Ponteiros

Aula 03

Ivone P. Matsuno Yugoshi
ivone.matsuno@ufms.br

Ronaldo Fiorilo dos Santos
ronaldo.santos@ufms.br

Universidade Federal de Mato Grosso do Sul
Câmpus de Três Lagoas
Bacharelado em Sistemas de Informação

Algoritmos e Programação II

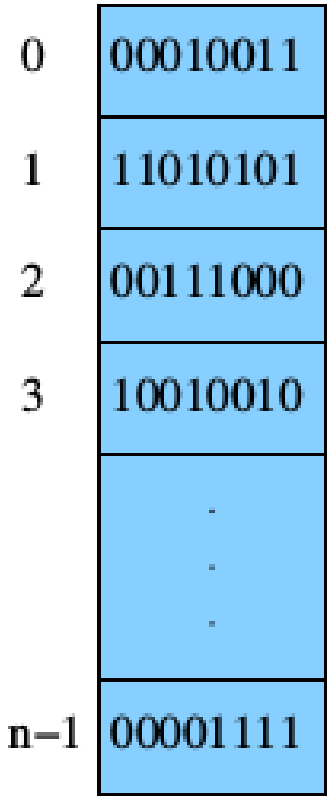
Introdução

- Ponteiros ou apontadores (do inglês *pointers*)
- Característica da linguagem C (mais poder e flexibilidade)
- Estruturas de dados complexas, modificação de argumentos passados a funções, alocação dinâmica de memória, etc.

Introdução

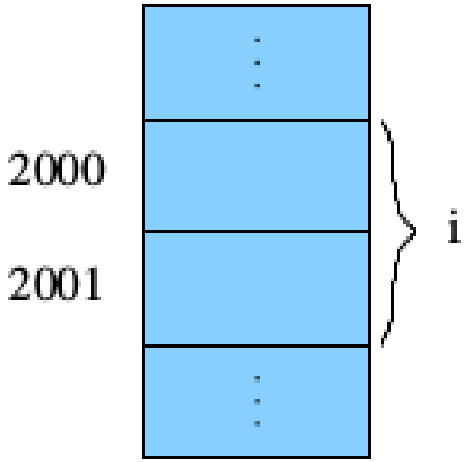
- A memória de um computador é constituída de muitas posições dispostas continuamente, cada qual podendo armazenar um valor na base binária
- Ou seja, a memória é um grande vetor que pode armazenar valores na base binária e que, por sua vez, esses valores podem ser interpretados como valores de diversos tipos
- Os índices desse vetor, numerados sequencialmente a partir de 0 (zero), são chamados de **endereços de memória**

Introdução



Introdução

- Cada variável do programa ocupa um ou mais bytes na memória. O endereço do primeiro byte de uma variável é dito ser o endereço da variável



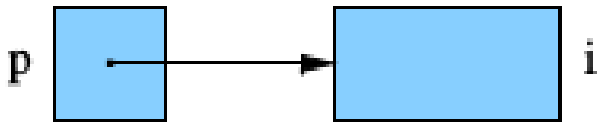
- No exemplo acima, a variável *i* ocupa os bytes dos endereços 2000 e 2001. Logo, o endereço da variável *i* é 2000.

Variáveis ponteiros

- **Ponteiro** é uma variável que armazena um valor especial, que é um endereço de memória, e por isso nos permite acessar indiretamente o valor armazenado nesse endereço
- **Indireção**, isto é, **acesso indireto** a um valor armazenado em algum ponto da memória

Variáveis ponteiros

- Quando armazenamos o endereço de uma variável *i* em uma variável ponteiro *p*, dizemos que *p* **aponta para** *i*
- Um ponteiro nada mais é que um endereço e uma variável ponteiro é uma variável que pode armazenar endereços
- Ao invés de mostrar endereços como números, usaremos uma notação simplificada para indicar que uma variável ponteiro *p* armazena o endereço de uma variável *i*: mostraremos o conteúdo de *p* – um endereço – como uma flecha orientada na direção de *i*



Variáveis ponteiros

- Declaração de uma variável ponteiro:

```
int *p;
```

- A linguagem C obriga que toda variável ponteiro aponte apenas para objetos de um tipo particular, chamado de **tipo referenciado**
- variável ponteiro $=$ ponteiro

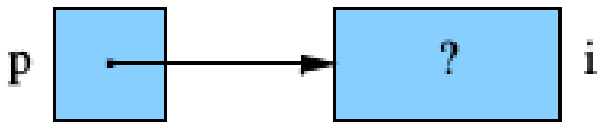
Operadores de endereçamento e de indireção

- A linguagem C possui dois operadores para uso específico com ponteiros
  - **operador de endereçamento (ou de endereço)**, utilizado para obter o endereço de uma variável, cujo símbolo é `&`
    - Se *v* é uma variável, então `&`*v* é seu endereço na memória
  - **operador de indireção**, utilizado para ter acesso ao objeto que um ponteiro aponta, cujo símbolo é `*`
    - Se *p* é um ponteiro, então `*`*p* representa o objeto para o qual *p* aponta no momento

Operadores de endereçamento e de indireção

- A declaração de uma variável ponteiro reserva um espaço na memória para um ponteiro mas não a faz apontar para um objeto
- É crucial inicializar um ponteiro antes de usá-lo
- Uma forma de inicializar um ponteiro é atribuir-lhe o endereço de alguma variável usando o operador **&**

```
int i, *p;

p = &i;
```



Operadores de endereçamento e de indireção

- Uma vez que uma variável ponteiro aponta para um objeto, podemos usar o operador de indireção `*` para acessar o valor armazenado no objeto
  - Se *p* aponta para *i*, por exemplo, podemos imprimir o valor de *i* de forma indireta

```
printf("%d\n", *p);
```

  - A função **`printf`** mostrará o valor de *i* e não o seu endereço

Operadores de endereçamento e de indireção

- Aplicar o operador `&` a uma variável produz um ponteiro para a variável e aplicar o operador `*` para um ponteiro retoma o valor original da variável

```
j = *&i;
```

é o mesmo que

```
j = i;
```

Operadores de endereçamento e de indireção

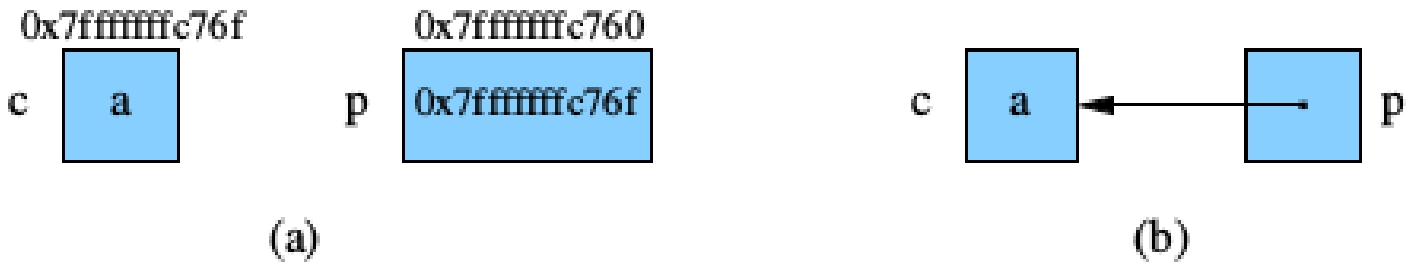
- Enquanto dizemos que *p* **aponta para** *i*, dizemos também que $*p$ é um **apelido** para *i*
- Não apenas $*p$ tem o mesmo valor que *i*, mas alterar o valor de $*p$ altera também o valor de *i*
- Uma observação importante que auxilia a escrever e ler programas com variáveis ponteiros é "traduzir" os operadores unários:
  - de endereço: `&x` para *endereço da variável x*
  - de indireção: `*x` para *conteúdo da variável apontada por x*

Operadores de endereçamento e de indireção

```c
#include <stdio.h>

int main(void)
{
   char c, *p;

   p = &c;
   c = 'a';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   c = '/';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   *p = 'Z';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   return 0;
}
```

Note que um endereço pode ser impresso pela função `printf` usando o especificador de tipo `%p`

Operadores de endereçamento e de indireção



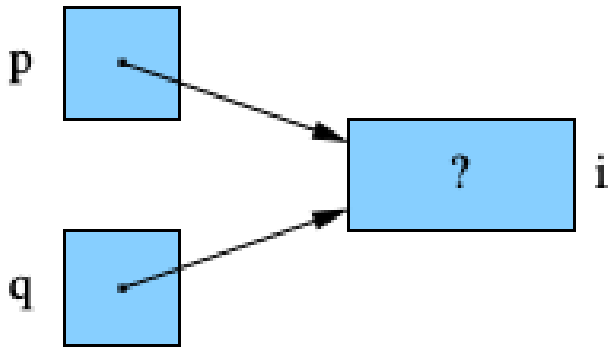

```
&c = 0x7fffffffc76f    c = a
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = a

&c = 0x7fffffffc76f    c = /
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = /

&c = 0x7fffffffc76f    c = Z
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = Z
```

Operadores de endereçamento e de indireção

- A linguagem C permite ainda que o operador de atribuição copie ponteiros, supondo que possuam o mesmo tipo

```
int i, j, *p, *q;
```

```
p = &i;
q = p;
```



Ponteiros em expressões

- Ponteiros podem ser usados em expressões aritméticas de mesmo tipo que seus tipos referenciados
- Os operadores `&` e `*`, por serem operadores unários, têm precedência sobre os operadores binários das expressões aritméticas em que se envolvem

Ponteiros em expressões

```c
#include <stdio.h>

int main(void)
{
   int i, j, *p1, *p2;

   p1 = &i;
   i = 5;
   j = 2 * *p1 + 3;
   p2 = p1;

   printf("i = %d, &i = %p\n\n", i, &i);
   printf("j = %d, &j = %p\n\n", j, &j);
   printf("&p1 = %p, p1 = %p, *p1 = %d\n", &p1, p1, *p1);
   printf("&p2 = %p, p2 = %p, *p2 = %d\n\n", &p2, p2, *p2);

   return 0;
}
```

```
i =  5, &i = 0x7fffffffc55c
j = 13, &j = 0x7fffffffc558

&p1 = 0x7fffffffc550, p1 = 0x7fffffffc55c, *p1 = 5
&p2 = 0x7fffffffc548, p2 = 0x7fffffffc55c, *p2 = 5
```

Ponteiros e funções

- Aprendemos algumas “regras” de como construir funções que têm parâmetros de entrada e saída ou argumentos passados por referência
- Esses argumentos/parâmetros são na verdade ponteiros
- O endereço de uma variável é passado como argumento para uma função
- O parâmetro correspondente que recebe o endereço é então um ponteiro
- Qualquer alteração realizada no conteúdo do parâmetro tem reflexos externos à função, no argumento correspondente

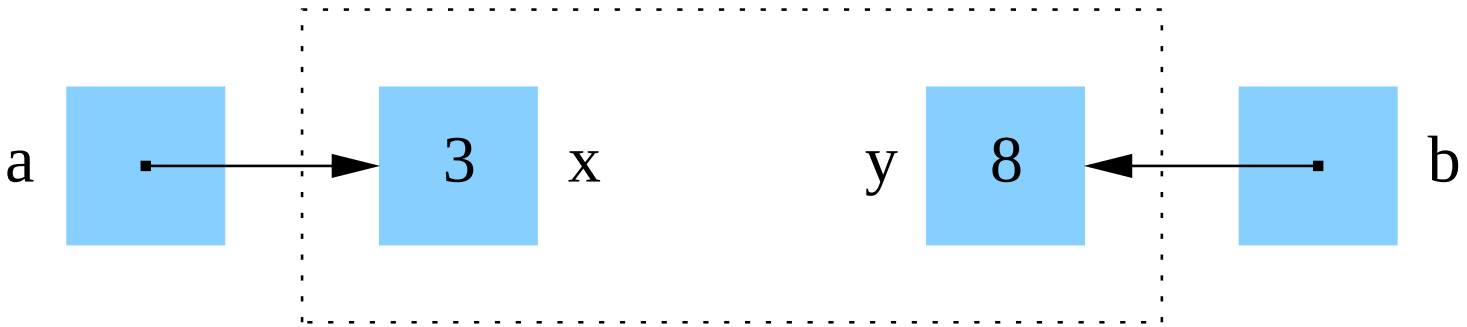
Parâmetros de entrada e saída?

```
#include <stdio.h>

void troca(int *a, int *b)
{
   int aux;

   aux = *a;
   *a = *b;
   *b = aux;
}

int main(void)
{
   int x, y;

   scanf("%d%d", &x, &y);
   printf("Antes da troca : x = %d e y = %d\n", x, y);
   troca(&x, &y);
   printf("Depois da troca: x = %d e y = %d\n", x, y);
   return 0;
}
```

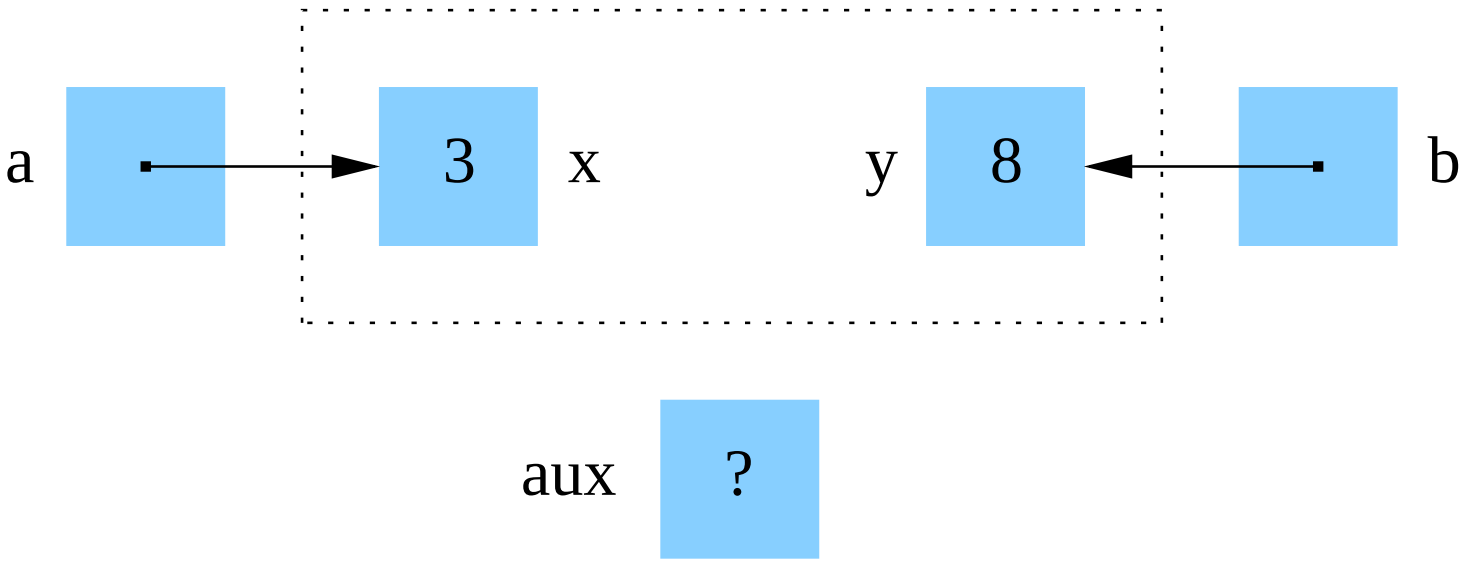
Parâmetros de entrada e saída?

Parâmetros de entrada e saída?

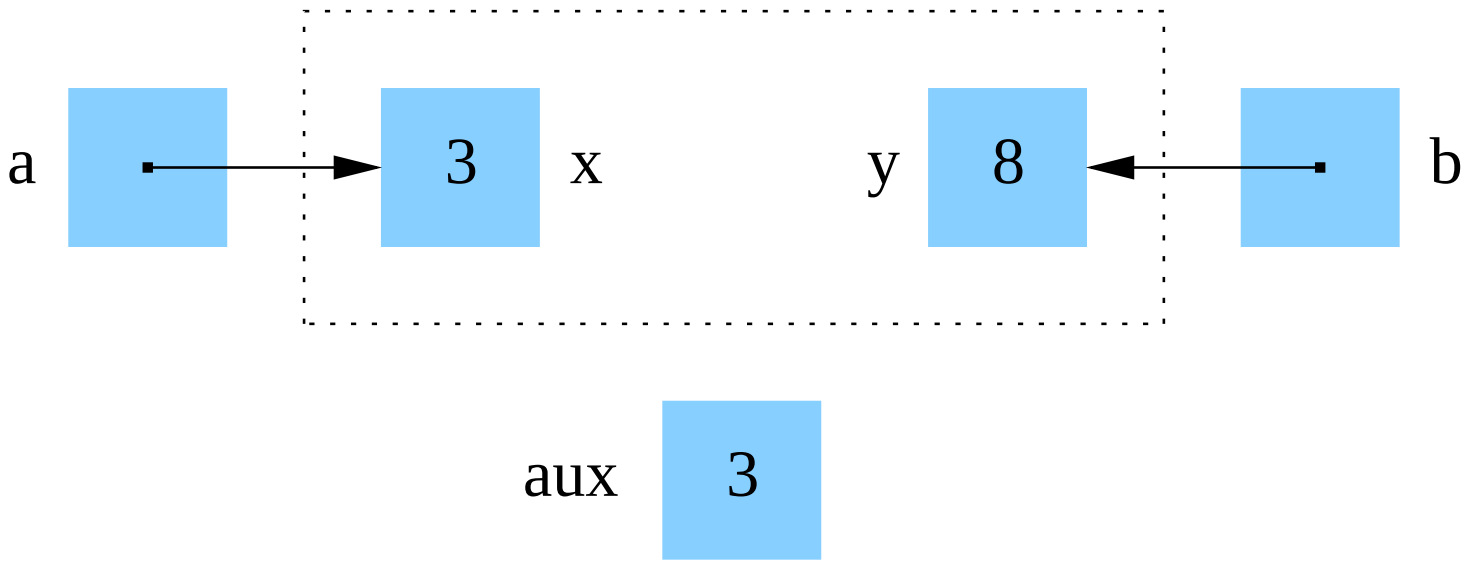
Parâmetros de entrada e saída?



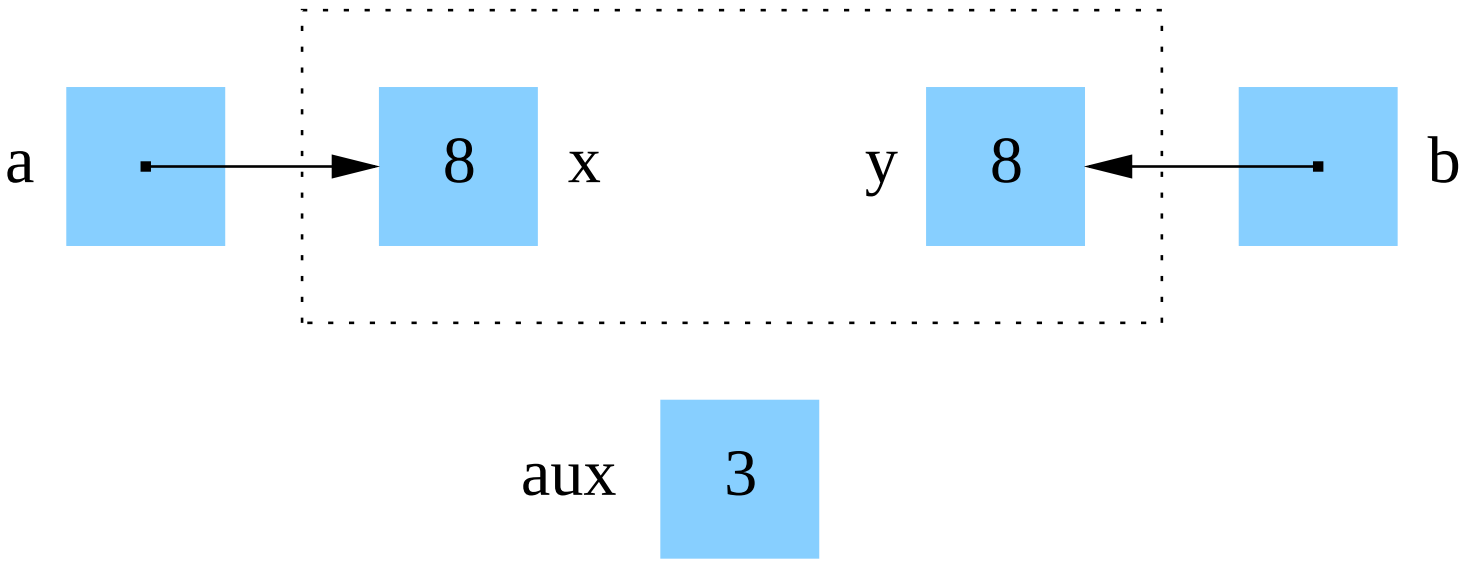
Parâmetros de entrada e saída?



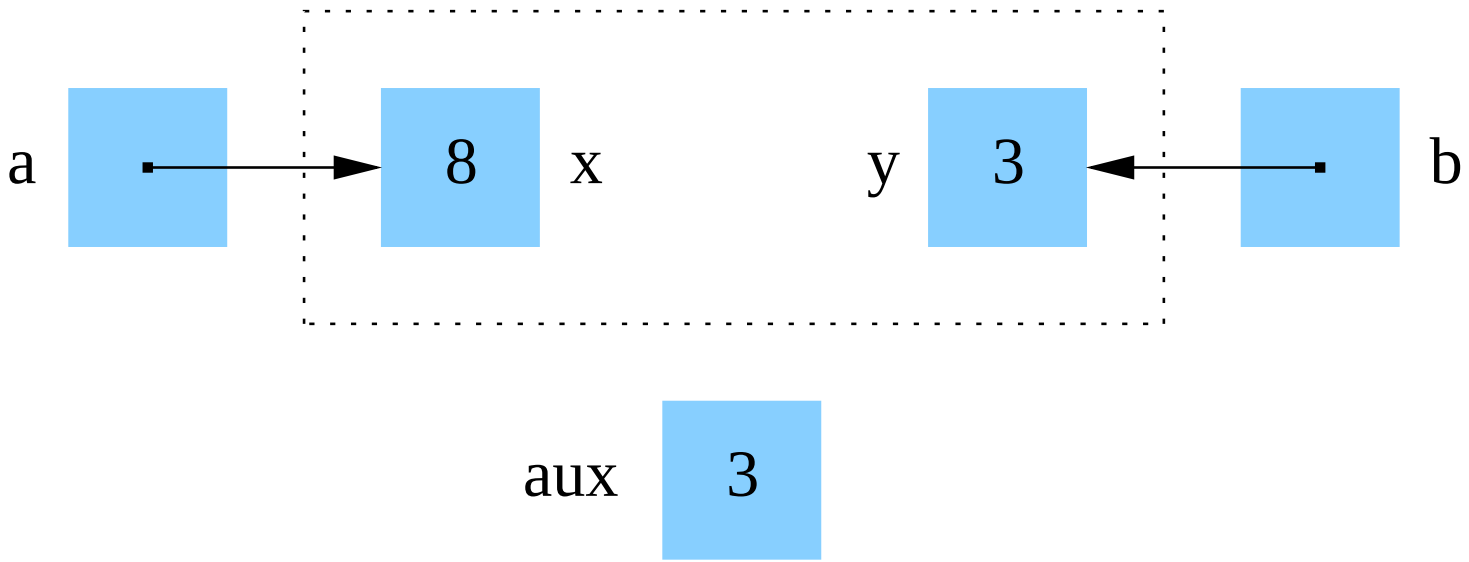
Parâmetros de entrada e saída?



Parâmetros de entrada e saída?



Parâmetros de entrada e saída?



Parâmetros de entrada e saída?

Parâmetros de entrada e saída?

- Esse exemplo destaca que são realizadas cópias do valores dos argumentos – que nesse caso são endereços das variáveis da função `main` – para os parâmetros respectivos da função `troca`
- No corpo dessa função, sempre que usamos o operador de indireção para acessar algum valor, estamos na verdade acessando o conteúdo da variável correspondente dentro da função `main`, que chamou a função `troca`
- Isso ocorre com as variáveis `x` e `y` da função `main`, quando copiamos seus endereços nos parâmetros `a` e `b` da função `troca`

Parâmetros de entrada e saída?

- Cópia???
- Aprendemos que parâmetros passados desta mesma forma são parâmetros de entrada e saída, ou seja, são parâmetros passados por referência e não por cópia
- Só há passagem de argumentos por cópia na linguagem C
- Não há passagem de argumentos por referência na linguagem C
- Simulamos a passagem de um argumento por referência usando ponteiros

Parâmetros de entrada e saída?

- Ao passar (por cópia) o endereço de uma variável como argumento para uma função, o parâmetro correspondente deve ser um ponteiro e, mais que isso, um ponteiro para a variável correspondente cujo endereço foi passado como argumento
- Qualquer modificação indireta realizada no corpo dessa função usando esse ponteiro será realizada na verdade no conteúdo da variável apontada pelo parâmetro, que é simplesmente o conteúdo da variável passada como argumento na chamada da função
- Não há nada de errado com o que aprendemos nas aulas anteriores sobre argumentos de entrada e saída, isto é, passagem de argumentos por referência
- Passagem de argumentos por referência é um tópico conceitual quando falamos da linguagem de programação C

Devolução de ponteiros

- Além de passar ponteiros como argumentos para funções também podemos fazê-las devolver ponteiros
  - comuns, por exemplo, quando tratamos de cadeias de caracteres
- ATENÇÃO! não é possível que uma função devolva o endereço de uma variável local sua, já que ao final de sua execução, essa variável será destruída

Devolução de ponteiros

```
int *max(int *a, int *b)
{
   if (*a > *b)
      return a;
   else
      return b;
}
```

```
   int i, j, *p;
   .
   .
   .
   p = max(&i, &j);
```

Ponteiros e vetores

- A linguagem C nos permite usar expressões aritméticas com ponteiros que apontam para elementos de vetores
- Forma alternativa de trabalhar com vetores e seus índices
- Relação íntima entre ponteiros e vetores na linguagem C
- Programa executável mais eficiente quando usamos ponteiros para vetores

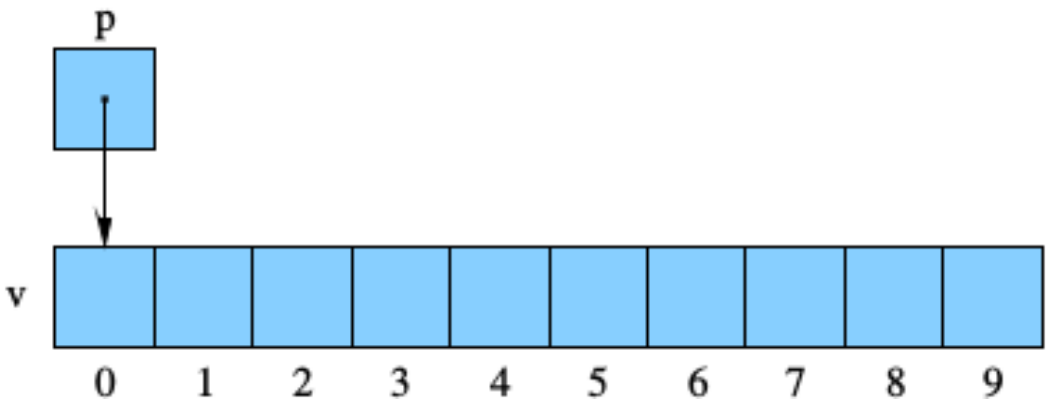
Aritmética com ponteiros

- Suponha que temos declaradas as seguintes variáveis:

```
int v[10], *p;
```

- Podemos fazer o ponteiro *p* apontar para o primeiro elemento do vetor *v* fazendo a seguinte atribuição:

```
p = &v[0];
```



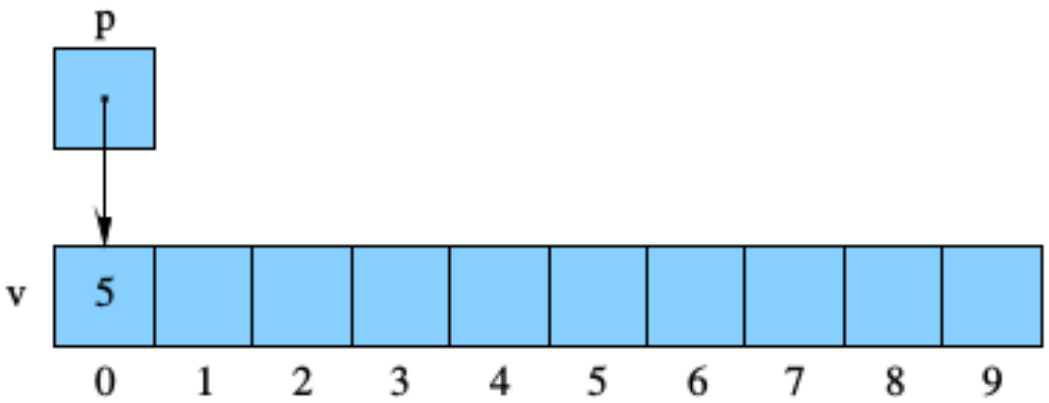
Aritmética com ponteiros

- Suponha que temos declaradas as seguintes variáveis:

```
int v[10], *p;
```

- Podemos acessar o primeiro compartimento de *v* através de *p*:

```
*p = 5;
```



Aritmética com ponteiros

- Podemos executar **aritmética com ponteiros** ou **aritmética com endereços** sobre *p* e assim acessamos outros elementos do vetor
- A linguagem C possibilita três formas de aritmética com ponteiros:
  - adicionar um número inteiro a um ponteiro
  - subtrair um número inteiro de um ponteiro
  - subtrair um ponteiro de outro ponteiro

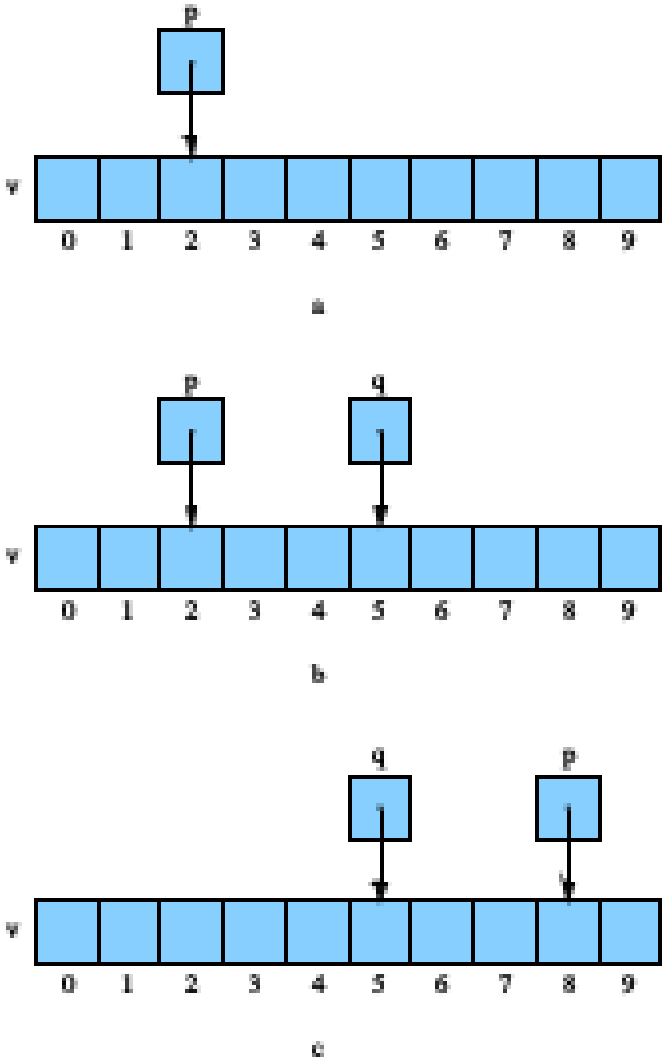
Aritmética com ponteiros

- Suponha que temos declaradas as seguintes variáveis:

```
int v[10], *p, *q, i;
```

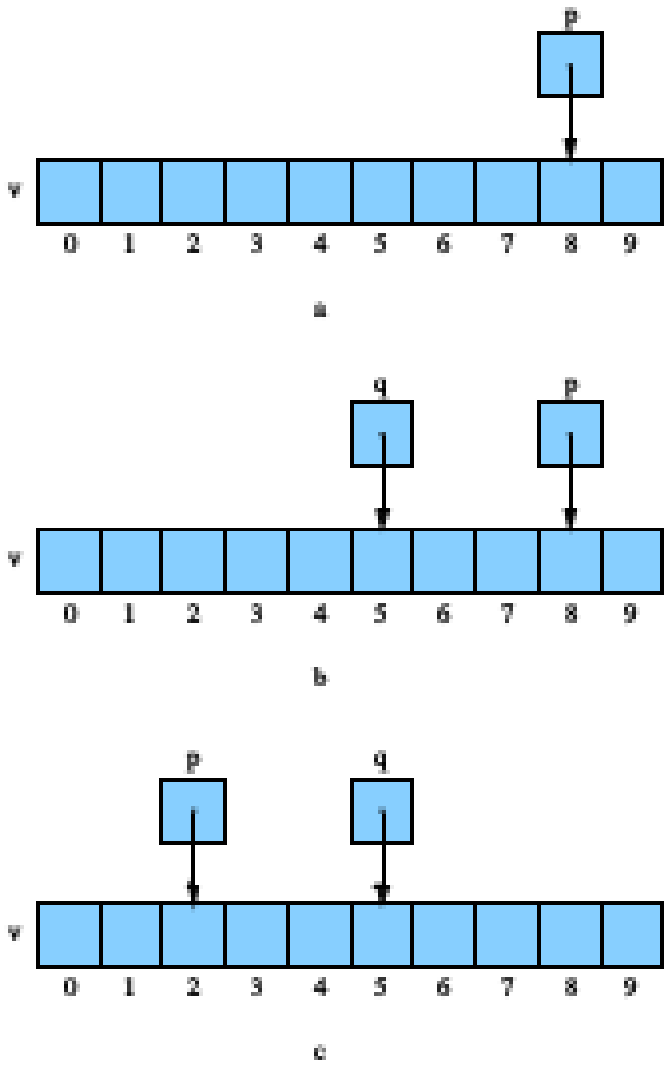
- Se p aponta para o elemento $v[i]$, então $p+j$ aponta para $v[i+j]$
- Se p aponta para o elemento $v[i]$, então $p-j$ aponta para $v[i-j]$
- Se p aponta para $v[i]$ e q aponta para $v[j]$, então $p-q$ é igual a $i-j$
  - quando um ponteiro é subtraído de outro, o resultado é a distância, medida em elementos do vetor, entre os ponteiros

Aritmética com ponteiros



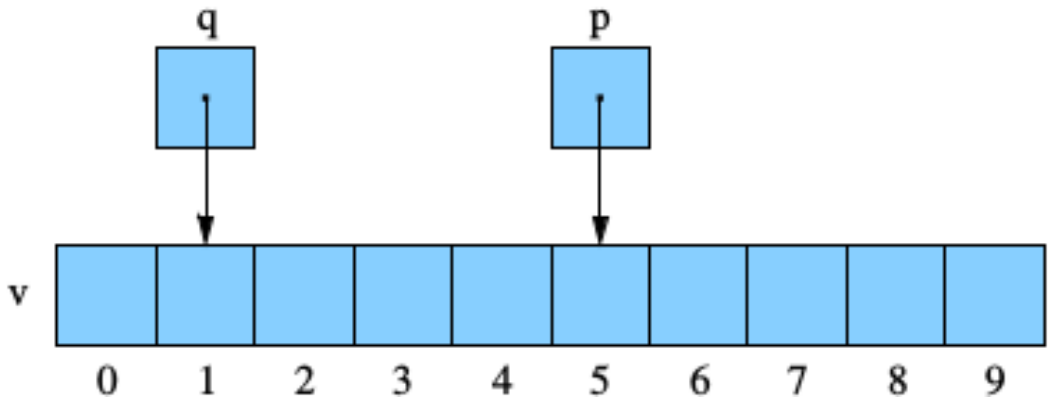
(a) `p = &v[2];` (b) `q = p + 3;` (c) `p = p + 6;`

Aritmética com ponteiros



(a) `p = &v[8];` (b) `q = p - 3;` (c) `p = p - 6;`

Aritmética com ponteiros



`p = &v[5];` e `q = &v[1];`

A expressão $p - q$ tem valor 4 e a expressão $q - p$ tem valor -4

Aritmética com ponteiros

- Podemos comparar variáveis ponteiros entre si usando os operadores relacionais (`<` , `<=` , `>` , `>=` , `==` e `!=`)
- O resultado da comparação depende das posições relativas dos dois elementos do vetor
- Por exemplo, depois das atribuições dadas a seguir:

```
p = &v[5];
q = &v[1];
```

o resultado da comparação `p <= q` é falso e o resultado de `p >= q` é verdadeiro.

Uso de ponteiros para processamento de vetores

- Podemos usar aritmética de ponteiros para visitar os elementos de um vetor
- Fazemos isso através da atribuição de um ponteiro para seu início e do seu incremento em cada passo

```
.
.
.
#define DIM 100

int main(void)
{
   int v[DIM], soma, *p;

   .
   .
   .
   soma = 0;
   for (p = &v[0]; p < &v[DIM]; p++)
      soma = soma + *p;

   .
   .
   .
```

Uso de ponteiros para processamento de vetores

- A condição $p < \&v[DIM]$ na estrutura de repetição `for` necessita de atenção especial
- Apesar de estranho, é possível aplicar o operador de endereço para $v[DIM]$, mesmo sabendo que este elemento não existe no vetor v
- Usar $v[DIM]$ dessa maneira é perfeitamente seguro, já que a sentença `for` não tenta examinar o seu valor
- O corpo da estrutura de repetição será executado com p igual a $\&v[0], \&v[1], \ldots, \&v[DIM-1]$, mas quando p é igual a $\&v[DIM]$ a estrutura de repetição termina.

Uso de ponteiros para processamento de vetores

- Podemos combinar o operador de indireção `*` com operadores de incremento `++` ou decremento `--` em sentenças que processam elementos de um vetor
- Queremos armazenar um valor em um vetor e então avançar para o próximo elemento
  - Usando um índice, podemos fazer diretamente:

```
v[i++] = j;
```

  - Se *p* está apontando para o *i*-ésimo elemento do vetor, a sentença correspondente usando esse ponteiro é:

```
*p++ = j;
```

Uso de ponteiros para processamento de vetores

- Devido à precedência do operador `++` sobre o operador `*`, o compilador enxerga a sentença `*p++ = j;` como

```
*(p++) = j;
```

- O valor da expressão `*p++` é o valor de `*p`, antes do incremento; depois que esse valor é devolvido, a sentença incrementa `p`

Uso de ponteiros para processamento de vetores

- Podemos escrever `(*p)++` para incrementar o valor de `*p`; nesse caso, o valor devolvido pela expressão é também `*p`, antes do incremento; em seguida, a sentença incrementa `*p`
- Podemos escrever `*++p` para incrementar *p* e o valor da expressão é `*p`, depois do incremento
- Podemos escrever `++*p` para incrementar `*p` e o valor da expressão é `*p`, depois do incremento

Uso de ponteiros para processamento de vetores

- O trecho de código que realiza a soma dos elementos do vetor *v* usando aritmética com ponteiros, pode então ser reescrito como a seguir, usando uma combinação dos operadores `*` e `++`

```
.
.
.
soma = 0;
p = &v[0];
while (p < &v[DIM])
   soma = soma + *p++;
.
.
.
```

Uso do identificador de um vetor como ponteiro

- O identificador de um vetor pode ser usado como um ponteiro para o primeiro elemento do vetor
  - Isso simplifica a aritmética com ponteiros e estabelece ganho de versatilidade em ambos, ponteiros e vetores
- Suponha que temos o vetor *v* declarado como abaixo:

```
int v[10];
```

- Usando *v* como um ponteiro para o primeiro elemento do vetor, podemos modificar o conteúdo de `v[0]` da seguinte forma:

```
*v = 7;
```

- Podemos também modificar o conteúdo de `v[1]` através do ponteiro `v + 1`:

```
*(v + 1) = 12;
```

Uso do identificador de um vetor como ponteiro

- Em geral, $v + i$ é o mesmo que `&v[i]`, e `*(v + i)` é equivalente a `v[i]`
- Em outras palavras, índices de vetores podem ser vistos como uma forma de aritmética de ponteiros
- O fato que o identificador de um vetor pode servir como um ponteiro facilita nossa programação de estruturas de repetição que percorrem vetores

Uso do identificador de um vetor como ponteiro

- Considere a estrutura de repetição do exemplo dado anteriormente

```
soma = 0;
for (p = &v[0]; p < &v[DIM]; p++)
   soma = soma + *p;
```

- Para simplificar essa estrutura de repetição, podemos substituir `&v[0]` por *v* e `&v[DIM]` por `v + DIM`

```
soma = 0;
for (p = v; p < v + DIM; p++)
   soma = soma + *p;
```

Uso do identificador de um vetor como ponteiro

- Apesar de podermos usar o identificador de um vetor como um ponteiro, não é possível atribuir-lhe um novo valor
- A tentativa de fazê-lo apontar para qualquer outro lugar é um erro

```
while (*v != 0)
   v++;
```

Uso do identificador de um vetor como ponteiro

```
#include <stdio.h>

#define N 10

int main(void)
{
   int v[N], *p;

   printf("Informe %d números: ", N);
   for (p = v; p < v + N; p++)
      scanf("%d", p);

   printf("Em ordem inversa: ");
   for (p = v + N - 1; p >= v; p--)
      printf("%d ", *p);
   printf("\n");

   return 0;
}
```

Uso do identificador de um vetor como ponteiro

- Outro uso do identificador de um vetor como um ponteiro é quando um vetor é um argumento em uma chamada de função
- O vetor é sempre tratado como um ponteiro

Uso do identificador de um vetor como ponteiro

- Considere a seguinte função que recebe um vetor de n números inteiros e devolve um maior elemento nesse vetor

```
int max(int n, int v[MAX])
{
   int i, maior;
   maior = v[0];
   for (i = 1; i < n; i++)
      if (v[i] > maior)
         maior = v[i];
   return maior;
}
```

- Suponha que chamamos a função **`max`** da seguinte forma:

```
M = max(N, u);
```

- Essa chamada faz com que o endereço do primeiro compartimento do vetor u seja atribuído à v; o vetor u não é de fato copiado

Uso do identificador de um vetor como ponteiro

- Um *parâmetro* que é um vetor pode ser declarado como um ponteiro

```
int max(int n, int *v)
{
   int i, maior;

   maior = v[0];
   for (i = 1; i < n; i++)
      if (v[i] > maior)
         maior = v[i];

   return maior;
}
```

- Neste caso, declarar o parâmetro *v* como sendo um ponteiro é equivalente a declarar *v* como sendo um vetor
- O compilador trata ambas as declarações como idênticas

Uso do identificador de um vetor como ponteiro

- Cuidado! apesar de a declaração de um *parâmetro* como um vetor ser equivalente à declaração do mesmo *parâmetro* como um ponteiro, o mesmo não vale para uma *variável*
- Por exemplo, as declarações das *variáveis*

```
int v[10];
```

e

```
int *v;
```

determinam *variáveis* muito diferentes

- Assim, se logo em seguida fazemos a atribuição

```
*v = 7;
```

armazena o valor 7 onde *v* está apontando. Como não sabemos para onde *v* está apontando, o resultado da execução dessa linha de código é imprevisível

Uso do identificador de um vetor como ponteiro

- Podemos usar uma variável ponteiro, que aponta para uma posição de um vetor, como um vetor

```
.
.
.
#define DIM 100

int main(void)
{
   int v[DIM], soma, *p;

   .
   .
   .
   soma = 0;
   p = v;
   for (i = 0; i < DIM; i++)
      soma = soma + p[i];
```

- O compilador trata a referência `p[i]` como `*(p + i)`

Exercícios

1. Entenda o que o programa abaixo faz, simulando sua execução passo a passo. Depois disso, implemente-o.

```
#include <stdio.h>

int main(void)
{
   int a, b, *pt1, *pt2;

   pt1 = &a;
   pt2 = &b;
   a = 1;
   (*pt1)++;
   b = a + *pt1;
   *pt2 = *pt1 * *pt2;
   printf("a=%d, b=%d, *pt1=%d, *pt2=%d\n", a, b, *pt1, *pt2);

   return 0;
}
```

Exercícios

2 Entenda o que o programa abaixo faz, simulando sua execução passo a passo. Depois disso, implemente-o.

```
#include <stdio.h>

int main(void)
{
   int a, b, c, *ptr;

   a = 3;
   b = 7;
   printf("a=%d, b=%d\n", a, b);
   ptr = &a;
   c = *ptr;
   ptr = &b;
   a = *ptr;
   ptr = &c;
   b = *ptr;
   printf("a=%d, b=%d\n", a, b);

   return 0;
}
```

Exercícios

3 Entenda o que o programa abaixo faz, simulando sua execução passo a passo. Depois disso, implemente-o.

```
#include <stdio.h>

int main(void)
{
   int i, j, *p, *q;

   p = &i;
   q = p;
   *p = 1;
   printf("i=%d, *p=%d, *q=%d\n", i, *p, *q);
   q = &j;
   i = 6;
   *q = *p;
   printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);

   return 0;
}
```

Exercícios

4 (a) Escreva uma função com a seguinte interface:

```
void min_max(int n, int v[MAX], int *max, int *min)
```

que receba um número inteiro n, com $1 \leq n \leq 100$, e um vetor v com $n > 0$ números inteiros e devolva um maior e um menor dos elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um menor elemento desse conjunto. Simule no papel a execução de seu programa antes de implementá-lo.

Exercícios

5 (a) Escreva uma função com a seguinte interface:

```
int *maximo(int n, int v[MAX])
```

que receba um número intero n, com $1 \leq n \leq 100$, e um vetor v de n números inteiros e devolva o endereço do elemento de v onde reside um maior elemento de v.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior elemento desse conjunto. Simule no papel a execução de seu programa antes de implementá-lo.

Exercícios

6 Qual o conteúdo do vetor *v* após a execução do seguinte trecho de código?

```
.
.
.
#define N 10

int main(void)
{
   int v[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int *p = &v[0], *q = &v[N - 1], temp;

   while (p < q) {
      temp = *p;
      *p++ = *q;
      *q-- = temp;
   }

   .
   .
   .
```