1. Pointers

As Kernighan and Ritchie state, “a pointer is a variable that contains the address of a variable”. They have been created to allow us to directly access requested variables without working with a copy of them. Pointers can be very useful in many tasks such as modifying numerous inputs of a function or navigate in array data. They also save memory as data are not copied but just pointed to (like the table of content of an article that refers to sections without copying the whole content of these sections).

1.1. Declaration of a pointer. The declaration of a pointer variable is done by adding the symbol * in the declaration of your variable. For example, the following lines of code define a pointer of an integer variable.

```c
int * pt_int;
```

The * can also be placed side by side to the type (here int) as follows:

```c
int* pt_int;
```

The above declarations are equivalent.

It is important to define correctly the type of variable we are going to point to. Indeed, a pointer contains the address in memory of a variable and integer or float are not written on the same number of bytes. So using a pointer associated to an integer to point to a float variable will only create error when compiling/executing the program.

Note that declaring a pointer does not initialized it, meaning it does not contains any memory address. In order to not let pointers undefined it is possible to initialize a pointer to zero as follows:

```c
int * pt_int= NULL;
```

The following section describes how to assign the address of a variable to a pointer.

1.2. Definition of a pointer by assigning an address. The memory address of a variable is the number in the memory of the first byte in which the variable is stored. As variable can be stored in multiple bytes, we remind that is important to correctly define the type of pointer you are using. The address of a variable can be accessed with the operator & known as address-of pointer. Here is an example that create a pointer that points to an integer variable:
// Define integer i
int i = 2;
// Declare pointer of an integer
int *pt_int;
// Initialize pointer ip with the memory address of integer i
pt_int = &i;

In the above example, the pointer ip now contains the address of the integer i. This address has nothing to do with the value of i apart that i is an integer. The type of the pointer will be important when reading or modifying the value stored in the memory address contained by the pointer. It allows the program to know how many bytes should be read/modify. Note that a pointer is not an integer, it contains an address written with hexadecimal. It can be displayed with the function printf using the option "\%p.

1.3. Dereferencing. After assigning the address of a variable to a pointer, one can access the actual value of the variable. This operation is known as dereferencing and used the operator *. This operator is different than the one used when declaring pointer or when doing a multiplication of two numbers. It only uses one operand that is a pointer. When applied to a pointer pt, *pt refers to the value stored in the address pt. The following example shows how to use this operator.

```
int a = 2, b = 3;
int *pt_int;
pt_int = &a;
b = *pt_int; // (reading the value), b=2
*pt_int = 4; // (modifying the value) a=4
```

1.4. Functions and pointers. Pointers can be used as input of functions. It allows functions to have access to the memory address where the input is stored so the function can now modify the value of the input. It becomes very handy when a function needs to modify different inputs that can be of different type. Here is an example of such a function.

```
void add_one(int *, double *);

int main(void){
    int a = 2,;
    double b = 3.0;
    add_one(&a, &b);
    // now a=3 and b=4.0
}

void add_one(int * x, double * y){
    *x = *x + 1;
    *y = *y + 1.0;
}
```

Note that when calling the function, we give a memory address via the operator &. In the function, the value stored at this address can be modified using the dereferencing operator *.
1.5. **Pointer Arithmetic.** In addition of working with the value stored at an address in memory, a pointer can also be used to work with the address in memory. Such feature is called pointer arithmetic and allows a pointer to look at value stored at a different address that the one used to define the pointer. Here is an example of such operation:

```c
int i=2;
int *pt_i;
pt_i = &i; // ip points to integer i
*ipt_i =4;  // Change value of i
*(pt_i+1)=5; // Set value on next memory address to 5

//but what was stored in pt_i+1?
```

The operation *(pt_i+1) tells the program to look for the value store at the address pt_i+1. This address depends of the type of pointer you are using. If using a pointer of integer, this address is the one of pt_i plus four bytes. If the pointer is associated to a double, pt_i+1 refers to the address of pt_i plus eight bytes.

It is important to understand that operating/working on the address memory associated to a pointer can have disastrous consequences. Indeed, in the previous example nothing guarantee that an integer is defined in the bytes associated to pt_i+1, in fact nothing guarantee that this address memory is used by your program. So changing the value represented by these bytes can change the value of a variable of your program or also change data associated to your computer (see buffer overflow). As a consequence, such operation should only be used when one is sure that he ends up on the address of a known variable of the C program. This will be the case when we create array with a given dimension (or multi dimension).

To end this section, we note that there exists two other ways of dereferencing a pointer. The following example displays how it can be done.

```c
int i=2;
int *ip;
ip = &i; // ip points to integer i
*ip=4;  // Change value of i
*(ip+0)=4; // Equivalent to *ip=4
ip[0]=4;  // Equivalent to *ip=4
```

The number 0 in the last two lines tell the program if the address used should be modified using a pointer arithmetic operation. For instance, ip[1] is equivalent to *(ip+1).

## 2. Array in C

Arrays in C programming language are a data structure that stores multiple variables of the same type. Array can be used to declare vectors or multidimensional matrices of integers, characters, real (float or double) but also of pointers.

### 2.1. Definition and initialization.

To declare an array, the programmer needs to specify the type of the variable stored in the array, the name of the array and the size of the array. Such declaration has the following form:

```c
type_variable name_array[size_array];
```

Here is an example where we declare an array of dimension 10 that contains integers.
int tab_int[10]; //array of 10 integers

One can also declare an array of pointers or a pointer to an array as follows.

int *tab_int[10]; //pointer to an array of 10 integers
int (*tab_int)[10]; //array of 10 pointers of integers.

There is two ways to assign a value to the elements of an array.

• Initialize all of the array’s elements during the declaration of the array. It is done using curly brackets as follows.

//array whose value is equals to their index
int tab_int[10]={0,1,2,3,5,5,6,7,8,9}

Note that elements with missing value are set to zero. As a result, this code initializes the array tab_int to zero.

int tab_int[10]={0}

• Assign a value to a specific/single element of the array.

tab_int[0]=2; // Indexation starts with zero in C
tab_int[5]=8;

Remember that the indexation of an array starts at zero.

2.2. Connection with pointers. When declaring an array tab_int of \( n \) integers, the compiler allocates \( n \) consecutive blocks of memory of an integer size (4 bytes). The address of the first block of memory is stored in the name of the array such that tab_int can be used as pointer. For instance, tab_int[4] and *(tab_int+4) both refers to the value of the fourth element of the array tab_int. It allows the programmer to set or modify the value of an element with the two following ways:

tab_int[5]=2; //Set the 6th element to 2
*(tab_int+5)=2; // equivalent to the above line

The difference between an array and a pointer is that the address memory associated to an array cannot be changed. On the other hand, the address stored in a pointer can be modified during a program. The programmer just needs to take care that the pointer keep pointing to the same type of variable (integer, double, etc.).

2.3. Outlook: multidimensional array. The C programming language can consider multidimensional array. It is done by adding multiple dimension between square brackets as follows.

int tab_int[10][5]; // integer matrix with dimension (10,5)

The initialization of a multidimensional array can be done by fixing a specific index for each dimension and set a value. It can also be done when declaring the array. Here in an example for a 2D array:
C PROGRAMMING LANGUAGE: POINTERS, ARRAYS, OPERATORS.

```c
int tab[3][4] = { {4, 6, 8, 10}, // First row
                  {1, 2, 3, 4},   // Second row
                  {-10, 100, -100, 0} // Third row
               };
```

and here an example for a 3D array:

```c
int tab[2][4][3] = {
    { // Set first index to 0
        {1, 2, 3},  // tab[0][0][:]
        {4, 5, 6},  // tab[0][1][:]
        {7, 8, 9},  // tab[0][2][:]
        {10, 11, 12} // tab[0][3][:]
    },
    { // Set first index to 1
        {-1, -2, -3}, // tab[1][0][:]
        {-4, -5, -6}, // tab[1][1][:]
        {-7, -8, -9}, // tab[1][2][:]
        {-10, -11, -12} // tab[1][3][:]
    }
};
```

2.4. **Remark on array memory allocation.** Declaring an array as shown previously uses memory space in a region called stack. Although the stack gives a fast access to data, its size is limited. As a consequence, only small array should be declared as previously. We will introduce later in the course how to allocate large array on the heap using the function malloc. It is possible to avoid the use of the function malloc by using the attribute static when declaring your array. However such memory allocation is permanent unlike malloc that uses a dynamic memory allocation.

3. **Operators**

In C language, operators can be used to do operation between variable of a given type such as integer, logical, float, bits or even string of characters. The output of an operator is not necessarily of the same type than the input(s). In this section, we focus on introducing some operators that are essential to scientific programming: arithmetic, assignment, relational and logical operators.

3.1. **Arithmetic operators.** Here is a list of the arithmetic operators available in C.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>a+b</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
<td>a-b</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>a*b</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>a/b</td>
</tr>
<tr>
<td>%</td>
<td>Remainder of euclidean division</td>
<td>5%3 returns 2</td>
</tr>
</tbody>
</table>
It is also possible to compute exponentiation in C with the function pow. This function is included in the library math.h. The input and output of the function are double, it can be called as follows.

```c
#include <math.h>
int main(void){
    double a=2;
    double b =5;
    double c=pow(a,b);  //Compute 2.0^5.0=32.0
}
```

**Remark:** It is important to note that C only allows the implementation of type operations. Meaning that the above operators can be applied to a certain type of variable like integer or double but not to an array (one or multidimensional). Unlike C++ that allows to multiply two arrays (scalar product) or an array by a scalar, such operations requires the creation of a function in C. The function will be defined such that the requested operation (addition, multiplication, etc.) is done on each elements of your array.

### 3.2. Assignment operators
An assignment operator modifies the value of an existing variable. The modification can involve addition, subtraction, multiplication and division. Here is a description of these assignment operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Add one to a variable</td>
<td>i ++</td>
</tr>
<tr>
<td>--</td>
<td>Subtract one to a variable</td>
<td>i --</td>
</tr>
<tr>
<td>+=</td>
<td>Add the value on the right of the operator to the variable on the left of the operator</td>
<td>a+=2 means &quot;a=a+2&quot;</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract a value to a given variable</td>
<td>a-=4 means &quot;a=a-4&quot;</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply a given variable</td>
<td>a*=b means &quot;a=a*b&quot;</td>
</tr>
<tr>
<td>/=</td>
<td>Divide a given variable</td>
<td>a/10.0 means &quot;a=a/10.0&quot;</td>
</tr>
<tr>
<td>%=</td>
<td>Compute the remainder of an euclidean division</td>
<td></td>
</tr>
</tbody>
</table>

The operator %= can be used to compute the remainder of an euclidean division.

### 3.3. Relational operators
These operators are used to compared the value of different variable. They return a logical (true or false). The following table that describes the different relational operators available in C.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>strictly smaller than</td>
<td>1 &lt; 2 returns true</td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly larger than</td>
<td>1 &gt; 2 returns false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>smaller or equal than</td>
<td>1.5 &lt;= 1.0 return false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>large or equal than</td>
<td>2.5 &gt;= 2.0 returns true</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td>2 == 3 returns false</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>2! != 3 returns true</td>
</tr>
</tbody>
</table>
3.4. **Logical operators.** Unlike the relational operators whose input are integer or real, a logical operator is applied to logical (true/false or 1/0). There is three logical operator that represents the operation and, or and not. These operators can be implemented as follows.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>AND operator</td>
<td><code>(1 &lt; 2) &amp;&amp; (3 &lt; 2)</code> returns false</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>!</code></td>
<td>NOT operator</td>
<td><code>!(3 &lt; 2)</code> returns true</td>
</tr>
</tbody>
</table>

In addition to the previous operators, I would like to introduce the conditional operator. This operator takes a logical as input and execute different single instruction depending of the value of the logical. The form of this operator is the following.

```
(condition) ? (instruction if true) : (instruction if false);
```

It can be seen as a if-else statement that are introduced in the section 4. Here is a possible definition of the absolute value of a number using the conditional operator:

```
(x > 0) ? (x) : (-x);
```

4. **Control statement: if/#if/#endif**

### 4.1. **Statement if-else.** A if statement is going to evaluate if a logical, called condition, is true or false. Depending of the value of the logical it tells the program to execute different instructions. The general structure of a if statement is the following.

```
if (condition1){
    // instructions if condition1 is true.
}
else if (condition2){
    // instructions if condition1 false and condition2 true.
}
else{
    // instructions if condition1 and condition2 are false.
}
```

The statements else if and else are optional. If nothing should be done when condition1 is false, one can remove the block else if and else from the above example. Note that adding a block "else if" is equivalent to adding a statement if in the block else.

**Remark:** Similar preprocessor if statement can be used in C. We can divided these statements into three groups:

- Check if a macro is defined.

```
#define DEBUG
printf("I am in DEBUG mode");
#endif
```
• Check if a macro is not defined.

```c
#ifndef DEBUG
#define DEBUG
#endif
```

• Do a classic if statement depending of a macro’s value.

```c
#if MACRO == 1
// instructions
#elif MACRO == 2
// instructions
#else
// instructions
#endif
```

Attention “else if” is denoted ”elif”. Note that these instructions are done by the preprocessor before the rest of the code is compiled. As a consequence, the condition tested by #if should involve macro and not classic C variable.

### 4.2. Loop for

The general form of a for loop is as follows:

```c
for (initialization; condition; increment){
// loop instructions to do when condition is true
}
```

Its operation can be summarized in these three steps:

- Step 1: Execute the initialization.
- Step 2: If the condition is false, the program continues after the loop for (step 3 not executed).
- Step 3: If the condition is true, the program executes the instruction in the loop then execute the increment. Then is goes back to step 2 (testing the condition).

Here is an example of loop that display the integers from 0 to 9:

```c
for (int i = 0; i < 10; i++){
    printf("i = %d \n", i);
}
```

Note that originally, it was not possible to declare the type of the initialization in the loop for (only in C++). The declaration ”int i” had to be done before the loop for. It is worth mentioning that the initialization, condition and increment are all optional. However the semi columns are required. If no condition is given, the program assumes that the condition is true and sets it to 1. You can use this to create infinite loop as follows:

```c
for (; ; ){
    // do nothing (or something that is repeated infinitely)
}
```

If you test this example in a terminal, you can stop your program pressing Ctrl+C (or by closing the terminal).
4.3. **Loop while.** The while loop has the following structure:

```c
while (condition) {
    // loop instructions to do when condition is true
}
```

It allows the program to execute a set of instructions while a given condition is true. The program first checks if the condition is true. If it is, it executes the instructions. As soon as the condition is false, the program continues after the loop. A while loop is preferred to a for loop when the number of iterations is not known in advance.

As the for loop, a while loop can be skipped and never be executed by the program if the initial condition is false. It is possible to implement a loop that executes the set of instructions at the first iteration. It is called a do-while loop and has the following structure:

```c
do {
    // loop instructions
} while (condition);
```

If the condition is true after the first iteration of instructions, the instructions are executed a second time and so on.

4.4. **Command break and continue.** The C programming language supports loop control statements. These statements are used to change the execution of a loop from its normal sequence. We focus here on two of these special commands that are:

- **break.** Terminate the loop so the program executes the instructions that follow the loop.
- **continue.** Force the program to go to the next iteration of the loop.

These commands are usually associated with a if statement. Here is an example that display on the terminal the integers 0 to 4 but omit to write the integer 3.

```c
for (int i=0, i<5, i+=1){
    if(i==3) {
        continue;
    }
    printf("i = %d \n",i);
}
```

4.5. **On the use of curly brackets.** In the description of the control statement if/for/while, the examples provided use curly brackets to signal the beginning and ending of a statement. It delimits the instructions that should be done when a statement is true. In the case where a statement is followed by a unique instruction, the curly brackets can be omitted. Here is an example:

```c
for (int i=0, i<5, i+=1)
    printf("i = %d \n",i);
```

However using curly brackets helps the clarity of your code and also help to avoid mistake so it is recommended to use them.