Unidade VII - *Arrays* e Ponteiros

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

ANSI C

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

Que assuntos serão abordados nesta unidade?

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

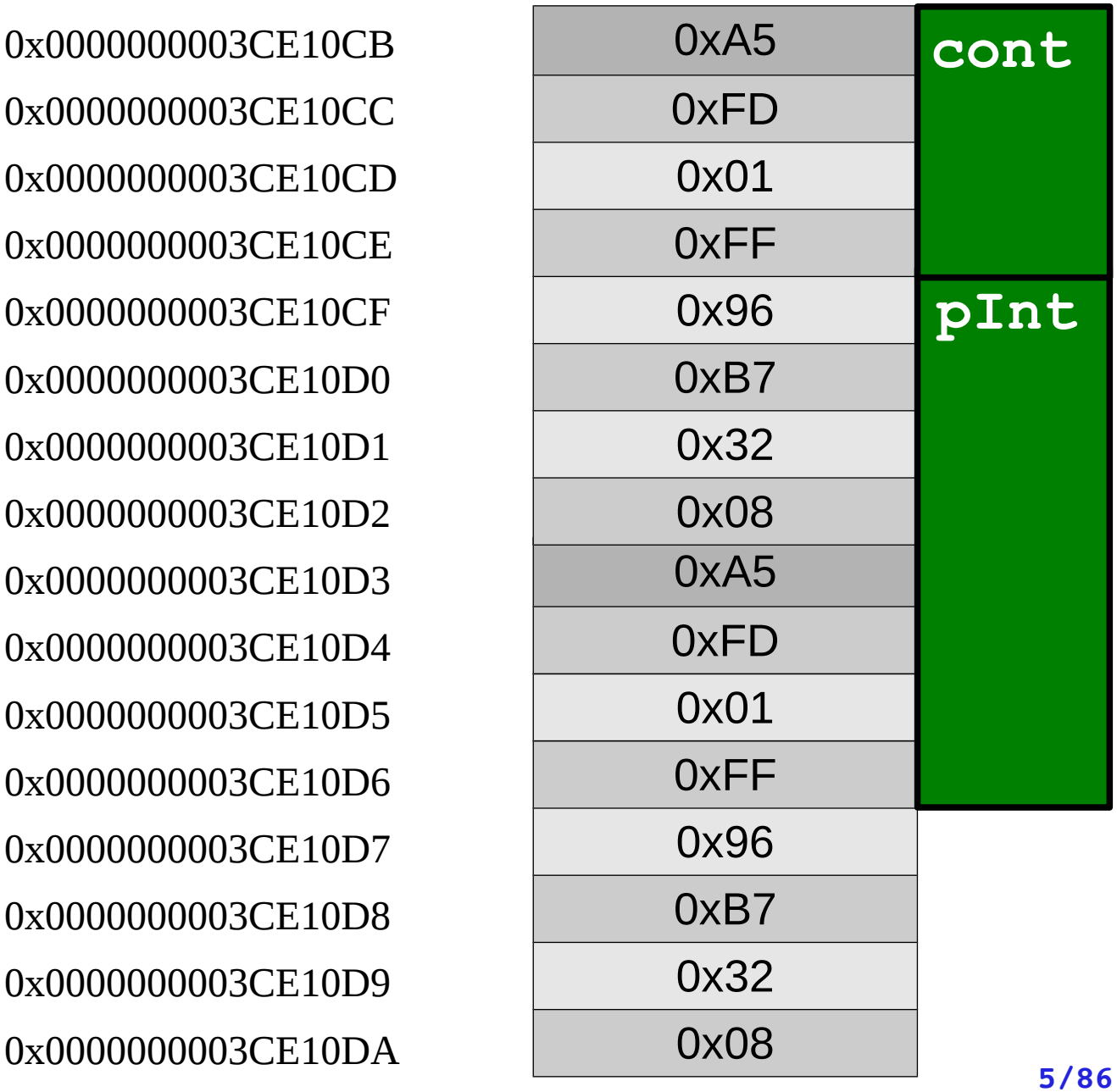
Introdução

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 ponteiros 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.

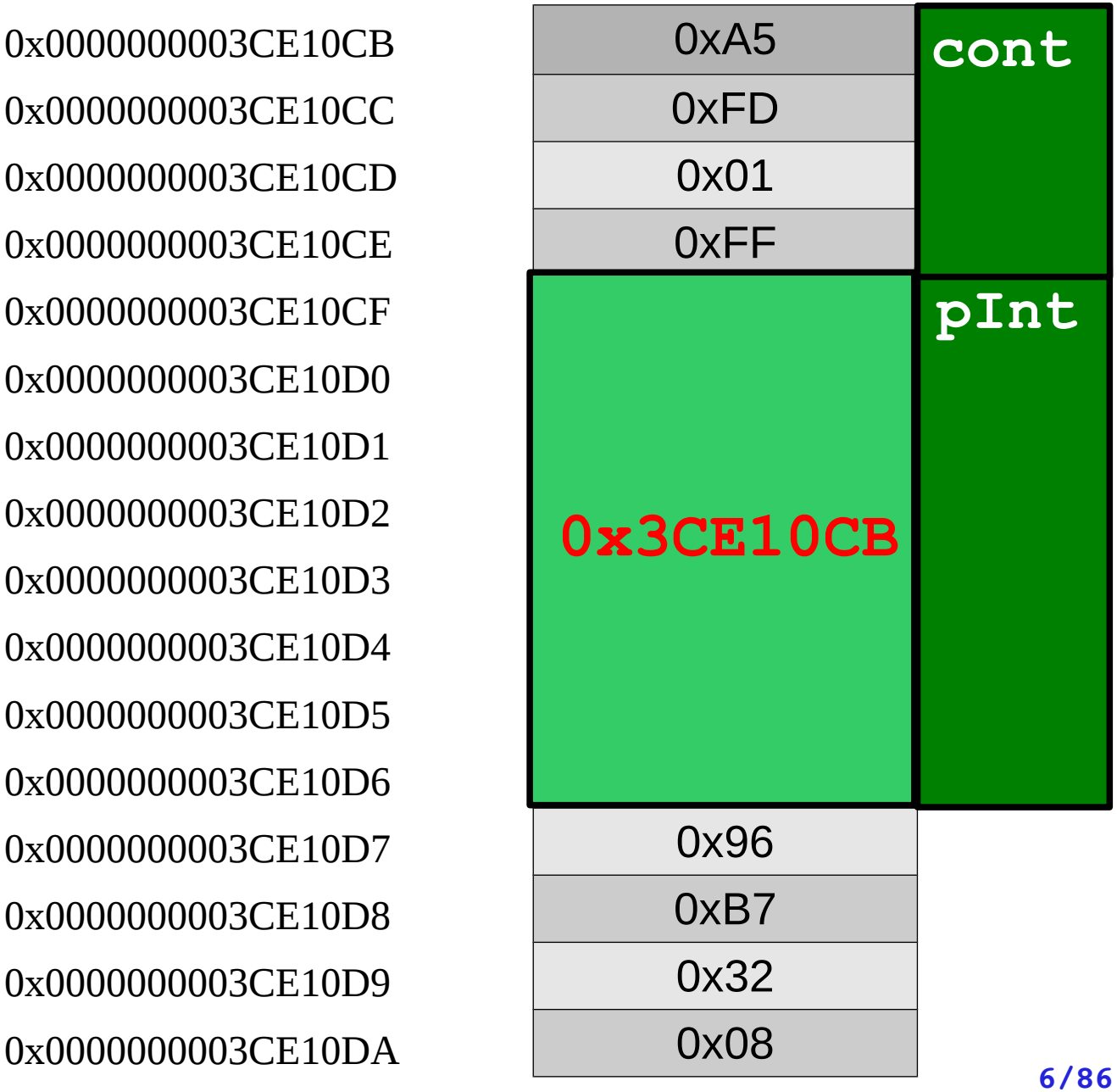
Operador * e declaração de ponteiros

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



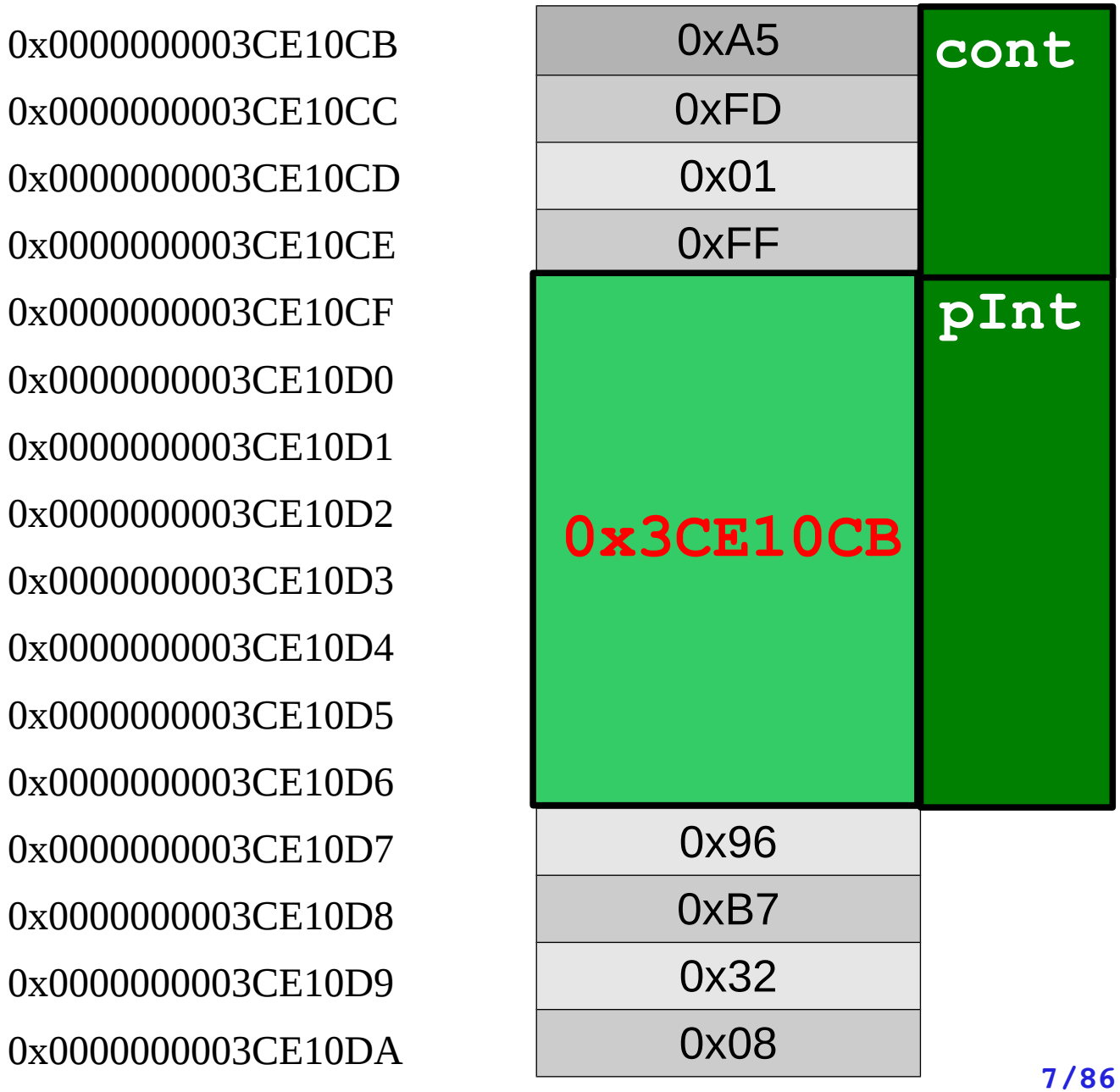
Operador & e endreço de variáveis

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



Operador & e endreço de variáveis

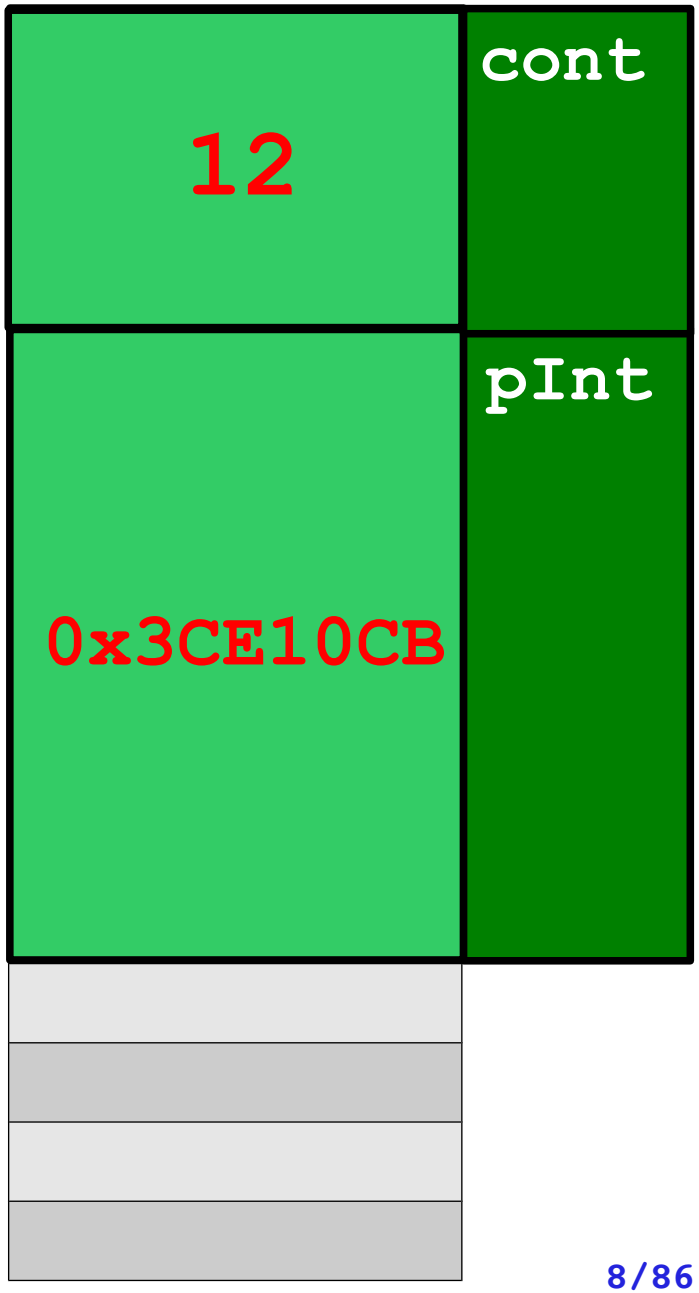
```
main()
{
   int cont;
   int* pInt;
   pInt = &cont;
}
```



Operador * indireção - “conteúdo apontado por”

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

0x000000003CE10CB
0x000000003CE10CC
0x000000003CE10CD
0x000000003CE10CE
0x000000003CE10CF
0x000000003CE10D0
0x000000003CE10D1
0x000000003CE10D2
0x000000003CE10D3
0x000000003CE10D4
0x000000003CE10D5
0x000000003CE10D6
0x000000003CE10D7
0x000000003CE10D8
0x000000003CE10D9
0x000000003CE10DA



Um outro exemplo

```
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);
}
```

Passagem de parâmetros por referência

Funções e Ponteiros - Passagem por referência

- Não modifica o escopo de `main`

```
void swap (int, int);

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

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

- Modifica o escopo de `main`

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

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

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 caracter lido não seja um espaço, digito, '+' 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 ponteiro 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.

Ponteiros e Arrays

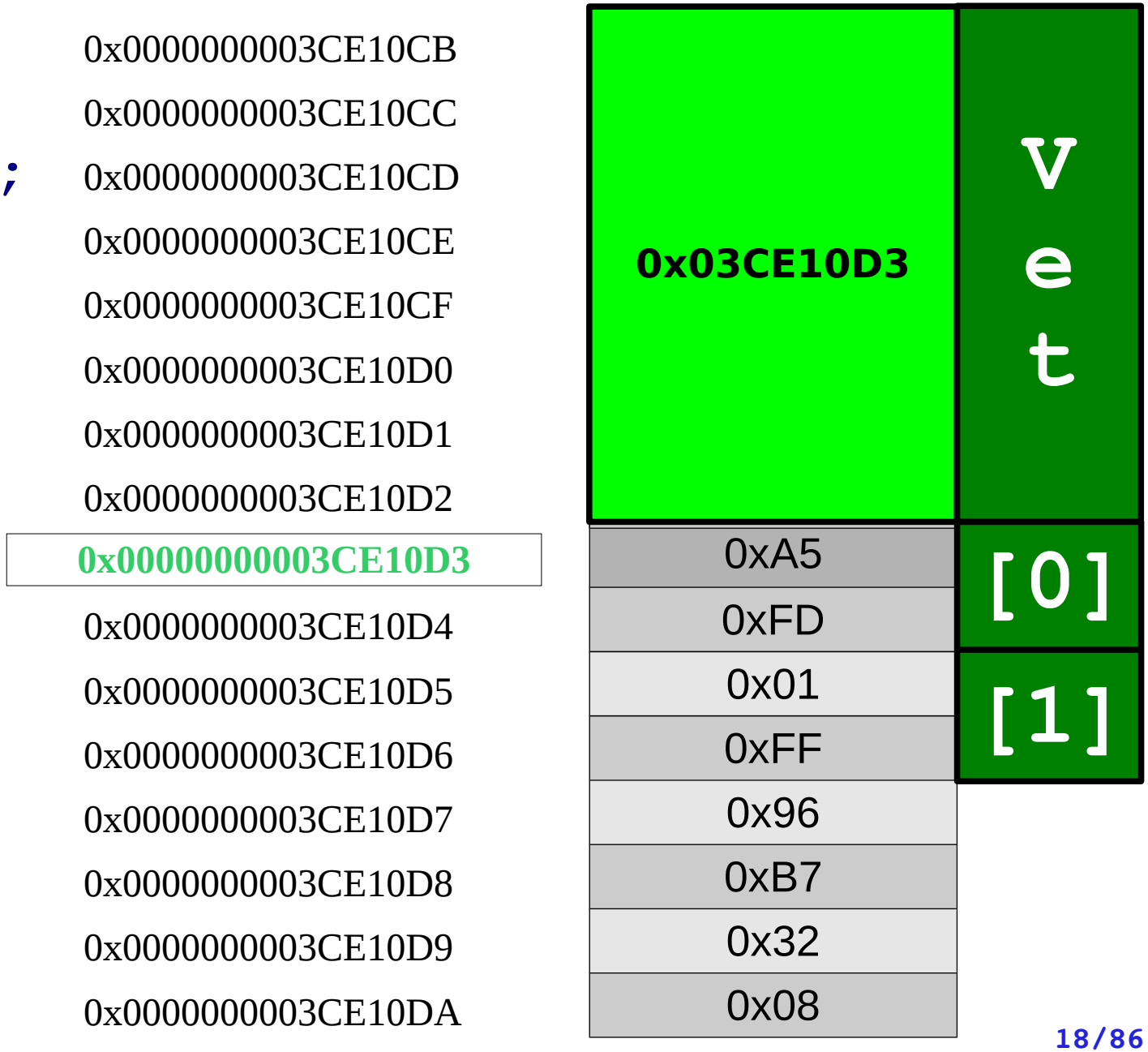
Ponteiros e arrays na linguagem C

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

	Array	**Ponteiro como array**
Declaração	`Tipo MeuArray[]`	`Tipo *ponteiro`
Alocação	Os elementos do array são alocados automaticamente	Somente o ponteiro 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]`	`ponteiro[i]` ou `*(ponteiro+i)`

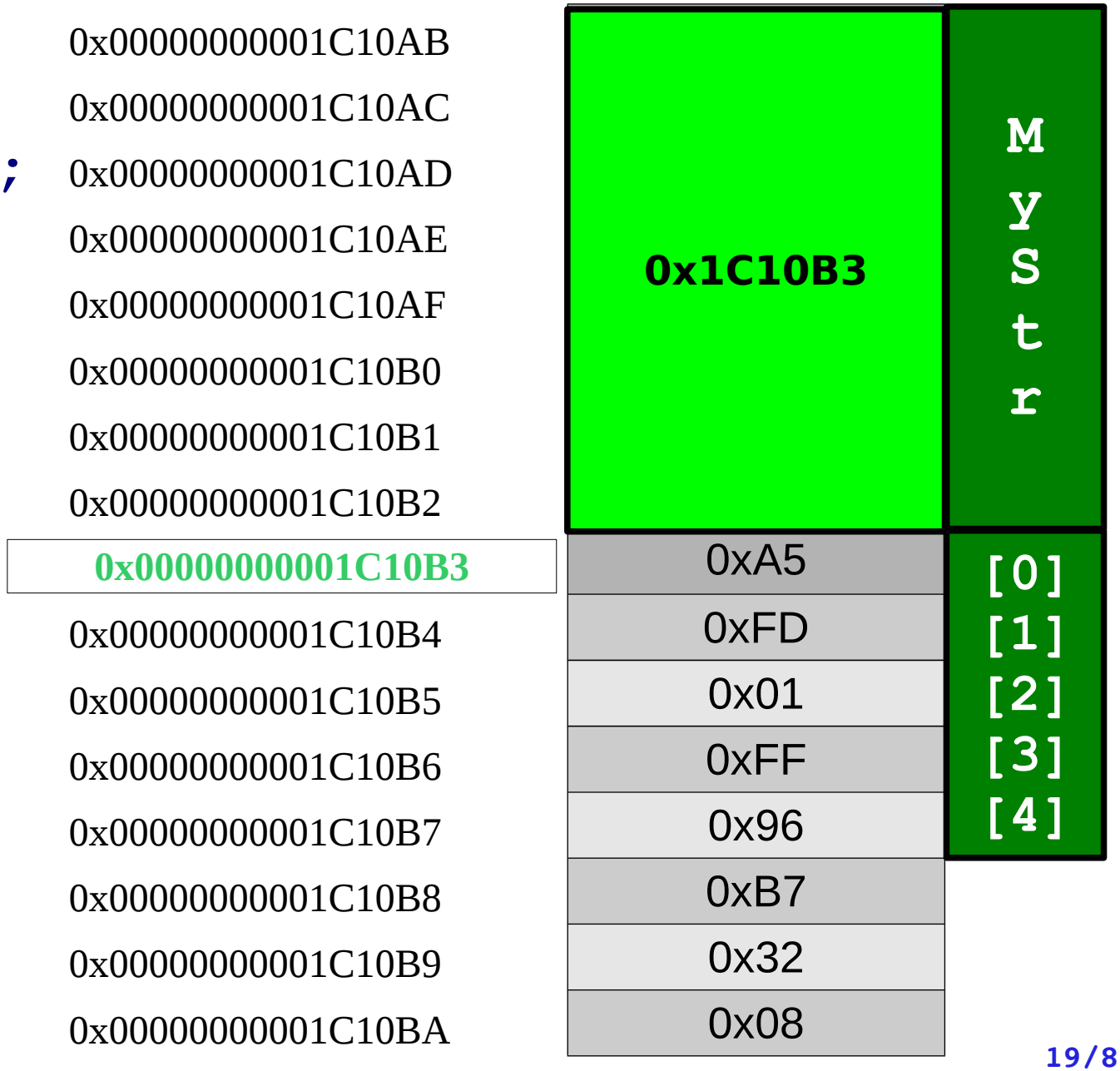
Arrays automáticas

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



Arrays automáticas

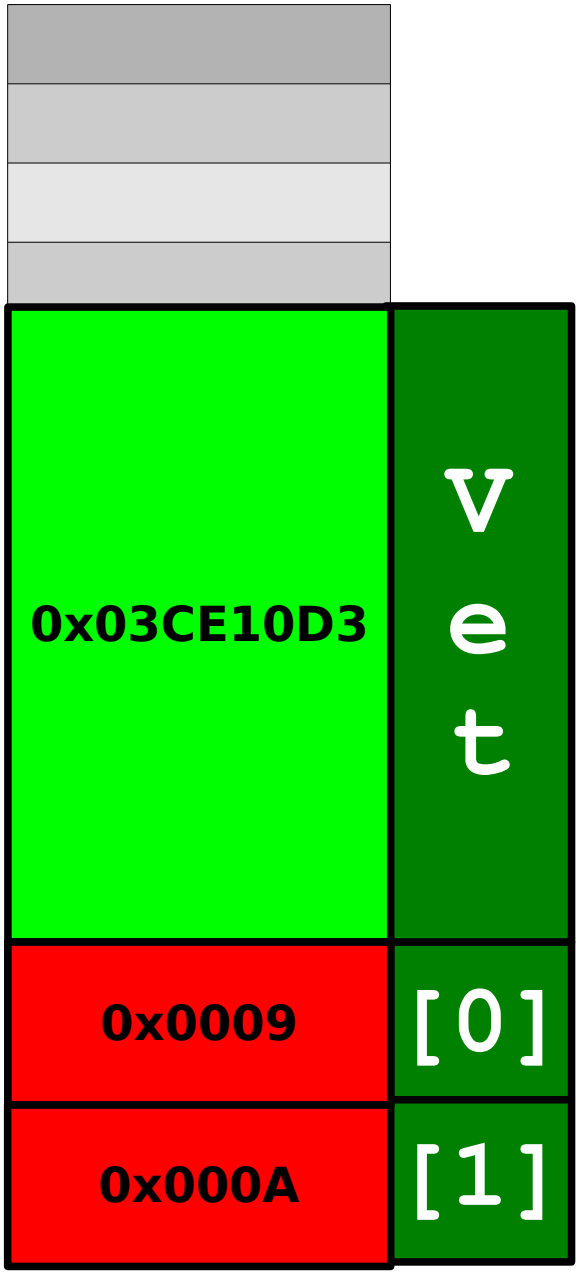
```
main()
{
   char MyStr[5];
}
```

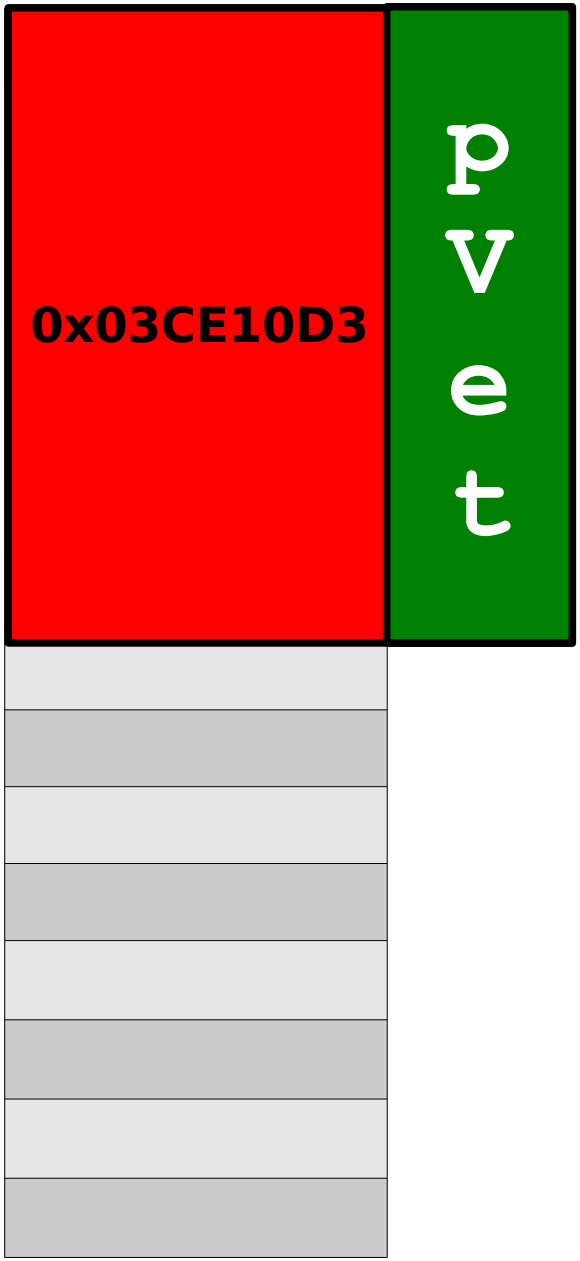


Ponteiros e Arrays

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

0x0000000003CE10D3





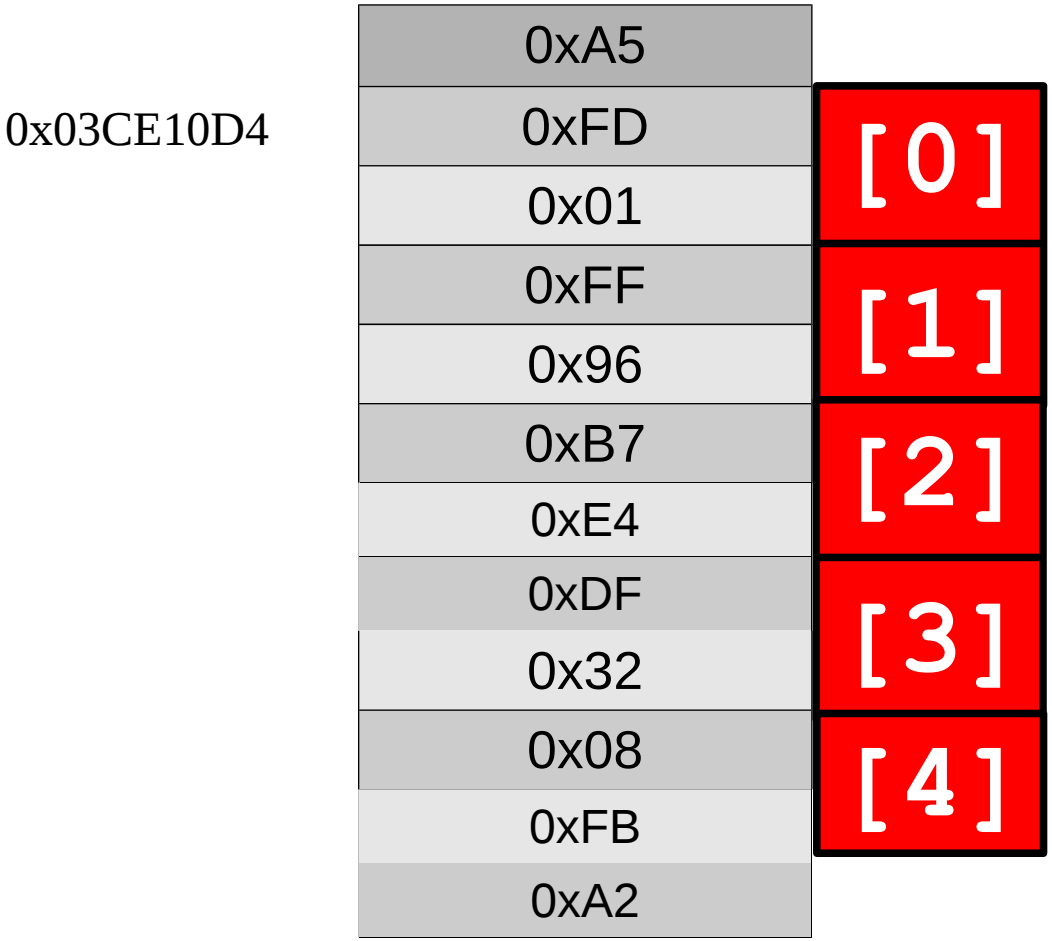
Alocação Dinâmica

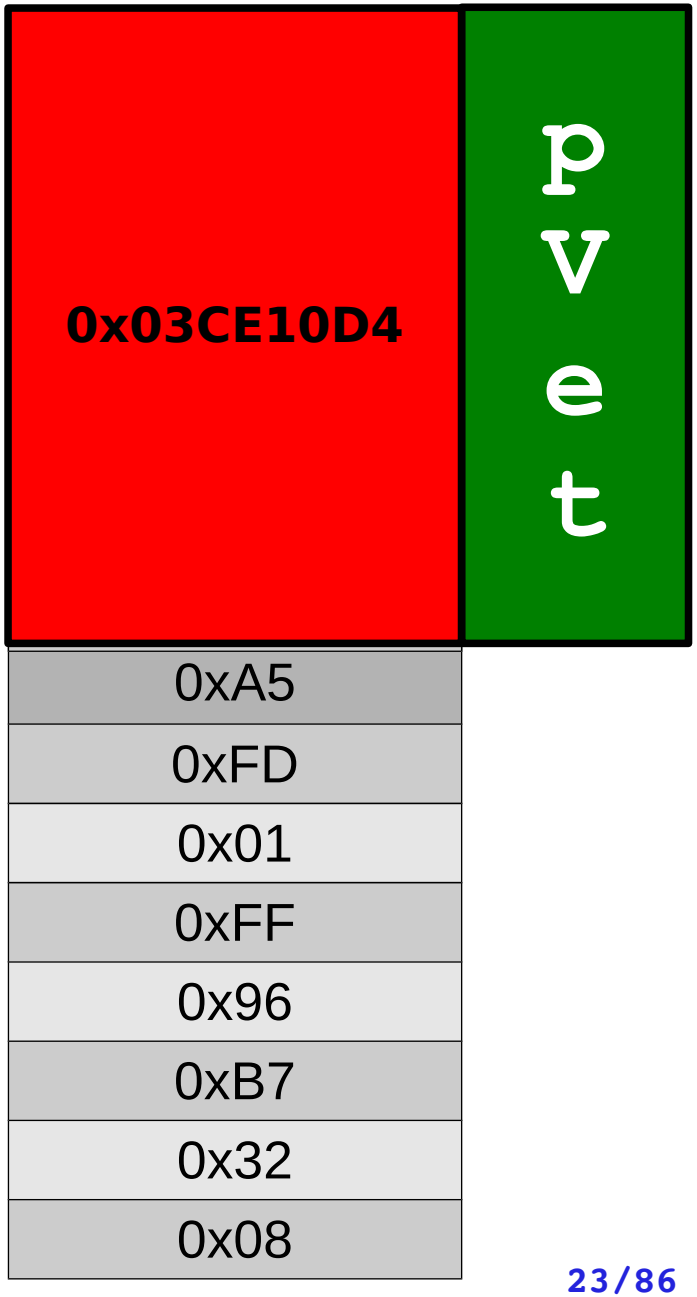
Alocação Dinâmica

- *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

```
main()
{
   short* pVet;
   pVet = (short*) malloc(5*sizeof(short));
}
```





Alocação dinâmica de memória

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

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

Alocação dinâmica de memória

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

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

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

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

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

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

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

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

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, *buf2, *buf3;
  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 Ponteiros

Aritmética de ponteiros

- Ponteiros podem ser submetidos a operações lógicas e aritméticas
- Ponteiros 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`

Manipulação de arrays e Aritmética de Ponteiros

- Como *arrays*

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

- Aritmética de Ponteiros

```
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 ponteiro e pelo operador de declaração de ponteiro *.

```
/* 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 ponteiro e pelo operador de declaração de ponteiro *.

```
/* 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

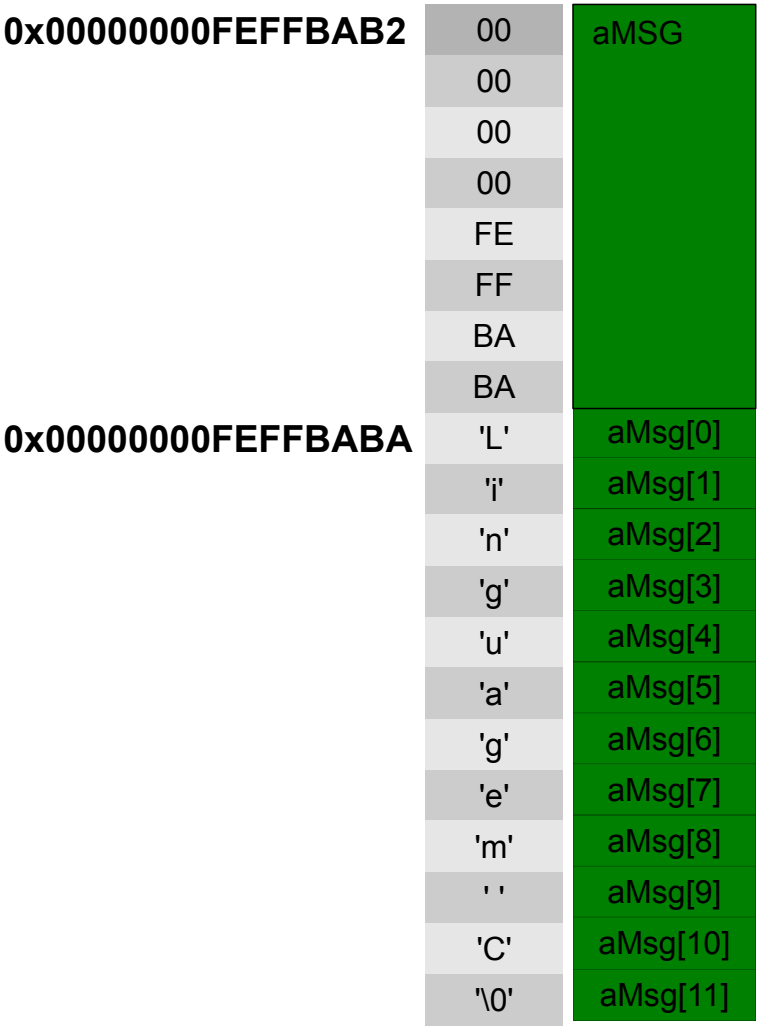
```
#include <stdio.h>
#include <stdlib.h>
float* getvect(int *);
float* sum(float*, float*, int);
void print (float*, int);

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);
}
```

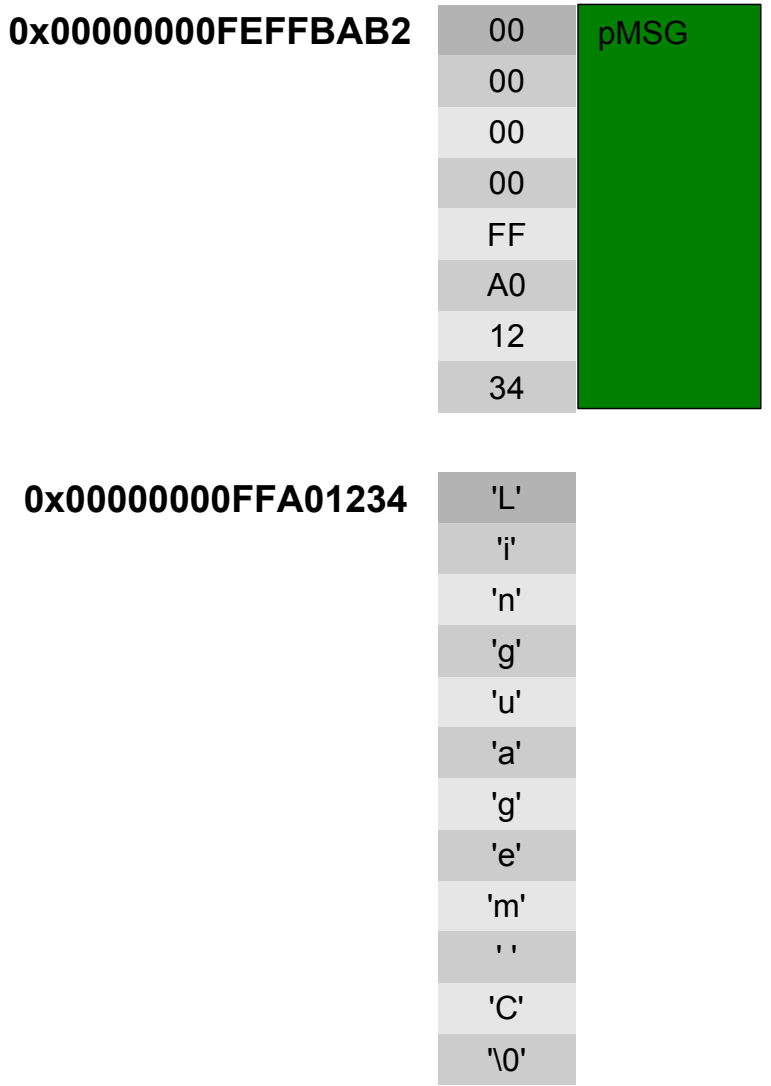
Funções e Ponteiros de char

Arrays, Constantes e Ponteiros de char

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




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



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 ponteiros

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

Exemplos Função strcpy

- Copia t para s → versão aritmética de ponteiros 2.0

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

Exemplos Função strcpy

- Copia t para s → versão aritmética de ponteiros 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 ponteiro 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 ponteiro
- `val = ++*p` → incrementa imediatamente o conteúdo apontado pelo ponteiro

Arrays de Ponteiros e Ponteiros de Ponteiros

Arrays de Ponteiros

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

```
char* StringsArray[10];

int* IntPointerArray [25];

struct Student{
   char Name[255];
   char ID [12];
}
struct Student* StructPointerArray[50];
```

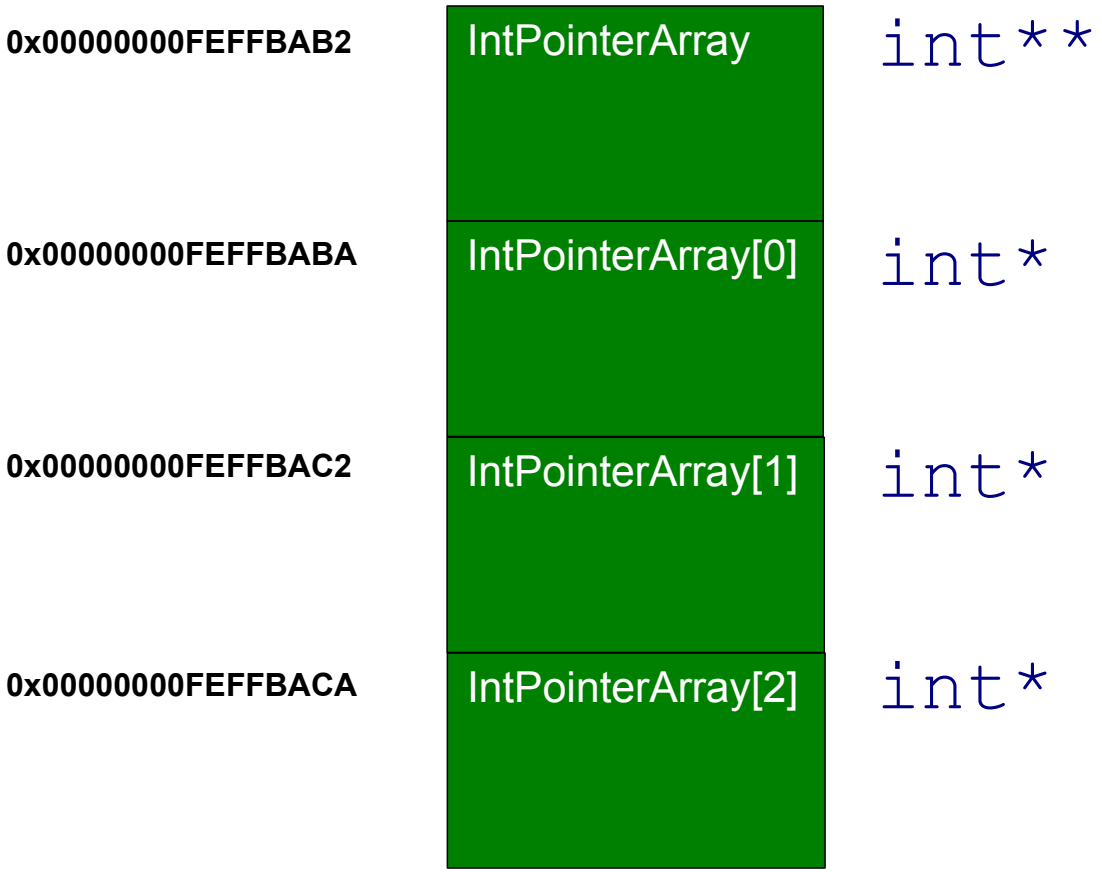
Arrays de Ponteiros

- O cabeça do array é do tipo ponteiro para o tipo do elemento
- Assim, se cada elemento é um ponteiro para `int`, o cabeça do array é do tipo `int **`

```
int* IntPointerArray [25];
```

Arrays de Ponteiros

```
int* IntPointerArray [3];
```



Programa exemplo array de ponteiros

```c
#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 ponteiros

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

Programa exemplo array de ponteiros

```
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

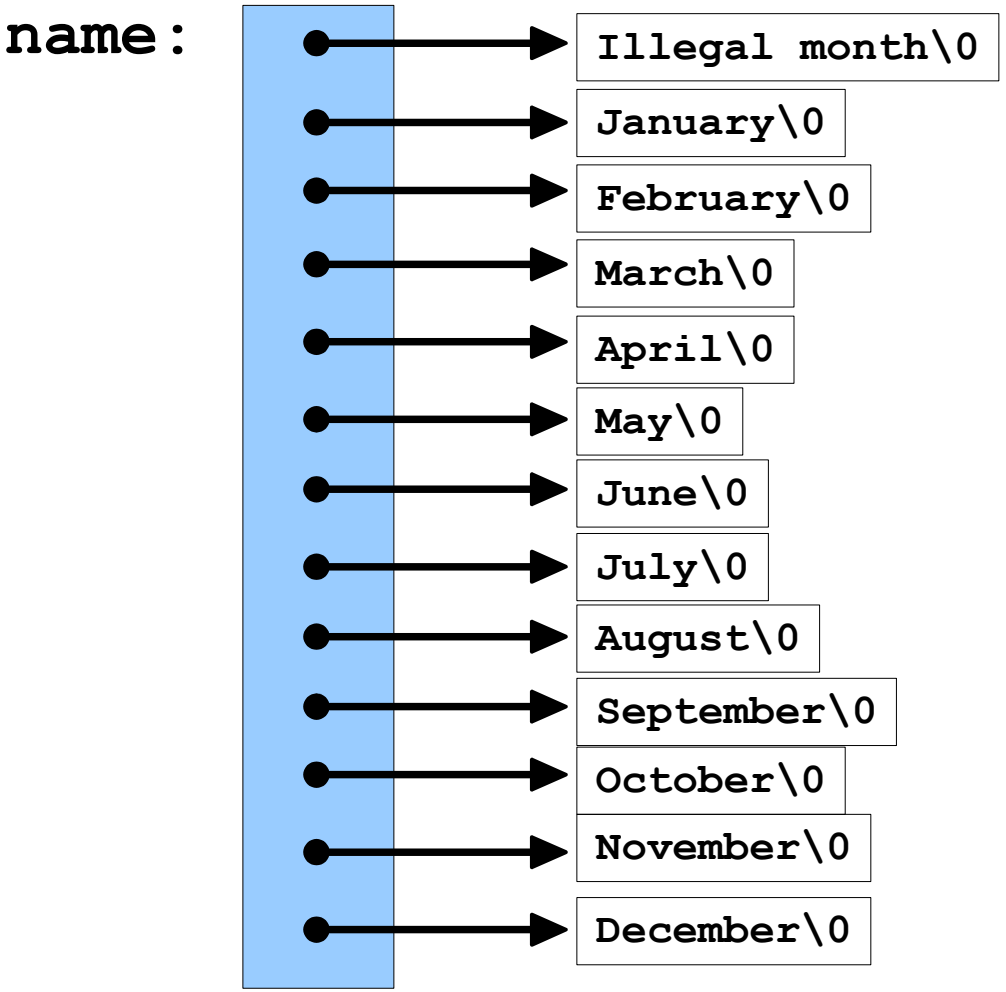
Inicialização de Arrays de Ponteiros

```
/* 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];
}
```

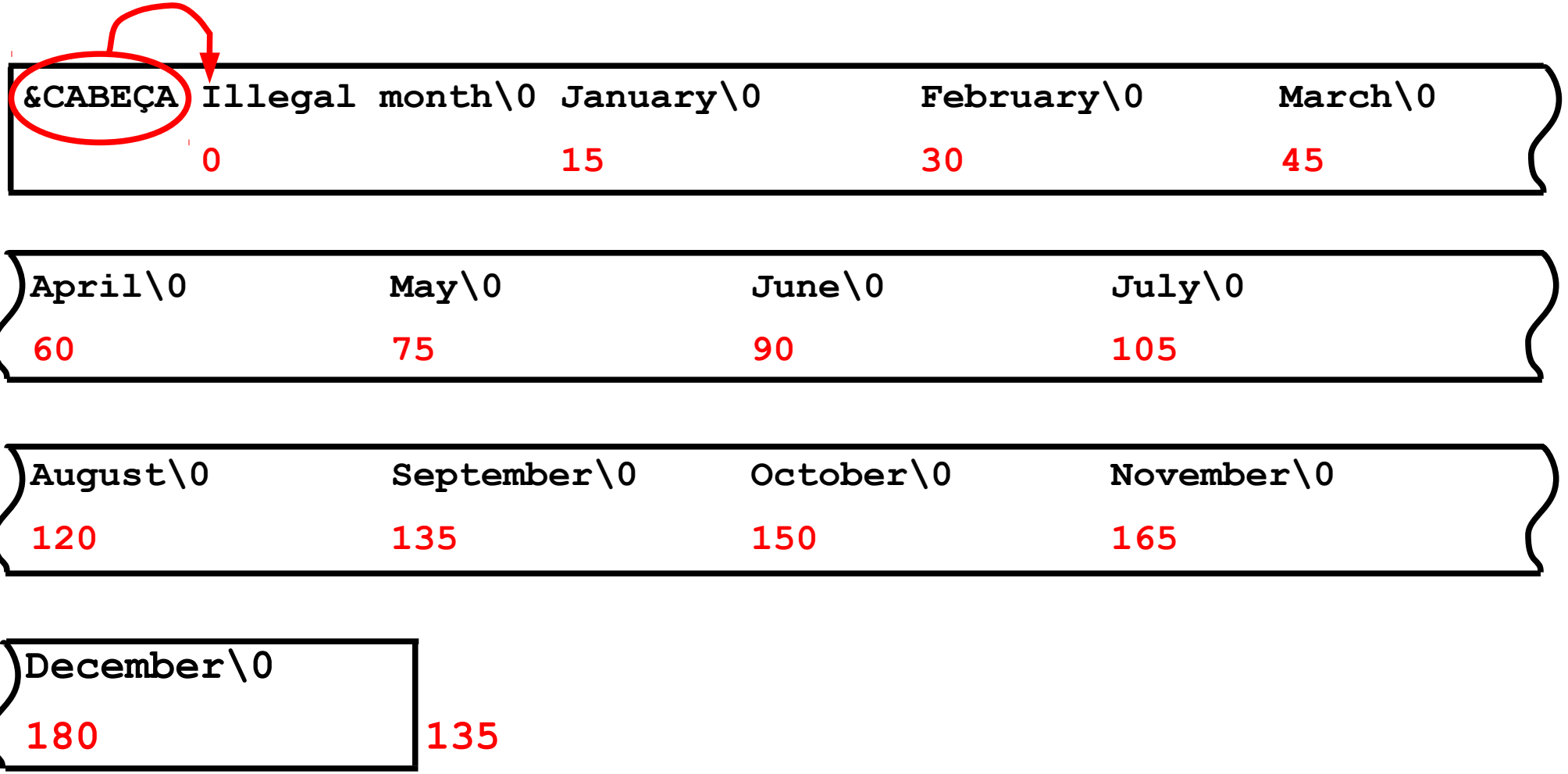
Arrays de Ponteiros

```
char* name[15]={"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" };
```



Exemplos de Tipos Multidimensionais Compatíveis

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

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

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

int* daytab[13]
/* incompatível! Este tipo é um array de 13 posições de
ponteiro para inteiro*/
```

Redução de Array Bidimensional para Ponteiro

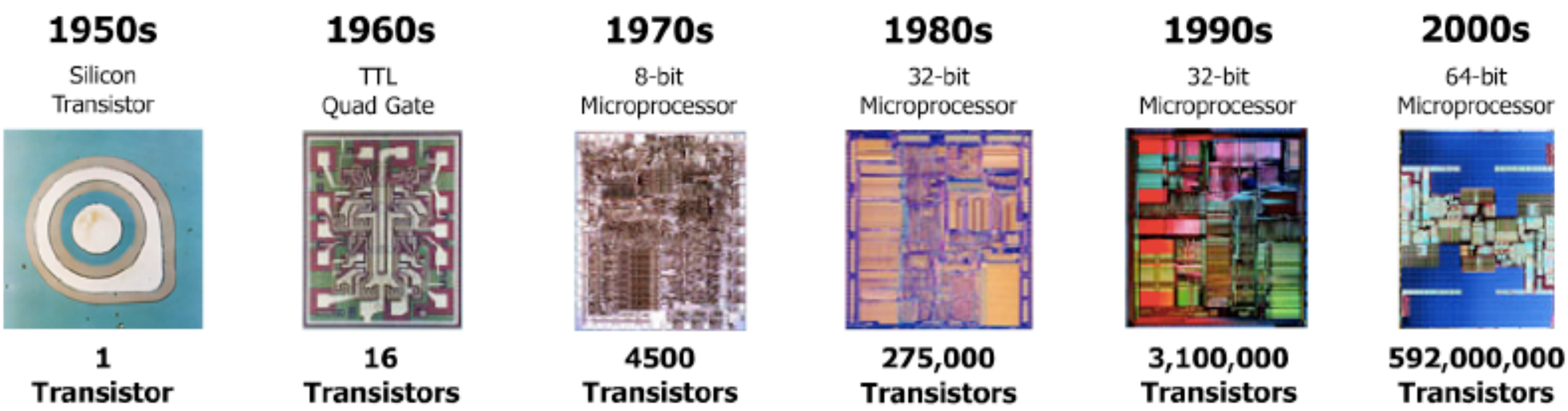
```
void printMat(int * Mat, int rows, int cols)
{
    int i, j;
    printf("{\n");
    for(i=0; i<rows; i++)
    {
       printf("  {");
       for (j=0; j<cols; j++)
          printf("%d%s", *(Mat+cols*i+j), (j<cols-1)?", ":"}\n" );
    }
    printf("}\n");
}

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

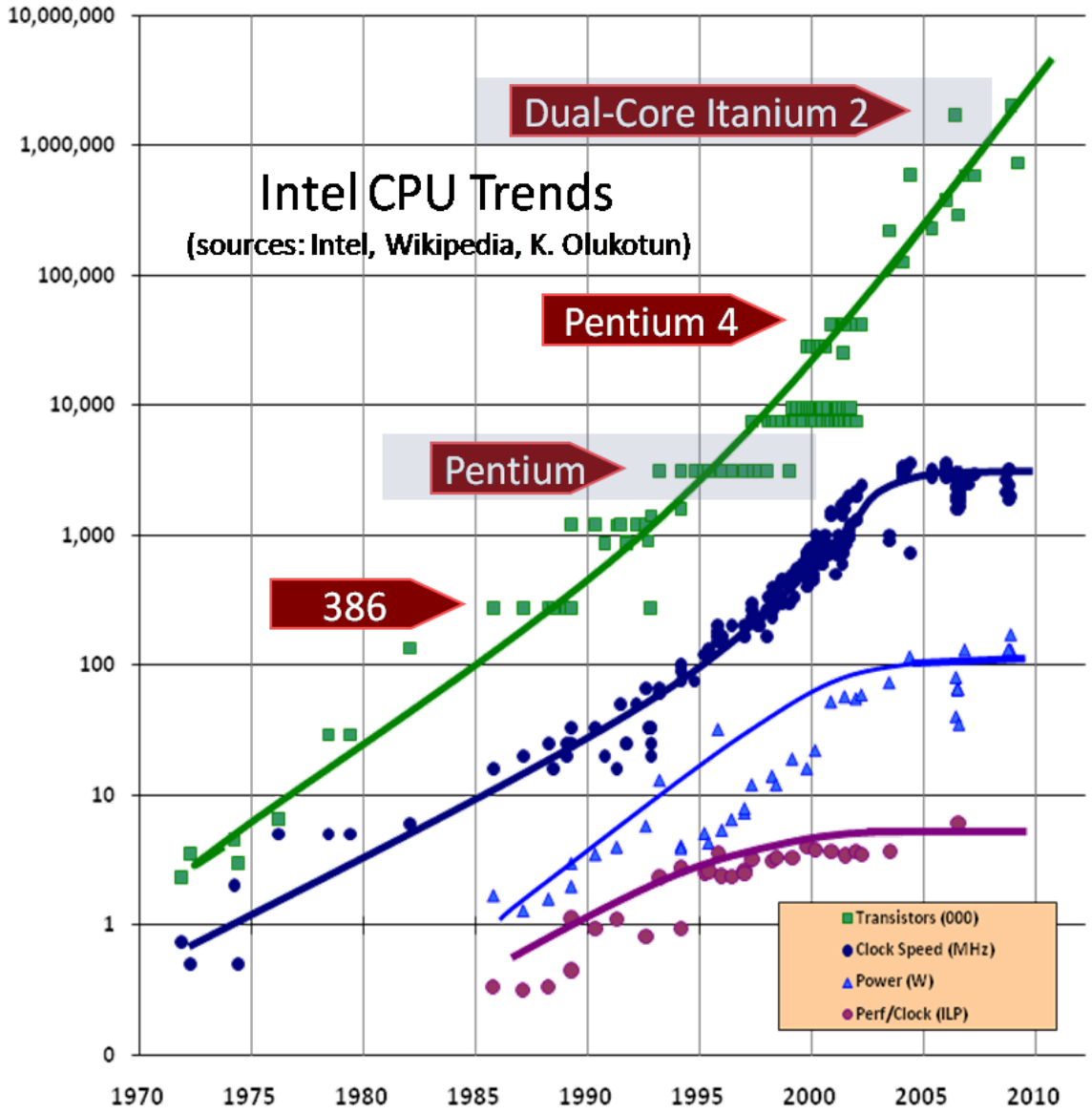
Lei de Moore

- “A densidade de transistores nos circuitos integrados dobra a cada 2 anos.”

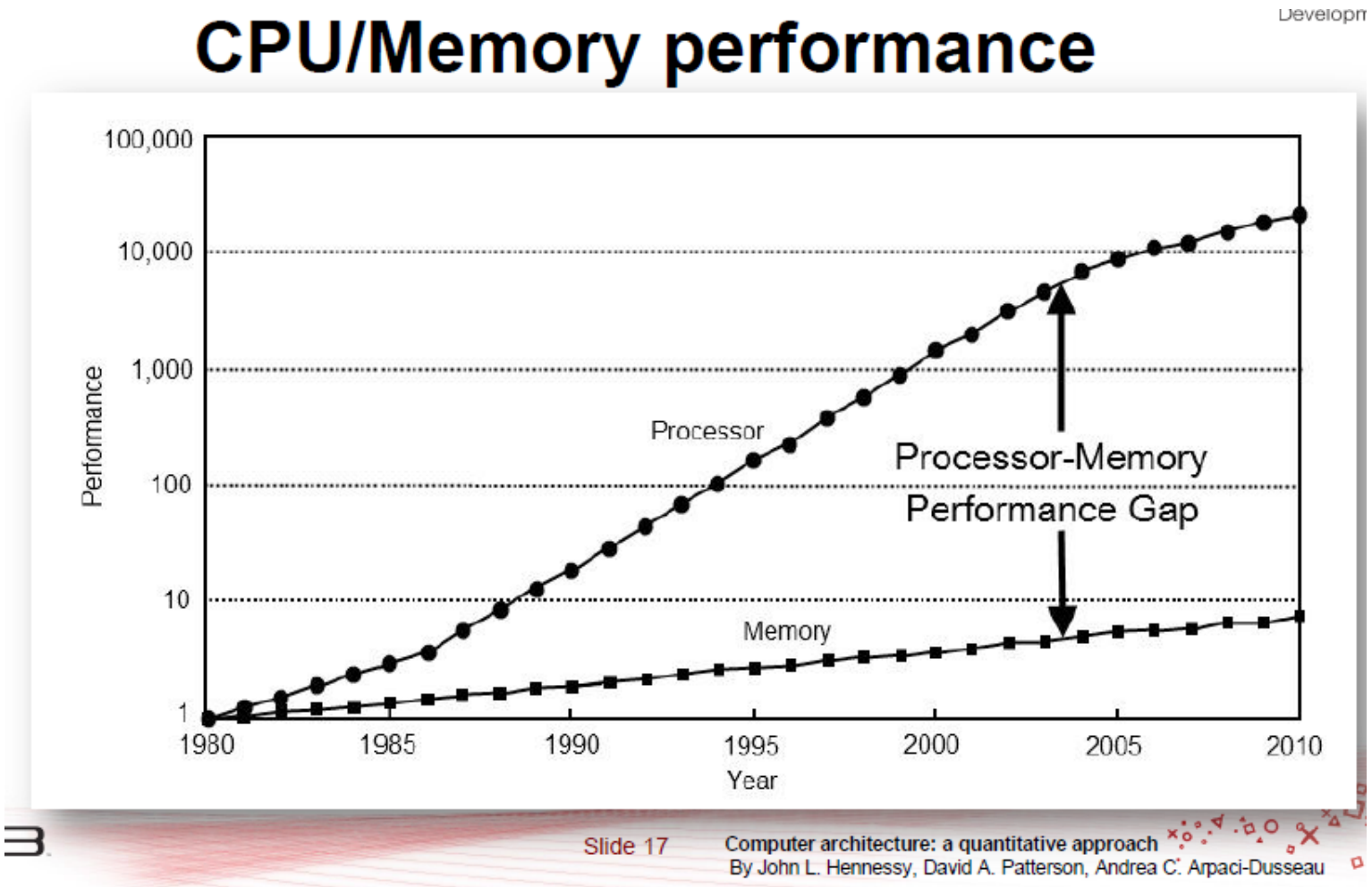


“Moore’s Law: Raising the Bar” (Intel Corporation 2005)

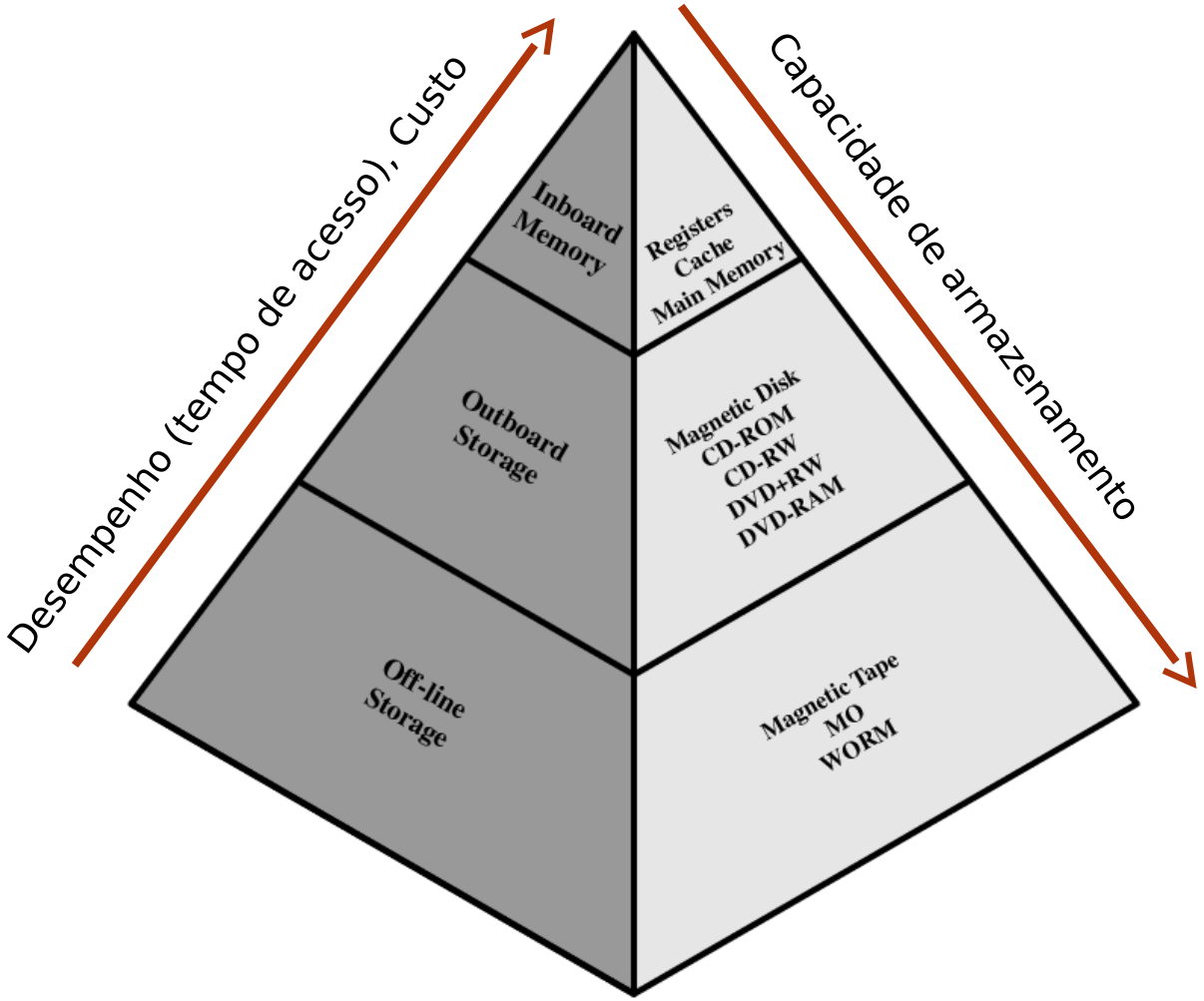
Lei de Moore


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010

Memory Wall


CPU/Memory performance
100,000
10,000
1,000
100
10
1
Performance
Processor
Memory
Processor-Memory
Performance Gap
1980
1985
1990
1995
2000
2005
2010
Year
Slide 17
Computer architecture: a quantitative approach
By John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

Hierarquia de Memória



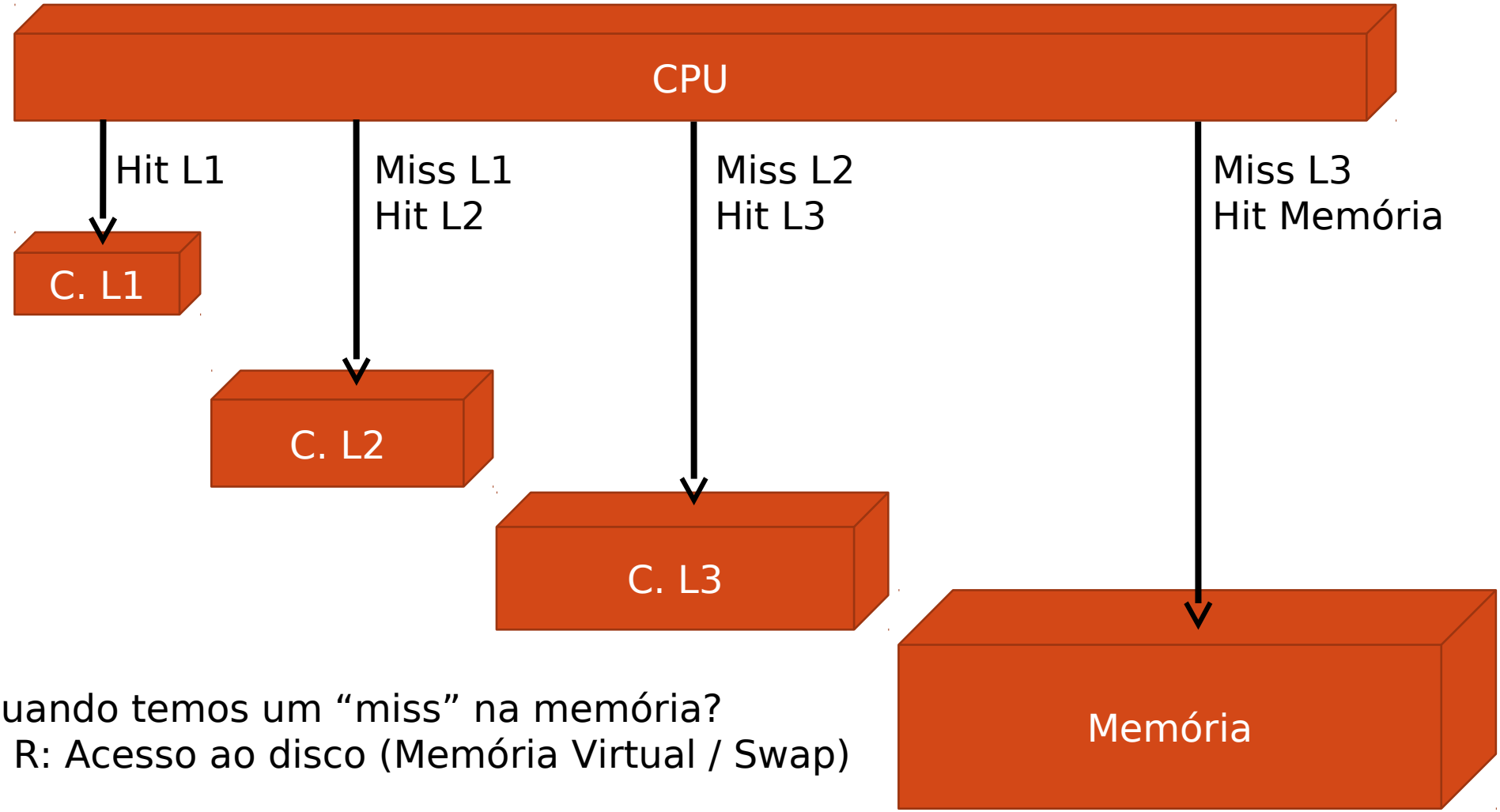
Características dos Níveis da Memória

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

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)

Acesso

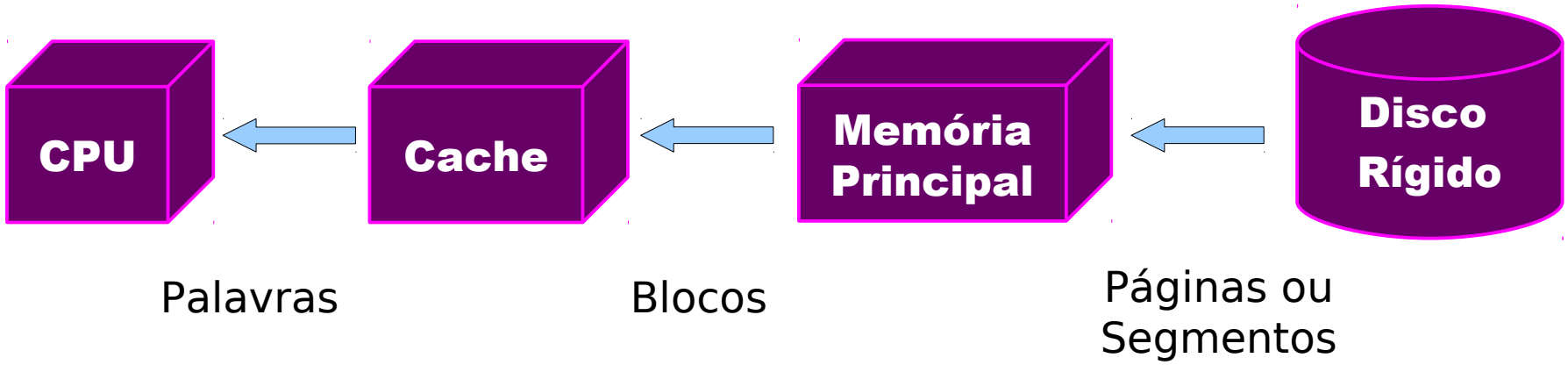


E quando temos um “miss” na memória?

R: Acesso ao disco (Memória Virtual / Swap)

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*colunas+j) ou preencher manualmente o vetor de ponteiros para as linhas	Abstração [][] é direta
Custo da alocação	1 malloc para os dados + 1 malloc para o vetor de ponteiros (opcional)	1 malloc por linha + 1 malloc para o vetor de ponteiros
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: %u\n", ((int)m)/4);
for (i=0; i<10; i++)
{
   printf("&m[%d]: %u\n", i,((int)&(m[i*5]))/4);
   for (j=0; j<5; j++)
      printf("m[%d][%d]: %u\n", i,j,((int)&(m[i*5+j]))/4);
}

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

/*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]: %u\n", i,j,((int)&(m2[i][j]))/4);
}

/*libera m*/
free(m);
free(m2);
```

Ponteiros de funções

Ponteiros de Funções

- Em C, não há analogia entre funções e variáveis, apesar disto, é possível definir ponteiros de funções
- Ponteiros de função permitem:
  - atribuição direta
  - armazenamento em arrays
  - passagem com parâmetro para outras funções

Sintaxe de Ponteiros de Funções

```
<tipo> (NomeDoPonteiro *) (<tipoArg1>, ..., <tipoArgN>)
```

- Ponteiro de função para função como argumento

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


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

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);
}
```

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`] ponteiros para `char`
- `int main(int argc, char* argv[])`

Exemplo

```
/* grep: finds lines that match pattern from 1st arg */
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 ponteiros 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