Unidad 1 - Programación de computadoras

Parte I - Memoria de Computadoras

Laboratorio Avanzado

enero, 2021

1. Introducción

2. La memoria de una computadora
   - Organización
   - Representación binaria y hexadecimal
   - Interpretación de valores de memoria
   - Más allá de 1 byte
   - Conjuntos de instrucciones
   - Tarea

Introducción

El objetivo de esta parte es reafirmar algunos conceptos fundamentales sobre el funcionamiento de la memoria de las computadoras modernas previo a tratar el tema *asignación dinámica de memoria*. Es necesario entender de forma general cómo funciona la memoria dentro de una computadora. La memoria es el lugar donde una computadora almacena tanto datos como las instrucciones que debe ejecutar.

La memoria de una computadora

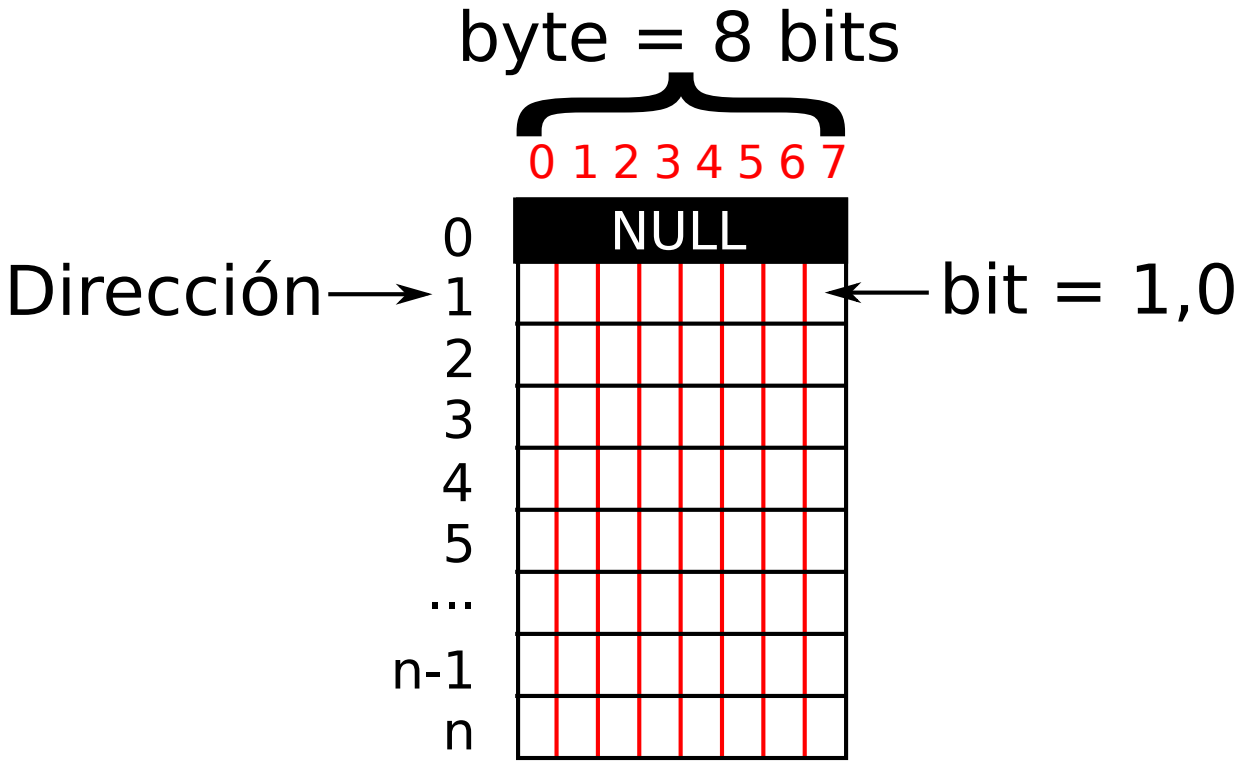
Organización

Una computadora moderna tiene varios tipos de memoria como tal, entre ellos:

- **Registros del procesador**. Esta es la memoria que está integrada directamente en el procesador. Es de acceso muy rápido ya que está totalmente sincronizada con este. Esta memoria es del tipo volátil, esto quiere decir que todo lo que se almacena en ella desaparece al momento de apagar la computadora. Un procesador no tiene una gran cantidad de registros generalmente, siendo este tipo de memoria del orden de unos cuantos MB en el mejor de los casos. Se utiliza principalmente para almacenar las entradas y las salidas de las instrucciones que se están ejecutando en el procesador. Actualmente los procesadores también los utilizan para almacenar porciones de programas que se detectan que son *muy repetitivas*.

- **RAM**: *Random Access Memory* Generalmente es una memoria de acceso rápido, pero más lenta que los registros ya que está físicamente separada del procesador y la comunicación con esta se debe realizar por medio de un *bus de datos*. También es de tipo volátil. En esta es donde usualmente se almacenan los programas que se encuentran en ejecución junto con sus datos. En las computadoras modernas de 64 bits, la memoria RAM es del orden de algunos GB.
- **Memoria permanente**. En esta se encuentra dispositivos como discos duros y almacenamiento externos. Es de acceso mucho mas lento que la RAM. La cantidad de espacio que se puede tener de almacenamiento en forma permanente depende fuertemente de la tecnología con que funciona el dispositivo, variando desde kB hasta PB. La velocidad de lectura y escritura de esta memoria también es muy variable.

En terminos generales, la memoria en una computadora puede ser vista como un arreglo unidimensional de *bytes*, siendo el índice de este arreglo lo que se conoce como la *dirección de memoria*.

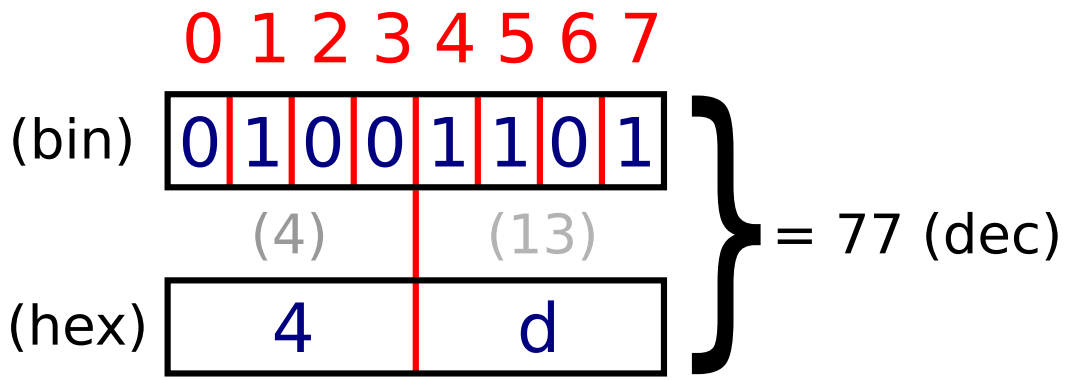


No todas la direcciones de memoria son accesibles desde el espacio de usuario, que es donde corren los programas que se utilizan regularmente. Generalmente los primeros bloques de memoria están reservados y solo son accesibles al núcleo (kernel) del sistema operativo, ya que con ellos se direcciona a dispositivos físicos del computador y son utilizados para comunicarse con estos. Tampoco los registros del procesador son accesibles de manera ordinaria con lenguajes de alto nivel.
Esto hace que desde la programación ordinaria se pueda accedar esencialmente a la memoria RAM, que es donde se realiza la asignación dinámica de memoria. También se puede acceder al almacenamiento del disco duro, pero ese se hace por medio de archivos.

Representación binaria y hexadecimal

La unidad más básica de memoria en una computadorea es el bit, que puede tomar solo dos valores: 1 ó 0. Los bits nunca están vacíos, ya que su valor se determina por el estado del circuito en el que son implementados, que al iniciar una computadora en general, tendrá valores aleatorios, por lo que la memoria *siempre* tendrá algo asignado. Los bit se agrupan en paquetes de 8, los que se llaman *bytes*. Cada byte puede tomar en consecuencia $2^8 = 256$ valores posibles.

Evidentemente la representación natural para los valores de memoria de una computadora es la binaria, pero resulta incómoda para los humanos que estamos acostumbrados a una notación decimal. Es de suma utilidad la notación hexadecimal, ya que un dígito hexadecimal puede tomar $16 = 2^4$ valores, un byte puede se representado de manera exacta con dos dígitos hexadecimales, lo cual representa una abreviación.


0 1 2 3 4 5 6 7
(bin)
0 1 0 0 1 1 0 1
(4)
(13)
(hex)
4
d
= 77 (dec)

Interpretación de valores de memoria

Cómo ya se ha visto, en la memoria se pueden almacenar valores binarios. Lo que representan esos valores es cuestión de como se quiera interpretarlos. En realidad *debemos* saber como interpretar lo que está almacenado en la memoria para que esta tenga sentido. Aquí surge otro tipo de problema, que es cuanta memoria se requiere para almacenar lo que se desea representar teniendo la limitante tener que expresarlo en binarios.

El caso más simple es representar valores booleanos, falso o verdadero. Lo primero que se puede venir a la mente es que para ello solo se necesita un bit, lo cual es cierto. Sin embargo, a nivel de procesador, es muy costoso separar los bits de un byte para utilizarlos individualmente, por lo que se opta siempre por utilizar *bytes enteros*, así que una variable de tipo booleano (`bool`) utiliza 1 byte. Es generalizado que en el caso de variables booleanas se define el valor falso como 0 (`false = 0b00000000 = 0x00`), y se considera cualquier otro caso como verdadero (`true`).

El siguiente tipo de dato que es muy simple es el que representa caracteres (`char`). Con un byte se pueden representar 256 caracteres diferentes. Sin embargo, por razones de compatibilidad, se continua utilizando la codificación ASCII para caracteres, que utiliza solamente 7 bits, con lo que se tienen 128 caracteres. Han existido muchos otros estándares que extienden la codificación ASCII, pero el único que ha ido paulatinamente ganando universalidad es el UTF8.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

El tipo de dato que mas importa almacenar en memoria son números. En este caso, que pareciera que la representación es directa, no es tan simple por dos razones principales: El espacio de memoria es discreto y limitado. Esto tiene como como consecuencia que con un byte, en su forma mas simple, solo se puedan representar enteros positivos de 0 a 255.
Si se desea que los enteros puedan también ser negativos, se puede utilizar el *bit de signo*, este indicará si el número que sigue es positivo o negativo, pero tiene como consecuencia que con un byte ya solo 7 bits sean para el número propiamente, con lo cual los enteros puden ir de -127 a +127.
En la práctica se utilizan sistemas de complemento a 2 para representar números negativos ya que es la elección más conveniente para la aritmética binaria implementada en los procesadores.

En el caso de los números reales, se utilizan formas optimizadas de la notacion de *mantiza-exponente* donde ambos, mantiza y exponente pueden ser positivos o negativos.
A manera de ejemplo: Suponiendo que se utilizan 5 bits para la mantiza y 3 para el exponente. Los números mas grandes que se puden representar son: -16×10^3 (negativos); 15×10^3 (positivos). Los más pequeños: -16×10^{-4} (negativos); 15×10^{-4} (positivos). Siempre con la limitación que tanto la mantiza y el exponente solo pueden variar de forma entera, y por supuesto, se incluye el cero (¡repetido!).
Notese que en todos los casos mencionados, siempre se tienen 256 posibles representaciones.

Boleano	1 1 0 0 1 1 0 1 ➔ verdadero
Caracter	1 1 0 0 1 1 0 1 ➔ (no imprimible)
Entero sin signo	1 1 0 0 1 1 0 1 ➔ 205 8 bits valor
Entero con signo*	1 1 0 0 1 1 0 1 ➔ -77 1 bit signo 7 bits valor
Real*	1 1 0 0 1 1 0 1 ➔ $-9x10^{-1}$ 1 bit signo mantiza 4 bits valor mantiza 1 bit signo exponente 2 bits valor exponente

* Reprecentaciones de ejemplo, NO son las utilizadas en una computadora

Más allá de 1 byte

En el caso de variables numericas, es evidente que un tener solo las 256 posibilidades que se logran con 1 byte no es de mucha utilidad. Para resolver eso, el *tamaño* de una variable generalmente se extiende en varios bytes. Cuántos bytes se utilizan por variable es algo que puede depender de muchos factores, pero finalmente es en el compilador donde se define. El extender la reprecentación de una variable en multiples bytes aumenta el rango de esta, pero sigue siendo discreta y limitada, algo que siempre hay que tener en mente.

```cpp
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    bool        boolVar;
    char        charVar;
    unsigned    unsignedVar;
    int         intVar;
    float       floatVar;

    cout<< "Bytes utilizados por bool:\t"      << sizeof(boolVar)      << endl;
    cout<< "Bytes utilizados por char:\t"      << sizeof(charVar)      << endl;
    cout<< "Bytes utilizados por unsigned:\t"  << sizeof(unsignedVar)  << endl;
    cout<< "Bytes utilizados por int:\t"       << sizeof(intVar)       << endl;
    cout<< "Bytes utilizados por float:\t"     << sizeof(floatVar)     << endl;

    return 0;
}
```

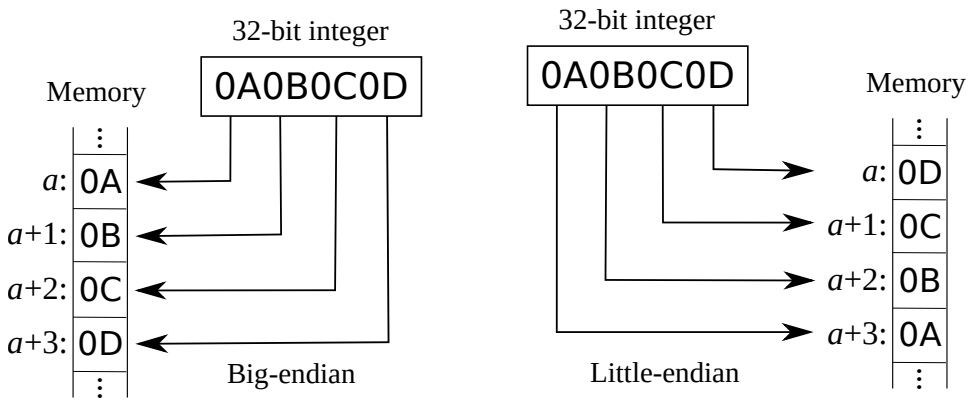
codes/sizes.cpp

```
1 Bytes utilizados por bool:     1
2 Bytes utilizados por char:     1
3 Bytes utilizados por unsigned: 4
4 Bytes utilizados por int:      4
5 Bytes utilizados por float:    4
```

codes/sizes.out

En el caso de una variable que usa 4 bytes, esta tendrá $8x4 = 32$ bits a disposición, esto implica que podra almacenar $2^{32} = 4\,294\,967\,296$ valores diferentes. Si esta es un entero positivo (*unsigned*) su rango ira de 0 a 4 294 967 295. Si es entera con signo, de -2 147 483 648 a 2 147 483 647. En el caso de las reales (*float*) el rango depende de como se implementen las partes de mantiza y exponente.

Cuando una variable ocupa mas de un byte, esta ocupa bytes continuos en el arreglo de memoria.



Fuente: Wikipedia Endianness

Conjuntos de instrucciones

Como se mencionó al inicio, en memoria también se almacenan conjuntos de instrucciones. Estas instrucciones se agrupan en *subrutinas* que al ser llamadas, pueden ser trasladadas al procesador o copidadas a otra dirección de memoria hasta que tengan que ser ejecutadas. En el caso de lenguajes como C, estas subrutinas son las funciones. En C++ el compilador se encarga de *traducir* los elementos del lenguaje de objetos a manera de construir las subrutinas.

Tarea

Tarea 1.I

- Compile, corra y analice el funcionamiento de los códigos:
  - `sizes.cpp`
  - `inter.cpp`
- Investigue la representación binaria con complemento a 2 para números enteros negativos. Calcule la representación a 32 bits de los siguientes números: -125, -4096, -1000000. Escriba un pequeño código en C++ donde asigne directamente el binario a variables enteras y las muestre en pantalla.