Programação de Computadores I

Aula 05

Programação: Tipos, Variáveis e Expressões

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2011-1

Valores

- **Valor** é uma entidade da linguagem que pode ser manipulada durante a execução do programa através de operações.
- Um valor representa um dado que pode ser processado.
- Exemplos:
  - números inteiros: 101, 453, 1231
  - textos: ”bom dia”, ”digite um número”

Tipos

- **Tipo** é um conjunto de valores que apresenta comportamento uniforme sob um conjunto de opreações.
- Os tipos resultam da classificação dos valores em categorias que levam em conta a sua representação na memória do computador, e as operações que podem ser realizadas com eles.
- Assim os números inteiros formam um tipo, e seus valores podem ser manipulados através de operações aritméticas, por exemplo.
- Já as sequêcias de caracteres (textos) formam outro tipo, e podem ser exibidas na tela, por exemplo.

Tipos primitivos e tipos derivados

- Os valores de um **tipo primitivo** são indivisíveis, e o tipo não depende de outros tipos. Exemplos: tipos numéricos e tipo dos caracteres.
- Os valores de um **tipo derivado** são estruturas formadas por valores mais simples, e o tipo é expresso em função de outros tipos. Exemplos: listas de inteiros, matriz de números reais, registro de um usuário da biblioteca.

Números inteiros I

- Números inteiros com sinal:
  - **char** ou **signed char**
  - **short** ou **signed short**
  - **int** ou **signed int** ou **signed**
  - **long** ou **signed long**
- Na representação de complemento de 2 com n bits:

número de valores possíveis	2^n
menor valor	-2^{n-1}
maior valor	$2^{n-1}-1$

- Por exemplo, numa representação de complemento de 2 com 32 bits:

número de valores possíveis	4.294.967.296
menor valor	$-2^{31} = -2.147.483.648$
maior valor	$2^{31}-1 = 2.147.483.647$

Números inteiros II

- É garantido que:
  - número mínimo de bits e maior valor:

tipo	número mínimo de bits	maior valor que é garantido
char	8	127
short	16	32.767
int	16	32.767
long	32	2.147.483.647

  - tamanho(**char**) $\leq$ tamanho(**short**) $\leq$ tamanho(**int**) $\leq$ tamanho**(long**)

Números inteiros III

- Números inteiros sem sinal:
  - **unsiged char**
  - **unsiged short**
  - **unsiged int** ou **unsigned**
  - **unsiged long**
- Utiliza o mesmo número de bits que o inteiro com sinal correspondente.
- Na representação com n bits:

número de valores possíveis	2^n
menor valor	0
maior valor	$2^n - 1$

- Por exemplo, numa representação com 32 bits:

número de valores possíveis	4.294.967.296
menor valor	0
maior valor	$2^{32} - 1 = 4.294.967.295$

Números inteiros IV

- Como escolher o tipo?
  - Normalmente use `int`.
  - Se precisar de valores grandes, use **long**.
  - Se precisar economizar espaço na memória, use **short**.
  - Se valores negativos não são importantes, use o tipo sem sinal correspondente.
  - Evite usar os tipos **char** para representar números inteiros, pois costumam ser problemáticos.
  - Se precisar de inteiros maiores que os fornecidos pelos tipos básicos, use uma biblioteca de *inteiros de precisão arbitária*.

Números inteiros V

- **Literais** expressam diretamente um valor.
- Sistema de numeração:
  - **Decimal**: sequência de um ou mais dígitos decimais que não começa com zero.
  - **Octal**: sequência de um ou mais dígitos octais começando com 0.
  - **Hexadecimal**: sequência de um ou mais dígitos hexadecimais precedida de 0x.
- Literais **sem sinal**: acrescenta-se o sufixo u ou U.
- Literais **long**: acrescenta-se o sufixo l ou L.
- Exemplos:

decimais	2011, 452, 324892
octais	0, 0632, 0234, 03100
hexadecimais	0x324, 0xA3, 0x400FBC01
sem sinal	2011u, 0234u, 0x420Bu
longas	2011u, 0234u, 0x420Bu
longas sem sinal	23423420ul, 02200174ul, 0x10A20Bul

Números em ponto flutuante I

- Podem ser representados na memória por um bit de sinal, seguindo de uma mantissa (algarismos significativos), seguidos por um expoente de uma potência de 2:

$$(-1)^{\text{sinal}} \times \text{mantissa} \times 2^{\text{expoente}}$$

Exemplo:

$$0.5 = (-1)^{0} \times 1 \times 2^{-1}$$

- São usados como aproximações para os números reais.
- Existem três tipos que podem diferir na precisão (quantidade de algarismos significativos) e na faixa (expoente da potência de 2).
  - **float**
  - **double**
  - **long double**

Números em ponto flutuante II

- **Literal**: é formado por:
  - uma parte inteira
  - um ponto decimal
  - uma parte fracionária
  - uma parte de expoente
  - um sufixo
- as partes inteira e fracionária são formadas por dígitos decimais
- a parte de expoente é formada por e ou E seguido de um sinal opcional (+ ou −), seguido de um ou mais dígitos decimais
- a parte inteira ou a parte integral podem ser omitidas, mas não ambas
- o ponto decimal ou a parte de expoente podem ser omitidas, mas não ambas
- os sufixos f e F indicam o tipo **float**
- os sufixos l e L indicam o tipo **long double**
- se nenhum sufixo é especificado, o literal é do tipo **double**

Números em ponto flutuante III

- Exemplos:
  `23.898e-7`, `43.736`, `32301.`, `.56`, `34E+87`, `0.612e102`

Caracteres I

- O tipo **char** é utilizado para representar os caracteres: letras, dígitos decimais, sinais de pontuação e caracteres de controle.
- **char** é um tipo numérico correspondente ao código do caracter, porém deve-se evitar utilizá-lo simplesmente como um *inteiro muito pequeno*.
- Normalmente utiliza-se a tabela ASCII estendida para codificação dos caracteres.

Caracteres II

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Caracteres III

- **Literais**: Escreve-se o caracter entre apóstrofos (também chamados de aspas simples): `'A'`, `'+'`, `'!'`, `'#'`
- **Sequências de escape**:
  - forma especial de indicar o caracter
  - começam com \

`'\n '`	nova linha
`'\r '`	retorno de carro
`'\t '`	tabulação horizontal
`'\''`	o caracter '
`'\"'`	o caracter "
`'\\'`	o caracter \
`'\ddd '`	o caracter cujo código é ddd, sendo ddd uma sequência de três dígitos decimais

Cadeias de caracteres

- Uma **string** (ou cadeia de caracteres) é uma sequêcia de caracteres e pode ser usada como uma representação de um texto.
- Uma string é representada na memória como uma sequêcia de valores do tipo **char**.
- O final da sequência é indicada pelo **caracter nulo** `'\0'`.
- O tipo das strings é um tipo especial chamado **ponteiro para caracteres**, e é escrito como **char***.
- Ponteiros serão estudados no decorrer do curso.
- Um **literal string** é escrito colocando a sequêcia de caracteres entre aspas.
- Exemplos: `"abc"`, `"ana paula"`, `"bom dia\nBrazil"`, `"3+4 = 7"`

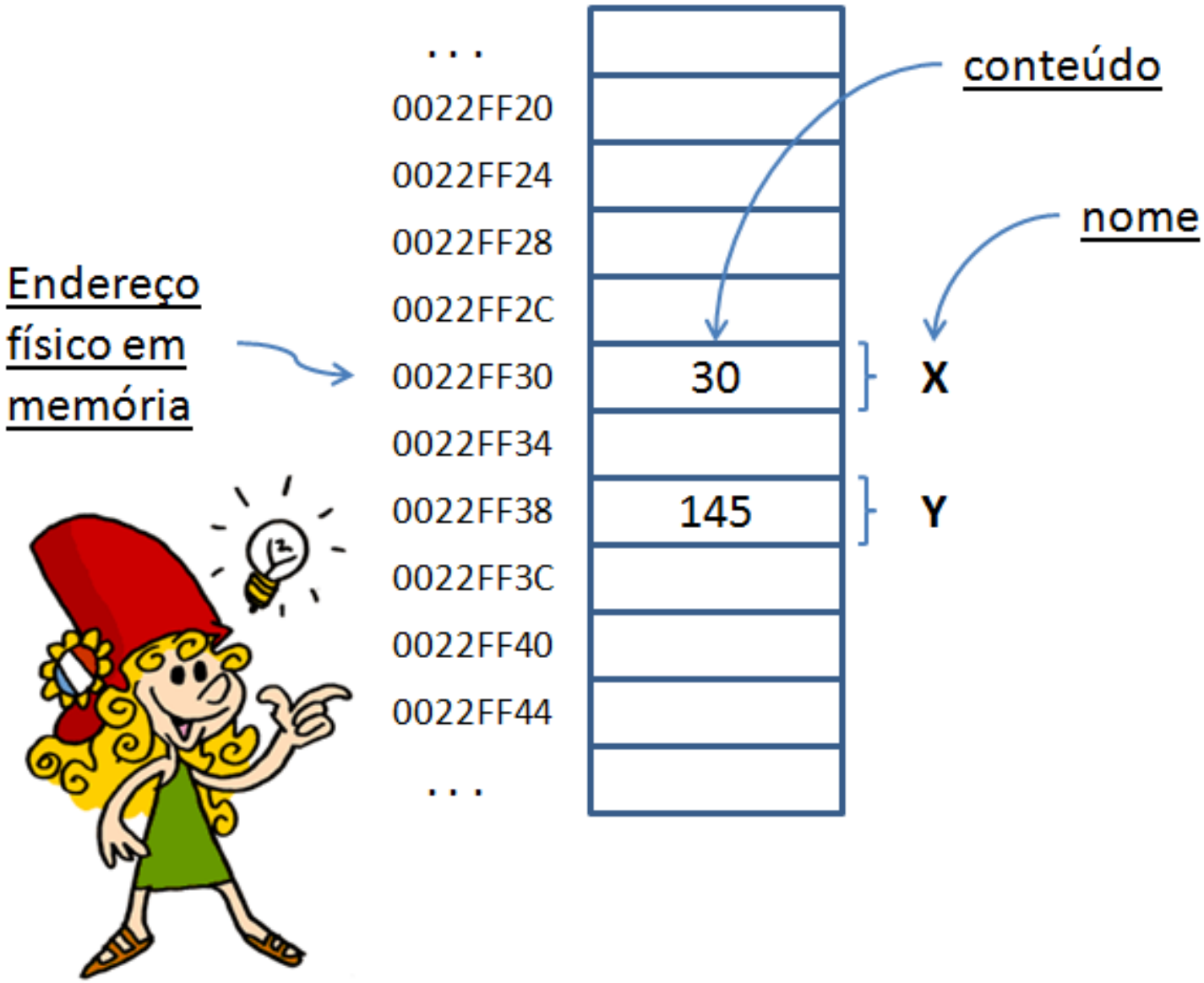
Variáveis I

- VARIÁVEL

Variáveis II

- **Variável** é uma **posição de memória**, e usada para **guardar um valor**.
- Pode ser **identificada** através de um **endereço** ou através de um **nome**.
- É caracterizada por um **tipo**, que define como a sequência de bits armazenada na variável é interpretada.

Variáveis III



Variáveis IV

- O programador usa variáveis nos algoritmos visando atingir os resultados esperados.

Variáveis V

- **Escopo de uma variável**
  - É o segmento de programa em que a variável pode ser usada.
  - Começa a partir da declaração da variável.
  - Termina no menor bloco contendo a declaração da variável.
  - pode ser um bloco, uma rotina ou todo o programa (locais $\times$ globais)

global (todos acessam)

local(só pertence a ele)

Variáveis VI

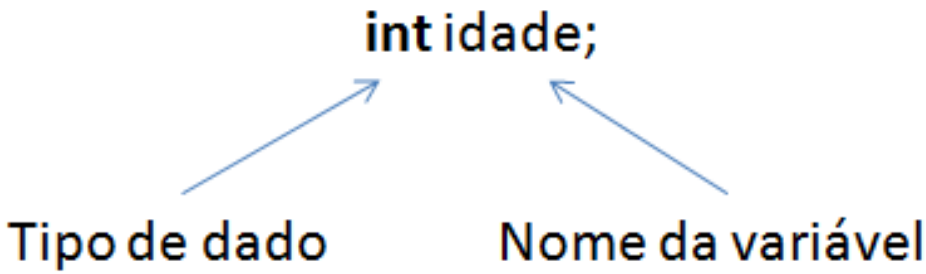
- **Identificador**
  - Nome de variáveis, funções, rótulos e vários outros objetos definidos pelo usuário
- **Constantes**
  - Identificadores que não podem ter seus valores alterados durante a execução do programa

Variáveis VII

- **Alocação estática $\times$ Alocação dinâmica**
  - Alocação estática
    - Reserva de espaço de memoria antes da execução
    - Variáveis locais e globais armazenadas de forma FIXA
    - Necessidade de previsão de tamanho do espaço (ex. vetor)
  - Alocação dinâmica
    - Reserva de espaço de memoria em tempo de execução
    - Necessidade de funções para alocação
    - Uso de ponteiro para a área reservada e crescimento dinâmico

Declaração de variáveis I

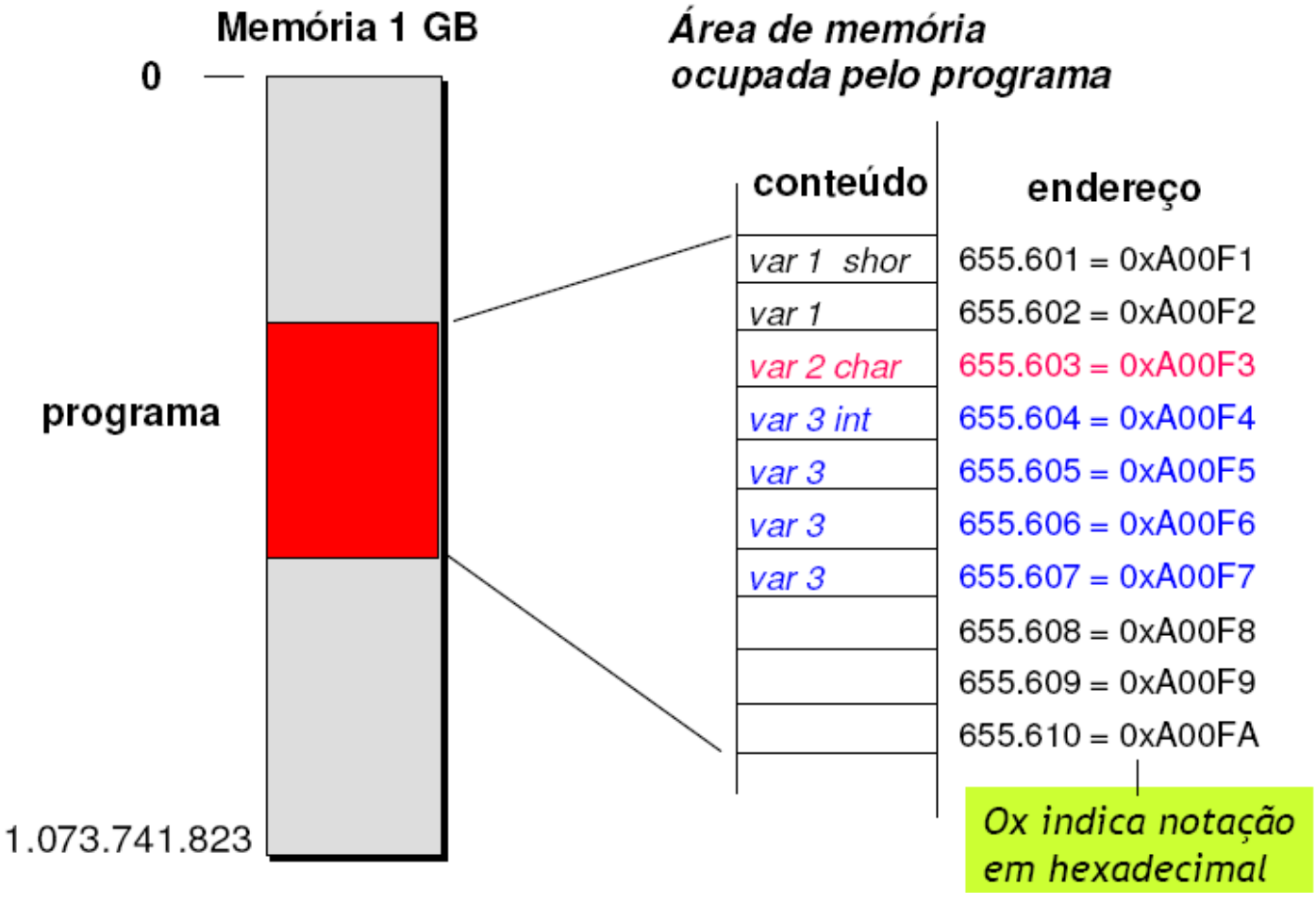
- Uma **declaração de variável** introduz uma nova variável para ser utilizada no programa.
- É necessário especificar o **nome** e o **tipo** da variável.



- O compilador se encarrega de gerar código para **alocar** espaço na memória para a variável. O tamanho em bits do espaço alocado depende do tipo da variável.

Declaração de variáveis II

Espaço de memoria utilizado pelas variáveis



x	bin	dec
0	= 0000 =	0
1	= 0001 =	1
2	= 0010 =	2
3	= 0011 =	3
4	= 0100 =	4
5	= 0101 =	5
6	= 0110 =	6
7	= 0111 =	7
8	= 1000 =	8
9	= 1001 =	9
A	= 1010 =	10
B	= 1011 =	11
C	= 1100 =	12
D	= 1101 =	13
E	= 1110 =	14
F	= 1111 =	15

Declaração de variáveis III

- **Identificador** é um nome dado a algum elemento da linguagem.
- Utilizamos identificadores para nomear variáveis.
- Regras para nomes de variáveis em C:
  - É formado por uma sequência de letras (maiúsculas ou minúsculas), dígitos e sublinhados (_ *underscore*).
  - Deve começar com uma letra ou sublinhado.
- Portanto **nunca** pode começar com um número.
- Não se pode utilizar `{ ( + - / \ ; . , ?` como parte do nome de uma variável.
- C é uma linguagem *case-sensitive*, ou seja, que faz diferença entre nomes com letras maiúsculas e nomes com letras minúsculas: `Peso` e `peso` são identificadores diferentes.
- Costuma-se usar maiúsculas e minúsculas para separar palavras dentro do identificador `PesoDoCarro`.
- Mas pode-se também utilizar sublinhado: `peso_do_carro`.

Declaração de variáveis IV

- Identificadores devem ser únicos no mesmo escopo (não podem haver variáveis com mesmo identificador dentro do mesmo bloco).

Declaração de variáveis V

- As seguintes palavras, chamadas de **palavras-reservadas**, já tem um significado na linguagem C e por esse motivo não podem ser utilizadas como identificadores:

auto	*double*	*int*	*struct*	*break*
enum	*register*	*typedef*	*char*	*extern*
return	*union*	*const*	*float*	*short*
unsigned	*continue*	*for*	*signed*	*void*
default	*goto*	*sizeof*	*volatile*	*do*

Declaração de variáveis VI

- Quais dos nomes a seguir são nomes corretos de variáveis? Se não forem corretos, porque não são?

3ab	a3b	fim	int
`\meu`	`_A`	`n_a_o`	papel-branco
`a*`	`c++`	`*nova_variavel`	

Declaração de variáveis VII

- Uma variável é manipulada no programa através do nome dado na declaração.
- Pode-se *consultar o conteúdo da variável*, bastando para isto utilizar o seu nome.
- Pode-se também *alterar o conteúdo da variável* através de uma expressão de atribuição.

Expressões

- **Expressão** é uma frase na linguagem de programação que pode ser **avaliada** para produzir um **valor**.
- Toda expressão tem um tipo, que é o tipo do seu valor.
- Exemplos:

expressão	valor	tipo
2+3*4-1	13	int
(2.3 - 0.2)*3.0	6.3	double
'N'	'N'	char
"Computador"	"Computador"	char*

Expressões: Literais

- A forma mais simples de expressões são **literais**, que expressam diretamente o valor que se deseja, sem necessidade de cálculo.
- As formas literais foram apresentadas junto com os seus tipos.
- Exemplos:

expressão	valor	tipo
4752	4752	int
077	63	int
0xFFu	255	unsigned int
6783L	6783	long int
346783ul	346783	unsigned long int
0.452F	0.452	float
56.7327e-45	56.7327e-45	double
8.324E-65L	8.324e-65	long double
'z'	'z'	char
"UFOP"	"UFOP"	char*

Expressões: Variáveis

- Uma **variável** é uma expressão cujo tipo é o próprio tipo da variável, e cujo valor é o conteúdo da variável.
- Exemplo: Se `peso` é uma variável do tipo `double` cujo conteúdo é 5.6, então `peso` também é uma expressão do tipo `double` e valor 5.6.

Expressões: operadores

- Um operador permite realizar uma operação com um ou mais operandos (que são expressões).
- Classificação quanto ao número de operandos:

operador	número de operandos	exemplo
unário	1	`- 3`
binário	2	`2 + 3`
ternário	3	`x>5 ? z = x+2 : z = x-2`

- Classificação quanto à posição do operador:

operador	posição	exemplo
prefixo	antes dos operandos	`! encontrou`
infixo	entre os operandos	`5*x`
sufixo	depois dos operandos	

Prioridade dos operadores

- Quando usamos vários operadores em uma determinada expressão, torna-se difíciel decidir quem é o operando de cada operador.
- Exemplo: `2+3*4`
  3 é operando do `+` ou do `*` ?
- Existe uma relação de **prioridade** (ou precedência) dos operadores que permite tomar a decisção. neste caso o `*` tem prioridade maior que o `+`, portanto o 3 é um operando do `*` e não do `+`, e a expressão é avalidada como `2+(3*4)`, resultando em 14.
- Operações cujos operadores tem maior prioridade são realizadas primeiro.

Associatividade dos operadores

- Quando usamos vários operadores de mesma prioridade em uma expressão, torna-se difíciel decidir quem é o operando de cada operador.
- Exemplo: `2-3-4`
  3 é operando do primeiro ou do segundo `-`?
- Um operador pode ser:

associativo à esquerda	as operações realizam-se da esquerda para a direita
associativo à direita	as operações realizam-se da direita para a esquerda
não associativo	não é possível usar o operador em sequência

- Exemplo: O operador `-` é associativo à esquerda, logo a expressão `2-3-4` é avaliada como `(2-3)-4` resultando em `-5`. Se ele fosse associativo à direita, a expressão seria avaliada como `2-(3-4)` resultando no valor `3`.

Expressões: uso de parênteses

- Qualquer expressão pode ser escrita entre parênteses.
- O tipo e o valor da expressão parentetizada são os mesmos da expressão sem parênteses.
- Parênteses são úteis para mudar a ordem de avaliação das operações quando não se deseja usar a prioridade ou associatividade de um operador.
- Exemplos:

expressão	valor
(2+4)	6
2+(3*6-4)/7	4
((5-1)*(3+7)*2)/4	20

Expressões: operadores básicos

Alguns operadores agrupados por ordem decrescente de prioridade

operador	operação	associat.
var = *expr*	atribuição	direita
- *expr*	simétrico	direita
+ *expr*	nenhum	
++ *var*	pre-incremento	
var ++	pós-incremento	
-- *var*	pre-decremento	
var --	pós-decremento	
(*tipo*) *expr*	promoção de tipo	
expr1 * *expr2*	multiplicação	esquerda
expr1 / *expr2*	quociente da divisão	
expr1 % *expr2*	resto da divisão	
expr1 + *expr2*	adição	esquerda
expr1 - *expr2*	subtração	
(*expr*)	parênteses	esquerda

Expressões: observações sobre operadores I

- Se os operandos de uma opeação são de um mesmo tipo, então o resultado também é deste mesmo tipo.
- Assim o valor de `15/2` é `7`, e não `7.5`, pois os dois argumentos são do tipo **`int`**.
- Já o valor da expressão `15.0/2.0` é `7.5`, pois os tipos dos argumentos é **`double`**, levando a um resultado também do tipo **`double`**.
- O operador `\%` calcula o resto da divisão de seus operandos. Exemplo: O valor de `17 \% 3` é `2`.
- O segundo operando dos operandos de divisão `/` e `\%` não pode ser zero.
- Não há nenhum operador para a operação de potenciação.

Expressões: atribuição I

- **Atribuir** um valor de uma expressão a uma variável **significa calcular o valor** daquela expressão e **copiar** aquele valor para uma **determinada posição de memória** representada pela variável.
- A avaliação da expressão de atribuição

```
var = expressão
```

é feita pela avaliação de *expressão*, cujo valor é usado para atualizar o conteúdo da variavel,

- Exemplo:

```
soma = a + b;
```

A variável `soma` recebe o valor da expressão `a+b`.

- O resultado da expressão de atribuição é próprio valor final de *expressão*.

Expressões: atribuição II

- Sempre que se faz uma atribuição, o valor da variável é substituído pelo valor da expressão. Portanto o valor anterior é perdido.

Expressões: atribuição III

- Exemplo:

```
double altura;
altura = 1.72;
altura = 2 * altura;
```

  - A primeira linha declara a variável `altura`, porém não a inicializa. Isto significa que o valor da variável logo após a sua declaração é desconhecido (lixo). Este valor inicial é formado de acordo com a sequência de bits armazenada na posição de memória no momento em que ela é alocada.
  - A segunda linha armazena o valor `1.72` na variável `altura`.
  - A terceira linha calcula o valor da expressão `2 * altura`, que é `3.44`, e o armazena na variável `altura`. O valor anterior (`1.72`) é perdido.

Expressões: atribuição combinada com operação I

- **Incremento**
  - Modificam a variável somando 1 ao seu conteúdo.
  - **Pré-incremento**: `++x`
    - Incrementa a variável primeiro e só depois usa a variável no restante da expressão.
    - Exemplo:

```
int x = 2;
int y = 3 * (++x) - 4;
```

Primeiro a variável x é incrementada (o seu novo valor é 3) e depois a expressão `3 * x - 4` é avaliada, resultando em 5.

Expressões: atribuição combinada com operação II

- **Pós-incremento**: `x++`
  - Primeiro usa a variável no restante da expressão, e só depois incrementa a variável.
  - Exemplo:

```
int x = 2;
int y = 3 * (x++) - 4;
```

Primeiro a expressão `3 * x - 4` é avaliada, resultando em 2, e somente depois a variável `x` é incrementada (o seu novo valor é 3).

Expressões: atribuição combinada com operação III

- **Decremento**
  - Modificam a variável subtraindo 1 do seu conteúdo.
  - **Pré-decremento**: `--x`
    - Decrementa a variável primeiro e só depois usa a variável no restante da expressão.
    - Exemplo:

```
int x = 2;
int y = 3 * (--x) - 4;
```

Primeiro a variável x é decrementada (o seu novo valor é 1) e depois a expressão `3 * x - 4` é avaliada, resultando em -1.

Expressões: atribuição combinada com operação IV

- **Pós-decremento**: `x--`
  - Primeiro usa a variável no restante da expressão, e só depois decrementa a variável.
  - Exemplo:

```
int x = 2;
int y = 3 * (x--) - 4;
```

Primeiro a expressão `3 * x - 4` é avaliada, resultando em 2, e somente depois a variável `x` é incrementada (o seu novo valor é 1).

Expressões: atribuição combinada com operação V

- **Operação combinada com atribuição**: `+=` `-=` `*=` `/=` `%=`
  - Realizam a operação indicada com a variável que aparece do lado esquerdo e a expressão que aparece do lado direito.
  - O resultado da operação é armazenado na variável que aparece do lado esquerdo.
  - Exemplo:

```
int x = 7;
x += 2; // x = x + 2
x -= 2; // x = x - 2
x *= 2; // x = x * 2
x /= 2; // x = x / 2
x %= 2; // x = x % 2
```

Expressões: atribuição combinada com operação VI

- Exemplo: qual é valor exibido na tela pelo programa?

```c
/* Operadores */
int main(void)
{
    int i = 10, j = 20;
    i = i + 1;
    i++;
    j -= 5;
    prinft("i + j = %d", i+j);
    return 0;
}
```

Expressões: promoção implícita de tipos

- Quando os operandos de algumas operações não são compatíveis com o tipo esperado, é possível que o compilador automaticamente gere código para converter o operando para o tipo adequado.
- Os tipos numéricos podem ser convertidos implicitamente para tipos numéricos com tamanhos maiores.
- Exemplo:

```
double x;
x = 23 + 5.6;
```

Na expressão `23 + 5.6` os operandos `23` e `5.6` são do tipo **`int`** e **`double`** respectivamente. Para que a operação possa ser realizada, é necessário que os dois operandos sejam do mesmo tipo. Logo o operando do tipo **`int`** é automaticamente promovido ao tipo **`double`**, e o resultado final é `28.6`.

Expressões: promoção explícita de tipos

- O programador pode realizar uma promoção explícita de tipos através das expressões de promoção de tipo.

```
(tipo desejado) expressão
```

- Exemplo: o valor da expressão

```
(int)(23.4/2.0 + 2.2) / 2
```

é `6`.

Expressões: chamadas de função I

- *Funções* são partes de um programa que permite calcular um valor segunda uma determinada regra, de maneira semelhante ao que se faz com as funções matemáticas.
- No decorrer do curso aprenderemos a definir nossas próprias funções.
- A **biblioteca** do C oferece uma grande quantidade de funções que podem ser usadas nos nossos programas.
- Quando uma função é definida, especifica-se o tipo dos argumentos e também o tipo do resultado.
- Para usarmos uma função é preciso informar ao compilador como a função pode ser usada. Isto é feito informando o tipo dos argumentos e o tipo do resultado.
- Se a função foi definida em uma biblioteca, deve-se incluir o **arquivo de cabeçalho** desta biblioteca.
- O arquivo de cabeçalho contém informações sobre como usar as funções ali definidas.

Expressões: chamadas de função II

- A chamada de função é uma expressão da forma

```
nome (expressão1, ..., expressãon)
```

  onde *nome* é o nome da função, e $expressão_1, \ldots, expressão_n$ são os argumentos que serão utilizados para se obter o resultado segundo a regra implementada pela função.
- Os tipos dos argumentos devem ser compatíveis com os tipos declarados na função.
- O resultado será do tipo especificado na função.
- Os argumentos são avaliados da esquerda para a direita, e em seguida a regra que define a função é avaliada para se obter o resultado.

Expressões: chamadas de função III

- **Potenciação**: a função

```
double pow(double x, double y);
```

  da biblioteca **math.h** calcula a potência de dois números. O primeiro argumento é a base e o segundo é o expoente. Tanto os argumentos quanto o resultado são do tipo **`double`**.

- Exemplo: o valor da expressão `pow(2.1, 3.0)` é `9.261`.

Expressões: chamadas de função IV

- **Raiz quadrada**: a função

```
double sqrt(double x);
```

também da biblioteca **math.h** calcula a raiz quadrada de um número. O único argumento é o radicando. Tanto o argumento quanto o resultado são do tipo **`double`**.

- Exemplo: o valor da expressão `sqrt(0.81)` é `0.9`.

Expressões: tamanho de tipo I

- A expressão `sizeof`(tipo) retorna o tamanho, em bytes, de um determinado tipo. (Um byte corresponde a 8 bits).
- Exemplo: o valor da expressão `sizeof(int)` no meu computador é 4.

FIM

Créditos:
Baseado no material preparado pelo
Prof. Guillermo Cámara-Chávez.