

Unidade VII - *Arrays* e Apontadores

Disciplina Linguagens de Programação I
Bacharelado em Ciência da Computação da Uerj
Professores Guilherme Mota e Leandro Marzulo

ANSI C

```
#include <stdio.h>
int main ()
{
    printf("Hello World!");
    return 0;
}
```

Que assuntos serão abordados nesta unidade?

- Memória, apontadores e endereços
- Operadores * e &
- Passagem de parâmetros por referência
- Vetores e apontadores
- Aritmética de apontadores
- Alocação dinâmica de *arrays* e *strings*
- *Arrays* de apontadores e apontadores de apontadores
- Precedência de operadores e declarações avançadas
- Argumentos de linha de comando em programas
- apontadores para funções
- Funções com número variável de argumentos

Introdução

Na linguagem C, para que controlar a memória?

- Fazer mais de uma variável apontar para o mesmo conteúdo.
- Alocar memória em função do tamanho da entrada, otimizando o uso de recursos
- Emular a passagem de parâmetros por referência
- Manipular arrays através de apontadores e de forma otimizada
- Variar a forma da execução de uma função através do uso de diferentes funções núcleo
- Passar argumentos pela linha de comando
- Escrever funções que assim como `printf` e `scanf` possam receber de 1 a n argumentos.

Operador * e declaração de apontadores

```
int main()
{
    int cont;
    int* pInt;
    return 0;
}
```

0x0000000003CE10CB
0x0000000003CE10CC
0x0000000003CE10CD
0x0000000003CE10CE
0x0000000003CE10CF
0x0000000003CE10D0
0x0000000003CE10D1
0x0000000003CE10D2
0x0000000003CE10D3
0x0000000003CE10D4
0x0000000003CE10D5
0x0000000003CE10D6
0x0000000003CE10D7
0x0000000003CE10D8
0x0000000003CE10D9
0x0000000003CE10DA

0xA5	cont
0xFD	
0x01	
0xFF	
0x96	pInt
0xB7	
0x32	
0x08	
0xA5	
0xFD	
0x01	
0xFF	
0x96	
0xB7	
0x32	
0x08	

Operador & e endereço de variáveis

```
int main()
{
    int cont;
    int* pInt;
    pInt = &cont;
    return 0;
}
```

0x0000000003CE10CB

0x0000000003CE10CC

0x0000000003CE10CD

0x0000000003CE10CE

0x0000000003CE10CF

0x0000000003CE10D0

0x0000000003CE10D1

0x0000000003CE10D2

0x0000000003CE10D3

0x0000000003CE10D4

0x0000000003CE10D5

0x0000000003CE10D6

0x0000000003CE10D7

0x0000000003CE10D8

0x0000000003CE10D9

0x0000000003CE10DA

0xA5	cont
0xFD	
0x01	
0xFF	
0x3CE10CB	pInt
0x96	
0xB7	
0x32	
0x08	

Operador * indireção - “conteúdo apontado por”

```
int main()
{
    int cont=10;
    int* pInt;
    pInt=&cont;
    (*pInt)++;
    ++*pInt;
    return 0;
}
```

0x0000000003CE10CB

0x0000000003CE10CC

0x0000000003CE10CD

0x0000000003CE10CE

0x0000000003CE10CF

0x0000000003CE10D0

0x0000000003CE10D1

0x0000000003CE10D2

0x0000000003CE10D3

0x0000000003CE10D4

0x0000000003CE10D5

0x0000000003CE10D6

0x0000000003CE10D7

0x0000000003CE10D8

0x0000000003CE10D9

0x0000000003CE10DA

12	cont
0x3CE10CB	pInt
0x96	
0xB7	
0x32	
0x08	

Um outro exemplo

```
int main()
{
    int* pCoord;
    int linha=1;
    pCoord=&linha;
    printf("linha: %d\n", linha);
    printf("*pCoord: %d\n", (*pCoord)++);
    printf("linha: %d\n", linha);
    printf("*pCoord: %d\n", *pCoord);
    return 0;
}
```


Passagem de parâmetros por referência

Passagem de parâmetros por referência

- Não modifica o escopo de `main`

```
void swap (int, int);

int main()
{
    int a = 10, b = 20;
    swap(a, b);
    printf("%d, %d\n", a, b);
    return 0;
}

void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

- Modifica o escopo de `main`

```
void swap (int*, int*);

int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
    printf("%d, %d\n", a, b);
    return 0;
}

void swap(int* x, int* y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Exemplo: parâmetros por referência

```
int getint(int *pn);
```

Descrição:

Obtém o inteiro equivalente a partir de uma sequência de dígitos proveniente de stdin. São válidos caracteres numéricos, além dos sinais de '+' e '-' somente caso estejam na primeira posição. Espaços, tabs e outros caracteres de espaço no início da sequência são ignorados. Após o início de um número inteiro, caracteres não dígito terminam a conversão do inteiro.

Caso o primeiro character lido não seja um espaço, dígito, '+' ou '-', a função retorna EOF.

Retorno:

EOF - caso não haja caracteres em stdin

Zero - se a sequência não corresponde a um inteiro válido

Número positivo - para um inteiro válido

O apontador pn aponta para o inteiro lido.

Exemplo: parâmetros por referência

```
int getint(int *pn) /* get next integer from input into *pn */
{
    int c, sign;
    while (isspace(c = getc(stdin))) /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-')
    {
        ungetc(c, stdout); /* it is not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getc(stdin);
    for (*pn = 0; isdigit(c); c = getc(stdin))
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetc(c, stdout);
    return c;
}
```

Exercício

Exercício

- Escreva uma função `maxmin` que receba dois argumentos do tipo `float`, `max` e `min`. Independentemente de quem seja o maior dos parâmetros no momento da chamada da função, ao final da execução de `maxmin`, `max` deve conter o maior dos argumentos e `min` o menor. Este resultado deve se refletir na função chamadora.

Apontadores e Arrays

apontadores e arrays na linguagem C

- Existe um fortíssimo relacionamento entre apontadores e arrays na linguagem C
- Tamanho é a força deste relacionamento que arrays e apontadores podem ser discutidos conjuntamente
- As principais diferenças são:

	Array	Apontador como array
Declaração	Tipo MeuArray[]	Tipo *apontador
Alocação	Os elementos do array são alocados automaticamente	Somente o apontador é alocado automaticamente
Atribuição	Não pode estar no lado esquerdo de uma atribuição	Endereço apontado pode ser alterado em tempo de execução
Acesso a um elemento	MeuArray[i]	apontador[i] ou *(apontador+i)

Arrays automáticas

```
int main()  
{  
    short Vet[2];  
    return 0;  
}
```

0x0000000003CE10CB
0x0000000003CE10CC
0x0000000003CE10CD
0x0000000003CE10CE
0x0000000003CE10CF
0x0000000003CE10D0
0x0000000003CE10D1
0x0000000003CE10D2
0x0000000003CE10D3
0x0000000003CE10D4
0x0000000003CE10D5
0x0000000003CE10D6
0x0000000003CE10D7
0x0000000003CE10D8
0x0000000003CE10D9
0x0000000003CE10DA

0x03CE10D3		V e t
0xA5	[0]	
0xFD		
0x01	[1]	
0xFF		
0x96		
0xB7		
0x32		
0x08		

Arrays automáticas

```
int main()
```

```
{
```

```
    char MyStr[5];
```

```
    return 0;
```

```
}
```

0x000000000001C10AB

0x000000000001C10AC

0x000000000001C10AD

0x000000000001C10AE

0x000000000001C10AF

0x000000000001C10B0

0x000000000001C10B1

0x000000000001C10B2

0x000000000001C10B3

0x000000000001C10B4

0x000000000001C10B5

0x000000000001C10B6

0x000000000001C10B7

0x000000000001C10B8

0x000000000001C10B9

0x000000000001C10BA

0x1C10B3

M
y
S
t
r

0xA5

0xFD

0x01

0xFF

0x96

0xB7

0x32

0x08

[0]

[1]

[2]

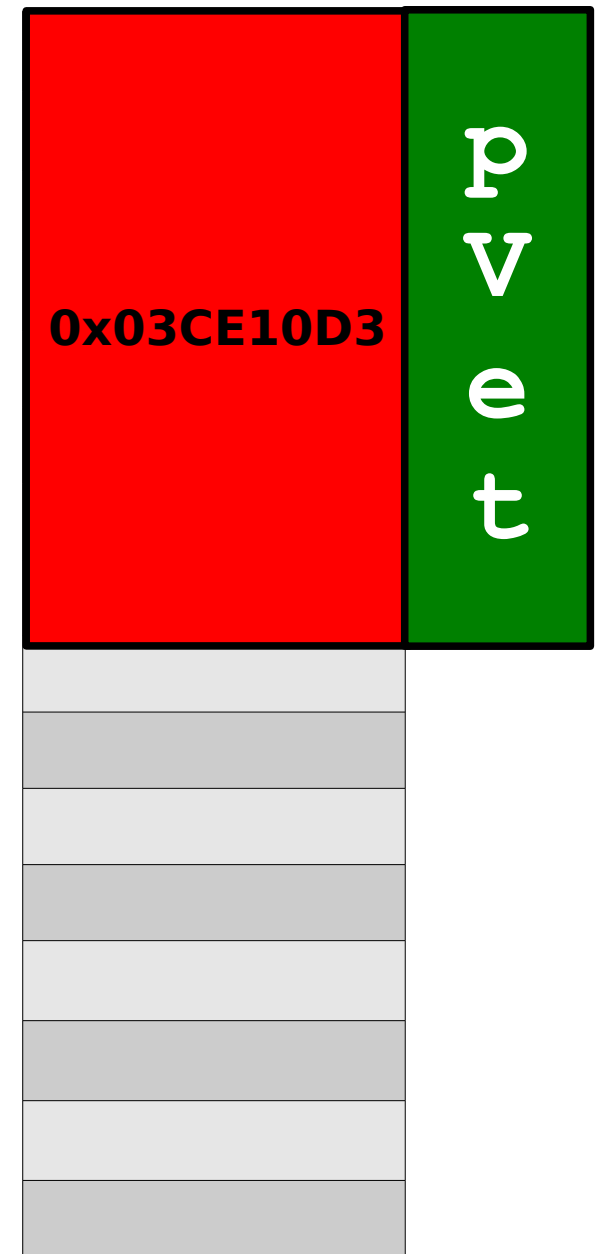
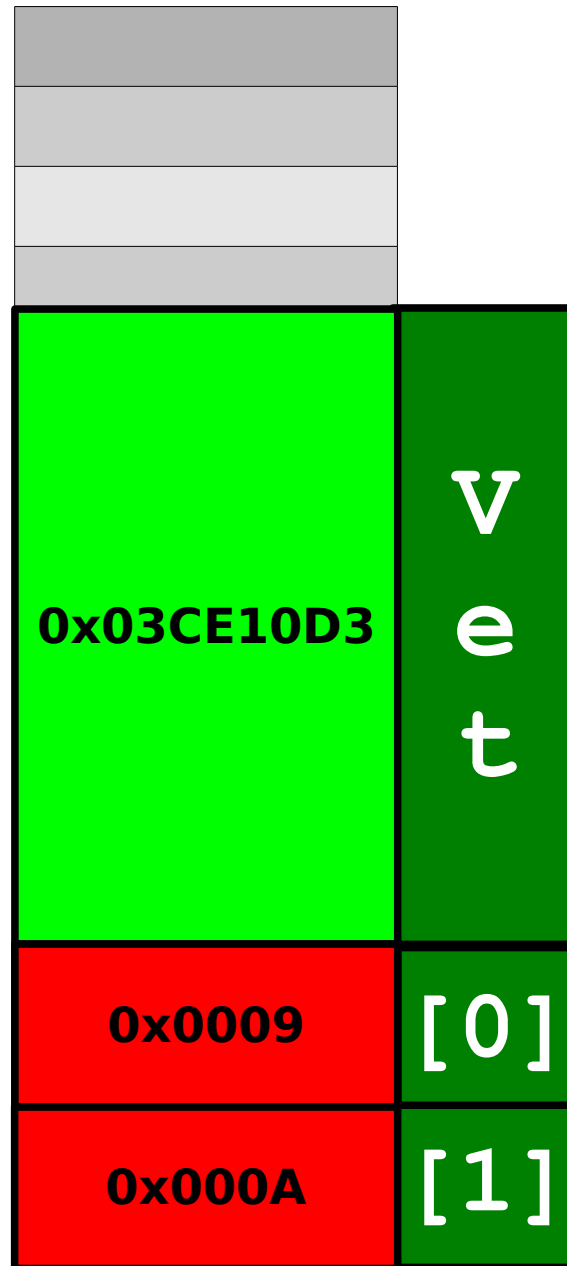
[3]

[4]

Apontadores e Arrays

```
int main()
{
    short Vet[2];
    short* pVet = NULL;
    pVet = Vet;
    *pVet=5;
    pVet[0]=9;
    pVet[1]=10;
    return 0;
}
```

0x0000000003CE10D3



Alocação Dinâmica

Alocação Dinâmica

- *Arrays* têm dimensões definidas em tempo de compilação
- E se o tamanho das entradas variar entre uma execução e outra?
- Podemos superdimensionar o tamanho do *array* e mudarmos a parcela efetivamente utilizada, ou
- alocar em tempo de execução o “*array*” variando o seu comprimento em função das entradas

Alocação Dinâmica de memória

```
int main()
{
    short* pVet = NULL;
    pVet = (short*) malloc(5*sizeof(short));
    return 0;
}
```

0x03CE10D4

0xA5	
0xFD	[0]
0x01	
0xFF	[1]
0x96	
0xB7	[2]
0xE4	
0xDF	[3]
0x32	
0x08	[4]
0xFB	
0xA2	

0x03CE10D4	p v e t
0xA5	
0xFD	
0x01	
0xFF	
0x96	
0xB7	
0x32	
0x08	

Alocação dinâmica de memória

- Operador:
 - `sizeof()`
 - Retorna o tamanho em bytes de uma variável ou tipo

```
int main()  
{  
    printf("%d\n", sizeof(int*));  
    printf("%d\n", sizeof(void*));  
    printf("%d\n", sizeof(double*));  
    return 0;  
}
```

Alocação dinâmica de memória

- Operador:
 - **(Tipo)**
 - Força a conversão do tipo - *Type casting*

```
int main()  
{  
    printf("%d\n", (int) 'A');  
    printf("%f\n", (float) 200);  
    return 0;  
}
```


Alocação dinâmica de memória

- Funções de `stdlib.h`:
 - `void* malloc(size_t size)`
 - Aloca um bloco de `size` bytes de memória e retorna o endereço do início do bloco

```
int main()
{
    int* pInt = (int*) malloc(5*sizeof(int));
    return 0;
}
```

Alocação dinâmica de memória

- Funções de `stdlib.h`:
 - `void* calloc(size_t num, size_t size)`
 - Aloca uma região de memória contendo `num` elementos, cada um com `size` bytes, zera todos os bits do bloco e retorna o endereço do início do bloco

```
int main()
{
    int* pInt = (int*) calloc(4, sizeof(int));
    return 0;
}
```

Alocação dinâmica de memória

- Funções de `stdlib.h`:
 - `void* realloc(void* ptr, size_t size)`
 - Redimensiona o bloco apontado por `ptr` para `size` bytes de memória e retorna o endereço inicial
 - O conteúdo é preservado a menos que `size` seja menor que o tamanho do bloco original

```
int main()  
{  
    float* pFloat = (float*) calloc(4, sizeof(float));  
    pFloat = (float*) realloc(pFloat, 8 * sizeof(float));  
    return 0;  
}
```

Alocação dinâmica de memória

- Funções de `stdlib.h`:
 - `void free(void* ptr)`
 - Libera bloco de memória previamente alocado apontado por `ptr`

```
int main ()
{
    int *buf1 = NULL, *buf2 = NULL, *buf3 = NULL;
    buf1 = (int*) malloc (100*sizeof(int));
    buf2 = (int*) calloc (100,sizeof(int));
    buf3 = (int*) realloc (buf2,500*sizeof(int));
    free (buf1);
    free (buf3);
    return 0;
}
```

Aritmética de apontadores

Aritmética de apontadores

- apontadores podem ser submetidos a operações lógicas e aritméticas
- apontadores podem ser incrementados ou decrementados de x posições
 - Operadores: $+$, $-$, $++$, $--$, $+=$, $-=$
 - Neste processo o tipo é usado para saber o tamanho do passo em bytes
- Dois endereços podem ser comparados
 - Operadores: $<$, $>$, $>=$, $<=$
- Endereços podem ser convertidos
 - Operador de `typecasting`

Arrays e Aritmética de apontadores

- Como *arrays*

```
int strlen(char s[])
{
    int len=0;
    while (s[len] != '\0')
        len++;
    return len;
}
```

- Aritmética de apontadores

```
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
```

Exercícios

Exercício

- Converta a função abaixo substituindo o operador [] por aritmética de apontador e pelo operador de declaração de apontador *.

```
/* copy: copy 'from' into 'to' */
void copy(char to[], char from[])
{
    int i;
    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Exercício

- Converta a função abaixo substituindo o operador [] por aritmética de apontador e pelo operador de declaração de apontador *.

```
/* atoi: convert s to integer */
int atoi(char s[])
{
    int i, n;
    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Programa Principal Soma Vetores

```
#include <stdio.h>
#include <stdlib.h>
float* getvect(int *);
float* sum(float*, float*, int);
void print (float*, int);

int main()
{
    int size_v1, size_v2, size_v3;
    float *v1 = getvect(&size_v1);
    float *v2 = getvect(&size_v2);
    if (size_v1==size_v2)
    {
        float *v3 = sum(v1,v2,size_v1);
        print(v3, size_v1);
        free (v3);
    }
    else
        printf("Erro: os vetores tem que ter o mesmo tamanho.\n");
    free (v1);
    free (v2);
    return 0;
}
```

Funções e apontadores de char

Arrays, Constantes e apontadores de char

```
char aMsg[] = "Linguagem C";
```

0x00000000FEFFBAB2	00	aMSG
	00	
	00	
	00	
	FE	
	FF	
	BA	
	BA	
	BA	
	BA	
0x00000000FEFFBABA	'L'	aMsg[0]
	'i'	aMsg[1]
	'n'	aMsg[2]
	'g'	aMsg[3]
	'u'	aMsg[4]
	'a'	aMsg[5]
	'g'	aMsg[6]
	'e'	aMsg[7]
	'm'	aMsg[8]
	' '	aMsg[9]
	'C'	aMsg[10]
	'\0'	aMsg[11]

```
char* pMSG = "Linguagem C";
```

0x00000000FEFFBAB2	00	pMSG
	00	
	00	
	00	
	FF	
	A0	
0x00000000FFA01234	12	
	34	
	'L'	
	'i'	
	'n'	
	'g'	
	'u'	
	'a'	
	'g'	
	'e'	
	'm'	
	' '	
	'C'	
	'\0'	

Exemplos Função strcpy

- Copia t para s → versão índice de array

```
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Exemplos Função strcpy

- Copia t para s → versão aritmética de apontadores

```
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0')
    {
        s++;
        t++;
    }
}
```

Exemplos Função strcpy

- Cópia t para s → versão aritmética de apontadores 2.0

```
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```


Exemplos Função strcpy

- Cópia t para s → versão aritmética de apontadores 3.0

```
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Exemplos Função strcpy

```
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Recordando Precedência de Operadores

A ordem de avaliação dos operadores unários `++`, `--` e `*` é da direita para esquerda

- `*p++ = val` → faz indireção com o apontador original e incrementa endereço
- `val = *--p` → decrementa o endereço original e faz a indireção a partir do novo endereço
- `val = (*p)++` → incrementa ao final da execução da instrução o conteúdo apontado pelo apontador
- `val = ++*p` → incrementa imediatamente o conteúdo apontado pelo apontador

Arrays de apontadores e apontadores de apontadores

Arrays de apontadores

- Apontadores podem ser utilizados como tipos primitivos ou declarados pelo programa
- Assim é possível declarar arrays de apontadores

```
char* StringsArray[10];
```

```
int* IntPtrArray [25];
```

```
struct Student{  
    char Name[255];  
    char ID [12];  
}
```

```
struct Student* StructPointerArray[50];
```

Arrays de apontadores

- O tipo do cabeça do array é: apontador para o tipo do elemento
- Portanto, se cada elemento é um apontador para inteiro `int*`, o cabeça do array é do tipo `int **`

```
int* IntPtrArray [25];
```

Arrays de apontadores

```
int* IntPtrArray [3];
```

0x00000000FEFFBAB2	IntPtrArray	int**
0x00000000FEFFBABA	IntPtrArray[0]	int*
0x00000000FEFFBAC2	IntPtrArray[1]	int*
0x00000000FEFFBACA	IntPtrArray[2]	int*

Programa exemplo *array* de apontadores

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

void printText(char* [], int);
void sortText(char* [], int);
void swap(char * [], int, int);

int main (void)
{
    char * text []={"defghi", "jklmnopqrst", "abc"};
    printText(text, 3);
    sortText(text, 3);
    printText(text, 3);
    return 0;
}
```


Programa exemplo array de apontadores

```
void printText(char* lText[], int Size)
{
    printf("\n-----\n");
    while (Size>0)
    {
        printf("%s\n", *lText++);
        Size--;
    }
    printf("-----\n");
}
```

Programa exemplo array de apontadores

```
void sortText(char * lText[], int Size)
{
    int i, j;
    for(i=Size-2; i >= 0; i--)
        for (j=0; j<=i; j++)
            if(strcmp(lText[j], lText[j+1])>0)
                swap(lText, j, j+1);
}
```

```
void swap(char* lText[], int i, int j)
{
    char * Temp;
    Temp = lText[i];
    lText[i] = lText[j];
    lText[j] = Temp;
}
```

Arrays

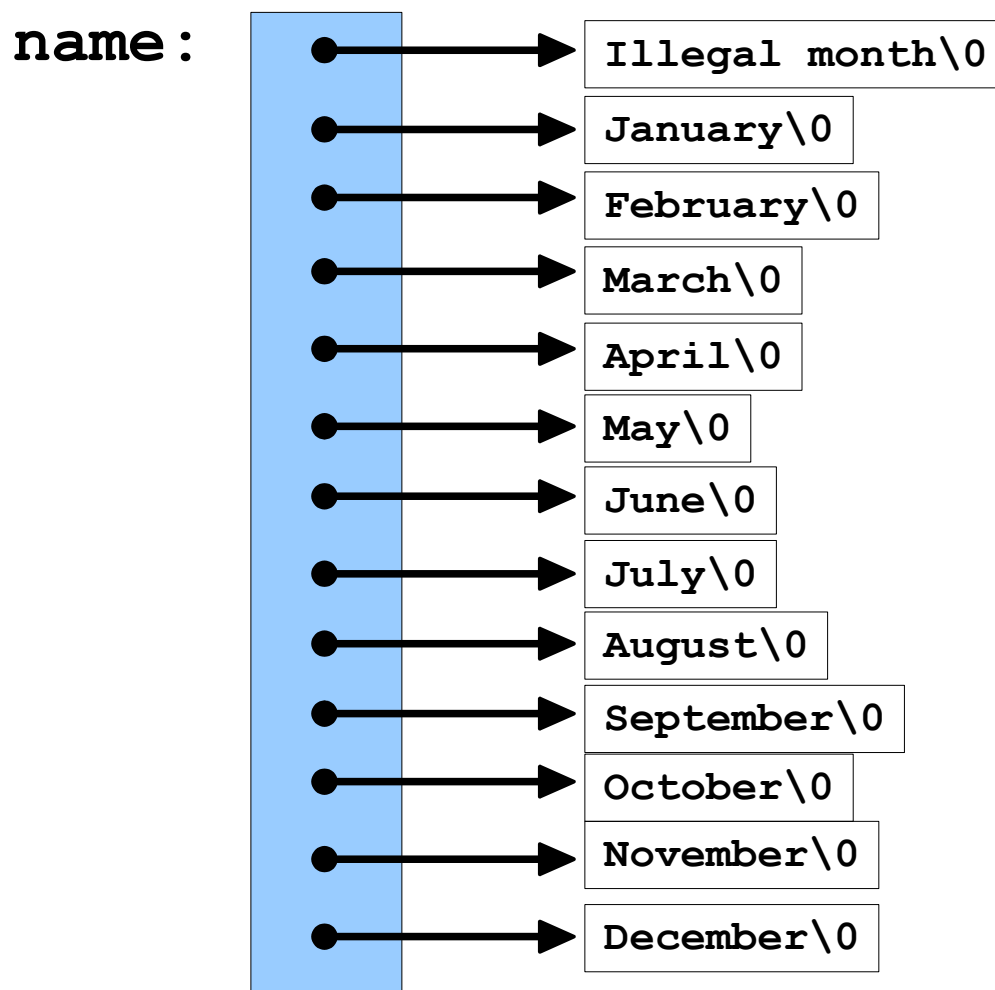
Multidimensionais

Inicialização de Arrays de apontadores

```
/* month_name: return name of n-th month */  
  
char *month_name(int n)  
{  
    static char *name[] =  
    {  
        "Illegal month", "January", "February",  
        "March", "April", "May", "June", "July",  
        "August", "September", "October",  
        "November", "December"  
    };  
    return (n < 1 || n > 12) ? name[0] : name[n];  
}
```

Arrays de apontadores

```
char* name[13]={ "Illegal month", "January", "February", "March",  
"April", "May", "June", "July", "August", "September", "October",  
"November", "December" };
```



Arrays Multidimensionais

```
char aname[][15]={"Illegal month", "January", "February", "March",  
"April", "May", "June", "July", "August", "September", "October",  
"November", "December" };
```

&CABEÇA	Illegal month\0	January\0	February\0	March\0
	0	15	30	45

April\0	May\0	June\0	July\0
60	75	90	105

August\0	September\0	October\0	November\0
120	135	150	165

December\0
180

135

Exemplos de Tipos Multidimensionais Compatíveis

```
f(int daytab[2][13]) { ... }
```

```
f(int daytab[][13]) { ... }  
/*tamanho da primeira dimensão é irrelevante*/
```

```
f(int (*daytab)[13]) { ... }  
/*apontador para array de 13 elementos inteiros*/
```

```
f (int* daytab[13]) {...}  
/* incompatível! Este tipo é um array de 13 posições de  
apontador para inteiro*/
```

Redução de *Array* Bidimensional para apontador

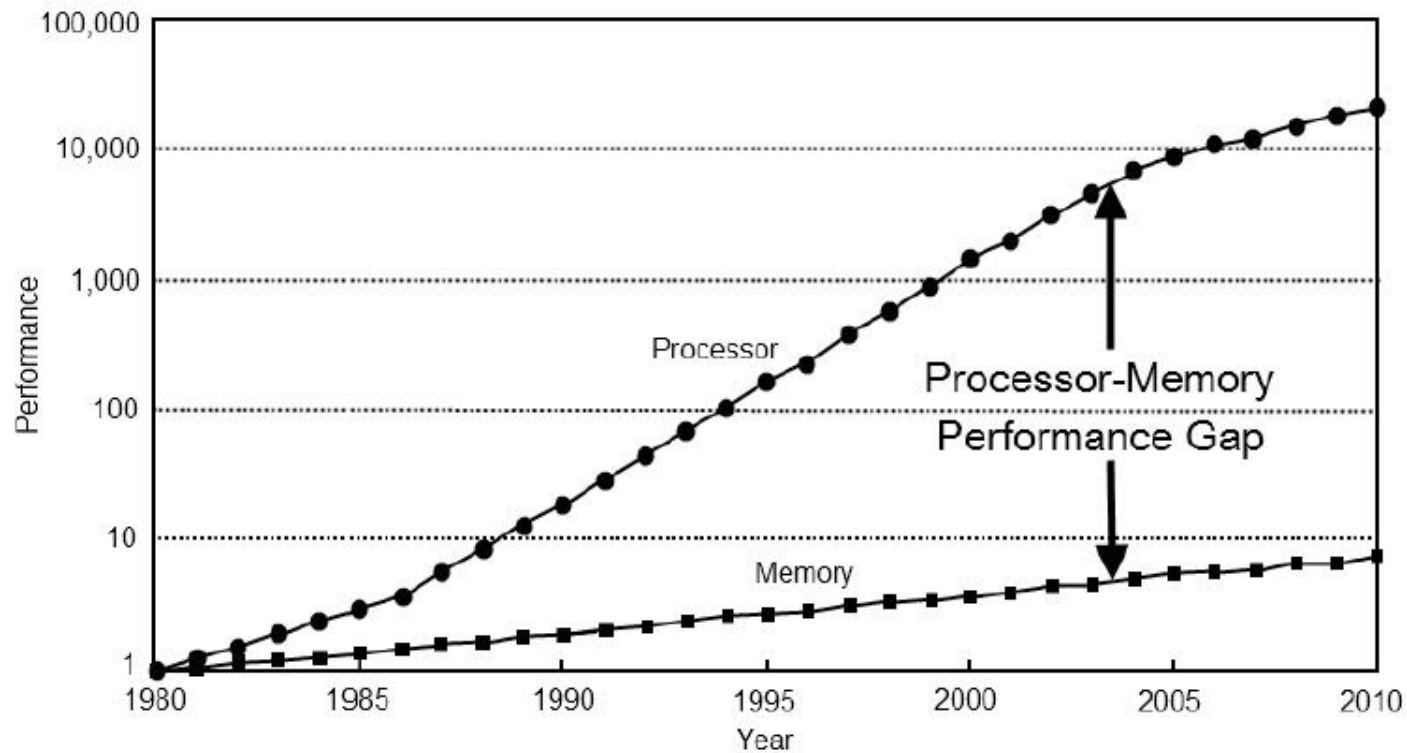
```
void printMat(int * Mat, int rows, int cols)
{
    int i, j;
    printf("{\n");
    for(i=0; i<rows; i++)
    {
        printf("  {");
        for (j=0; j<cols; j++)
            printf("%d%s", *(Mat+cols*i+j), (j<cols-1)?"", ":\n" );
    }
    printf("}\n");
}

int main()
{
    int Mat1[3][3]={ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    int Mat2[2][4]={ {1, 0, 0, 0}, {0, 0, 0, 1} };
    printMat((int *)Mat1, 3, 3);
    printMat((int *)Mat2, 2, 4);
    return 0;
}
```


Alocação Dinâmica de *Arrays* Multidimensionais

Memory Wall

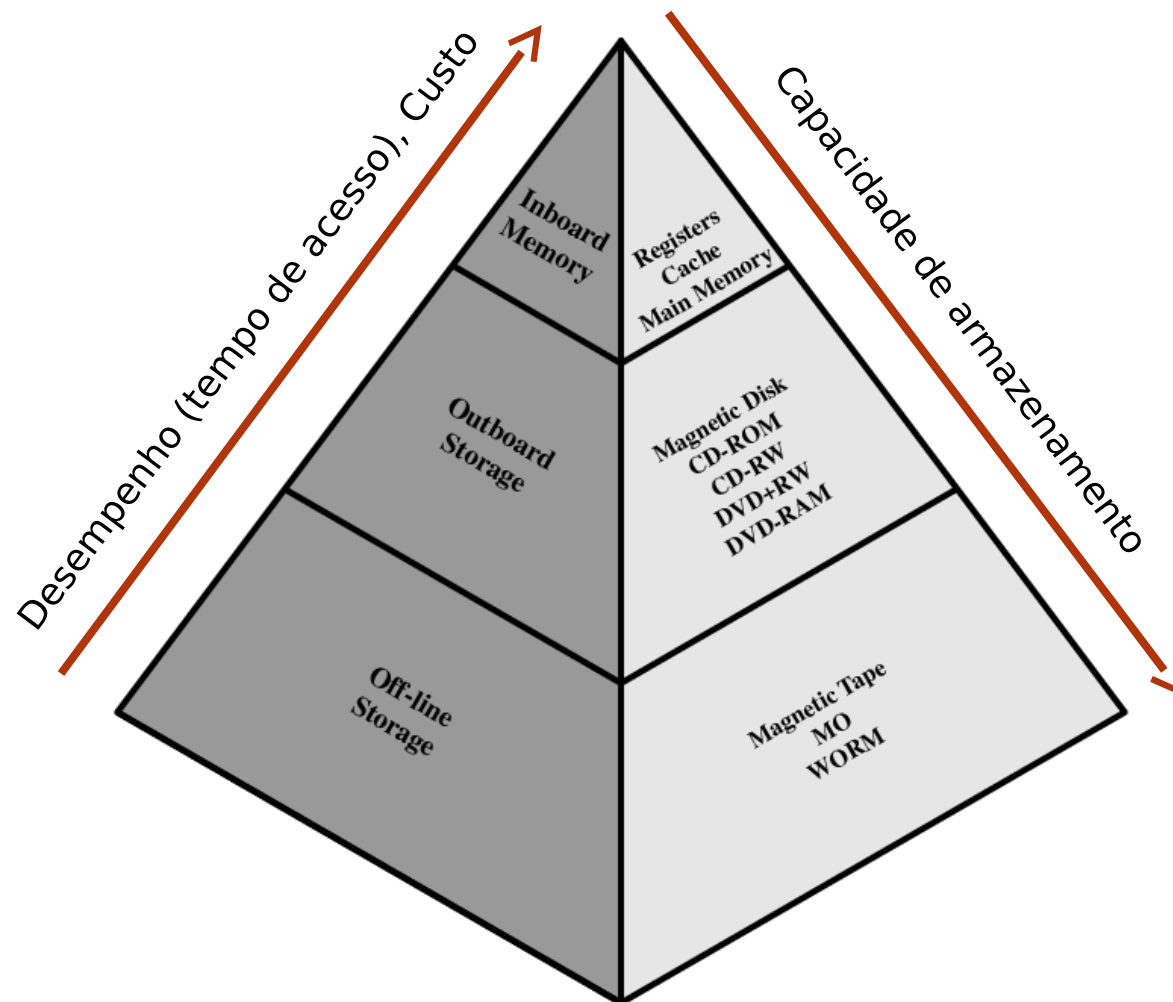
CPU/Memory performance



Slide 17

Computer architecture: a quantitative approach
By John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

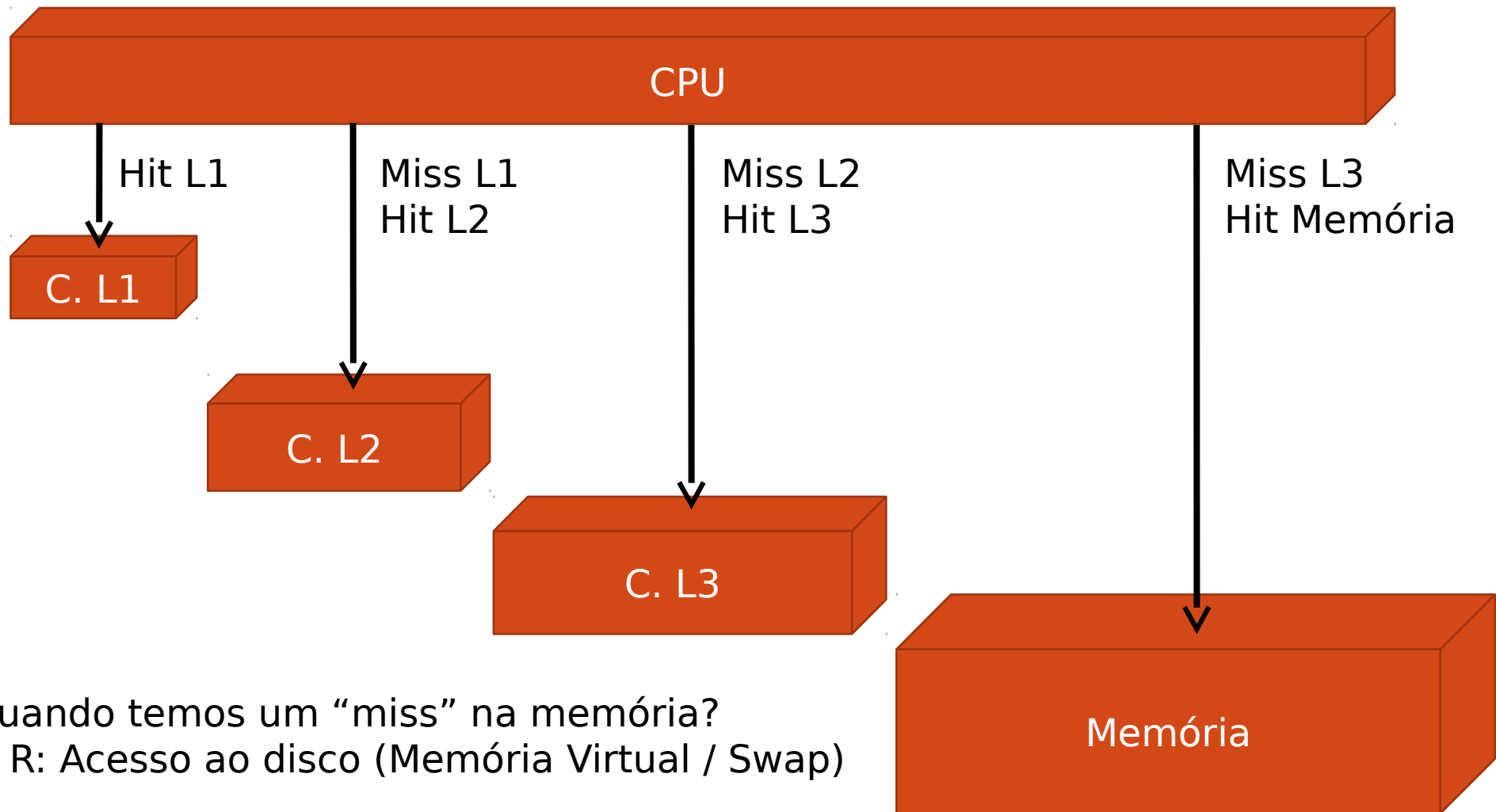
Hierarquia de Memória



Características dos Níveis da Memória

Nível	1	2	3	4
Nome	Registradores	Cache	Memória	Disco
Tamanho	<1KB	<16MB	<512GB	>1TB
Tecnologia	Memória customizada com múltiplas portas, CMOS	CMOS SRAM (flip-flops) on-chip ou off-chip	CMOS DRAM (capacitores)	Disco magnético
Tempo de Acesso (ns)	0,25 - 0.5	0,5 - 25	50-250	5.000.000
Banda (MB/s)	50.000 -500.000	5.000 - 20.000	2.500 - 10.00	50 - 500
Gerenciado pelo	Compilador	Hardware	SO	Operador do SO
Backup	cache	Memória	Disco	CD ou Fita

Acesso



E quando temos um “miss” na memória?

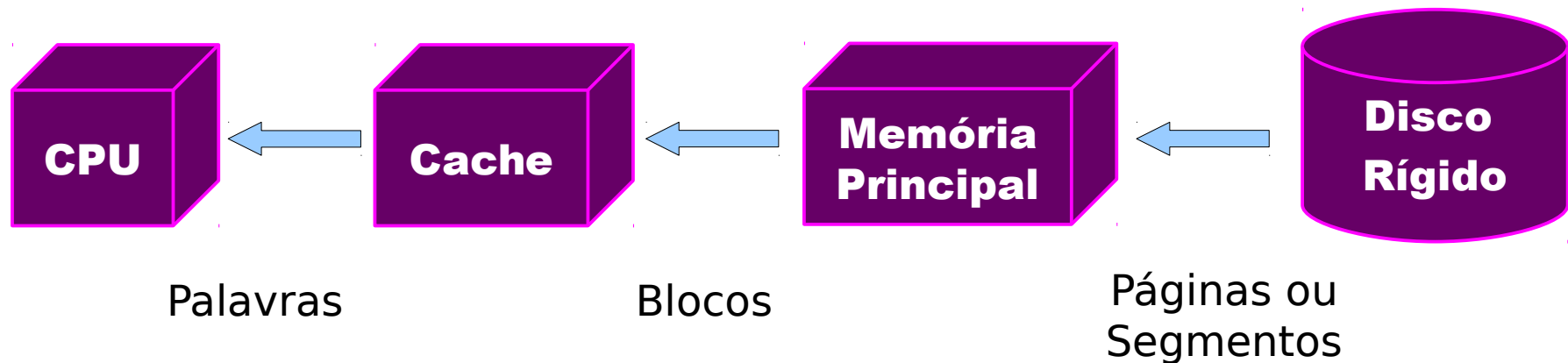
R: Acesso ao disco (Memória Virtual / Swap)

Princípio da Localidade

- Programas tendem a acessar uma porção relativamente pequena do executável em um determinado intervalo.
- Dois tipos de localidade:
 - Temporal: se um item é referenciado, ele tende a ser referenciado novamente em breve (ex: loops e reuso de variáveis)
 - Espacial: se um item é referenciado, itens adjacentes (com endereço próximo) tendem a ser referenciados em breve (ex: vetores)

Reflexo da localidade na hierarquia de memória

- Tempo: Manter os dados mais frequentemente, ou recentemente acessados nos níveis mais altos.
- Espaço: carregar blocos.



Alocação contínua vs alocação por linha

	Contínua	Por linha
Acesso	É preciso usar fórmula de mapeamento ($i \times \text{colunas} + j$) ou preencher manualmente o vetor de apontadores para as linhas	Abstração <code>[][]</code> é direta
Custo da alocação	1 malloc para os dados + 1 malloc para o vetor de apontadores (opcional)	1 malloc por linha + 1 malloc para o vetor de apontadores
Uso da cache	Mais eficiente. Para matrizes pequenas, várias linha cabem em um bloco de cache.	Menos eficiente. Para matrizes grandes é indiferentes, pois uma linha ocupa vários blocos de cache.
Erros na alocação	Para matrizes grandes, pode não ser possível achar um espaço que comporte a matriz completa	Se a linha for muito grande também podemos ter erros no malloc (menos provável)
Resumo	Melhor desempenho (cache e custo de alocação). Maior complexidade. Faz mais sentido para matrizes menores.	Pior desempenho. Menor complexidade. Compensa para matrizes grandes.

Array Multidimensional Dinâmico Degenerado

```
int *m;
int i,j;

/*aloca m*/
if ((m = (int *) malloc(10 * 5 * sizeof(int))) == NULL)
{
    fprintf(stderr, "Erro no malloc\n");
    return 1;
}

printf("m: %p\n", m);
for (i=0; i<10; i++)
{
    printf("&m[%d]: %p\n", i, &m[i*5]);
    for (j=0; j<5; j++)
        printf("m[%d][%d]: %p\n", i, j, &m[i*5+j]);
}

/*libera m*/
free(m);
```

Alocação Dinâmica Fragmentada por Linha

```
int **m;
int i,j;
/*aloca m*/
if ((m = (int **) malloc(10 * sizeof(int *))) == NULL)
{
    fprintf(stderr, "Erro no malloc\n");
    return 1;
}

for (i=0; i<10; i++)
{
    if ((m[i] = (int *) malloc(5 * sizeof(int))) == NULL)
    {
        fprintf(stderr, "Erro no malloc\n");
        return 1;
    }
}
```

Alocação Dinâmica Fragmentada por Linha

```
printf("m: %p\n", m);
for (i=0; i<10; i++)
{
    printf("m[%d]: %p\n", i, m[i]);

    for (j=0; j<5; j++)
        printf("m[%d][%d]: %p\n", i, j, &(m[i][j]));
}

/*libera m*/
for (i=0; i<10; i++)
    free(m[i]);
free(m);
```

Array Multidimensional com Alocação Contínua

```
int *m;
int **m2;
int i,j;
/*aloca m*/
if ((m = (int *) malloc(10 * 5 * sizeof(int))) == NULL)
{
    fprintf(stderr, "Erro no malloc\n");
    return 1;
}

if ((m2 = (int **) malloc(10 * sizeof(int *))) == NULL)
{
    fprintf(stderr, "Erro no malloc\n");
    return 1;
}

for (i=0; i<10; i++)
{
    m2[i] = &(m[i*5]);
}
```

Array Multidimensional com Alocação Contínua

```
printf("m: %u\n", ((int)m)/4);  
for (i=0; i<10; i++)  
{  
    for (j=0; j<5; j++)  
        printf("m[%d][%d]: %p\n", i, j, &m2[i][j]);  
}  
  
/*libera m*/  
free(m);  
free(m2);
```

Apontadores de funções

Apontadores de Funções

- Em C, é possível definir apontadores de funções
- Apontadores de função permitem:
 - atribuição direta
 - armazenamento em *arrays*
 - passagem como parâmetro para outras funções

Sintaxe de apontadores de Funções

<tipo> (*NomeDoapontador) (<tipoArg1>, ..., <tipoArgN>)

- Apontador de função para função como argumento

```
void qsort(void *lineptr[], int left, int right,  
int (*comp)(void *, void *));
```

```
int numcmp(char *s1, char *s2);
```

```
int main()  
{  
    qsort((void**) lineptr, 0, nlines-1,  
        (int (*)(void*,void*)) (numeric ? numcmp : strcmp));  
    return 0;  
}
```


Exemplo

```
void qsort(void *v[], int left, int right,
int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);
    if (left >= right)
        return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Exemplo

```
int numcmp(char *s1, char *s2)
{
    double v1, v2;
    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Argumentos de Linha de Comando

Argumentos de Linha de Comando

- Em C existe uma forma de passar argumentos para o programa que se deseja executar
- `int argc` → número de argumentos de linha de comando
- `char * argv[]` → array de `[argc]` apontadores para `char`
- `int main(int argc, char* argv[])`

Exemplo

```
/* grep: finds lines that match pattern from 1st arg */
int main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;
    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL)
            {
                printf("%s", line);
                found++;
            }
    return found;
}
```

Exercícios

1) Faça um programa que receba como argumentos de linha de comando uma sequência de números reais e imprima o respectivo somatório

2) Faça um programa que receba como entrada pela linha de comando três argumentos: operando 1, operador e operando 2 e retorne o resultado da operação: use apontadores de função.

Funções com número variável de argumentos

Funções com número variável de argumentos

- Várias das funções do padrão ansi C estudadas podem receber número variável de argumentos:

```
int printf ( const char * format, ... );
```

```
int scanf ( const char * format, ... );
```

```
int fprintf ( FILE * stream, const char *  
              format, ... );
```

```
int fscanf ( FILE * stream, const char * format,  
            ... );
```

```
int sprintf ( char * str, const char * format,  
            ... );
```

```
int sscanf ( const char * s, const char * format,  
            ... );
```


Funções com número de argumentos variável

- Recursos utilizados em sua criação:
 - `stdarg.h` → biblioteca de funções relacionadas
 - `va_list` → tipo usado para armazenar listas de argumentos
 - `void va_start(va_list ap, paramN);` → inicializa e retorna lista contendo os argumentos após `paramN`.
 - `type va_arg(va_list ap, type)` → retorna próximo argumento da lista.
 - `void va_end(va_list ap)` → finaliza a estrutura `ap`
 - `...` → operador que representa os argumentos de número variável

Funções com número de argumentos variável

- Exemplo:

```
#include <stdio.h>
#include <stdarg.h>
void PrintFloats (int n, ...)
{
    int i;
    double val;
    va_list vl;
    va_start(vl, n);
    for (i=0; i<n; i++)
    {
        val=va_arg(vl, double);
        printf (" [%.2f] ", val);
    }
    va_end(vl);
    printf ("\n");
}
int main ()
{
    PrintFloats (1, 3.14159);
    PrintFloats (2, 3.14159, 2.71828);
    PrintFloats (3, 3.14159, 2.71828, 1.41421);
    return 0;
}
```

Exercício

- 1) Faça uma função que calcule o produtório de qualquer quantidade de argumentos do tipo double.
- 2) Crie um programa principal que demonstre o funcionamento de sua função.

Fim