NAME

bin_dec_hex – How to use binary, decimal, and hexadecimal notation.

DESCRIPTION

Most people use the decimal numbering system. This system uses ten symbols to represent numbers. When those ten symbols are used up, they start all over again and increment the position to the left. The digit 0 is only shown if it is the only symbol in the sequence, or if it is not the first one.

If this sounds cryptic to you, this is what I've just said in numbers:

```
     0
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
    11
    12
    13
```

and so on.

Each time the digit nine is incremented, it is reset to 0 and the position before (to the left) is incremented (from 0 to 1). Then number 9 can be seen as "00009" and when we should increment 9, we reset it to zero and increment the digit just before the 9 so the number becomes "00010". Leading zeros we don't write except if it is the only digit (number 0). And of course, we write zeros if they occur anywhere inside or at the end of a number:

```
 "00010" -> " 0010" -> " 010" -> "  10", but not "  1 ".
```

This was pretty basic, you already knew this. Why did I tell it? Well, computers usually do not represent numbers with 10 different digits. They only use two different symbols, namely "0" and "1". Apply the same rules to this set of digits and you get the binary numbering system:

```
     0
     1
    10
    11
   100
   101
   110
   111
  1000
  1001
  1010
  1011
  1100
  1101
```

and so on.

If you count the number of rows, you'll see that these are again 14 different numbers. The numbers are the same and mean the same as in the first list, we just used a different representation. This means that you have to know the representation used, or as it is called the numbering system or base. Normally, if we do not explicitly specify the numbering system used, we implicitly use the decimal system. If we want to use any other numbering system, we'll have to make that clear. There are a few widely adopted methods to do so. One common form is to write 1010(2) which means that you wrote down a number in its binary representation. It is the number ten. If you would write 1010 without specifying the base, the number is interpreted as one thousand and ten using base 10.

In books, another form is common. It uses subscripts (little characters, more or less in between two rows). You can leave out the parentheses in that case and write down the number in normal characters followed by a little two just behind it.

As the numbering system used is also called the base, we talk of the number 1100 base 2, the number 12 base 10.

Within the binary system, it is common to write leading zeros. The numbers are written down in series of four, eight or sixteen depending on the context.

We can use the binary form when talking to computers (...programming...), but the numbers will have large representations. The number 65'535 (often in the decimal system a ' is used to separate blocks of three digits for readability) would be written down as 1111111111111111(2) which is 16 times the digit 1. This is difficult and prone to errors. Therefore, we usually would use another base, called hexadecimal. It uses 16 different symbols. First the symbols from the decimal system are used, thereafter we continue with alphabetic characters. We get 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. This system is chosen because the hexadecimal form can be converted into the binary system very easily (and back).

There is yet another system in use, called the octal system. This was more common in the old days, but is not used very often anymore. As you might find it in use sometimes, you should get used to it and we'll show it below. It's the same story as with the other representations, but with eight different symbols.

Binary	(2)
Octal	(8)
Decimal	(10)
Hexadecimal	(16)

(2)	(8)	(10)	(16)
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	10	8	8
01001	11	9	9
01010	12	10	A
01011	13	11	B
01100	14	12	C
01101	15	13	D
01110	16	14	E
01111	17	15	F
10000	20	16	10
10001	21	17	11
10010	22	18	12
10011	23	19	13
10100	24	20	14
10101	25	21	15

Most computers used nowadays are using bytes of eight bits. This means that they store eight bits at a time. You can see why the octal system is not the most practical for that: You'd need three digits to represent the eight bits and this means that you'd have to use one complete digit to represent only two bits (2+3+3=8).

This is a waste. For hexadecimal digits, you need only two digits which are used completely:

```
 (2)      (8)  (10) (16)
11111111 377  255   FF
```

You can see why binary and hexadecimal can be converted quickly: For each hexadecimal digit there are exactly four binary digits. Take a binary number: take four digits from the right and make a hexadecimal digit from it (see the table above). Repeat this until there are no more digits. And the other way around: Take a hexadecimal number. For each digit, write down its binary equivalent.

Computers (or rather the parsers running on them) would have a hard time converting a number like 1234(16). Therefore hexadecimal numbers are specified with a prefix. This prefix depends on the language you're writing in. Some of the prefixes are "0x" for C, "$" for Pascal, "#" for HTML. It is common to assume that if a number starts with a zero, it is octal. It does not matter what is used as long as you know what it is. I will use "0x" for hexadecimal, "%" for binary and "0" for octal. The following numbers are all the same, just their representation (base) is different: 021 0x11 17 `%00010001`

To do arithmetics and conversions you need to understand one more thing. It is something you already know but perhaps you do not "see" it yet:

If you write down 1234, (no prefix, so it is decimal) you are talking about the number one thousand, two hundred and thirty four. In sort of a formula:

```
1 * 1000 = 1000
2 *  100 =  200
3 *   10 =   30
4 *    1 =    4
```

This can also be written as:

```
1 * 10^3
2 * 10^2
3 * 10^1
4 * 10^0
```

where ^ means "to the power of".

We are using the base 10, and the positions 0,1,2 and 3. The right-most position should NOT be multiplied with 10. The second from the right should be multiplied one time with 10. The third from the right is multiplied with 10 two times. This continues for whatever positions are used.

It is the same in all other representations:

0x1234 will be

```
1 * 16^3
2 * 16^2
3 * 16^1
4 * 16^0
```

01234 would be

```
1 * 8^3
2 * 8^2
3 * 8^1
4 * 8^0
```

This example can not be done for binary as that system only uses two symbols. Another example:

`%1010` would be

```
1 * 2^3
0 * 2^2
1 * 2^1
0 * 2^0
```

It would have been easier to convert it to its hexadecimal form and just translate `%1010` into 0xA. After a while you get used to it. You will not need to do any calculations anymore, but just know that 0xA means 10.

To convert a decimal number into a hexadecimal you could use the next method. It will take some time to be able to do the estimates, but it will be easier when you use the system more frequently. We'll look at yet another way afterwards.

First you need to know how many positions will be used in the other system. To do so, you need to know the maximum numbers you'll be using. Well, that's not as hard as it looks. In decimal, the maximum number that you can form with two digits is "99". The maximum for three: "999". The next number would need an extra position. Reverse this idea and you will see that the number can be found by taking 10^3 (10*10*10 is 1000) minus 1 or 10^2 minus one.

This can be done for hexadecimal as well:

```
16^4 = 0x10000 = 65536
16^3 =  0x1000 =  4096
16^2 =   0x100 =   256
16^1 =    0x10 =    16
```

If a number is smaller than 65'536 it will fit in four positions. If the number is bigger than 4'095, you must use position 4. How many times you can subtract 4'096 from the number without going below zero is the first digit you write down. This will always be a number from 1 to 15 (0x1 to 0xF). Do the same for the other positions.

Let's try with 41'029. It is smaller than 16^4 but bigger than 16^3−1. This means that we have to use four positions. We can subtract 16^3 from 41'029 ten times without going below zero. The left-most digit will therefore be "A", so we have 0xA????. The number is reduced to 41'029 − 10*4'096 = 41'029−40'960 = 69. 69 is smaller than 16^3 but not bigger than 16^2−1. The second digit is therefore "0" and we now have 0xA0??. 69 is smaller than 16^2 and bigger than 16^1−1. We can subtract 16^1 (which is just plain 16) four times and write down "4" to get 0xA04?. Subtract 64 from 69 (69 − 4*16) and the last digit is 5 −−> 0xA045.

The other method builds up the number from the right. Let's try 41'029 again. Divide by 16 and do not use fractions (only whole numbers).

```
41'029 / 16 is 2'564 with a remainder of 5. Write down 5.
2'564 / 16 is 160 with a remainder of 4. Write the 4 before the 5.
160 / 16 is 10 with no remainder. Prepend 45 with 0.
10 / 16 is below one. End here and prepend 0xA. End up with 0xA045.
```

Which method to use is up to you. Use whatever works for you. I use them both without being able to tell what method I use in each case, it just depends on the number, I think. Fact is, some numbers will occur frequently while programming. If the number is close to one I am familiar with, then I will use the first method (like 32'770 which is into 32'768 + 2 and I just know that it is 0x8000 + 0x2 = 0x8002).

For binary the same approach can be used. The base is 2 and not 16, and the number of positions will grow rapidly. Using the second method has the advantage that you can see very easily if you should write down a zero or a one: if you divide by two the remainder will be zero if it is an even number and one if it is an odd number:

```
 41029 / 2 = 20514 remainder 1
 20514 / 2 = 10257 remainder 0
 10257 / 2 =  5128 remainder 1
  5128 / 2 =  2564 remainder 0
  2564 / 2 =  1282 remainder 0
  1282 / 2 =   641 remainder 0
   641 / 2 =   320 remainder 1
   320 / 2 =   160 remainder 0
   160 / 2 =    80 remainder 0
    80 / 2 =    40 remainder 0
    40 / 2 =    20 remainder 0
    20 / 2 =    10 remainder 0
    10 / 2 =     5 remainder 0
     5 / 2 =     2 remainder 1
     2 / 2 =     1 remainder 0
     1 / 2 below 0 remainder 1
```

Write down the results from right to left: `%1010000001000101`

Group by four:

```
 %1010000001000101
 %101000000100 0101
 %10100000 0100 0101
 %1010 0000 0100 0101
```

Convert into hexadecimal: 0xA045

Group `%1010000001000101` by three and convert into octal:

```
 %1010000001000101
 %1010000001000 101
 %1010000001 000 101
 %1010000 001 000 101
 %1010 000 001 000 101
 %1 010 000 001 000 101
 %001 010 000 001 000 101
    1   2   0   1   0   5 --> 0120105

 So: %1010000001000101 = 0120105 = 0xA045 = 41029
 Or: 1010000001000101(2) = 120105(8) = A045(16) = 41029(10)
 Or: 1010000001000101(2) = 120105(8) = A045(16) = 41029
```

At first while adding numbers, you'll convert them to their decimal form and then back into their original form after doing the addition. If you use the other numbering system often, you will see that you'll be able to do arithmetics directly in the base that is used. In any representation it is the same, add the numbers on the right, write down the right-most digit from the result, remember the other digits and use them in the next round. Continue with the second digit from the right and so on:

```
   %1010 + %0111 --> 10 + 7 --> 17 --> %00010001
```

will become

```
        %1010
        %0111 +
         ||||
         |||+-- add 0 + 1, result is 1, nothing to remember
         ||+--- add 1 + 1, result is %10, write down 0 and remember 1
         |+---- add 0 + 1 + 1(remembered), result = 0, remember 1
         +----- add 1 + 0 + 1(remembered), result = 0, remember 1
                nothing to add, 1 remembered, result = 1
     --------
       %10001 is the result, I like to write it as %00010001
```

For low values, try to do the calculations yourself, then check them with a calculator. The more you do the calculations yourself, the more you'll find that you didn't make mistakes. In the end, you'll do calculi in other bases as easily as you do them in decimal.

When the numbers get bigger, you'll have to realize that a computer is not called a computer just to have a nice name. There are many different calculators available, use them. For Unix you could use ''bc'' which is short for Binary Calculator. It calculates not only in decimal, but in all bases you'll ever want to use (among them Binary).

For people on Windows: Start the calculator (start−>programs−>accessories−>calculator) and if necessary click view−>scientific. You now have a scientific calculator and can compute in binary or hexadecimal.

AUTHOR

I hope you enjoyed the examples and their descriptions. If you do, help other people by pointing them to this document when they are asking basic questions. They will not only get their answer, but at the same time learn a whole lot more.

Alex van den Bogaerdt <alex@vandenbogaerdt.nl>