A brief booklet for my C course

by Matteo Fraschini

Introduction

Dear student, this document is for you. I know that attending the lectures may be sometimes hard, sometimes boring and even when it is not, there are several other reasons that make it difficult to go back home and study what you have listened.

Here you find the most significant concepts that I have discussed during the classes, with a summary and a description of what I think it is more important for you, the *take-home* messages. At the following link https://matteogithub.github.io/teaching/ you find all the slides (in PDF and md format) and all the *.c* scripts.

This is an ongoing work and obviously **can not replace a book** and can not be enough if you do not spend time *playing* with the code. Learning to code is not merely learning the *syntax* of a specific language. It means much more. Creativity, logical thinking, and determination are extremely important factors in this context.

Level 0: to start

Probably, right now, your level is **0**. It means that, as for most of your colleagues, you have never coded before. This is not a problem. Indeed, this booklet is for students like you. There are not specific prerequisites for this course. The number 0 is very important, you will read about this number again in this booklet. Your level will change, will increase, adding more competencies, the offset from the initial level will change. Just remember this when we will talk about *arrays*. You will understand that starting from 0 may be relevant!

All you need to start is to set up a *development environment*. There are several solutions for this task - not independent from your Operating System - choose the one you like more. As a Unica student, you can apply for a free license of **cLion**: https://www.jetbrains.com/shop/eform/students. Another good solution is to use **Atom**, which however may need more tuning on the settings.

Now that your system is ready to work, just try a very simple program. All you need is to open a new file, save it as .c and assign a name. Copy and paste the code you find below in your editor:

```c
//My first program
#include <stdio.h>
int main() {
  printf("Hello world");
  return 0;
}
```

Now you need to compile and run the program. This program should work correctly. You should not have any error or warning. What does this mean? This means that you should see a *terminal* window pop-up and the sentence "*Hello world*" appear on the screen. If this happened, well, that's great. You have written your first program.

Before to go on with something more complex and interesting, just try to understand what you have written in your .c file. I am quite confident that it is very important to understand the basic structure of a C program. This is especially easy to understand now when the program is short.

The line `//My first program` is a comment. This line is ignored by the compiler. However, using comments into your scripts is really useful. You should include more comments you can. It will be of great help when you (or others) will open your file later.

The line `#include <stdio.h>` includes an external library to your program. A library allows you to use a set of functionalities that you do not need to re-implement by yourself. In this specific case the `stdio.h` (standard input and output) library gives you the possibility to use the `printf` function (and others). When it will be necessary, I will show you other interesting libraries that you can easily include in your code.

Every C program has a **main** function. This function is called `main`. Your program will start from there, always from there. You will learn more about functions later. All you need to know now is that this function will contain the instructions (but not only) that your program will perform and that it (as in this case) will return an integer (see the `int` before the name of the function). Your `main` function has a name, has an `int` value that is expected to be returned and has a *body*. The body is delimited by the braces `{}`.

The last instruction of the main function is `return 0;`. This is the value your function returns. If you omit to include this instruction your program will work, but you will discover later that this is not a simple aesthetic point.

Level 1: variables

Now you are ready to use *variables*. Think about *variables* as containers that will store the data you want to use during your program. The easy part is that you can refer to this container (a block of the memory of your computer) with a name, wherever it is. You can choose the name you like, there are only a few simple rules you have to respect for the name. However, just try to use a name that allows understanding what the variable will store.

In C, before to use a variable you have to *declare* it. The declaration is important because it links your variable with a *type*. When you choose a type, you are defining the dimension (number of bytes) of the container and the operations you will be able to perform on this variable. The dimension may still depend on the architecture of your computer/machine. This is an example of a variable declaration:

```
int number;
```

With this instruction, you are declaring a variable named `number` as an integer (`int`). What happens is that a block of memory will be reserved for this variable and that you can access this block of memory simply using its name. How is this block long? Probably 4 bytes, but just test it yourself:

```
#include <stdio.h>
int main() {
  int number;

  printf("\nAn integer need %ld byte",sizeof(number));
  printf("\nAn integer need %ld byte",sizeof(int));
}
```

As you can see, the `printf` function may be used to print the result of an expression or the content of a variable. You just need to indicate where you want the value to appear using a specific format (`%ld` in this case, which refers to `long integer`) and report the expression/s or the variable/s after the comma. The character `\n` represents a new line, so that the output in the terminal may be read more easily.

Let's go back to declarations and types. There are few basic types. You will see later that you can define new types. The basic types are: `int`, `float`, `double` and `char`. You will choose the type to assign to a variable depending on the context of the program. From this choice, it will depend (i) what you can do with your variable and (ii) the dimension of the block of memory reserved for this variable.

To interact with your variables you can use the arithmetic operators (`+`, `-`, `*` and `/`) to build some very simple expressions. The symbol `=` is used to assign a value to a variable as shown below:

```
  #include <stdio.h>
  int main() {
    int number;
    long int num_byte;

    num_byte = sizeof(int);
    printf("\nAn integer need %ld byte",num_byte);
  }
```

The variable `num_byte` is declared and successively its value will be initialised with the result of the expression `sizeof(int)`. As a consequence, the block of memory that you call `num_byte` (which is 4 bytes long) will store the (binary) representation of the corresponding number (4, in this case).

It is very important to do not confuse the symbol `=` with the symbol `==`. The first is intended to assign a value to a variable, the second is intended to compare variables or expressions. If you use `=` when you were intended to use `==` you can not get an error and therefore your program will work but not as you would expect!

Level 2: input and output

To further increase your level, you need to learn how to use a very important and basic function. This is the `scanf`. The `scanf` is not always easy to use (you will understand this later) however, it represents the first tool you can use to make a user interacting with your program. In particular, by using the `scanf` you can pass an *external* input to your program - reading the information from the keyboard. Here a simple example:

```
  #include <stdio.h>
  int main() {
    int base, height;
    printf("Insert the base: ");
    scanf("%d", &base);
    printf("Insert the height: ");
    scanf("%d", &height);
    printf("The area is: %d\n",base * height);
  }
```

This very simple program takes as input two integers from the keyboard (base and height) and computes the area of a rectangle. The `scanf` is in some way similar to the `printf` since you still have to define the format (`%d` in this case as it is a decimal integer value). Note that you need to write the symbol `&` before the variable name. This symbol allows referring to the *address* of the variable. This is important because you want to change the content of the variable and since all arguments in C are passed (to functions) *by value* you need to pass the address of a variable if you want the function to change the content (and not to work on a copy of the variable). We will go back to this point later in more details.

Level 3: flow control statements

You can extend your program adding *flow control statements* which allows selecting what actions to take on the basis of specific conditions. The more common *flow control statement* is the **if** statement which represents the most simple form of the *branching*. It takes and evaluates an expression and if the expression is true then the instruction (or block of instructions) gets executed otherwise the instruction (or block of instructions) is skipped. You can combine with the **else** statement to create an alternative branch in the case the condition is false. Remember that a block is defined by using the `{}`, and this is very important if you want to specify a set of instructions, otherwise your program will execute only the first instruction that follows the expression to be tested. See this simple program to evaluate if a number is even or odd:

```
#include <stdio.h>
int main() {
  int n;

  printf("Type a positive integer: ");
  scanf("%d",&n);
  if(n%2 == 0)
    printf("\n%d is even\n",n);
  else
    printf("\n%d is odd\n",n);
}
```

It is important to keep in mind that in C any **non-zero** value is interpreted as **true** (**false** if it is **zero**). *Relational* and *logical* operators may be used to build more complex expressions, see this example:

```
#include <stdio.h>
int main() {
   int year;
```

```
    printf("Insert a year (i.e., 2018): ");
    scanf("%d",&year);
    if ((year%4 == 0 && year%100 != 0) || year%400 == 0)
       printf("\n%d is a leap year\n", year);
    else printf("\n%d is not a leap year\n", year);
}
```

Another commonly used *flow control statement* is the `switch-case`. This statement allows selecting a specific block of instructions based on the value (`int` or `char`) assumed by a variable or by an expression tested for equality against a list of constant values. See the following example:

```
#include <stdio.h>
int main() {
  int n;
  printf("Type a positive integer: ");
  scanf("%d",&n);

  switch (n % 2)
  {
    case 0:
        printf("\nThe number %d is even\n",n);
        break;
    case 1:
      printf("\nThe number %d is odd\n",n);
        break;
  }
}
```

Note that the expression `n % 2`, where `%` is called modulus operator, returns the remainder of `n` divided by 2.

Level 4: loop statements

If you want to build even more powerful program, more clear and concise scripts, all you need is a *loop statement*. When you need to execute an instruction (or a block of instructions) more than one time you have to define a loop. The basic rule, in this case, is: *never write the same instruction twice*. The loop statement you will use more frequently is the **for**. In its more simple form, a `for` loop will look like this: `for(i=0;i<n;i++)` followed by an instruction or a block of instructions (in this case you need to define a block using `{}`). As you may have noticed this statement contains three different fields (separated by a two `;`). In the first field, the variable `i`, that needs to be declared somewhere, represents the variable that controls the loop. The instruction `i=0` will be executed only the first time and allows to initialize this variable. Keep in mind the initializing is very important, and that a wrong assignment may strongly affect your program even if you do not get any error from the compiler. The second field contains a conditional expression (`i<n`) that will be evaluated **before** to execute the instruction or the block. This is the reason why a `for` loop (as well as a `while` loop) may be never executed. If the expression is false the instruction (or the block) will be skipped. The third field (`i++`), an increment of the control variable will be executed, this happens after all the instructions contained in the loop have been executed. The instruction `i++` is just another way to write `i=i+1`. When you use this compact form, you may need, in some specific case, to pay

attention to the difference between `i++` and `++i`. This is not the case of a `for` loop when the increment of `i` will be done after all the other instructions, but if the increment is inside an expression you may observe a different behavior between the two different approaches. Try to assign to a variable the result of the increment and see what happens in the two different cases (i..e, `n=i++;` and `n=++i;`). Another potential problem that you have to consider when you design your loop is to avoid the definition of an *infinite loop*. This is the case when the conditional expression inside the loop is always true.

Now see this very simple example on the use of a `for` loop:

```
#include <stdio.h>
int main() {
  int i,n,num,sum=0;
  float mean;

  printf("How many integers you want to add? ");
  scanf("%d",&n);
  for(i=1;i<=n;++i) {
    printf("\nInteger #%d: ",i);
    scanf("%d",&num);
    sum+=num;
  }
  mean=(float)sum/n;
  printf("\nThe mean is equal to: %f\n",mean);
}
```

In this simple example, you have seen a few new things. In particular, the sum of the numbers is expressed in the compact form (`+=`) derived by combining two different operators. The instruction `sum+=num;` is equals to `sum=sum+num;`. Although the first expression is more compact and in any case more commonly used, you need to pay attention since it makes less clear that the variable `sum` is also contained in the right side of the expression, and therefore it is very important to initialize it before to use it. The other novelty is due to the use of the *cast* (`(float)sum`). Using this operator you are changing the type of the variable `sum`. This is important in this case because, if you don't use it, the result of `sum` divided by `n` will be an integer (as both are integers) and it is not exactly what you would expect. Just try to remove the cast and see what the result looks like.

As previously mentioned there is at least another very useful loop statement in C, the `while` loop. Most of the time you will experience that you can easily use these two different loops in an interchangeably way. My preference is to use the `for` when I have a priori knowledge of the number of iterations. One of the more evident differences is that after the keyword `while`, inside the `()` you only have to define the conditional expression. There is no formal initialization of any variable to control the loop and there is not a formal increment/decrement of this variable. This does not mean that you may just ignore this variable. Probably, all you need to do is to perform these steps in a different way. See the following example which is equivalent to the one seen before in the case of the `for` loop:

```
#include <stdio.h>
int main() {
  int i=1,n,num,sum=0;
```

```
    float mean;

    printf("How many integers you want to add? ");
    scanf("%d",&n);
    while(i<=n) {
      printf("\nInteger #%d: ",i);
      scanf("%d",&num);
      sum+=num;
      ++i;
    }
    mean=(float)sum/n;
    printf("\nThe mean is equal to: %f\n",mean);
  }
```

The two programs are equivalent. You may, in general, use a `for` or a `while` indifferently.

Although it is less frequently used, the `do-while` loop may be not completely ignored. The only difference with the `while` is that the `do-while` allows executing the instruction (or a block) at least one time. We will use this loop later in some exercise when necessary.

Level 5: arrays

Now things will start to be more interesting. Arrays are a very powerful tool. If you need to manage a set of homogenous variables (all of the same *type*) you may not need to declare them separately. As for every other variable, an array is first declared: `int v[100];`. This declaration defines a monodimensional array, named `v` where each element has the same name and the same type `int`. Between `[]` you have to define its dimension. This means that 100 blocks of memory of the same dimension (4 bytes) will be reserved for your variable `v`. It is very important to remember that these blocks are physically adjacent. Therefore, if you know the address of the first element, you can easily access to all the other elements just moving to the next block. To select an element you need to use an *index* as follows: `v[5]` (this represents the element of index 5). Remember that the first element of an array has index `zero` (i.e., `v[0]` is your first element). As a consequence, if your array has 100 elements, the last one has index equals to 99 (i.e., `v[99]` is your last element). Pay extra attention to do not try to access elements outside this range (`0:N-1` if N is the dimension of your array)! Your compiler may not give you an error, but this is a very dangerous situation since you have not the control of the content of the next memory block. Also, try to avoid using a fixed number to specify the dimension of the array. It is preferable to use a `#define N 100` that can be more easily changed later if you need to increase the size of your array.

See this very simple example on the usage of an array:

```
  #include <stdio.h>
  #define NGRADE 10
  int main() {
    int i,n,grade[NGRADE],sum=0;
    float mean;

    printf("How many grades you want to insert? ");
    scanf("%d",&n);
    for(i=0;i<n;++i) {
```

```
    printf("\nInsert grade #%d: ",i+1);
    scanf("%d",&grade[i]);
    sum+=grade[i];
    }
  mean=(float)sum/n;
  printf("\nThe mean is eqaul to: %f\n",mean);
}
```

The main difference with the previous example is that in this case, after the `for` loop you still have all the grades stored inside the array, each one in a specific and consecutive block of memory. Remember that as a programmer it is you that need to *design* your array to store a sufficient number of elements. If 10 is not enough, just change the value into the `#define`, so you do not need to change your code in any other place.

Remember that you can not assign an array to another array using this instruction: `v=w;`. The name of an array is a *constant pointer*, meaning that `v` stores the address of the first element of the array: `v=&v[0]` and you cannot make `v` points to another memory address. If you need to copy an array into another array, you have to copy every single element at a time (therefore acting with an index into a loop).

To have a good control of the dimension of your array it is a good practice to test that it will not go to accept more elements of its capacity. See this example:

```
#include <stdio.h>
#define N 10
int main() {
  int i,n,v[N],sum=0;

  do  {
    printf("Dimension? ");
    printf("It must be in the range 1-10!");
    scanf("%d",&n);
  } while(n<1 || n>N);

  for(i=0;i<n;++i) {
    printf("\nType the integer #%d: ",i+1);
    scanf("%d",&v[i]);
    sum+=v[i];
    }
  printf("\nThe sum is: %d\n",sum);
}
```

Let's see some more examples on the use of arrays. The next asks to store N grades in an array and evaluate the mean, the max, and the min grade.

```
#include <stdio.h>
#define NGRADES 10
int main() {
  int i,n,grades[NGRADES],sum=0,min,max;
  float mean;

  do  {
    printf("How many grades? ");
    scanf("%d",&n);
  } while(n<1 || n>NGRADES);

  for(i=0;i<n;++i)  {
    printf("\nType grade #%d: ",i+1);
    scanf("%d",&grades[i]);
    sum+=grades[i];
    }
  mean=(float)sum/n;
  printf("\nThe mean is: %.1f\n",mean);

  min=grades[0];
  max=grades[0];
  for(i=0;i<n;++i) {
    if(grades[i]<min) min=grades[i];
    if(grades[i]>max) max=grades[i];
  }
  printf("\nThe max grade is: %d\n",max);
  printf("\nThe min grade is: %d\n",min);
}
```

As you can see, in order to compute the min (max) of the elements stored in an array you can start assuming that its first element is indeed the min (max). After that, you scan all the elements and if you find an element minor (greater) of min (max) than min (max) will be changed to the value of the current element.

Now let's see how to *search for* a specific value in an array of integers.

```
#include <stdio.h>
#define DIM 100
int main() {
    int v[DIM], n, i, elem;

    do  {
      printf("Dimension of the array: \n");
      scanf("%d", &n);
    } while(n<1 || n>DIM);

    for(i=0;i<n;i++)  {
        printf("Type element of index - %d : ",i);
        scanf("%d",&v[i]);
    }
    printf("\nType the integer you want to search: ");
    scanf("%d",&elem);
    i=0;
    while(elem!=v[i] && i<n) ++i;
    if(elem==v[i]) printf("Element found in position %d\n",i);
    else printf("Element not found!\n");
}
```

After the array is loaded, the user enters an integer (`elem`) to be found in the array. In this case, the problem is solved using a `while` loop (note the use of the variable `i`, initialized outside the loop and later incremented inside the loop). The condition (`elem!=v[i] && i<n`) that is tested before to execute the instruction of the loop (`++i`) is about to check if `elem` was found and that you still have elements to scan on the array (avoiding to match `elem` with blocks of memory outside the array `v`). Therefore, you can go out from the loop when you have found the integer or alternatively when you have verified all the elements of your array. Anyway, you still do not know, so you need to check what is the reason that let your program go out from the `while` loop. Try with `if(elem==v[i])`, if this condition is true it means you have indeed found `elem` inside the array, otherwise you have not.

In the following example, you will see how *to order* the elements of an array in ascending order. The proposed solution is very simple and intuitive, however, it does not represent an optimal way to solve the problem as we will briefly discuss later.

```
#include <stdio.h>
#define DIM 100
int main() {
    int v[DIM], n, i, j, tmp;

    do  {
      printf("Dimension of the array: \n");
      scanf("%d", &n);
    } while(n<1 || n>DIM);
    for(i=0;i<n;i++)  {
        printf("Type element of index - %d: ",i);
        scanf("%d",&v[i]);
    }
    //start ordering
    for(i=0; i<n-1; i++)  {
        for(j=0; j<n-1; j++) {
            if(v[j] > v[j+1]) {
                tmp = v[j];
                v[j] = v[j+1];
                v[j+1] = tmp;
            }
        }
    }
    //ordering finished

    printf("\nYour ordered array:\n");
    for(i=0; i<n; i++)
        printf("%d  ", v[i]);
}
```

As you can see, two `for` loops have been used to order the array. The innermost loop compares every single element of the array with the following, if the first is greater you swap them, otherwise, you go to test the next element. Since the innermost loop will access the last element using the index `j+1` you have to pay attention to do not access to elements outside the array, so the condition inside the loop is set to `j<n-1`. If you are lucky (your array is not too messy), it is enough and you get your array properly ordered. But you are not sure this is the case, so to be sure that the array will be finally ordered (irrespectively of its initial order) you need to use another loop, the outermost one. Despite the code will work and the result will be fine, you can understand that the two `for` loops will be executed even if the array is already ordered. There are several approaches (out of the scope of this course) that can be used to optimize this code, thus avoiding to go on with further iterations when this is not necessary.

[In progress: bidimensional arrays and strings]

Level 6: structures

Arrays are sets of homogenous elements (all the elements are of the same type). With the *structures* you can define a completely new and personalized type. The variables declared of this type may have several components, each of specific and different type. Here's a very simple example of structure *definition* and *declaration*.

```
struct student {
        int ID_number;
        char name[DIM_NAME];
        char surname[DIM_NAME];
    };

    struct student my_student;
```

You can access the components in this as follow: `my_student.ID_number`, `my_student.name` and `my_student.surname`.

To load data into the variable `my_student` see this code:

```
#include <stdio.h>
#define DIM_NAME 20

struct student {
        int ID_number;
        char name[DIM_NAME];
        char surname[DIM_NAME];
    };

int main() {
    struct student my_student;

    printf("\nName: ");
    scanf("%s",my_student.name);
    printf("\nSurname: ");
    scanf("%s",my_student.surname);
    printf("\nID: ");
    scanf("%d",&my_student.ID_number);
    printf("\n\nStudent data: ");
    printf("%s %s with ID
%d\n",my_student.name,my_student.surname,my_student.ID_number);
}
```

Note that when you use the `scanf` for a string you do not need to use the symbol `&`. This is because `scanf` expect the address of a variable, but the name of a string is the pointer to the first element of the array of characters so it is not necessary to specify the address using the symbol `&` as you do for any other variable.

Anyway, it is better do not use `scanf` for strings at all. This is because `scanf` does not explicitly allow to check for the dimension of the string and because it may not work as you expect. If you type a *space* your input will be interrupted. The best solution to input strings is by using the function `fgets`. See the new version of the last code:

```
#include <stdio.h>
#define DIM_NAME 20

struct student {
        int ID_number;
        char name[DIM_NAME];
        char surname[DIM_NAME];
    };

int main() {
    struct student my_student;

    printf("\nName: ");
    fgets(my_student.name, DIM_NAME, stdin);
    printf("\nSurname: ");
    fgets(my_student.surname, DIM_NAME, stdin);
    printf("\nID: ");
    scanf("%d",&my_student.ID_number);

    printf("\n\nStudent data: ");
    printf("%s %s with ID
%d\n",my_student.name,my_student.surname,my_student.ID_number);
}
```

As you can see, with the `fgets` you have control over the dimension of the string. The field with `stdin` is to specify that the input will be from the standard input (the keyboard). You will see later that the `fgets` can be used to read data from *file*.

You have just seen that **structures can contain arrays**, furthermore, you can also have **arrays of structures** and **structures that contain structures** as shown in the following examples:

```c
#include <stdio.h>
#define DIM_NAME 20
#define DIM_STUD 100

struct student {
        int ID_number;
        char name[DIM_NAME];
        char surname[DIM_NAME];
    };

int main() {
    struct student students[DIM_STUD];
    int n_students;
    char enter;

    printf("\nHow many students? ");
    scanf("%d",&n_students);
    printf("\nAdd students\n");
    for(int i=0;i<n_students;++i) {
        printf("\nStudent - %d: ", i+1);
        printf("\nName: ");
        scanf("%c",&enter);
        fgets(students[i].name, DIM_NAME, stdin);
        printf("Surname: ");
        fgets(students[i].surname, DIM_NAME, stdin);
        printf("ID: ");
        scanf("%d",&students[i].ID_number);
    }

    printf("\n\nStudents:\n");
    for(int i=0;i<n_students;++i) {
        printf("%s %s - ID:
%d\n",students[i].name,students[i].surname,students[i].ID_number);
    }
}
```

Note the usage of `scanf("%c",&enter);` that allows ignoring the character `\n` (in the buffer after the user inserts the number of students and later the student ID) as input in the following `fgets` call.

```c
#include <stdio.h>
#define DIM_NAME 50
#define DIM_STUD 1000
#define DIM_COURSE 50

struct course {
    int grade;
    char name[DIM_NAME];
};

struct std {
    char name[DIM_NAME];
    char surname[DIM_NAME];
    struct course esame[DIM_COURSE];
};

int main() {
    struct std students[DIM_STUD];
    int n_students,n_ex,sum[DIM_STUD]={0};
    float mean[DIM_STUD]={0};
    char enter;

    printf("How many students? ");
    scanf("%d",&n_students);
    scanf("%c",&enter);
    for(int i=0;i<n_students;++i) {
        printf("Student - %d", i+1);
        printf(" Name: ");
        fgets(students[i].name, DIM_NAME, stdin);
        printf("Surname: ");
        fgets(students[i].surname, DIM_NAME, stdin);
        printf("How many exams: ");
        scanf("%d",&n_ex);
        scanf("%c",&enter);
        for(int j=0;j<n_ex;++j) {
            printf("Exam - %d",j+1);
            printf(" Name exam: ");
            fgets(students[i].esame[j].name, DIM_NAME, stdin);
            printf("Grade: ");
            scanf("%d",&students[i].esame[j].grade);
            scanf("%c",&enter);
            sum[i]+=students[i].esame[j].grade;
        }
        mean[i]=(float)sum[i]/n_ex;
    }
    printf("\n\nList of students\n");
    for(int i=0;i<n_students;++i) {
        printf("%s %s - mean:
%.1f\n",students[i].name,students[i].surname,mean[i]);
    }
}
```

Level 7: pointers

Pointers in C are everywhere and may be used for almost everything. *Pointers are variables that can store the address of another variable*. As you will see, pointers are used to access and to interact with arrays, to change the content of variables when they are passed to functions, to use files, to manage dynamic memory allocation and to cope with linked lists. To use a pointer you need to declare it as follows: `int *my_pointer;`. The declaration contains a name to be assigned to the pointer where the name comes after the symbol `*`. Furthermore, with the declaration, you have to specify the type of the variable pointed. There are no restrictions to the type of variables that you may need to point.

Let's see ho to use pointers:

```
#include <stdio.h>
int main() {
  int n=10;
  int *p;

  p=&n;
  printf("%d %d",n,*p);
}
```

In this example you see the pointer declaration `int *p;` and successively the pointer initialization `p=&n;`. This last instruction allows creating a link between the pointer `p` and the variable `n`. Now, `p` contains the address of the variable `n`, and you can access the value of the variable `n` using `*p` (meaning the content of the variable pointed by `p`).

You may access a structure using a pointer:

```
#include <stdio.h>
#define DIM_NAME 20
struct student {
        int ID_number;
        char name[DIM_NAME];
        char surname[DIM_NAME];
    };

int main() {
    struct student my_student;
    struct student *my_pointer;

    my_pointer=&my_student;
    printf("\nName: ");
    fgets(my_pointer->name, DIM_NAME, stdin);
    printf("\nSurname: ");
    fgets(my_pointer->surname, DIM_NAME, stdin);
    printf("\nID: ");
    scanf("%d",&my_pointer->ID_number);

    printf("\n\nStudent data: ");
    printf("%s %s with ID 
%d\n",my_student.name,my_student.surname,my_student.ID_number);
}
```

With the instruction `my_pointer=&my_student;` you create a link between the pointer ``my_pointer`and the variable`my_student`. As a consequence, you can use the pointer to add the information inside the components of the structure. To check if this is indeed the case just try to print the structure using the classic formalism.

The pointers represent the more important ingredient for the linked lists. Even in their more simple form, when you do not use the dynamic allocation of memory, linked list are essentially organized by using pointers. **Pointers in the linked lists are essential**. There are three main reasons that make the use of pointers in linked lists so important.

1. Differently from arrays, the elements of a linked list are not stored in adjacent blocks of memory, so the only way to know where the next element is located is to store its address in the previous element. As a consequence, every single element of a linked list must contain a pointer to a variable of the same type.
2. Your linked list, if all the elements are correctly linked, is identified by a pointer to its first element. You need to do not lose this pointer, otherwise, you will not have your list anymore.
3. If you want to access all the elements of the linked list you have to clearly detect the last element. So, you need to remember to make the pointer included in the last element of the list points to the **NULL** pointer.

Here an example of linked list with static allocation (all the elements of the list must be declared).

```
#include <stdio.h>
    struct list {
        int num;
        struct list *next;
    };
int main() {
    struct list val1,val2,val3;
    struct list *punt_list;

    val1.num=10;
    val1.next=&val2;
    val2.num=20;
    val2.next=&val3;
    val3.num=30;
    val3.next=NULL;
    punt_list=&val1;
    while(punt_list != NULL) {
      printf("%d ",punt_list->num);
      punt_list=punt_list->next;
    }
}
```

Despite the example is easy, you can observe the presence of all the previously discussed relevant points in this code.

There are several other reasons that make linked lists a very powerful tool in C. For example, if you want to add or to remove an element of a list you will see that this is a very simple task. All you need to do is to update the pointers of the involved elements. The same task is much more complicated when using arrays. Try to insert a new element into an array (in any position) or to remove it. You will discover that it is not easy at all since it will require to act with all the other elements, shifting them to left or to right depending you are removing or adding the element.

It is very important to understand that **arrays and pointers are strictly connected**. You can use an external pointer to manage an array by means of *pointer arithmetics*. So, if you have a situation like this:

```
int v[N];
int *v_ptr;

v_ptr = &v[0];
```

you can use the pointer `v_ptr` to interact with the array `v`. This last instruction `v_ptr = &v[0];` meaning that your external pointer points to the first element of the array.

```
#include <stdio.h>
int main() {
  int i,v[3]={1,2,3};
  int *v_ptr;

  for(i=0;i<3;i++) printf("%d ",v[i]); //1 2 3
  v_ptr=v; //v_ptr points to v
  //v_ptr = &v[0]; //this is equivalent to v_ptr=v;
  *(v_ptr+1)=10;

  for(i=0;i<3;i++) printf("%d ",v[i]);
    //You will see that the second element of v has be changed to 10
}
```

This last example showing how to change elements of the array `v` using the external pointer `v_ptr` and pointer arithmetics. In particular, the instruction `*(v_ptr+1)=10;` allows to access and change the value of the second element of the array `v`. This is possible because `v_ptr` points to the first element of `v` (see `v_ptr=v;`) and `v_ptr+1` says to move to the next element using an offset equals to 1. The number 1 refers to 1 block of memory, where its size depends on the type you use to declare the array. It will be 4 bytes if the array is of integers, 1 byte is the arrays is of chars. The operator `*` allows accessing the value of the element.

The name of the array `v` contains the address of its first element, therefore the previous expression `v_ptr = &v[0]` is equivalent to `v_ptr = v`. The only, but very important difference is that `v` is a constant and you can not change it (so, you can not write `v+1` as you can do with pointers).

Finally, it should be clear that, in this context, the expressions `v[i]` and `*(v_ptr+i)` are equivalent.

You can also use the increment `v_ptr++` (or decrement `v_ptr--`) operator to move to the following (previous) elements of the array. However, keep in mind that in this last case you are changing the pointer and at the end of the instruction (or set of instructions) the pointer `v_ptr` is not pointing to `v` anymore unless you move it back `v_ptr=v;`.

Level 8: functions

Functions are subprograms, blocks of code that can be reused when a specific functionality is needed. Functions represent the first step towards *modularity*. You may use standard functions or implement your own. A function should act as a *black box*, should give some functionalities hindering the implementation.

When using a function, you need to pass (most of the times) some parameters and expecting (most of the times) a value returned.

If you want to create a function, you have to **declare** it, **call** it in the calling function (can be different from the *main*) and **define** it. See this trivial example:

```
#include <stdio.h>
int sum(int,int); //function declaration

int main() {
  int a=1,b=1,s;

  s=sum(a,b); //function call
  printf("\nThe sum is %d\n",s);
}

int sum(int c,int d) { //function definition
  int s;
  s=c+d;
  return s;
}
```

The declaration `int sum(int,int);` allows the compiler to know how the function will work, it sets the name `sum`, specify the type of the returned value `int` and also specify the number and type of parameters `(int,int)`. If you like, for clarity, you can also indicate a name for the two parameters, however, the names will be ignored.

The call `s=sum(a,b);` in the *main* allows to refer to the function and execute the instructions contained in the definition. The variables `a` and `b` are called *actual parameters*.

The definition, starting with `int sum(int c,int d)` and followed by a block of instructions, contains the set of operations to be performed when the function is called. The variables `c` and `d` are called `formal parameters` and are *local* variables, can not be seen or accessed outside the function, and contain the value of the *actual parameters*. C passes the parameters by *value*, so **you are working on a copy** of the *actual parameters*, meaning you can not change their content inside the function. What happens is that the *formal parameters* are initialized by the value of the *actual parameters*. *Call* and *definition* need to share the **same number**, the **same type** and the **same order** for the parameters.

If, for some reasons, you want to change the value of the *actual parameter* from the function, you need to use a pointer. See this example:

```
#include <stdio.h>
void swap(int *, int *);
int main() {
  int x=0,y=1;

  printf("\nBefore the call, x is %d and y is %d",x,y);
  swap(&x, &y);
  printf("\nAfter the call, x is %d and y is %d\n",x,y);
}

void swap(int *x, int *y) {
  int tmp;
  tmp=*x;
  *x=*y;
  *y=tmp;
}
```

As you can see, using the pointers you can change the values of `x` and `y` defined in the *main* function. Note that this works even if the pointers have different names, there is not any relationship between `x` and `y` contained in the *main* and `x` and `y` declared local inside the function `swap`.

I strongly suggest avoiding *global variables* as an implicit mechanism to pass parameters to functions!

One important point to consider when using functions is the *scope* of variables. The *scope* of a variable is the block of a program where the variable is seen and can be used. You can learn more about the *scope* with the exercises contained in the slides and in the code you can access from the main page of the course (https://matteogithub.github.io/teaching/).

It is very common to use functions that have to work on arrays. Obviously, you can easily pass a single element of an array (`v[i]`) to a function. That's really not problematic at all. But, if you need the function to work on the whole array you have to use another strategy. Unfortunately, you can not pass an array to a function, this is because it would mean to pass all the elements and therefore to make a copy of the array as a whole. How to solve this problem? Well, you need to pass the *name* of the array (`v`) to the function. Doing like that is equivalent to pass the pointer to the first element of the array, and that's really all you need. Once the function knows where the array starts you can move to the following elements using the external pointer (the formal parameter, which will be initialized with the address of the first element of the array). Remember that your function needs to know the dimension of the array, so don't forget to pass this information to the function during the call.

Now let's see a simple example, where you have to define a function to compute the *min* of an array of integers.

```
#include <stdio.h>
#define DIM 10

int compute_min(int *, int);

int main() {
  int n,v[DIM],i,min;

  do  {
    printf("Dimension of the array: \n");
    scanf("%d", &n);
  } while(n<1 || n>DIM);

  printf("Add the %d elements.\n",n);
  for(i=0;i<n;i++)  {
      printf("Element with index - %d: ",i);
```

```
        scanf("%d",&v[i]);
    }

    min=compute_min(v, n);
    printf("\nThe min is: %d\n", min);
  }

  int compute_min(int *v, int n) {
    int i,min=*v;

    for(i=0;i<n;i++) {
      if(*(v+i)<min)
        min=*(v+i);
    }
    return min;
  }
```

As you can see, the call of function `compute_min` requires two actual parameters: the array `v` and its dimension `n`. The pointer `v`, the formal parameter, declared in the function `compute_min` is a copy of `v` (array declared in the `main`) and points to the address of the first element of the array `v`. It is not a matter of names, the two `v` are completely different things that, however, point to the same block of memory (`&v[0]`).

See this other example, which should help to understand this latter point.

```
#include <stdio.h>
#define DIM 10

float compute_mean(int *, int);

int main() {
  int n,v[DIM],i;
  float mean;

  do  {
    printf("Dimension of the array: \n");
    scanf("%d", &n);
  } while(n<1 || n>DIM);

  printf("Add the %d elements.\n",n);
  for(i=0;i<n;i++)  {
      printf("Element with index - %d: ",i);
      scanf("%d",&v[i]);
  }

  mean=compute_mean(v, n);
  printf("\nThe mean is: %.1f\n", mean);
}

float compute_mean(int *v, int n) {
  int i;
```

```
    float mean=0;

    for(i=0;i<n;i++,v++) {
      mean+=*v;
    }
    return mean/n;
  }
```

As you can see, you can write `v++` in the function `compute_mean`, without getting an error, because the pointer `v` declared inside the function is a copy of the constant pointer `v` that you can use inside the `main` function.

Level 9: file

...

Level 10: memory management and lists

A *(singly) linked list* is a data structure composed by a set of homogeneous elements (all of the same type: a user-defined *struct*) which is implemented by the use of *pointers*. Here you will see how to create and manage a *linked list* where its elements are not explicitly declared but they are created during the execution of the program.

There are few but essential points that it is very important to remember when using *linked list*:

1- every time you need to create an element you need to allocate the necessary memory for it. You can do this using the `malloc` function (including the library `stdlib.h`). The `malloc` returns a pointer and needs to know the size of the element. Here an example of call for a user-defined *struct* named *my_list*: `malloc(sizeof(struct my_list))`. See the following example used to allocate the memory needed for an element of the list `my_list`:

```
struct my_list * p;
p=(struct my_list *)malloc(sizeof(struct my_list));
```

Now, you can use the pointer `p` to access the new allocated memory location.

2- every single element of your list will be linked to the following one by using an *internal* pointer. As a consequence, when you define your *struct* do not forget to include a *pointer* (as a component) to this *struct*:

```
struct my_list {
    int num;
    struct my_list *next;
};
```

If you forget to include this pointer (here named `next`) you will not be able to link the elements of the list.

3- if you want to use your list, you still need to remember to make your last element (by means its internal pointer) points to `NULL`. This is the way you can access and browse its elements and stopping when the list is ended.

4- where is your list? Well, your list will simply be a pointer to its first element. If you have correctly linked all the elements and if its last element points to `NULL`, you only need a single *external pointer* to keep track of the starting point. **The more important thing for you, therefore, is: <u>do not lose this pointer, otherwise, you will not have your list anymore!</u>** Also, as a consequence, you will not be able to de-allocate (make free) the memory, and that locations will not be available during the execution of the program.

These are essentially the things you need to know when managing a *linked list*. Anyway, you will not have a full understanding of this topic since you will not try using this data structure by yourself. See all the examples reported in the slides and don't forget to read a good book!

Important note: it is very common to find the use of **typedef** in C. Therefore, even if we have not used it, it may be important to know what it means. With **typedef** you can create an *alias*. For example, you can use an *alias* for your structure as follows:

```
struct my_list {
    int num;
    struct my_list *next;
};

typedef struct my_list list;
list list_01, list_02;
```

Or, if you prefer:

```
typedef struct my_list {
    int num;
    struct my_list *next;
} list;

list list_01, list_02;
```

As you can see, in this way you can avoid repeating the name `struct` for every single declaration.

There is not a built-in dynamic array in C. Nevertheless, you can still use the `malloc` or `calloc` functions to get the result. See the following (simple) example:

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

int main () {
    int i,n;
    int *ptr;
    printf("Inserisce dimensione array: ");
    scanf("%d",&n);

    ptr=(int*) calloc(n, sizeof(int));
    //ptr=(int*) malloc(n * sizeof(int));

    for(i=0;i<n;i++) {
        printf("\nInserisce valore: ");
        //scanf("%d",&ptr[i]);
        scanf("%d",ptr+i);
    }

    for(i=0;i<n;i++)
        printf("%d ",*(ptr+i));
        //printf("%d ",ptr[i]));

    free(ptr);
}
```