REPRESENTACION DE DATOS

Ing. Edgar Ruiz Lizama

RESUMEN

Presenta la forma cómo se representan los datos en una computadora sustentados en conceptos de matemática discreta y prueba los mismos utilizando los lenguajes de programación C y C++.

ABSTRACT

It is described how data is represented in a computer according to discreet mathematics approach. Examples are given using C/ C++.

Representación de Enteros

El tremendo crecimiento de las computadoras se debe particularmente al hecho de que estos dispositivos físicos son cada vez más baratos y se distinguen porque manipulan dos estados a grandes velocidades. De hecho las computadoras son dispositivos que manipulan dos estados (0,1); es decir binario y también con representaciones en octal y hexadecimal los que son utilizados para la representación de datos. Al respecto ver Tabla 1.

Decimal	Octal	Hexadecimal	Binario
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111
16	20	10	10000

Tabla 1: Representación de datos en cada base.

Las operaciones en cada base son análogas a la base 10,. por ejemplo, el número 841 en base 10 es calculado como:

$$841 = 8x10^2 + 4x10^1 + 1x10^0$$

y el número decimal 841.35 base 10 es calculado como

$$841 = 8x10^2 + 4x10^1 + 1x10^0 + 3x10^{-1} + 5x10^{-2}$$

En forma más precisa, si un número X, tiene *n* dígitos antes del punto decimal y *m* dígitos después del punto decimal, se tiene:

$$X = a_{n-1}a_{n-2}....a_1a_0.b_{m-1}b_{m-2}....b_1b_0$$

Luego la base 10 puede representarse como:

$$X = \sum_{j=0}^{n-1} a_j 10^j + \sum_{k=0}^{m-1} b_{m-1-k}10^{-k} \quad 0 \le a_j, b_j \le 9$$

Para la base hexadecimal se tiene:

$$X = \sum_{j=0}^{n-1} a_j 16^j + \sum_{k=0}^{m-1} b_{m-1-k}16^{-k} \quad 0 \le a_j, b_j \le F$$

Para la base octal,

$$X = \sum_{j=0}^{n-1} a_j 8^j + \sum_{k=0}^{m-1} b_{m-1-k}8^{-k} \qquad 0 \le a_j, b_j \le 7$$

Generalizando para cualquier base *r*

$$X = \sum_{j=0}^{n-1} a_j r^j + \sum_{k=0}^{m-1} b_{m-1-k}r^{-k} \qquad 0 \le a_j, b_j \le r$$

El siguiente programa en C++ computa la representación de algunos números en base decimal, octal y hexadecimal para datos de tipo entero.

```
#include <iostream.h>

int x[]={256,1,-1,350,-45,1024,128};

void main()      // bases.cpp
{ int i;
  for(i=0;i<sizeof(x)/sizeof(int);i++)
     {  cout<<endl<<"En decimal " <<dec<<x[i];
        cout<<endl<<"En hexadecimal "
        <<hex<<x[i];
        cout<<endl<<"En octal " <<oct<<x[i]<<endl;
     }
}
```

La salida para *bases.cpp* es:
En decimal 256
En hexadecimal 100
En octal 400

En decimal -1
En hexadecimal ffffffff
En octal 37777777777

En decimal 1
En hexadecimal 1
En octal 1

En decimal 350
En hexadecimal 15e
En octal 536

En decimal -45
En hexadecimal ffffffd3
En octal 37777777723

En decimal 1024
En hexadecimal 400
En octal 2000

En decimal 128
En hexadecimal 80
En octal 200

Notación sin signo

La notación sin signo es usada para representar enteros no negativos. Esta notación no soporta enteros negativos ni números de punto flotante. Un número *n*-bit, *A*, en notación sin signo es representado como:

$$A \equiv a_{n-1}a_{n-2}....a_1a_0$$

con un valor de:

$$A = \sum_{k=0}^{n-1} a_k 2^k \qquad \text{donde} \qquad a_k \in \{0,1\}$$

Los números negativos no son representables en formato sin signo. El rango de los números en *n*-bit sin signo es:

$$0 \le A \le 2^n - 1$$

El cero es el único que está representado en notación sin signo. C++ posee los siguientes tipos de notación sin signo en sus programas.

- **unsigned char** 8 bits
- **unsigned short** 16 bits
- **unsigned int** (tamaño nativo de la máquina)
- **unsigned long** (depende de la máquina)

Nota: El número de bits para cada tipo puede ser dependiente del compilador.

El siguiente programa muestra el rango de los enteros sin signo en C++

```
#include <iostream.h>
#include <limits.h>

void main()      // sinsigno.cpp
{
   cout<<endl<<"Entero sin signo máximo
   "<<UINT_MAX;
   cout<<endl<<"Entero corto sin signo máximo
   "<<SHRT_MAX;
   cout<<endl<<"Entero largo sin signo máximo
   "<<LONG_MAX;
   cout<<endl<<"Carácter sin signo máximo
   "<<UCHAR_MAX;
}
```

La salida para *sinsigno.cpp* es la siguiente:

Entero sin signo máximo 65535
Entero corto sin signo máximo 32767
Entero largo sin signo máximo 2147483647
Carácter sin signo máximo 255

Notación de magnitudes con signo

Los números en magnitud con signo se usan para representar enteros positivos y negativos. La notación de magnitud con signo no soporta números en punto flotante. Un número *n*-bit, *A*, en notación de magnitud con signo se representa como:

$$A \equiv a_{n-1}a_{n-2}....a_1a_0$$

con un valor de:

$$A = (-1)^{a_{n-1}} \left[\sum_{k=0}^{n-2} a_k 2^k \right] \qquad \text{donde} \qquad a_k \in \{0,1\}$$

Un número *A*, es negativo sí y solo sí $a_{n-1} = 1$. El rango de números en *n*-bit notación de magnitud con signo es:

$$-\left(2^{n-1} - 1\right) \le A \le 2^{n-1} - 1$$

El rango de las magnitudes de notación con signo es simétrico y el cero es el único no representado.

El siguiente programa muestra el rango de los enteros con signo en C++.

```
#include <iostream.h>
#include <limits.h>

void main()      // consigno.cpp
{
   cout<<endl<<"Numero de Bits del tipo
   char\t"<<CHAR_BIT;
   cout<<endl<<"Caracter con signo maxi-
   mo\t"<<CHAR_MAX;
```

```
    cout<<endl<<"Caracter con signo mini-
    mo\t"<<CHAR_MIN;
    cout<<endl<<"Entero con signo maxi-
    mo\t\t"<<INT_MAX;
    cout<<endl<<"Entero con signo mini-
    mo\t\t"<<INT_MIN;
    cout<<endl<<"Entero corto con signo maxi-
    mo\t"<<SHRT_MAX;
    cout<<endl<<"Entero corto con signo mini-
    mo\t"<<SHRT_MIN;
    cout<<endl<<"Entero largo con signo maxi-
    mo\t"<<LONG_MAX;
    cout<<endl<<"Entero largo con signo mini-
    mo\t"<<LONG_MIN;
}
```

La salida para *consigno.cpp* es:

```
Numero de Bits del tipo char      8
Caracter con signo maximo         127
Caracter con signo minimo         -128
Entero con signo maximo           32767
Entero con signo minimo           -32768
Entero corto con signo máximo     32767
Entero corto con signo mínimo     -32768
Entero largo con signo máximo     2147483647
Entero largo con signo mínimo     -2147483648
```

Operadores para manejo de bits

C y C++ poseen un conjunto de operadores especiales para el manejo de bits, los cuales sólo se pueden aplicar a operandos integrales tales como: **int**, **char** y **enum**.

Operador	Significado
&	AND
|	OR
^	XOR
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
~	Complemento a uno

Tabla 2: Operadores a nivel de bits en C y C++

El programa *bits1.cpp* imprime un entero sin signo en bits.

```
//imprime un entero sin signo en bits
#include <iostream.h>
#include <stdio.h>

void main()  // bits1.cpp
{
    unsigned n;
    void mostrarBits(unsigned);

    cout<<endl<<"Ingrese un entero sin signo: ";
    cin>>n;
    mostrarBits(n);
}

void mostrarBits(unsigned valor)
{
    unsigned i, mostrarMask = 1 << 15;

    cout<<valor<<"\t";

    for (i = 1; i <= 16; i++) {      put-
char(valor&mostrarMask?'1':'0');
        valor <<= 1;

        if (i % 8 == 0)
          putchar(' ');
    }
    putchar('\n');
}
```

La salida para tres corridas sucesivas de *bits1.cpp* es:

```
Ingrese un entero sin signo: 32767
32767   01111111 11111111

Ingrese un entero sin signo: 1
1       00000000 00000001

Ingrese un entero sin signo: 100
100     00000000 01100100
```

Notación con complemento a dos

La notación con complemento a dos es usada por la mayoría de las computadoras para representar enteros positivos y enteros negativos. Un número *n*-bit, *A*, en notación con complemento a dos es representado como:

$$A \equiv a_{n-1}a_{n-2}....a_1a_0$$

con un valor de

$$A = \left(\sum_{k=0}^{n-2} a_k 2^k\right) - a_{n-1}2^{n-1} \qquad a_k \in \{0,1\}$$

Un número *A*, es negativo sí y solo sí $a_{n-1} = 1$. De la ecuación anterior, el negativo de *A*; *-A*, está dado por:

$$-A = \left(\sum_{k=0}^{n-2} -a_k 2^k\right) + a_{n-1}2^{n-1}$$

El cual puede ser escrito como:

$$-A = 1 + \left(\sum_{k=0}^{n-2} (\overline{a}_k)2^k\right) - \overline{a_{n-1}}2^{n-1}$$

Donde x está definido como el complemento unario.

$$\overline{X} = \begin{Bmatrix} 1, si....X = 0 \\ 0, si....X = 1 \end{Bmatrix}$$

El complemento a uno de un número *A*, denotado por A, está definido como:

$$\overline{A} = \overline{a_{n-1}}\ \overline{a_{n-2}}....\overline{a_0}$$

Resumiendo se tiene que

$$-\overline{A} = 1 + \overline{A}$$

Cuando un número *A* es positivo o negativo el complemento a dos de *A* es equivalente a *-A*.

```
00000001=+1
11111110  ~1 (complemento a uno)
       +1
11111111= -1 (complemento a dos)
```

Nótese que en este caso la representación de -1 se puede obtener también como:

-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1

De modo similar

```
11111111 = -1
00000000   ~(-1) complemento a uno
       +1
00000001 = 1     complemento a dos
```

El rango de números en *n*-bit con notación de complemento a dos está dado por

$$-2^{n-1} \le A \le 2^{n-1} - 1$$

Dicho rango no es simétrico, porque el número cero está representado.

En el Lenguaje de Programación C++ los tipos de datos que utilizan notación con complemento a dos son:

char	(8 bit)
short	(16 bit)
int	(16, 32, o 64 bit)
long	(32 bit)

El número de bits para cada tipo depende del compilador.

El programa *bits4.cpp* muestra el uso de los operadores lógicos &, |, ^ y las notaciones con complemento a uno y complemento a dos.

```
#include <stdio.h>
#include <conio.h>

void mostrarBits(int );

void main()  // bits4.cpp
{
   clrscr();
   int num1, num2, mask;

   num1 = 32767;        num2 = 11100;
   printf("\nEl resultado de combinar lo siguien-
   te:\n");
   mostrarBits(num1);   mostrarBits(num2);
   printf("usando el operador a nivel de bits AND &
   es\n");
   mostrarBits(num1 & num2);

   printf("\nEl resultado de combinar lo siguien-
   te:\n");
   mostrarBits(num1);   mostrarBits(num2);
   printf("usando el operador a nivel de bits OR |
   inclusivo es\n");
   mostrarBits(num1 | num2);

   printf("\nEl resultado de combinar lo siguien-
   te:\n");
   mostrarBits(num1);   mostrarBits(num2);
   printf("usando el operador a nivel de bits OR ^
   exclusivo es\n");
   mostrarBits(num1 ^ num2);

   num1 = 10;
   printf("\nEl complemento a uno de:\n");
   mostrarBits(num1);   printf("es\n");
   mostrarBits(~num1);
   printf("El complemento a dos es\n");
   mostrarBits(~num1+1);
}

void mostrarBits(int valor)
{
   int i, displayMask = 1 << 15;

   printf("%7d = ", valor);
   for (i = 1; i <= 16; i++) {
   putchar(valor&displayMask?'1':'0');
                valor <<= 1;

      if (i % 8 == 0)
                putchar(' ');
   }
   putchar('\n');
}
```

La salida para bits4.cpp es

El resultado de combinar lo siguiente:

32767 = 01111111 11111111
11100 = 00101011 01011100
usando el operador a nivel de bits OR | inclusivo es
32767 = 01111111 11111111

El resultado de combinar lo siguiente:
32767 = 01111111 11111111
11100 = 00101011 01011100
usando el operador a nivel de bits OR ^ exclusivo es
21667 = 01010100 10100011

El complemento a uno de:
10 = 00000000 00001010
es
-11 = 11111111 11110101
El complemento a dos es
-10 = 11111111 11110110

Representación en punto flotante

Los números en punto flotante en las computadoras binarias, se representan en notación científica o exponencial. Un número en punto flotante positivo o negativo tiene un signo en valor fraccionario o mantisa y un exponente tal como sigue:

$$N = SxMxb^e$$

donde *N* es un número fraccionario,

S es positivo o negativo
M es la mantisa $0< M >1$
b es la base entera positiva * 2
e es el exponente entero con signo

también se tiene que

$$M = 0.d_1d_2...d_k,$$

donde M es normalizada si

$$d_1 \neq 0, d_i$$

es entero y

$$0 \leq d_i < b$$

En los lenguajes de alto nivel la representación en punto flotante se utiliza para cálculos que involucren números reales positivos o negativos. El estándar IEEE 754 para punto flotante en bits es el más ampliamente usado. Este estándar especifica que un número en punto flotante se puede representar en 32 bits, 64 bits y en 80 bits.

Estándar IEEE de 32 bits

Este estándar es usado por C y C++ para los reales de precisión simple como son los de tipo **float**. Este formato consiste en 23 bits para la fracción o mantisa M, 8 bits para el exponente y el exceso (cuyo uso permite considerar positivos y negativos) y 1 bit para el signo, siendo 1 positivo 0 negativo. Los resultados se normalizan después de cada operación. Esto significa que la mayoría de los bits significativos de la fracción se fuerzan para ser uno, ajustando el exponente, dado que el bit debe ser uno, este no es almacenado como parte del número. A esto se le llama el bit implícito.

1 bit	8 bit	23bit
S	e + 7Eh	Mantisa normalizada sin primer bit

S = bit con signo (1 positivo y 0 negativo)
e = exponente
7Eh = exceso fijado a 128; su uso es permitir exponentes positivos y negativos.

Siendo la base b =2.

Nota: El número 0.0 se representa con 32 bits 0.

El siguiente programa *muestflo.cpp* permite visualizar cinco números en punto flotante a su respectiva representación en hexadecimal.

```
#include <stdio.h>
#include <conio.h>

void main()      // muestflo.cpp
{
     float f;
     char *p;

     for ( int i=1; i<=5; i++)
     {   printf("Numero? ");
         scanf("%f",&f);
         p=(char *)&f;
         printf("%2x%2x%2x%2x\n",
         p[0],p[1],p[2],p[3]);
     }
     getch();
}
```

La salida de *muestflo.cpp* para 5 números flotantes ingresados es la siguiente

```
Numero? -123.025
ffcd cfff6ffc2
Numero? 123.025
ffcd cfff642
Numero? 50.015
5c f4842
Numero? -50.015
5c f48ffc2
Numero? 100.752
6ff81ffc942
```

Estándar IEEE de 64 bits

Este estándar lo utiliza C y C++ para los reales de precisión doble como son los tipos **double.** Este formato consiste en 52 bit para la fracción o mantisa M, 11 bit para el exponente y el exceso más 1 bit para el signo. Al igual que con los reales de precisión simple los resultados son normalizados después de cada operación. Para él numero 0.0, la norma indica que se representará con 64 bit 0.

El programa *muesdob1.cpp* presenta un número **double** en bits, para ello utiliza una **unión.**

```
#include <iostream.h>

unión bits {
        bits(double n);
      void mostrar_bits();
      double dob;
      unsigned char c[sizeof(double)];
};

bits :: bits(double n)
{
      dob = n;
}

void bits :: mostrar_bits()
{
   int j;
   for(j=sizeof(double)-1; j>=0; j--)
   {
     for (int i = 128; i; i>>=1)
                if (i & c[j])
                        cout<< "1";
                else
                        cout<< "0";
     //separando byte a byte desde byte 7 al byte 0
        cout<<" ";
   }
}

void main()      // muesdob1.cpp
{ double dob;
   for (int i=0; i<3; i++)
   { cout<<"\nIngrese numero double:";
        cin>>dob;
        bits num(dob);
        num.mostrar_bits();
   }
}
```

la salida de *muesdob1.cpp* para tres números es:

```
Ingrese numero double: 123.0125
01000000 01011110 11000000 11001100
11001100 11001100 11001100 11001101

Ingrese numero double: -123.025
11000000 01011110 11000001 10011001
10011001 10011001 10011001 10011010

Ingrese numero double: 0.0
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

Representación de caracteres

Los caracteres (Letras, dígitos, signos de puntuación, etc.) son representados en el sistema de caracteres ASCII, el cual es un conjunto que posee 256 códigos (0,...,255), donde cada código es un 1 byte. Los primeros 128 códigos ASCII son caracteres de control los cuales fueron de uso común en las primeras máquinas. Como puede verse, los caracteres ASCII son ordinales y a cada carácter le corresponde un número.

El siguiente programa muestra los caracteres alfabéticos tanto en mayúsculas como en minúsculas y su correspondiente valor ASCII.

```
#include <iostream.h>
#include <ctype.h>

void main()      // carascii.cpp
{
  char c;

   cout<<"\nCaracteres alfabéticos en Mayúsculas"<<endl;
     for (c='A'; c<='Z'; c++)
        cout<<c<<" = "<<toascii(c)<<"\t";

   cout<<endl;

   cout<<"\nCaracteres alfabeticos en Minusculas"<<endl;
  for (c='a'; c<='z'; c++)
     cout<<c<<" = "<<toascii(c)<<"\t";
}
```

La salida para *carascii.cpp* es la siguiente:

```
Caracteres alfabéticos en Mayúsculas
A = 65  B = 66  C =  D = 68  E = 69  F = 70  G
= 71  H = 72  I = 73  J = 74
K = 75  L = 76  M = 77  N = 78  O = 79  P = 80
Q = 81  R = 82  S = 83  T = 84
U = 85  V = 86  W = 87  X = 88  Y = 89  Z = 90

Caracteres alfabéticos en Minúsculas
a = 97  b = 98  c = 99  d = 100  e = 101  f = 102
g = 103  h = 104  i = 105  j = 106
k = 107  l = 108  m = 109  n = 110  o = 111
p = 112  q = 113  r = 114  s = 115  t = 116
u = 117  v = 118  w = 119  x = 120  y = 121
z = 122
```

En cada base numérica se tiene dígitos significativos *bm*; donde *b* es la base y *m* es el conjunto de dígitos válidos en la base *b*. Por ejemplo en base 2 se tiene que *m* = {0,1}; en base 8(octal), *m*={0,1,2,3,4,5,6,7}; en base 10(decimal); se tiene *m*={0,1,2,3,4,5,6,7,8,9}; finalmente en base 16 (hexadecimal) se tiene:
m={0, 1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}.

El siguiente programa *esdighex.cpp* averigua si un carácter ingresado por el usuario es o no un dígito hexadecimal.

```
#include <iostream.h>

void main()      // esdighex.cpp
{
  char c;
  cout<<endl<<"Ingrese carácter: ";
  cin>>c;

   if((c>='0'&& c<='9')||(c>='a'&& c<='f')||(c>='A' &&
   c<='Z'))
        cout<<c<<
        " es digito hexadecimal";
  else
        cout<<c<<" no es digito
   hexadecimal";
}
```

La salida para cuatro corridas sucesivas de *esdighex.cpp* es:

```
Ingrese carácter: a
a es dígito hexadecimal
Ingrese carácter: 4
4 es dígito hexadecimal
Ingrese carácter: m
m no es dígito hexadecimal
Ingrese carácter: f
f es dígito hexadecimal
```

Bibliografía

1. Deitel H.M. y Deitel P.J.,"*Como Programar en C/C++* ".México, 2da. ed. Prentice Hall: 927p. (1995).

2. Parker Alan, "*Algorithms and Data Structures in C++* ". USA CRC Press:257p.(1993).

3. Ruiz Lizama Edgar, "*Curso de Lenguage C*". Lima, Facultad de Ingeniería Industrial UNMSM: 234p.(1999).