C Programming

By: Mohamed Aziz Tousli

About C

- C is a high level language
- Code source = a written program
- IDE = Integrated Development Environment = Compiler + Text editor + Debugger
- Window program (paint) vs Console program (cmd)
- Every code source in C must end with an empty line
- We can’t print accents in C + Windows

Basics

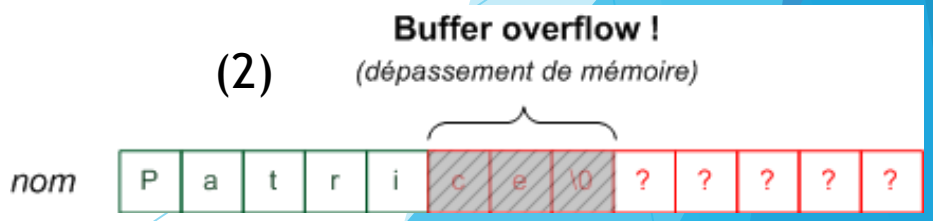
- //This is a short comment
- /* This is a long comment */
- Common libraries: stdio.h, stdlib.h
- Main function: obligatory cause the program starts with it

```
int main(){
Instructions ;
return 0} //return 0 for orgazional purposes
```

- Operations: + - * / %
- Quotient = 5/2 = 2; Rest = 5%2 = 1; Division = 5.0/2.0 = 2.5
- x=x+1 ⇔ x++ ⇔ x+=1 //Incrementation
- x=x-1⇔ x-- ⇔ x-=1 //Decrementation

Variables

- const constType CONST_NAME=Value; //Can't be changed
- variableType variableName1, variableName2=Value; //Ab_1✓; 1✗; é✗
- Initially, a variable takes the value of the content of the address: random
- ➔ We have to do initialization for variables to avoid randomness problems
- variableType = signed char, int, long; float, double //float x=3.5
- variableType = unsigned char, unsigned int, unsigned long
- printf("The variable %d is not %f",variable1,variable2); //To write
- "\n" "\t" special characters for formatting the message
- scanf("%d %f",&variable1,&variable2); //To read
- Print float/long float with "%f", scan float with "%f" and long float with "%lf"
- "%p" for hexadecimal; "%c" for character; "%s" for string
- scanf problems: (1) it stops at a space or a special character
- c=getchar(); ⇔ scanf("%c", &c); //Read first character
- 'C'=toupper('c'); //Upcase a character, we need the library ctype.h



Math library & Random library

- #include<math.h>
- fabs(-5.0)=5.0 //|x|
- abs(x) exists in stdio.h for integers only
- ceil(25.5)=26.0 //Ceiling
- floot(25.5)=25.0 //Floor
- pow(5,2)=5^2=25 //Power
- sqrt(25)=5 //Square root
- sin(x), x is in radian
- #include<time.h>
- srand(time(NULL)); //Must be called only one time
- x=rand() % (MAX-MIN+1) + MIN; //Generate a random number in [MIN,MAX]

Conditioning

```
if (/* condition1 */)
{ /* code */ }
else if (/* condition2 */)
{ /* code */ }
else
{ /* code */ }
```

```
switch(x)
{
    case 1:
        /* code */
        break;
    default:
        /* code */
}
```

- Comparative forms: ==, <, <=, >, >=, != ; && **AND**, || **OR**, ! **NOT**
- If code has 1 instruction, {} can be removed
- if (x) <=> if(x==1) : Boolean variables - 0=False, !0=True
- Boolean variables do not exist in C, so we use int instead
- x = (/* condition */) ? ifChoice : elseChoise; //Conditional teranary

Loops

```
while(/* condition */)
{
	/* code */
}
```

```
for (int i = 0; i < count;
i++)
{
	/* code */
}
```

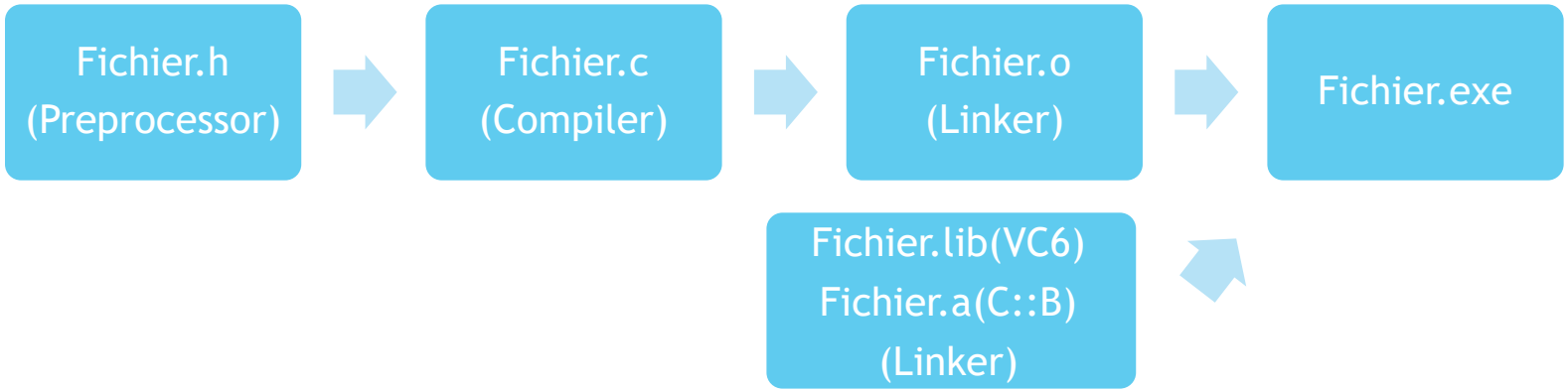
```
do
{
	/* code */
} while(/* condition */);
```

Functions

- typeResult FunctionName(typeA A, typeB B) //parameters or arguments
- {
- /* code */
- return x;
- }
- Type is void if function has no return
- x=FunctionName(A,B) //Call a function
- Modular programming: Program using librairies .h
- type FunctionName(typeA, typeB); //Prototype
- If prototype before main, function can be anywhere; else function before main
- Project = a set of .c & .h files ; .c contains functions & .h contains prototypes

Compilation & Global/Local variables

- Seperate compilation:



- Created .h call .c & standard .h call .a/.lib
- Global variable: x after #include → Visible for all the files in the project
- Local variable: x in a function → Visible for the function only
- Global variable: static x after #include → Visible for the file only
- Local variable: static x in a function → x keeps its value when the function exits
- Global function: by default → Visible in all the files in the project
- Local function: static function → Visible in the file only

Pointers

- For variables:
  - x //Shows its value
  - &x //Shows its address
- For pointers:
  - x //Shows its value
  - *x //Shows the value of the variable that x points on
- T* pointer=&x; T *pointer1=NULL,*pointer2; //Create a pointer on type T
- void FunctionName(Type *Pointer) {Everything *Pointer} //In prototype
- FunctionName(&x); //In main {1}
- Type *Pointer=&x;FunctionName(Pointer);Use(*Pointer or x); //In main {2}

Arrays

- Array = Sequence of variables of the same type, located in contiguous space in memory
- int array[N]; //Create an array of size N; array itself is a pointer on array[0]
- N must not be a variable/constant, N must be a number
- array[i]; //ith+1 value of array because arrays start with index 0
- int Array[N]={Value1, Value2}; //array=[Value1, Value2, 0, .., 0] //It completes with 0 by default
- void FunctionName(Type *Array, int ArraySize) //Use functions to call arrays {1}
- void FunctionName(Type Array[], int ArraySize) //Use functions to call arrays {2}

Strings (1)

- char c='A'; //Create ONE character
- 'A'=65 - ASCII Table
- char str[N]; //Create a string of size N, we can str[i]='X'
- char* str; //Create a string, we can't str[i]='X'
- " " for strings; ' ' for characters
- **A string ends with the character '\0'**
- For arrays of type char (i.e. string), we don't have to pass ArraySize as an argument
- char str[]="StringName"; //string={'S','t', .., '\0'}; Size is automatically calculated
- Last line can't be done in code. It is done only in initialization

Strings (2)

- #include<string.h>
- sizeStr = strlen(str); //Size of a string
- strcopy(str2,str1); //str2=str1
- strcat(str2,str1); //str2=str2+str1: Concatenation
- We have to put N too big to assure that it has no limits problem
- strcmp(str2,str1); //Compare str2 to str1; 0 if equal, !0 if not
- strRest = strchr(str, character); //Look for ch in str, return the rest of str after ch; NULL if not
- strRest = strpbrk(str, characters); //Look for one of the chs and return the rest of str after it
- strRest = strstr(str1, str2); //Look for str2 in str1 and return the rest of str1 after str
- sprintf(str,"message"); //str="message"
- A string must be initialized with “” to avoid memory problems

Preprocessor directives (1)

- Preprocessor: Replace #'s with other values before compiling
- #include<library.h> //Standard library (Replace contents of .h in .c file)
- #include”library.h” //Created library
- #define N 5 //Replace N with 5 in the whole file
- const takes place in memory, define doesn't (because it is done in the preprocessing)
- Predefined constants by the preprocessor:
  - __LINE__ //number of current line
  - __FILE__ //name of current file
  - __DATE__ //date of compilation
  - __TIME__ //hour of compilation
- Macro without parameters:
  - #define HELLO() printf("Message1"); \
  - printf("Message2"); //Replace HELLO() with printf("Message")
- Macro with parameters:
  - #define Function(x,y) if(x||y) {};
  - Function(5,6) //Replace Function with if and (x,y) with (5,6)

Preprocessor directives (2)

- Conditional compilations:
- #if condition1
- /* code source to compile if condition1 is true */
- #elif condition2
- /* code source to compile if condition2 is true */
- #endif
- Utility of #define constant without value:
- #define WINDOWS
- #ifdef WINDOWS
- /* code source for WINDOWS */
- #endif
- To avoid infinite inclusions:
- #ifndef DEF_hFILE //If DEF_hFILE was not defined, i.e., DEF_hFILE was never included
- #define DEF_hFILE //We define DEF_hFILE so that it won't be called next time
- /* contents of DEF_FILE.h */
- #endif

Types

- typedef struct structureName structureName;
- struct structureName {
- Type1 variable1;
- Type2 variable2;
- }; //Do not forget about ';'
- → typedef create an equivalent of the structure. 'struct structName' is the name of the structure that we want to copy. 'structureName" is the name of the equivalent ➔ Writing 'structureName' is the same as 'struct structureName' since it is annoying to create a structure without a typedef
- structureName variableName={,}; //Create a variable with type structureName
- variableName.variable1 //Value1 of VariableName
- (*variableName).Variable1 ⇔ variableName->Variable1 //Pointers on structures
- '.' is for variables and '->' is for pointers
- We put new types usually in .h files
- typedef enum typeName typeName;
- enum typeName {VALUE1=x, VALUE2=y, VALUE3=z};
- If no (x,y,z), compiler does an automatic association to values 0, 1 and 2

Files

- FILE* f=fopen("C:\\FileDirectory.type",OpeningMode); //Open a file; "r,w,a,r+,w+,a+" (read, write, append)
- if f=NULL, it is impossible to open the file
- "w+" is dangerous because it deletes the content of the file
- "w" if the file exits, it is replaced, else it is created
- a=fclose(f); //Close the file, a=0 if done right, else a='EOF' (End Of File)
- fputc(character,f); //Write a character in the file
- fputs(string,f); //Write a string in the file
- fprintf(f,string); //Write a string in the file (It can have variables within it, like printf)
- → These functions return the integer 'EOF' in case of error
- fgetc(f); //Read a character, return 'EOF' if it doesn't exist
- fgets(str,NumberOfChars,f); //Read a line, return 'NULL' if it doesn't exist
- There is a 'cursor' that goes through the character/lines every time the function is called
- fscanf(f,"%d",&Variable); //Read like scanf()
- position=ftell(f); //Give the position of the 'cursor'
- fseek(f,displacement,origin); //displacement can be positive or negative, origin=SEEK_SET,SEEK_CUR,SEEK_END
- rewind(f); <=> fseek(f,0,SEEK_SET);
- rename(oldName,newName); //Rename a file, return 0 if done right
- remove(f); //Delete a file from the hard disk!

Dynamic allocation

- sizeof(Type) //Functionality in C to know the size of a type (not a function)
- → char occupies 1 byte, int occupies 4 bytes..
- So far, the program has been demanding the OS to free space for the variables: Automatic allocation
- But now, we are going to demand the OS manually to free space for the variables: Dynamic allocation
- z=malloc(numberOfNecessaryBytes); //Memory Allocation, returns a pointer on void (on any type) i.e. @
- z=malloc(sizeof(Type)); //'Indicate' for the pointer the type of its variable
- if z==NULL, exit(0); else do the work
- Be careful: In dynamic allocation, we use pointers instead of standard variables
- free(z); //Free the memory allocation
- Dynamic allocation is good for arrays, since we don't know its size and we can't put a variable for it
- arrayType* array=(arrayType*)malloc(arraySize*sizeof(arrayType)); //Create a table dynamically

Secured read

- str=gets(); //Like fgets but doesn't avoid buffer overflow
- str=fgets(str,size(-'\0'),stdin); //Avoid buffer overflow compared to scanf, stdin else FILE*f
- stdin: what is written by the keyboard, pointer on buffer
- Buffer = Memory zone that receives the stdin
- fgets stop when it finds '\n' (and keep it) or when it reaches the size limit, the rest will stay in the buffer and will be extracted from another fgets
- getchar is equivalent to fgets but for characters
- PS: 2 '\0's count as 1, since the program stop at the first one
- long=strtol(str,NULL,base=10); //Convert str to long, 0 if wrong (" 43.5abc" → 43)
- Double=strtod(str,NULL); //Convert str to double, 0 if wrong (" 43.5abc" → 43.5)

SDL library (1)

- → Create 2D games
- Standard library (by default) vs Third party library (has to be installed)
- GPL License (General Public License) vs LGPL License (Lesser GPL); both are open source
- #include<SDL/SDL.h>
- x=SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_CDROM | SDL_INIT_JOYSTICK | SDL_INIT_EVERYTHING); ⇔ Malloc //Load SDL; x=0 if good, -1 if error; SDL_INIT_X: flag
- SDL_Quit(); ⇔ Free //Stop SDL, (and free screen from memory)
- fprintf(stderr,"%s",SDL_GetError()); //To write the error in stderr.txt, and get the latest SDL error
- exit(EXIT_FAILURE); /*main*/ return EXIT_SUCCESS; //Variables of exit that go with any OS
- SDL_Surface* surface = NULL; //Create a pointer on screen (basically, a surface)
- SDL_Surface* screen=SDL_SetVideoMode(width,height,number of colors{32 bits/px},SDL_HWSURFACE{video memory, faster & less space} | SDL_SWSURFACE{RAM, slow & more space} | SDL_NOFRAME | SDL_FULLSCREEN | SDL_RESIZABLE | SDL_DOUBLEBUF{for fluid motion}); //Open a window, NULL if error
- → Without a pause function, it closes automatically
- SDL_WM_SetCaption(newWindowName,NULL); //Change the name of the window

SDL library (2)

- SDL_FillRect(screen,NULL,color); //Color the screen; Type of 'color' is 'Uint32'=int in SDL
- color=SDL_MapRGB(screen->format, Red, Green, Blue); //Return a color
- SDL_Flip(screen); //Update the screen (after modifications)
- SDL_Surface* surface=SDL_CreateRGBSurface(SDL_HWSURFACE | SDL_SWSURFACE,W,H,32b/p,0,0,0,0); //Create a new surface inside the main surface, i.e., the screen
- SDL_FreeSurface(surface); //Free surface from memory
- SDL_Rect position; position.x = 0; position.y = 0; //Create a position
- SDL_BlitSurface(surface, NULL, screen, &position); //Blitter surface on screen