Course: C language

Ali ZAINOUL

for Keyce Academy
March 19, 2023

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Introduction

Language Overview

- C programming language is a general purpose, high-level language that was developed by **Dennis M. RITCHIE** in order to develop the UNIX operating system.

- UNIX OS, C compiler (gcc for instance) and all UNIX apps are written in C. The C language is widely used in industry for the reasons below:
  - C is easy to learn
  - C is a structured language
  - C is efficient
  - C is compiled in many Operating Systems and many computer platforms.
- **A C program or C source code is written into one or more text files with the extension ".c" for source files and ".h" for library files.**

Language Overview

- C was invented to write the operating system **UNIX**
- C is a successor of **B language**, which was invented in 1970s.
- The language was formalized in about the year 1988 by the **American National Standard Institute** (ANSI)
- The **UNIX OS is totally written in C in 1973**
- C is most widely used and **popular** system programming language
- Most Linux Operating Systems, and many industrial applications are written in C.

Language Overview

C was initially used for system development and in order to write Operating Systems as stated above. C is as fast as the **assembly language**. C is used for:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Networks
- Databases
- Modern Programs
- Utilities

Environment setup

In order to program in C, you need the following:

- Either: (1): **a Text Editor** and (2): a **C compiler** using command line terminal
- Either: an **Integrated Development Environment** (IDE).

Environment setup

If you choose the first option: text editor plus a C compiler, you can choose from the following:

- Text editor: one can choose either Atom (consuming a lot of memory), Sublime Text, Visual Studio, gEdit, VIM, Notepad++.
- for C compiler: It all depends on your Operating System (OS)
  - **Linux / Unix based systems** (most recommanded): we recommand installing the most stable version of Linux called Ubuntu. Once it is done, one may install **GNU Compiler Collection** (**GCC**) by following these command lines:

```
sudo apt update
sudo apt install build-essential
sudo apt-get install manpages-dev
gcc –version
```

Environment setup

- for C compiler: It all depends on your Operating System (OS)
  - **Linux / Unix based systems**
    - The first command updates the system.
    - The second one installs a bunch of new packages including **gcc** (for C), **g++** (for C++) and make (for both languages).
    - The third command installs the manual pages about using GNU/Linux for development.
    - The last one is used to check your version of GCC.
  - **Macos based systems**

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install gcc
gcc –version
```

    - The first command installs Homebrew (The Missing Package Manager for macOS)
    - The second one tells Homebrew to install GCC
    - The last one checks which version of gcc is installed.

Environment setup

- for C compiler: It all depends on your Operating System (OS)
  - **Windows based systems** One could follow the steps in this website: Installing GCC using command line on Windows

Environment setup

If one has chosen to use the second option (i.e. installing an Integrated Development Environment), one may install one of the following:

- Code::Block
- Codelite
- NetBeans

Program structure

A C program consists of the following parts:

- Preprocessor commands
- Functions
- Variables
- Statements
- Expressions
- Comments

Program structure

Open a text editor or your favorite IDE, create a new file, and name it "**helloCWorld.c**" and copy paste the code below:

```
// This is a comment
/*
This is a comment on many
lines.
*/
#include <stdio.h>
// This library is for using the function below printf
// std io stands for standard input/output stream
int main()
{
// This is my first C program
printf(" Hello, C World! \n ");
return 0;
}
```

Program structure

Let's analyze the helloCWorld.c code above:

- The first line is a comment, a comment is not considered as code, the compiler will ignore it.
- The second bloc represents a comment in many lines as stated.
- The **#include <stdio.h>** is for including the library **stdio.h**, which will help us call the function **printf** as stated.
- The function **printf**("my_text") will simply output the text "my_text", however the termination **"\n"** is for returning to the next line, we will see more in details all the special characters and their usage in C.

Compiling and executing the program

Let's look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

- **Open a text editor** or your **favorite IDE** and **copy paste the HelloCWorld code** above then save it in a directory as **HelloCWorld.c**
- **Open a command prompt** and **go to the directory where your file is saved**.
- Type " **gcc -o myHelloProgram helloCWorld.c** " 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 the " myHelloProgram " executable**.
- **Tap " ./myHelloProgram " in order to execute it**.
- You'll be able to see **"Hello, C World!" printed on the screen.**

Compiling and executing the program

```
$ cd my_directory_where_is_my_file
$ gcc -o myHelloProgram helloCWorld.c
$ ./myHelloProgram
Hello, C World!
%
```

Basic Syntax

- This section gives details about the basic syntax about C language including tokens, keywords, identifiers, etc.
- We saw a basic structure of C program (helloCWorld.c), so it'll be easy to understand the basic building blocks of the C language.

Tokens in C

- A C program consists of various **tokens**.
- A token is either a **keyword**, an **identifier**, a **constant**, a **string literal**, or a **symbol**. For example, the following C statement consists of five tokens:

```
printf (”Hello, C World! \n”);
```

Tokens in C

- The tokens above are:

```
printf
(
"Hello, C World! \n"
)
;
```

Semicolons ; in C

- A semicolon in a C program is a statement terminator. Each individual line must be ended with a semicolon. It indicates the end of one logical entity.

```
printf("Hello, C World! \n");
return 0;
```

- We know that there is exactly two statements in the code above, the **printf("my_text")** directive and the **return 0** directive.

Comments in C

- Comments are statements in your C program that are **ignored by the compiler**. They start either with:
  - // for a comment in one line, or:
  - With /* and terminates with the characters */ for comments in many lines, as shown below:

```
// This is a comment on one line
```

```
/* This is a comment on many lines
This is the same comment until we reach the "*/" statement that
terminates the comment */
```

- **Remark:** You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers in C

- A C identifier is a name used in order to identify:
  - a variable, e.g.: double my_double
  - a function, e.g.: void my_print(string my_string)
  - any user-defined item.
- An identifier starts with a letter (lowercase and/or uppercase) or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).
- C does not allow special characters such as: , $, and % within identifiers.
- C is case sensitive. Thus, "m**yvariable**" and "M**yvariable**" are two different identifiers in C.

Keywords in C

- There is **reserved words** that must not be used as constant or variable or as any other identifier name. The **reserved words** are reserved to the language itself. The non-exhaustive list is below:

auto	else	Long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_packed
double			

Figure: Reserved keywords

Whitespaces in C

- A line containing only whitespace.s, possibly within (or not) a comment, is known as a **blank line**, and a C compiler totally **ignores it**.
- **Whitespace** is the term used in C to describe **blanks**, **tabs**, **newline characters** and **comments**.
- Whitespace separates one part of a statement from another one.

Whitespaces in C

- Whitespace enables the compiler to identify where one element in a statement, such as **int**, ends and the next element begins. Therefore, in the following statement: **int mark;**
- There must be at least one whitespace character (usually a space) between **int** and the identifier **mark** in order for the compiler to be able to distinguish them. Otherwise, the compiler will identify **intmark** which may be an identifier on its own...

Data types

In C, data types refers to the fact that every variable or function has it's own type, either basic or generic (user-defined) and/ or derived types. The type of a variable will determine how much space it must occupy in order to store it into memory, and how the bit pattern will arise. So as stated, there is four types of variables in C, classified as follow:

- **Basic types:** arithmetic types; either **integer** ones (natural numbers and integers in mathematics), or **floating-point** ones (real numbers).
- **Enumerated types:** arithmetic types too, used to define variables only assignable by certain **discrete integer values**.

Data types

- **Void types:** the **void** type indicates that no value is assigned. (think of void in real world)
- **Derived types: derived** types may be pointers, arrays, structures, classes (in any Oriented Object Programming languages), union types and functions. (the list is not exhaustive).

  Remark: In C, they are two aggregate types: arrays and structures. Aggregate types are collections of scalar values.

Data types

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	byte	8	-128	127	From +127 to -128	`byte b = 65;`
	char	16	0	2^{16}-1	All Unicode characters[3]	`char c = 'A'; char c = 65;`
	short	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	`short s = 65;`
	int	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	`int i = 65;`
	long	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	`long l = 65L;`
Floating-point	float	32	2^{-149}	$(2-2^{-23})\cdot2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	double	64	2^{-1074}	$(2-2^{-52})\cdot2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	`double d = 65.55;`
Other	boolean	--	--	--	false, true	`boolean b = true;`
	void	--	--	--	--	--

Figure: Primitive Data Types

- Primitive types are the most basic data types that can be found in programming languages such as C, C++ or Python. We distinguish 8: **boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**. A combination of char**s** results in a **string**.

Data types

- In order to get the exact size of a type or a variable on a particular operating system with a particular compiler, you can use the sizeof operator. The expressions sizeof(type) yields the storage size of the object or type in **bytes**. **This link sizes.c** gives a complete example of the size of basic data types on a **64-bit architecture system**.

Remark: be aware of the fact that the sizeof operator will return slightly different values depending on the architecture system, OS, compiler etc.

Data types: Format Specifiers

Format specifiers are used during Input and Output processings. They are used in order to tell the compiler what type of data is in a variable while scanning a variable using scanf() or printing a variable using printf().

This article explicits the **format specifiers** used in C.

Data types: the void type

The **void type** is an important concept built-in type and specifies that no value is available. It is used in three particular situations:

- **The return_type of a function is of type void.** Function returns as void. There are various functions in C which do not return value hence they are returning **void type**. Example:

```
void draw_line () {
printf( " - – — - – — — – - — – - \n ");
}
```

Data types: the void type

- **Function arguments as void**. Functions in C taking arguments list as void are simply functions with unknown parameters, whereas in C++, f(void) is exactly the same as f().

- **Pointers to void**. A pointer of type **void *** (the star * is for declaring a pointer!) represents the **address of an object**, but ***NOT*** its type. For example, a memory allocation function void *malloc(size_t size); returns a pointer to void which can be **casted** (will return to that later) to any data type.

Remark: the void type is an important concept in C language, it is useful for handling and manipulating memory areas, pointers and casting as said. The only object that can be declared with the type specifier void is a pointer.

Variables

Definition: A **variable** is a name given to a storage area that a program may manipulate.

- Each variable in C has:
  - **a specific type**, which determines the size and layout of the variable's memory;
  - **the range of values** that can be stored within that memory;
  - and **the set of operations** that can be performed to the variable.

Variables

- The name of a variable have to follow these two rules:
  - ***it must begin*** with either a letter, or an underscore.
  - the variable has to be composed of alphanumerical values and/or the underscore character. E.g: _1, _myvar, a_var, var, and var123_L are all valid examples.
  - C programming language also allows to define various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc. Here we will only study basic variables.

Remark: We recall that C is case-sensitive, so Upper and lowercase letters are distinct, e.g.: myvar is ***NOT*** myVar.

Variables

- In the following frames, we will highlight the difference between the **declaration**, the **definition** and the **initialization** of a variable, ***AN IMPORTANT CONCEPT*** in programming languages, and especially in C/C++ ones.

Variables
Variable **Declaration** in C

Definition: Declaration of a variable is generally an introduction to a new memory allocated and highlighted by an identifier with a given type (built-in, basic or user-defined).

- The three properties of declaration are:
  - Memory space creation occurs at the same time of declaration itself.
  - Variables declared ***but not defined or initialized**** may have garbage values (random value of the given type).
  - Variables ***CANNOT**** be used before declaration.

Variables
Variable **Declaration** in C

- It is done by the following:

```
type variable_list;
```

- The **type *must*** be a **valid C data type** including: char, int, float, double, bool or any user-defined object, etc.

- **variable_list** may consist of **one** or **more** identifier names separated by commas. Some valid declarations are shown below:

Variables
Variable **Declaration** in C

```
// This is a simple comment and will be ignored by the compiler
int a; // Declaration of variable a with int type
char b, c; // Declaration of distinct variables b and c with char type
float d, e, f; // Declaration of distinct variables d, e and f with float type
double _d; // Declaration of the variable _d with double type.
```

Remark: the same is applied to functions, one may **declare** a function and **define** it later. E.g: void foo(); is a declaration of the function foo. And without giving it any definition, the code can compile provided that the function isn't used elsewhere in your program. (which is quite useless...).

Variables
Variable **Definition** in C

Definition: Definition of a variable is assigning a value to a variable, typically with the **assignment operator =**. It is the fact of assigning a ***valid*** value to the previously declared variable.

```
int a, b; // Declaration of distinct variables a and b of type int
a = 1; // Definition of variable a
b = a + 1; // Definition of variable b as the value of variable a + 1 (=2 here)
```

Variables
Variable **Initialization** in C

Definition: Initialization of a variable is simply the fact of assigning (defining) the value at the time of declaration.

For example:

```
float f = 0.3;
```

is declaring a variable with identifier f, of type float and defining / assigning to it the value 0.3.

Function Declaration and Definition

A **Function Declaration** as seen above has the signature:

```
return_type name_of_my_function(args if any);
```

Whereas a **Function Definition** is given by:

```
return_type name_of_my_function(args if any) {
Stuff_that_my_function_will_do;
}
```

Function Declaration and Definition

A complete example is as follows :

```
// Function Declaration
int my_function();
int main()
/*
Function call, the Definition may comes:
- just after the Declaration in the same source program
- be part of another source program that must be compiled before we use the function.
*/
// Function Call
my_function();

// Function Definition
int func() return (-123) ;
// The function simply returns the integer value -123, the parenthesis are here for the sake of exactness.
```

Notion de recursivity: recursive functions

Definitions:

- An algorithm is called **recursive** if it is defined by it-self.
- In the same manner, a function is called **recursive** if it references itself.

Lvalues and Rvalues in C

There are two kinds of expressions in C:

- **lvalue**: An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue**: An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Constants and literals

Definition: constants refers to fixed values that a program may not alter during its execution. These fixed values are called literals.

- Constants can be of any of the basic data types:
  - an **integer** constant;
  - a **floating** constant;
  - a **character** constant;
  - a **string** literal;
  - an **enumeration** constant.

Number systems

We distinguish four common number systems used in everyday communication protocol. We detail them as following:

- **Binary system**, working on base **2**;
- **Octal system**, working on base **8**;
- **Decimal system**, working on base **10**;
- **Hexadecimal system**, working on base **16**.

Number systems

Here is a picture illustrating the equivalence between the systems:

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Figure: Number systems

Constants and literals
Integer Literals

An integer literal may be a *decimal*, *octal*, or *hexadecimal* constant. A prefix specifies the base or radix:

- **0x** or **0X** for **hexadecimal** (base **16**);
- **0** for **octal** (base **8**);
- **nothing** for **decimal** (base **10**).

An example of integer literals:

Constants and literals
Float Literals

A floating-point literal is composed of:

- an integer part;
- a decimal point;
- a fractional part;
- an exponent part.

Floating point literals may be represented either in **decimal** form or **exponential** form.

Constants and literals
Float Literals

- **Decimal form**: must include the decimal point, the exponent, or both;
- **Exponential form**: one must include the integer part, the fractional part, or both. While the signed exponent is introduced by lowercase or uppercase **e** (**e** or **E**).

Some examples from IBM website:

floating-point constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Figure: float literals

Constants and literals
Character constants

Definition: Character literals are enclosed in single quotes and can be stored in a simple variable of char type.

- a character literal can be a plain character (e.g. char a = 'a' ;) ;
- an escape sequence '\t' (we will come to this later) ;
- an universal character (we will come to this later).

Constants and literals

Character constants

Certain characters in C, when preceded by a backslash, have a special meaning. A non-exhaustive list is below:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Figure: Escape Sequences

Constants and literals
String literals

Definition: String literals or constants are enclosed in double quotes "".
A string is simply an array of character literals: plain characters, escape sequences, and/or universal characters.

Constants and literals
String literals

- We may want to split a long string into multiple ones using literals by separating them using whitespaces. One example may be the following:

```
”Hello world!”
”Hello \
world!”
”Hello ” ”w” ”orld!”
```

Constants and literals
Defining Constants

There are two ways in order to define constants in C and C++ :

- with the **#define** preprocessor directive;
- with the **const** keyword. Here are the two ways:

  - #define my_Identifier my_Value (e.g. #define PI 3.14)
  - const type my_Identifier = my_Value; (e.g. const float PI = 3.14...)

Remark: no semi-colon needed for preprocessor constants. **Note:** it is recommanded / a good practice to define constants in CAPITAL letters.

Storage classes

Definition: In C programming language, a **Storage Class** defines the scope (visibility) and lifetime of variables and functions in a C program. These specifiers (a.k.a storage classes) precede the type that they modify.

We distinguish four (4) storage classes: **auto**, **extern**, **static** and **register**. We specify the storage class for a variable as follows:

```
storageClass variableDataType variableName;
```

Operators

Definition: In mathematics and computer programming, an **operator** is a symbol or character that tells the compiler to perform specific mathematical or logical processes.

Every programming language has its own built-on operators, however the operators presented here are common to many programming languages. C provides these built-in operators: **Arithmetic** operators, **Relational** operators, **Logical** operators, **Logical** operators, **Bitwise** operators, **Assignment** operators and **Miscellaneous** operators (Misc).

Operators
Truth Table

An important concept about manipulating operators is the truth table. Assume that we have two propositions (P) and (Q). Then:

P	Q	P AND Q	P OR Q	P XOR Q	NOT P	NOT Q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Operators
Arithmetic Operators

The following table explicits the C built-in Arithmetic Operators.
This example shows these arithmetic operators in a C program.

Operator	Meaning
+	Adds the Right Hand Side (RHS) operand value to the Left Hand Side (LHS) operand value (a+b)
−	Subtracts the RHS operand value from the LHS operand value (a-b)
*	Multiplies the LHS and the RHS operand values together (a*b)
/	Divides the LHS operand value by the RHS operand value (a/b)
%	Modulus returns the remainder of an euclidian division of the LHS by the RHS (a%b)
++	Increments an integer value by one (++a **preincrément**) or (a++ **postincrément**)
−−	Decrements an integer value by one (–a **predécrement**) or (a– **postdecrément**)

Operators
Relational Operators

The following table explicits the C built-in relational operators.
This example shows these Relational Operators in a C program.

Operator	Meaning
==	Evaluated to true only if the two operands are equal
!=	Evaluated to true only if the two operands aren't equal
>	Evaluated to true only if LHS (Left Hand Side) operand value is strictly bigger than the Right Hand Side (RHS)
<	Evaluated to true only if Right Hand Side (RHS) operand value is strictly bigger than the Left Hand Side (LHS)
>=	Evaluated to true only if LHS operand value is strictly bigger OR equal than the RHS
<=	Evaluated to true only if RHS operand value is strictly bigger OR equal than the LHS

Operators
Logical Operators

The following table explicits the C built-in logical operators. **This example** shows these Logical Operators in a C program.

Operator	Meaning
&&	Logical AND operator. The condition is true only if the two operands aren't zeroes.
||	Logical OR operator. The condition is true only if one of the operands isn't zero.
!	Logical NOT operator. It reverses the logic state. If condition is true, then logical NOT operator will make it false, and vice-versa.

Operators
Bitwise Operators

The following table explicits the C built-in Bitwise operators.
This example shows these Bitwise Operators in a C program.

Operator	Meaning
&	Bitwise binary AND Operator, evaluates to TRUE only if the two bits are the TRUE.
|	Bitwise binary OR Operator, evaluates to TRUE only if one of the bits is TRUE.
^	Bitwise binary XOR operator. evaluates to TRUE only if a bit is TRUE in one operand but not both.
~	Bitwise unary NOT operator (or Complement Operator). Complement Operator simply flips the bits (same logic as NOT).
«	Bitwise binary Left Shift Operator. The LHS operands value is moved left by the number of bits specified by the RHS operand.
»	Bitwise binary Right Shift Operator. The LHS operands value is moved right by the number of bits specified by the RHS operand.

Operators
Assignment Operators

The following table explicits the C built-in Assignment operators. **This example** shows these Assignement Operators in a C program.

Operator	Meaning
=	Assignment Operator. It assigns values from RHS operands to LHS operand.
+=	Add AND Assignment Operator. It adds RHS operand to the LHS operand and assign the result to the LHS operand.
-=	Substract AND Assignment Operator. It substracts RHS operand to the LHS operand and assign the result to the LHS operand.
*=	Multiply AND Assignment Operator. It multiply RHS operand to the LHS operand and assign the result to the LHS operand.
/=	Divide AND assignment Operator. It divides LHS operand with the RHS operand and assign the result to LHS operand
%=	Modulus AND Assignment Operator. Assigns remainder of the Euclidian Division of LHS operand by the RHS operand to LHS operand.
«=	Left Shift AND Assignment Operator.
»=	Right Shift AND Assignment Operator.
&=	Bitwise AND Assignment Operator.
^=	Bitwise Exclusive OR, AND Assignment Operator.
|=	Bitwise Inclusive OR, AND Assignment Operator.

Operators
Miscellaneous Operators

The following table explicits the C built-in Miscellaneous Operators.
This example shows these Miscellaneous operators in a C program.

Operator	Meaning
sizeof()	It returns simply the size of a variable in bytes.
&	It returns the address of a variable.
*	Pointer operator.
? :	Ternary Operator. Conditional Expression.

Operators
Precedence and **priority** in C

Operator precedence decides the grouping of terms in an expression. It will affects how an expression is evaluated. Certain operators have higher precedence than others.

- **The rule is**: within an expression, higher precedence operators will be evaluated first.
- In the table below, operators with the highest precedence appear at the top, whereas those with the lowest precedence appear at the bottom.

Operators
Precedence and **priority** in C

Operator	Classification	Associativity
() [] -> . ++ –	Postfix	Left to right
+ - ! ~ ++ – (type)* & sizeof	Unary	Right to left
*/ %	Multiplicative	Left to right
+-	Additive	Left to right
« »	Shift	Left to right
< <= > >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional	Right to left
= += -= *= /= %= »= «= &= ^= |=	Assignement	Right to left
,	Comma	Left to right

Alternate and Repeating Conditional Structures & Loops

- In algorithms and programming, we call a **conditional structure** the set of instructions that test whether a condition is true or not. There are three of them:
  - The conditional structure **if**:

```
if (condition) {
my_instructions;
}
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation:**
  - The Conditional Structure **if ...else** :

```
if (condition) {
my_instructions;
}
else {
my_other_instructions
}
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation:**
  - The conditional structure **if ...elif ...else** :

```
if (condition_1) {
my_instructions_1;
}
elif (condition_2) {
my_instructions_2
}
else {
my_other_instructions
}
```

Alternate and Repeating Conditional Structures & Loops

- Furthermore, an **iterative conditional structure** allows the same series of instructions to be executed several times (iterations). The **while** (while) instruction executes the instruction blocks as long as the while condition is true. The same principle for the **for** loop (for), the two structures are detailed below:
  - The iterative Conditional Structure **while**:

```
while (condition)
my_instructions;
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation**:
  - The iterative conditional structure **for**:

```
for (my_initialization; my_condition; my_increment){
my_instructions;
}
```

- **Remark**: The major difference between for loop and the while loop is that for loop is used when the number of iterations is known, whereas execution is done in the while loop until the statement in the program is proved wrong.

Alternate and Repeating Conditional Structures & Loops

- **Continuation**:
  - The iterative conditional structure **do ...while**:

```
do {
my_instructions;
while(my_condition);
}
```

- **Note**: a difference lies between the **while** loop and the **do while** loop: in the **while** loop the check condition executes before the first iteration of the loop, while **do while**, checks the condition **after** the execution of the instructions inside the loop.

Scopes

Definition: A **scope** in any programming language is the area where a function or variable is visible and accessible. Whereas outside the same scope the function or variable is not accessible.

There are three regions where variables may be declared in C:

- Inside a function or a block which is called **local variables**;
- Outside of all functions which is called **global variables**;
- In the definition of function parameters which is called **formal parameters**.

Scopes

Here are three code source examples of local, global variables and formal parameters:

- **Local Variables example;**
- **Global Variables** example;
- **Formal Parameters example.**

Scopes: behaviour of initializing local and global variables

When a **Local Variable** is **declared**, it is ***not initialized*** by the compiler. Whereas **Global Variables** are **initialized automatically** when declared:

Data Type	Default Value when declared
int	0
char	'\0'
float	0.0 or simply 0
double	0.0 or simply 0
Pointer	NULL

Important note: It is highly recommanded to **initialize** variables properly. Otherwise, the variable not initialized will have a garbage value already available at it's memory location, and it may lead to an unexpected behaviour and/or results.

Arrays

- An array is a collection of data elements of the same type (generally) that satisfies the following conditions:
  - A collection of data that have the **same type** (bool, int, double, string...)
  - Items are stored **contiguously** in memory (image 6)
  - The **size of the array** must be known when declaring the array.
  - Considering an array of size *N*, the **index of the first element** of the array is 0, while the **index of the last element** is $N - 1$. (in most programming languages: C/C++, Python etc.)
  - The elements of an array are **accessible** from their **position**, hence their **index**.

Example of array representation
in memory of an array of ints

Memory Location	400	404	408	412	416	420
Index	0	1	2	3	4	5
Data	18	10	13	12	19	9

Figure: arrayExample

Arrays

- Advantages of arrays:
  - **Code optimization:** the complexity to add or delete an element at the end of the array, access to an element *via* its index, add an element or delete it is in $O(1)$. The **sequential** (linear) search for an element in an array has a complexity of $O(n)$.
  - **Ease of access:** direct access to an element *via* its index (e.g: myArray[3]), iterating over the elements of an array using a loop, simplified sorting.
- Drawbacks of arrays:
  - The constraint of having to declare its size prior to **compile time** (what is called a **static array**, we will come back to this), and that the size of the array does not grow during **runtime**.
  - Inserting and deleting items can be expensive as items have to be managed according to the newly allocated memory area (Example: wanting to add an item in the middle of an array).

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n]$: creates an array of size *n* (*n* elements) of type textbftypename with random values. **Example: int a**$[3]$ produces:

```
Index: 0 | address: 0x7ffee253c4c0 | value: -497826512
Index: 1 | address: 0x7ffee253c4c4 | value: 32766
Index: 2 | address: 0x7ffee253c4c8 | value: 225226960
```

  - **typename myArray**[n]=$\{v_1, ..., v_n\}$: creates an array of size *n* of type **typename** with values initialized to v_1 until v_n. **Example: int a**$[3] = \{1, 4, 9\}$ creates an array of 3 elements initialized to 1, 4 and 9 respectively.

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{v, ..., v\}$: creates an array of size *n* of type **typename** with all values initialized to *v*. **Example: int a**$[5] = \{1, 1, 1, 1, 1\}$ creates an array of 5 elements, all initialized to 1.

```
Index: 0| address: 0x7ffeea95b510| value: 1
Index: 1| address: 0x7ffeea95b514| value: 1
Index: 2| address: 0x7ffeea95b518| value: 1
Index: 3| address: 0x7ffeea95b51c| value: 1
Index: 4| address: 0x7ffeea95b520| value: 1
```

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{\}$: creates an array of size *n* of type **typename** with all values initialized to 0. **Example: int a**$[4] = \{\}$ creates an array of 4 elements, all initialized to 0.

```
Index: 0| address: 0x7ffee3257510| value: 0
Index: 1| address: 0x7ffee3257514| value: 0
Index: 2| address: 0x7ffee3257518| value: 0
Indexes: 3| address: 0x7ffee325751c| value: 0
```

- **typename myArray**$[n] = \{v\}$: creates an array of size *n* of type **typename** with the first value initialized to *v*, the rest to 0 .

Pointers
Introduction to Pointers

Definition: Compile time is the period when the soure program code (such as C, C++, C#, Java, or Python) is converted to binary code.

Definition: Run time is the period of time when a program is running and generally occurs after compile time.

Pointers
Introduction to Pointers

Definition: A **memory address** or **address** is the location of where a variable is stored on the computer. An address is always a non-negative integer. The address of a variable can be accessed using ampersand (&) operator, it denotes the address of the variable.

Definition: Memory allocation is the process of reserving (virtual or physical) **computer space** or **location** for a computer program to run.

Pointers
Introduction to Pointers

- Two main types of memory allocation exists:

Definition: Static Memory Allocation at Compile Time (referred also to Compile Time Memory Allocation). It occurs when declaring a variable, the compiler allocates automatically a memory location for it.

Definition: Dynamic Memory Allocation at Run Time (referred also to Run Time Memory Allocation). It occurs when memory can be allocated for data variables after the program begins execution.

Pointers

Definition: Pointers, or pointer variables, are *special* variables that are used to store addresses. A pointer then is a variable that stores the memory address of another variable as its value.

One should distinguish two types of defining a pointer:

- Defining a pointer at the **STACK**. Roughly speaking, is when declaring a pointer pointing into a local variable or function. **This article** explains how function calls resolves in a C program, particurarly how local variables and hence pointers on local vars are processed.
- Defining a pointer at the **HEAP**. Is when you are declaring a pointer dynamically using either malloc(), calloc() or realloc() stdlib built-in functions.

Pointers

Assume we have the declaration of a variable of type **typename** and name **var**:

```
// Declaration:
typename var;
```

Here we are declaring a pointer of type **typename *** and name **pvar**:

```
// Declaration:
typename * pvar;
```

Pointers

- Some important notes:
  - **pvar** is simply a convention in order to be consistent about the naming of our variables. Here we are startit the name pvar with letter p in order to say that it is a pointer, pointing on variable var. Hence **pvar** as the name of our pointer.
  - **typename** have to be one of the built-in types: char, int, float, double.

Important note: Let's now point the pointer **pvar** to variable **var**:

```
// Definition:
pvar = &var;
```

Pointers

- An important fact about pointers is:
  - ***pvar** refers to the value of **var**.
  - **pvar** refers to the address of variable **var**: (**&var**).
  - **&pvar** refers to the address of the pointer (which is simply a variable that owns an address).

Pointers

Let’s now explain with a real example how it is done:

```
// Declaration of a variable f
float f;
// Initializing f to 1.2
f = 1.2;
// Declaration of pointer pf
float * pf;
// Setting the pointer pf to the address of f
pf = &f;
// Now pf is pointing to f.
```

Pointers

One can define directly a pointer as follows:

```
// Declaration of a variable f
float f;
// Initializing f to 1.2
f = 1.2;
// Definition of pointer pf
// Declaring pointer pf and setting it to the address of f
float * pf = &f;
// Now pf is pointing to f.
```

A more real example is given in this example: **pointersExample.c**

Strings

Definition: A **string** in C is an array of characters which is terminated by a null character.

- The following **Definition** creates a string named **myString** that consists of the word ”Programming”.
- In order to hold the terminated character (null character) at the end of the array, the size of the character array which contains the string **must be** one character more than the number of characters in the word ”Programming”.

```
char myString[12] = {'P','r','o','g','r','a','m','m','i','n','g','\0'};
```

Strings

- One may declare the same string by following the rules of array initialization (**literal string it will be**), and without writing the null terminated character, the following example highlight the how:

```
char myString[] = "Programming";
```

Here is the memory representation of the previously defined character array "Programming":

Figure: String Representation in Memory

Strings

- The header file **#include<string.h>** contains several functions that can be performed into strings, and string literals. Here is a non-exhaustive list:

  - **strcpy(s1, s2);** It will copies string s2 into string s1.
  - **strcat(s1, s2);** It concatenates string s2 onto the end of string s1.
  - **strlen(s);** It returns the length of the string s.
  - **strcmp(s1, s2);** It returns 0 if s1 and s2 contains the same characters; a negative number if s1<s2 and a positive one if s1>s2. More details may be found in **this link**.
  - **strchr(s, c);** It returns a pointer to the first occurrence of character c in string s.
  - **strstr(s1, s2);** It returns a pointer to the first occurrence of string s2 in string s1.

Structures

Definition: In C language, a **struct** or structure, is a collection of variables that may be of different types, under a single name.

We define a structure as follow:

```
struct structTag {
        typename_1 var_1;
        typename_2 var_2;
        ...
        typename_n var_n;
} structTypeTag;
```

Structures

A complete example may be found at: **structExample.c**.

Structures
Pointers to structures

A complete example may be found at: **pointersStruct.c**.