C Language

C is a general-purpose, procedural, compiled, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. The UNIX operating system, the C compiler, and essentially all UNIX application programs have been written in C. C was adopted as a system development language because it produces code that runs nearly as fast as the code written in assembly language.

Preliminaries

You need the following two software tools available on your computer :

(a) Text Editor (examples of few a editors include Windows Notepad, OS Edit command, EMACS, and vim or vi) and

(b) The C Compiler (gcc)

- check whether GCC is installed on your system by entering the following command from the command line : `$ gcc -v`

If you use your personal machine, and GCC is not installed, then you will have to install it yourself, or try the exercises using the on-line compiler available at :

- **CodingGround** (https://www.tutorialspoint.com/codingground.htm)
- Scroll down until « Online IDEs »
- Choose C among the options
- You can then write your C code, compile and execute it. The result is displayed in the terminal zone, at the bottom of the page.
- If you use CodingGround, please save your exercices in .c files so that the professor can validate your solutions.

Hello Word in C

```
#include <stdio.h>
int main(void){
    /* my first program in C */
    printf("hello, world\n");
    return 0;
 }
```

Compile and Execute C Program

- Open a text editor and add the above-mentioned code.
- Save the file as *hello.c*
- Open a command prompt and go to the directory where you have saved the file.
- Type `gcc hello.c -o hello.exe` and press enter to compile your code.
- If there are no errors in your code, the command prompt will take you to the next line and would generate *a.exe* executable file.
- Now, type `hello.exe` to execute your program.
- You will see the output *"hello, world"* printed on the screen.

Part I - The C Language : Basic building blocs

The Basics

- The files you create with your editor are called the **source files** and they contain the program source codes.
- The source files for C programs are typically named with the extension « .c »
- A C program basically consists of the following parts :
  - Preprocessor Commands
  - Functions
  - Variables
  - Statements & Expressions
  - Comments
- In a C program, each individual **statement** must be ended with a **semicolon**.

Comments

One-line comments begin with //, and multi-line comments should appear between the delimiters /* and */.

Data Types

Basic Types : They are arithmetic types and are further classified into: (a) integer types (char, int, short, long) and (b) floating-point types (float, double, long double).
Enumerated types :They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
The type *void* :The type specifier *void* indicates that no value is available.
Derived types : They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. Variables can be initialized (assigned an initial value) in their declaration.

```
type variable_list;
```

with an initial value :

```
type variable_name = value;
```

Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language :

- Inside a function or a block which is called **local** variables.

```
#include <stdio.h>
int main () {
  /* local variable declaration */
  int a;
  /* actual initialization */
  a = 10;
  printf ("value of a = %d", a);
  return 0;
}
```

- Outside of all functions which is called **global** variables.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main () {
  /* actual initialization */
  g = 20;
  printf ("value of g = %d\n", g);
  return 0;
}
```

- In the definition of function parameters which are called **formal parameters** (see Functions).

Input & Output

Input refers to feed some data into a program. An input can be given in the form of a file or from the command line. **Output** refers to display some data on screen, printer, or in any file.
C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The **int scanf(const char *format, ...)** function reads the input from the standard input stream *stdin* and scans that input according to the **format** provided.
The **int printf(const char *format, ...)** function writes the output to the standard output stream *stdout* and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Example :

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

   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;
}
```

Defining Constants

Constants refer to fixed values that the program may not alter during its execution (to the contrary to variables). There are two simple ways in C to define constants :

- Using **#define** preprocessor.
- Using **const** keyword.

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
```

```
#include <stdio.h>

int main() {

   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;
```

Logical Operators

&& : the AND operator

| | : the OR operator

! : the NOT operator

Assignment Operators

= : Simple assignment operator

+= : Add AND assignment operator (C += A is equivalent to C = C + A).

%= : Modulus AND assignment operator.

« ?: » : Conditional Expression. If Condition is true ? then value X : otherwise value Y

Decision making

Allows to specify one or more conditions to be evaluated by the program, along with statements to be executed if the condition is true, and optionally, other statements to be executed if the condition is false.

Using if - else

```
if  "condition"
    printf ("Hello");
else
    printf("World");
```

Using switch - case

```
#include <stdio.h>
int main()
{
   int x = 2;
   switch (x)
   {
       case 1: printf("Choice is 1");
               break;
       case 2: printf("Choice is 2");
                break;
       case 3: printf("Choice is 3");
               break;
       default: printf("Choice other than 1, 2 and 3");
                break;
   }
   return 0;
}
```

Loops

while: Repeats a statement or group of statements while a given condition is true. Tests the condition before executing the loop body.

for: Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Example 1

```
int main()
{
  int i = 0;
  for(i = 0; i < 3; i++)
  {
    printf("loop ");
  }
  return 0;
}
```

Example 2

```
int main()
{
  int i = 0;
  while(i < 3)
  {
    printf("loop");
    i++;
  }
  return 0;
}
```

Exercises

1) Calculator – Create a program that implements the four basic operations in Math.

2) Dice – Create a program that simulates the roll of a dice. Have the program output the rolls of six dices repeatedly until at least four of the dices roll the same number.

You may reseaarch the **rand()** function in <stdlib.h>

3) Transpose matrix – Create a program that transposes a matrix. The user should enter the matrix dimensions, which should not exceed 6x6. Example :

Original matrix

a	b	c	d	e	f
g	h	i	j	k	l
m	n	o	p	q	r

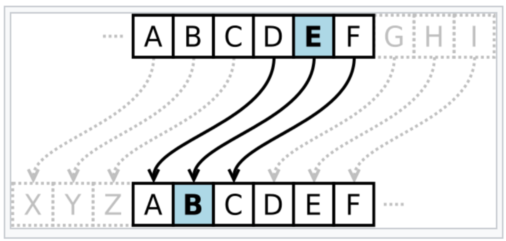
Transposed

a	g	m
b	h	n
c	I	o
d	j	p
e	k	q
f	l	r

4) Palindrome – Checks if the string entered by the user is a palindrome. That is that it reads the same forwards as backwards like "racecar".

You may research how to use strings in C

5) Caesar cipher



In cryptography, a Caesar cipher is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plain text is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on. The method is named after Julius Caesar, who used it in his private correspondence.

Example :

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Write an encryption algorithm that can **encrypt** and **decipher** a message using **modular arithmetic**, i.e. by first transforming the letters into numbers, according to the scheme, A → 0, B → 1, ..., Z → 25. The input values are the message and the key (the shift factor) and the output are both the ciphertext and the deciphered message.

Part II – The C Language : “Advanced” concepts

Pointers

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Example :

```
#include <stdio.h>
int main () {
   int  var1;
   printf("Address of var1 variable: %x\n", &var1  );
   return 0;
}
```

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is :

```
type *var-name;
```

The actual data type of the value of all pointers is the same, a long hexadecimal number that represents a memory address. Previously, *type* is the pointer's base type. Examples :

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
```

To **use pointers**, there are a few important operations :

- define a pointer variable
- assign the address of a variable to a pointer and finally
- access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

The following example makes use of these operations :

```
#include <stdio.h>
int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

Output :

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is *main()*.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

The general form of a **function definition** in C programming language is as follows :

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function is composed of the following parts :

- **Return Type** – A function may return a value. The *return_type* is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword *void*.
- **Function Name** – This is the actual name of the function.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – Contains a collection of statements that define what the function does.

In the following **example**, *max* takes two parameters and returns the maximum value between the two :

```
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

To use a function, you will have to **call** that function. When a program calls a function, the program control is transferred to the called function. To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, you can store the returned value.

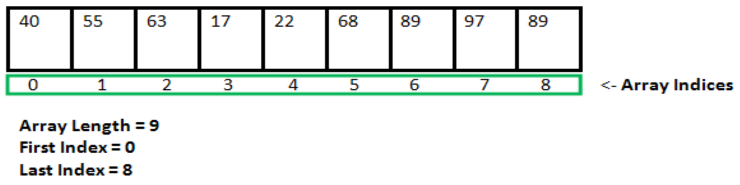
```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {
```

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. Function parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways in which arguments can be passed to a function :

- **Call by value** : This method copies the actual value of an argument into the parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- **Call by reference** : This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. The following example illustrates a call by reference.

Arrays

An array is collection of items stored at continuous memory locations. The idea is to declare multiple items **of same type** together.



Array declaration: In C, we can declare an array by specifying its type and size, or by initializing it, or by both.

```
// Array declaration by specifying size
int arr[10];

// Array declaration by initializing elements
int arr[] = {10, 20, 30, 40}

// Compiler creates an array of size 4.
// above is same as  "int arr[4] = {10, 20, 30, 40}"

// Array declaration by specifying size and initializing elements
int arr[6] = {10, 20, 30, 40}

// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as 0.
// above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"
```

Accessing Array Elements: Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1. Following are few examples.

```
int main(){
  int arr[5];
  arr[0] = 5;
  arr[2] = -10;
  arr[3/2] = 2; // this is same as arr[1] = 2
  arr[3] = arr[0];
  printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);
  return 0;
}
```

Output :

5 2 -10 5

Comparing arrays and pointers: If you write just `arr`, you get a pointer. This is a pointer to the first box of the array. Test it :

```
int arr[5];
printf("%d", arr);
```

Output : we see the address where `arr` is.

Note that as a table is a pointer, we can use the symbol * to know the first value:

```
int arr[5];
arr[0]=10;
printf("%d", arr);
```

Output : 10

Passing an array to a function: You will often need to display all the contents of an array. Why not write a function that does that? You will have to send 2 information to the function: the array (meaning the address of the array) and its size. Indeed, the function must be able to manipulate an array of any size. But since the function does not know the size of the array, you have to send it too. As said just before, an array can be considered as a pointer. So we can send it to the function as we would have done with a single pointer:

```
void display(int *tableau, int size);

int main(int argc, char *argv[]){
    int tableau[4] = {10, 15, 3};
    //  displays the content of the array
    display(tableau, 4);
    return 0;
}

void display(int *tableau, int size){
    int i;
    for (i = 0 ; i < size ; i++)
        printf("%d\n", tableau[i]);
}
```

Output :

```
10
15
3
0
```

Note that there is another way to indicate that the function receives an array. Rather than indicate that the function expects a `int *tableau`, put this:

```
void display(int tableau[], int size)
```

This is exactly the same, but the presence of brackets allows the programmer to see that it is an array that the function takes, not a simple pointer. This avoids confusion. It's better to use the brackets in the functions all the time to show that the function is waiting for an array. It is not necessary to put the size of the table between brackets.

Exercises

6) Pointers – Given the test program below, write the function `cuttingMinutes` to which a number of minutes is sent, and it returns the corresponding number of hours and minutes:

- if we send 45, the function returns 0 hours and 45 minutes;
- if you send 60, the function returns 1 hour and 0 minutes;
- if you send 90, the function returns 1 hour and 30 minutes.

More precisely, `cuttingMinutes` will receive two parameters, hours and minutes, and it will change both to the corresponding number of hours and minutes of the total minutes passed by parameter.

```
void cuttingMinutes (to be completed...)
int main(int argc, char *argv[]){
    int heures = 0, minutes = 90;

    /* We have a variable minutes which is 90.
     After calling the function, I want my variable
     "hours" to be 1 and my variable "minutes" to be 30 */
    cuttingMinutes (to be completed...)
    printf("%d heures et %d minutes", heures, minutes);
    return 0;
}

void cuttingMinutes (to be completed...){
  (to be completed...)
}
```

7) Maximum Array – Create a maximumArray function that will set to 0 all cells in an array received by parameter that have a value greater than a maximum value also received by parameter.

8) Contiguous locations – Create a program that shows that all elements of an array are stored at contiguous locations.

Manipulating Files

The problem with variables is that they only exist in RAM. Once your program is stopped, all your variables are removed from the memory and you can not find their value again. Fortunately, in C you can read and write in files. These files will be written to your computer's hard drive, so they stay there, even if you stop the program or computer.

To read and write to files, we will use functions located in the library `stdio`. Here's the sequence of steps to do when you want to open a file, whether to read it or to write on it :

1. We call the "file open" function `fopen`, which returns a pointer to the file;

```
FILE* fopen(const char* fileName, const char* mode);
```

This function returns a pointer to FILE. The pointer is initialized to NULL. The **mode** says what you want to do with the file: just write to the file ("w"), just read it ("r"), or both at once ("r+"). Other modes exist.

2. We check if the opening was successful (that is to say if the file existed) by testing the value of the pointer we received. If the pointer is `null`, it means that the opening of the file did not work, in this case we can not continue (it is necessary to display an error message);
3. If the opening has worked (if the pointer is different from `null`), then we can read and write in the file through functions that we will see a little further;
4. Once you have finished working on the file, you have to close it with the function `fclose`.

There you go an example to illustrate all these steps :

```
int main(int argc, char *argv[]){
  //step 1 :
  FILE* file = NULL;
  file = fopen("test.txt", "r+");
  //step 2 :
  if ( file != NULL){
    // step 3 :
    // we can read and write in the file
    //step 4 :
    fclose(file);
  }
  else
        printf("Impossible to open the file test.txt");
```

The file "test.txt" should be in the same folder of the program. Otherwise, the complet path should be specified (the default of absolute paths is that they only work on a specific OS).

Reading a file : We'll see three functions to read files :

- **`fgetc`**: read an unsigned char cast to an int, or EOF on end of file or error. It advances the cursor of a character each time you read one.

```
int fgetc(FILE* filePointer) ;
```

- **`fgets`**: read a chain of characters. The function stops reading the line if it contains more than `nbOfCharactersToRead` characters. It reads at most one line (it stops at the first \n it meets) . It returns `NULL` if it failed to read what you requested.

```
  char* fgets(char* chain, int nbOfCharactersToRead, FILE* filePointer);
```

- **`fscanf`**: read a formated chain of characters. If you would like to retrieve some numbers in a int variable :

```
    FILE* fichier = NULL;
    int score[3] = {0};
    fichier = fopen("test.txt", "r");
    fscanf(file, "%d %d %d", &score[0], &score[1], &score[2]);
```

Exercise

9) Reading a file – Read and display on the screen the whole content of a text file.

Threads

In most operating systems, the system reserves a *process* for each application, with some exceptions. Each *process* has its memory space and a single control *thread*, the main thread. From the programming point of view, the latter executes the *main* function.
Many programs perform several activities in parallel, at least in apparent parallelism. As at the *process* level, some of these activities can freeze, and so block this blocking to a single sequential *thread* allows not to stop the entire application.
The main advantage of *threads* over *processes* is the ease and speed of their creation. All the threads of the same process share the same address space, and therefore all variables. Beyond the creation, the superposition of the execution of the activities in the same application allows an important acceleration in the execution of this one.
All thread-related functions are included in the **<pthread.h>** header file (do not forget to add `#include <pthread.h>` at the beginning of your C files).

Manipulating threads : To create a thread, you must declare a variable representing it. This will be of type `pthread_t`. Then, to create the thread, just use the function:

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
           void *(*start_routine) (void*), (void*) arg);
```

- The function returns a value of type `int`: 0 if the creation was successful or another value if not.
- The first argument is a pointer to the thread identifier (value of type `pthread_t`).
- The second argument refers to the attributes of the thread. In our example, we will usually put NULL.
- The third argument is a pointer to the function to execute in the thread. The latter must be of the form `void *function (void* arg)` and will contain the code to be executed by the thread.
- Finally, the fourth and last argument is the argument to pass to the thread.

At the end of use, you need to delete the thread. This time, it's not a puzzle function. It takes as argument the value that must be returned by the thread, and must be placed last in the function concerned :

```
void pthread_exit((void*) ret);
```

Waiting the end of a thread : As with processes, the main thread will not wait the other threads to finish before executing the rest of its own code. Therefore, we will have to explicitely ask him to do so. For that, the function `pthread_join` was designed. Joining thread is a one way for synchronization between threads. A thread can wait for another thread to terminate by executing:

```
int pthread_join(pthread_t th, void **thread_return);
```

where `th` is the thread id which we wait for terminating, and `thread_return` is a pointer to a value returned by terminated thread.

Here is an example that reuses all the notions of threads that we have seen so far :

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

void *thread_1(void *arg){
     printf("We're inside the thread.\n");
     /* Just to remove the warning */
     (void) arg;
     pthread_exit(NULL);
}

int main(void){
  int i;
  pthread_t threads[3];

  printf("Before the creation of the thread.\n");

  for(i = 0; i < 3; i++) /* create threads */
   if (pthread_create(&threads[i], NULL, thread_1, (void *)i)){
     printf("Can not create a thread\n");
     exit(1);
    }

  for (i = 0; i < 3; i++) /* wait for terminate a threads */
    pthread_join(threads[i], NULL);

  printf("After the creation of the thread.\n");
  return EXIT_SUCCESS;
}
```

Compiling a program with threads : All the built-in functions/APIs of C are contained in the pre-compiled libraries that are linked with our written code, once the code is compiled. For more specific and not frequently used functions like thread specific functions, the functions are contained in other libraries. Thread related functions are contained in **libpthread.so**. To include these libraries during linking stage, we need to explicitly tell the linker which library we want using the -l switch. To include **libpthread.so**, the switch is `-lpthread`:

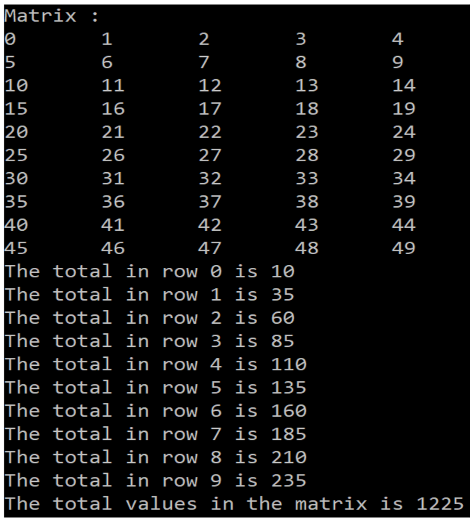
```
$gcc -o example example.c -lpthread
```

Exercise

10) Threads – Execute the code of the above example without the `pthread_join` statement. What are the behavior of the program ?

11) Sum of a matrix values – Create a program to sum up all values in a 5x10 matrix. The total of the values of the matrix is computed by N threads. Each thread computes the total value in the row and returns it (calling the `pthread_exit` function) to the main function which sums it. The main function initializes the matrix with ordered values from 0 to 49 and creates N threads. Threads share access to the matrix which is declared as a global variable. The main function waits for all threads to terminate using the `pthread_join` function and computes the total of the values of the matrix. It sums values returned by all threads. When the total is computed it prints it to standard output using the `printf` function.

Expected output :



```
Matrix :
0       1       2       3       4
5       6       7       8       9
10      11      12      13      14
15      16      17      18      19
20      21      22      23      24
25      26      27      28      29
30      31      32      33      34
35      36      37      38      39
40      41      42      43      44
45      46      47      48      49
The total in row 0 is 10
The total in row 1 is 35
The total in row 2 is 60
The total in row 3 is 85
The total in row 4 is 110
The total in row 5 is 135
The total in row 6 is 160
The total in row 7 is 185
The total in row 8 is 210
The total in row 9 is 235
The total values in the matrix is 1225
```

Extra exercise

12) Hangman – Randomly select a word from a file, have the user guess characters in the word. For each character they guess that is not in the word, have it draw another part of a man hanging in a noose. If the picture is completed before they guess all the characters, they lose.

You may use files