

Sistemas Numéricos

Sistema Decimal (base 10)

Descrevemos quantidades com números na base decimal usando dez símbolos diferentes:

0, 1, 2, 3, 4, 5, 6, 7, 8 e 9.

A regra para associar os números às respectivas quantidades é simples:

Comece com um número de apenas um dígito (o símbolo “0”) para representar o zero. Utilize os demais dígitos para os próximos nove números. Uma vez que o dígito na posição corrente atinge 9, procuramos para a esquerda dessa posição o primeiro dígito que não é 9. O dígito dessa nova posição avança para o próximo símbolo e todos os dígitos à direita dessa posição retornam para o 0. A nova posição corrente é a primeira posição à direita, e repetimos o processo.

Por exemplo:

00, 01, 02, ... 07, 08, 09 (o dígito à direita reinicia, e o à esquerda passa de 0 para 1)

10, 11, 12, ... 17, 18, 19 (o dígito à direita reinicia, e o à esquerda passa de 1 para 2)

20, 21, 22, ...

Por conveniência, omitimos os dígitos “0” à esquerda do número.

Outros Sistemas Numéricos

O sistema decimal não é a única maneira para se expressar quantidades. Note que a escolha dos símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9 foi totalmente arbitrária. Eles poderiam ser substituídos por quaisquer outros, sem comprometer a contagem. Por exemplo, os romanos preferiram utilizar “I”, “II”, “III”, “IV”, “V”, “VI”, “VII”, “VIII”, “IX”, etc. Existem outras formas para contar quantidades. Uma alternativa comum é a dúzia, que contém 12 unidades. A grossa é o resultado quando completamos 12 dúzias (e reiniciamos a contagem de dúzias). Outra forma comum é o sistema com 60 unidades. Por exemplo, ao expressar minutos e segundos. A forma mais simples é o sistema unário. Escolhe-se um símbolo (por exemplo, “x”) e o repetimos tantas vezes quanto for o valor do número. Para representar o valor 7, escreveríamos xxxxxxxx. Por ser muito primitivo, este sistema é útil somente para números pequenos. Diante de tantas formas de representar números, não é de estranhar que não exista uma única opção universal.

Neste capítulo, estudaremos os sistemas utilizados na linguagem C.

Sistema Binário (base 2)

Possui apenas dois símbolos (0 e 1) para compor números. A regra continua a mesma que aquela já enunciada para os números decimais. Começamos com o símbolo 0 e para o próximo número, utilizamos o 1. O dígito à esquerda avança para o próximo símbolo (0 ou 1) e a contagem à direita retorna para o símbolo 0.

000, 001

010, 011

100, 101

110, 111...

Sistema Octal (base 8)

Ele representa números com oito símbolos (0, 1, 2, 3, 4, 5, 6 e 7). A contagem será:

00, 01, 02, 03, 04, 05, 06, 07
10, 11, 12, 13, 14, 15, 16, 17,
20, 21, ...

Sistema Hexadecimal (base 16)

Ele representa números com dezesseis símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F). Como não possuímos tantos dígitos numéricos, convencionou-se representar o símbolo depois do nove como “A”, depois “B”, e assim por diante.

A contagem será:

00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F,
20, 21, 22, ...

Representações

Tome o número “101”. Qual é a quantidade que ele representa? Isto depende do sistema numérico em uso. Sem essa informação adicional, “101” é um dado que pode ser interpretado de diferentes maneiras, dependendo do sistema numérico.

Por este motivo, para evitar ambigüidade, adotaremos a seguinte convenção: números no sistema decimal serão representados na forma convencional. Nos demais casos, as bases serão indicadas como um subscrito no lado direito do número. Por exemplo: 101, 101₂, 101₈ ou 101₁₆.

Dadas uma representação e a base do sistema numérico, é fácil determinar a quantidade representada:

- Decimal (base 10): $101_{10} = 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = 101_{10}$
- Binário (base 2): $101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{10}$
- Octal (base 8): $101_8 = 1 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 65_{10}$
- Hexadecimal (base 16): $101_{16} = 1 \cdot 16^2 + 0 \cdot 16^1 + 1 \cdot 16^0 = 257_{10}$

Em geral, o valor de um número na base b , representado pelos dígitos

$$a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0$$

será dado por:

$$[a_n a_{n-1} a_{n-2} a_2 a_1 a_0]_b = \sum_{i=0}^n a_i \cdot b^i$$

Comparação entre sistemas numéricos

Decimal	Binário	Octal	Hexadecimal
0	0_2	0_8	0_{16}
1	1_2	1_8	1_{16}
2	10_2	2_8	2_{16}
3	11_2	3_8	3_{16}
4	100_2	4_8	4_{16}
5	101_2	5_8	5_{16}
6	110_2	6_8	6_{16}
7	111_2	7_8	7_{16}
8	1000_2	10_8	8_{16}
9	1001_2	11_8	9_{16}
10	1010_2	12_8	A_{16}
11	1011_2	13_8	B_{16}
12	1100_2	14_8	C_{16}
13	1101_2	15_8	D_{16}
14	1110_2	16_8	E_{16}
15	1111_2	17_8	F_{16}
16	10000_2	20_8	10_{16}
17	10001_2	21_8	11_{16}
18	10010_2	22_8	12_{16}
19	10011_2	23_8	13_{16}

Tabela 1 – Comparativo entre sistemas numéricos

Conversões de base

Conversão de base B para base decimal

Algoritmo de multiplicações sucessivas

Um número N , representado na base B por $A_n A_{n-1} A_{n-2} \dots A_2 A_1 A_0$ pode ser reescrito na base 10 utilizando o algoritmo de multiplicações sucessivas.

No passo inicial, o resultado será zero. Para cada dígito A_i (começando com A_n e terminando em A_0), multiplica-se o resultado pela base B e, em seguida, soma-se o valor do dígito A_i .

Exemplo

Transformar 1305_8 (octal) para a base decimal:

Etapas:	Valor:	Dígito	Novo Valor
0	$0 \cdot 8$	+ 1	= 1
1	$1 \cdot 8$	+ 3	= 11
2	$11 \cdot 8$	+ 0	= 88
3	$88 \cdot 8$	+ 5	= 709
4	709	FIM	

Portanto, $1305_8 = 709_{10}$

Exemplo

Transformar $1E31_{16}$ (hexadecimal) para a base decimal:

Observação: no sistema hexadecimal, o símbolo A corresponde ao décimo dígito, B ao décimo primeiro, e assim por diante. Logo, o símbolo E corresponderá ao décimo quarto dígito.

Etapas:	Valor:	Dígito	Novo Valor
0	$0 \cdot 16$	+ 1	= 1
1	$1 \cdot 16$	+ E (14)	= 30 ($16 + 14 = 30$)
2	$30 \cdot 16$	+ 3	= 483
3	$483 \cdot 16$	+ 1	= 7729
4	7729	FIM	

Portanto, $1E31_{16} = 7729_{10}$

Exemplo

Transformar 00101011_2 (binário) para a base decimal:

Observação: podemos ignorar os dígitos não significativos à esquerda.

Etapas:	Valor:	Dígito	Novo Valor
0	$0 \cdot 2$	+ 1	= 1
1	$1 \cdot 2$	+ 0	= 2
2	$2 \cdot 2$	+ 1	= 5
3	$5 \cdot 2$	+ 0	= 10
4	$10 \cdot 2$	+ 1	= 21
5	$21 \cdot 2$	+ 1	= 43
6	43	FIM	

Portanto, $00101011_2 = 43_{10}$

Conversão de base decimal para base b

Algoritmo de divisões sucessivas

Um número N , representado na base decimal, após ser convertido para a base b seria representado como $a_m a_{m-1} \dots a_2 a_1 a_0$. Para se obter os correspondentes símbolos na base b podemos usar o o algoritmo de divisões sucessivas.

Enquanto o número for não nulo, dividimo-lo pela base b . O resto da divisão será o valor do próximo dígito a_i (de a_0 até a_m).

Exemplo

Transformar 2034 para a base octal:

Etapas:	Valor:	Quociente	Resto
0	2034	$\div 8 = 254$	+ 2
1	254	$\div 8 = 31$	+ 6
2	31	$\div 8 = 3$	+ 7
3	3	$\div 8 = 0$	+ 3

Portanto, $2034_{10} = 3762_8$. Note que devemos ler a coluna “Resto” de baixo para cima.

Exemplo

Transformar 1253 para a base hexadecimal:

Etapas:	Valor:	Quociente	Resto
0	1253	$\div 16 = 78$	+ 5
1	78	$\div 16 = 4$	+ 14 (E)
2	4	$\div 16 = 0$	+ 4

Portanto, $1253_{10} = 4E5_{16}$

Exemplo

Transformar 35 para a base binária:

Etapas:	Valor:	Quociente	Resto
0	35	$\div 2 = 17$	+ 1
1	17	$\div 2 = 8$	+ 1
2	8	$\div 2 = 4$	+ 0
2	4	$\div 2 = 2$	+ 0
2	2	$\div 2 = 1$	+ 0
2	1	$\div 2 = 0$	+ 1

Portanto, $35_{10} = 100011_2$

De base B para base b

A forma mais simples para se reescrever um número de uma base para a outra, é primeiro convertê-lo para a base decimal (com o algoritmo de multiplicações sucessivas) e, em seguida, converter a representação obtida para a base desejada (com o algoritmo de divisões sucessivas).

A fórmula a seguir representa o resultado das duas operações:

$$[A_n A_{n-1} A_{n-2} \dots A_2 A_1 A_0]_B = \sum_{i=0}^n A_i \cdot B^i = N = \sum_{j=0}^m a_j \cdot b^j = [a_m a_{m-1} a_{m-2} \dots a_2 a_1 a_0]_b$$

Conversão entre binário, octal e hexadecimal

A conversão entre binário, octal e hexadecimal é particularmente simples.

Primeiro, consultando a tabela comparativa entre sistemas numéricos, nota-se que um trio de dígitos binários corresponde exatamente a um dígito octal. Uma quádrupla de dígitos binários forma um dígito hexadecimal.

Exemplo

Converter 00110110_2 para hexadecimal resulta em 36_{16} . Para tal, consultamos a tabela para transformar cada quatro dígitos binários em um dígito hexadecimal.

0	0	1	1	0	1	1	0
3				6			

Converter 00110110_2 para octal resulta em 66_8 . Para tal, consultamos a tabela para transformar cada três dígitos binários em um dígito octal.

0	0	1	1	0	1	1	0
0			6			6	

Sistemas numéricos em C

Já aprendemos que na linguagem C as constantes numéricas inteiras são formadas por uma sequência de dígitos numéricos sem ponto decimal. Elas são tratadas como números inteiros. O prefixo da constante indica a base do sistema de contagem.

As constantes em C podem ser na base:

- Decimal: sem prefixo, e o primeiro dígito não pode ser 0 (zero).
- Octal: começam com o prefixo “0”.
- Hexadecimal: começam com o prefixo “0x”. Neste caso, além dos dígitos 0, 1, 2, ..., 9, pode-se utilizar também os símbolos A, B, C, D, E e F para compor o número.
- Binária: não existe representação binária em C.

Exemplos:

`10`, `132`, `32179`: O número, sem ponto decimal, que começa com 1, 2, 3, ..., 9 é tratado como número inteiro na base 10 (decimal).

`012`, `0204`, `076663`: O número, sem ponto decimal, que começa com 0 é tratado como número inteiro na base 8 (octal).

`0xA`, ou `0xa`, `0x84`, `0x7dB3`, ou `0x7DB3`: O número, sem ponto decimal, que começa com 0x é tratado como número inteiro na base 16 (hexadecimal). Note que não faz diferença utilizar maiúsculas ou minúsculas.

Veja o seguinte trecho de código:

```
i=20;
printf("i(dec)=%d\n",i);
// constante base 8
i=020;
printf("i(dec)=%d\n",i);
```

```
//constante base 16
i=0x20;
printf("i(dec)=%d\n", i);
```

A saída seria:

```
i(dec)=20
i(dec)=16
i(dec)=32
```

A primeira atribuição armazena 20 em `i`, o que é confirmado pela primeira linha na saída.

A segunda atribuição armazena interpreta 20 na base 8, devido ao dígito 0 no início, e armazena o resultado em `i`. Confirmando: $2 \times 8 + 0 \times 8 = 16$.

A terceira atribuição armazena em `i` o valor correspondente a 20 na base 16, como sugerido pelo prefixo `0x`. O resultado seria: $2 \times 16 + 0 \times 16 = 32$.

Escrevendo na saída, nas três bases básicas.

Já sabemos que o modificador `%d`, quando usado na rotina `printf`, coloca na saída o valor da variável correspondente, e escrito na *base 10*. Em C temos também os modificadores `%o` para escrever um valor inteiro na *base 8*, e `%x` para escrever um valor na base 16. A função do modificador `%X` é idêntica a do modificador `%x`, exceto que agora os símbolos hexadecimais são impressos em letras maiúsculas.

Outro ponto importante é que valores escritos nas bases 8 e 16 são sempre entendidos como positivos, i.e., *o bit mais significativo não é interpretado como um bit de sinal*. Nestes casos, todos os bits são interpretados como parte de um valor positivo, como se este fosse do tipo `unsigned`.

O trecho de código a seguir, armazena uma sequência de bits em uma variável e imprime esse mesmo valor usando as três interpretações padrão. Estamos supondo que o computador usa 32 bits para armazenar valores tipo `int`. Além disso, também temos uma função que imprime a sequência de bits correspondente ao valor armazenado em uma variável, usando 32 bits. Note que não temos uma facilidade para imprimir diretamente a sequência de bits correspondente a um valor armazenado em uma variável.

```
int i=22;
int j=0xffffffff; // como se fosse -2
/* mesma sequencia de bits impressa de varias formas */
printf("Sequencia de bits de %d eh: ", i);
ImpSeqBits(i);
printf("Valores em decimal, octal e hexadecimal: \n");
printf("i(dec)=%d, i(oct)=%o, i(hex)=%x\n", i, i, i);
printf("-----\n");
printf("Sequencia de bits de %d eh: ", j);
ImpSeqBits(j);
printf("Valores em decimal, octal e hexadecimal: \n");
printf("j(dec)=%d, j(oct)=%o, j(hex)=%x\n", j, j, j);
```

As primeiras duas linhas definem `i` e `j` como 22 e -2, respectivamente. No caso, `0xffffffff` é a representação de -2 se usarmos 32 bits.

Em seguida, a função `ImpSeqBits` imprime a sequência de bits correspondente ao valor armazenado em `i`. A saída seria

Sequencia de bits de 22 eh: 00000000 00000000 00000000 00010110

Passando esse valor para a base 10 teríamos: $1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 22$.

A próxima linha imprime o valor armazenado em `i` interpretando-o como um número decimal, um número octal e um número hexadecimal:

Valores em decimal, octal e hexadecimal:

`i(dec)=22, i(oct)=26, i(hex)=16`

Note que $22 = 2 \times 8 + 6$, portanto 22 na base 8 corresponde a 26. E também $22 = 1 \times 16 + 6$, e daí 22 na base 16 é 16.

A próxima linha confirma que o valor armazenado em `j` é mesmo aquele que foi atribuído:

Sequencia de bits de -2 eh: 11111111 11111111 11111111 11111110

Note que, interpretada como um número decimal de 32 bits, essa sequência de bits representa -2, ou seja, o bit de sinal é respeitado. Escrevendo esses valores em decimal, octal e hexadecimal, resulta em:

Valores em decimal, octal e hexadecimal:

`j(dec)=-2, j(oct)=37777777776, j(hex)=fffffffe`

Usando a sequência binária exposta acima, fica facil verificar que os valores em octal e em hexadecimal estão corretos. Note que a representação octal e hexadecimal não tratam o primieiro bit como um bit de sinal. Elas simplesmente reportam diretamente quais bits estão armazenados na variável.

A função `ImpSeqBits` é fácil de ser programada:

```
void ImpSeqBits(int v) {
    int j;
    unsigned comp=0x80000000; // 32 bits com o primeiro sendo 1
    unsigned val;
    val=(unsigned) v;
    for(j=0; j<32; j++) { // 32 bits
        if ( (j==8)||(j==16)||(j==24) ) {putchar(' ');} //separa bytes
        if (val >= comp) { putchar('1'); } // 1 na posicao mais significativa
        else { putchar('0'); } // 0 na posicao mais significativa
        val=2*val; // equivale a deslocar uma casa para esquerda
    }
    putchar('\n');
}
```

Primeiro, o valor de `v` é colocado em `val`, como `unsigned`.

Em seguida, cada iteração do `for` imprime um dos bits de `val`.

O primeiro teste apenas coloca na saída um espaço em branco, para separar cada 8 bits da sequência de 32 bits de saída, facilitando sua leitura.

Se o segundo teste for verdadeiro, então o bit mais significativo de `val` é 1; caso contrário é 0.

A atribuição `val=2*val` desloca os bits e `val` uma casa para a esquerda. Note que estamos trabalhando na base 2 e, nessa base, multiplicar por 2 equivale a um deslocamento de uma casa para a esquerda. Desta forma, na próxima iteração, estaremos testando o segundo bit mais significativo de `val`, e assim por diante.

A última linha termina a impressão dos 32 bits de `v` e passa o cursor para a próxima linha.

Valores na entrada nas três bases básicas.

Também podemos ler valores nas três bases básicas: decimal, octal e hexadecimal. Os modificadores são os mesmos empregados para a impressão de valores nessas mesmas bases, ou seja, `%o` lê uma constante na base 8, e `%x` ou `%X` lê uma constante na base 16.

Veja o código abaixo:

```
int i;
// constante base 10
printf("Entre com numero na base 10: ");
scanf("%d",&i);
printf("Valor de i (em decimal)=%d\n",i);
// constante base 8
printf("Entre com numero na base 8: ");
scanf("%o",&i);
printf("Valor de i (em decimal)=%d\n",i);
//constante base 16
printf("Entre com numero na base 16: ");
scanf("%x",&i);
printf("Valor de i (em decimal)=%d\n",i);
```

Cada uma das partes lê um valor da entrada em uma das três bases. Note o uso dos modificadores apropriados.

Ao ler um valor na base 8, o computador captura os dígitos na base 8 sucessivamente, até encontrar um dígito que não seja um dos dígitos da base 8. Assim, se na segunda leitura entrarmos com 78, por exemplo, o modificador `%o` vai forçar a leitura apenas o dígito 7. Em seguida, o computador imprimirá o valor 7 e a mensagem para entrar como um número na base 16. O dígito 8, entretanto, ainda se encontra na entrada pois não foi lido pela invocação anterior na forma `scanf("%o")`. Logo, o próximo comando `scanf("%x")` vai ler o dígito 8, que é um dígito válido na base 16, armazenando esse valor na variável `i`. Em seguida, imprimirá o valor 8 na saída.

Representação de números no computador

Bits, Nibbles, Bytes e Palavras

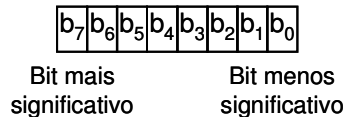
Os computadores atuais operam somente com dois níveis de voltagem (normalmente 0V e 5V). Por este motivo, uma unidade de informação é representada por uma das duas voltagens (baixa ou alta). O significado atribuído a cada uma das voltagens depende do computador, e normalmente será 0 ou 1, falso ou verdadeiro, ou qualquer opção com duas possibilidades.

Quando tratamos de números, essa unidade de informação é denominada **bit** (*binary digit*) e representa o valor 0 ou o valor 1. Uma sequência de 8 bits forma um **byte**. Cada meia parte de um byte (4 bits) forma um **nibble**. Os computadores modernos trabalham com **palavras** (*words*), que podem ser seqüências de 2, 4 ou 8 bytes (16, 32 ou 64 bits).

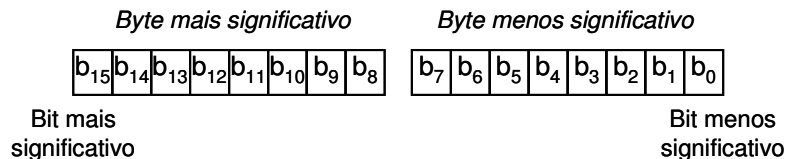
Representação binária no computador

O computador é uma máquina que manipulação de bits. Qualquer informação (principalmente números) é processada, armazenada, recebida ou enviada como uma sequência de bits. Por este motivo, um número precisa ser convertido para o sistema binário antes de ser manipulado pelo computador. Nesta representação, cada dígito será representado pelo estado de um bit.

Em um byte, os bits são numerados da direita para a esquerda, começando com 0 (**bit menos significativo**) até 7 (**bit mais significativo**). É muito comum encontrar as siglas LSB (*least significant bit*) e MSB (*most significant bit*).



Um byte é capaz de representar números binários de 00000000_2 até 11111111_2 . Transformando estes limites para base decimal, obtemos 0 e 255. Os bits $b_7b_6b_5b_4b_3b_2b_1b_0$ correspondem respectivamente aos dígitos $a_7a_6a_5a_4a_3a_2a_1a_0$ de um número em base binária. É claro que com dois bytes (ou seja, uma palavra), será possível expressar um número binário ainda maior. A numeração dos bits continua da mesma forma: direita para a esquerda, de 0 (bit menos significativo) até 15 (bit mais significativo).



O byte com o bit mais significativo é chamado de **byte mais significativo**, o outro byte é chamado de **byte menos significativo**.

Nessa forma, podemos representar valores de $00000000\ 00000000_2$ até $11111111\ 11111111_2$, ou seja, de 0 até 65535.

Resumindo, uma sequência de n bits pode representar valores variando de 0 até $2^n - 1$.

Tipos de Dados

Uma variável em C está associada com uma posição de memória. O tipo da variável determina a quantidade de bits reservada para armazenar o valor da variável. Conseqüentemente, o tipo de dados especifica quantos bits estarão disponíveis para armazenar os dígitos da representação binária de um número.

No Visual Studio .NET, compilando em uma plataforma Intel Pentium, os tipos inteiros reservam memória conforme a tabela:

Tipo	Quantidade de memória
<i>Char</i> <i>signed char</i> <i>unsigned char</i>	1 byte (8 bits)
<i>(signed) short int</i> <i>unsigned short int</i>	2 bytes (16 bits)
<i>(signed) int</i> <i>unsigned int</i>	4 bytes (32 bits)
<i>(signed) long int</i> <i>unsigned long int</i>	4 bytes (32 bits)
<i>(signed) long long int</i> <i>unsigned long long int</i>	8 bytes (64 bits)

Isto mostra por que uma variável com tipo *int* é capaz de armazenar números maiores que uma variável *short int*: simplesmente pelo fato de possuir o dobro do número de bits e, portanto, sendo capaz de armazenar o dobro de dígitos na representação binária.

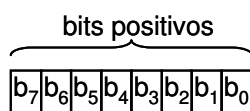
signed* vs. *unsigned int

Em C, para cada tipo inteiro, existe a opção *signed* e *unsigned* (consulte o capítulo “Outros Tipos Inteiros”). Com o conhecimento da representação binária de números, é possível explicar a diferença entre estas duas opções.

unsigned

Uma variável inteira de tipo *unsigned* utiliza uma palavra de n bits, onde n depende do tipo da variável: *char*, *short int*, *long int* ou *long long int* (respectivamente, no Visual Studio .NET, 8, 16, 32 ou 64 bits).

A estrutura de um *unsigned char* está ilustrada abaixo:



O valor deste *unsigned char* será:

$$b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Todos os bits $b_{n-1}, b_{n-2}, \dots, b_1, b_0$ são usados para armazenar os dígitos $a_{n-1}a_{n-2}\dots a_1a_0$ da representação binária do número. Não existe um mecanismo para representar o sinal do número. Portanto o tipo *unsigned* será sempre um número inteiro positivo ou zero.

O menor valor possível se dará quando todos os n bits são 0, formando o número 0. O maior valor possível será obtido quando todos os n bits são 1, gerando o número $2^n - 1$ (verifique isso com o algoritmo de multiplicações sucessivas).

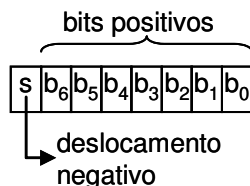
Para o tipo *unsigned char*, que usa 8 bits, isto significa que o seu menor valor possível é 0 e o maior é $2^8 - 1$, ou seja, 255.

signed

Já as variáveis inteiras de tipo *signed* empregam uma estratégia diferente. Em uma palavra de n bits ($b_{n-1}b_{n-2}...b_1b_0$), somente os bits $b_{n-2}...b_1b_0$ armazenam os dígitos $a_{n-2}...a_1a_0$ da representação binária do número.

A solução mais simples seria o bit b_{n-1} indicar o sinal do número (por exemplo, 0 para positivo e 1 para negativo). Esse bit também é conhecido como o *bit de sinal*, *s*. Numa das possíveis convenções para números negativos, usada nos computadores atuais, o bit b_{n-1} acrescenta um deslocamento negativo de 2^{n-1} unidades ao valor formado pelos bits $b_{n-2}...b_0$.

A estrutura de um *signed char* está ilustrada abaixo:



O valor deste *signed char* será (para $n = 8$):

$$-2^{8-1} \cdot s + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Quando o bit s for zero (convenção para números positivos), o valor se reduz a

$$b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Portanto, o maior valor positivo possível nessa representação é $2^{8-1} - 1 = 127$, e o menor valor é 0. Quando o bit s for um (convenção para números negativos), o valor se reduz a

$$-2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Teremos o maior valor quando os bits $b_6, ..., b_1, b_0$ forem todos 1. Nesse caso, o valor da parte positiva seria $2^7 - 1$, e o valor total ficaria em $-2^7 + (2^7 - 1) = -1$. O menor valor se dará quando todos os bits da parte positiva forem 0, resultando num valor final de $-2^7 + 0 = -2^7 = -128$. Repare como os valores positivos e negativos que a convenção acomoda nunca se misturam. Resumindo, para o tipo *signed char* (8 bits), o menor valor possível é -128 e o maior é 127. Em geral, com n bits, teremos um valor máximo de $2^{n-1} - 1$ e um valor mínimo de -2^{n-1} .

Mais adiante justificaremos porque os computadores atuais usam essa representação.

Exemplos:

Caso 1: Representar, usando 8 bits, o número 66 como *signed char* e como *unsigned char*. Em binário, este número é: 1000010_2 .

Unsigned int:								Signed int							
b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀	s	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0

Neste caso, a representação binária é igual tanto para *signed* como para *unsigned int*.

Caso 2: Representar o número 175 usando 8 bits. Em binário: 10101111_2 .

Unsigned int:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	0	1	1	1	1

Signed int

s	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	0	1	1	1	1

Impossível!

A representação como *unsigned char* é trivial. Mas para *signed char* ocorre um problema. Os 7 bits $b_6b_5b_4b_3b_2b_1b_0$ não são suficientes para armazenar os 8 bits necessários para representar 175. O bit b_7 será equivocadamente interpretado como o bit s (deslocamento negativo). Ao invés de 175, o valor resultante será: $-2^{n-1} \cdot 1 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -81$. Isto explica alguns dos erros de conversão que podem ocorrer quando atribuímos o valor de uma variável de tipo *unsigned* para uma variável de tipo *signed*.

Caso 3: Representar o número -90 com 8 bits. Como o número é negativo, precisamos aplicar o deslocamento negativo -2^{n-1} , que para o caso o *signed char* será -2^7 , e descobrir quanto somar a -2^7 até obter -90 . Neste caso: $90 = -2^7 + 38$. A representação binária de 38 é 100110_2 .

Unsigned int:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	0	0	1	1	0

Signed int

s	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	0	0	1	1	0

Impossível!

A representação como *signed char* é fácil, basta atribuir 1 para o bit s de deslocamento negativo e preencher os bits positivos para formar o número 38 (100110_2). Repare que os bits b_6 até b_0 não representam +90, que é o valor absoluto do número que queremos representar, mas indicam o *complemento de* +90 para 128. Para *unsigned char*, a representação é impossível pelo simples fato de não existir um mecanismo para informar que o número é negativo. Se insistirmos em utilizar a mesma sequência de bits que em *signed char*, então os bits serão interpretados de forma diferente da desejada: $1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 166$. Isto explica alguns dos erros de conversão que podem ocorrer quando atribuímos o valor de uma variável de tipo *signed* para uma variável de tipo *unsigned*.