FUNDAMENTALS OF PROGRAMMING WITH C LANGUAGE

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." – Martin Fowler.

By Aphrodice Rwagaju

Rwanda Coding Academy

CHAP 1. INTRODUCTION TO C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc. ANSI C standard emerged in the early 1980s, it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. C was initially designed for programming UNIX operating system. Now the software tools as well as the C compiler are written in C. Major parts of popular operating systems like Windows, UNIX, Linux are still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices, one needs to write device driver programs. These programs are exclusively written in C. C seems so popular because it is **reliable**, **simple and easy to use**. Often heard today is – "C has been already superseded by languages like C++, C# and Java."

1.1 Program

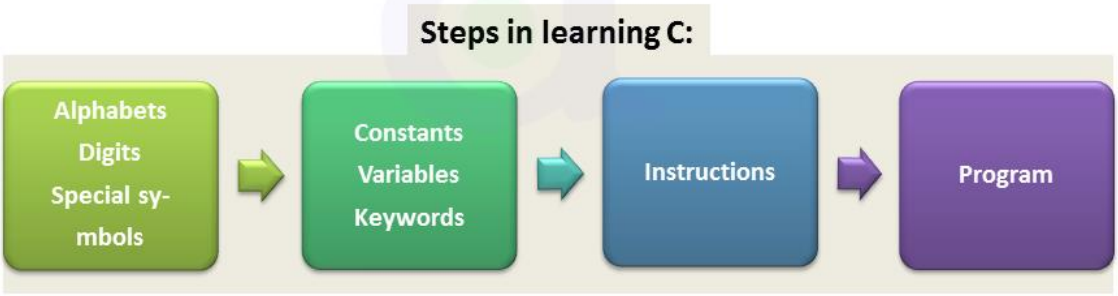
There is a close analogy between learning a language like English and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them. Constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program.

So, a **computer program** is just a collection of the instructions necessary to solve a specific problem when it is executed by a computer. The basic operations of a computer system form what is known as the **computer's instruction set**. And the approach or method that is used to solve the problem is known as an **algorithm**.

A computer program is usually written by a **computer programmer** and can be written in either high-level or low-level languages, depending on the task and the hardware being used.

1.2 Programming language

A **programming language** is a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages consist of instructions for computers and they are used in computer programming to implement specific algorithms.



Programming languages are divided into two different levels:

1. Low level language
2. High level language

1.2.1 Low-level language

Low level languages are **machine level** and **assembly level** language.

In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form of binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone.

The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word such as ADD, SUM, MOV etc. It is easy to write and understand but not understood by the machine. So the translator used here is **assembler** to translate into machine level. Although the language is bit easier, programmer has to know low level details related to low level language. In the assembly level language, the data are stored in the computer register, which varies for different computer. Hence, the **assembly language is not portable**.

1.2.2 High level language

These languages are machine independent, means **they are portable**. Examples of languages in this category are Pascal, Cobol, Fortran, C, Python, Java, etc.

High level languages are not understood by the machine. So it needs to be translated into machine level by the translator.

1.2.2.1 Translator

A **translator** is software which is used to translate high level language as well as low level language in to machine level language.

There are three types of translators namely: **Compiler, Interpreter** and **Assembler.**

Compiler and **interpreter** are used to convert the high-level language into machine-level language.

The program written in high-level language is known as source program and the corresponding machine-level language program is called as object program.

Both compiler and interpreter perform the same task but how they work is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. Interpreter checks error, statement by statement and hence this process takes more time.

An **assembler** translates assembly language into machine code and is effectively a compiler for the assembly language, but can also be used interactively like an **interpreter**.

Assembly language uses words called '**mnemonics**', such as **LOAD**, **STORE** and **ADD**.
The instructions are specific to the hardware being programmed because different
CPUs use different programming languages.

And finally, every assembly language instruction is translated into a single machine
code instruction.

Difference between Compiler and Interpreter

Compiler	Interpreter
Translates the whole program to produce the executable object code.	Translates and executes one line at a time.
Compiled programs execute faster as they are already in machine code.	Interpreted programs take more time to execute because each instruction is translated before being executed.
Users do not need to have the compile installed on their computer to run the software.	Users must have the interpreter installed on their computer and they can see the source code.
Users cannot see the actual source code when you distribute the program.	Users can see the source code and can copy it.
Used for software that will run frequently or copying software sold to a third party.	Used for program development and when the program must be able to run on multiple hardware platforms.

Difference between low-level and high-level languages

Type of Language	Example Language	Description	Example Instructions
High-level Language	Pascal, Cobol, Fortran, Visual Basic, C, C++, Python, Java	Independent of hardware (portable). Translated using either a compiler or interpreter. One statement translated into many machine code instructions.	payRate = 10000 hours = 33.5 salary = payRate * hours
Low-level Language	Assembly language	Translated using an assembler. One statement translated into machine code instruction.	LDA181 ADD93 STO185
	Machine Code	Executable binary code produced either by a compiler, interpreter or assembler.	11010010101000001110101000101101

1.2.3 Advantages of high-level and low-level programming language

1.2.3.1 Advantages of high-level language

Most software are developed using high-level languages for the following reasons:

- High-level languages are relatively easy to learn and much faster to program in.
- Statements within these languages look like the natural English language, including mathematical operators making it easier to read, understand, debug and maintain.
- Complexed assignment statements such as **x=(sqr(b^2-4*a*c))/(2*a)**
- Allowing the programmer to show how the algorithm will solve a problem in a clearer and more straightforward way.

- Specialized high-level languages have been developed to make the programming as quick and easy as possible for particular applications such as, SQL specially written to make it easy to search and maintain databases. HTML, CSS and JavaScript were also developed to help people create web pages and websites.

1.2.3.2 Advantages of low-level languages

- Assembly language is often used in embeded systems such as systems that control a washing machine, traffic lights, robots, etc.
- It has features that make it suitable for the following types of applications:
    - It gives complete control to the programmer over the system components, so it can be used to control and manipulate specific hardware components.
    - It has very efficient code that can be written for a particular processor architecture, so will occupy less memory and execute (run) faster than a compiled/interpreted high-level language.

1.3 Integrated Development Environments (IDE)

Integrated Development Environment or IDE for short is an application or software which programmers use for programming.

An IDE is a windows-based program that allows programmers to easily manage large software programs. It often helps them in editing, compiling, linking, running, and debugging programs.

On Mac OS X, CodeWarrior, Xcode and Eclipse are examples of IDEs that are used by many programmers. Under Windows OS, Visual Studio Code, Eclipse, NetBeans, CodeBlocks are good example of a popular IDEs. Kylix, Dev-C++, Eclipse, NetBeans are popular IDEs for developing applications under Linux OS.

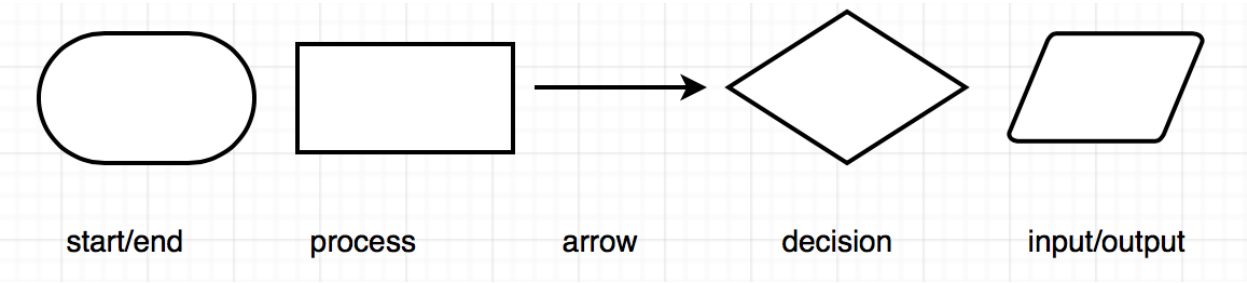
Most IDEs also support program development in several different programming languages in addition to C, such as C#, C++, Java, etc.

CHAP 2. ALGORITHM

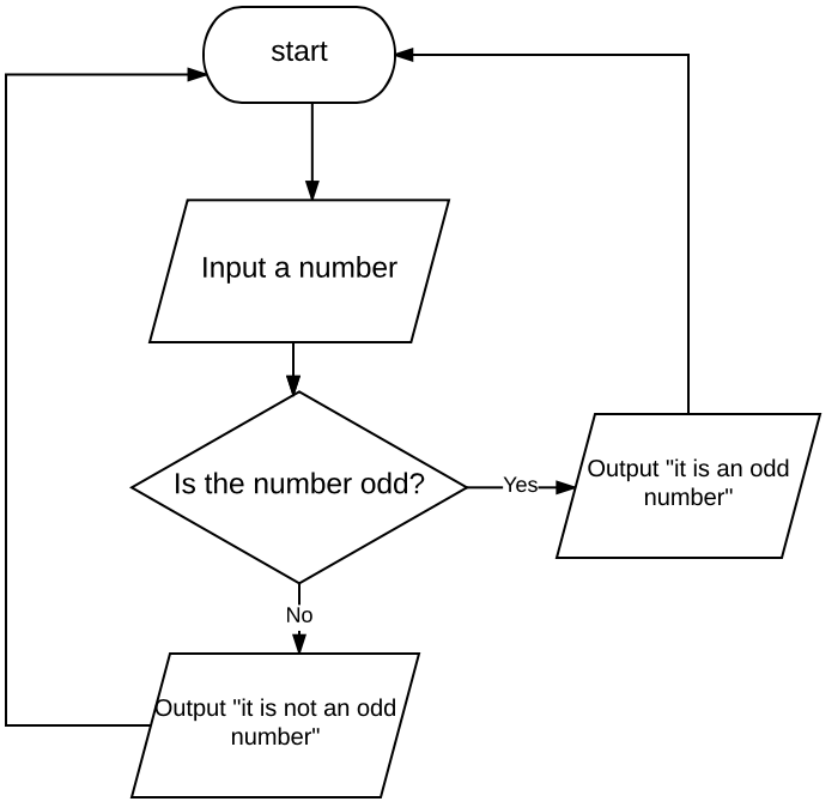
An **Algorithm** specifies a series of steps that performs a particular computation or task. It is a step by step procedure of solving a problem. It was originally born as part of mathematics from the Arabic writer Muḥammad ibn Mūsā al-Khwārizmī, and now strongly associated with computer science. A good algorithm means an efficient solution to be developed.

Flowchart

Flow chart symbols



Example of flow chart



2.1. Qualities of a good algorithm

1. Input and output should be well defined with precision.
2. Each step in an algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm should be human understandable, it shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.
5. A good algorithm should be programming language independent, i.e can be implemented by any software developer using any programming language.

2.2. Real life Example

Producing a bread at the bakery

To bake a **bread,** we must have some followed steps:

0. Start
1. Preheat the oven;
2. Mix flour, sugar, and other ingredients;
3. Pour into a baking pan;
4. Heat for some times
5. … …

n-1. Get the bread out of the baking pan

n. Stop

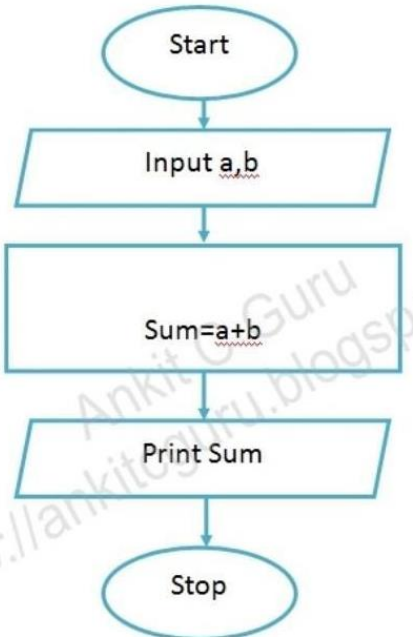
2.3. Programming Examples of Algorithm

a) Calculate the sum of two numbers (A and B) and print the sum to the console
   1. Start
   2. Declare number type variables: number one A, number two B and the sum
   3. Input A
   4. Input B
   5. Compute A+B into Sum (Sum ← A+B)
   6. Output Sum
   7. Stop

Pseudo code

```
var A,B, Sum;
get A;
get B;
Sum=A+B;
print Sum;
```

Flow chart



b) Find the maximum number in a collection of numbers
c) Calculate the sum and average of all even numbers between 1 and n
d) Write an algorithm to find the largest among three different numbers
e) Write an algorithm to check whether a number entered by user is prime or not

CHAP 3: C DATA TYPES

Broadly, there are 5 different categories of data types in the C language, they are:

Category	Example
Basic	character, integer, floating-point, double.
Derived	Array, structure, union, pointer, function.
Enumeration	Enums
Bool type	true or false
Void	Empty value

3.1 C Primary Data types

The C language has 5 basic (primary or primitive) data types, they are:

1. **Character** - ASCII character set or generally a single alphabet like **'a'**, **'B'**, etc.
2. **Integer** - Used to store whole numbers like **1**, **2**, **100**, **1000**, etc.
3. **Floating-point** - Decimal point or real numbers values like **99.9**, **10.5**, etc.
4. **Double** - Very large numeric values which are not allowed in Integer or Floating point type.
5. **Void** - This means no value. This data type is mostly used when we define functions.

There are different keywords to specify these data types, the keywords are:

Datatype	Keyword
Character	Char
Integer	Int
Floating-point	Float
Double	Double
Void	Void

Each data type has a **size** defined in **bits/bytes** and has a **range** for the values that these data types can hold.

3.2 Size of different Datatypes

The size for different data types depends on the compiler and processor types, in short, it depends on the Computer on which you are running the C language and the version of the C compiler that you have installed.

3.2.1 char

The char datatype is **1 byte** in size or **8 bits**. This is mostly the same and is not affected by the processor or the compiler used.

3.2.2 int

There is a very easy way to remember the size for int datatype. The size of int datatype is usually equal to the word length of the execution environment of the program. In simpler words, for a **16-bit environment**, int is **16 bits** or **2 bytes**, and for a **32-bit environment**, int is **32 bits** or **4 bytes**. Unfortunately, this rule is not applicable in **64-bit environment** where the size remains as the same as in **32-bit environment.**

3.2.3 float

The float datatype is **4 bytes** or **32 bits** in size. It is a **single-precision data type** that is used to hold decimal values. It is used for storing large values.

float is a faster data type as compared to double, because double data type works with very large values, hence it is slow.

3.2.4 double

The double datatype is **8 bytes** or **64 bits** in size. It can store values that are **double the size of what a float data type can store**, hence it is called double.

In the 64 bits, **1 bit** is for **sign** representation, **11 bits** for the **exponent**, and the rest **52 bits** are used for the **mantissa**.

The double data type can hold approximately **15 to 17 digits**, before the decimal and after the decimal.

3.2.5 void

The void data type is **0 bytes** means nothing, hence it doesn't have a size.

Before moving on to the range of values for these data types, there is one more important concept to learn, which is **Datatype modifiers**.

3.3 C Datatype Modifiers

In the C language, there are **4 datatype modifiers**, that are used along with the basic data types to categorize them further.

For example, if you say, there is a playground, the other person will know that there is a playground, but you can be more specific and say, there is a Cricket playground or a Football playground, which makes it even more clear for the other person.

Similarly, there are modifiers in the C language, to **make the primary data types more specific**.

Following are the modifiers:

1. signed
2. unsigned
3. long
4. short

As the name suggests, signed and unsigned are used to represent the **signed (+ and -)** and **unsigned (only +) values** for any data type. And **long** and **short** affects the **range of the values** for any datatype.

For example, **signed int**, **unsigned int**, **short int**, **long int**, etc. are all valid data types in the C language.

Now let's see the range for different data types formed as a result of the 5 primary data types along with the modifiers specified above.

3.4 C Datatype Value Range

In the table below we have the range for different data types in the C language.

Type	Typical Size in Bits	Minimal Range	Format Specifier
Char	8	-127 to 127	%c
unsigned char	8	0 to 255	%c
signed char	8	-127 to 127	%c
Int	16 or 32	-32,767 to 32,767	%d, %i
unsigned int	16 or 32	0 to 65,535	%u
signed int	16 or 32	Same as int	%d, %i
short int	16	-32,767 to 32,767	%hd
unsigned short int	16	0 to 65,535	%hu
signed short int	16	Same as short int	%hd
long int	32	-2,147,483,647 to 2,147,483,647	%ld, %li
long long int	64	$-(2^{63} - 1)$ to $2^{63} - 1$ (Added by C99 standard)	%lld, %lli
signed long int	32	Same as long int	%ld, %li
unsigned long int	32	0 to 4,294,967,295	%lu

Type	Typical Size in Bits	Minimal Range	Format Specifier
unsigned long long int	64	2^{64} - 1 (Added by C99 standard)	%llu
Float	32	1E-37 to 1E+37 with six digits of precision	%f
Double	64	1E-37 to 1E+37 with ten digits of precision	%lf
long double	80	1E-37 to 1E+37 with ten digits of precision	%Lf

As you can see in the table above, with different combinations of the datatype and modifiers the range of value changes.

When we want to print the value for any variable with any data type, we have to use a **format specifier** in the **printf()** statement.

What happens if the value is out of Range?

Well, if you try to assign a value to any datatype which is more than the allowed range of value, then the C language compiler will give an error. Here is a simple code example to show this,

```
#include <stdio.h>
int main() {
  // allowed value up to 65535
  unsigned short int x = 65536;
printf("%d\n", x);
  return 0;
}
```

```
warning: large integer implicitly truncated to unsigned type [-Woverflow]

unsigned short int x = 65536;

^
```

When a **type modifier is used without any data type**, then the int data type is set as the default data type. So, unsigned means unsigned int, signed means signed int, long means long int, and short means short int.

What does signed and unsigned means?

This is a little tricky to explain, but let's try.

In simple words, the unsigned modifier means **all positive values**, while the signed modifier means **both positive and negative values**.

When the compiler gets a numeric value, it converts that value into a binary number, which means a combination of 0 and 1. For example, **32767** in binary is **01111111 11111111**, and **1** in binary is **01 (or 0001)**, **2** is **0010**, and so on.

In the case of a **signed integer**, the **highest order bit** or the first digit from left (in binary) is used as the **sign flag**. If the **sign flag is 0**, the number is **positive**, and if it is **1**, the number is **negative**.

And because one bit is used for showing if the number is positive or negative, hence there is one less bit to represent the number itself, hence the range is less.

For **signed int**, **11111111 11111111 means -32,767** and because the first bit is a **sign flag** to mark it as a negative number, and rest represent the number. Whereas in the case of **unsigned int**, **11111111 11111111** means **65,535**.

A simple program to display the size of different variables:

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

  printf("size of char: %u bytes\n", sizeof(char));
  printf("size of signed char: %u bytes\n", sizeof(signed char));
  printf("size of unsigned char: %u bytes\n", sizeof(unsigned char));
  printf("size of int: %u bytes\n", sizeof(int));
  printf("size of short int: %u bytes\n", sizeof(short int));
  printf("size of long int: %u bytes\n", sizeof(long int));
  printf("size of long long int: %u bytes\n", sizeof(long long int));
  printf("size of signed int: %u bytes\n", sizeof(signed int));
  printf("size of unsigned int: %u bytes\n", sizeof(unsigned int));
  printf("size of float: %u bytes\n", sizeof(float));
  printf("size of double: %u bytes\n", sizeof(double));

  printf("size of long double: %u bytes\n", sizeof(long double));
  printf("size of an integer number: %u bytes\n", sizeof(439283));
  printf("size of a real number: %u bytes\n", sizeof(43.9283));
  printf("size of \"sizeof\" long double: %u bytes\n", sizeof(sizeof(long double)));

 return 0;
}
```

The output:

```
size of char: 1 bytes
size of signed char: 1 bytes
size of unsigned char: 1 bytes
size of int: 4 bytes
size of short int: 2 bytes
size of long int: 4 bytes
```

```
size of long long int: 8 bytes
size of signed int: 4 bytes
size of unsigned int: 4 bytes
size of float: 4 bytes
size of double: 8 bytes
size of long double: 16 bytes
size of an integer number: 4 bytes
size of a real number: 8 bytes
size of "sizeof" result: 8 bytes

Process returned 0 (0x0)   execution time : 0.011 s
Press any key to continue.
```

3.5 C Derived Datatypes

While there are 5 primary data types, there are some derived data types too in the C language which are used to store complex data.

Derived data types are nothing but primary data types but a little twisted or grouped together like an **array**, **structure**, **union,** and **pointers**. The derived datatypes will be discussed in detail later.