Interpretation of data in C programming language (how is machine code stored in memory?)

Sahil Sharma

Computer Science and Engineering Amity University, Haryana Gurgaon, India
Email: sahilsharma_081995@yahoo.com

Abstract—there are millions of programmers in this entire world. Every programmer, be it any programming language has a goal to learn its syntax, the logics, predefined functions and all the programming related concepts that the programming language offers. But very few programmers has the curiosity to understand how data are stored in the memory. It's necessary to learn how coding make things happen but if you add up knowledge of how that coding is stored in computer memory, it intrigues you to go into the unmatched depth as well as gives you a vision to think in a different way that could be productive as a programmer. As C is arguably the first language that changed the world, it is a great choice to deal with C. It also follows a good approach of American Standard Code for Information Interchange (ASCII) for storing data. This paper will deal with how data are stored in the machine when the C compiler converts the source code into machine code, the interaction of operating system, CPU and the programming language on various occasions, how strings and characters are interpreted by the memory and a few more things. After this study the reader will feel slightly confident about knowing how programming in C happens on the inside.

Keywords—ASCII; programming language; nibble; unicode;

I. INTRODUCTION

The only language that the computer comprehend is binary that is 1's and 0's. Everything in a computer system is encoded in binary like texts, images, movies, audios and so on. The representation is done in binary as it is the quickest and the shortest way to represent values.

Consider an example of Bit Map Picture (BMP) which is a simple graphics file format. If a picture is drawn using paint, one of the various ways to save that picture is in the form of BMP files.

0100 1100 0101 0011←represents the file format (.bmp): 2 bytes

0110 0000 0101 1110 0011 1110 0001 0001← represents the file size: 4 bytes

The first 16 bits (2 bytes) always represent the file format of bmp files or in other words, files having .bmp extension in an image. The first two bytes represent the file format, be it bmp, jpeg, png, gif etcetera. It will be different set of bits for jpeg, different for png and so on. It has been fixed by the designers of the operating system. It works like, as soon as the user saves the file as .bmp extension, the computer fixes the 2 bytes for file format that enables it to distinguish from other files. The bytes that follow represent the file size that could be of any size. This is a simple instance of how image is stored in binary. Spacing after every nibble (4 bits) is a way of representing group of bits to make it look more readable otherwise there is no spacing in actual computer system. This is just an instance to show how computer stores the instruction that is performed within a couple of clicks on the desktop. These binary bits are generated in nanoseconds. This is a basic instance of how differently data is perceived by us and the system.

It is essential to know about how written code is expected to be stored before going into the literal binary code. For example, explaining how 'int x' will be stored in memory in explanatory terms, after which binary code could be understood.

II. PROGRAMS ARE DATA TOO

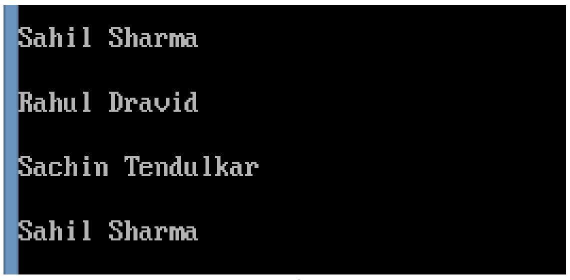
A. Interaction of programs with memory and CPU

- All sorts of programs are stored and retrieved using Random Access Memory (RAM).
- Operating system ensures that there is enough space in the RAM for the program data. RAM used by one program cannot be used by any other program and hence ambiguity is averted.
- Consider a string of text, say "Hello Sahil", we had to know at which address this string will be stored in RAM but C allows English words and itself keep track of the binary numbers.
- This explains that why there is no requirement to rote the string of bits associated with the string "Hello Sahil", because programming language takes care of it by keeping track of the binary addresses associated with the English coding.

- CPU keeps track of the address in memory. Just like pointer in C, the CPU has a pointer of its own called internal address tracker that points to the address of the next instruction.

- This gives the advantage of reusability too as execution of the previous instructions is done by pointing the instruction pointer backwards.

- For example consider a simple C program that has to print:-



```
Sahil Sharma

Rahul Dravid

Sachin Tendulkar

Sahil Sharma
```

Fig 1: Output of a C program

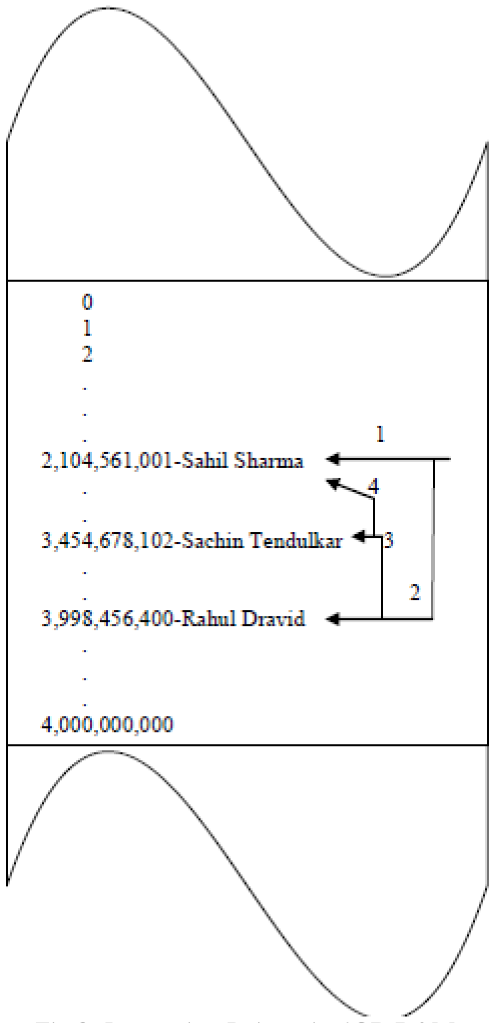

0
1
2
2,104,561,001-Sahil Sharma
3,454,678,102-Sachin Tendulkar
3,998,456,400-Rahul Dravid
4,000,000,000
1
2
3
4

Fig 2: Instruction Pointer in 4GB RAM

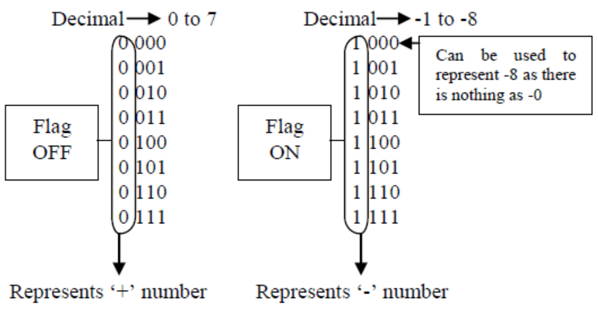
Fig 2 explains how the instruction pointer will rotate within the RAM. Firstly, the instruction pointer will execute the string “Sahil Sharma” (1) then it will pass on to the string “Rahul Dravid” (2). The important thing to note is that the same string is not saved into memory again as the instruction pointer is flexible to move in both forward and reverse directions. Therefore, it moves to “Sachin Tendulkar” (3) and then again “Sahil Sharma” (4) in reverse direction.

The process of executing one instruction after the other according to the sequence is called program flow. As soon as the program is loaded into memory and executed, the first sequence of ones and zeroes will be executed. CPU works in accordance to the command of the instruction of the program, that is, program flow is controlled by the programmer.

B. Numbers in C

There are two types of numbers in C, namely signed and unsigned.

1) Signed Numbers: In simplest terms, whenever encoding of a number is done in binary where the number can be both positive and negative, it is known as a signed number and this method is called signed magnitude.



0000 and 1000 represents positive 0 and negative 0 respectively but there is nothing as such as positive or negative 0. So, 8 bits are required to store the positive numbers from 0-7 and 7 bits are required to store negative numbers from -1 to -7. So, in total 15 bits are required to store both positive and negative numbers from 0-7.

In C, a signed integer is declared using ‘signed int var_name’ or simply ‘int var_name’. C is said to store two bytes for an integer.

It is understandable that integer stores 2 bytes in C but what will be the maximum (+) and the minimum value (-) that 2 bytes can accommodate? Say that the total number of possibilities is T then,

((T/2)-1) possibilities to store **positive values**

$\xrightarrow{}$ Subtraction by 1 because 0 is also stored. So 1 bit is required for storing that 0.

$$((T/2)-1) = (2^{16}/2)-1 \xrightarrow{} \boxed{\text{No. of bits}}$$
$$=327267$$

No number greater than 327267 can be stored in 2 bytes (16 bits) of memory.

Similarly, **(-T/2)** possibilities to store negative values, 1 won't be subtracted here because no bit is required to store a negative 0, as nothing as such exists.

$$(-T/2) = (-2^{16}/2) \xrightarrow{} \boxed{\text{No. of bits}}$$
$$= -327268$$

No number smaller than -327268 can be stored in 2 bytes (16 bits) of memory.

Therefore, this is how the range of integer in C is.

From -327268 to 327267. [3]

2) Unsigned Numbers: Whenever encoding of a number is done in binary where the number can be only positive, it is known as a signed number. This is used when there is surety that the number that will be stored in the variable will be positive. Unsigned numbers are represented the same way as signed where the Most Significant Bit (MSB) does not mean any sign but the next set of bits.

Decimal $\rightarrow$ 0 to 7	Decimal $\rightarrow$ 8 to 15
0000	1000
0001	1001
0010	1010
0011	1011
0100	1100
0101	1101
0110	1110
0111	1111

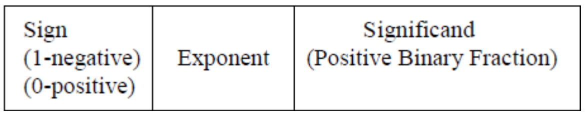
Similarly, say that T is the total number of possibilities, then, there are exactly **T-1 possibilities** to store **unsigned numbers**. Therefore, unsigned integers can store:-

$$T-1=2^{16}-1=65535$$

Therefore, the 2 bytes used for unsigned integer in C can store the maximum value of 65535 using the above stated formula.

C. Fractions in C

Fractions are effortless to write in decimal form. The decimal 2.5 in binary can be written as 0010.1001, but the question is how will the computer understand the radix (as it understands only binary)? Is there a set of bits to represent the radix? The answer is no. 2.5 is written just as 0010 1001 in binary and the compiler is told that the number that will be written will be fractional. This is told to the compiler by the programmer by using a **place holder**. A place holder or format specifier never gets printed but tells the compiler what format of data has been used. For example '%f' in C will tell the compiler that there has to be a fraction printed or taken as input and not any other. The compiler then generates a corresponding binary. After that, the CPU stores that fractional value in the following way that explains everything.



Sign (1-negative) (0-positive)	Exponent	Significand (Positive Binary Fraction)

Fig 3: Method to store fractions in CPU

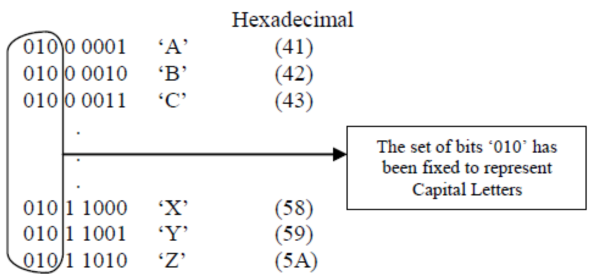
III. ENCODING OF CHARACTERS, NUMBERS AND STRINGS

This section of the paper will determine how characters, numbers and strings are represented in American Standard Code for Information Interchange (ASCII) and more importantly, how that representation is encoded in binary when they are defined in C programs. The first usage of ASCII was during 1963 as a seven-bit tele printer code for American Telephone & Telegraph's TWX network [1]. In its early days ASCII was known as Bemer-Ross code in Europe [4]. C follows an ASCII approach to convert the source code data into corresponding binary. Other languages might follow different approaches like Unicode (like Java). Each of the segment will be discussed individually.

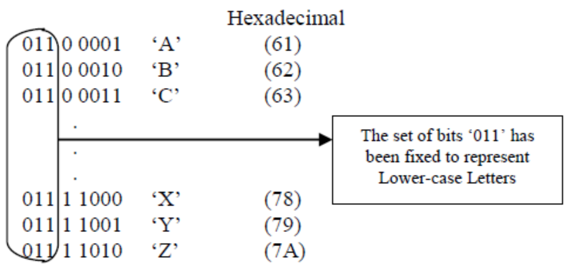
A. Character Encoding

A character in C is of 8 bits (this is not universal). The corresponding 8 bits of the character is taken globally by the ASCII table which gives a corresponding hexadecimal (base 64) value.

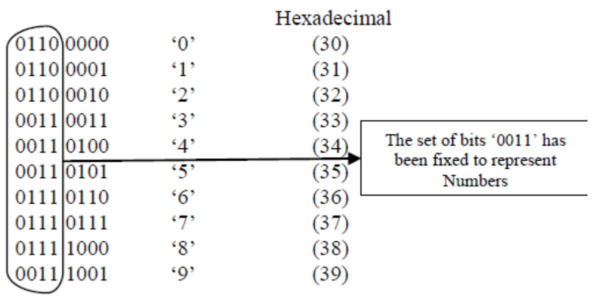
1) Capital Letters: Of the 8 bits, the first 3 bits are always **010** which is fixed to differentiate capital alphabet from other characters. Rest of the 5 bits represents 1, 2, 3....26 for the 26 English characters (A-Z).



2) Lower-case letters: In this case, the first 3 bits are fixed as **011** and the rest of the five bits represent the lower-case letters (a, b, c…z).



3) Numbers: Interestingly, numbers on the keyboard are not read as actual numbers by the computer. For example, if four is hit on the keyboard the corresponding code wouldn't be 0100. It will be taken as a character and has a way of representation as well. The first 4 bits are fixed for a number as 0011 and the next four as decimal from 0-9.



So, this is how encoding of characters is done in C. ASCII is one such encoding technique and it is not accepted universally.

B. String encoding

Strings is a collection of characters and is not limited to a single byte. It can be as many bytes for example, consider a string of text "123". It looks simple on the source code but the binary code will be as follows: -

0011 0001: 0011 0010: 0011 0011

0011 is the fixed code to represent number and 0001 0010 and 0011 is 1, 2 and 3 respectively. The ':' and spaces are just to make the program more readable and there won't be any such spaces in the memory. Before and after this string of text there will be a series of 1's and 0's that could mean anything. So to avoid ambiguity, a terminating character is added at the end of string which is 8 bits of 0's also known as **null** character (\o). Hence, the exact way the string "123" in memory would be like: -

001100010011001000110011**00000000**

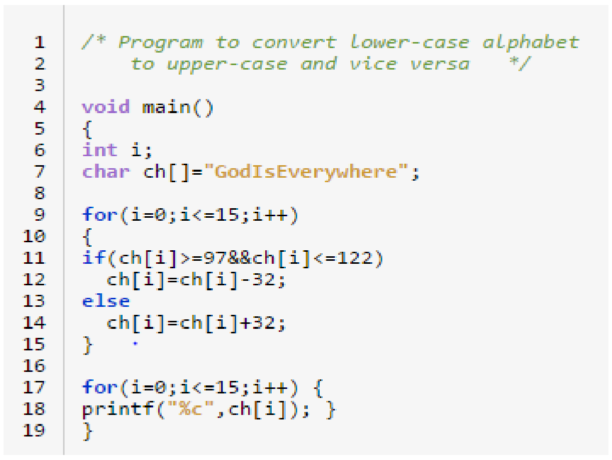
This string of text would require 4 bytes of memory. 1 additional byte to store null character. Programmers usually learn that strings end with a null character without understanding the simple theory behind it. If there is no proper termination of the string, say "123", there is no way to figure out when to stop. The next million bits will be executed until 8 bits of zero is found. For example, consider a music file being played and there is no termination of it. Due to this, that file could be damaged as after its execution a different format or any other string of bits can be executed. To put it in one line, the outcome will be unpredictable. In C, printf function automatically generates 00000000 at the end because the algorithm in C is set so, that anything within double quotations should end like that.

BCPL had already established the null-termination convention and this concept in C was taken from there, the creator of C realized that this convention was better than keeping count of the bits. [2].

IV. USE OF ASCII IN C PROGRAMS

ASCII is not only bounded to understand the storage procedure of the data in memory but is used in the programs itself. For example, programs to know if the user entered character, number or special symbol or a program to find corresponding ASCII value or to find corresponding character or any similar type of program and more. This could be well understood through a sample program.

Consider a program that converts the lower-case letter to upper-case letter and vice-versa.



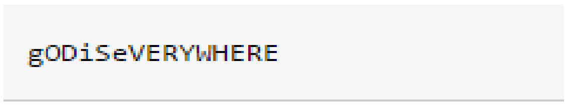
```
/* Program to convert lower-case alphabet
    to upper-case and vice versa   */

void main()
{
int i;
char ch[]="GodIsEverywhere";

for(i=0;i<=15;i++)
{
if(ch[i]>=97&&ch[i]<=122)
  ch[i]=ch[i]-32;
else
  ch[i]=ch[i]+32;
}

for(i=0;i<=15;i++) {
printf("%c",ch[i]); }
}
```

Fig 4: Usage of ASCII in a program



```
gODiSeVERYWHERE
```

Fig 5: Output of the program

This a simple program that briefs about the usage of ASCII and how it plays such an important part of C programming.

V. CONCLUSION

This paper dealt with how certain data are stored in the main memory when the source code is converted to executable code. This is a good step towards building a strong foundation for programing. As programming is the interface between programmers and the CPU there should be an understanding of both the sides. As every coin has two sides, it is essential to know about both the programmer's side and the computer's side, how programmers interpret things and how computers understand things. Programming in today's world is mainly English and Mathematics. A quality programmer is the one who knows the trio of 'that' English, Mathematics and Binary. The programmer must know how each word, each instruction written will be perceived by the machine. This paper dealt with the data, there are a million more interpretation concepts in programming.

VI. FUTURE WORK

As procedural languages has mostly been replaced by object oriented programming languages, it is important to understand about how these languages tend to work. Although, the descendant of C, C++ also works on ASCII, there are languages that works on Unicode as well like Java. ASCII were the most common character encodings until 2007 but has been replaced by UTF-8 which includes ASCII as a subset [5][6][7]. Therefore, it is time to have similar information about Unicode and UTF-8. But not to forget, to understand how data is stored in memory, a procedural language like C is the best that could be taken because object oriented languages like Java use objects to store the data that follows concept like encapsulation and data hiding and therefore memory access is difficult. Future papers will focus more on how data is stored in other languages, specifically an object oriented languages to show how Unicode works. As this paper discussed about the data, there is a scope of writing how file handling in C happens on the inside. Overall, this paper is just the beginning of something interesting and will be followed by some deep concepts of programming taking place on the inside.

ACKNOWLEDGMENT

I would like to put on record, my appreciation and gratitude to all those who have furnished their support and input, most importantly my sister Miss Sakshi Sharma, WASEian, for guiding me to take programming very seriously. I have always received tremendous guidance and support throughout the paper writing from my guide Mr. Ankit Garg, Assistant Professor, Amity University, Haryana. I would like to thank my mother for her words of encouragement.

REFERENCES

[1] Brandel, Mary (July 6, 1999). "1963: The Debut of ASCII". CNN.

[2] Kamp, Poul-Henning (25 July 2011), "The Most Expensive One-byte Mistake", ACM Queue 9 (7), ISSN 1542-7730

[3] ISO/IEC 9899:1999 specification, TC3 (PDF).

[4] Bob Bemer (n.d.). Bemer meets Europe. Trailing-edge.com. Accessed 2008-04-14. Employed at IBM at that time.

[5] Dubost, Karl (May 6, 2008). "UTF-8 Growth On The Web". W3C Blog. World Wide Web Consortium.

[6] Davis, Mark (May 5, 2008). "Moving to Unicode 5.1".

[7] Davis, Mark (Jan 28, 2010). "Unicode nearing 50% of the web".