FEELT31201
Programação Procedimental
...

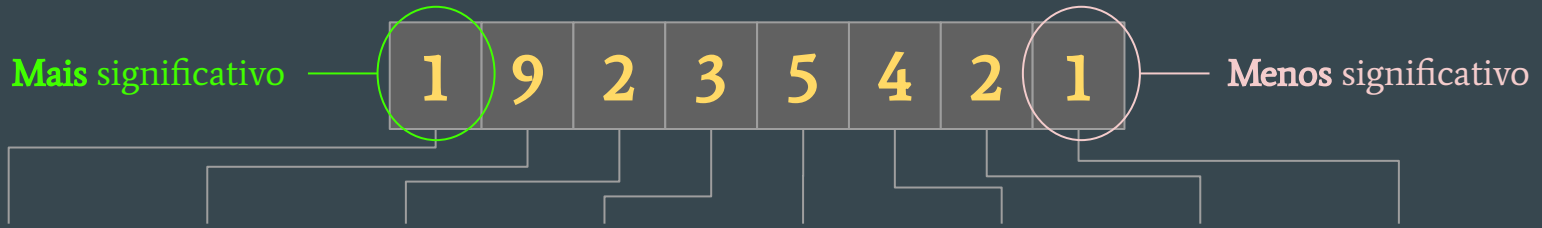
Aula 02:
Bases Numéricas, Tipos de Dados Primitivos, Qualificadores, Aritmética

Prof. Igor Peretta

Bases Numéricas de Interesse (sistemas numéricos posicionais)

Decimal N_{10}

Caracteres: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

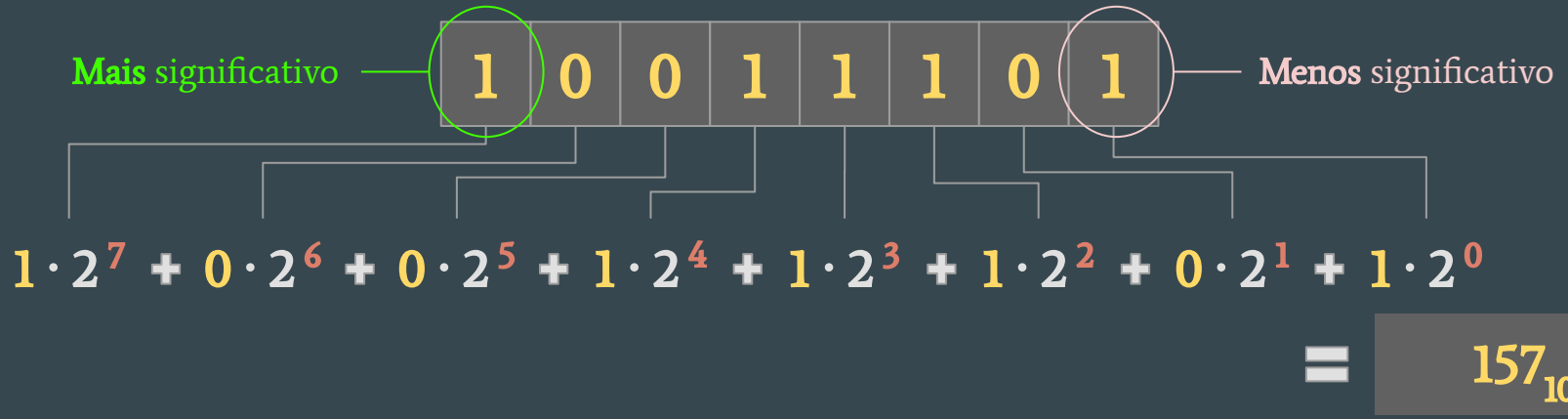


$$1 \cdot 10^7 + 9 \cdot 10^6 + 2 \cdot 10^5 + 3 \cdot 10^4 + 5 \cdot 10^3 + 4 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0$$

$$= 19.235.421_{10}$$

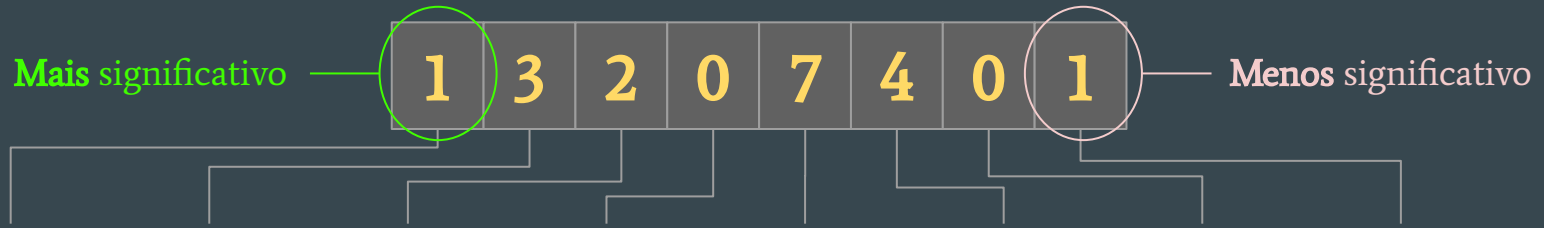
Binário N_2

Caracteres: 0, 1



Octal N_8

Caracteres: 0, 1, 2, 3, 4, 5, 6, 7

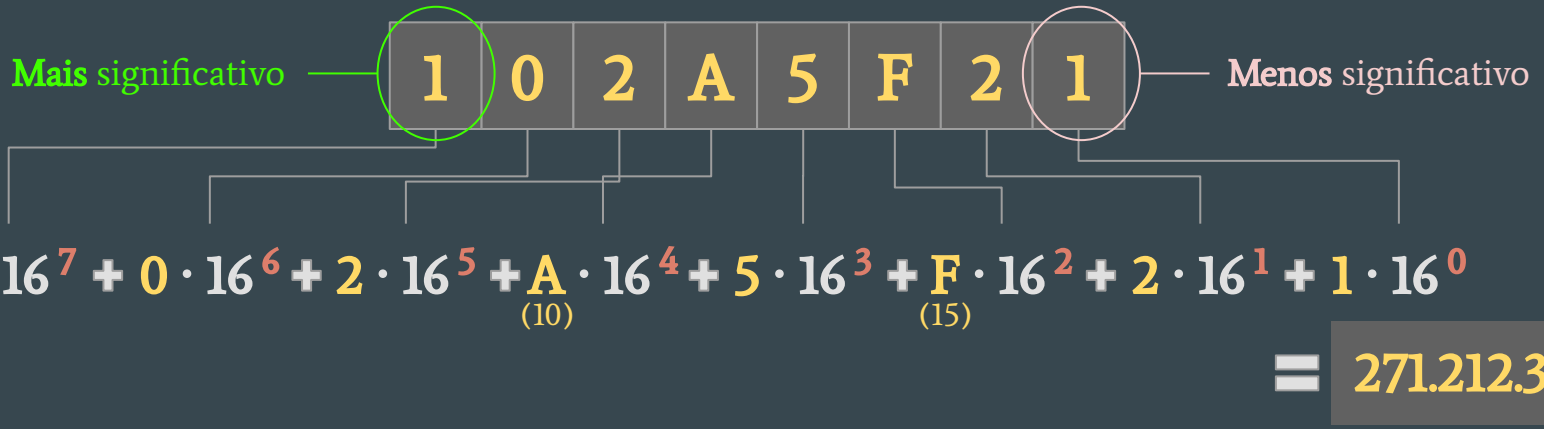


$$1 \cdot 8^7 + 3 \cdot 8^6 + 2 \cdot 8^5 + 0 \cdot 8^4 + 7 \cdot 8^3 + 4 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0$$

$$= 2.952.961_{10}$$

Hexadecimal N_{16}

Caracteres: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F



Combinações

O número de dígitos utilizados para representar a informação impacta diretamente no número de possíveis diferenças, ou seja, é determinante no universo de dados diferentes representáveis.

Na base decimal, por exemplo, um número com 4 dígitos possui 10.000 possíveis representações diferentes (10^4): de **0000** a **9999**.

Como saber em outras bases? Basta usar **combinatória**: $\#base^{\#dígitos}$

Exemplo: Base hexadecimal (16 caracteres possíveis) com 3 dígitos = $16 \cdot 16 \cdot 16 = 16^3 =$ **4096** possíveis representações diferentes (de **000** a **FFF**).

Base binária (2 caracteres possíveis) com 8 dígitos = $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$ representações possíveis (de **00000000** a **11111111**).

Mínima quantidade de dígitos

Para uma base *b*, o número de dígitos *d* fornece *P* possibilidades, de 0_{10} a $(P-1)_{10}$; logo:

$$b^d = P;\ \log_b P = d$$

Logo, o número mínimo de dígitos d_{min} para representar um número n_{10} na base *b*:

$$\lfloor \log_b n \rfloor + 1 = d_{min}$$

Exemplos:

- 129_{10} na base binária: $\lfloor \log_2 129 \rfloor + 1 = 8$ ($129_{10} = 10000001_2$)
- 256_{10} na base binária: $\lfloor \log_2 256 \rfloor + 1 = 9$ ($256_{10} = 100000000_2$)
- 129_{10} na base hexadecimal: $\lfloor \log_{16} 129 \rfloor + 1 = 2$ ($129_{10} = 81_{16}$)

Decimal para outras bases

Considere o número x_{10}, a base numérica n e a mínima quantidade de dígitos $ndig$ para conseguir representar x_n.

Logo, o algoritmo para conversão de representação é:

```
ndig ⇐ 1 + max( 0, ⌊log(x)/log(n)⌋ )
para i ⇐ ( ndig - 1 ) .. 0:
   p ⇐ n ^ i
   d ⇐ ⌊x/p⌋
   x ⇐ x mod p
   apresenta d
```

Exemplo: 157_{10} em binário (base 2)?

```
ndig = 1 + max( 0, ⌊log_2(157)⌋ ) = 8
⌊157 / 2^7⌋ = 1; 157 mod 2^7 = 29
⌊29 / 2^6⌋ = 0; 29 mod 2^6 = 29
⌊29 / 2^5⌋ = 0; 29 mod 2^5 = 29
⌊29 / 2^4⌋ = 1; 29 mod 2^4 = 13
⌊13 / 2^3⌋ = 1; 13 mod 2^3 = 5
⌊5 / 2^2⌋ = 1; 5 mod 2^2 = 1
⌊1 / 2^1⌋ = 0; 1 mod 2^1 = 1
⌊1 / 2^0⌋ = 1; 1 mod 2^0 = 0
```

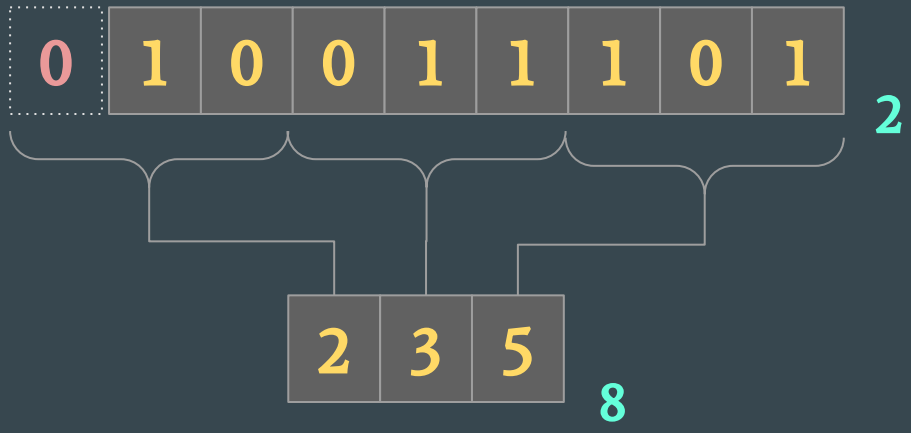
Resposta: 10011101_2

Exemplo: 157_{10} em hexadecimal (base 16)?

```
ndig = 1 + max( 0, ⌊log_16(157)⌋ ) = 2
⌊157 / 16^1⌋ = 9; 157 mod 16^1 = 13
⌊13 / 16^0⌋ = 13 = D; 13 mod 16^0 = 0
```

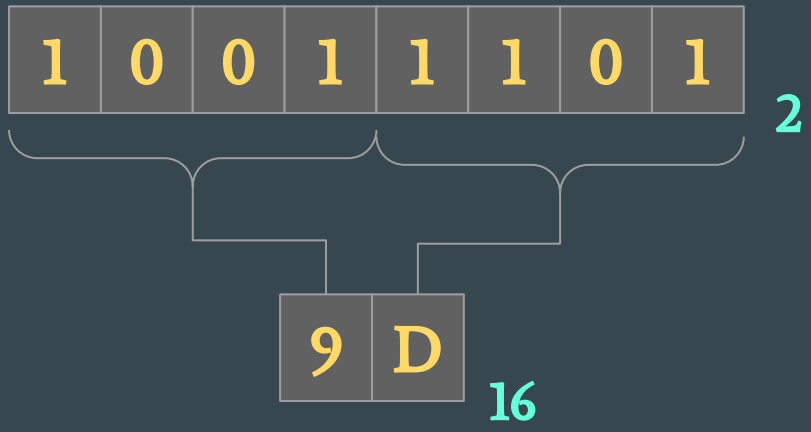
Resposta: $9D_{16}$

Octal de/para binário



BIN	OCT
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Hexadecimal de/para binário



BIN	HEX
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Equivalências

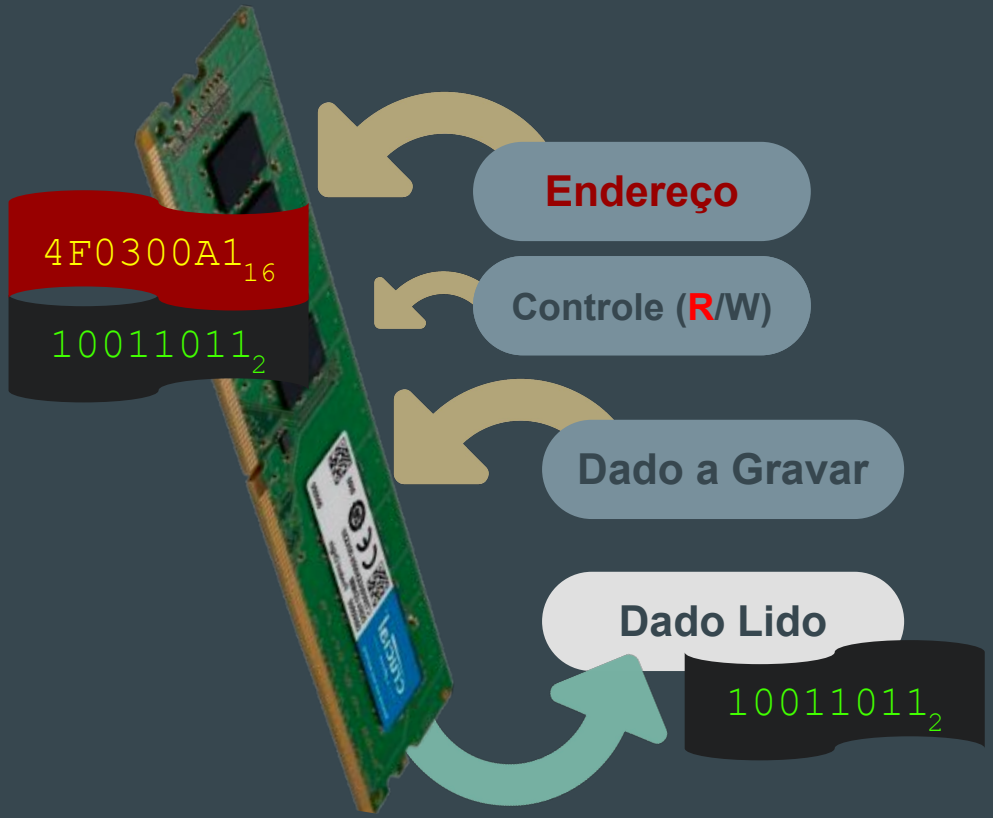
Como o idioma não altera o valor de um número (e.g. “11” pode ser chamado de *onze, eleven, once, elf, jū ichi, shíyī, 'ahad eashar, kumu n'emu, dek unu, pateĩ, jedenaście, unsprezece, on bir, undecim,* entre outros), a base numérica também não altera o número, apenas sua forma de registro concreto.

Portanto, o número “11” é representável por: 11_{10}; 1011_{2}; 13_{8}; B_{16}; 23_{4}; 10_{11}; $10001010_{\sqrt{2}}$; $101{,}01022122\ldots_{\pi}$; etc.

DEC	BIN	OCT	HEX
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Memória

Representando Memória



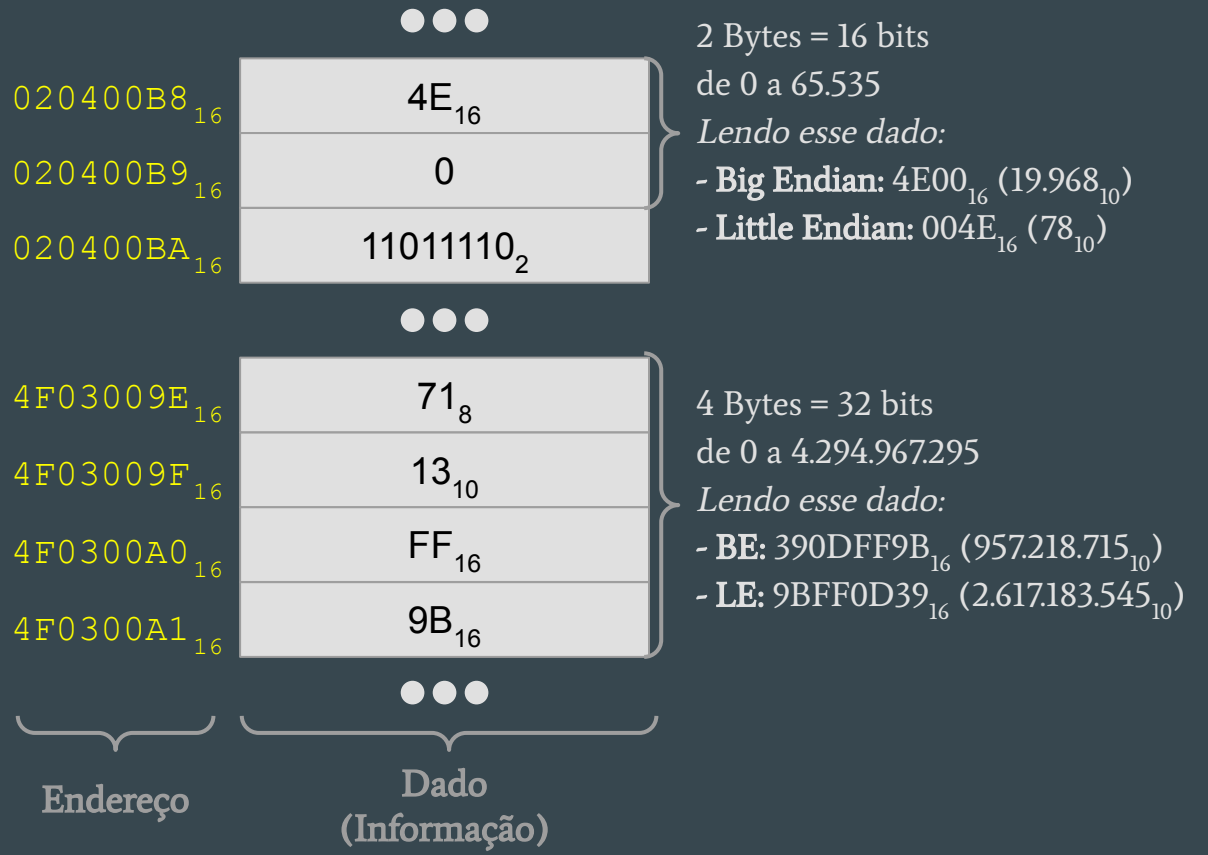
Cada endereço = 1 Byte de informação

Endereço	Dado (Informação)
$020400B8_{16}$	$4E_{16}$
$020400B9_{16}$	0
$020400BA_{16}$	11011110_2
●●●	●●●
$4F03009E_{16}$	71_8
$4F03009F_{16}$	13_{10}
$4F0300A0_{16}$	FF_{16}
$4F0300A1_{16}$	$9B_{16}$
	●●●

Dados Grandes

Cada endereço de memória armazena **1 Byte** de informação, possibilitando 256 (2^8) diferentes possibilidades , ou seja, números entre **0 e 255**.

E se quisermos armazenar dados com mais de 1 Byte?



Sistema Internacional (base 2, ordens de 2^{10} em 2^{10})

Medida	Nome	Bytes	bits
1 B	Byte $(2^{10})^0$	1	8
1 kB	QuiloByte $(2^{10})^1$	1.024	8.192
1 MB	MegaByte $(2^{10})^2$	1.048.576	8.388.608
1 GB	GigaByte $(2^{10})^3$	1.073.741.824	8.589.934.592
1 TB	TeraByte $(2^{10})^4$	1.099.511.627.776	8.796.093.022.208
1 PB	PetaByte $(2^{10})^5$	1.125.899.906.842.624	9.007.199.254.740.992

Identificadores e Relação com Memória

Variáveis

- **Variáveis** são “apelidos” para **endereços de memória**
- **Atribuições** e **usos da variável** implicam **escrita** ou **leitura** na memória
- Para **saber o endereço** ao qual a variável se refere, basta usar o operador `&` antes do identificador da variável
- **Ponteiros** são variáveis que **armazenam endereços de memória** (acesso indireto), mas vamos vê-los mais pra frente (exemplo, parâmetro de `scanf`)
- Os **bytes de memória** necessários para armazenar um dado referenciado pela variável vai depender do **tipo de dado da variável**

Identificadores e Sensibilidade à Caixa Alta

O nome de uma variável em C pode ser qualquer **identificador válido**:

- Um identificador válido é uma série de caracteres que consiste em letras (maiúsculas ou minúsculas, sem diacríticos), dígitos e o caractere *underscore* (_) que não começa com um dígito e que não seja uma palavra-chave

Letras	`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z` `a b c d e f g h i j k l m n o p q r s t u v w x y z`
Dígitos	`0 1 2 3 4 5 6 7 8 9`
Caracteres especiais	`_`

- C é sensível à caixa alta, ou seja, **maiúsculas ou minúsculas são letras diferentes** (codificações distintas); exemplo `A1` e `a1` são identificadores distintos

Palavras-chave, não usar como identificador

Keywords				
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Keywords added in C99 standard

`_Bool` `_Complex` `_Imaginary` `inline` `restrict`

Keywords added in C11 standard

`_Alignas` `_Alignof` `_Atomic` `_Generic` `_Noreturn` `_Static_assert` `_Thread_local`

Tipos de Dados Primitivos

Primitive Data Types

The primitive data types in C language are the inbuilt data types provided by the C language itself. The following primitive data types in C are available:

- **Integer** Data Type, `int` : is used to declare a variable that can store numbers without a decimal. The keyword used to declare a variable of integer type is “int”.
- **Float** data Type, `float` : declares a variable that can store numbers containing a decimal number.
- **Double** Data Type, `double` : can store floating point numbers but gives precision double than that provided by float data type.
- **Character** Data Type, `char` : declares a variable that can store a character constant; can only store one single character.
- **Void** Data Type, `void` : doesn’t create any variable but returns an empty set of values.

Data Type Qualifiers

There are certain data type qualifiers that can be applied to them in order to alter their range and storage space and thus, fit in various situations as per the requirement. The data type qualifiers available in C are:

- `short`
- `long`
- `signed`
- `unsigned`

It should be noted that the above qualifiers cannot be applied to float and can only be applied to integer and character data types.

Adicionados ao C99

Bool

Complex

Imaginary

C Data Types		Size (in bytes)	Range
Integer Data Types*	int	4	
	signed int	4	
	unsigned int	4	
	short *int*	2	-32,768 to 32,767
	signed short *int*	2	-32,768 to 32,767
	unsigned short int	2	0 to 65535
	long *int*	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	signed long *int*	8	
	unsigned long *int*	8	
Floating Point Data Types	float	4	1.4E-45 to 3.4E+38 (*precision 6 decimal places*)
	double	8	4.9E-324 to 1.7E+308
	long double	16	3.6E-4951 to 1.2E+4932
Character Data Types	char	1	-128 to 127
	signed char	1	-128 to 127
	unsigned char	1	0 to 255

$-2^{8 \cdot bytes-1}$ to $2^{8 \cdot bytes-1} - 1$

0 to $2^{8 \cdot bytes} - 1$

`sizeof(.)`;
e.g. sizeof(int)

*(*) some old compilers may consider **1**, **2** and **4 bytes** for `short`, `int` and `long` data types, respectively, and don't implement `long double`*

`printf` specifiers per Data Type

specifier	Output
d or i	Signed decimal integer
u	Unsigned decimal integer
o	Unsigned octal
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (uppercase)
f	Decimal floating point, lowercase
F	Decimal floating point, uppercase
e	Scientific notation (mantissa/exponent), lowercase
E	Scientific notation (mantissa/exponent), uppercase
g	Use the shortest representation: %e or %f
G	Use the shortest representation: %E or %F
a	Hexadecimal floating point, lowercase
A	Hexadecimal floating point, uppercase
c	Character
s	String of characters
p	Pointer address
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.
%	A % followed by another % character will write a single % to the stream.

Source: http://www.cplusplus.com/reference/cstdio/printf/

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Aritmética

Arithmetic in C

- Most C programs perform calculations using the C arithmetic operators.
- The asterisk (`*`) indicates multiplication and the percent sign (%) denotes the remainder operator, which is introduced below.
- In algebra, to multiply *a* times *b*, we simply place these single-letter variable names side by side as in *ab*.
- In C, however, if we were to do this, `ab` would be interpreted as a single, two-letter name (or identifier).
- Therefore, C requires that multiplication be explicitly denoted by using the * operator as in `a * b`.
- The arithmetic operators are all binary operators.
- For example, the expression `3 + 7` contains the binary operator `+` and the operands `3` and `7`.

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	`f + 7`
Subtraction	–	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Integer Division and the Remainder Operator

- Integer division yields an integer result
- For example, the expression `7 / 4` evaluates to `1` and the expression `17 / 5` evaluates to `3`
- C provides the remainder operator, %, which yields the remainder after integer division
- Can be used only with integer operands
- The expression `x % y` yields the remainder after `x` is divided by `y`
- Thus, `7 % 4` yields `3` and `17 % 5` yields `2`

Rules of Operator Precedence

- C applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those in algebra.
- The rules of operator precedence specify the order C uses to evaluate expressions. When we say evaluation proceeds from left to right, we're referring to the associativity of the operators.
- As in algebra, it's acceptable to place unnecessary parentheses in an expression to make the expression clearer; these are called redundant parentheses.

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the *innermost* pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they're evaluated left to right.
* / %	Multiplication Division Remainder	Evaluated second. If there are several, they're evaluated left to right.
+ -	Addition Subtraction	Evaluated third. If there are several, they're evaluated left to right.
=	Assignment	Evaluated last.

Lógica Booleana e Operadores Relacionais

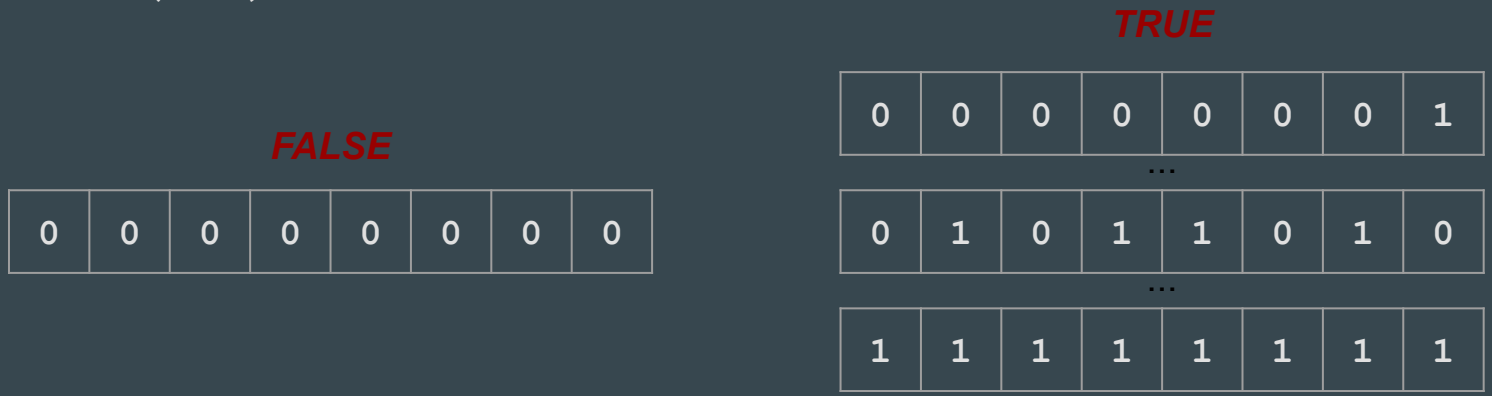
Boolean Operators NOT, OR, and AND

Boolean condition	Operator	Code	Description
NOT	!	!A	*True* if A is *false* *False* if A is *true*
OR	||	A || B	*True* if A is *true*, or B is *true*, or both are *true* *False* only when both A and B are *false*
AND	&&	A && B	*True* only when both A and B are *true*, Otherwise *false*

Equality and Relational Operators

- If the condition is true (i.e., the condition is met) the statement in the body of the if statement is executed.
- If the condition is false (i.e., the condition isn't met) the body statement is not executed.
- Whether the body statement is executed or not, after the if statement completes, execution proceeds with the next statement after the if statement.
- Conditions in if statements are formed by using the equality operators and relational operators.

- The relational operators all have the same level of precedence and they associate left to right.
- The equality operators have a lower level of precedence than the relational operators and they also associate left to right.
- In C, a condition may actually be *any expression that generates a zero (false) or nonzero (true) value*.



Algebraic equality or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Relational operators			
$>$	`>`	`x > y`	x is greater than y
$<$	`<`	`x < y`	x is less than y
$\geq$	`>=`	`x >= y`	x is greater than or equal to y
$\leq$	`<=`	`x <= y`	x is less than or equal to y
Equality operators			
$=$	`==`	`x == y`	x is equal to y
$\neq$	`!=`	`x != y`	x is not equal to y