Tipos básicos

`char int float double`

C	x86-64	ia-32	P16	Java		Kotlin	
char	8	8	8	byte	8	Byte	8
short int	16	16	16	short	16	Short	16
int	32	32	16	int	32	Int	32
long int	64	32	32	long	64	Long	64
long long int	64	64			-		
float	32	32		float	32	Float	32
double	64	64		double	64	Double	64
long double	128	128					
				char	16	Char	16
				boolean		Boolean	

Na linguagem C os valores do tipo **`char`** são considerados valores numérico.

Modificadores de dimensão – **`short`**, **`long`** Modificadores de sinal – **`signed`**, **`unsigned`**

A palavra **`int`** pode ser omitida quando se lhe aplica um modificador. Exemplo: a palavra **`short`** isolada é equivalente a **`short int`**.

Operador **`sizeof`** – devolve a dimensão de uma variável ou tipo. A unidade de medida é a dimensão do tipo **`char`**.

Dimensões definidas pela especificação da linguagem C para cada tipo:

`char` >= 8 **`bit`** **`short`** >= 16 bit **`int`** >= 16 bit **`long`** <= 32 bit

`sizeof(char)` <= **`sizeof(short)`** <= **`sizeof(int)`** <= **`sizeof(long)`** <= **`sizeof(long long)`**

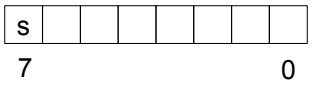
`sizeof(float)` <= s**`izeof(double)`** <= s**`izeof(long double)`**

A linguagem C não define a representação interna dos *floats*. A norma IEE754 é a mais utilizada.

Na linguagem C uma expressão pode envolver valores numéricos de tipos diferentes, acontecendo conversão implícita dos valores de tipos menores para valores de tipos maiores antes de serem operados.

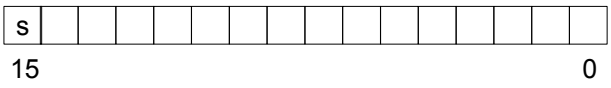
Valores inteiros

char



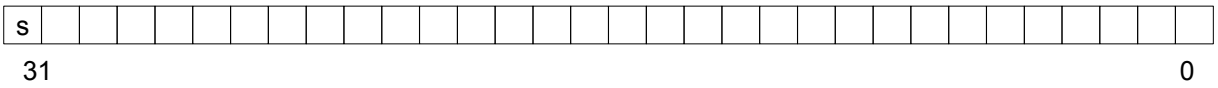
CHAR_BIT	**8**	número de bits num **char**
CHAR_MAX	**UCHAR_MAX** or **SCHAR_MAX**	valor máximo de **char**
CHAR_MIN	**0** or **SCHAR_MIN**	valor mínimo de **char**
SCHAR_MAX	**+127**	valor máximo de **signed char**
SCHAR_MIN	**-128**	valor mínimo de **signed char**
UCHAR_MAX	**255**	valor máximo de **unsigned char**

short



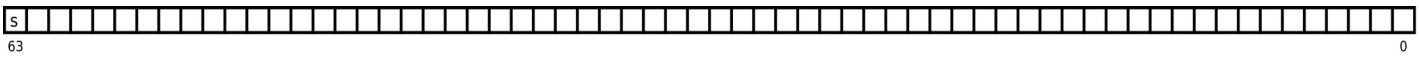
SHRT_MAX	**+32767**	valor máximo de **short**
SHRT_MIN	**-32768**	valor mínimo de **short**
USHRT_MAX	**65535**	valor máximo de **unsigned short**

int



INT_MAX	**+2147483647**	valor máximo de **int**
INT_MIN	**-2147483648**	valor mínimo de **int**
UINT_MAX	**4294967295**	valor máximo de **unsigned int**

long



LONG_MAX	**+9223372036854775807L**	valor máximo de **long**
LONG_MIN	**-9223372036854775808L**	valor mínimo de **long**
ULONG_MAX	**18446744073709551615UL**	valor máximo de **unsigned long**

Limites

No ficheiro (**/usr/include/**)**limits.h** encontram-se declarados os símbolos que representam os valores limite de cada tipo. Estes valores podem ser diferentes entre sistemas.

Portabilidade – stdint

Se se pretender portabilidade entre sistemas diferentes terá que se prestar atenção ao domínio de valores dos tipos utilizados.

Uma forma de garantir a mesma dimensão para os tipos básicos em sistemas diferentes, é usar os tipos com dimensão explicita, definidos em **`stint.h`**.

```
#include <stdint.h>

int8_t int16_t int32_t int64_t
uint8_t uint16_t uint32_t uint64_t
```

Possíveis definições de **`uint64_t`** para as arquiteturas IA-32 e X86-64 respetivamente:

```
typedef   unsigned long long  uint64_t;      IA-32

typedef   unsigned long       uint64_t;      x86-64
```

Constantes

notação decimal	**`ddd (0..9)`**	**`123`**
notação octal	**`000 (0..7)`**	**`023`** (dezanove)
notação hexadecimal	**`0xhh (0..7,a..f,A..F)`**	**`0xff`**

Não existe notação binária. (Ex: 0b00110110 – Kotlin).

Não é possível usar *underscore* como separador (Ex: 0xff_00_ff_00 – Kotlin).

Modificadores U e L em constantes

Por omissão uma constante é do tipo **`int`**. Os sufixos **`U`** e **`L`** modificam o tipo da constante para **`unsigned`** e **`long`**, respetivamente.

`3U` representa o valor três do tipo **`unsigned int`** (inteiro sem sinal).

`3UL` representa o valor três do tipo **`unsigned long int`** (inteiro longo sem sinal).

```
long d = 1L << 31;          resulta o valor 2147483648
long e = 1  << 31;          resulta o valor -2147483648

int f = (1  << 31) >> 31;   resulta o valor -1
int g = (1U << 31) >> 31;   resulta o valor 1
```

Valores reais

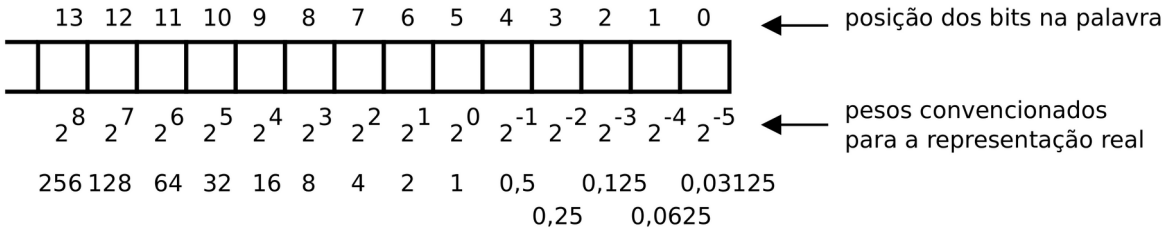
Representação em vírgula fixa

Os números reais podem ser representados em base binária usando as mesmas regras de significância posicional usadas em base decimal.

Por exemplo, 23,625 representa em base decimal, o mesmo valor que 10111,101 em base binária.

Em base decimal as posições representadas valem respetivamente 10^1 (10), 10^0 (1), 10^{-1} (0,1), 10^{-2} (0,01) e 10^{-3} (0,001), o valor representado resulta da adição de duas dezenas, mais três unidades, mais seis décimas, mais duas centésimas e mais cinco milésimas.

Em base binária as posições representadas valem respetivamente 2^4 (16), 2^3 (8), 2^2 (4), 2^1 (2), 2^0 (1), 2^{-1} (0,5), 2^{-2} (0,25) e 2^{-3} (0,125), o valor representado resulta da adição de dezasseis, mais quatro, mais dois, mais um, mais 0,5 e mais 0,125.



Exemplo

Realizar uma função para converter uma string, representando um valor real em base decimal para representação em binário sobre uma variável do tipo **`unsigned long int`**. Assumir como unidade o *bit* da posição 5.

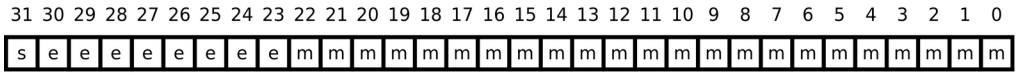
```
unsigned long string_to_real(char string[]) {
      unsigned long value = 0;
      int i;
      for (i = 0; string[i] != '.'; ++i) {
            int digit = (string[i] - '0') << 5;
            value = value * 10 + digit;
      }
      int fraction = 1;
      for (++i; string[i] != 0; ++i) {
            int digit = (string[i] - '0') << 5;
            value = ( value * 10 + digit);
            fraction *= 10;
      }
      return value / fraction;
}
```

Exercício

Completar o esboço de programa apresentado abaixo que realiza as quatro operações aritméticas básicas com números reais representados em binário com 5 casas fracionárias.

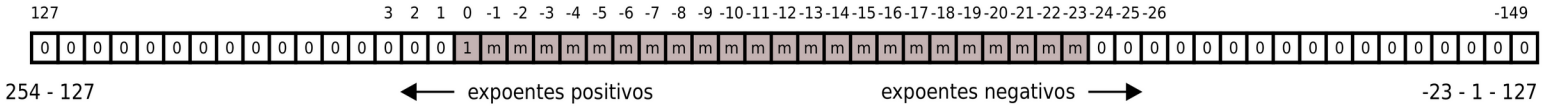
```
int main() {
      char op1_string[100];
      char op2_string[sizeof op1_string];
      char operator[sizeof op1_string];

      scanf("%s", op1_string);
      scanf("%s", operator);
      scanf("%s", op2_string);

      unsigned long op1 = string_to_real(op1_string);
      unsigned long op2 = string_to_real(op2_string);
      unsigned long result;

      switch (operator[0]) {
        case '+':
            result = op1 + op2;
            break;
        case '-':
            ...
            break;
        case 'x':
            ...
            break;
        case '/':
            ...
            break;
    }
    char result_string[20];
    real_to_string(result, result_string, sizeof(result_string));
    printf("%s\n", result_string);
}
```

Representação em vírgula flutuante (formato IEEE 754)



Sinal	Expoente	Mantissa	Valor
–	0 < e < 255	–	$(-1)^s * 2^{e-127} * 1.m$ *(normalized)*
–	0	!= 0	$(-1)^s * 2^{-126} * 0.m$ *(unnormalized)*
0	0	0	zero
1	0	0	- zero
–	255	!= 0	NaN (*Not a Number)*
0	255	0	+ infinito
1	255	0	- infinito

Na mantissa, os bits das posições 22 a 0 valem, respetivamente, os valores 2^{-1} a 2^{-23}.



Exemplo

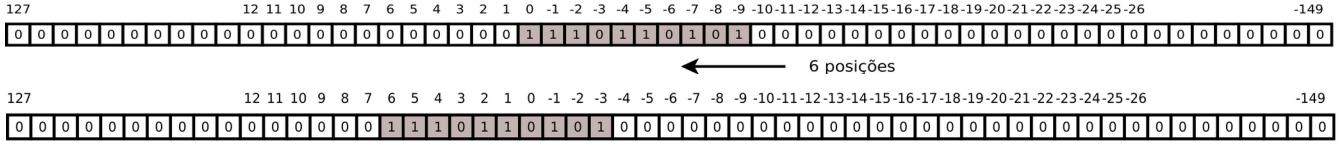
1 10000101 11011010100000000000000

s = –1 e = 133 m = 110110101

Mantissa $= 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-9}$

$= 0{,}5 + 0{,}25 + 0{,}0625 + 0{,}03125 + 0{,}0078125 + 0{,}001953125 = 0{,}853515625$

Valor representado = $(-1)1 * 2^{133-127} * 1{,}853515625 = -118{,}625$



Constantes

	Dimensão total	Dimensão da mantissa	Dimensão do expoente	
float	32	23	8	**23.5F 123.45e10F**
double	64	52	11	**23.5 123.45e10**
long double	128	113	15	**23.5L 123.45e10L**

Limites

25	33554432
24	16777216
23	8388608
22	4194304
21	2097152
20	1048576
19	524288
18	262144
17	131072
16	65536
15	32768
14	16384
13	8192
12	4096
11	2048
10	1024
9	512
8	256
7	128
6	64
5	32
4	16
3	8
2	4
1	2
0	1
-1	0,5
-2	0,25
-3	0,125
-4	0,0625
-5	0,03125
-6	0,015625
-7	0,0078125
-8	0,00390625
-9	0,001953125
-10	0,0009765625
-11	0,00048828125
-12	0,000244140625
-13	0,0001220703125
-14	0,0000610351563
-15	0,0000305175781
-16	0,0000152587891
-17	0,0000076293945
-18	0,0000038146973
-19	0,0000019073486
-20	0,0000009536743
-21	0,0000004768372
-22	0,0000002384186
-23	0,0000001192093
-24	0,0000000596046
-25	0,0000000298023

```
#include <float.h>
```

Maior magnitude codificável:

```
0 11111110 111 1111 1111 1111 1111 1111
```

$2^{254-127} * (1,11111111111111111111111)_2$

```
#define FLT_MAX \
    3.40282346638528859811704183484516925e+38F
```

Menor magnitude codificável (normalizado):

```
0 00000001 000 0000 0000 0000 0000 0000
```

$2^{1-127} * (1,00000000000000000000000)_2$

```
#define FLT_MIN \
     1.17549435082228750796873653722224568e-38F
```

Maior magnitude sem erro:

```
0 10010110 111 1111 1111 1111 1111 1111
```

$2^{150-127} * (1,11111111111111111111111)_2$

Valor 16777215.

Sempre que a mantissa (ou significante) tenha um afastamento entre dígitos significativos maior que 23 posições, há erro na codificação do *float*.

Exemplo

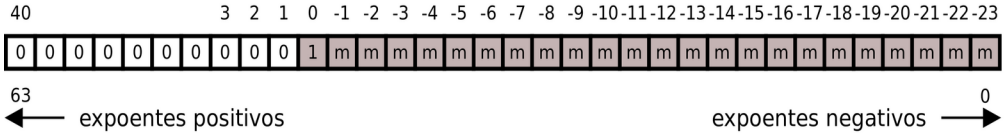
Programa que imprime o conteúdo binário de um valor do tipo **`float`**.

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

typedef union {
      float f;
      unsigned int i;
} Float;

int main() {
      Float x = {.f = 30.4375};

      int bit = sizeof bit * CHAR_BIT - 1;

      printf("%c ", ((x.i & (1 << bit)) == 0) ? '0' : '1');       /* sinal */

      for (--bit; bit >= 23; --bit)                               /* expoente */
            printf("%c", ((x.i & (1 << bit)) == 0) ? '0' : '1');
      putchar(' ');
      for ( ; bit >= 0; --bit)
      /* mantissa */
            printf("%c", ((x.i & (1 << bit)) == 0) ? '0' : '1');
      putchar('\n');
}
```

Exercícios

1. Fazer um programa que leia um valor real da consola e imprima a respetiva respetiva representação binária. Verificar que a codificação do valor 16777217,0 é igual à do valor 16777216,0.
2. Fazer um programa que verifique se a parte inteira de um valor do tipo **`float`** é representável como um valor do tipo **`int`**.

Representação do valor real num long com perda dos pesos de -24 a -149 e dos pesos de 41 a 127.



Valores e variáveis

Em linguagem C não há inferência de tipo na definição de variáveis – é necessário explicitar o tipo.

C	Kotlin
`char a;`	`var a: Byte`
`int b;`	`var b: Int`
`int c = 10;`	`var c: Int = 10`
`int d = 3;`	`var d = 3;`

Em linguagem C os valores não alteráveis (o equivalente a **`val`** da linguagem Kotlin) podem ser definidos de duas formas: através de macros ou colocando a palavra **`const`** no início da definição de variável.

```
#define     TEN   10
```

ou

```
const int ten = 10;
```

As macros são um mecanismo de substituição textual que ocorre antes da compilação do programa. Podem portanto ser utilizadas para outros fins. No exemplo, todas as ocorrências de **`TEN`** no texto do programa, serão substituídas por **`10`**. A utilização de macros é uma forma muito comum de se definirem valores ou constantes como são designados em linguagem C.

A definição prefixada de **const** significa em Kotlin, que o valor é determinado em compilação. Em linguagem C depende da posição, se for externa às funções o valor é determinado em compilação, como em Kotlin, se for local às funções é calculado em execução.

Por convenção, estes identificadores são formados por letras maiúsculas. Nos identificadores compostos, as palavras são separadas por *underscore*. Por exemplo, o identificador que represente a dimensão máxima de uma palavra:

```
#define     WORD_MAX_SIZE     50
```

Na formação de nomes de variáveis ou funções a convenção geral é utilizar letras minúsculas e no caso de nomes compostos por várias palavras, separar com *underscore*. Por exemplo uma variável que represente o número de pessoas numa sala poderia ter o nome **`people_in_room`**. Tradicionalmente, embora atualmente se desaconselhe, os nomes podem ser formados por aglomeração de abreviaturas como por exemplo **`errno`**, para designar a variável que armazena o número de um erro.

Operações numéricos

Operações aritméticas

`+`	adição	`*`	multiplicação	`%`	resto da divisão inteira	`++`	incremento
`-`	subtração	`/`	divisão			`--`	decremento

Operações diretas sobre bits (*bitwise operations*)

`<<`	deslocar para a esquerda	`a << 3`	`a shl 3`
`>>`	deslocar para a direita	`b >> 8`	`b shr 8`
`&`	conjunção	`a & b`	`a and b`
`|`	disjunção	`a | b`	`a or b`
`^`	disjunção exclusiva	`a ^ b`	`a xor b`
`~`	negação	`~a`	`inv a`

Na deslocação para a direita, o *bit* de maior peso mantém o valor se estiver a operar sobre um tipo com sinal ou recebe zero se estiver a operar sobre um tipo sem sinal.

Enumerados

Em linguagem C um enumerado têm a seguinte sintaxe:

enum *identifier* **{** *identifier* [= *constant-expression*] [, *identifier* = [*constant-expression*]] **};**

Os elementos de um enumerado são do tipo **int** e têm o valor numérico da expressão associada. Se não existir expressão associada, o valor numérico é o sucessor do valor anterior ou zero no caso de ser o primeiro elemento.

```
enum mounth { JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
NOV, DEZ};

enum mounth m = FEV;

int n = m;
```

O identificador de um elemento do enumerado é global. Não pode existir um identificador igual em mais do que um enumerado.

Avaliação booleana

Em linguagem C não existem variáveis do tipo booleano. No entanto qualquer valor numérico do tipo **char**, **short**, **int** ou **long** pode ser avaliado do ponto de vista booleano. O critério é o seguinte: o valor numérico zero é avaliado como **falso**; um valor numérico diferente de zero é avaliado como **verdadeiro**.

Operações de comparação

`==`	igualdade	`>`	maior	`>=`	maior ou igual
`!=`	diferença	`<`	menor	`<=`	menor ou igual

Das operações de comparação resultam valores booleanos – verdadeiro ou falso.

Um valor booleano pode ser afetado a uma variável de qualquer tipo numérico ou ser operado com operadores numéricos. Para este efeito, o valor booleano falso é equivalente a **zero** e o valor booleano verdadeiro é equivalente a **um**.

```
int x = 20;
int y = x == 20;
```

A variável **y** é afetada com o valor **um**, resultante da expressão **x == 20**, tem o valor booleano **verdadeiro**.

Os valores booleanos verdadeiro e falso não têm representação própria na linguagem como os literais **true** e **false** em Java ou Kotlin. Quando se pretende representar estes valores usam-se os valores numéricos 1 ou 0. Diretamente, ou simbolicamente como macros

```
#define     FALSE       0
#define     TRUE        !FALSE
```

ou como enumerado

```
typedef enum {false, true} boolean;
```

Operações booleanas

`||`	disjunção
`&&`	conjunção
`!`	negação

Nas expressões lógicas a avaliação dos operandos realiza-se da esquerda para a direita. Se nesta avaliação um dos resultados for igual ao elemento absorvente os restantes operandos já não serão avaliados.

Arrays

Em linguagem C, como em geral, *arrays* são sequências de valores do mesmo tipo armazenados em posições contiguas da memória.

Array unidimensional

Sintaxe de definição de *array* unidimensional:

***type identifier* [*constant-expression*];**

Exemplos:

- *Array* com cinco elementos do tipo **`int`**.

```
int x[5];
```

Equivalente em Java	Equivalente em Kotlin
`int[5] x;`	`var x: IntArray(5)`

- *Array* com cinco elementos do tipo **`int`** inicializado com os valores indicados.

```
int x[5] = { 10, 20, 30, 40, 50 };
```

Equivalente em Java	Equivalente em Kotlin
`int[] x = { 10, 20, 30, 40, 50 };`	`var x: IntArray` `= intArrayOf(10, 20, 30, 40, 50)`

O acesso aos elementos do *array* realiza-se apenas através do operador indexação. Os índices vão de zero – primeira posição – à dimensão menos um – última posição.

(desenho de um array)

Em linguagem C não se pode afetar um *array* com a totalidade do conteúdo de outro *array* numa operação de afetação. Poder-se-á contudo, realizar a afetação posição a posição.

Exemplo:

```
int a[3];
int b[] = {10, 20, 30}:
```

`a = b;`	é inválido em linguagem C.

`a[0] = b[0];`
`a[1] = b[1];`
`a[2] = b[2];`
	é válido, embora não seja a solução adequada para copiar um *array* extenso.

Texto

Caracteres

Os caracteres são codificados como valores numéricos segundo uma tabela de codificação (Unicode, ASCII, ISO-8859-xx, etc).

Um caractere é representado por um valor numérico que pode ser afetado a uma variável numérica de qualquer tipo (**`char`**, **`short`**, **`int`** ou **`long`**). Embora o mais adequado seja o tipo **`char`**.

Em linguagem C qualquer das seguintes definições de variável é válida, embora algumas possam não ser adequadas.

```
char c = 'a';
char d = 'ç';
int e = 'f';
long f = 'ã';
```

A especificação de um caractere constante, incluindo as sequências de escape, é semelhante à linguagem Java e Kotlin.

```
'a'..'z' 'A'..'Z' '\a' '\b' '\f' '\n' '\r' '\t' '\v' '\\' '\'' '\"'
'\0123' '\123' '\xhh'
```

`\a` - alerta	**`\b`** - backspace	**`\f`** – avanço de página
`\n` – nova linha	**`\r`** – coloca cursor na coluna 0	**`\t`** – tabulador horizontal
`\v` – tabulador vertical	**`\\`** - o próprio **`\`**	**`\'`** - plica
`\"` - aspas		

Há métodos de codificação, como o caso do Unidoce UTF-8, em que certos caracteres podem ter valores numéricos superiores à capacidade de representação do tipo **`char`**. É o caso dos caracteres com acento e 'c' de cedilha.

Strings

A linguagem C não define formalmente a existência de variáveis do tipo *string* como String em Java ou em Kotlin. No entanto utiliza a mesma notação sintática para definir literais do tipo *string*.

```
"Isto é uma string em C, em Java ou em Kotlin"
```

Em linguagem C um literal do tipo *string* é formalmente considerado um *array* de elementos do tipo **`char`**. Por exemplo:

```
char greeting[] = "Olá";
```

Os códigos numéricos dos caracteres que compõem a palavra "Olá" são armazenados nas sucessivas posições do *array* **`greeting`**. Se se utilizar o codificação UTF-8 o código do caractere 'á' é constituído por dois *bytes* e ocupa as posições 2 e 3.

Na posição a seguir à do código do último caractere é colocado o valor numérico zero, para indicar o final da *string* (indiretamente define a dimensão da *string*).
Assim, a definição acima produz um *array* de valores do tipo **`char`** com cinco posições e a seguinte ocupação:

`'O'`	`'l'`		`'á'`	Indicação de fim de *string*
0	**1**	**2**	**3**	**4**
79	**108**	**\303**	**\241**	**\0**

Utilizando o operador indexação sobre este *array* poderemos obter ou modificar os valores numéricos de cada posição. **`greeting[0]`** corresponde ao valor numérico 79 que representa o caractere 'O'; **`greeting[2]`** dá acesso ao valor numérico 303 que é parte do código do caractere 'á' em codificação UTF-8.

Em linguagem C não é possível embutir valores em *strings* usando $, nem é possível definir texto em bruto delimitado por três aspas `"""`.

Exemplos de definição de strings em C:

```
char string1[] = "string terminada com mudança de linha\n";
char string2[] = "uma string separada"
                 "em duas linhas";
```

Expressões

Uma expressão tem um valor associado que resulta da aplicação de operações sobre valores. Esses valores podem ser constantes, variáveis, chamadas a funções ou o resultado de outras operações.

Na avaliação de uma expressão estão envolvidos três importantes conceitos: prioridade dos operadores, ordem de associação de operadores e ordem de avaliação de operandos.

Operadores	Associatividade	Operação
`() [] -> .`	left to right	função; indexação; campo de estrutura
`! ~ ++ -- + - * & (type) sizeof`	right to left	falso; negar bit a bit; incrementar; decrementar; positivo; negativo; desreferenciar; endereço de; converter o tipo; dimensão
`* / %`	left to right	multiplicação; divisão; resto da divisão
`+ -`	left to right	soma subtracção
`<< >>`	left to right	deslocamento dos bits
`< <= > >=`	left to right	relacionais
`== !=`	left to right	igual; diferente
`&`	left to right	e bit-a-bit
`^`	left to right	xor bit-a-bit
`|`	left to right	ou bit-a-bit
`&&`	left to right	e
`||`	left to right	ou
`?:`	right to left	expressão condicional
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left	afetação e afetação com operação
`,`	left to right	operador vírgula

Prioridade dos operadores

Exemplo 1

```
a + b * c = a + (b * c)
```

Exemplo 2

Considerando `a = 6; mask = 3;` qual o valor de `a & mask == mask` ?

`a & (mask == mask);` 6 & (3 == 3) é igual a 0

`(a & mask) == mask;` (6 & 3) == 3 é igual a 0

Considerando **`a = 6; mask = 2;`** qual o valor de **`a & mask == mask`** ?

`a & mask == mask;` 6 & (1 == 1) é igual a 0

`(a & mask) == mask;` (6 & 2) == 2 é igual a 1

Ordem de associação

Ordem com que são realizadas as operações (a maioria associa das esquerda para a direita)

```
a + b + c = (a + b) + c
```

`f1() + f2() + f3()` A ordem de associação não define a ordem de avaliação.

Ordem de avaliação dos operandos

A ordem de avaliação dos operandos só está definida para as operações **`&&`**, **`||`**, **`?:`**, e ‘**`,`**’.

(*lazzy evaluation*)

```
f1() && f2()
f1() || f2()
```

Expressões como as seguintes produzem resultados imprevisíveis.

`x = f() + g();` se existir alguma dependência entre f() e g()

```
printf("%d %d\n", ++n, power2(2, n));


int x = 5;
int f() {
     return  x += 1;
}
int g() {
     return  x *= 10;
}
f() + g();
```

Em geral, quando numa expressão estão envolvidos operadores de incremento ou decremento, afetações ou chamadas a funções podem ser produzidos efeitos indefinidos.

Expressão condicional

A expressão **`var = expr1 ? expr2 : expr3`** é equivalente a

```
if (expr1)
     var = expr2;
else
     var = expr3;
```

com a vantagem de poder ser utilizada onde é suposto aparecer um valor.

Exemplo

```
int minor = a < b ? a : b;
```

Operador vírgula

O operador vírgula permite colocar mais do que uma expressão em locais onde sintaticamente só poderia ser colocada uma expressão. As expressões individuais vão sendo avaliadas da esquerda para a direita e os seus valores vão sendo descartados, exceto o valor da última expressão que é utilizado como o valor associado à sequência de expressões.

Exemplo 1

```
for (i = 0, j = 0; i < MAX; ++i, j += 2)
```

Exemplo 2

```
if (failure)
      return (error = failure_code, -1);
```

Exemplo 3

```
x = 1, 2;   /* valor final de x é 1 */
y = (3, 4); /* valor final de y é 4 */
```

Conversão entre tipos básicos

Quando um operador tem operandos de tipos diferentes o de menor amplitude é convertido para o de maior amplitude antes de se realizar a operação.

Conversão implícita

Por cada operação o tipo mais pequeno é promovido para o tipo maior para que não se perca informação. É o caso da conversão no sentido **`char -> short -> int -> long; int-> float; float-> double`**.

Expansão com manutenção de sinal.

Comparação entre valores com sinal (signed) e valores sem sinal (unsigned).

Se se afetar uma variável de tipo menor com uma expressão de tipo maior pode haver perda de informação. É caso da conversão no sentido **`float -> int`** – trunca, **`double -> float`** – trunca ou arredonda. Nestes casos o compilador pode emitir um aviso.

Uma expressão booleana é convertida para **`int`** e vale 1 ou 0.

```
int is_space(char c) {
    return c == ' ';
}
```

Conversão explícita (cast)

O programador indica explicitamente para que tipo o operando é convertido. É equivalente à conversão implícita para o tipo da variável em caso de afetação.

Referências

The C programming Language – capítulo 2

Computer Systems – capítulo 2