Linguagem C

Emerson Ribeiro de Mello, Dr.

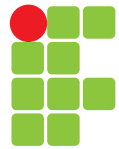
Instituto Federal de Santa Catarina – IFSC
campus São José
mello@sj.ifsc.edu.br

10 de março de 2009

Sumário

- Ambientação a linguagem C
- Conhecendo mais sobre a linguagem C

Sumário



INSTITUTO FEDERAL

Ambientação a linguagem C

Sumário

Conhecendo mais sobre a linguagem C

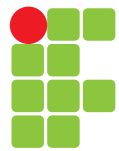
Conhecendo mais sobre a linguagem C

Conhecendo mais sobre a linguagem C

Conhecendo mais sobre a linguagem C

Parte I

Ambientação a linguagem C



Sumário

Visão geral sobre a linguagem C

- A linguagem C derivou da linguagem B, que por sua derivou da linguagem BCPL
  - Foi projetada e implementada para o sistema operacional UNIX
  - Criada entre 1969 e 1973 no *Bell Labs* por Dennis Ritchie

Visão geral sobre a linguagem C

- A linguagem C derivou da linguagem B, que por sua derivou da linguagem BCPL
  - Foi projetada e implementada para o sistema operacional UNIX
  - Criada entre 1969 e 1973 no *Bell Labs* por Dennis Ritchie
- C é uma **linguagem de baixo nível**
  - Lida com o mesmo tipo de objetos que um computador lida
    - caracteres, números e endereços de memória
    - Podem ser combinados e movidos com operadores aritméticos e lógicos implementados por máquinas reais
  - **Assembly** é uma linguagem de baixo nível

Visão geral sobre a linguagem C

- A linguagem C derivou da linguagem B, que por sua derivou da linguagem BCPL
  - Foi projetada e implementada para o sistema operacional UNIX
  - Criada entre 1969 e 1973 no *Bell Labs* por Dennis Ritchie
- C é uma **linguagem de baixo nível**
  - Lida com o mesmo tipo de objetos que um computador lida
    - caracteres, números e endereços de memória
    - Podem ser combinados e movidos com operadores aritméticos e lógicos implementados por máquinas reais
  - **Assembly** é uma linguagem de baixo nível
- C é uma **linguagem de alto nível**
  - Provê suporte ao conceito de tipos de dados
    - Conjunto de valores que uma variável pode armazenar e operações que podem ser realizadas com essa variável

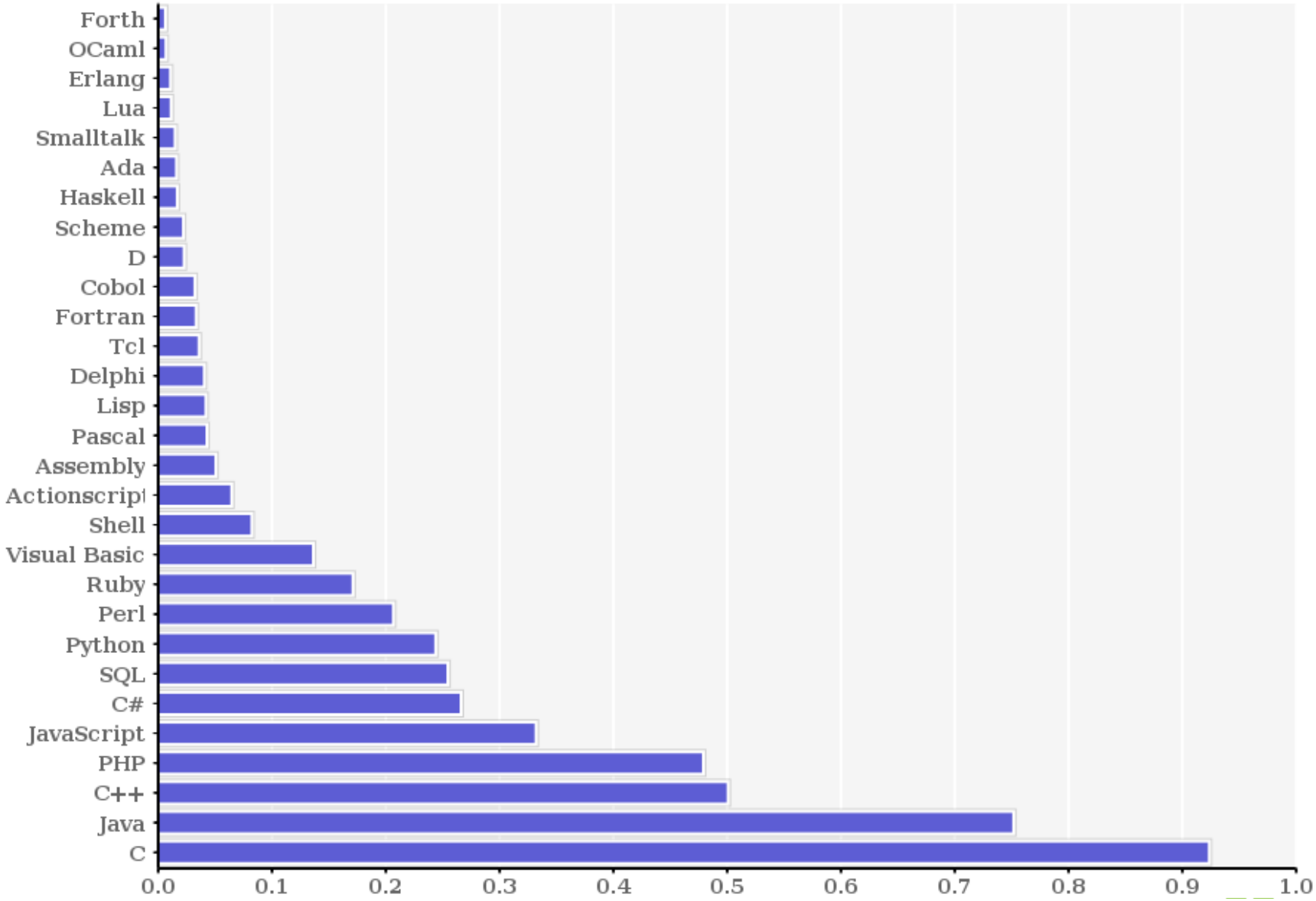
Visão geral sobre a linguagem C

- A linguagem C derivou da linguagem B, que por sua derivou da linguagem BCPL
  - Foi projetada e implementada para o sistema operacional UNIX
  - Criada entre 1969 e 1973 no *Bell Labs* por Dennis Ritchie
- C é uma **linguagem de baixo nível**
  - Lida com o mesmo tipo de objetos que um computador lida
    - caracteres, números e endereços de memória
    - Podem ser combinados e movidos com operadores aritméticos e lógicos implementados por máquinas reais
  - **Assembly** é uma linguagem de baixo nível
- C é uma **linguagem de alto nível**
  - Provê suporte ao conceito de tipos de dados
    - Conjunto de valores que uma variável pode armazenar e operações que podem ser realizadas com essa variável
- C atua tanto no baixo quanto no alto nível

Uma linguagem da década de 70 ainda é popular?

- A linguagem é amplamente utilizada em diferentes tipos de sistemas computacionais
  - Sistemas Operacionais, aplicações *desktop*, aplicações embarcadas
  - Linux, Asterisk, TinyOS, etc.
- O sítio **LangPop.Com** apresenta algumas técnicas para coletar dados que possam ser usados para medir a popularidade de uma linguagem de programação
  - Máquinas de busca (Yahoo, Google)
  - Sítios de venda de livros (Amazon)
  - Sítios de tecnologia (Slashdot)
  - Sítios para hospedagem de projetos (Freshmeat, Google Code)
  - Redes de relacionamento (Delicious)
  - Canais de bate papo no IRC

Popularidade das linguagens: Normalizada



Visão geral sobre a linguagem C

- Trata-se de uma linguagem **portável** e **simples**
  - É independente da arquitetura de máquina
  - Apenas 32 palavras reservadas
- A linguagem se tornou muito popular
  - Diversos grupos criaram compiladores para a linguagem
    - GCC, TinyCC, ICC, BCC, etc
- A diversidade de grupos estava tornando difícil garantir a portabilidade de código pelos diversos compiladores

INSTITUTO FEDERAL

Padronização da linguagem

- Em 1983 o Instituto Norte-Americano de Padrões (ANSI) estabeleceu um comitê para padronizar a linguagem C
  - Em 1989 surge o padrão chamado "Linguagem de Programação C", mais comumente conhecido como "ANSI C" (ou C89)
- Qualquer código escrito em **ANSI C**, sem adicionar qualquer dependência explícita a um *hardware*, irá rodar perfeitamente em qualquer ambiente que possua a implementação C

Padronização da linguagem

- Em 1983 o Instituto Norte-Americano de Padrões (ANSI) estabeleceu um comitê para padronizar a linguagem C
  - Em 1989 surge o padrão chamado "Linguagem de Programação C", mais comumente conhecido como "ANSI C" (ou C89)
- Qualquer código escrito em **ANSI C**, sem adicionar qualquer dependência explícita a um *hardware*, irá rodar perfeitamente em qualquer ambiente que possua a implementação C
- Em 1999 foi lançada uma nova padronização da linguagem: C99
  - Enquanto que o desenvolvimento da linguagem C ficou parado, a linguagem C++ continuou a evoluir
  - O C99 traz para a linguagem C algumas melhorias propostas para o C++
  - Exemplos:
    - Funções em linha, comentários de uma linha (//), etc.

C e C++

- C++ é uma linguagem baseada em C, porém foi projetada para o paradigma da programação orientada a objetos
  - C está para o paradigma da programação estruturada
- C++ contém toda a linguagem C
  - Um código C pode ser compilado por um compilador C++
  - O inverso não é verdade
- Assim, para programar em C++ deve-se conhecer a linguagem C

Visão geral sobre a linguagem C

- Trata-se de uma linguagem estruturada
  - O código pode ser dividido em blocos
- Possui estruturas de controle de fluxo simples (decisão, repetição)
  - Não permite realizar operações paralelas, multiprogramação
- Não provê operações para lidar com objetos complexos como cadeias de caracteres e vetores, nem para realizar operações de entrada e saída (I/O)
- Tudo isso torna a linguagem simples!
  - O que falta é conseguido através das **bibliotecas de funções**
  - Todo compilador C possui uma biblioteca C padrão de funções

Palavras reservadas: 32 no total

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

Biblioteca C padrão

Biblioteca	Descrição
`stdio.h`	Entrada e Saída
`math.h`	Funções matemáticas
`stdlib.h`	Funções de utilidade
`time.h`	Funções de data e hora
`limits.h`	Limites dos tipos de dados
`float.h`	Limites dos tipos de dados
`ctype.h`	Testes de caracteres
`string.h`	Funções para lidar com cadeias de caracteres
`assert.h`	Diagnósticos
`stdarg.h`	Lista de argumentos variável
`setjmp.h`	Chamadas de funções
`signal.h`	Sinais de interrupção

Passos para a geração de um executável em C

1. Criar o programa (código fonte)
   - Através de um editor de textos comum (vi, gedit)
2. Compilar o programa
   - Criação de código objeto (código de máquina) a partir do código fonte
3. Criação de um executável a partir do código objeto
   - Ligação do programa com as funções de biblioteca

Ambiente Integrado de Desenvolvimento – AID

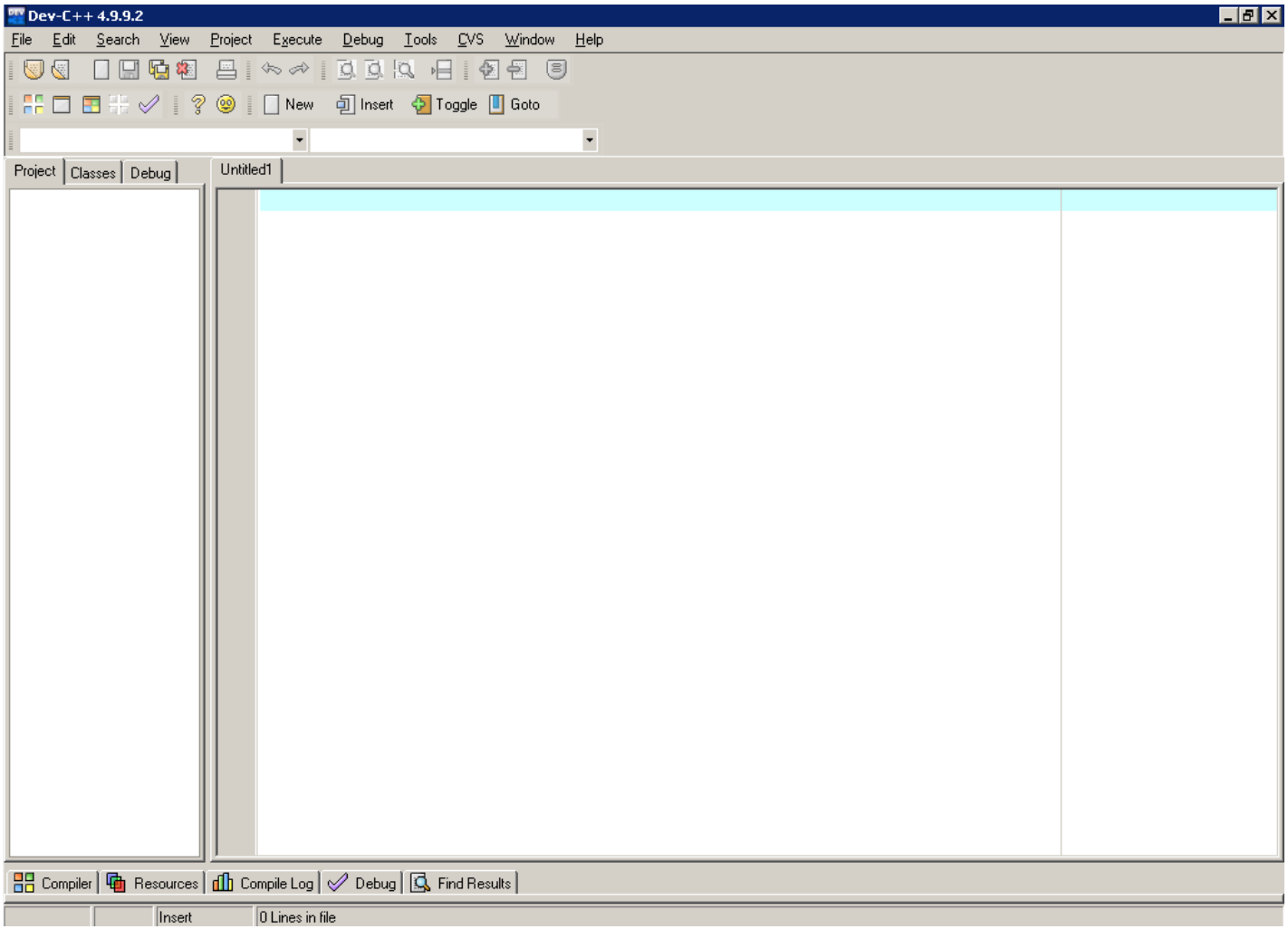
- As três fases para gerar um programa em C podem ser executadas em aplicações separadas ou pode-se fazer uso de um ambiente integrado que reúne todas as aplicações necessárias para editar e compilar um programa em C
- Alguns AID para linguagem C/C++
  - Linux
    - Anjuta, KDevelop, Netbeans, Eclipse
  - Windows
    - DevC++, Netbeans, Eclipse

Dev C++

- Trata-se de um ambiente completo para o desenvolvimento de programas em C/C++
- Faz uso do Mingw[1]
  - GCC (*GNU Compiler Collection*) que foi portado para Windows
- É um *software* livre e foi escrito em Delphi 6
- `http://www.bloodshed.net/dev/index.html`

[1] `http://www.mingw.org`

Dev C++


Dev-C++ 4.9.9.2
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes Debug
Untitled1
Compiler Resources Compile Log Debug Find Results
Insert
0 Lines in file

Sumário

Sumário

Estrutura de um programa em C

- Todo programa em C é composto por uma ou mais funções
- Função **main** deve estar presente em todo programa C, mas isso não implica em todo arquivo fonte C
  - Projetos grandes costumam ser espalhados por diversos arquivos de código fonte, visando facilitar a manutenção

Dica

Em um programa C bem escrito, a função **main** contém somente a essência do que o programa faz de fato

Olá mundo em C

```c
#include<stdio.h>

int main(void)
{
   printf("Ola mundo!");
   return 0;
}
```

Tipos de dados

- A linguagem C possui **5** tipos básicos de dados, são chamados de básicos pois a linguagem permite realizar operações aritméticas e lógicas com estes. São estes:
  - caractere, inteiro, ponto flutuante, ponto flutuante de precisão dupla e sem valor
  - **char**, **int**, **float**, **double** e **void**

Tipos de dados

- A linguagem C possui **5** tipos básicos de dados, são chamados de básicos pois a linguagem permite realizar operações aritméticas e lógicas com estes. São estes:
  - caractere, inteiro, ponto flutuante, ponto flutuante de precisão dupla e sem valor
  - **char**, **int**, **float**, **double** e **void**
- O tipo do processador e a implementação do compilador influenciam o tamanho e faixa que esses tipos podem representar
  - O padrão ANSI C só estipula apenas a faixa mínima de cada tipo de dado e **não** o tamanho ocupado por este
  - 1 caractere ocupa 1 byte (8 bits), já um inteiro geralmente corresponde ao tamanho natural de uma palavra do computador em questão
  - A faixa dos tipos **float** e **double** é dada em dígitos de precisão

Tipos de dados pelo padrão ANSI

- A faixa dos tipos pode ser alterada pelos modificadores **long**, **short**, **signed** e **unsigned**

Tipo	**Faixa mínima**	**Tamanho mínimo**
char	-127 a 127	8
unsigned char	0 a 255	8
int	-32.767 a 32.767	16
unsigned int	0 a 65.535	16
long int	-2.147.483.648 a 2.147.483.648	32
unsigned long int	0 a 4.294.967.295	32
float	6 dígitos de precisão	32
double	10 dígitos de precisão	64
long double	10 dígitos de precisão	80

Antes de escrevermos alguns códigos

- Todo programa em C é composto por uma ou mais funções
- As funções constituem um **bloco** que agregam **instruções**
  - Um bloco é delimitado pelos símbolos { e }
  - As instruções são encerradas pelo símbolo ;
  - `printf(‘‘Ola mundo’’);` é uma instrução

```
1 int main(void)
2 {
3    printf("Ola mundo!");
4    return 0;
5 }
```

Antes de escrevermos alguns códigos

- É possível inserir textos (**comentários**) no código fonte como forma de documentação. O compilador irá ignorá-los
  - Trata-se de uma boa prática de programação
  - São delimitados pelos símbolos /* e */

```
1  /* Arquivo: comentarios.c
2   * Autor:  Emerson R. de Mello
3   * Data:   2009-03-10
4   */
5
6  /* Funcao principal */
7  int main(void)
8  {
9     printf("Ola mundo");
10
11    /* Enviando um valor de retorno da funcao */
12    return 0;
13 }
```

Antes de escrevermos alguns códigos

- Uma **variável** consiste em uma posição nomeada de memória, que é usada para guardar algum valor (tipo do dado)
- Em C todas as variáveis devem ser declaradas antes de serem usadas (sempre no início de um **bloco**) e a forma geral para sua declaração é:
  - `tipo lista_de_variáveis;`
  - Exemplos:
    - `char sexo;`
    - `int dia, mes, ano;`

Antes de escrevermos alguns códigos

- Uma **variável** consiste em uma posição nomeada de memória, que é usada para guardar algum valor (tipo do dado)
- Em C todas as variáveis devem ser declaradas antes de serem usadas (sempre no início de um **bloco**) e a forma geral para sua declaração é:
  - `tipo lista_de_variáveis;`
  - Exemplos:
    - `char sexo;`
    - `int dia, mes, ano;`
- Existem algumas restrições para nomear uma variável
  - Os nomes devem ser formados por letras e dígitos, sendo que o primeiro caractere deve ser uma letra. O caractere _ equivale a uma letra
  - Letras minúsculas são distintas das maiúsculas, assim **CURSO** é diferente de **curso**
  - Não se pode usar as palavras reservadas para nomear uma variável

Operadores aritméticos e de atribuição

Descrição	Operador	Exemplo
soma	+	1 + 2
subtração	-	1 - 2
divisão	/	1 / 2
multiplicação	*	1 * 2
resto da divisão	%	1 % 2
atribuição	=	a = 10
Outras combinações	+=, -=, *=, /=, %=	a+=1
Incremento	++	a++
Decremento	--	a--

```
1 int a = 1, b = 2, c, d;
2
3 c = a + b;
4 d = a + 10;
5 c++;
```

Operadores relacionais e lógicos

- Operadores relacionais

Descrição	Operador	Exemplo
Igual	==	a == b
Diferente	!=	a != b
Maior	>	a > b
Menor	<	a < b
Maior ou igual	>=	a >= b
Menor ou igual	<=	a <= b

- Operadores lógicos

Descrição	Operador	Exemplo
NÃO (NOT)	!	!a
E (AND)	&&	a && b
OU (OR)	||	a || b

Tabela verdade

- Operador **E** (AND)

A	B	(A E B)
0	0	0
0	1	0
1	0	0
1	1	1

- Operador **OU** (OR)

A	B	(A OU B)
0	0	0
0	1	1
1	0	1
1	1	1

Convenção de codificação em C

- O compilador ignora comentários, espaços em branco e quebras de linha
  - Só verifica se a sintaxe está correta (blocos, instruções, atribuição de variáveis, etc)

Convenção de codificação em C

- O compilador ignora comentários, espaços em branco e quebras de linha
  - Só verifica se a sintaxe está correta (blocos, instruções, atribuição de variáveis, etc)

```
1 int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
2 o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Convenção de codificação em C

- O compilador ignora comentários, espaços em branco e quebras de linha
  - Só verifica se a sintaxe está correta (blocos, instruções, atribuição de variáveis, etc)

```
1 int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
2 o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

- A organização do código facilita o compreendimento deste por nós humanos
  - Em projetos grandes tal organização é essencial, pois várias pessoas irão lidar com o código
  - Em projetos pequenos é bem desejável, pois torna a leitura fácil mesmo após ter passado muito tempo desde que o código foi escrito

Convenção de codificação em C

- Os comentários são essenciais para facilitar o compreendimento do código, contudo só isto não é suficiente
- A **convenção de código** ou **estilo de codificação** garante uma maneira uniforme de escrever o código
  - Existem diversas convenções e não existe uma mandatória
  - Cada empresa está livre para escolher uma das convenções já existentes, adaptá-la as suas necessidades ou mesmo criar uma própria

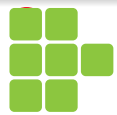
Convenção de codificação em C

- Os comentários são essenciais para facilitar o compreendimento do código, contudo só isto não é suficiente
- A **convenção de código** ou **estilo de codificação** garante uma maneira uniforme de escrever o código
  - Existem diversas convenções e não existe uma mandatória
  - Cada empresa está livre para escolher uma das convenções já existentes, adaptá-la as suas necessidades ou mesmo criar uma própria

Nota:

- Seguiremos uma convenção de codificação específica para este curso
  - Baseada em *Indian Hill C Style and Coding Standards*[a]

[a] `http://www.chris-lott.org/resources/cstyle/indhill-cstyle.pdf`



Um programa simples

```
1  /* Nome do arquivo_fonte.c */
2  #include<nome da biblioteca>
3
4  /* Funcao principal */
5  int main(void)
6  {
7     /* As variaveis devem ser declaradas no inicio da funcao */
8     int ano = 2009; /* armazena o ano atual */
9
10    /* imprimindo o valor de uma variavel */
11    printf("O ano atual e': %d", ano);
12
13    /* Enviando um valor de retorno da funcao */
14    return 0;
15 }
```

Escrevendo o primeiro programa

Calculadora

Desenvolva um programa que leia dois números inteiros e exiba o resultado da soma destes.

Calculadora

Desenvolva um programa que leia dois números inteiros e exiba o resultado da soma destes.

- Temos operações de entrada e saída (leitura de dados e exibição de resultado)
- A biblioteca **stdio.h** provê as funções necessárias
  - A função **printf** é usada para saída de dados
  - A função **scanf** é usada para entrada de dados

Pseudocódigo para o programa

```
1  variaveis:
2     a, b, resultado :inteiro
3  inicio
4     Escrever "Entre como o primeiro numero: "
5     Ler a
6     Escrever "Entre como o segundo numero: "
7     Ler b
8     resultado := a + b
9     Escrever "A soma e': ", resultado
10 fim
```

Sumário

Estruturas de decisão

- Um programa C quando invocado, realiza todas as **instruções** presentes na função **main**
- Estruturas de decisão realizam um desvio do fluxo de processamento, permitindo ao programa apresentar saídas diferentes de acordo com o resultado do teste lógico
- Testes lógicos só assumem dois valores: **verdadeiro** ou **falso**
  - Em C o **verdadeiro** é representado por **1** e o **falso** por **0**

```
1 SE ( teste logico )
2    ENTAO
3       instrucoes...
4    SENAO
5       instrucoes...
6 FIM_SE
```

Estruturas de decisão

```
1 if ( sexo == 'M' ){
2    printf("E' necessario apresentar quitacao militar");
3 }
```

Estruturas de decisão

```
1 if ( sexo == 'M' ){
2    printf("E' necessario apresentar quitacao militar");
3 }
```

```
1 if ( a == 1 ){
2    printf("a e' igual a 1");
3 }else{
4    printf("a nao e' igual a 1");
5 }
```

INSTITUTO FEDERAL

Estruturas de decisão

```c
if ( sexo == 'M' ){
   printf("E' necessario apresentar quitacao militar");
}
```

```c
if ( a == 1 ){
   printf("a e' igual a 1");
}else{
   printf("a nao e' igual a 1");
}
```

```c
if ( a == 1 ){
   printf("a e' igual a 1");
}else if ( a < 1 ){
   printf("a e' menor que 1");
} else {
   printf("a e' maior que 1");
}
```

Estruturas de decisão

- Estruturas de decisão aninhadas

```
1  if ( sexo == 'M' ){
2     if ( idade >= 18 ){
3        printf("E' necessario apresentar quitacao militar");
4     }
5  }
```

Estruturas de decisão

- Estruturas de decisão aninhadas

```
1 if ( sexo == 'M' ){
2    if ( idade >= 18 ){
3       printf("E' necessario apresentar quitacao militar");
4    }
5 }
```

- Fazendo uso de operadores lógicos

```
1 if ( ( sexo == 'M' ) && ( idade >= 18 ) ){
2    printf("E' necessario apresentar quitacao militar");
3 }
```

Exercícios

1. Desenvolva um programa que leia a categoria da habilitação de um motorista e informe o tipo de veículo que ele está apto a dirigir.
   - A - moto, B - carro, C - caminhão, D - ônibus e E - carreta.
   - Se a categoria não for qualquer acima, exibir uma mensagem de erro

2. Desenvolva um programa que leia a idade de uma pessoa e informe se ela está apta a obter um título de eleitor (acima de 16 anos) e se está apta a obter uma carteira de habilitação (acima de 18 anos). Exiba a mensagem "você é muito novo" somente se esta pessoa não puder obter o título e nem a habilitação.

Estruturas de decisão

- A instrução `if...else` é usada para a tomada de decisões, contudo diante de múltiplas decisões (como o exercício da categoria da habilitação) seu uso é um pouco incômodo
  - Diversas sequências `if...else if...else if...`

Estruturas de decisão

- A instrução `if...else` é usada para a tomada de decisões, contudo diante de múltiplas decisões (como o exercício da categoria da habilitação) seu uso é um pouco incômodo
  - Diversas sequências `if...else if...else if...`
- A instrução **switch** possibilita testes com múltiplos casos
  - Cada **caso** é rotulado por uma ou mais constantes inteiras (um caractere é na prática uma constante inteira)
  - Os casos devem ser únicos
  - O caso **default** é executado se nenhum outro caso for satisfeito
- O **switch** difere do **if**, pois o primeiro só pode testar igualdade, já o **if** pode testar uma operação lógica ou relacional

Switch. . . case

```
1  switch ( expressao ) {
2     case constante_inteira:
3             instrucoes...
4
5     case constante_inteira:
6             instrucoes...
7     .
8     .
9     .
10    default:
11          instrucoes...
12 }
```

Switch. . . case

```
1  switch ( voltagem ){
2     case 110:
3        printf("operando com 110 Volts");
4        break;
5     case 220:
6        printf("operando com 220 Volts");
7        break;
8     default:
9        printf("Voltagem fora das especificacoes");
10 }
```

Switch. . . case

```
1  switch ( caractere ){
2     case '0': case '1': case '2': case '3': case '4':
3     case '5': case '6': case '7': case '8': case '9':
4        numeros++;
5        break;
6     case '+':
7     case '-':
8     case '/':
9     case '*':
10       operadores++;
11       break;
12    default:
13       outros++;
14 }
```

Operador ternário

- O operador ternário pode substituir a instrução `if...else`, sendo este uma forma mais compacta para realizar uma decisão

```
1  /* ( teste ) ? verdade : falso */
2
3  res = ( a > 1 ) ? 100 : 200;
4
5  /* equivalente com if-else */
6  if ( a > 1 ) {
7     res = 100;
8  }else {
9     res = 200;
10 }
```

Sumário

Estruturas de repetição

- As estruturas de repetição ou laços, permitem que um conjunto de instruções seja executado até que uma certa condição ocorra
- A condição de término do laço pode ser pré-definida ou pode estar em aberto
  - A instrução **for** tem a condição de pré-definida
  - As instruções **while** e **do. . . while** deixam a condição em aberto
- A instrução **break** pode ser usada para sair de um laço sem que seja necessário atingir a condição deste
- A instrução **continue** faz com que a linha de execução volte para o início do bloco do laço sem executar as instruções que estão após o **continue**

Estruturas de repetição

- **while**

```
1  int a = 0;
2
3  while ( a < 10 ){
4     printf("%d\n", a);
5     a++;
6  }
```

- **do. . . while**

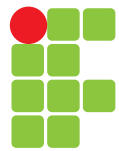
```
1  int a = 0;
2
3  do{
4     printf("%d\n", a);
5     a++;
6  } while ( a < 10 );
```

Estruturas de repetição

```
1  /* for ( iniciar variaveis; condicao de parada ; incremento) */
2
3  int a;
4
5  for( a = 0; a < 10; a++){
6     printf("%d\n", a);
7  }
```

Parte II

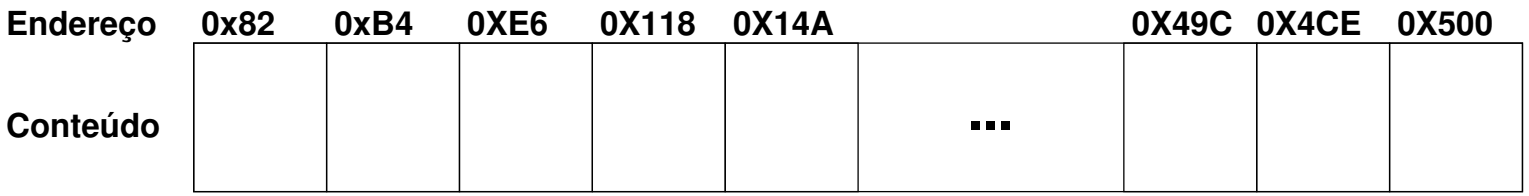
Conhecendo mais sobre a linguagem C



Sumário

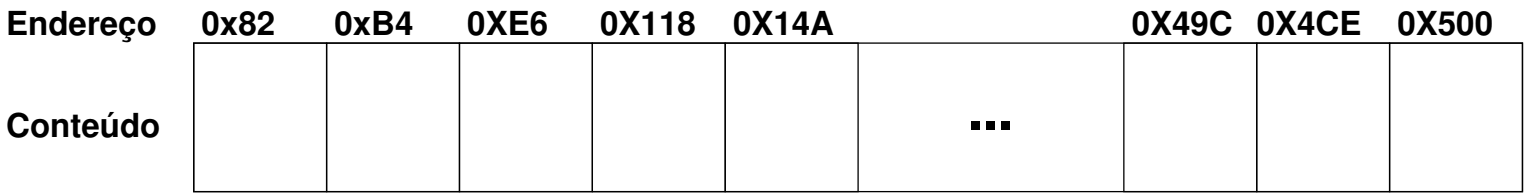
Variáveis

Qual a relação das variáveis com a memória do computador?



Variáveis

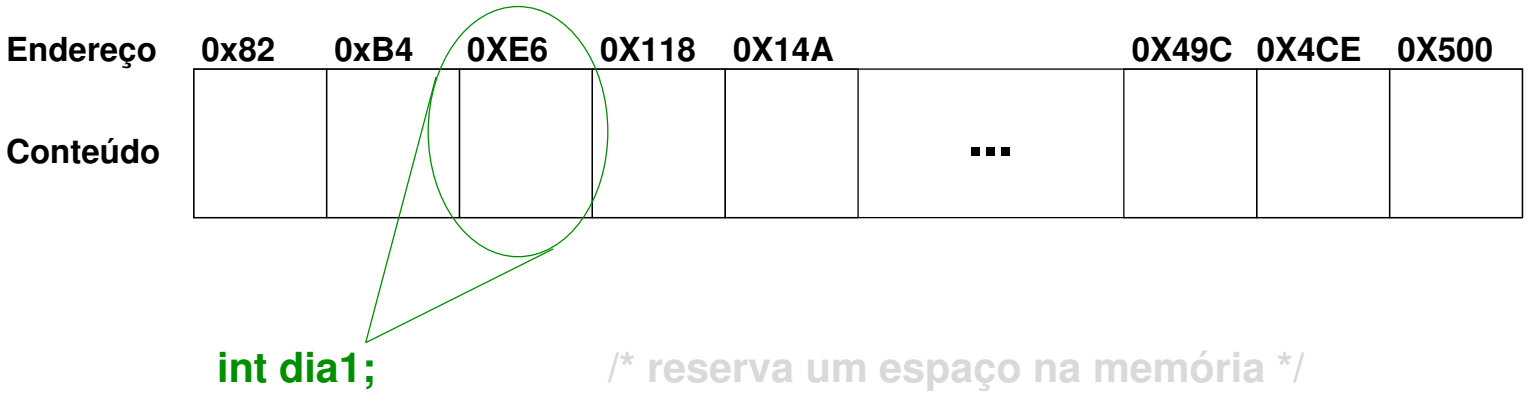
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */
```

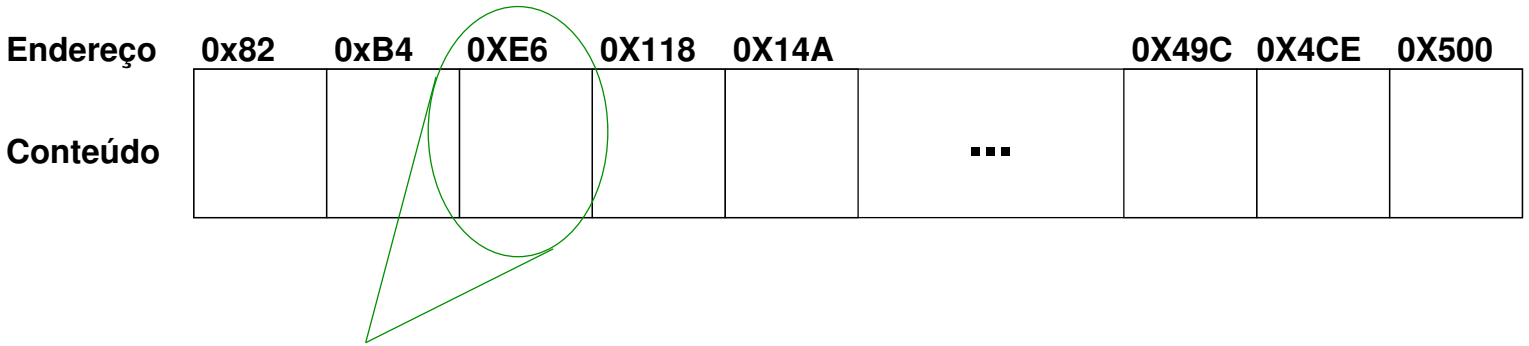
Variáveis

Qual a relação das variáveis com a memória do computador?



Variáveis

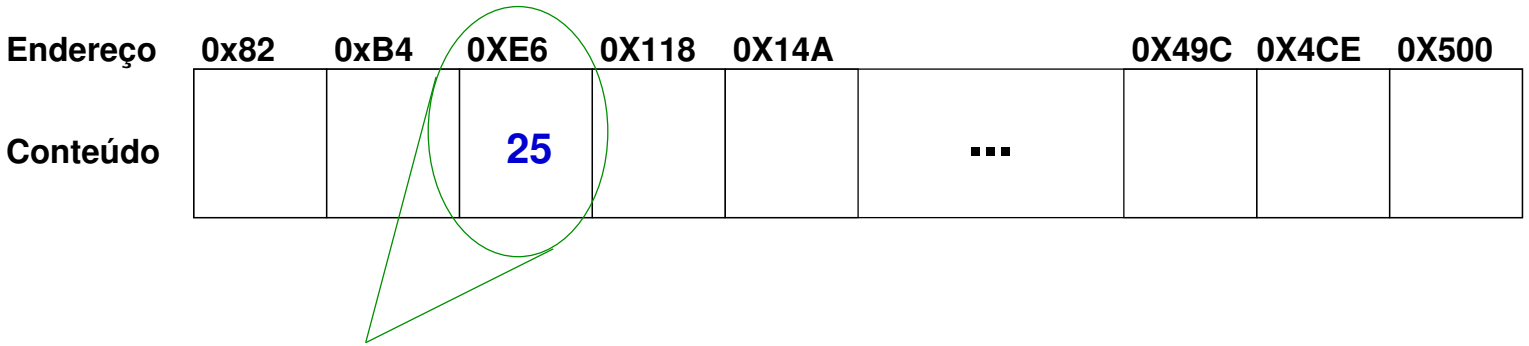
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */

dia1 = 25;         /* atribuiu um valor a variável */
```

Variáveis

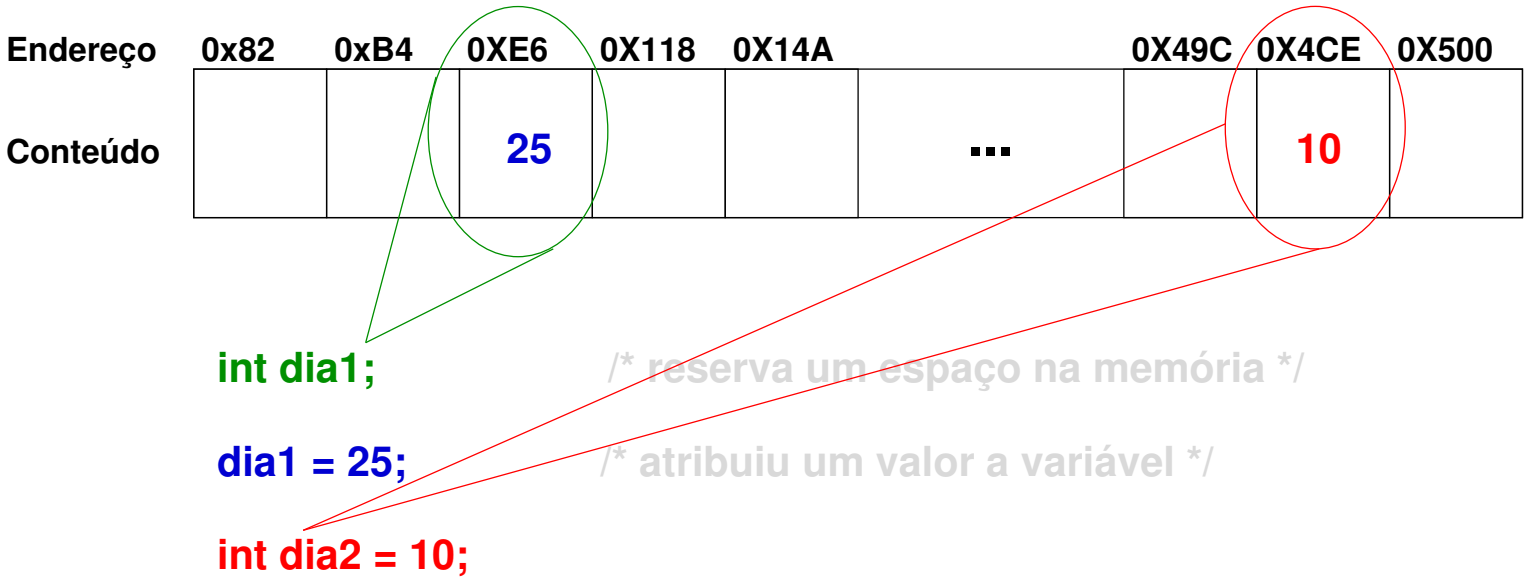
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */

dia1 = 25;         /* atribuiu um valor a variável */
```

Variáveis

Qual a relação das variáveis com a memória do computador?



Variáveis

- Variáveis são referências para um endereço de memória
  - Eu prefiro lembrar o nome `dia` a `0xE6`
- Inicialmente não temos como prever o valor contido em uma variável que não foi iniciada por nós
  - Aquela posição de memória pode já ter sido usada por algum outro processo
  - Assume-se que toda variável não iniciada contém **lixo**

Variáveis

- Variáveis são referências para um endereço de memória
  - Eu prefiro lembrar o nome `dia` a `0xE6`
- Inicialmente não temos como prever o valor contido em uma variável que não foi iniciada por nós
  - Aquela posição de memória pode já ter sido usada por algum outro processo
  - Assume-se que toda variável não iniciada contém **lixo**

Questão

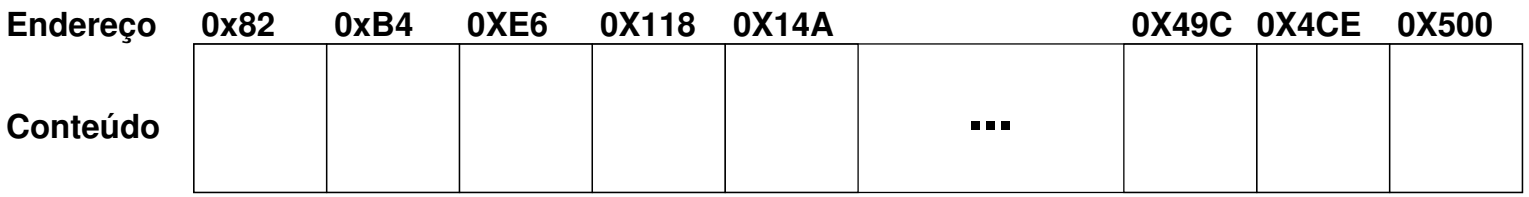
Desenvolva um programa que leia e armazene o ano de nascimento de 5 pessoas. Após efetuar a leitura de todas as datas, informe quantos anos possui cada pessoa.

Uma definição para vetores

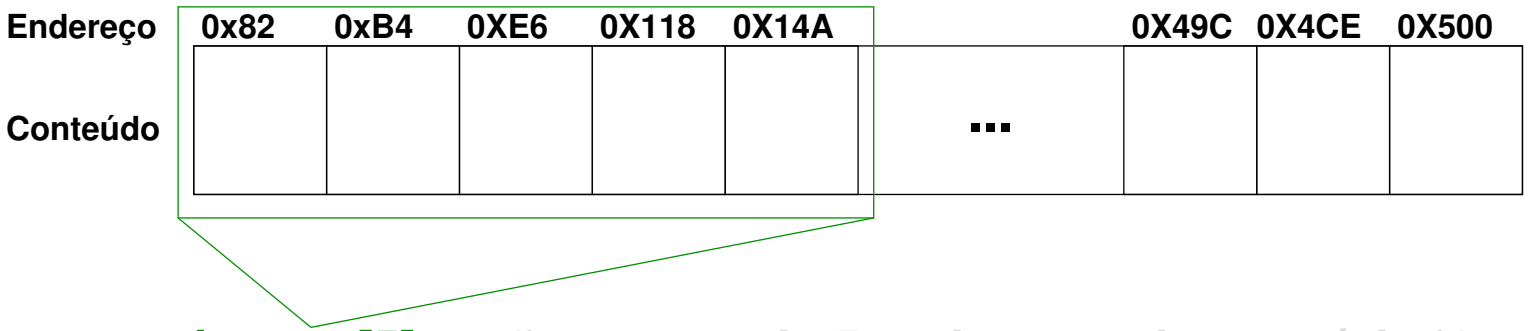
Coleção de variáveis de um mesmo tipo que é referenciado por um nome comum

- Em C, vetores consistem em uma posição contígua na memória
- Elementos específicos de um vetor são acessados através de um **índice**
  - O primeiro elemento é referenciado pelo índice `0`, o segundo por `1`, ...
- Em C, vetores e ponteiros estão intimamente ligados
  - Veremos mais sobre na parte de ponteiros

Vetores

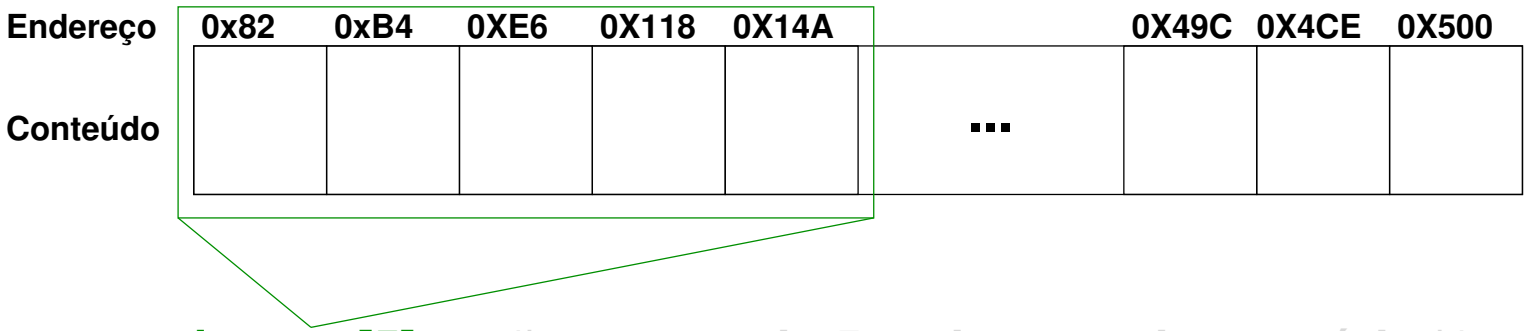

Endereço
0x82
0xB4
0XE6
0X118
0X14A
0X49C
0X4CE
0X500
Conteúdo
...

Vetores



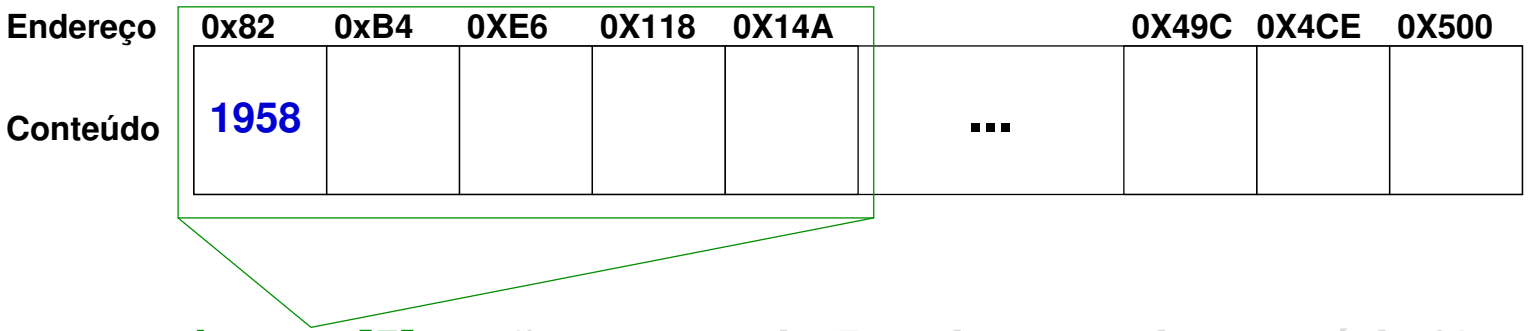

```
int ano[5];     /* reservando 5 endereços de memória */
```

Vetores



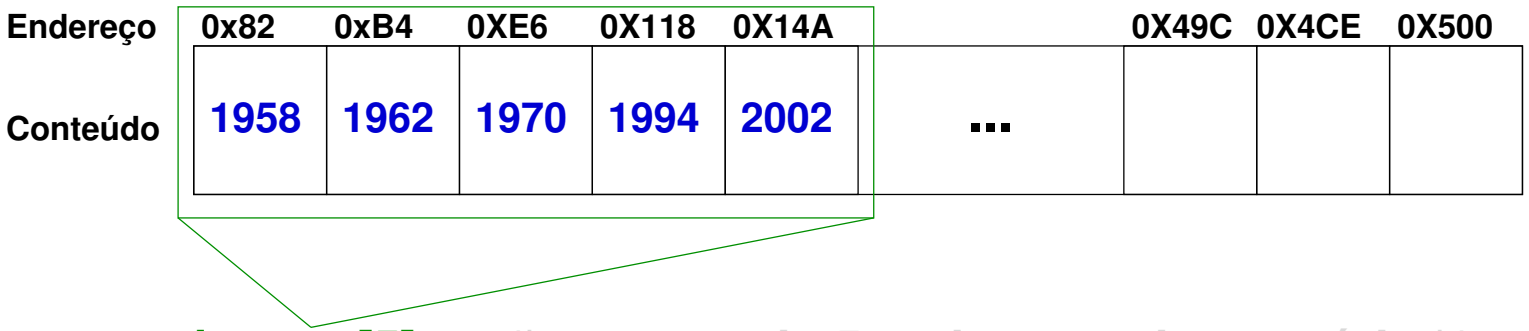

```
int ano[5];     /* reservando 5 endereços de memória */

ano[0] = 1958;
```

Vetores




```
int ano[5];     /* reservando 5 endereços de memória */

ano[0] = 1958;
```

Vetores




```
int ano[5];     /* reservando 5 endereços de memória */
```

```
ano[0] = 1958;
ano[1] = 1962;
ano[2] = 1970;
ano[3] = 1994;
ano[4] = 2002;
```

índice	valor
0	1958
1	1962
2	1970
3	1994
4	2002

Vetores: exercício

1. Desenvolva um programa em C que leia todos os anos em que o Brasil foi campeão mundial de futebol e após coletar todos os dados:
   1. Informar a quantidade de anos que já se passaram desde a conquista de cada título até hoje
   2. Quantos anos se passaram desde a conquista de um título até a conquista do próximo

Exemplo: Ano 1994

- Já se passaram 15 anos desde este título!
- Foi necessário aguardar 8 anos para que pudéssemos levantar o caneco novamente

Vetores

- O tamanho em bytes ocupado por vetor pode ser obtido através de:

```
1 int ano[5];
2 int tamanho;
3 tamanho = sizeof(ano); /* ou tamanho = sizeof(int) * 5 */
```

- Em C não é feita a verificação dos limites de um vetor. Cabe ao programador ficar atento a isto

Vetores

- O tamanho em bytes ocupado por vetor pode ser obtido através de:

```
1 int ano[5];
2 int tamanho;
3 tamanho = sizeof(ano); /* ou tamanho = sizeof(int) * 5 */
```

- Em C não é feita a verificação dos limites de um vetor. Cabe ao programador ficar atento a isto
  - Apesar de incômodo, traz poderes à linguagem (aguarde por ponteiros!)

```
1 int ano[5]; /* vetor declarado com 5 posicoes de 0 a 4 */
2
3 ano[6] = 2000; /* O compilador nao ira' reclamar, mas esta'
      errado */
```

Iniciando vetores

```
1  int anos[5] = { 1958, 1962, 1970, 1994, 2002 };
2
3  int numeros[10];
4  int i;
5
6  for(i=0; i < 10; i++){
7     numeros[i] = 0;
8  }
```

Sumário

Vetores de caracteres

- Não existe um tipo de dados primitivo para armazenar cadeias de caracteres (*strings*)
  - Mas é possível fazer uso de um vetor de caracteres para tal tarefa
- O caractere nulo (\0) é usado para indicar o fim da cadeia de caracteres
  - Assim, para armazenar uma cadeia de caracteres com até 10 letras, será necessário declarar um vetor de caracteres de 11 posições
  - Não é preciso adicionar manualmente o \0 no final de constantes *strings*. O compilador já faz isso por você

char nome[11] = "Curso de C";

'C'	'u'	'r'	's'	'o'	' '	'd'	'e'	' '	'C'	'\0'
0	1	2	3	4	5	6	7	8	9	10

Trabalhando com vetores de caracteres

```
1   char nome[20];
2   printf("Entre com seu nome: ");
3   gets(nome); /* quer tentar com scanf("%s", nome); ? */
4   printf"Boa tarde %s\n", nome);
5
6   /* A funcao gets nao e' segura, ela nao reserva espaco
7    * Problema: estouro de pilha (buffer overflow)
8    * Opte pela fgets
9    */
10  printf("Entre com seu nome: ");
11  fgets(nome, sizeof(nome), stdin);
12  printf"Boa tarde %s\n", nome);
```

Vetores de caracteres: exercício

Contador de palavras

Desenvolva um programa em C que leia uma frase com até 80 caracteres e exiba o tamanho da frase e a quantidade de palavras que a compõe

- Frase: Estou aprendendo a programar em C
- A frase possui 33 caracteres
- Total de palavras: 6

Iniciando vetores não dimensionados

- A linguagem C permite definir vetores não dimensionados, tornando mais simples o processo de iniciar vetores
- Se um vetor for declarado sem dimensão, o compilador criará um vetor grande o suficiente para armazenar a informação atribuída ao vetor

```
1 // declarando um vetor dimensionado
2 char v1[12] = "Linguagem C";
3
4 // declarando um vetor nao dimensionado
5 char v2[] = "Facilidades da linguagem C";
```

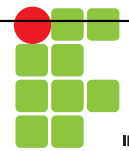
Trabalhando com cadeias de caracteres

- Cadeias de caracteres em C não são tipos de dados primitivos
- Assim, **não é possível** realizar as operações que poderíamos fazer, por exemplo, com inteiros
  - Operações aritméticas: `a + b` ou `(a == b)`
  - Operações relacionais: `a > b`
  - Operações lógicas: `(a && b)`
- A biblioteca padrão **string.h** provê as funções necessárias para atender essas necessidades
  - Por exemplo: Como poderíamos verificar se as cadeias "Curso" e "C" são iguais?

Biblioteca **string.h**

Algumas funções para trabalhar com cadeias de caracteres

Função	**Descrição**
`char *strcpy(destino,origem)`	Copia o conteúdo da origem para o destino
`char *strncpy(destino,origem,n)`	Copia **N** caracteres da origem para o destino
`char *strcat(destino,origem)`	Concatena o conteúdo da origem ao final do destino
`char *strncat(destino,origem,n)`	Concatena **N** caracteres da origem ao final do destino
`int strcmp(a,b)`	Retorna 0 se $a == b$, < 0 se $a < b$, e > 0 se $a > b$
`size_t strlen(cadeia)`	Retorna o tamanho da cadeia de caracteres



Cadeias de caracteres: exercício

Autenticação de usuários

Desenvolva um programa em C que simule a autenticação de usuários do Linux. O programa deverá solicitar um de **nome de usuário** e uma **senha** e confrontá-los com duas respectivas constantes que representariam o único usuário autorizado no sistema. Exibir a mensagem de **acesso autorizado** se os dados forem corretos, caso contrário exibir **acesso negado**

Sumário

INSTITUTO FEDERAL

Vetores com várias dimensões

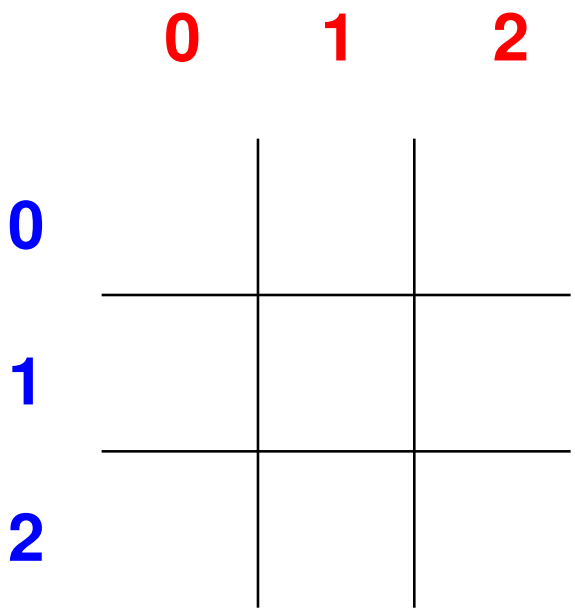
- C permite vetores retangulares multidimensionais (matriz)
  - Até então só vimos vetores unidimensionais
  - `int numeros[10];`
- As dimensões de um vetor são descritas entre colchetes
  - Um vetor de duas dimensões
    - `tipo_do_dado nome_do_vetor[linha][coluna]`
  - É possível ter vetores com mais de duas dimensões
    - `tipo_do_dado nome_do_vetor[a][b][c][d]`

Vetores com várias dimensões

- C permite vetores retangulares multidimensionais (matriz)
  - Até então só vimos vetores unidimensionais
  - `int numeros[10];`
- As dimensões de um vetor são descritas entre colchetes
  - Um vetor de duas dimensões
    - `tipo_do_dado nome_do_vetor[linha][coluna]`
  - É possível ter vetores com mais de duas dimensões
    - `tipo_do_dado nome_do_vetor[a][b][c][d]`

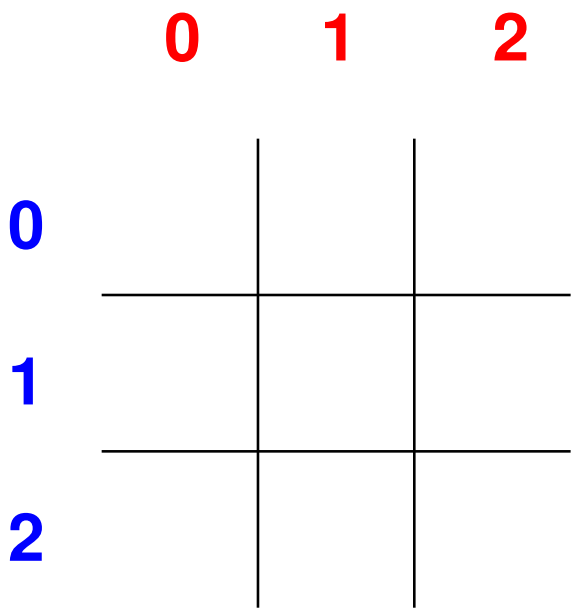
```
1 int matriz[3][3];
2
3 scanf("%d", &mat[1][1]);
4 printf("O valor armazenado em 1,1 e': %d", mat[1][1]);
```

Exemplo de um vetor com duas dimensões



int matriz[3][3];

Exemplo de um vetor com duas dimensões




```
int matriz[3][3];

/* inicie matriz com –1 */
```

Exemplo de um vetor com duas dimensões

	0	1	2
0	–1	–1	–1
1	–1	–1	–1
2	–1	–1	–1

```
int matriz[3][3];

/* inicie matriz com –1 */
```

Exemplo de um vetor com duas dimensões

	0	1	2
0	–1	–1	–1
1	–1	–1	–1
2	–1	–1	–1

```
int matriz[3][3];

/* inicie matriz com –1 */

matriz[0][1] = 5;
```

Exemplo de um vetor com duas dimensões

	0	1	2
0	–1	5	–1
1	–1	–1	–1
2	–1	–1	–1

```
int matriz[3][3];

/* inicie matriz com –1 */

matriz[0][1] = 5;
```

Exemplo de um vetor com duas dimensões

	0	1	2
0	–1	5	–1
1	–1	–1	–1
2	–1	–1	–1

```
int matriz[3][3];

/* inicie matriz com –1 */

matriz[0][1] = 5;

matriz[2][0] = 12;
```

Exemplo de um vetor com duas dimensões

	0	1	2
0	–1	5	–1
1	–1	–1	–1
2	12	–1	–1

```
int matriz[3][3];

/* inicie matriz com –1 */

matriz[0][1] = 5;

matriz[2][0] = 12;
```

Iniciando vetores não dimensionados (cont)

- É possível criar vetores não dimensionados com mais de uma dimensão
- Contudo, é necessário especificar todas as dimensões exceto a dimensão mais à esquerda
  - O compilador necessita disto para fazer a indexação correta

```
1 // declarando um vetor nao dimensionado
2 int vet[][2] = {
3 1,2,
4 2,4,
5 3,8,
6 4,16
7 };
```

Vetores com várias dimensões: exercício

Multiplicação de matrizes

Desenvolva um programa em C que faça a multiplicação de duas matrizes, A e B. O usuário deverá entrar com os valores para cada matriz

Teoria: A multiplicação: $A \times B$ só é possível se o número de colunas de A for igual ao número de linhas de B. Assim:

$A_{m,n} \times B_{n,p} = C_{m,p}$

$C_{i,j} = A_{i,1} \times B_{1,j} + A_{i,2} \times B_{2,j} + \ldots + A_{i,n} \times B_{n,j}$

Vetores com várias dimensões: exercício

Multiplicação de matrizes

Desenvolva um programa em C que faça a multiplicação de duas matrizes, A e B. O usuário deverá entrar com os valores para cada matriz

Teoria: A multiplicação: $A \times B$ só é possível se o número de colunas de A for igual ao número de linhas de B. Assim:
$A_{m,n} \times B_{n,p} = C_{m,p}$
$C_{i,j} = A_{i,1} \times B_{1,j} + A_{i,2} \times B_{2,j} + \ldots + A_{i,n} \times B_{n,j}$

$$\begin{bmatrix} 0 & 2 & 4 \\ 1 & 1 & 3 \end{bmatrix} \times \begin{bmatrix} 1 & 5 \\ 6 & 2 \\ 3 & 4 \end{bmatrix} =$$

$$\begin{bmatrix} (0 \times 1 + 2 \times 6 + 4 \times 3) & (0 \times 5 + 2 \times 2 + 4 \times 4) \\ (1 \times 1 + 1 \times 6 + 3 \times 3) & (1 \times 5 + 1 \times 2 + 3 \times 4) \end{bmatrix} = \begin{bmatrix} 24 & 20 \\ 16 & 19 \end{bmatrix}$$

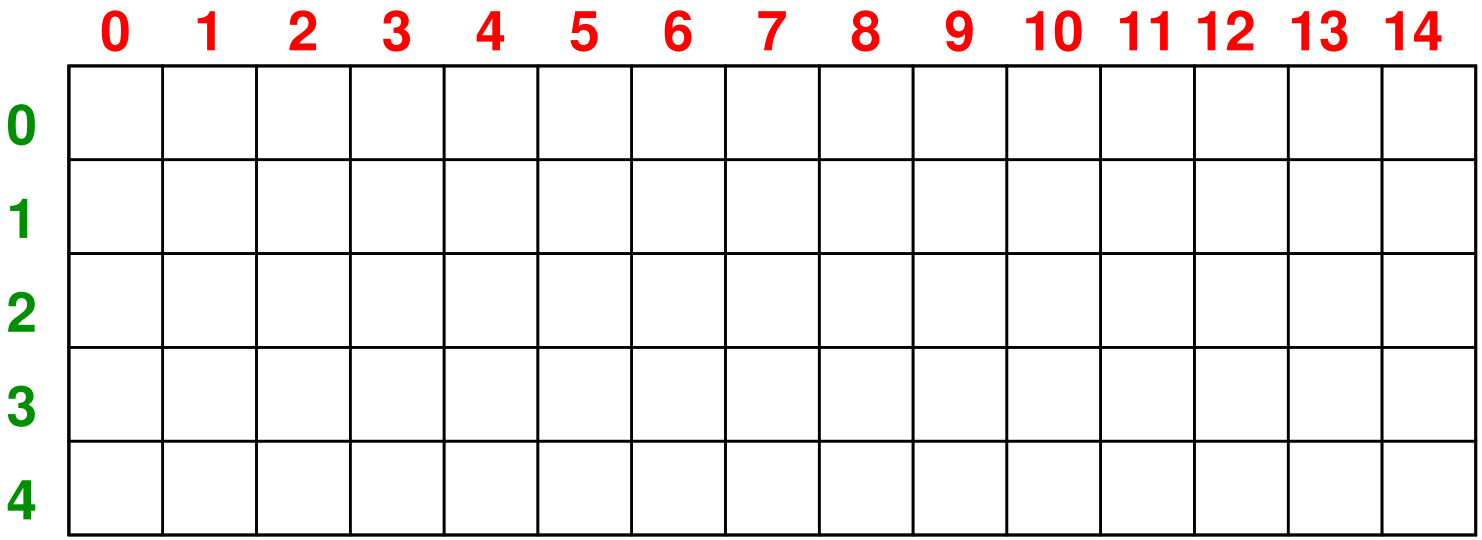
Vetores multidimensionais de caracteres

- Haverá casos que desejaremos criar um vetor que contenha cadeias de caracteres (*strings*)
- O código **char alunos[5][15]** declara um vetor multidimensional que poderá armazenar 5 nomes de alunos, sendo que cada nome poderá ter no máximo 14 caracteres
  - Lembre-se que o caractere '\0' delimita cadeia de caracteres

```
1 char alunos[5][15];
2
3 /* Fazendo a leitura do 1o. aluno */
4 fgets(alunos[0], sizeof(alunos[0]), stdin);
```

Vetores multidimensionais de caracteres

char alunos[5][15];



Vetores multidimensionais de caracteres

char alunos[5][15];

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'									
1	'J'	'u'	'c'	'a'	'\0'										
2	'M'	'a'	'r'	'i'	'a'	'\0'									
3	'A'	'l'	'i'	'c'	'e'	'\0'									
4															

Vetores multidimensionais de caracteres

char alunos[5][15]; **L = LIXO**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'	L	L	L	L	L	L	L	L	L
1	'J'	'u'	'c'	'a'	'\0'	L	L	L	L	L	L	L	L	L	L
2	'M'	'a'	'r'	'i'	'a'	'\0'	L	L	L	L	L	L	L	L	L
3	'A'	'l'	'i'	'c'	'e'	'\0'	L	L	L	L	L	L	L	L	L
4	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L

Vetores multidimensionais de caracteres: exercício

Nome do mês por extenso

Desenvolva um programa em C que permita ao usuário fornecer o dia, mês e ano da seguinte forma: 10/03/2009 e exiba o mês por extenso.
Exemplo: 10 de março de 2009

Sumário

Sumário

Ponteiros

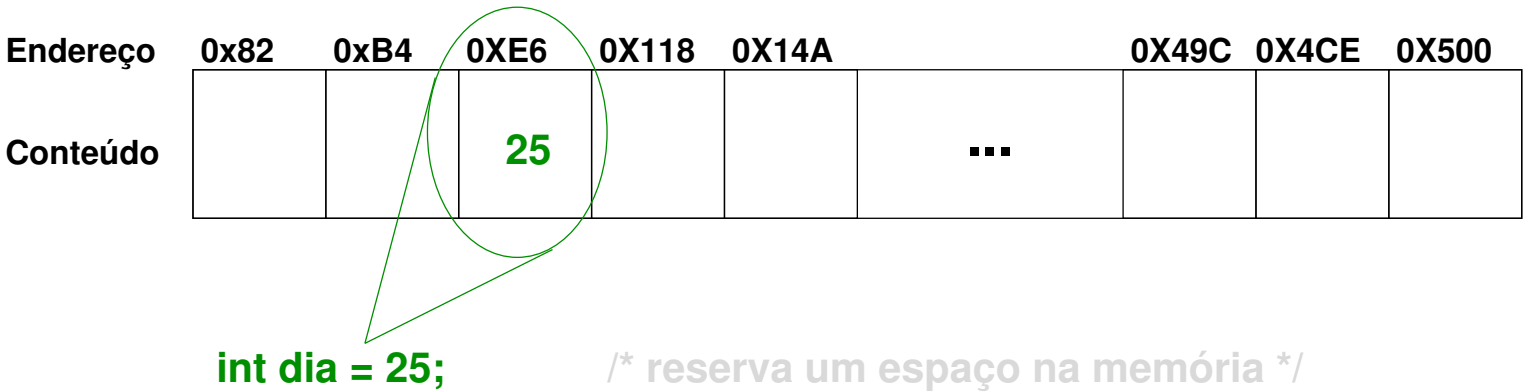
- **Ponteiro** é uma variável que contém o **endereço de memória** de uma variável
  - Traz grandes poderes a linguagem, contudo é fonte de grandes confusões
  - É fácil fazer o uso incorreto de ponteiros, por isso o perigo
- Os ponteiros são largamente utilizados em programas complexos
  - Alocação dinâmica de recursos
  - Argumentos das funções
  - Aumenta a eficiência de alguns códigos

Ponteiros

- Qualquer *byte* na memória pode ser um `char` e quatro pares de *bytes* podem ser um `long`, etc.
- Um ponteiro é um grupo de células (geralmente 2 ou 4) que armazenam endereços

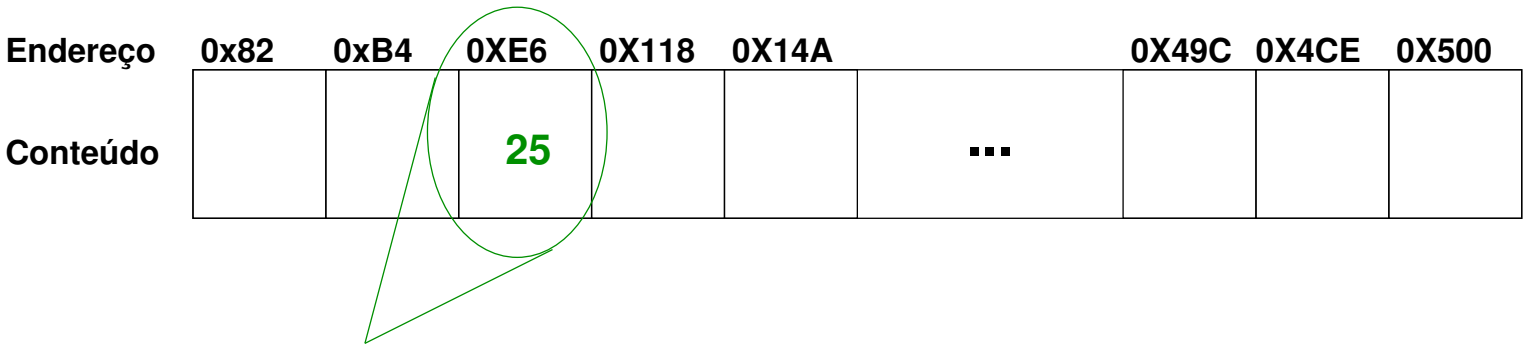
Ponteiros

- Qualquer *byte* na memória pode ser um `char` e quatro pares de *bytes* podem ser um `long`, etc.
- Um ponteiro é um grupo de células (geralmente 2 ou 4) que armazenam endereços



Ponteiros

- Qualquer *byte* na memória pode ser um `char` e quatro pares de *bytes* podem ser um `long`, etc.
- Um ponteiro é um grupo de células (geralmente 2 ou 4) que armazenam endereços



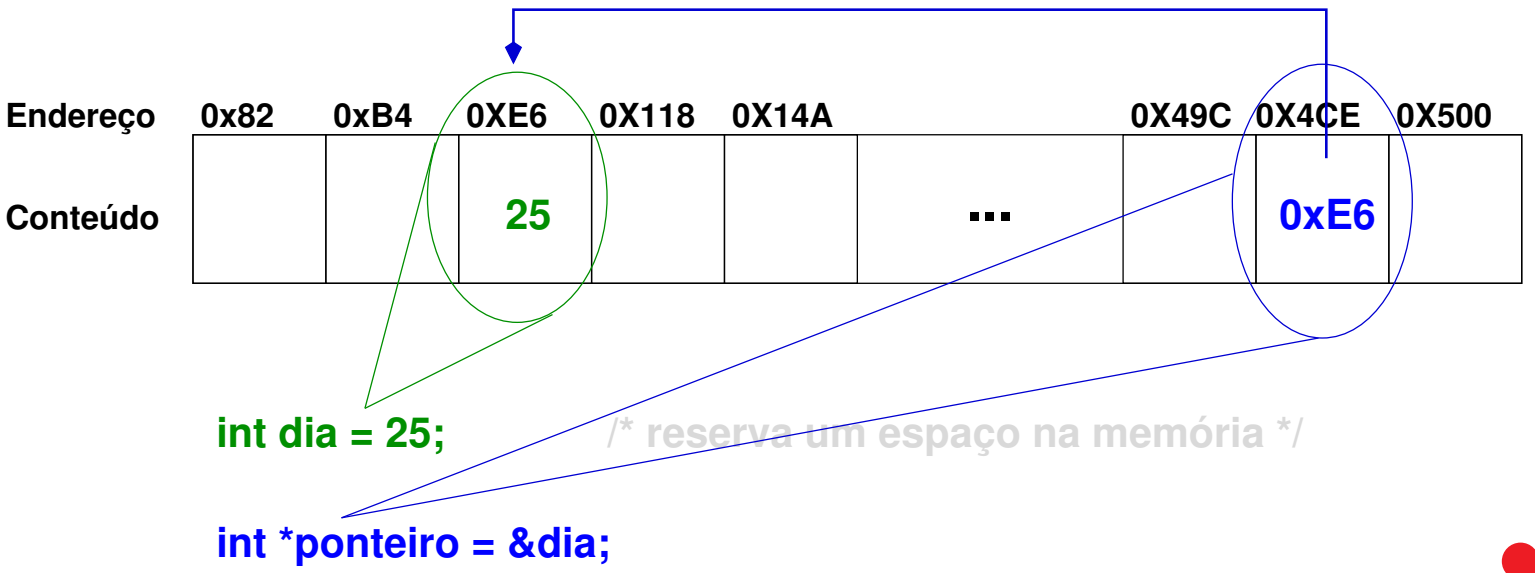

```
int dia = 25;          /* reserva um espaço na memória */

int *ponteiro = &dia;
```

Ponteiros

- Qualquer *byte* na memória pode ser um `char` e quatro pares de *bytes* podem ser um `long`, etc.
- Um ponteiro é um grupo de células (geralmente 2 ou 4) que armazenam endereços

Ponteiros

- Qualquer *byte* na memória pode ser um `char` e quatro pares de *bytes* podem ser um `long`, etc.
- Um ponteiro é um grupo de células (geralmente 2 ou 4) que armazenam endereços



Ponteiros

- A declaração de uma variável do tipo ponteiro consiste em definir um tipo base, o símbolo * e o nome da variável
  - `int *ponteiro;`
- O tipo base serve para indicar o tipo das variáveis que o ponteiro irá apontar
  - Mesmo assim é possível apontar para variáveis de outro tipo, porém não é uma boa prática, principalmente se desejarmos trabalhar com a **aritmética de ponteiros**

Operadores de ponteiros

- O operador unário & retorna o endereço na memória do seu operando

```
1 p = &c; // Atribui o endereco da variavel c a variavel p
```

  - Assim, p **aponta para** c
    - Só pode ser usado com objetos presentes em memória, como variáveis e vetores.
    - Não se pode usar para constantes, expressões ou variáveis *register*

- O operador unário * retorna o valor do objeto para o qual o ponteiro está apontando

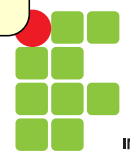
```
1 i = *p; // i recebe o valor que esta' no endereco onde p
      aponta
```

  - Assim, i recebe o valor que está no endereço de p

Ponteiros: exemplo

Variável	Endereço	Valor
x	0x82	1
y	0xB4	2
p	0xE6	
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
```



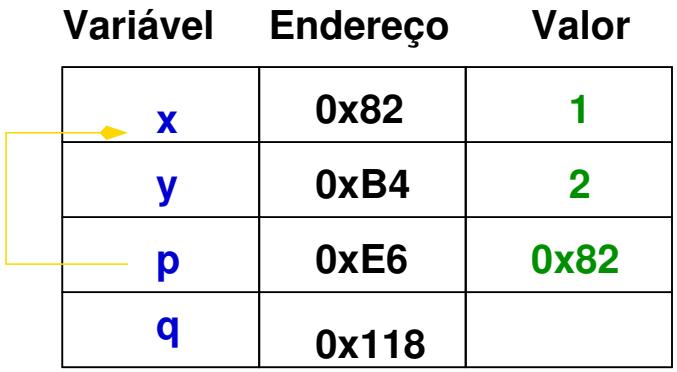
INSTITUTO FEDERAL

Ponteiros: exemplo

Variável	Endereço	Valor
x	0x82	1
y	0xB4	2
p	0xE6	
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;                 // atribui o endereço de x a p
```

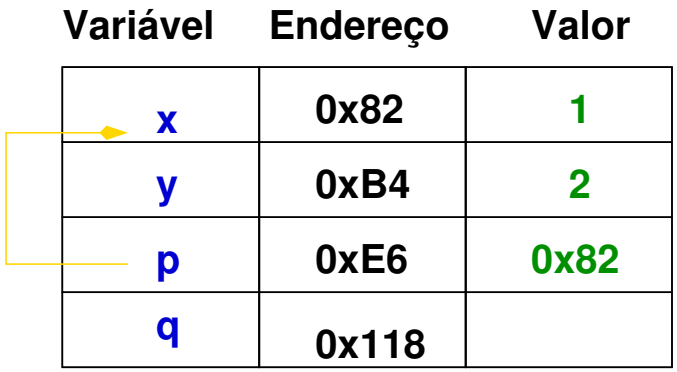
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	1
y	0xB4	2
p	0xE6	0x82
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;                 // atribui o endereço de x a p
```

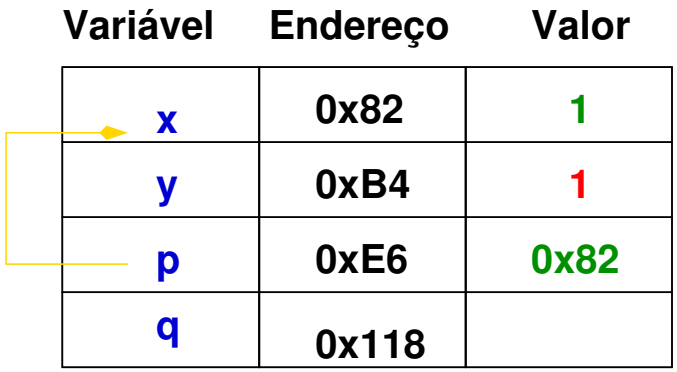
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	1
y	0xB4	2
p	0xE6	0x82
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
```

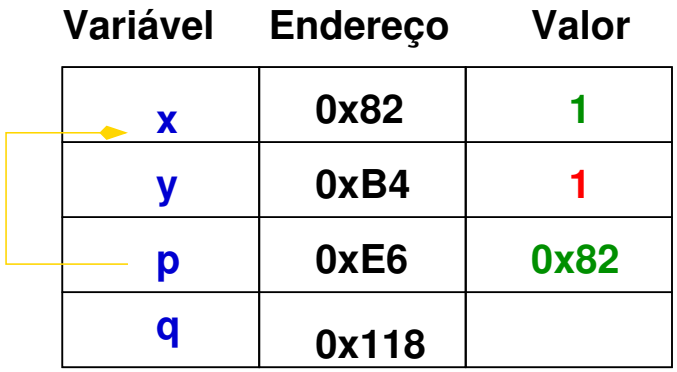
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	1
y	0xB4	1
p	0xE6	0x82
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
```

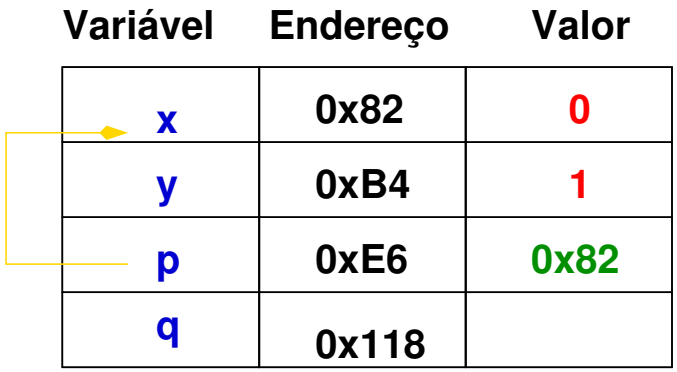
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	1
y	0xB4	1
p	0xE6	0x82
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
```

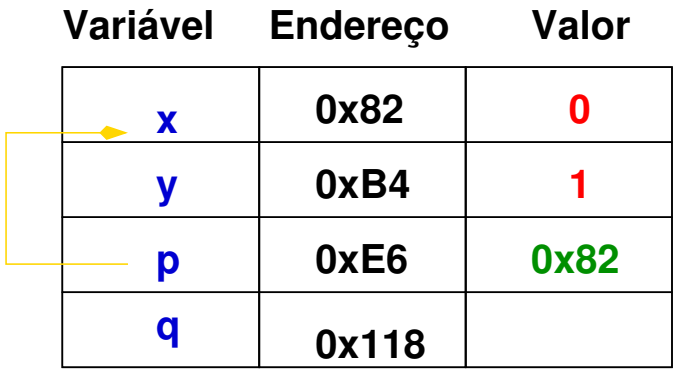
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	0
y	0xB4	1
p	0xE6	0x82
q	0x118	

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
```

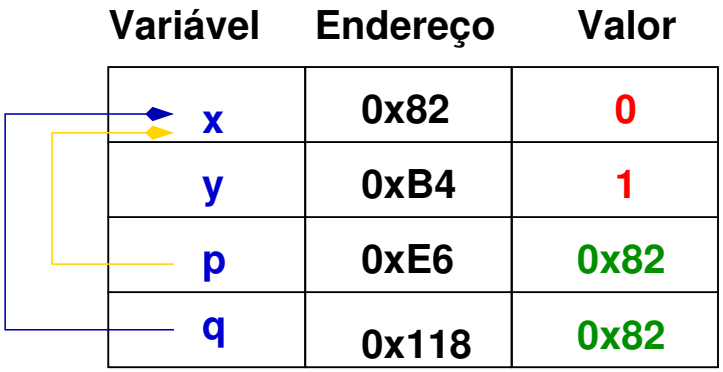
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	0
y	0xB4	1
p	0xE6	0x82
q	0x118	

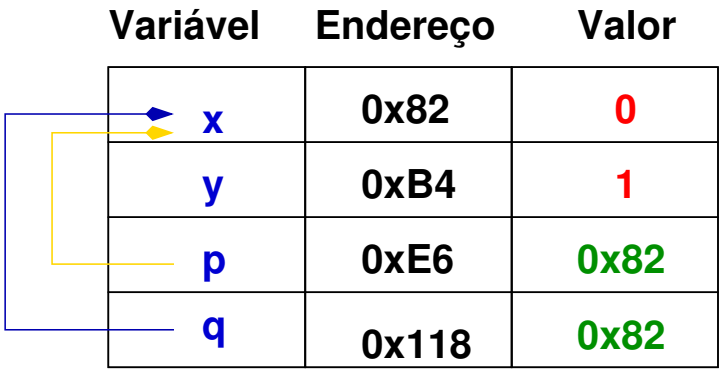
```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
q = p;           // atribui o valor de p em q
```

Ponteiros: exemplo




```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
q = p;           // atribui o valor de p em q
```

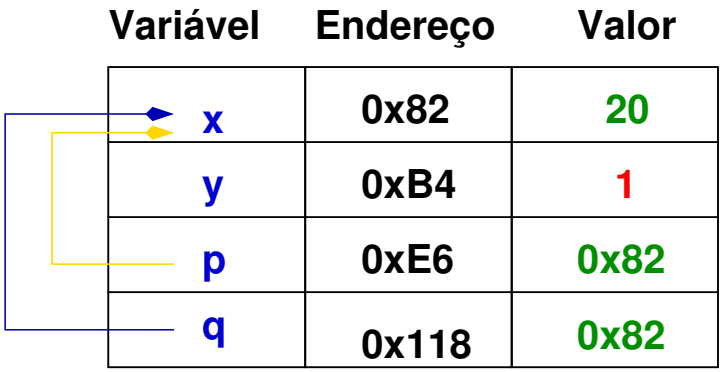
Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	0
y	0xB4	1
p	0xE6	0x82
q	0x118	0x82

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
q = p;           // atribui o valor de p em q
*q = 20;         // atribui 20 no endereço onde q aponta
```

Ponteiros: exemplo



Variável	Endereço	Valor
x	0x82	20
y	0xB4	1
p	0xE6	0x82
q	0x118	0x82

```
int x = 1, y = 2;
int *p, *q;
p = &x;          // atribui o endereço de x a p
y = *p;          // atribui o valor contido no endereço onde p aponta
*p = 0;          // atribui 0 no endereço onde p aponta
q = p;           // atribui o valor de p em q
*q = 20;         // atribui 20 no endereço onde q aponta
```

Ponteiros: exemplo

```
1  int i = 10, *p, *r;
2  char *frase = "Curso de C";
3
4  p = &i;
5
6  printf("Em p esta' armazenado: %p\n", p);
7  printf("O valor em i eh %d e i esta' no endereco %p\n\n", i, &i);
8
9  printf("No endereco %p, para onde p aponta, esta' armazenado: %d\n
       ", p, *p);
10
11 r = p;
12 printf("No endereco %p, para onde r aponta, esta' armazenado: %d\n
       ", r, *r);
13
14 printf("Valor da cadeia de caracteres eh: %s", frase);
```

Boa prática de programação

- Uma variável declarada como ponteiro, como qualquer variável de outro tipo, terá armazenado um valor desconhecido
- Fazer uso de um ponteiro que não foi iniciado gera grandes problemas que muitas vezes resulta no encerramento não esperado do programa
  - Sistemas operacionais menos seguros podem até travar
- Por convenção, um ponteiro que não aponta para um endereço de memória válido deverá ser nulo
  - O nulo é obtido através da atribuição do valor **0** ao ponteiro
  - Qualquer ponteiro que aponta para 0 não deverá ser usado

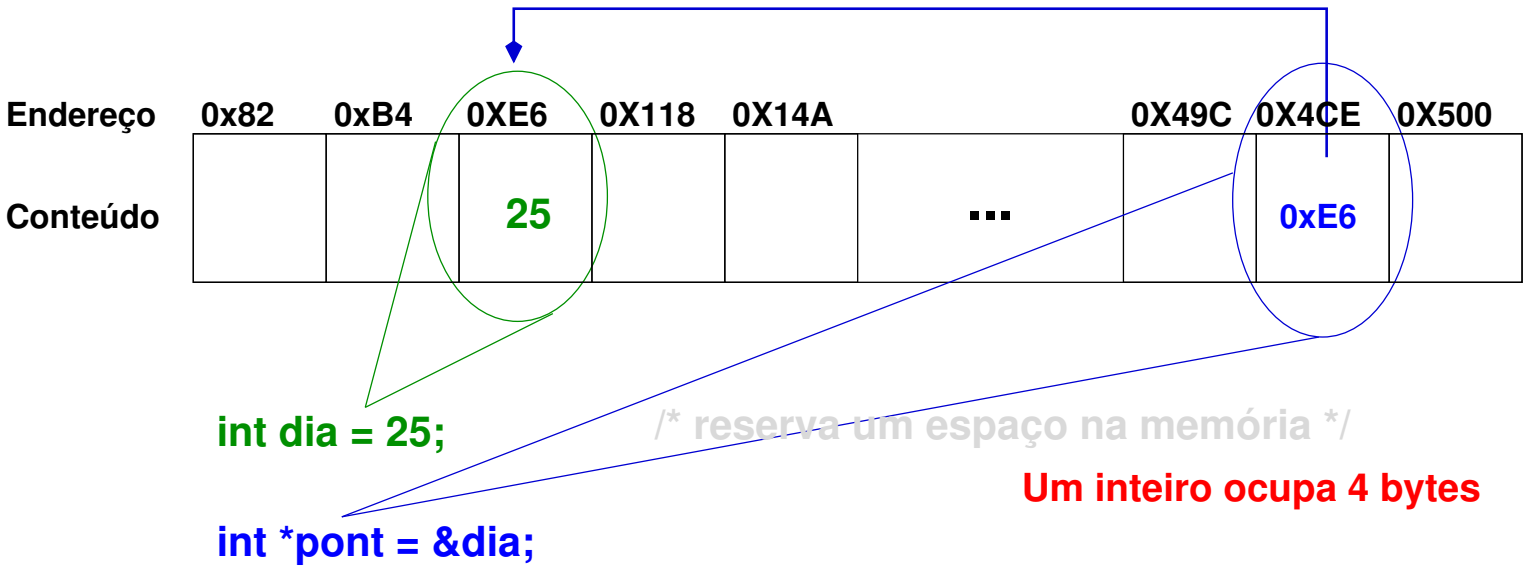
Sumário

Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base

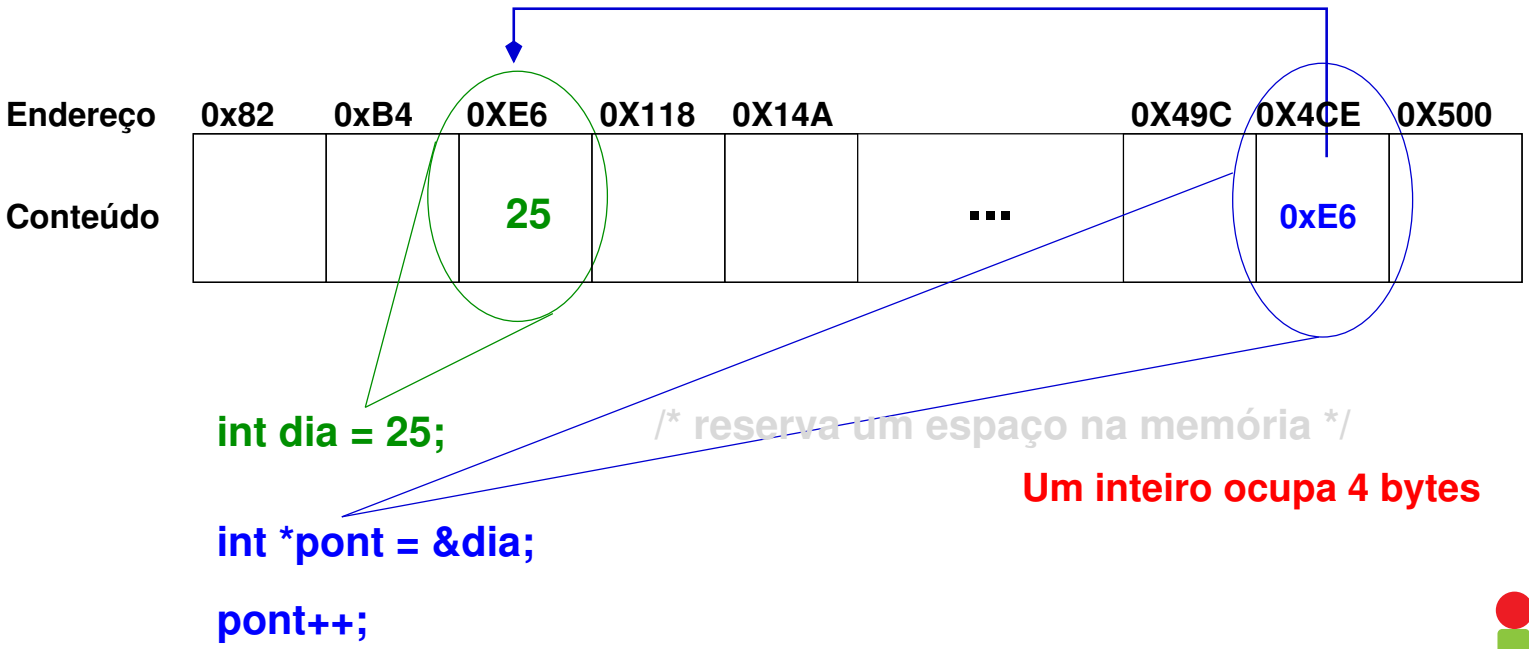
Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base



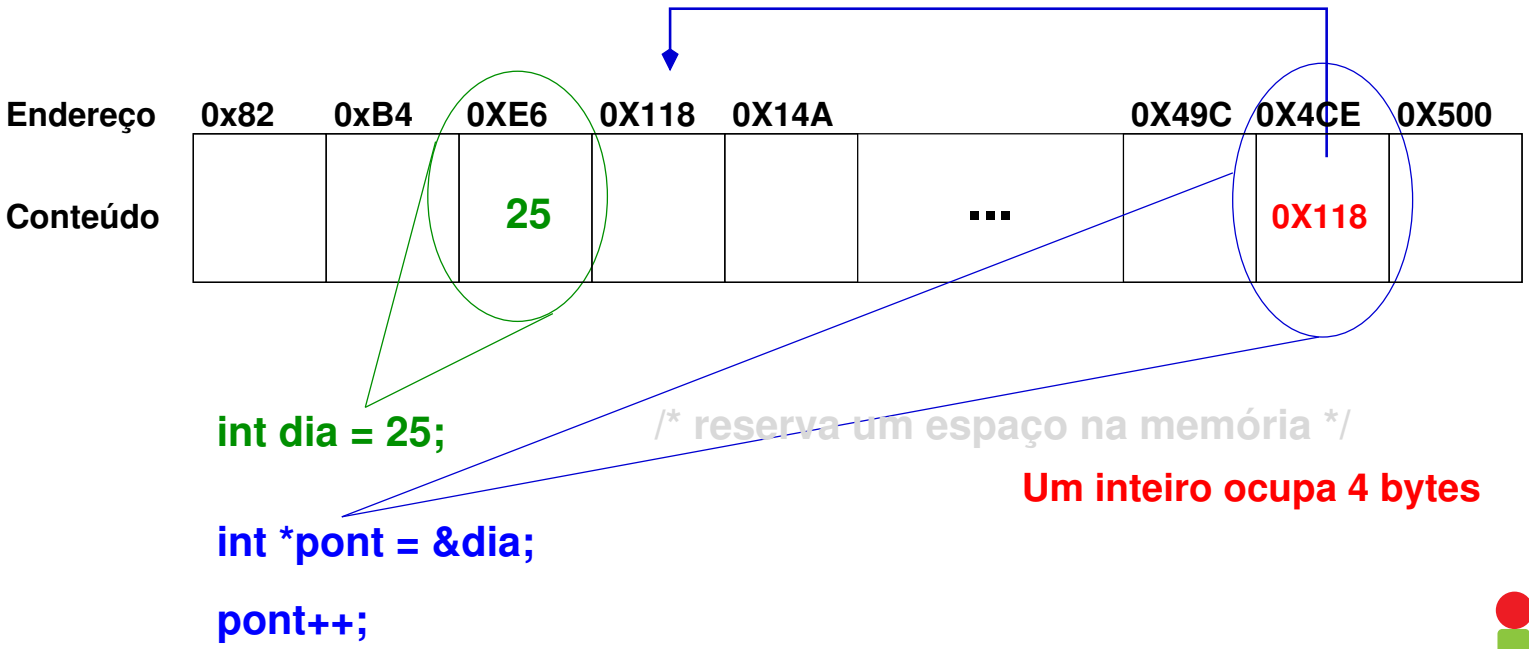
Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base



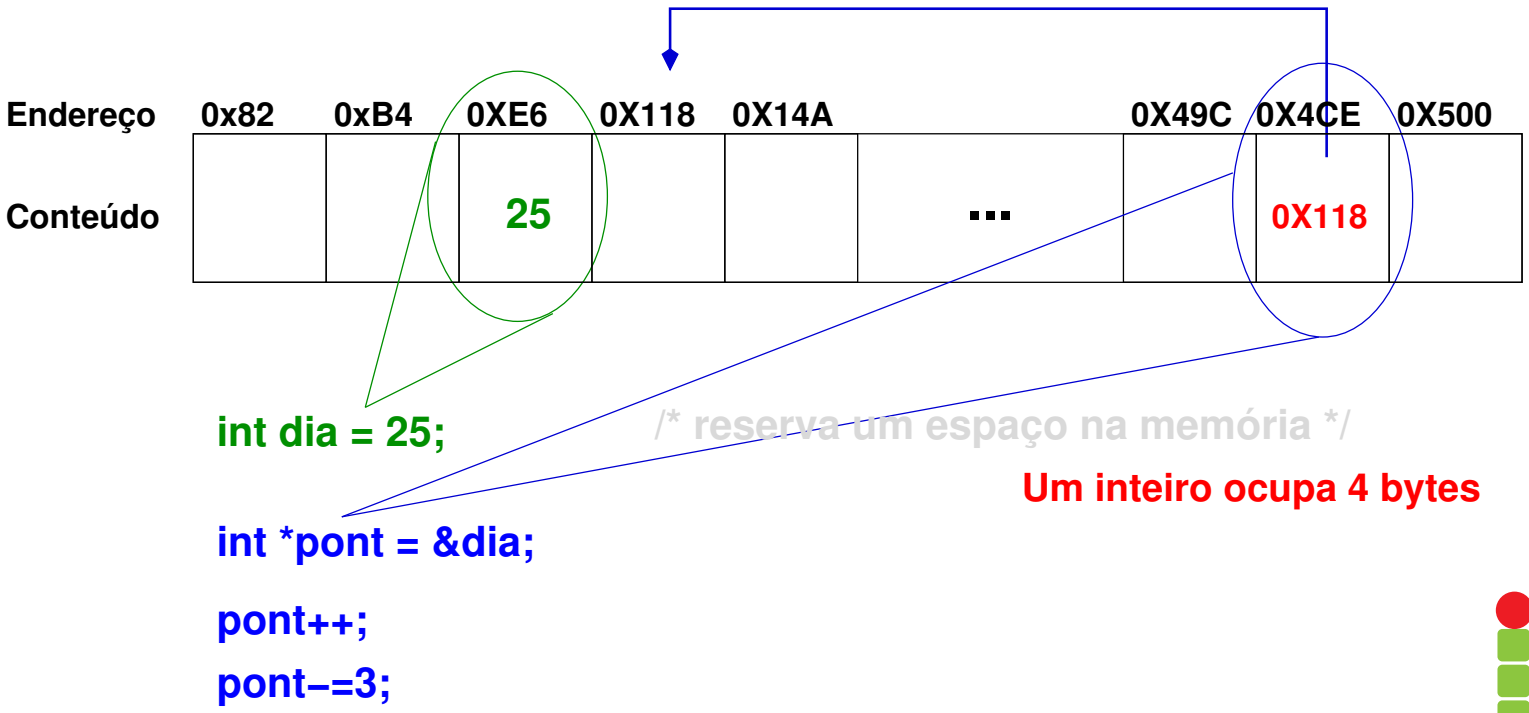
Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base



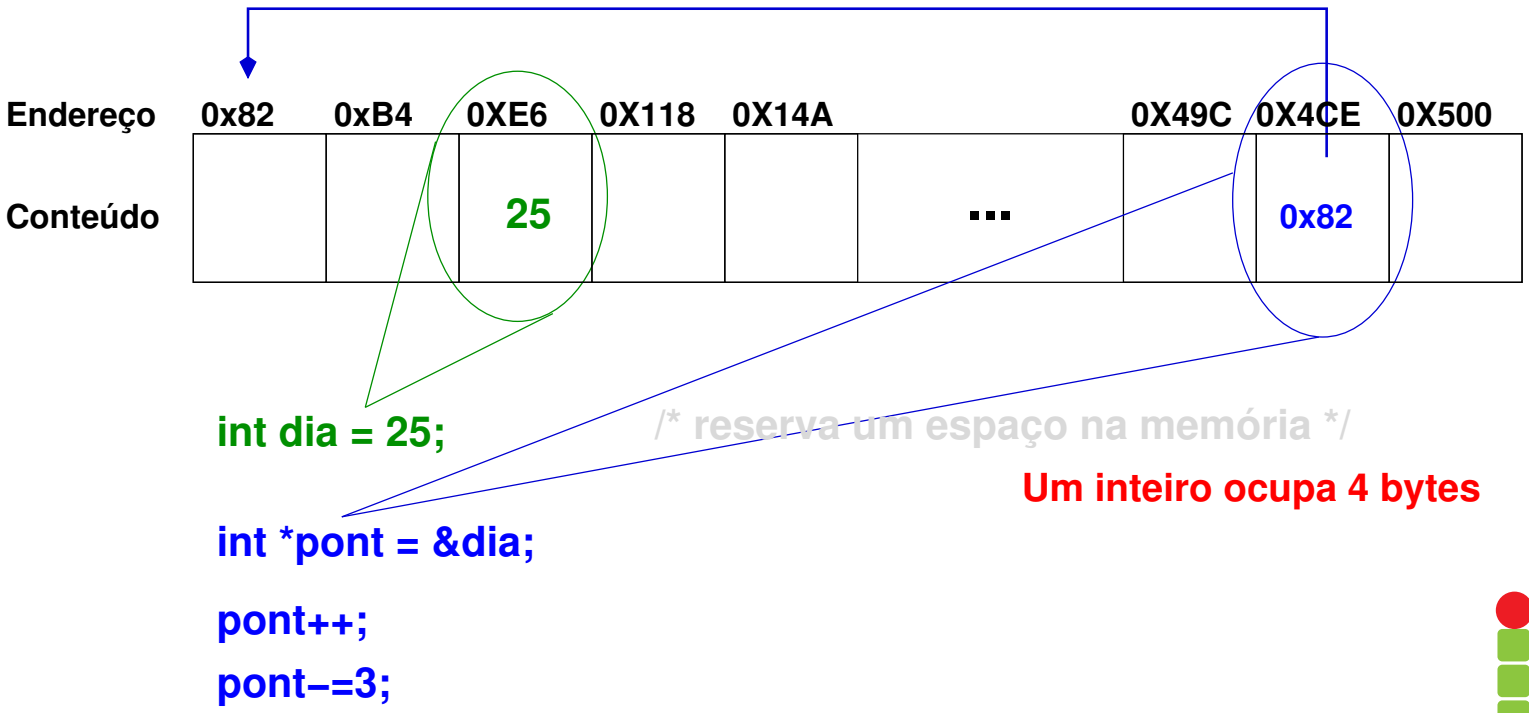
Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base



Aritmética de ponteiros

- Só é possível realizar duas operações aritméticas com ponteiros: **adição** e **subtração**
  - Exemplos: `p++; p--; p = p + 1; p+=10; p+=i;`
- Todo ponteiro ao ser incrementado em *i* irá apontar para a posição de memória do próximo elemento do seu tipo base



Aritmética de ponteiros: exercício

```
1  int i = 10, j, *p;
2
3  p = &i;
4  *p = *p + 1;
5  j = *p + 2;
6  *p += 1;
7  (*p)++;
8  *p++;
9  p++;
10
11 printf("p aponta para %p, que contem %d\n", p, *p);
```

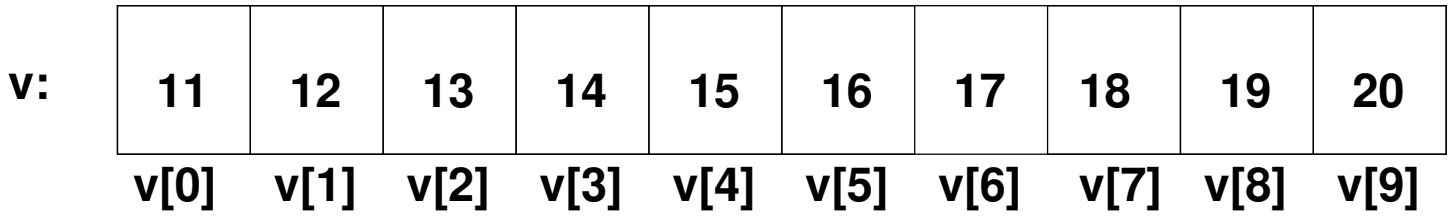
- Após realizar a operação de cada linha do trecho acima, imprima o endereço para onde **p** aponta e o valor que está contido neste endereço. Um exemplo de como fazer isso está na linha 11.

Sumário

Ponteiros e vetores

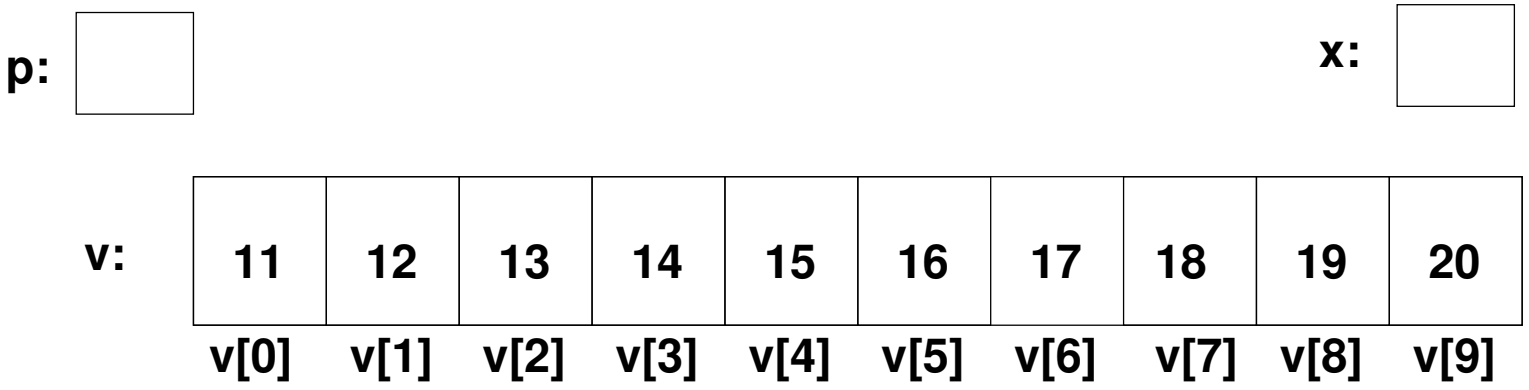
- Ponteiros e vetores possuem uma estreita relação
- Qualquer operação que possa ser obtida através da **indexação de vetores** também poderá ser feita com **ponteiros**
- A versão usando ponteiro geralmente é mais rápida que a versão usando indexação de vetores
  - Inicialmente pode parecer mais complexa para entender

Ponteiros e vetores



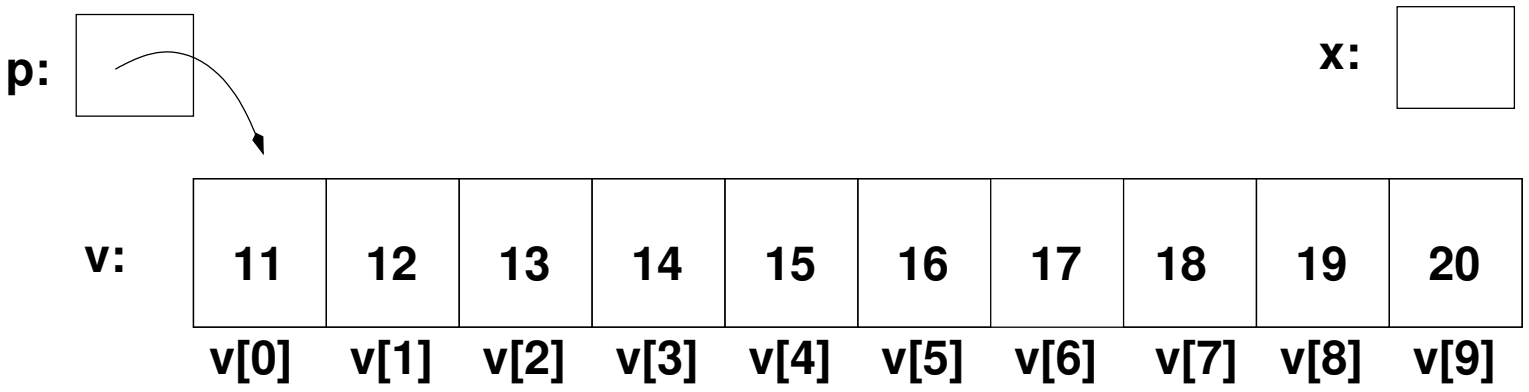

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
```

Ponteiros e vetores



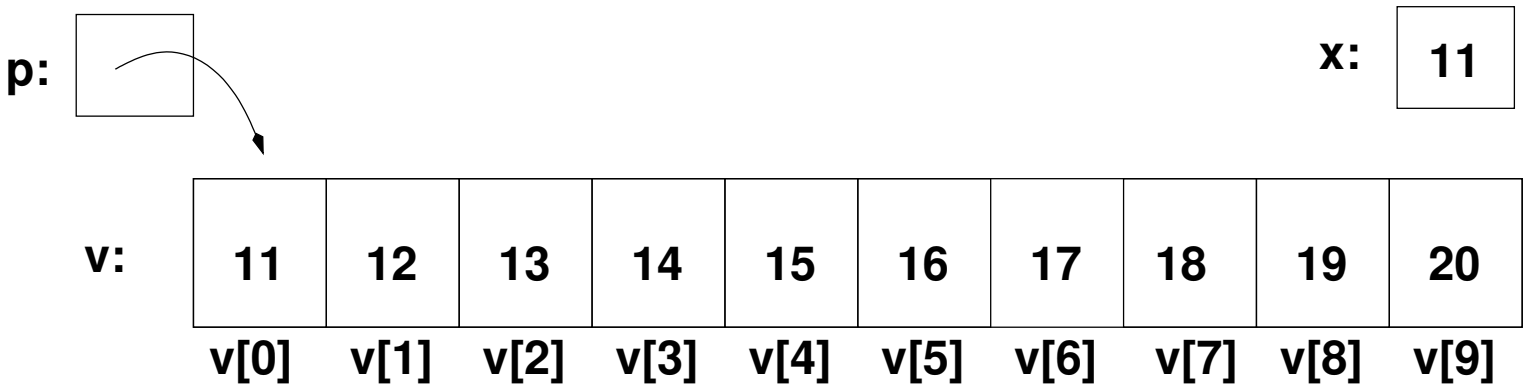

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
```

Ponteiros e vetores



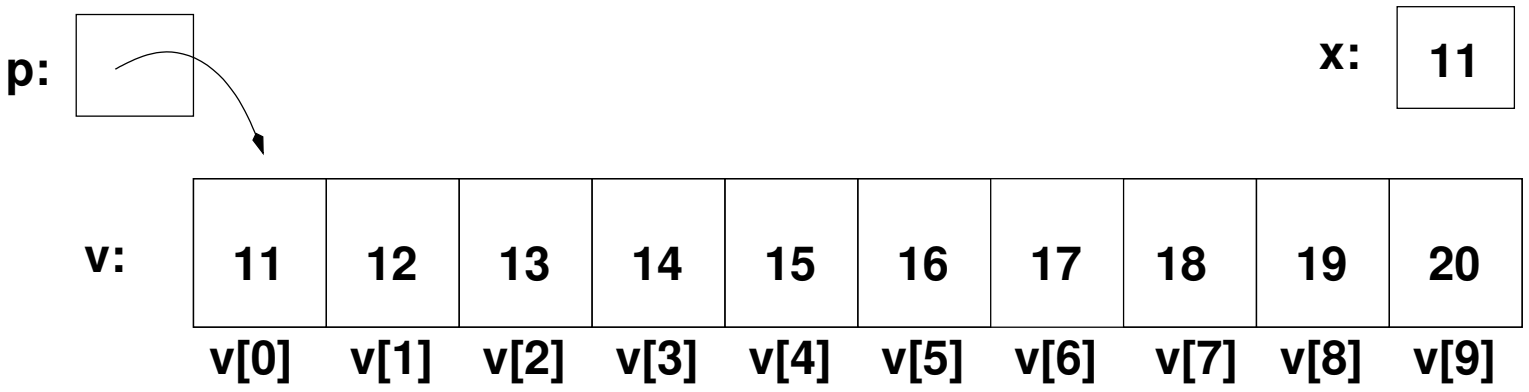

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
```

Ponteiros e vetores



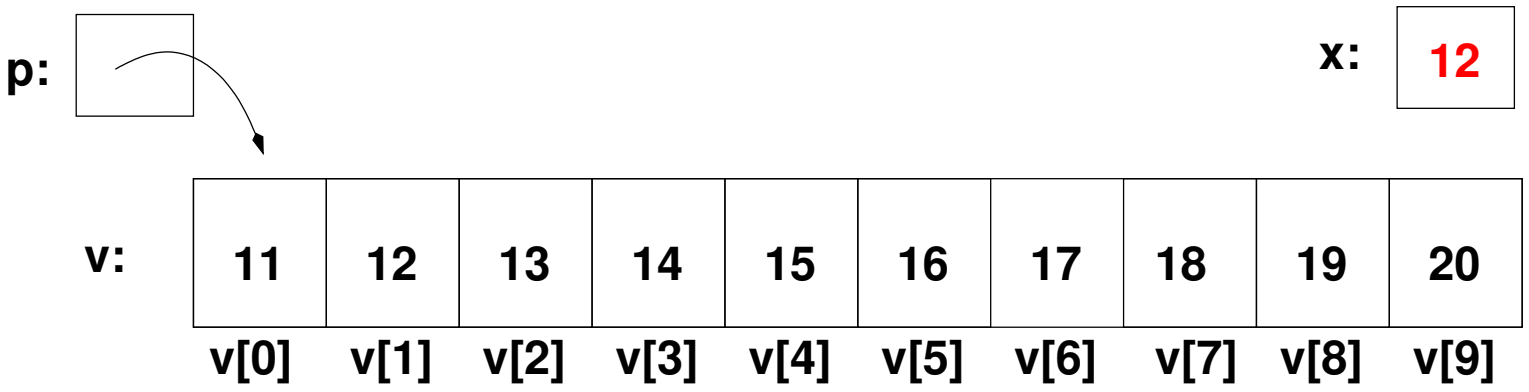

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
```

Ponteiros e vetores



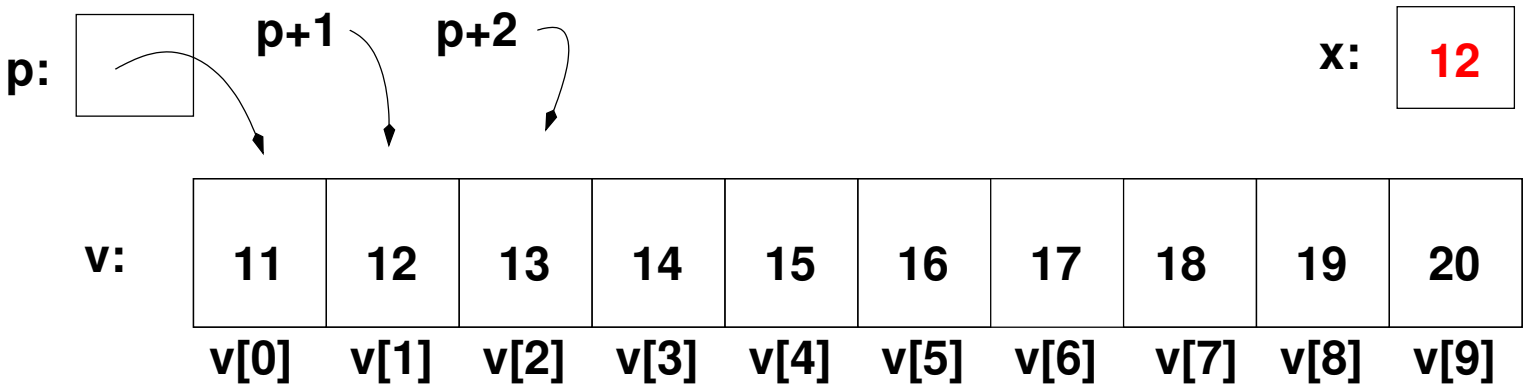

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores




```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores




```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores

```
1   int i, vet[10], vet2[10], *p;
2
3     // usando indices de vetores
4     for(i = 0; i < 10; i++){
5        vet[i] = 0;
6     }
7
8     // usando ponteiros
9     p = vet2;
10  i
11    for(i = 0; i < 10; i++){
12       *(p+i) = 0;
13    }
```

Ponteiros e vetores

```
1  int i, j, vet[3][3], vet2[3][3], *p;
2
3  // usando indices de vetores
4  for(i = 0; i < 3; i++){
5     for(j = 0; j < 3; j++){
6        vet[i][j] = 0;
7     }
8  }
9
10 // usando ponteiros
11 p = &vet2[0][0];
12
13 for(i = 0; i < 9; i++){
14    *(p+i) = 0;
15 }
```

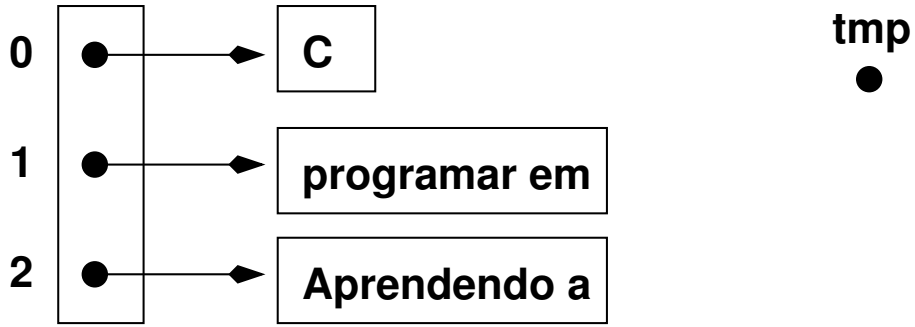
- 9 deslocamentos *vs* 9 incrementos.

Vetores de ponteiros

- Ponteiros também são variáveis, logo é possível definir vetores de ponteiros

```
1  char *frases[3];
2
3  frases[0] = "Aprendendo a";
4  frases[1] = "programar em";
5  frases[2] = "C";
6
7  printf("%s %s %s\n", frases[0], frases[1], frases[2]);
```

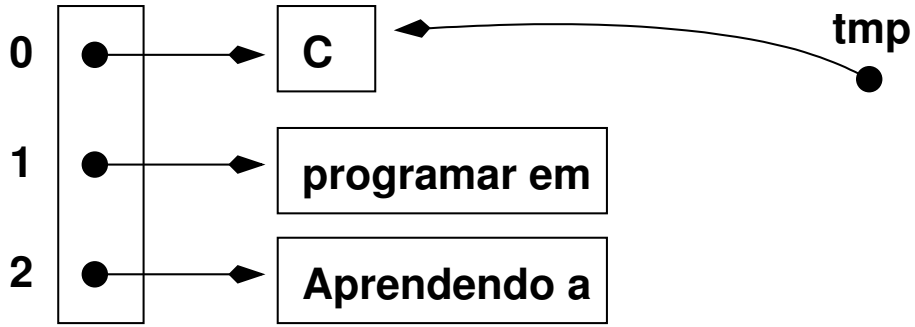
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
```

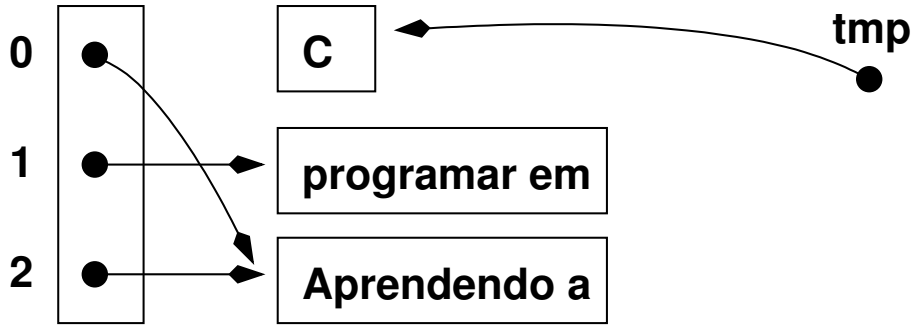
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
```

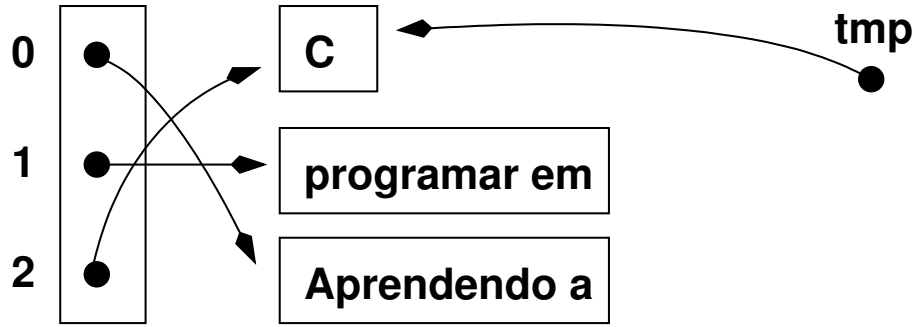
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
```

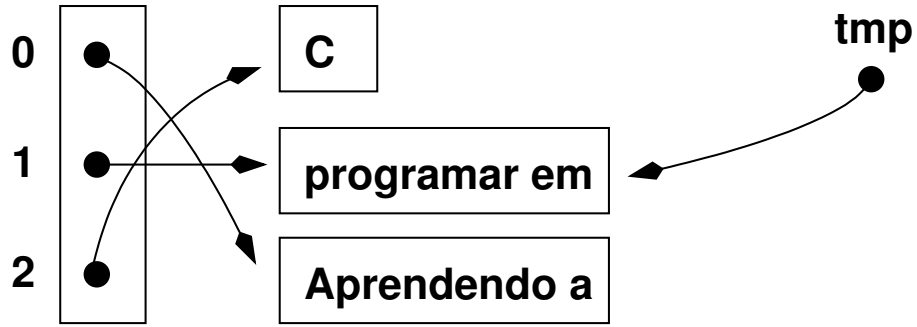
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
frases[2] = tmp;
```

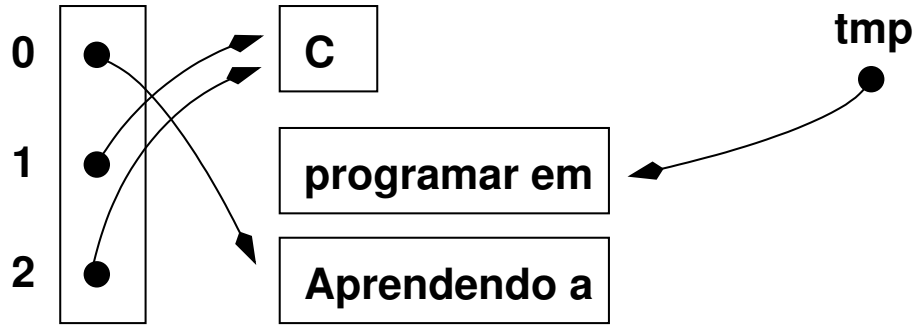
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
frases[2] = tmp;
tmp = frases[1];
```

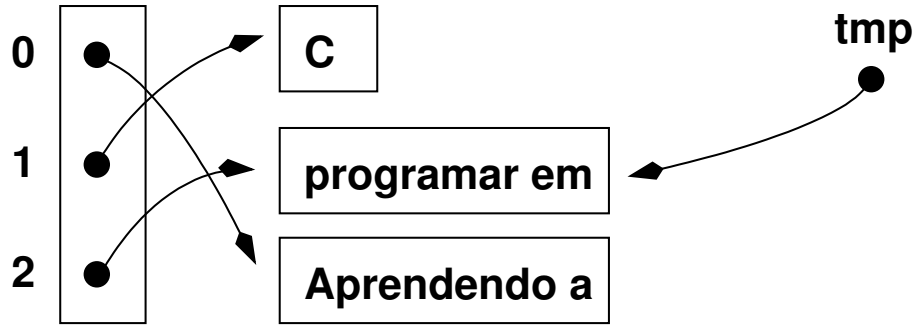
Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
frases[2] = tmp;
tmp = frases[1];
frases[1] = frases[2];
```

Vetores de ponteiros. Exemplo: ordenar vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

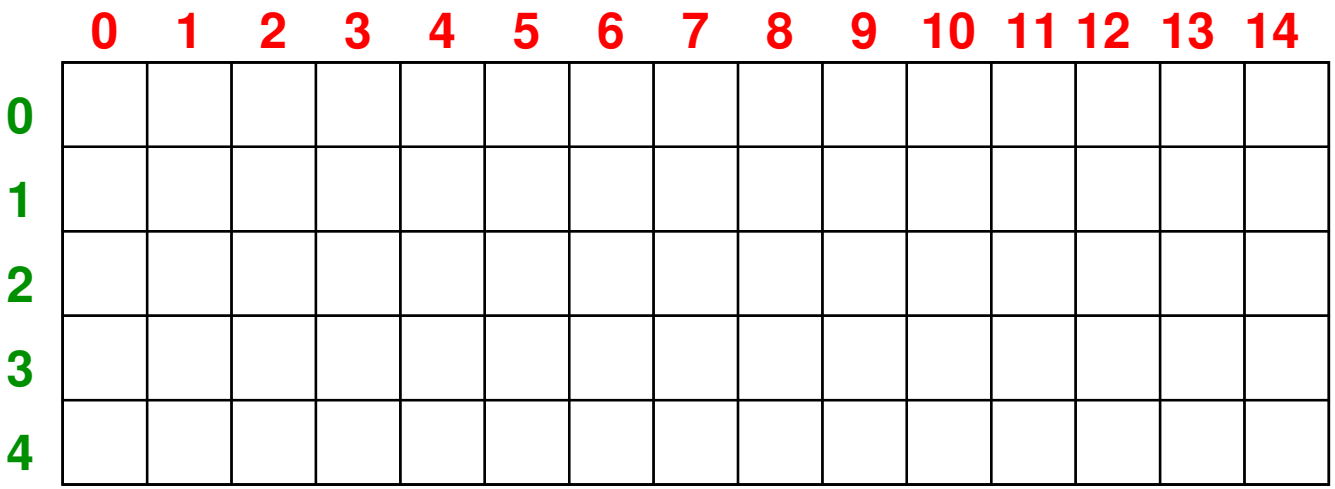
```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
frases[2] = tmp;
tmp = frases[1];
frases[1] = frases[2];
frases[2] = tmp;
```

Ponteiros vs. vetores multidimensionais

- A definição **char a[5][15]** reserva 75 posições para caracteres
  - Através do cálculo $15 \times \textit{linha} + \textit{coluna}$ é possível encontrar o elemento `a[linha][coluna]`
- A definição **char *b[5]** somente reserva 5 posições para ponteiros e não os inicia com valores
  - Assumindo que cada elemento de **b** aponte para um vetor de caracteres de tamanho 15, então teríamos reservadas 75 posições de caracteres + 5 posições para ponteiros
- Deixando a suposição acima de lado, a grande vantagem do vetor de ponteiros consiste em permitir linhas de diferentes tamanhos no vetor
  - Cada elemento em **b** pode apontar para vetores de caracteres de diferentes tamanhos

Ponteiros vs. vetores multidimensionais

```
char alunos[ 5 ][ 15] = {  "Pedro", "Juca", "Maria", "Alice"};
```



Ponteiros vs. vetores multidimensionais

char alunos[5][15] = { "Pedro", "Juca", "Maria", "Alice"};

L = LIXO

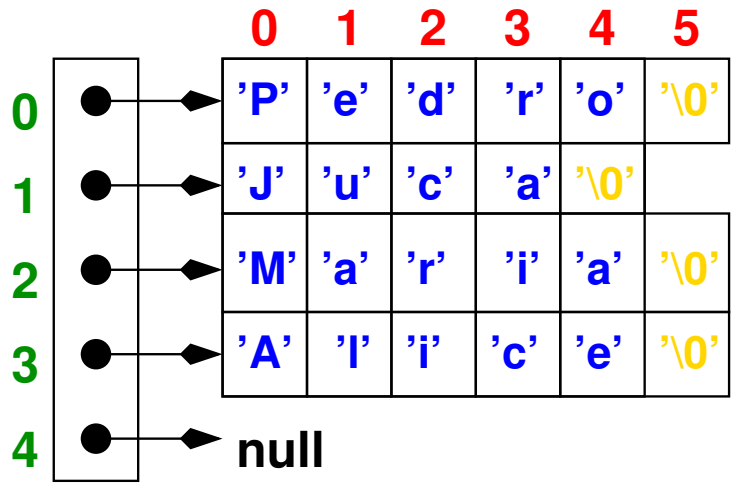
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'	L	L	L	L	L	L	L	L	L
1	'J'	'u'	'c'	'a'	'\0'	L	L	L	L	L	L	L	L	L	L
2	'M'	'a'	'r'	'i'	'a'	'\0'	L	L	L	L	L	L	L	L	L
3	'A'	'l'	'i'	'c'	'e'	'\0'	L	L	L	L	L	L	L	L	L
4	'\0'	L	L	L	L	L	L	L	L	L	L	L	L	L	L

Ponteiros vs. vetores multidimensionais

```
char *nomes[ 5 ] = { "Pedro", "Juca", "Maria", "Alice"};
```

Ponteiros vs. vetores multidimensionais

```
char *nomes[ 5 ] = { "Pedro", "Juca", "Maria", "Alice"};
```



Sumário

Sumário

Do que é formado um programa em C?

- Programas em C se resumem em um conjunto de definições de variáveis e funções
  - Todo programa em C deve possuir pelo menos a função **main()**

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5     printf("Ola mundo!");
6     return 0;
7  }
```

- É possível deixar todo o código do programa dentro da função **main()**, porém não seria fácil administrá-lo e é provável que se tenha trechos de códigos redundantes dentro desta função

Objetivos de uma função

- O uso de funções permite que grandes tarefas computacionais sejam divididas em pequenas partes
  - propiciando um código mais legível e;
  - a reutilização de código – uma função pode ser reutilizada diversas vezes no programa
- Cada função é destinada a realizar uma computação específica cujo detalhes não são necessários para as demais partes do programa
  - As funções escondem detalhes de implementação das demais partes do programa
  - Imagine a relação entre um gerente e seus subordinados

Declaração e definição de funções

```
1  #include <stdio.h>
2
3  /* Declaracao da funcao - ou prototipo da funcao */
4
5  tipo_de_retorno nome_da_funcao(lista de parametros);
6
7  /* Definicao ou implementacao da funcao */
8
9  tipo_de_retorno nome_da_funcao(lista de parametros){
10
11    /* Instrucoes */
12
13   return (valor de acordo com o tipo de retorno)
14 }
```

Exemplo: Função ola()

```
1  #include<stdio.h>
2
3  /* Declaracao da funcao*/
4  void ola(void);
5
6  /* Definicao ou implementacao da funcao */
7  void ola(void){
8     printf("Ola mundo!");
9  }
10
11 int main(void)
12 {
13    ola(); // invocando a funcao ola
14    return 0;
15 }
```

Características de uma função

- As funções são invocadas dentro do corpo de outras funções (Ex: dentro da função *main*) e a **comunicação** entre funções é feita através da lista de parâmetros e do valor de retorno
- A **lista de parâmetros** permite que dados externos sejam fornecidos à função.
  - A função pode receber qualquer quantidade de parâmetros, inclusive é possível que uma função não receba parâmetro algum
- O **valor de retorno** é o valor que uma função retorna para a função que a invocou.
  - É possível retornar valores pela lista de parâmetros. Detalhes mais adiante

Exemplo: Comunicação entre funções

```
1  /* Declaracao */
2  void digaOla(void);
3  int maior(int, int);
4
5  /* Definicao ou implementacao */
6  void digaOla(void){
7     printf("Ola mundo!\n");
8  }
9
10 int maior(int a, int b){
11    int x = a;
12    if (b > x){
13       x = b;
14    }
15    return (x);
16 }
```

Exemplo: Comunicação entre funções

```
1  int main(void){
2     int i = 10, j = 20, k;
3     /* invocando a funcao que nao possui
4       lista de parametros e nem retorno */
5     digaOla();
6
7     /* invocando funcao que possui lista
8       de parametros e retorno */
9     k = maior(i, j);
10
11    return 0;
12 }
```

Funções: exercício

1. Desenvolver uma função que retorne o nome do mês relacionado ao número que foi fornecido via parâmetros. Se o número fornecido estiver fora da faixa de 1 a 12, deve retornar a mensagem: "Número inválido".

```
1 /* Declaracao */
2 char *mes(int);
3
4 int main(void){
5    printf("Estamos em %s", mes(4)); // resultado: abril
6    return 0;
7 }
```

Sumário

Chamada por valor e por referência

Os parâmetros de uma função podem ser passados de duas formas:

- **Chamada por valor**
  - Não permite que a função que está sendo invocada modifique as variáveis fornecidas pela função que a invocou
  - A função invocada cria uma cópia local da variável e executa a computação sobre essa cópia
- **Chamada por referência**
  - Permite que a função que está sendo invocada modifique as variáveis fornecidas pela função que a invocou
  - A função invocada recebe os endereços de memória das variáveis fornecidas e com estes endereços é possível que a função modifique os valores lá armazenados

Chamada por valor

```
1  /* Declaracao das funcoes */
2  void troca(int, int);
3
4  /*passagem por valor*/
5  void troca(int a, int b){
6     int x;
7     x = a; a = b; b = x;
8  }
9  int main(void){
10    int i = 10, j = 20;
11    /* invocando funcao com passagem por valor*/
12    troca(i, j);
13
14    printf("i = %d, j = %d", i, j); //i continua com 10 e j com 20
15
16    return 0;
17 }
```

Chamada por referência

```
1  /* Declaracao das funcoes */
2  void trocaRef(int *, int *);
3
4  /*passagem por referencia*/
5  void trocaRef(int *a, int *b){
6     int x;
7     x = *a; *a = *b; *b = x;
8  }
9  int main(void){
10    int i = 10, j = 20;
11    /* invocando funcao com passagem por referencia */
12    trocaRef(&i, &j);
13
14    printf("i = %d, j = %d", i, j); // i passa a ter 20 e j 10
15
16    return 0;
17 }
```

Analisando a chamada por referência

- Em C, a chamada por referência é feita através da utilização de ponteiros

```
1  void trocaRef(int *a, int *b){
2     int x;
3     x = *a; *a = *b; *b = x;
4  }
5  int main(){
6     int i = 10, j = 20;
7     trocaRef(&i, &j);
8  }
```

- Os argumentos ***a** e ***b** são ponteiros para `int` que esperam receber endereços
- O operador & fornece o endereço da variável que este precede, assim **&i** e **&j** retornam os endereços de **i** e **j**, respectivamente.

Sumário

Argumentos de linha de comando

- Permite que argumentos sejam fornecidos a um programa no momento de sua execução
  - Por exemplo, ao invocar o comando `mkdir` para criar um diretório, é necessário informar ainda o nome do diretório. Este nome é o argumento que é passado para o programa `mkdir`
- Todo programa C ao ser executado, invoca a função **main** e os argumentos de linha de comando são passados a esta função
  - Como qualquer função em C, a função **main** também pode receber argumentos

Argumentos de linha de comando

São definidos dois argumentos, que por convenção são chamados de **argc** e **argv**

- **int argc** (*argument count*)
  - Armazena o número de argumentos que foram fornecidos pela linha de comando
- **char *argv[]** (*argument vector*)
  - É um ponteiro para um vetor de caracteres o qual armazena de fato os argumentos fornecidos
  - Por convenção em `argv[0]` está contido o nome do programa que foi invocado, assim o valor mínimo armazenado em `argc` será 1

Argumentos de linha de comando

```c
#include<stdio.h>

int main(int argc, char *argv[]){
   int i;

   /* Percorrendo a lista de argumentos */
   for(i = 0; i < argc; i++){
      printf("argumento[%d]: %s\n", i, argv[i]);
   }

   return 0;
}
```

- Faça o teste: Execute este programa sem fornecer argumentos e depois execute-o fornecendo argumentos
  - Exemplo: `./programa ola mundo`

Argumentos de linha de comando: exercício

1. Desenvolva uma calculadora que receba dois operandos e um operador através de argumentos de linha de comando e exiba o resultado da operação. **Exemplo: 10 + 20**. Cada operação deverá ser implementada em funções e deverá prover suporte as seguintes operações: soma (+), subtração (-), multiplicação (*), divisão (/) e exponenciação (^). Deve-se ainda tratar possíveis erros de entrada do usuário e na ocorrência destes, informar o usuário sobre a maneira correta de fazer uso da calculadora.

Sumário

Recursividade

- Dentro de cada função é possível invocar qualquer outra função conhecida, inclusive é possível que uma função faça uma invocação a si própria. Tal caso é conhecido como **recursividade**.
- Códigos recursivos são geralmente mais compactos e mais fáceis de escrever e entender que seus equivalentes não recursivos
  - Contudo, não serão mais rápidos e não resultará em economia de recursos

Recursividade: fatorial não recursivo

```c
#include <stdio.h>
int fatorial(int);

int fatorial(int n){
   int r = 1;
   while( n > 0){
      r*=n;
      n--;
   }
   return r;
}

int main(void){
   int resultado;

   resultado = fatorial(3);
   printf("O fatorial de 3 e: %d", resultado);
   return 0;
}
```

Recursividade: fatorial recursivo

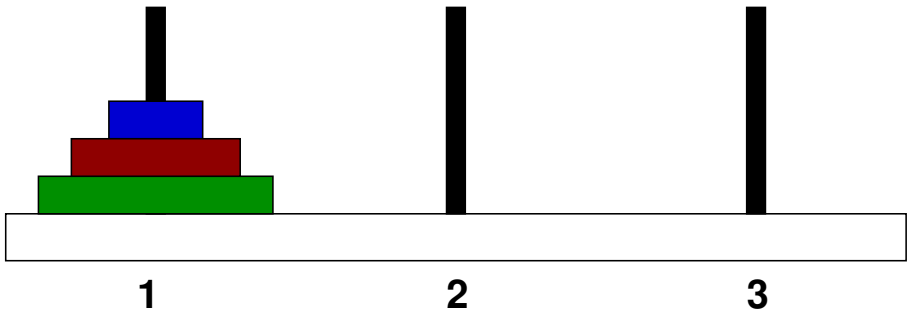
```c
#include <stdio.h>
int fatorial(int);

int fatorial(int n){
   return (n > 0) ? (n * fatorial(n - 1)) : 1;
}

int main(void){
   int resultado;

   resultado = fatorial(3);
   printf("O fatorial de 3 e: %d", resultado);

   return 0;
}
```

Exercício

Torre de Hanoi

Desenvolver um programa que apresente a evolução (solução passo a passo) do jogo Torre de Hanoi. O objetivo do jogo é transportar os discos do pino 1 para o pino 3. O pino 2 pode ser usado como auxiliar. As regras do jogo são:

1. Só pode movimentar um disco por vez e somente o disco que estiver por cima
2. Um disco maior não pode ficar sobre um disco menor

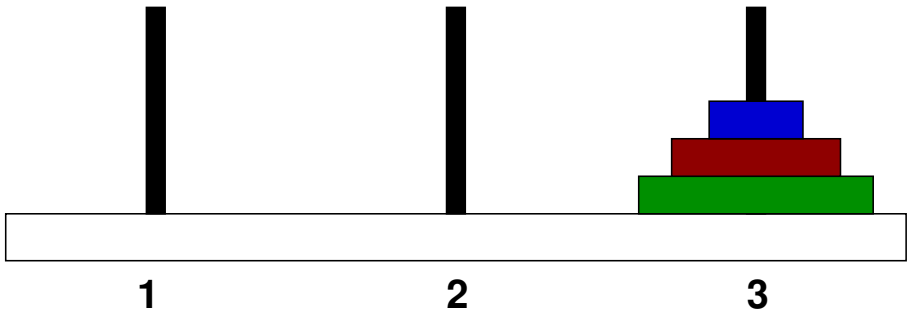


Exercício

Torre de Hanoi

Desenvolver um programa que apresente a evolução (solução passo a passo) do jogo Torre de Hanoi. O objetivo do jogo é transportar os discos do pino 1 para o pino 3. O pino 2 pode ser usado como auxiliar. As regras do jogo são:

1. Só pode movimentar um disco por vez e somente o disco que estiver por cima
2. Um disco maior não pode ficar sobre um disco menor



Sumário

Sumário

Estruturas: struct

- Além dos tipos básicos (`char`, `int`, `float`, `double`) a linguagem C permite aos desenvolvedores que construam tipos compostos, chamados de **estruturas** (*struct*)
- Uma estrutura consiste na coleção de uma ou mais variáveis, de diferentes tipos ou não, agrupadas sob um único nome

Exemplo

Um ponto no plano cartesiano é representado pelas coordenadas X e Y. Com o uso de estruturas poderia-se agrupar essas coordenadas sob o nome único: **ponto**

Struct: sintaxe

```c
/*
 struct nome{
   membros da estrutura;
 };
*/

// Criando a estrutura: ponto
struct ponto{
   int x;
   int y;
}; //atencao a este ;

// criando variaveis do tipo ponto
struct ponto inicio;
struct ponto final = {10, 10}; //informando valores iniciais
```

Acessando os membros de uma estrutura

- O operador "." é usado para conectar o nome da estrutura aos seus membros

```
1  struct ponto{
2     int x;
3     int y;
4  };
5
6  // criando variaveis do tipo ponto
7  struct ponto p1;
8  struct ponto p2 = {10, 10}; //informando valores iniciais
9
10 // acessando os membros de uma estrutura
11 p1.x = 0;
12 p1.y = 1;
13
14 printf("Coordenadas iniciais, x: %d, y: %d\n", p1.x, p1.y);
```

Membros da estrutura

- Os membros de uma estrutura podem ser variáveis de qualquer tipo básico ou ainda tipos compostos, ou seja, de outras estruturas
- A estrutura **ponto** nos permitiu agrupar duas variáveis (X, Y) para representar um ponto no plano cartesiano
- Sabemos que um retângulo é identificado por 2 pontos no plano cartesiano. Assim, poderíamos:

Membros da estrutura

- Os membros de uma estrutura podem ser variáveis de qualquer tipo básico ou ainda tipos compostos, ou seja, de outras estruturas
- A estrutura **ponto** nos permitiu agrupar duas variáveis (X, Y) para representar um ponto no plano cartesiano
- Sabemos que um retângulo é identificado por 2 pontos no plano cartesiano. Assim, poderíamos:
  - Criar duas variáveis do tipo **ponto**, ou;

Membros da estrutura

- Os membros de uma estrutura podem ser variáveis de qualquer tipo básico ou ainda tipos compostos, ou seja, de outras estruturas
- A estrutura **ponto** nos permitiu agrupar duas variáveis (X, Y) para representar um ponto no plano cartesiano
- Sabemos que um retângulo é identificado por 2 pontos no plano cartesiano. Assim, poderíamos:
  - Criar duas variáveis do tipo **ponto**, ou;
  - Criar uma nova estrutura que tivesse como membros duas variáveis do tipo **ponto**

Membros da estrutura

- Os membros de uma estrutura podem ser variáveis de qualquer tipo básico ou ainda tipos compostos, ou seja, de outras estruturas
- A estrutura **ponto** nos permitiu agrupar duas variáveis (X, Y) para representar um ponto no plano cartesiano
- Sabemos que um retângulo é identificado por 2 pontos no plano cartesiano. Assim, poderíamos:
  - Criar duas variáveis do tipo **ponto**, ou;
  - Criar uma nova estrutura que tivesse como membros duas variáveis do tipo **ponto**

```
1  struct retangulo{
2     struct ponto p1;
3     struct ponto p2;
4  };
5  struct retangulo ret;
6
7  ret.p1.x = 0; ret.p1.y = 0;
8  ret.p2.x = 3; ret.p2.y = 4;
```

Estruturas: exemplo

```
1  #include<stdio.h>
2
3  // declarado fora da funcao main
4  struct ponto{
5     int x, y;
6  };
7
8  int main(void){
9
10    struct ponto p1={0,1};
11
12    printf("x: %d, y: %d\n", p1.x, p1.y);
13
14    return 0;
15 }
```

Ponteiros para estruturas

- É possível passar uma estrutura como parâmetro em uma função
  - É mais eficiente fazer essa passagem por referência, pois não necessita fazer uma cópia de toda a estrutura
- Abaixo a forma para declarar um ponteiro para uma estrutura e a forma para acessar os membros desta estrutura

```
1  struct ponto inicio = {1,2};
2  struct ponto *p;
3  // atribuindo o endereco de inicio ao p
4  p = &inicio;
5
6  // acessando os elementos da estrutura atraves do ponteiro p
7  // necessita do () por causa da precedencia: . e' maior que *
8  printf("x: %d, y: %d\n", (*p).x, (*p).y);
9
10 // forma simplificada, usando o operador ->
11 printf("x: %d, y: %d\n", p->x, p->y);
```

Estruturas, vetores e ponteiros: exemplo

```
1  void imprimir(struct ponto *tab){
2     printf("x: %d, y: %d\n", tab[0].x, tab[0].y);
3  }
4
5  int main(void){
6     struct ponto vetor[5];
7
8     vetor[0].x = 10;
9     vetor[0].y = 5;
10
11    imprimit(vetor);
12
13    return 0;
14 }
```

Sumário

Definição de tipos: typedef

- C permite a definição de novos nomes para tipos de dados
  - **Sintaxe:** `typedef tipo_dados novo_nome;`

```
1  typedef int Inteiro; // Novo nome para inteiro
2  Inteiro x = 10;
3
4  typedef char *String; // Nome String para ponteiro de
       caracteres
5  String nome = "Curso de C";
6
7  typedef struct ponto{ // Nome para estrutura
8     int x,y;
9  } Ponto;
10
11 Ponto p1 = {1,2};
```

Tipos de dados compostos: exercício

- Desenvolver uma aplicação para cadastro de clientes. O programa deverá prover um menu para que o usuário possa invocar as operações: cadastrar, consultar, listar, editar e remover clientes.
  - Deve-se fazer uso de ponteiros, funções, vetores, estruturas, definição de tipos
- Ao realizar as operações de cadastro, edição e remoção deve-se garantir que o número máximo de clientes não foi atingido, ou se o cliente existe para ser editado ou removido
- Deve-se armazenar as seguintes informações do cliente:
  - nome; cpf; telefone; data de nascimento

Sumário

Arquivos e fluxos

- C provê um conjunto padrão de interfaces de entrada e saída independente do dispositivo real que é acessado
  - terminais, arquivos em disco, fita
- C implementa uma abstração entre o programador e o dispositivo utilizado, assim os programadores só lidam com a abstração e não precisam se preocupar com qual dispositivo real que está sendo acessado
  - Essa abstração é chamada de **fluxos** (*streams*) e o dispositivo real é chamado de **arquivo**
- Tipos de fluxos

  Texto. Sequência de caracteres

  Binário. Sequência de *bytes*

Arquivos

- Em C um **arquivo** pode ser um arquivo em disco, um terminal ou até uma impressora
  - Nem todos os arquivos apresentam os mesmos recursos. Um arquivo em disco pode ser acessado de forma aleatória, um teclado já não permite
  - Todos os fluxos são iguais, mas não todos os arquivos
- A associação de um fluxo a um arquivo ocorre através da operação de abertura
  - Após isto, informações podem ser trocadas entre o arquivo e o programa
- É importante lembrar de fechar os arquivos abertos. Só assim as informações que estão em um área de armazenamento temporário serão descarregadas para o arquivo
  - Todos arquivos são fechados de forma automática após o término correto da função **main**.

INSTITUTO FEDERAL

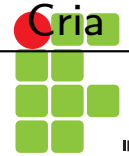
Funções para trabalhar com arquivos

Função	Descrição
`FILE *fopen(char *, char *)`	Abre um arquivo
`int fclose(FILE *)`	Fecha um arquivo
`int putc(int, FILE *)`	Escreve um caractere no arquivo
`int fputs(char *, FILE *)`	Escreve uma cadeia de caractere no arquivo
`int getc(FILE *)`	Lê um caractere de um arquivo
`int fgets(char *, int tam, FILE *)`	Lê uma cadeia de caracteres de tamanho *tam*
`int fseek(FILE *, long, int)`	Posiciona o arquivo em um byte específico
`int fprintf(FILE *, char *, ...)`	Semelhante ao printf
`int fscanf(FILE *, char *, ...)`	Semelhante ao scanf
`int feof(FILE *)`	Retorna verdadeiro se o fim do arquivo for atingido
`int ferror(FILE *)`	Retorna verdadeiro se ocorreu algum erro
`void rewind(FILE *)`	Posiciona no início do arquivo
`int remove(char *)`	Exclui um arquivo
`int fflush(FILE *)`	Descarrega dados do *buffer* no arquivo

Abrindo um arquivo

```c
FILE *arquivo;
arquivo = fopen("texto.txt", "w");

if (arquivo == NULL){
   printf("Erro: arquivo nao pode ser aberto\n");
   exit(1);
}
```

Modo		Descrição	Se o arquivo	
Texto	Binário		Existir	Não existir
r	rb	Abre para leitura	Abre	Erro
w	wb	Cria um arquivo	Sobreescreve	Cria
a	ab	Anexa a um arquivo	Anexa	Cria
r+	rb+	Abre para leitura e escrita	Abre	Erro
w+	wb+	Cria um arquivo para leitura e escrita	Sobreescreve	Cria
a+	ab+	Anexa a um arquivo para leitura e escrita	Anexa	Cria



Arquivo texto: Criando arquivo com putc

```c
int main(void){
   char letra;
   FILE *arq;

   if ((arq = fopen("teste.txt", "w")) != NULL){
      do{
         letra = getchar();
         putc(letra, arq);
      }while(letra != '0');
      fclose(arq);
   }else {
      printf("Erro: arquivo nao pode ser aberto\n");
   }
   return 0;
}
```

Arquivo texto: Criando arquivo com fputs

```c
int main(void){
   char frase[80];
   FILE *arq;

   if ((arq = fopen("teste.txt", "w")) != NULL){
      do{
         printf("Entre com a frase: ");
         fgets(frase, sizeof(frase), stdin);
         fputs(frase, arq);
      }while(*frase != '\n');
      fclose(arq);
   }else {
      printf("Erro: arquivo nao pode ser aberto\n");
   }
   return 0;
}
```

Arquivo texto: Lendo o conteúdo

```
1  int main(void){
2     char frase[80];
3     FILE *arq;
4
5     if ((arq = fopen("teste.txt", "r")) != NULL){
6        while(!feof(arq)){
7           fgets(frase, sizeof(frase), arq);
8           printf("%s",frase);
9        }
10    }else {
11       printf("Erro: arquivo nao pode ser aberto\n");
12    }
13    return 0;
14 }
```

INSTITUTO FEDERAL

Arquivo binário

- É possível gravar e ler estruturas definidas pelo usuário diretamente em um arquivo através das funções **fread** e **fwrite**
  - `size_t fread(void buffer, size_t tamanho_em_bytes, size_t contador, FILE *arq)`
  - `size_t fwrite(void *buffer, size_t tamanho_em_bytes, size_t contador, FILE *arq);`
- `buffer` é a região de memória para onde os dados obtidos do arquivo serão armazenados (`fread`) ou de onde os dados serão obtidos para serem armazenados no arquivo (`fwrite`)
- O `tamanho_em_bytes` especifica a quantidade de *bytes* que deverá ser lida ou gravada
- O contador indica quantos elementos (cada um de comprimento `tamanho_em_bytes`) será lido ou gravado
- `arq` é o ponteiro para o fluxo

Arquivo binário: exemplo: criando arquivo

```
1  typedef struct pessoa{
2     char nome[80];
3     int idade;
4  } Pessoa;
5
6  int main(void){
7     Pessoa p1 = {"Joao",20};
8     FILE *arq;
9
10    if ((arq = fopen("contatos.dat", "wb+")) != NULL){
11       fwrite(&p1, sizeof(p1), 1, arq);
12    }else {
13       printf("Erro: arquivo nao pode ser aberto\n");
14    }
15    return 0;
16 }
```

Arquivo binário: exemplo: lendo conteúdo

```
1  int main(void){
2     Pessoa p;
3     FILE *arq;
4
5     if ((arq = fopen("contatos.dat", "rb")) != NULL){
6
7        while(!feof(arq)){
8           fread(&p, sizeof(p), 1, arq);
9           if (!feof(arq){
10             printf("nome: %s\t idade: %d\n", p.nome, p.idade);
11          }
12       }
13    }else {
14       printf("Erro: arquivo nao pode ser aberto\n");
15    }
16    return 0;
17 }
```

Bibliografia


Brian W. Kernighan and Dennis M. Ritchie
The C Programming Language, 2nd edition
Prentice Hall, 1988


Herbert Schildt
C Completo e Total, 3ª edição
Makron Books, 1996.