Introdução à Linguagem C

Prof. Márcio Castro

marcio.castro@ufsc.br

Na aula de hoje

- Introdução
- Construções básicas da linguagem
- Funções
- Ponteiros
- Arrays e matrizes
- Strings
- Estruturas

INTRODUÇÃO

Introdução

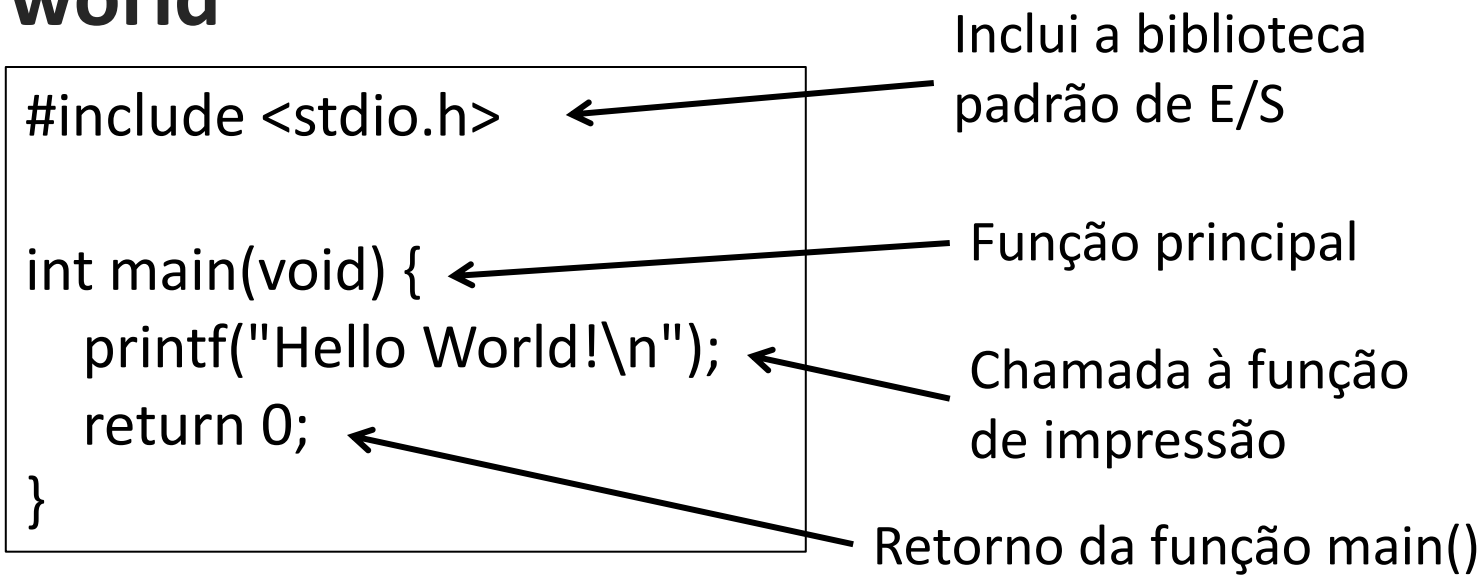
- **Características gerais da linguagem C**
  - Linguagem de programação de **propósito geral**
  - Usada em inúmeros domínios de aplicação
    - sistemas operacionais, computação numérica, aplicações gráficas, sistemas embarcados, ...
  - Apenas 32 palavras-chave (*keywords*)
  - Diversas facilidades para **programação estruturada**
  - Possibilidade de **manipulação de *bytes* e endereços**

Introdução

- **Funções**
  - Programas em C são organizados de maneira estruturada através de **rotinas** (**funções**)
  - A linguagem possui uma biblioteca padrão (*standard library - stdio.h*) que oferece uma coleção de operações úteis
  - Exemplos
    - printf(): permite imprimir algo na tela
    - scanf(): permite ler algo do teclado

Introdução

- **Um programa em C**
  - Consiste em **funções** e **variáveis**
  - **Funções:** especificam as operações a serem executadas
  - **Variáveis:** armazenam valores durante a computação
- **Hello world**



Introdução

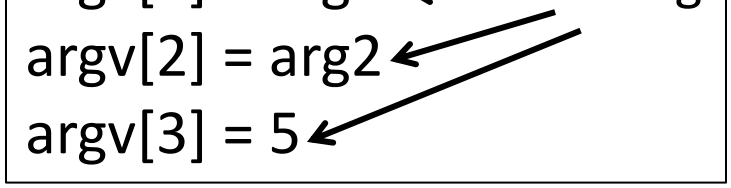
- **A função main()**
  - Todos os programas em C tem como "ponto de entrada" a função **main()**
- **Duas formas de declarar a função main()**
  - **int main(void)**: sem argumentos de entrada
  - **int main(int argc, char *argv[])**: receber os argumentos de entrada pela linha de comando
    - **argc:** quantidade de argumentos informados na linha de comando
    - **argv[]:** vetor de *strings*. Cada string representa um valor passado à função main()

Introdução

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc == 1) printf("Sem argumentos de entrada.\n");
  else {
    int i;
    for(i = 1; i < argc; i++)
      printf("argv[%d] = %s\n", i, argv[i]);
  }
  return 0;
}
```

```
$ ./prog1
Sem argumentos de entrada.
```

```
$ ./prog1 arg1 arg2 5
argv[1] = arg1
argv[2] = arg2
argv[3] = 5
```



Introdução

Tipo	Tamanho
char	Usualmente 8 bits (1 byte)
int	Usualmente o tamanho de uma palavra definida pelo hardware/SO (e.g., 16, 32 ou 64 bits)
short int	Pelo menos 16 bits
long int	Pelo menos 32 bits
unsigned int	Idem int, mas sem sinal
float	Usualmente 32 bits
double	Usualmente 64 bits
long double	Pelo menos 64 bits

- **sizeof(*tipo*):** retorna o tamanho (em bytes) do *tipo* de dado ou *estrutura de dados*
  - **Exemplo:** int tamanho_int = sizeof(int);

Introdução

- **Constantes simbólicas**
  - Representam valores constantes

```
#define BLOCK_SIZE 100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define HELLO "Hello World\n"
#define EXP 2.7183
```

  - Toda vez que uma constante é utilizada no código, ela é substituída pelo valor definido na sua declaração (#define)

```
printf(HELLO);
printf("Valor de block size = %d\n", BLOCK_SIZE);
```

Introdução

- **Imprimindo na saída padrão (terminal)**
  - int printf(const char *format, arg1, arg2, ...)

String contendo o texto de saída e operadores de formatação

Variáveis a serem impressas separadas por vígulas

Tipo da variável	Formatação
char	%c
int	%d
unsigned int	%u
unsigned long int	%lu
float	%f
double	%g
char * (string)	%s

```
int a = 1234;
float b = 3.45;
char c = 'f';
const char texto[12] = "Hello World";

printf("a = %d\n", a);
printf("B vale %f e C vale %c\n", b, c);
printf("%s", texto);
```

```
a = 1234
B vale 3.450000 e C vale f
Hello World
```

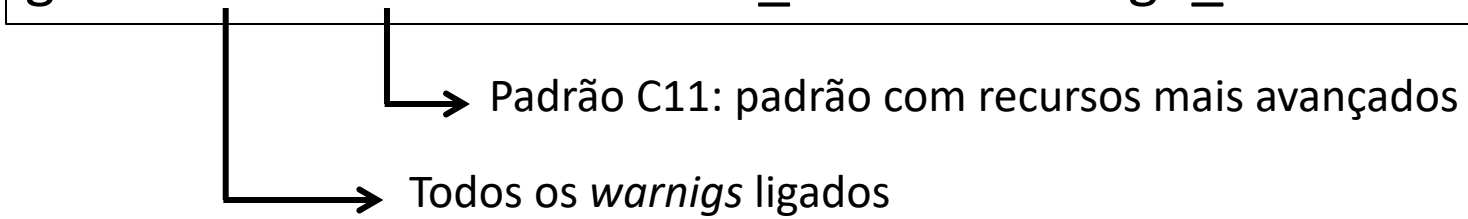
Introdução

- **Compilação**
  - GNU Compiler Collection (gcc)
- **Sintaxe básica:**

```
gcc -o <nome_binário> codigo_fonte.c
```

- **Argumentos recomendados:**

```
gcc -Wall -std=c11 -o <nome_binário> codigo_fonte.c
```



CONSTRUÇÕES BÁSICAS

Construções básicas

- **Construções básicas: if**

```
int i = 10;

if (i > 10) {
    printf("i maior que 10.\n");
    printf(”i ainda maior que 10.\n");
}

if (i <= 10) print ("i menor ou igual a 10.\n");
```

```
if (1) {
   printf("Sempre sera impresso!\n");
}
if (0) {
   printf("Nunca sera impresso!\n");
}
```

Construções básicas

- **Construções básicas: while**

```
i = 0;

// executado 10 vezes
while (i < 10) {
    printf("i vale %d!\n", i);
    i++;
}
```

Construções básicas

- **Construções básicas: while e do-while**

```
i = 0;

// executado 10 vezes
while (i < 10) {
    printf("i vale %d!\n", i);
    i++;
}
```

```
i = 10;

/* executado uma vez, pois a condição só é avaliada após
   o fim do bloco */
do {
    printf("do-while: i vale %d\n", i);
    i++;
} while (i < 10);
```

Construções básicas

- **Construções básicas: for**

```
// executado 10 vezes
for (i = 0; i < 10; i++) {
    printf("i vale %d\n");
}
```

```
for(;;) {
    printf("Sera impresso eternamente\n" );
}
```

Construções básicas

- **Construções básicas: switch**

```
switch(a) {
    case 0:
        printf("a vale zero!\n");
        break;
    case 1:
        printf("a vale 1!\n");
        break;
    default:
         printf("nem zero nem 1\n");
}
```

FUNÇÕES

Funções

- **Programas em C são estruturados com uso de funções**
  - Cada função deve ter um **comportamento bem definido**
  - Pode ter (ou não) **argumentos de entrada**
  - Pode **retornar (ou não)** um valor
    - Quando a função não retorna nenhum valor ela também é chamada de **procedimento**

Funções

- **Funções definidas pelo usuário**

```
int plus_one(int n) /* definição da função */
{
   return n + 1;
}

int main(void)
{
   int i = 10, j;
   j = plus_one(i); /* chamada */
   printf("i + 1 = %d\n", j);
   return 0;
}
```

Funções

- **Funções definidas pelo usuário**

```
int plus_one(int n) /* definição da função */
{
   return n + 1;
}
```

```
int a = 5, b = 10;

plus_one(a);              /* o tipo de a é int */
plus_one(10);             /* o tipo de 10 é int */
plus_one(1+10);           /* o tipo de toda a expressão é int */
plus_one(a+10);           /* o tipo de toda a expressão é int */
plus_one(a+b);            /* o tipo de toda a expressão é int */
plus_one(plus_one(a)); /* o valor de retorno é int  */
```

Funções

- **Protótipos de funções**
  - Funções precisam ser **declaradas antes de serem utilizadas no código**
  - **Protótipos** permitem que o compilador saiba da existência das funções
    - A implementação das funções pode então aparecer em qualquer parte do código
  - **Boa prática:** colocar os protótipos em arquivos ".h"

```
int foo(void); /* protótipo da função */

int main(void) {
   int i = foo();
   return 0;
}

int foo(void) { /* essa é a implementação da função */
   return 3490;
}
```

Funções

- **Passagem por valor**
  - Por padrão, funções operam sobre **cópias dos valores passados como argumentos**
  - No caso de **passagem por valor**, o valor da variável passada como argumento à função permanece **inalterado** após o fim da execução da função

```
void increment(int i) {
    i++;
}
int main(void) {
    int i = 10;
    increment(i); //i será igual à 10 depois da chamada
    return 0;
}
```

Funções

- **Escopo das variáveis**
  - Variáveis declaradas **dentro de funções** são ditas “privadas” e **só existem dentro do escopo da função**
  - Variáveis declaradas **fora de qualquer função** são ditas “globais” e **podem ser acessadas e modificadas em qualquer função**

Funções

- **Escopo das variáveis**

```
#include <stdio.h>

/* Variável global, pois está declarada em "escopo global", ou seja, não está
declarada dentro de nenhum bloco específico ou função */
int g = 10;

void afunc(int x) {
   g = x; /* seta a variável global com o valor de x */
}

int main(void) {
   afunc(20);          /* essa função seta g para 20 */
   printf("%d\n", g); /* o valor impresso será 20 */
   return 0;
}
```

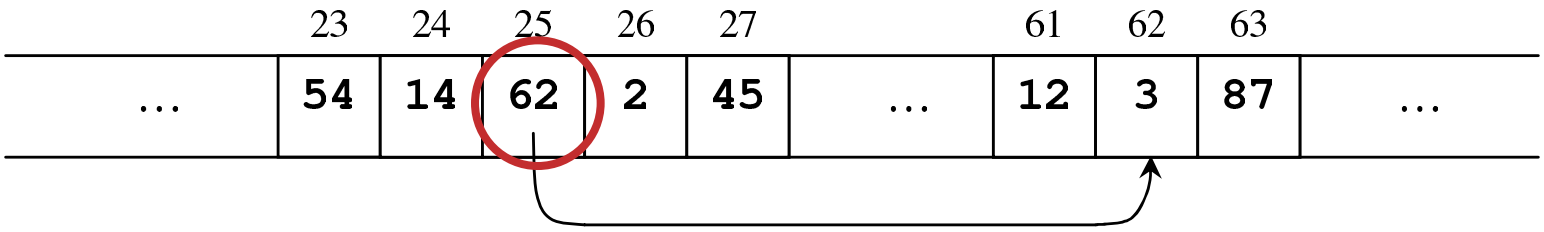
PONTEIROS

Ponteiros

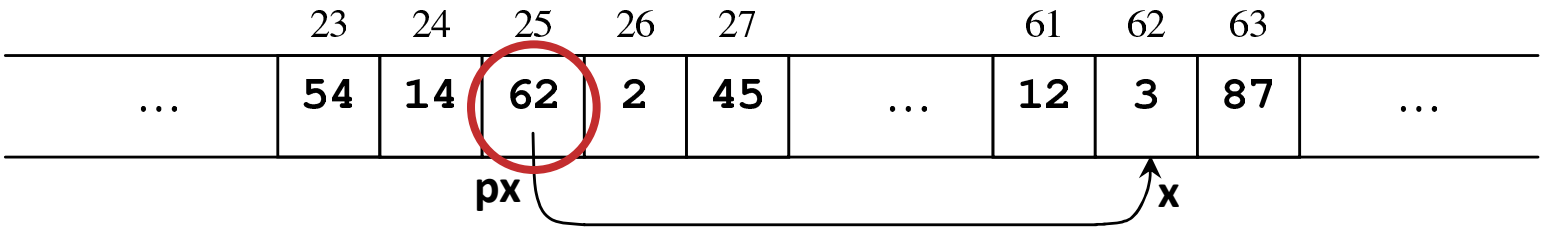
- **Permitem manipular diretamente a memória do computador**
  - *Feature* mais **poderosa** e mais **perigosa** da linguagem C
- **Para entender ponteiros é necessário entender como a memória é organizada**
  - A memória é um ***array* de células numeradas e consecutivas**
  - O número de cada célula do *array* (memória) representa o seu **endereço**
  - Cada célula consiste em um certo número de *bits* que armazenam dados (**valores**)

Ponteiros

- **Alocação de dados em memória**
  - Para cada variável definida no código é alocado uma **porção de memória capaz de armazenar o valor da variável**
  - Logo, toda variável possui um **endereço** (local na memória onde a variável foi alocada) e um **valor** (dado armazenado na memória)
  - **Ponteiro:** é uma **variável** cujo o **valor** corresponde ao **endereço de uma outra variável**



Ponteiros



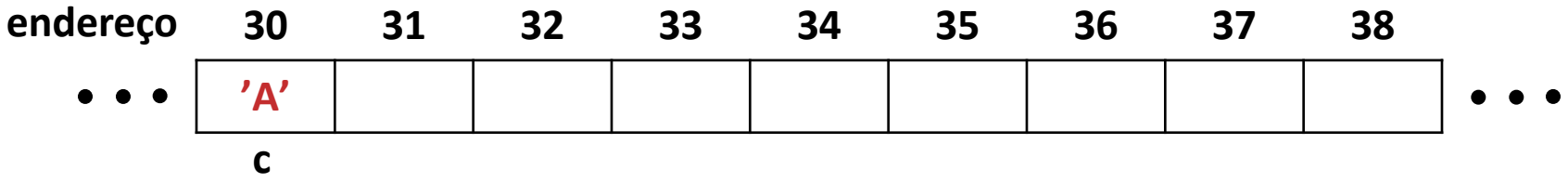

```
/* assuma que a variável x foi alocada
no endereço 62 da mem. */
char x = 3;

/* assuma que a variável px foi alocada
no endereço 25 da mem. */
char *px;

/* px aponta para o endereço de x */
px = &x;

/* x2 recebe o valor 3 */
char x2  = *px;
```

- **Ponteiros**
  - São declarados com * antes do nome da variável
  - Endereço de uma variável é obtido através do operador **&** (**referenciar**)
  - Valor armazenado no endereço apontado pelo ponteiro é obtido através do operador * (**dereferenciar**)

Ponteiros

Exemplos de operações com ponteiros

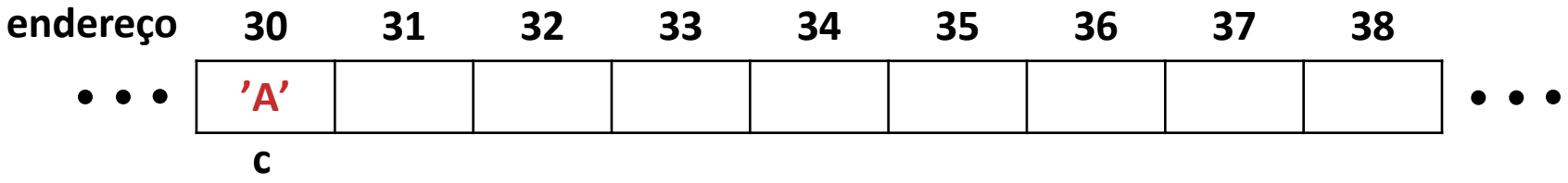
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

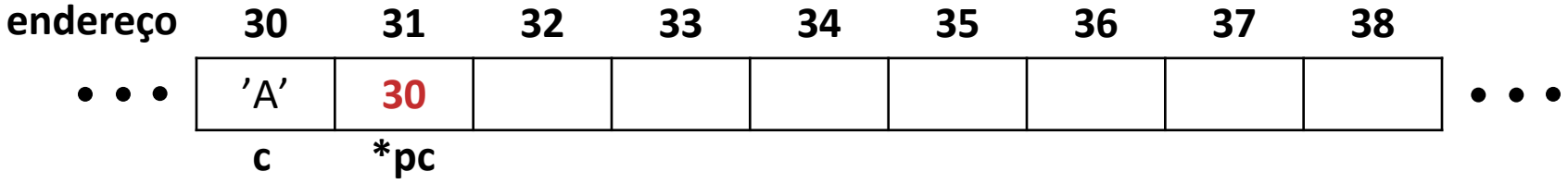
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

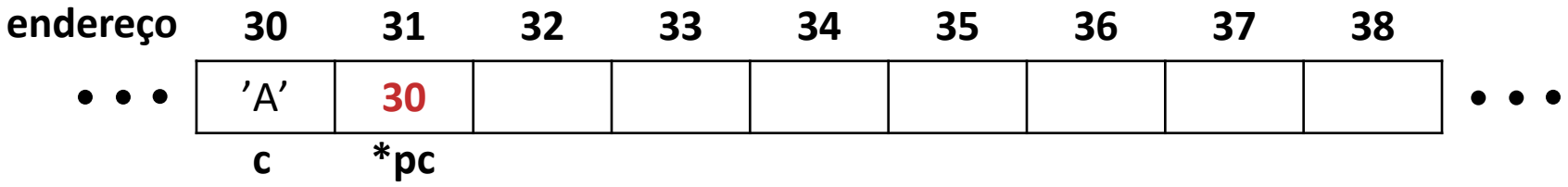
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

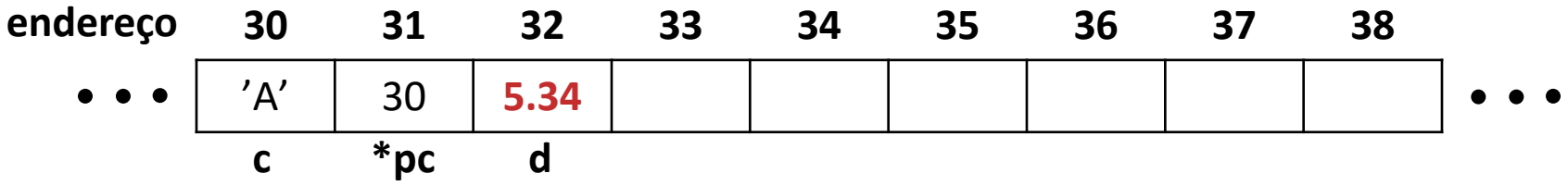
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

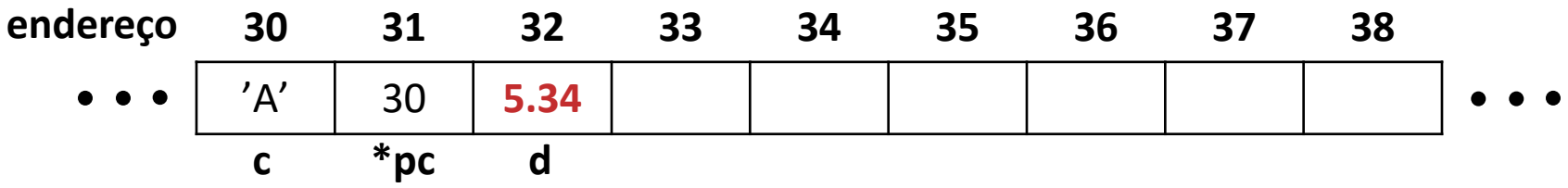
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

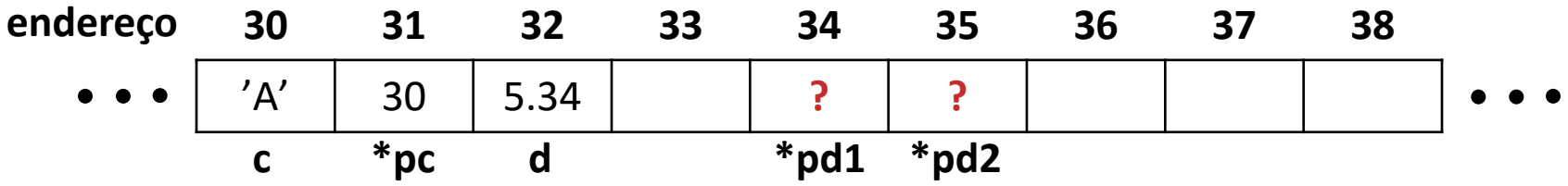
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

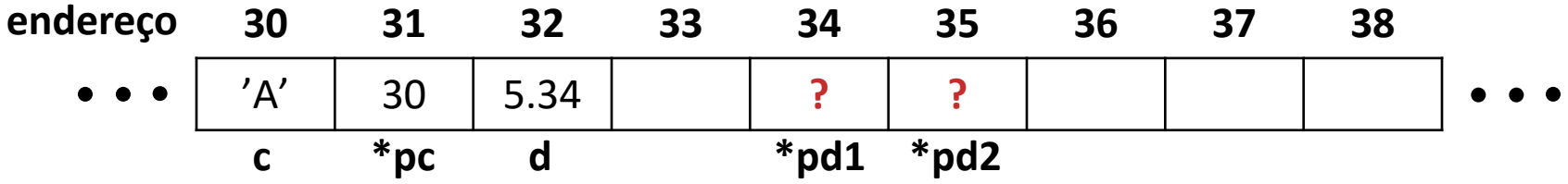
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

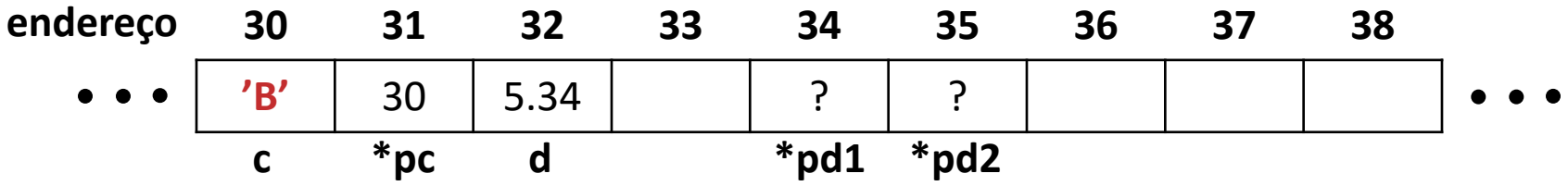
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

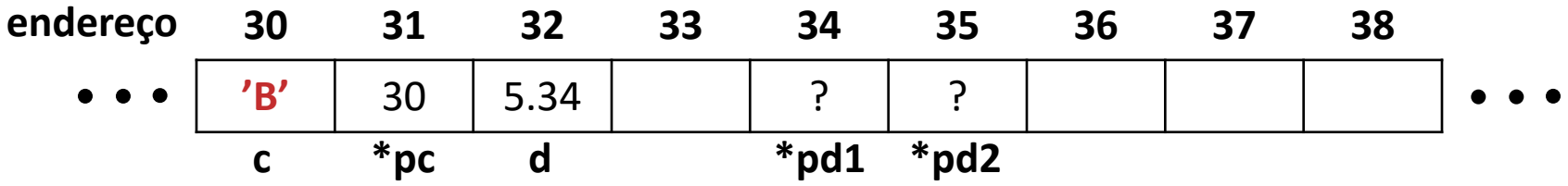
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

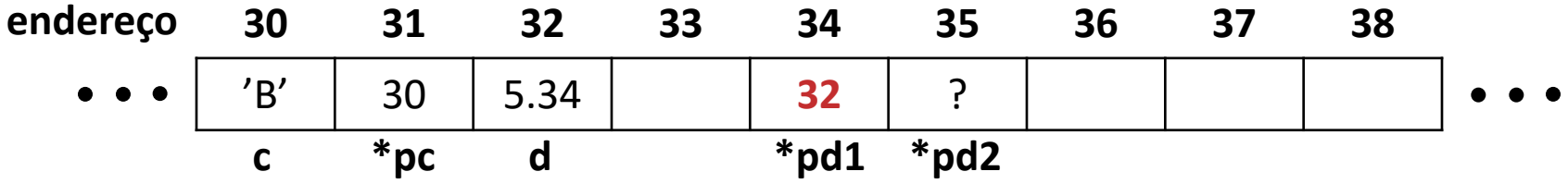
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1=&d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

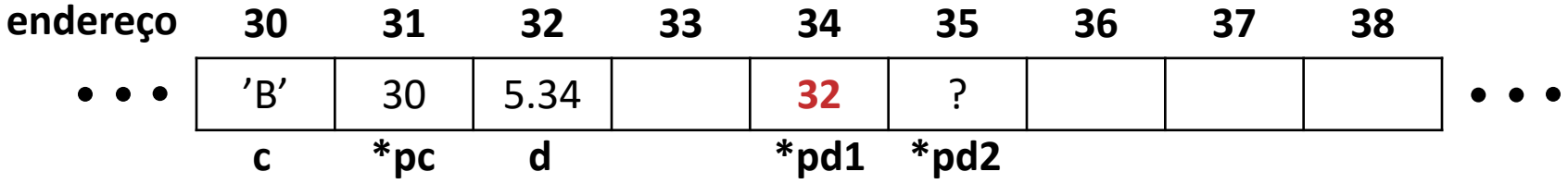
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1 = &d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

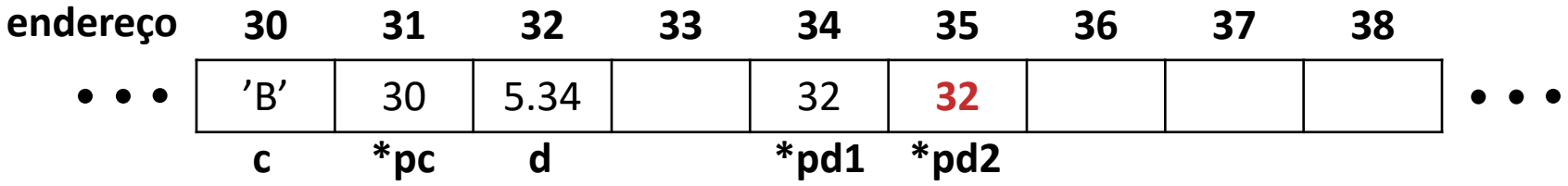
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1 = &d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

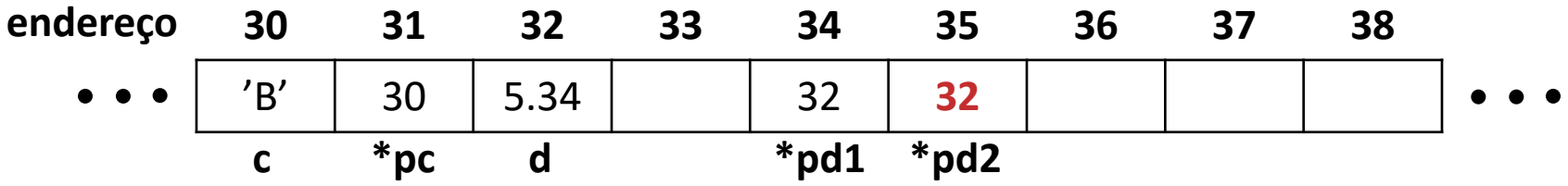
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1 = &d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

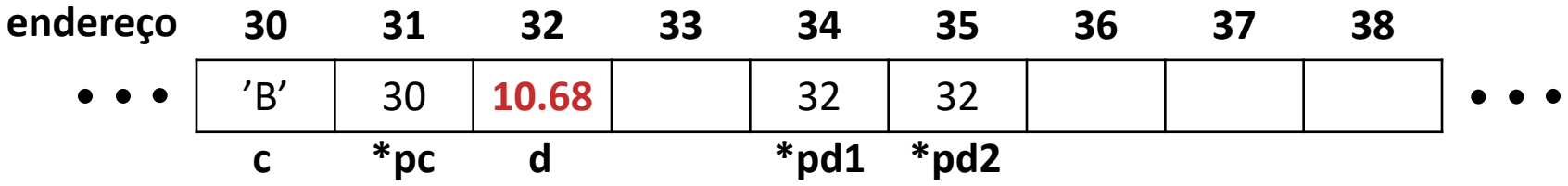
```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1 = &d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

Exemplos de operações com ponteiros

```
1. char c = 'A';
2. char *pc = &c;
3. double d = 5.34;
4. double *pd1, *pd2;
5. *pc = 'B';
6. pd1 = &d;
7. pd2 = pd1;
8. *pd1 = *pd2 * 2.0;
```



Ponteiros

- Quando uma variável é passada por parâmetro a uma função ocorre uma cópia (**passagem por valor**)
  - Modificações dentro da função **não alteram o valor da variável após o retorno da função**

```
void swap(int x, int y) { /* x e y são cópias de a e b. */
   int tmp = x;
   x = y;     /* essa operação não afeta a variável a. */
   y = tmp;  /* essa operação não afeta a variável b. */
}

int main(void) {
   int a = 2, b = 5;
   swap(a, b); /* passagem por valor das variáveis a e b */
}
```

Ponteiros

- **Por que ponteiros são muito úteis?**
  - Ponteiros permitem realizar passagem de parâmetros para funções por **referência**
  - Ao passar o **endereço** de uma variável, ocorrerá uma **cópia do endereço**
  - Logo, qualquer modificação dentro da função irá modificar o valor da variável original

```
void swap(int *px, int *py) { /* x e y apontam para a e b. */
   int tmp = *px; /* equivale à: tmp = a */
   *px = *py;     /* o valor de px é o endereço de a. Equivale à: a = b */
   *py = tmp;     /* o valor de py é o endereço de b. Equivale à: b = tmp*/
}

int main(void) {
   int a = 2, b = 5;
   swap(&a, &b); /* passagem por valor das variáveis a e b */
}
```

Ponteiros

- **Por que ponteiros são muito úteis?**
  - Imagine que o parâmetro passado à função não seja uma variável simples, mas sim **um enorme *array* ou matriz**
  - A passagem por referência permite passar somente o **endereço inicial do *array* ou da matriz**, não sendo necessário realizar a cópia de todos os dados
  - **Grande ganho de desempenho!**

Ponteiros

- **Por que ponteiros são muito úteis?**
  - Permitem "retornar" diversos valores em uma função
  - A passagem por referência permite que diversas variáveis passadas como argumento para uma função possam ser alteradas por elas

```
int vetor[10] = {1,2,3,4,5,6,7,8,9,10}; //array contendo 10 posições
void max_min(int *max, int *min) {
   int i; *min = *max = vetor[0];
   for(i=1; i < 10; i++) if(vetor[i] < *min) *min = vetor[i];
   for(i=1; i < 10; i++) if(vetor[i] > *max) *max = vetor[i];
}
int main(void) {
   int max, min;
   max_min(&max, &min); /* max e min servirão como "retorno" */
   return 0;
}
```

ARRAYS E MATRIZES

Arrays e matrizes

- ***Array***
  - Coleção **linear de dados**
  - "Pedaço" **contíguo de memória** que armazena um certo número de dados de um mesmo tipo

índice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
vetor	32	3	2	4	15	27	66	81

```
int main(void) {
   int vetor[8];    /* declaração de um vetor de 8 posições (0 à 7) */
   vetor[0] = 32; /* modificação do valor armazenado na posição 0 */
   vetor[1] = 3;   /* modificação do valor armazenado na posição 1 */
   ...
   int i;
   for(i = 0; i < 8; i++) printf("vetor[%d] = %d\n", i, vetor[i]);
}
```

Arrays e matrizes

- **Entendendo como o C lida com *arrays:***
  - Referência ao *array* pelo seu nome sem o uso de "[]" retorna o **endereço do primeiro elemento**
    - **Ex.:** "vetor" equivale a "&vetor[0]")
  - Os "[]" permitem acessar um elemento a partir do endereço inicial do *array* de forma transparente
    - vetor[3] **equivale à** *(vetor + 3) → **end. inicial + 3 ints**

```
int main(void) {
   int vetor[8]; /* declaração de um vetor de 8 posições (0 à 7) */
    int i;
   ...
   for(i = 0; i < 8; i++) printf("vetor[%d] = %d\n", i, vetor[i]);
   /* equivale à : */
   for(i = 0; i < 8; i++) printf("vetor[%d] = %d\n", i, *(vetor + i));
}
```

Arrays e matrizes

- **Passagem de *array* por parâmetro**
  - A passagem de *arrays* por parâmetro é sempre feita **por referência**
  - Logo, é feito somente uma **cópia do endereço inicial do *array*** (e não de todos os elementos)

```
void init_array(int a[], int count) {
  int i;
  for(i = 0; i < count; i++)
    a[i] = i * 10; /* inicializa os valores do array */
}
int main(void) {
  int mydata[10];
  init_array(mydata, 10);  /* note a ausência da notação "[]" e "*" */
  return 0;
}
```

Arrays e matrizes

- **Matriz**
  - É uma **junção contígua de *arrays***, onde cada *array* corresponde a uma linha da matriz

```
int main(void) {
  int m[2][2] = {{3,4},{5,6}};
  return 0;
}
```

m[2][2] =

	col 0	**col 1**
linha 0	m[0][0]	m[0][1]
linha 1	m[1][0]	m[1][1]

- **Como é armazenado na memória RAM:**

	m[0][0]	**m[0][1]**	**m[1][0]**	**m[1][1]**
m	3	4	5	6

Arrays e matrizes

- **Passagem de *matriz* por parâmetro**
  - A passagem de *matrizes* por parâmetro é sempre feita **por referência**
  - Logo, é feito somente uma **cópia do endereço inicial da matriz** (e não de todos os elementos)

```
void init_mat(int mat[][2], int count_l, int count_c) {
  int i, j;
  for(i=0; i < count_l; i++)
    for(j=0; j < count_c; j++)
      mat[i][j] = mat[i][j] * 10;
}
int main(void) {
  int mat[2][2] = {{3,4},{5,6}};
  init_mat(mat, 2, 2);  /* note a ausência da notação "[]" e "*" */
  return 0;
}
```

É necessário incluir o tamanho da 2ª dimensão

STRINGS

Strings

- **Strings em C são:**
  - Sequências de *bytes* em memória contendo caracteres
  - Utilização de aspas duplas ("")
- **Qual o tipo de strings?**
  - Strings não possuem um tipo específico em C
  - Strings são do tipo: **char ***
  - **Logo, strings são *arrays* do tipo char**

```
const char *s = "Hello!";   /* tam = 6+1*/
printf("%s\n", s);     /* imprime "Hello!" */
printf("%c\n", *s);   /* imprime 'H' */
printf("%c\n", s[0]); /* imprime 'H' */
printf("%c\n", s[1]); /* imprime 'e' */
printf("%c\n", s[4]); /* imprime 'o' */
```

Strings

- **Representação interna**
  - Strings devem obrigatoriamente terminar com um caractere **NUL ('\0')**
  - O caractere NUL define o **fim da string**
  - Strings devem ter um tamanho mínimo capaz de armazenar o **texto + '\0'**

const char s[8] = "Hello!";

0	1	2	3	4	5	6	7
'H'	'e'	'l'	'l'	'o'	'!'	'\0'	???

Strings

```
#include <stdio.h>

int main(void) {
   int i = 0;
   const char *str = "Hello World!\n";

   /* Imprime cada caractere até encontrar '\0' */
   while (str[i] != '\0')
     printf("%c", str[i++]);

   return 0;
}
```

Strings

- **Manipulação de strings**
  - A biblioteca *string.h* possui inúmeras funções para manipular strings
  - **strlen():** retorna o tamanho de uma string
  - **strcpy():** copia uma string para outra string
  - **strcat():** concatena duas strings
  - **strcmp():** compara duas strings
  - **...**
- **Conversão de strings para inteiros**
  - **atoi(str)** → biblioteca **stdlib.h**

ALOCAÇÃO DINÂMICA DE MEMÓRIA

Alocação Dinâmica de Memória

- **Permite alocar dinamicamente**
  - Variáveis simples (int, char, ...)
  - *Arrays* e matrizes com tamanho determinado em **tempo de execução**

- **Funções úteis (stdlib.h)**
  - malloc()
  - free()

Alocação Dinâmica de Memória

- **malloc():** aloca uma **área contígua** de memória e retorna um **ponteiro para o início da área** de memória alocada (**NULL** em caso de erro)
- **Sintaxe:**
  - ***type* *ptr = (*type* *) malloc(*byte-size*)**
  - ***type*:** tipo dos dados a serem alocados
  - ***byte-size:*** quantidade de bytes a serem alocados (deve ser múltiplo do tipo dos dados)

```
int *array_int = (int *) malloc(100 * sizeof(int));
char *str = (char *) malloc(200 * sizeof(char));
```

Alocação Dinâmica de Memória

- **free():** desaloca uma área contígua de memória alocada por malloc()
- **Sintaxe:**
  - **free(*ptr*)**
  - ***ptr*:** ponteiro para área alocada

```
free(array_int);
free(str);
```

ESTRUTURAS

Estruturas

- **Estruturas (structs)**
  - Forma de **agrupar variáveis**
  - Variáveis podem ser de **tipos diferentes**
  - Acesso aos campos com uso do operador **"."**

```
struct estrutura {
    int a;
    float b;
};
int main(void) {
    struct estrutura s;
    s.a = 3490;       /* atribuição a um elemento da struct! */
    s.b = 3.14159;  /* atribuição a um elemento da struct! */
    printf("Valor do campo a: %d\n", s.a);
    return 0;
}
```

Estruturas

- **Estruturas alocadas dinamicamente**
  - Acesso aos campos se faz com o operador **"->"**

```
struct estrutura {
    int a;
    float b;
};
int main(void) {
    struct estrutura *s2;
    s2 = (struct estrutura * ) malloc(sizeof(struct estrutura));

    s2->a = 321;
    s2->b = 3.9;
    return 0;
}
```

Estruturas

- **Definindo um novo tipo: typedef**
  - Permite definir um novo tipo de dados
  - Útil quando utilizado em conjunto com *structs*

```
typedef struct {
   int a;
   float b;
} meu_novo_tipo;

int main(void) {
   meu_novo_tipo var;
   var.a = 2;
   var.b = 5.1;
   return 0;
}
```