; // aloca espaço para o novo nó
	CNoLista *atual, *anterior;

	novo->dados = item;	// inicializa os dados do novo nó
	novo->proximo = nullptr;

	if (cabeca == nullptr) {  // se lista vazia então este é o primeiro nó
		cabeca = novo;
		return;
	}			// continua na página seguinte...
```

```
	// continuação da página anterior...

	atual = cabeca;
	anterior = nullptr;
	while((atual != nullptr) && (atual->dados<item)) {   // procura a posição de
		anterior = atual;                                // inserção
		atual = atual->proximo;
	}
	if (anterior == nullptr)  // se o ponteiro para o elemento anterior
		cabeca = novo;      // for nullptr então insere no princípio
	else
		anterior->proximo=novo; // caso contrário insere entre dois nós

	novo->proximo = atual; // em qualquer dos casos atualiza o ponteiro
}                              // para o próximo elemento no novo nó.
```

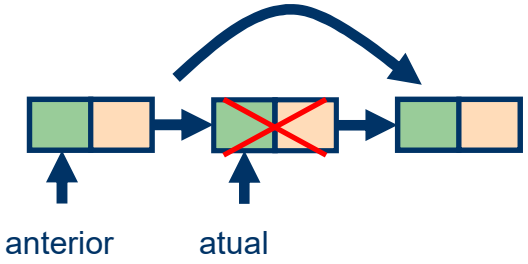
O método seguinte permite a visualização no ecrã dos valores armazenados na lista. Para tal, é necessário percorrer a lista desde o primeiro nó até ao último. Este processo designa-se normalmente por **travessia** da lista (*list traversal*).

```
void CListaInteiros :: escreveLista(void) const {
	CNoLista *atual = cabeca;

	if (cabeca == nullptr) cout << "Lista Vazia..." << endl;
	else {
		while (atual != nullptr) {
			cout << atual->dados << ", "; // tarefa a fazer em cada nó
			atual = atual->proximo;
		}
		cout << "FIM" << endl;
	}
}
```

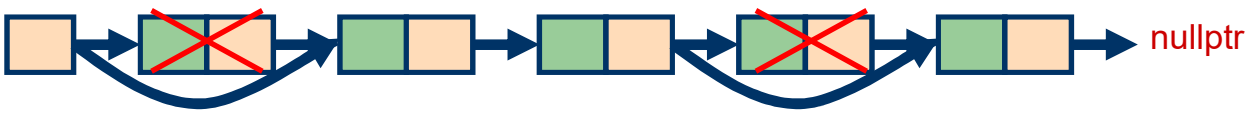
Um outro método, semelhante ao anterior, permite **procurar** um determinado item na lista. Este método devolve um valor lógico que indica se o elemento existe ou não na lista.

```
bool CListaInteiros :: procuraItem(int Item) const {
	CNoLista *atual = cabeca;

	if (cabeca == nullptr) return false;

	while (atual != nullptr) {
		if (atual->dados == item) return true;
		atual = atual->proximo;
	}
	return false;
}
```

O último método da classe `CListaInteiros` descrito aqui como exemplo permite a **eliminação** de um dado item da lista. Para tal é necessário, em primeiro lugar, encontrar o nó que contém o elemento a eliminar. Na procura, utiliza-se mais uma vez dois ponteiros, `atual` e `anterior`, que apontam respetivamente para o elemento atual e para o elemento que o antecede na lista.



Consoante a localização do nó a eliminar, é apenas necessário atualizar o valor do ponteiro guardado no elemento anterior, colocando-o a apontar para o nó posicionado a seguir ao elemento a eliminar. Tal como no caso da inserção ordenada, também aqui é necessário fazer a distinção entre a eliminação de um nó no meio da lista e a eliminação do primeiro nó da lista. A figura seguinte representa de forma esquemática o processo de eliminação de um elemento de uma lista ligada:



O algoritmo pode ser descrito da seguinte forma em pseudocódigo:

__1__. Se a lista não estiver vazia.
    __2__. Encontrar o nó pretendido.
    __3__. Se o elemento a eliminar é o primeiro elemento da lista:
        Colocar a cabeça da lista a apontar para o nó a seguir ao nó a eliminar.
    Caso contrário:
        Colocar o elemento anterior ao nó a eliminar a apontar para o elemento a seguir.
    Fim Se.
    __4__. Libertar a memória associada ao elemento a eliminar.
Fim Se.

A implementação deste método é apresentada a seguir:

```
void CListaInteiros :: apagaItem(int item) {

	CNoLista *atual = cabeca;
	CNoLista *anterior;

	if (cabeca == nullptr) return;   // desnecessário

	anterior = nullptr;
	while (atual != nullptr) {
		if (atual->dados == item) {  // elimina nó
			if (anterior == nullptr)
				cabeca = atual->proximo;
			else
				anterior->proximo = atual->proximo;
			delete atual;
			return;
		}
		anterior = atual;
		atual = atual->proximo;
	}
}
```

8.3. Exercícios sugeridos

Com o conjunto de exercícios apresentados a seguir pretende-se que o aluno consolide os seus conhecimentos sobre listas ligadas. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas. Todos os exercícios assumem a definição da classe `CListaInteiros` apresentada anteriormente. O código fonte da declaração desta classe e da definição de alguns dos seus métodos foi disponibilizado no InforEstudante.

Problema 8.1 – Fácil
Defina o método `int numeroElementosPares(void)` que devolve o número de elementos pares contidos na lista de números inteiros.

Problema 8.2 – Fácil
Defina o método `maioresQue(int numero)` que devolve o número de elementos contidos na lista de números inteiros que sejam maiores do que o número inteiro passado como parâmetro.

Problema 8.3 – Fácil
Considere que a lista ligada está ordenada. Defina o método `existeNumero(int numero)` que verifica se o número inteiro passado por parâmetro existe ou não na lista de números inteiros.

Problema 8.4 – Fácil
Defina o método `int ultimoElemento(void)` que devolve o último elemento contido na lista de números inteiros. No caso de a lista estar vazia, este método deve devolver -1.

Problema 8.5 – Médio
Defina a sobrecarga do operador `[ ]` de modo a devolver o elemento cujo nó ocupa na lista de números inteiros uma dada posição dada pelo parâmetro do operador. O índice zero corresponde ao primeiro elemento da lista. Caso o valor correspondente ao índice não exista na tabela, o operador deve devolver −1.

Problema 8.6 – Médio
Defina o método `void eliminaImpares( )` que elimina todos os números ímpares da lista de números inteiros.

Problema 8.7 – Médio
Defina um novo construtor `CListaInteiros(char *str)` que cria uma lista de inteiros a partir do código ASCII dos carateres que compõem uma *string* passada como parâmetro. Exemplo: se a *string* passada como parâmetro for "ABETO" o construtor deve criar a lista: 65→66→69→84→79.

Problema 8.8 – Médio
Defina a sobrecarga do operador == de forma a permitir que esse operador possa ser utilizado para verificar se duas listas de números inteiros são iguais. Considere que duas listas são iguais quando ambas contêm o mesmo número de elementos.

Problema 8.9 – Difícil
Defina a sobrecarga do operador += para que este permita adicionar à primeira lista de números inteiros os números contidos na segunda, desde que estes ainda não existam na primeira lista.
A segunda lista não deve ser modificada. No caso de a segunda lista estar vazia, o operador não deve fazer nada.

Problema 8.10 – Difícil
Defina o método `void inverteLista(void)` que permite inverter a ordem dos membros da lista de números inteiros sem criar novos nós ou apagar os já existentes.

Problema 8.11 – Difícil
Defina a sobrecarga do operador + para devolver uma lista de números inteiros que representa a fusão de duas listas numa só. No caso de ambas as listas terem itens com o mesmo valor, apenas deve ser inserido um deles na lista devolvida como resultado.

Problema 8.12 – Difícil

Defina a sobrecarga do operador `-=` para eliminar da primeira lista de números inteiros todos os elementos que também pertençam à segunda lista.

Problema 8.13 – Médio (saiu no teste de frequência de 04/05/2016)

Considere uma lista ligada `CListaOcorrencias` que permite armazenar o número de ocorrências de cada palavra num dado livro.

```
class CNoLista {
   public:
      char Palavra[25];
      unsigned int Contador;
      CNoLista *Proximo;
};
```

```
class CListaOcorrencias {
      CNoLista *PrimeiraPalavra;
   public:
      CListaOcorrencias (void);
      ~CListaOcorrencias (void);
        ...
      CListaOcorrencias & operator += (char *NovaPalavra);
};
```

a) Implemente um novo método `porOcorrencia()` que permita devolver a palavra contida na lista com o maior número de ocorrências, bem como esse número de ocorrências. No caso de haver mais do que uma palavra com o mesmo número de ocorrências, deverá apenas devolver uma delas.

b) Implemente nesta classe a sobrecarga do operador `+=` de forma a permitir adicionar uma nova palavra à lista de ocorrências (ver protótipo). Se a palavra já existe na lista, incrementa-se apenas o contador. Se a palavra ainda não existe, acrescenta-se a nova palavra no fim da lista e coloca-se o respetivo contador a um.

Problema 8.14 – Médio (saiu no exame de recurso de 05/07/2016)

Pretende-se implementar uma aplicação informática para gerir o sistema de apoio a clientes numa determinada empresa. Para isso decidiu-se criar uma lista ligada denominada `CListaHelpdesk` que armazena a informação relativa a todas as perguntas mais frequentes que são feitas pelos clientes. Cada elemento da lista, `CNoLista`, armazena pelo menos a seguinte informação: código da pergunta (5 carateres), resposta à pergunta (300 caracteres) e o número de vezes que a pergunta foi pesquisada (inteiro).

a) Declare as classes `CListaHelpdesk` e `CNoLista` que permitem armazenar a lista das perguntas mais frequentes.
b) Para diminuir o tempo gasto em localizar uma determinada pergunta na lista, de cada vez que é feito um acesso a um nó (para consulta ou alteração de um dos seus campos de informação), este nó é movido para o início da lista. Deste modo, os nós acedidos com maior frequência ficarão localizados no início da lista. Implemente um método `pesquisaLista()` que permita procurar uma pergunta na lista (dado o seu código) e devolver a resposta associada a essa pergunta. Pretende-se também que no caso de a pesquisa ter tido sucesso (o código foi encontrado), se mova o nó correspondente para o início da lista e se incremente o número de vezes que a pergunta foi pesquisada.

Problema 8.15 – Difícil (adaptado do teste de frequência de 16/06/2017)

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo construtor que aceite como parâmetros uma tabela de inteiros e a sua dimensão, e que construa uma lista com os números contidos na tabela. O primeiro número da tabela deve ficar na cabeça da lista e o último na cauda. O método não pode chamar outros métodos da classe.

b) Defina um novo método da classe que realize a sobrecarga do operador `<<`. O operador "roda" os nós armazenados na lista `n` (tipo `int`) posições, no sentido da cauda para a cabeça da lista. Por exemplo, se chamar o método para a lista `7, 5, 9, 0, 6, 10` e com `n=2`, a lista fica `9, 0, 6, 10, 7, 5`. O método não faz nada se `n` for menor que 1 ou maior ou igual que o número de nós da lista. O método não pode chamar outros métodos da classe.
Sugestão: Antes de começar a escrever o código, desenhe um esquema para perceber bem a estrutura do problema e desenhar com sucesso um algoritmo para o resolver.

Problema 8.16 – Médio (adaptado do exame de recurso de 03/07/2017)

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo construtor que aceita como parâmetros dois inteiros e constrói uma lista com todos os números sequenciais crescentes entre esses dois inteiros (inclusive). O menor valor dos dois parâmetros deve ser guardado no elemento situado na cabeça da lista e o maior na cauda. O método não pode chamar outros métodos da classe.

b) Defina o método constante `bool CListaInteiros::temNumeroImpar(void) const` que verifica se a lista tem um número par de elementos.

c) Defina o método `CListaInteiros& CListaInteiros::operator *= (const CListaInteiros &outra)` que multiplica o 1.º elemento da lista pelo 1.º elemento da lista `outra`, o 2.º elemento da lista pelo 2.º elemento da lista `outra`, e assim sucessivamente. Se a lista estiver vazia, o método não faz nada. Se a lista tiver mais elementos que a lista `outra`, os elementos que não tiverem correspondência com a lista `outra` devem permanecer inalterados.

Problema 8.17 – Médio (saiu no teste de frequência de 16/05/2018)

Considere a classe `CListaInteiros` para representar listas ligadas que armazenam números inteiros, que é objeto de estudo nesta ficha das aulas práticas.

a) Defina o método constante `int CListaInteiros::getEnesimoItem(int)` que devolve o enésimo elemento da lista. Se a lista estiver vazia ou o enésimo elemento não existir, o método deve devolver `INT_MIN`. Por exemplo, `getEnesimoItem(23)` devolve o 23º elemento da lista (se este existir).

b) Defina o método `CListaInteiros& CListaInteiros::operator--(void)` que decrementa cada elemento da lista.

ESTRUTURAS DE DADOS E ALGORITMOS

Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 9
PILHAS E FILAS

9.1. Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de pilha e o conceito de fila;
- Conhecer a sua utilidade em programação;
- Saber manipular de forma adequada as pilhas e as filas, podendo facilmente implementar/alterar funções que manipulam estes tipos particulares de listas ligadas.

9.2. Pilhas

Uma das estruturas ligadas de dados mais simples de manipular é a pilha. É uma das estruturas de dados mais utilizadas em programação, sendo inclusive usada na maioria das arquiteturas modernas de computadores. O princípio fundamental de uma pilha é que qualquer acesso aos seus elementos é feito através do elemento que está guardado no topo da pilha. Quando um elemento novo é inserido na pilha, passa a ser o elemento do topo e passa a ser também o único elemento que pode ser removido de imediato da pilha. Isto faz com que os elementos da pilha sejam retirados pela ordem inversa daquela com que foram inseridos: "o primeiro que sai é o último que entrou"; a sigla LIFO – "*last in, first out*" – é usada para descrever este tipo de acesso à estrutura de dados.

Para se entender o funcionamento de uma pilha, pode-se fazer uma analogia com uma pilha de pratos: ao adicionar-se um prato à pilha, este é colocado no topo; ao retirar-se um prato da pilha, é retirado o prato situado no topo da pilha. Uma estrutura de dados do tipo pilha funciona de maneira análoga.

Existem duas operações básicas para manipular uma estrutura de dados do tipo pilha: a operação para inserir um novo elemento no topo da pilha e a operação para retirar um elemento do topo da pilha. É comum designar estas duas operações através dos termos em inglês "*push*" (colocar na pilha) e "*pop*" (retirar da pilha).

Pode-se implementar uma pilha de números inteiros através das seguintes classes de objetos:

```
class CPilhaInteiros;

class CNoPilha{
    int dados;
    CNoPilha *proximo;
    friend class CPilhaInteiros; // esta classe acede aos membros private
};
```

```
class CPilhaInteiros{
    CNoPilha *topo;  // ponteiro para o elemento no topo da pilha
  public:
    CPilhaInteiros(void);
    ~CPilhaInteiros(void);

    void push(const int item);
    bool pop(int &item);
    void escrevePilha(void) const;
    bool pilhaVazia() const { return (topo==nullptr);}
};
```

A classe `CPilhaInteiros` tem um único atributo que armazena um ponteiro para o elemento que está no topo da pilha. São definidos alguns métodos que permitem manipular a pilha.

O primeiro dos métodos indicados é o construtor da classe que apenas inicializa o atributo `topo` com o valor `nullptr`, indicando assim que a pilha está inicialmente vazia.

```
CPilhaInteiros::CPilhaInteiros() {
      topo = nullptr;
}
```

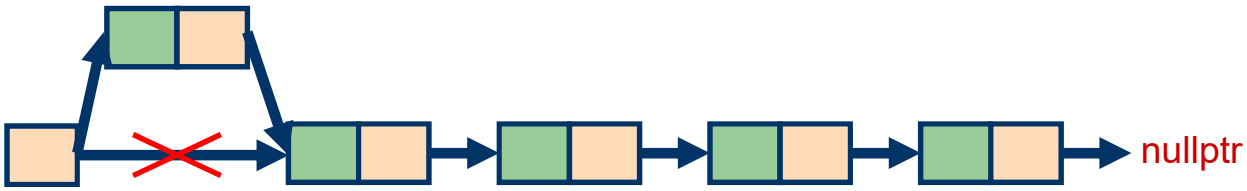
É importante também definir um destrutor porque é necessário libertar a memória associada a cada um dos elementos (nós) da pilha, antes de se destruir o objeto `CPilhaInteiros`. Para tal, percorre-se a pilha utilizando um ponteiro auxiliar que aponta para o elemento a seguir àquele que se pretende apagar na iteração atual. Desta forma, é possível destruir o elemento atual sem perder uma ligação aos elementos seguintes na pilha (apontados por `proximo`).

```
CPilhaInteiros::~CPilhaInteiros() {
      CNoPilha *seguinte;

      while (topo != nullptr) {
            seguinte = topo->proximo;
            delete topo;
            topo = seguinte;
      }
}
```

A definição do método `void push(const int)`, utilizado para inserir um novo inteiro no topo da pilha, é a seguinte:

```
void CPilhaInteiros::push(const int item) {
	CNoPilha *novo = new CNoPilha;

	novo->dados = item;
	novo->proximo = topo;

	topo = novo;
}
```

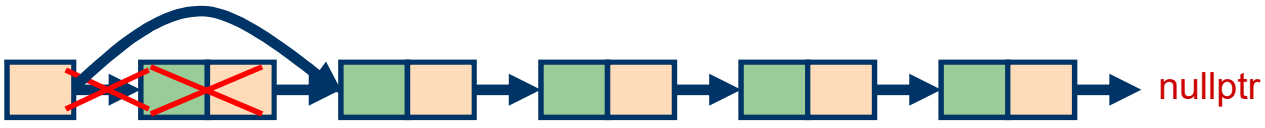
A figura a seguir apresenta de forma esquemática o método utilizado para inserir um novo elemento no topo da pilha:



A definição do método `bool pop(int&)`, utilizado para retirar um elemento do topo da pilha, é a seguinte:

```
bool CPilhaInteiros::pop(int &item) {
	if (topo == nullptr) return false;	// a pilha está vazia!

	CNoPilha *aux = topo;
	topo = topo->proximo;			// atualiza o topo da pilha
	item = aux->dados;
	delete aux;
	return true;
}
```

A figura a seguir apresenta de forma esquemática o método utilizado para retirar um dado elemento do topo da pilha:



Finalmente, apresenta-se um método que permite escrever no ecrã os valores armazenados na pilha de inteiros:

```
void CPilhaInteiros::escrevePilha(void) const {
      if (topo == nullptr)
          cout << "Pilha Vazia... " << endl;
      else {
          CNoPilha *aux = Topo;

          while(aux != nullptr) {
              cout << aux->dados << endl;
              aux = aux->proximo;
          }
      }
}
```

9.3. Filas

Outra estrutura de dados que é muito utilizada em computação é a fila. Numa fila, o acesso aos seus elementos segue uma regra diferente: enquanto na pilha "o último que entra é o primeiro que sai", na fila "o primeiro que entra é o primeiro que sai"; a sigla FIFO – "*first in, first out*" – é usada para descrever esta estratégia. A ideia fundamental da fila é que só se pode inserir novos elementos no final da fila e retirar elementos do início da fila.

A estrutura de uma fila é uma analogia natural com o conceito de fila que usamos no nosso dia-a-dia: o primeiro a entrar numa fila é o primeiro a ser atendido (a sair da fila). Um exemplo de utilização em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar estratégias para determinar que documento será impresso em primeiro lugar. A estratégia mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem pela qual foram submetidos – o primeiro a ser submetido é o primeiro a ser impresso.

Para implementar uma fila, é necessário inserir novos elementos numa das extremidades, o fim, e retirar elementos da outra extremidade, o início. Para tal, é necessário definir dois ponteiros para referenciar cada um destes dois elementos.

Pode-se implementar uma fila de inteiros através da seguinte definição de classe:

```
class CFilaInteiros;

class CNoFila{
      int dados;
      CNoFila *proximo;

      friend class CFilaInteiros; // esta classe acede aos membros private
};
```

```
class CFilaInteiros {
    CNoFila *inicio; // ponteiro para o início da Fila
    CNoFila *fim;    // ponteiro para o fim da Fila
  public:
    CFilaInteiros(void);
    ~CFilaInteiros(void);

    void insereNaFila(const int item);
    bool retiraDaFila(int &item);
    void escreveFila(void) const;
    bool filaVazia(void) const { return (inicio == nullptr); }
};
```

A classe `CFilaInteiros` tem como atributos dois ponteiros. O primeiro serve para referenciar o elemento que está no início da fila e o segundo o último elemento da fila. A classe define ainda alguns métodos que permitem manipular a fila. O primeiro dos métodos indicados é o construtor que apenas inicializa com o valor `nullptr` os dois ponteiros para as extremidades da fila, indicando assim que a fila está inicialmente vazia.

```
CFilaInteiros::CFilaInteiros() {
      inicio = nullptr;
      fim = nullptr;
}
```

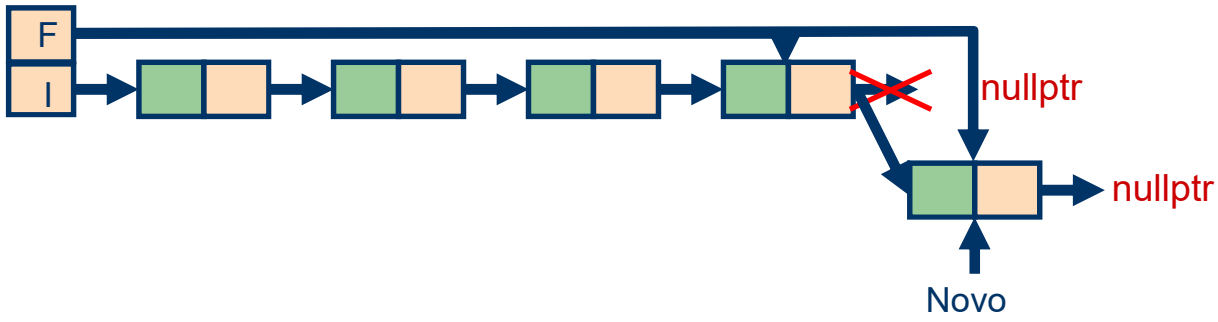
Mais uma vez, é importante libertar a memória associada a cada um dos nós que constituem a fila quando esta é destruída. Para tal, define-se um destrutor que percorre a fila utilizando um ponteiro auxiliar que aponta para o próximo elemento àquele que se pretende apagar na iteração atual. Desta forma, é possível destruir o elemento atual sem perder uma ligação aos elementos seguintes na fila.

```
CFilaInteiros::~CFilaInteiros (){
      CNoFila *seguinte;

      while (inicio != nullptr){
            seguinte = inicio->proximo;
            delete inicio;
            inicio = seguinte;
      }
      fim = nullptr;
}
```

A definição do método `void InsereNaFila(const int)`, utilizado para inserir um novo elemento no fim da fila, é a seguinte:

```
void CFilaInteiros::insereNaFila(const int item) {

	CNoFila *novo = new CNoFila;

	novo->dados = item;
	novo->proximo = nullptr;

	if (inicio == nullptr) inicio = novo;
	else fim->proximo = novo;
	fim = novo;
}
```

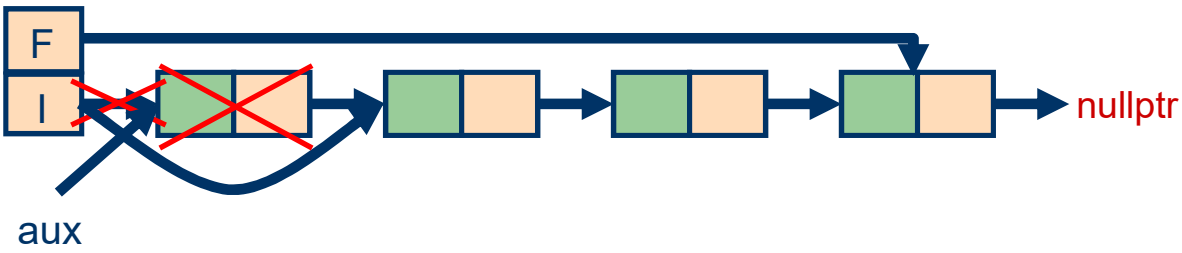
A figura a seguir apresenta de forma esquemática o método utilizado para inserir um novo elemento no fim da fila:



A definição do método `bool retiraDaFila(int&)` é a seguinte:

```
bool CFilaInteiros::retiraDaFila(int &item) {
	if (inicio==nullptr)  return false;

	item = inicio->dados;

	CNoFila *aux = inicio;		// ponteiro auxiliar para nó a eliminar

	inicio = inicio->proximo;	// atualiza ponteiro inicio da fila
	if (inicio == nullptr) fim = nullptr;  // se a fila ficar vazia
	delete aux;
	return true;
}
```

A figura a seguir apresenta de forma esquemática o método utilizado para retirar o elemento no início da fila:



Finalmente, apresenta-se um último método que permite escrever no ecrã os valores armazenados na fila de inteiros:

```
void CFilaInteiros::escreveFila(void) const {
	if (inicio == nullptr) {
		cout << “Fila vazia...” << endl;
		return;
	}

	for (CNoFila *aux=inicio; aux!=nullptr; aux=aux->proximo) {
		cout << aux->dados << endl;
	}
}
```

9.4. Exercícios sugeridos

Com o conjunto de exercícios apresentados a seguir, pretende-se que o aluno consolide os seus conhecimentos relativos às pilhas e filas. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas. Os problemas assumem a definição das classes `CPilhaInteiros` e `CFilaInteiros` apresentadas anteriormente. O código fonte da declaração destas classes e da definição de alguns dos seus métodos foi disponibilizado no InforEstudante juntamente com este enunciado.

Problema 9.1 – Fácil
Implemente uma função que, utilizando uma pilha, inverta uma *string* passada por parâmetro à função.

Problema 9.2 – Fácil
a) Defina um método para substituir o último elemento de uma fila de números inteiros pelo maior de todos os seus elementos. Note que o método pretendido só altera o valor do último elemento da fila, não alterando mais nenhum elemento.
b) Defina um método que troca o último elemento de uma fila de números inteiros com o maior de todos os seus elementos. Só precisa de trocar os dados, não sendo necessário mover os nós da fila.

Problema 9.3 – Fácil
Defina os seguintes métodos na classe `CPilhaInteiros`:

- `int numElementos(void)` – devolve o número de elementos contidos na pilha.
- `void trocaTopo(int valor)` – substitui o valor no topo da pilha pelo valor passado como parâmetro.

Problema 9.4 – Médio
Defina o método `bool CFilaInteiros::passaAfrente(int N)` que passa o último elemento na fila para a posição `N` a contar do início da fila. Se o valor de `N` é superior ao número de elementos contidos na fila o método devolve `false`, caso contrário devolve `true`.

Problema 9.5 – Médio
Defina um novo construtor `CPilhaInteiros(const CFilaInteiros &fila)` que constrói uma pilha de números inteiros a partir dos números inteiros contidos na fila passada como parâmetro.

Problema 9.6 – Médio
Implemente uma função booleana que, com o auxílio de uma estrutura de dados do tipo pilha, faça a análise de expressões matemáticas com o objetivo de verificar o balanceamento dos seus parêntesis retos e curvos. A função deve detetar erros de falta de parêntesis, como por exemplo, ***((a+b)***, e erros de parêntesis trocados, como por exemplo, **([a+b)]**.

Problema 9.7 – Médio
Defina um novo método `void CFilaInteiros::compactaFila(int a, int b)` que remova da fila todos os valores `v` não contidos num determinado intervalo de valores `a ≤ v ≤ b`. Este intervalo é passado como parâmetro. Por exemplo, se a fila for inicialmente **4→7→3→2→5→6→8→1→nullptr** e o método for chamado com os parâmetros `compactaFila(5,8)`, então, após a execução deste método, a fila passará a ser: **7→5→6→8→nullptr**.
Obs.: pode resolver este problema chamando os métodos `retiraDaFila()` e `insereNaFila()`. Alternativamente, pode tratar a fila como uma lista ligada e manipular diretamente os seus nós.

Problema 9.8 – Médio
Defina novo método para inverter a ordem dos `N` primeiros elementos de uma fila de números inteiros. Por exemplo, se a fila contiver os valores **4 → 7 → 8 → 11 → 15 → nullptr** e `N` tiver o valor **3**, então, como resultado da execução do método, a fila deverá ficar igual a **8 → 7 → 4 → 11 → 15 → nullptr**
Obs.: pode resolver este problema chamando os métodos `retiraDaFila()`, `insereNaFila()` e utilizar uma pilha auxiliar para inverter os elementos. Alternativamente, pode tratar a fila como uma lista ligada e manipular diretamente os seus nós.

Problema 9.9 – Médio
Adicione e defina métodos na classe `CPilhaInteiros` para sobrecarregar os operadores de comparação `<, >, <=, >=` que comparam duas pilhas em termos do número de elementos nelas contidos. Isto é, uma pilha é maior do que outra se o número de elementos que tiver for superior ao da outra pilha. Tenha em atenção a declaração *standard* de cada operador.

Problema 9.10 – Difícil
Defina o método `CFilaInteiros::operator+` (operador adição) que funde duas filas numa só. A nova fila a devolver deverá ser constituída alternadamente por elementos de cada uma das filas de origem. Tenha em atenção a declaração *standard* do operador.

Problema 9.11 – Difícil
Defina um método na classe `CPilhaInteiros` para reordenar os elementos da pilha de forma que no topo fiquem os elementos pares e no fim da pilha os elementos ímpares. A ordem relativa dos números pares e ímpares deve permanecer a mesma. Considere o seguinte exemplo:
Se a pilha contiver **3 → 5 → 4 → 17 → 6 → 83 → 1 → 84 → 16 → 37 → nullptr**, então, após a execução do método pretendido, a pilha passará a ser: **4 → 6 → 84 → 16 → 3 → 5 → 17 → 83 → 1 → 37 → nullptr**.

Problema 9.12 – Médio
Defina o método `CFilaInteiros::operator-` (operador subtração) que devolve uma fila que resulta de se retirar de uma fila (o 1º operando) os elementos que também estejam contidos numa segunda fila (2º operando). Tenha em atenção a declaração *standard* do operador.

Problema 9.13 – Difícil
Adicione um novo atributo `prioridade` na classe `CNoFila` e redefina a implementação de todos os métodos da classe `CFilaInteiros`, e em particular do método `insereNaFila()` de forma que a fila respeite sempre a regra: um novo elemento é inserido no final da fila, mas à frente de todos os elementos de prioridade menor e atrás de todos os elementos com prioridade igual ou superior; no limite, se todos os elementos atualmente existentes na fila tiverem prioridade menor que o novo elemento, o novo elemento é inserido no início da fila.
Defina ainda o método `bool CFilaInteiros::setPrioridade(int N, int p)` que mude para `p` o valor da prioridade do `N`-ésimo elemento da fila a contar do início da fila e que, em consequência, posicione corretamente na fila o elemento em causa, de forma que à sua frente só existam elementos de prioridade igual ou superior e atrás só existam elementos de menor prioridade. Se o `N`-ésimo elemento não existir, o método devolve `false`, caso contrário devolve `true`.

Problema 9.14 – Médio (adaptado do exame de recurso de 05/07/2016)
Considere as estruturas de dados definidas a seguir que representam uma pilha de números reais.
a) Implemente um método para contar os números negativos existentes na pilha, utilizando uma pilha auxiliar. Não deve manipular ponteiros.
b) Implemente novamente o método acima, manipulando ponteiros e não utilizando os métodos `push` e `pop`.

```
class CNoPilha{public: double dados;   CNoPilha *proximo; };

class CPilha{
   CNoPilha *topo;   // ponteiro para o topo da pilha
  public:
   CPilha();               // já definido (elsewhere)
   ~CPilha();              // já definido (elsewhere)
   void push(double dado); // já definido (elsewhere)
   bool pop(double &dado); // já definido (elsewhere)
};
```

Problema 9.15 – Médio (adaptado do exame de recurso de 5/07/2016)
Pretende-se implementar uma aplicação informática para gerir o sistema de apoio a clientes numa determinada empresa. Para isso decidiu-se criar uma lista ligada denominada `CListaHelpdesk` que armazena a informação relativa a todas as perguntas mais frequentes que são feitas pelos clientes. Cada elemento da lista, `CNoLista`, armazena pelo menos a seguinte informação: código da pergunta (5 caracteres), resposta à pergunta (300 caracteres) e o número de vezes que a pergunta foi pesquisada (inteiro).
a) Defina as classes `CListaHelpdesk` e `CNoLista` que permitem armazenar a lista das perguntas mais frequentes.
b) Para diminuir o tempo gasto em localizar uma determinada pergunta na lista, de cada vez que é feito um acesso a um nó (para consulta ou alteração de um dos seus campos de informação), este nó é movido para o início da lista. Deste modo, os nós acedidos com maior frequência ficarão localizados no início da lista. Implemente um método `PesquisaLista` que permita procurar uma pergunta na lista (dado o seu código) e devolver a resposta associada a essa pergunta. Pretende-se também que no caso de a pesquisa ter sucesso (o código ter sido encontrado), se mova o nó correspondente para o início da lista e se incremente o número de vezes que a pergunta foi pesquisada.

Problema 9.16 – Médio (adaptado do exame de recurso de 05/07/2016)
Considere as estruturas de dados definidas a seguir e que representam uma fila de países classificados através de um *rating* que representa o seu nível de desenvolvimento económico (um número inteiro entre 1 e 10). Implemente o método `void inserePais(char *Pais, int Rating)` que insere na fila um novo país, o mais perto possível do fim da fila, mas garantindo que o novo nó fica à frente de todos os nós com um rating maior e atrás de todos os nós com um rating igual ou inferior. Note que, no limite, se a fila estiver vazia, ou se os atuais nós da fila tiverem todos um rating superior ao do novo nó, este deve ser inserido no início da fila.

```
class CNoFila{    // Nó da fila
public:
  char pais[50];
  int  ranking;           //valor mais baixo indica um país mais desenvolvido
  CNoFila *proximo; //próximo nó
};

class CFilaRankings{  // Fila contendo países ordenados pelo seu ranking
  CNoFila *inicio;    // Ponteiro para o início da Fila
  CNoFila *fim;         // Ponteiro para o fim da Fila
public:
  CFilaRankings(){inicio=nullptr;   fim=nullptr;}
  ~CFilaRankings();
  ...
};
```

Considere o seguinte exemplo:
(Canada,1) → (Alemanha,1) → (Finlandia,2) → (Austria,2) → (Belgica,3) → (França,3) → nullptr

Problema 9.17 – Médio (adaptado do teste de frequência de 16/06/2017)
Considere a classe `CFilaInteiros`, para representar filas que armazenam números inteiros, objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo método constante da classe chamado `seek(int i)`, com visibilidade `private`, que devolve um ponteiro para o i-ésimo elemento da fila (a contar do início da fila). Assuma que o elemento com índice 1 é o que está no início da fila. Se o i-ésimo nó da fila não existir, o método devolverá `nullptr`.

b) Defina um novo método `public` da classe chamado `passaParaFim(int i)` que move o i-ésimo nó da fila para o fim da fila. O método não deve criar novos nós ou eliminar nós existentes; deve apenas religar a lista ligada (manipular ponteiros) para se obter o resultado pretendido. Se o i-ésimo nó da fila não existir, o método não fará nada e devolverá false, caso contrário devolverá `true`. Por exemplo, se a fila for `4, -1, 2, 8, 5` e chamar o método com o índice `i=3`, a fila fica `4, -1, 8, 5, 2`.

c) Defina um novo método `public` da classe chamado `passaParaInicio(int i)` que move o i-ésimo nó da fila para o início da fila. O método não deve criar novos nós ou eliminar nós existentes; deve apenas religar a lista ligada (manipular ponteiros) para se obter o resultado pretendido. Se o i-ésimo nó da fila não existir, o método não fará nada e devolverá false, caso contrário devolverá `true`. Por exemplo, se a fila for `4, -1, 2, 8, 5` e chamar o método com o índice `i=3`, a fila fica `2, 4, -1, 8, 5`.

   Sugestão: Conceba uma solução baseada na chamada ao método da alínea a) para obter um ponteiro para o i-ésimo elemento menos 1. Repare que desta forma, tem facilmente acesso ao i-ésimo elemento da fila e ao nó anterior a este.

Problema 9.18 – Médio (adaptado do exame de recurso de 03/07/2017)
Considere a classe `CPilhaInteiros`, para representar pilhas que armazenam números inteiros, que são objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo método constante da classe chamado `ponteiroPara(int i)`, com visibilidade `private`, que devolve um ponteiro para o elemento de índice i da pilha. Assuma que o elemento com índice `0` é o que está no topo da pilha. Se o nó de índice `i` da pilha não existir, o método devolve `nullptr`.

b) Defina um novo método da classe que realize a sobrecarga do operador `<<`. O operador elimina os últimos nós armazenados na pilha, a partir do nó de índice i (parâmetro `int`). O método não faz nada se `i` for menor que `0` ou maior ou igual ao número de nós da pilha, e elimina todos os nós da pilha se `i` for igual a `0`.

   Sugestão: Conceba uma solução baseada na chamada ao método da alínea a) para obter um ponteiro para o i-ésimo elemento menos 1. Repare que, desta forma, tem facilmente acesso ao i-ésimo elemento da pilha e ao nó anterior a este.

Problema 9.19 – Médio (adaptado do teste de 16/05/2018)
Considere a classe `CFilaInteiros` para representar filas que armazenam números inteiros, que são objeto de estudo nesta ficha das aulas práticas. Considere agora uma classe derivada da anterior, `CListaCircular`, que inclui mais um atributo (atual, privado) que é um ponteiro para o nó corrente (inicialmente o primeiro). Esta classe tem acesso a todos os atributos e métodos da classe mãe (ou classe base). Existem os métodos (públicos) da classe mãe e ainda o método `getNext()` que devolve o valor do próximo elemento (a seguir ao corrente, que passa a ser o novo nó corrente). O nó a seguir ao último (fim da fila) considera-se que é o primeiro (início da fila).

a) Escreva a declaração da classe de objetos `CListaCircular`. Nesta alínea, não deve definir/implementar qq. método *inline*.
b) Implemente o método `getNext()`.
c) Implemente uma função `main()` para teste que cria uma lista circular com os números 1 a 10 e mostra os elementos obtidos com 15 chamadas sucessivas a `getNext()`.

Problema 9.20 – Médio (adaptado do teste de 15/06/2018)

Considere as classes `CFilaInteiros` e `CPilhaInteiros` para representar filas e pilhas que armazenam números inteiros, que são objeto de estudo nesta ficha das aulas práticas.

a) Defina um novo construtor da classe `CFilaInteiros` que aceite como parâmetro uma pilha de inteiros e que construa uma fila com os números contidos na pilha. O número do topo da pilha deve ficar no início da fila e o de baixo, no fim.
b) Defina um novo construtor que aceite como parâmetros uma pilha de inteiros e um booleano e que construa uma fila com os números contidos na pilha. Se o booleano for `true`, só são inseridos na fila os elementos pares da pilha; se for `false`, só os ímpares.

Nota: No final, a pilha pode ficar vazia. As duas classes são independentes (nenhuma é amiga da outra).

FICHA 10
ÁRVORES BINÁRIAS DE PESQUISA

10.1. Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de árvore binária de pesquisa;
- Saber manipular as árvores, particularmente no que diz respeito à implementação de métodos de inserção/remoção de elementos e travessia das mesmas.

10.2. Introdução às árvores binárias de pesquisa

As árvores binárias são estruturas adequadas à representação de dados que obedeçam implicitamente a uma ordem hierárquica diádica, por exemplo a relação familiar entre um alguém e os seus ascendentes ou descendentes. Há ainda situações em que apesar de os dados não exibirem uma tal estruturação, determinadas operações que se pretendem efetuar sobre os mesmos podem ser muito facilitadas se estes forem armazenados numa árvore binária, por exemplo a pesquisa de conjuntos ordenados.

Na Figura 1, ilustra-se uma árvore binária destinada ao armazenamento de inteiros.

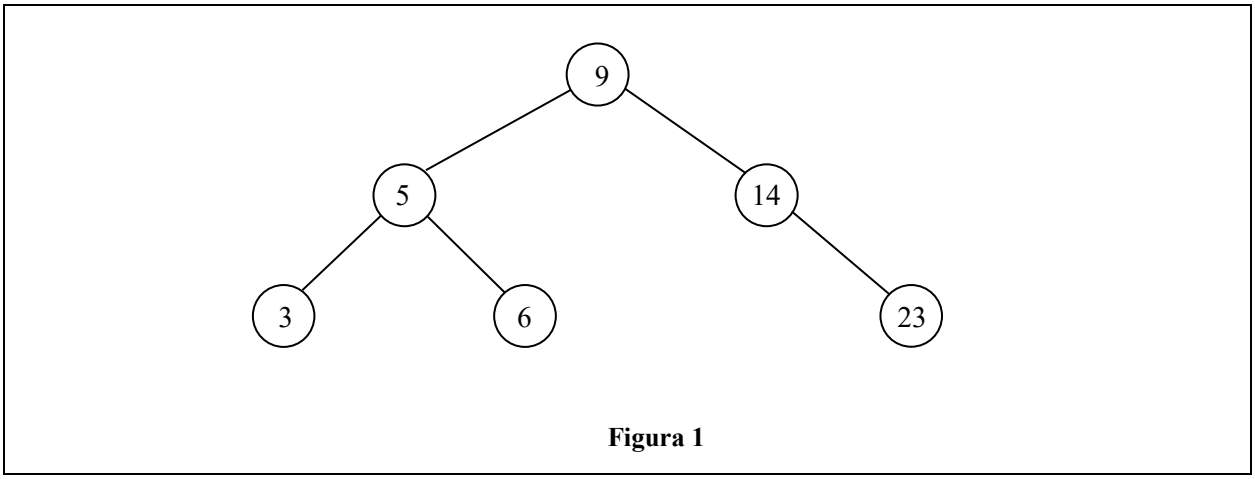


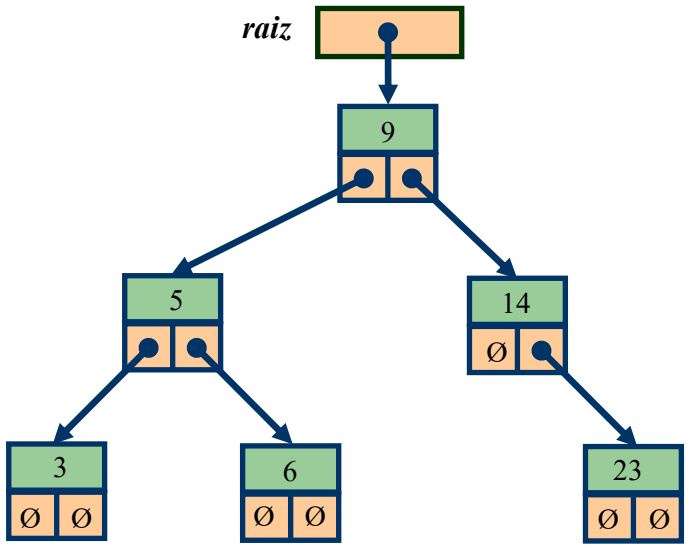
Figura 1

Repare na magnitude relativa dos elementos e correspondente posição na árvore. Verificará certamente que todos os elementos que constituem a subárvore esquerda de um determinado nó, caso ela exista, têm valor inferior ao valor desse nó. Analogamente, todos os elementos que constituem a subárvore direita de um determinado nó, caso ela exista, são maiores que o valor desse nó. Trata-se de uma **árvore binária de pesquisa.**

Há várias formas de implementar árvores binárias de pesquisa (e árvores em geral). Pode-se por exemplo optar por uma estrutura baseada numa tabela de elementos em que cada elemento representa um nó da árvore. Não obstante este tipo de representação ter algumas vantagens, neste documento trataremos apenas de uma forma alternativa baseada em ponteiros e variáveis dinâmicas, portanto uma estrutura de dados ligada, que passamos a descrever.

Na representação baseada em ponteiros e variáveis dinâmicas, cada nó é representado por uma *classe de objetos* que, para além de ter espaço reservado à informação, terá também um ponteiro para a subárvore esquerda e um ponteiro para a subárvore direita. Em certas situações, é vantajoso que esta estrutura de dados inclua também um ponteiro para o nó ascendente (nó pai). Por uma questão de simplicidade, não contemplaremos aqui essa possibilidade.

Com uma representação deste tipo, a árvore da Figura 1 teria o aspeto que se ilustra na Figura 2.



Legenda : Ø – Ponteiro Nulo

Figura 2

Para representar os nós de uma árvore com estas caraterísticas, podemos definir a classe `CNoArvore`:

```
class CArvoreBinaria; // "declarada" aqui para poder ser designada como amiga
                      //     antes de ser definitivamente declarada mais à frente

class CNoArvore {
  int dados;
  CNoArvore *esq; // ponteiro para subárvore esquerda
  CNoArvore *dir; // ponteiro para subárvore direita
  friend class CArvoreBinaria; // esta classe pode aceder a atributos private
};
```

Esta classe possui um atributo inteiro, `dados`, que representa a informação a armazenar (neste caso um inteiro) e dois atributos do tipo ponteiro, `esq` e `dir`, que permitem aceder às subárvores esquerda e direita, respetivamente.

A árvore propriamente dita, pode ser definida por uma classe semelhante à seguinte:

```
class CArvoreBinaria {
  CNoArvore *raiz;
  ...
  public:
  CArvoreBinaria();
  ~CArvoreBinaria();
  ...
};
```

Neste caso, o construtor tem como finalidade guardar no ponteiro `raiz` um valor (um endereço) consistente com o facto de inicialmente a árvore se encontrar vazia, ou seja, atribui a `raiz` o valor `nullptr`.

```
// Construtor
CArvoreBinaria::CArvoreBinaria() {
  raiz = nullptr;
}
```

O destrutor tem por fim "apagar" completamente a árvore e terá de efetuar a seguinte sequência de operações:

1. Se `raiz==nullptr` não há nada a fazer.
2. Caso contrário:
    a. Apaga subárvore esquerda,
    b. Apaga subárvore direita,
    c. Apaga nó atual.

Uma implementação possível é a que se apresenta a seguir, que foi construída com base num método que pode ser utilizado explicitamente para destruir toda a informação guardada na árvore, i.e. todos os nós:

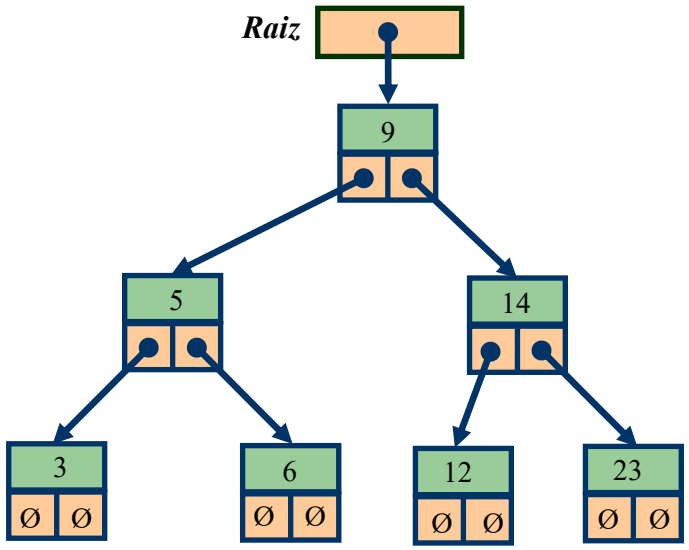
```
// Destrutor
CArvoreBinaria::~CArvoreBinaria() {
  destroi(); // método que elimina todos os nós
}
//-----------------------------------------------------------------------------
// Método public para destruir a árvore completa
void CArvoreBinaria::destroi() {
  destroi(raiz);
  raiz = nullptr;
}
//-----------------------------------------------------------------------------
// Método private e recursivo (apaga árvore "apontada" por 'pArvore')
```

```
void CArvoreBinaria::destroi(CNoArvore *pArvore) {
  if(pArvore != nullptr) {
    destroi(pArvore->esq);
    destroi(pArvore->dir);
    delete pArvore;
  }
}
```

Acrescentando estes métodos, a declaração da classe `CArvoreBinaria` passa a ser:

```
class CArvoreBinaria {
  // Dados e métodos privados
  CNoArvore *raiz;
  void destroi(CNoArvore *);
public:
  // Dados e métodos públicos
  CArvoreBinaria();
  ~CArvoreBinaria();
  void destroi();
};
```

Tente perceber as razões da necessidade de existência destes vários métodos e seus tipos de visibilidade (`public` ou `private`).



Legenda : Ø – Ponteiro Nulo

Figura 3

Uma estrutura de dados só tem alguma utilidade se for possível inserir nela elementos (dados). Temos assim que criar métodos para inserir novos dados, i.e. novos nós, na árvore binária. Vamos aqui tratar

apenas da inserção em árvores binárias ordenadas. Nessa situação, para inserir um elemento na árvore, é necessário um método que atravesse a árvore até atingir o "local" correto onde o novo elemento deve ser inserido.

Tomando como exemplo a árvore da Figura 2, se pretendermos inserir o elemento **12**, como este é maior que **9**, terá que fazer parte da subárvore direita. Sendo **12** menor que **14**, o novo elemento pertencerá à subárvore esquerda de **14**. Como, porém, esta subárvore está vazia, o novo elemento passará a ser a subárvore esquerda de **14**, tal como se vê na Figura 3.

Há uma situação particular que merece alguma reflexão. Neste exemplo, o elemento a inserir não estava presente na árvore. O que fazer se o elemento que pretendemos inserir já existe na árvore? Há duas soluções; ou se aceita que possam existir elementos duplicados, ou aquando de uma tentativa de inserção de um elemento já existente não se efetua a inserção. A escolha entre estas duas alternativas depende da aplicação do programa. No caso presente, optou-se pela segunda, ou seja só se procede à inserção se o valor a inserir ainda não existir na árvore.

Um método adequado para inserir um elemento inteiro de forma ordenada na nossa árvore binária é apresentado na caixa sequente. Também aqui se optou por separar o método num "front-end" público mantendo a "maquinaria" na parte privada.

```
// Metodo public chamado para fazer a inserção
void CArvoreBinaria::insere(int item) {
    if (raiz != nullptr) insere(item, raiz); // chama o metodo private
    else {  // cria o primeiro no da árvore
         raiz = new CNoArvore;
         raiz->dados = item;
         raiz->esq = raiz->dir = nullptr;
     }
}
//---------------------------------------------------------------------
// Método private que "faz o trabalho"
void CArvoreBinaria::insere(int item, CNoArvore *raiz) {
     if (item < raiz->dados) {
         if (raiz->esq != nullptr) insere(item, raiz->esq);
         else { // novo nó é o primeiro nó da subárvore esquerda
             raiz->esq = new CNoArvore;
             raiz->esq->dados = item;
             raiz->esq->esq = raiz->esq->dir = nullptr;
         }
     }
     else if (item > raiz->dados) {
         if (raiz->dir != nullptr) insere(item, raiz->dir);
         else { // novo nó é o primeiro nó da subárvore direita
             raiz->dir = new CNoArvore;
             raiz->dir->dados = item;
             raiz->dir->esq = raiz->dir->dir = nullptr;
         }
     }
}
```

A declaração da classe `CArvoreBinaria` atualizada com as declarações destes dois métodos é:

```
class CArvoreBinaria {
  CNoArvore *raiz;
  void destroi(CNoArvore *);
  void insere(int, CNoArvore *);
public:
  CArvoreBinaria();
  ~CArvoreBinaria();
  void destroi();
  void insere(int);
};
```

Por vezes é necessário pesquisar um determinado elemento numa árvore binária. Se a intenção for apenas determinar se esse elemento faz ou não parte da árvore, um método adequado devolve como resultado apenas um valor booleano indicativo da presença ou ausência do elemento. Em algumas situações, é importante devolver um ponteiro para o nó que contém o valor procurado. Na caixa seguinte, apresentam-se dois métodos que permitem fazer uma pesquisa da árvore e devolver um booleano. Também aqui existem dois métodos, sendo um o "front-end" público e outro o método `private` responsável pela pesquisa da árvore. Pense como o poderia alterar facilmente o código para devolver um ponteiro para o nó que contém o valor procurado.

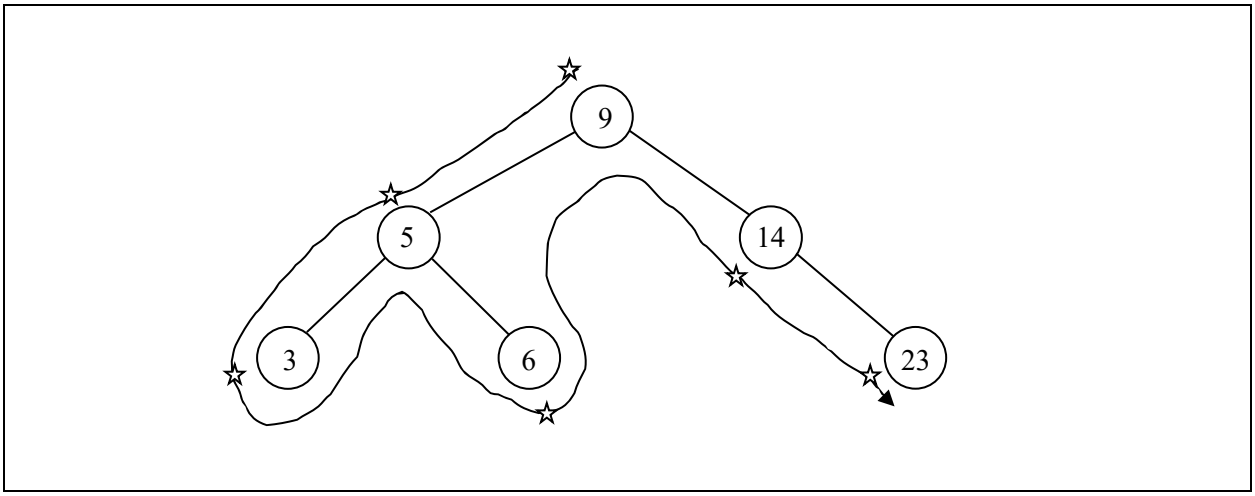
```
// Front-end public
bool CArvoreBinaria::procura(int item) const {
   return procuraRecursiva(raiz, item);
}
//-----------------------------------------------------------------------

// Método private
bool CArvoreBinaria::procuraRecursiva(CNoArvore *raiz, int item) const {
     // casos elementares
     if (raiz == nullptr) return false;
     if (raiz->dados == item) return true;

     // caso geral
    if (raiz->dados < item)
    return procuraRecursiva(raiz->dir, item);
    else return procuraRecursiva(raiz->esq, item);
}
```

Como sabe, listar todos os elementos de uma estrutura de dados linear, como as tabelas e listas, é uma tarefa trivial. O mesmo não se passa com as árvores pois, devido à sua estrutura, não possuem uma ordem natural. Com efeito, e tomando por exemplo a árvore binária da Figura 1, é possível mostrar o conteúdo da árvore segundo várias ordens.

Segundo a ordem **Pré-Ordem,**

Pré-Ordem
1. Lista elemento corrente, 2. Lista elementos da subárvore esquerda, 3. Lista elementos da subárvore direita.

como se ilustra a seguir,



onde a linha mostra o trajeto da travessia e as estrelas mostram em que ponto do trajeto o elemento adjacente é listado. Segundo esta ordem, os elementos seriam listados na sequência 9, 5, 3, 6, 14, 23. A listagem em pré-ordem é a adequada para guardar árvores binárias ordenadas em ficheiros, para se poder depois reproduzir a estrutura da árvore aquando da leitura desses ficheiros.

É possível definir outras duas ordens de travessia: a ordem **Por-Ordem** e a ordem **Pós-Ordem.**

Por-Ordem
1. Lista elementos da subárvore esquerda, 2. Lista elemento corrente, 3. Lista elementos da subárvore direita.

Pós-Ordem
1. Lista elementos da subárvore esquerda, 2. Lista elementos da subárvore direita, 3. Lista elemento corrente.

Elabore diagramas de travessia semelhantes ao apresentado acima para estas duas ordens e obtenha as respetivas sequências de listagem dos elementos. Observe que segundo a ordem **Em-Ordem**, os elementos seriam listados na sequência natural (i.e., por ordem crescente): 3, 5, 6, 9, 14, 23.

A listagem do conteúdo da árvore binária em ordem **Pré-Ordem** pode ser realizada usando os seguintes métodos:

```
// Front-end public
void CArvoreBinaria::escrevePreOrdem() const {
  if(raiz == nullptr){
    cout << "Árvore vazia!" << endl;
    return;
  }
  escrevePreOrdem(raiz);
}
//-------------------------------------------------------------
// Método private
void CArvoreBinaria::escrevePreOrdem(CNoArvore *pArvore) const {
  if (pArvore == nullptr) return;
  cout << pArvore->dados << " ";
  if (pArvore->esq != NULL) escrevePreOrdem(pArvore->esq);
  if (pArvore->dir != NULL) escrevePreOrdem(pArvore->dir);
}
```

Remover elementos de uma árvore binária ordenada é uma operação mais complexa do que a inserção. Tomando como exemplo a árvore da Figura 4, suponhamos que pretendemos remover o nó **1**. Como esse nó é um nó terminal, haveria apenas que destruir o nó (`delete` ...) e guardar no ponteiro `esquerdo` do nó **2** o valor `nullptr`. Igualmente simples é a remoção de um nó com uma só subárvore, pois basta alterar o ponteiro que apontava para o nó removido por forma a que este passe a apontar para a subárvore do nó eliminado (estude o caso da remoção do nó **14**). Quando o nó a remover não satisfaz nenhuma das duas condições anteriores, o processo é um pouco mais complexo.

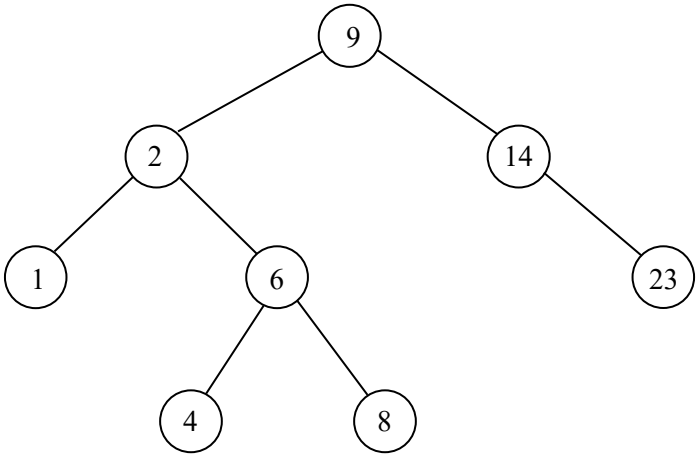


Figura 4

Tomemos por exemplo a remoção do nó **2**. Se apagarmos esse nó sem fazer mais nada, o resultado é o que se ilustra na Figura 5. Falta claramente fazer algo...

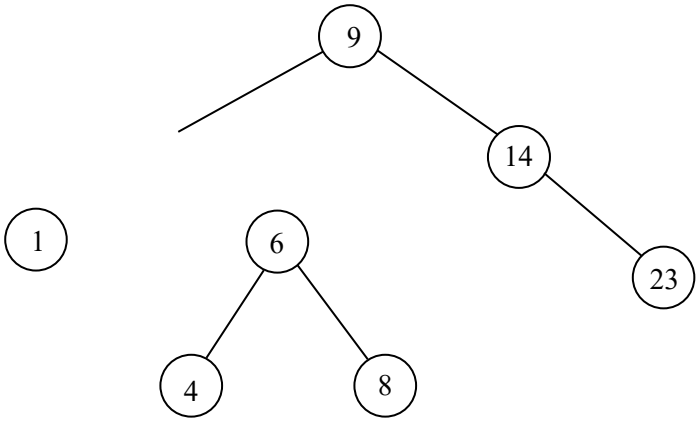


Figura 5

O que está mal nesta situação é que as subárvores esquerda e direita do elemento que foi removido da árvore não foram religadas à árvore. Coloca-se agora a questão: onde colocar estas subárvores, ou os seus elementos? Há várias soluções possíveis. Uma delas consiste em *"Substituir o nó eliminado pelo nó seguinte, segundo a ordem **Em-Ordem**"*. No caso exemplo, o nó que se segue ao nó **2** segundo esta ordem (EmOrdem) é o nó **4**, e o resultado seria o que se mostra na Figura 6.

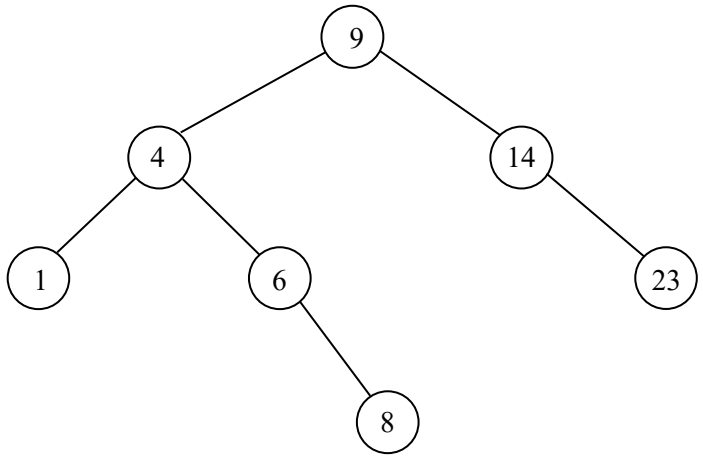


Figura 6

Como pode comprovar, a árvore final é uma árvore binária de pesquisa válida. Existem ainda casos que levantam problemas especiais, tais como a remoção do elemento `raiz`. Na caixa seguinte apresenta-se um método que implementa todos estes casos. Analise o código com cuidado e verifique que permite resolver todos os casos possíveis na remoção de um nó de uma árvore binária de pesquisa...

```
bool CArvoreBinaria::elimina(int item) {
     CNoArvore *pai, *aux, *filho, *sucessor;
     bool encontrado=false;

      // procura o nó a eliminar
     pai = nullptr; aux = raiz;
     while ( (aux != nullptr) && (!encontrado) ) {
         if (aux->dados == item) encontrado = true;
         else if (item < aux->dados) {
             pai = aux;
             aux = aux->esq;
         }
         else {
             pai = aux;
             aux = aux->dir;
        }
     }

     // se for encontrado 'aux' aponta para o nó
     // a eliminar e 'pai' aponta para o seu pai (se existir)
     if (encontrado) {
         // se nó tem no máximo um filho
         if ( (aux->esq == nullptr) || (aux->dir == nullptr) ) {
             if (aux->esq != nullptr) filho = aux->esq;
             else filho = aux->dir;
```

```
            if (pai == nullptr) // caso especial: apaga raiz
                raiz = filho;
            else {           // atualiza ligações
                if (aux->dados < pai->dados) pai->esq = filho;
                else pai->dir = filho;
            }
            delete aux;
        }
        else {  // eliminar nó com dois filhos
            // procura sucessor
            sucessor = aux->dir;
            pai = aux;
            while (sucessor->esq != nullptr) {
                pai = sucessor;
                sucessor = sucessor->esq;
            }
            aux->dados = sucessor->dados; // troca val.
            if (sucessor == aux->dir) aux->dir = sucessor->dir;
            else // substitui pelo seu sucessor `a dir.
                 pai->esq = sucessor->dir;
            delete sucessor;
        }
        return true;
    }
    return false;
}
```

A declaração completa da classe `CArvoreBinaria` é então:

```
class CArvoreBinaria {
   CNoArvore *raiz;
   // ponteiro para a raiz da árvore
   bool procuraRecursiva(CNoArvore*, int) const;
   void escrevePreOrdem(CNoArvore*) const;
   void insere(int, CNoArvore*);
   void destroiSubArvore(CNoArvore*);
public:
   CArvoreBinaria();
   ~CArvoreBinaria();
   bool procura(int) const;
   void escrevePreOrdem() const;
   void insere(int);
   bool elimina(int);
   void destroi();
};
```

10.3. Problemas Sugeridos

Com o conjunto de problemas enunciados a seguir, pretende-se que o aluno consolide os seus conhecimentos relativos a árvores binárias de pesquisa. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa todos os exercícios não realizados durante as aulas práticas. Todos os problemas assumem a declaração das classes `CNoArvore` e `CArvoreBinaria` apresentadas anteriormente e que esta última é estendida para responder ao problema enunciado.

Problema 10.1 – Fácil

Adicione um novo método para listar na consola o conteúdo de uma árvore binária ordenada, segundo a ordem *Por-Ordem*.

Problema 10.2 – Fácil

Adicione um novo método para listar na consola o conteúdo de uma árvore binária ordenada, segundo a ordem *Pós-Ordem*.

Problema 10.3 – Fácil

Adicione um novo método que devolva os valores máximo e mínimo de entre todos os elementos constituintes de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.4 – Fácil

Adicione um novo método que devolva a soma de todos os elementos de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.5 – Fácil

Adicione um novo método que devolva a soma de todos as folhas de uma árvore binária de pesquisa que guarda números inteiros.

Problema 10.6 – Fácil

Adicione um novo método que devolva a soma de todos as folhas esquerdas da árvore, ou seja, elementos aos quais se chega através do ponteiro esquerdo dos respetivos nós pai.

Problema 10.7 – Fácil

Adicione um novo construtor da classe `CArvoreBinaria` que inicialize a árvore com os elementos de uma tabela de inteiros passada por parâmetro. Existe um segundo parâmetro, inteiro, que contém o nº de elementos da tabela.

Problema 10.8 – Médio

Adicione um novo método que permita gravar a informação de uma árvore binária de pesquisa num ficheiro, segundo a ordem *Pré-Ordem*. Crie também o método de leitura que permita realizar a operação inversa, ou seja ler o ficheiro e carregar os dados na memória usando uma árvore binária de pesquisa.

Problema 10.9 – Médio

Definindo desequilíbrio de uma árvore binária de pesquisa como a diferença entre a profundidade da folha mais profunda e da folha menos profunda, adicione um novo método que calcule e devolva este valor.

Problema 10.10 – Médio (saiu no teste de frequência de 23/06/2015)

Defina o método público `int CArvoreBinaria::profundidade(int item) const` que pesquisa o valor item na árvore. Se o valor for encontrado, o método devolve a profundidade na árvore do nó em que se encontra guardado, ou seja o comprimento do caminho desde a raiz até esse nó. Se não for encontrado, devolve `-1`. O método pretendido não deve chamar nenhum dos métodos já existentes na classe `CArvoreBinaria` e não pode escrever mensagens no ecrã.

Sugestão: Comece por definir um método privado e recursivo que pesquisa o valor numa subárvore e devolve a profundidade do nó, se o valor existir; devolve `-1` se não existir. Para além de `item`, terá de passar como parâmetro a este método auxiliar um ponteiro para o nó na raiz da subárvore.

Problema 10.11 – Difícil (saiu no exame de recurso de 10/07/2015)

Considere a classe `CListaInteiros` que é objeto de estudo na ficha 8 das aulas práticas. Defina o método público `bool CArvoreBinaria::caminhoDesdeRaiz(int item, CListaInteiros &cam)` que pesquisa o valor `item` na árvore. Se for encontrado, o método devolve `true` e, através do parâmetro `cam`, passado por referência, devolve a sequência de valores guardados na árvore, ou seja o caminho, desde a `raiz` (valor guardado na cabeça da lista a devolver) até ao nó onde for encontrado o valor pretendido (inclusive). Se não for encontrado, devolve `false` e, através do parâmetro `cam` passado por referência, devolve uma lista vazia (caminho inexistente). O método pretendido não pode chamar nenhum dos métodos já existentes na classe `CArvoreBinaria` e não pode escrever mensagens no ecrã.

Sugestão: Comece por definir um método privado e recursivo que pesquisa o valor numa subárvore e constrói o caminho desde a raiz da subárvore até ao nó onde for encontrado o valor pretendido. Para além de item e da lista `cam`, terá de passar como parâmetro a este método auxiliar um ponteiro para o nó na raiz da subárvore.

Problema 10.12 – Médio (adaptado do teste de frequência de 16/06/2017)

Defina um novo método da classe `CArvoreBinaria` chamado `obtemMinExcetoFolhas()`, com visibilidade `public`, constante, que devolve o menor número armazenado na árvore dentre todos os nós que não são folhas da árvore (só contam os nós que tenham descendentes). Se a árvore estiver vazia ou contiver apenas um nó (sem descendentes), deverá ser devolvido o valor `INT_MAX` (o maior dos números inteiros). Para além do método pedido, poderá implementar outros métodos auxiliares que considere úteis para a sua resolução. Os métodos implementados não devem escrever mensagens no ecrã.
Nota: Pretende-se um algoritmo que não tenha de percorrer sempre todos os nós da árvore.

Problema 10.13 – Médio (adaptado do exame de recurso de 03/07/2017)

Defina um novo método público da classe `CArvoreBinaria` chamado com a declaração `int valorPai(int n) const` que devolve o número armazenado no nó pai do nó que contém o número `n`. Se `n` estiver na raiz da árvore ou se a árvore estiver vazia deverá ser devolvido o valor `INT_MIN` (o menor dos números inteiros).
Sugestão: Comece por definir um novo método `private` da classe chamado `ponteiroParaPai(..., int n, ...)`, recursivo, que devolve um ponteiro para o nó pai do nó que contém o número `n`. Se `n` estiver na raiz da árvore ou a árvore estiver vazia, este método auxiliar deverá devolver o valor `nullptr`.

Problema 10.14 – Médio (adaptado do exame de recurso de 03/07/2018)

Adicione à classe `CArvoreBinaria` os métodos seguintes:

a) Método `CNoArvore* getNo(int, CNoArvore*) const`, com visibilidade `private`, que procura um número inteiro (valor do 1.º parâmetro) numa subárvore – o 2º parâmetro é o ponteiro para a raiz da subárvore –, devolvendo um ponteiro para o nó da subárvore onde for encontrado o número procurado, ou `nullptr` se este não for encontrado na subárvore.

b) Método `int altura(CNoArvore *pArvore) const`, com visibilidade `private`, que devolve a altura do nó na raiz da subárvore passado como parâmetro, ou seja o comprimento máximo do caminho entre esse nó e as suas folhas descendentes. Se a subárvore estiver vazia, o método deve devolver `-1`.

c) Defina o método `int altura(int item) const`, com visibilidade `public`, que devolve a altura do nó da árvore que contém `item`. Se `item` não existir na árvore, o método deve devolver `-1`. O método deve ser implementado com base num algoritmo que consiste na chamada aos dois métodos definidos nas alíneas anteriores.