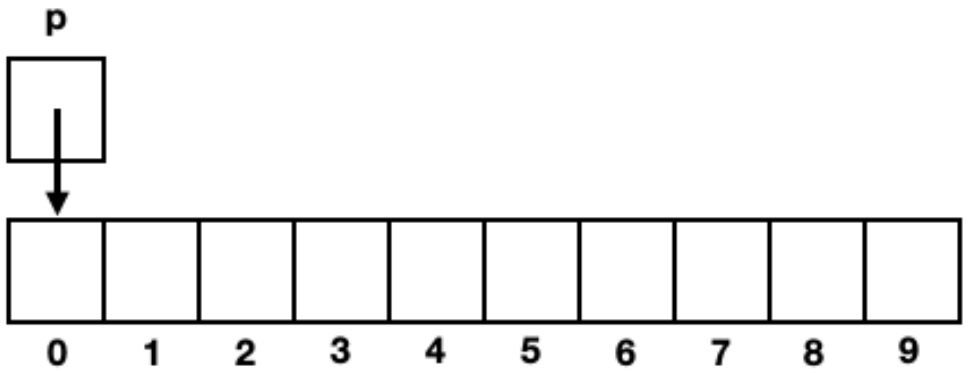
Programação Estruturada

Aula 15 - Ponteiros e Arrays

Yuri Malheiros (yuri@ci.ufpb.br)

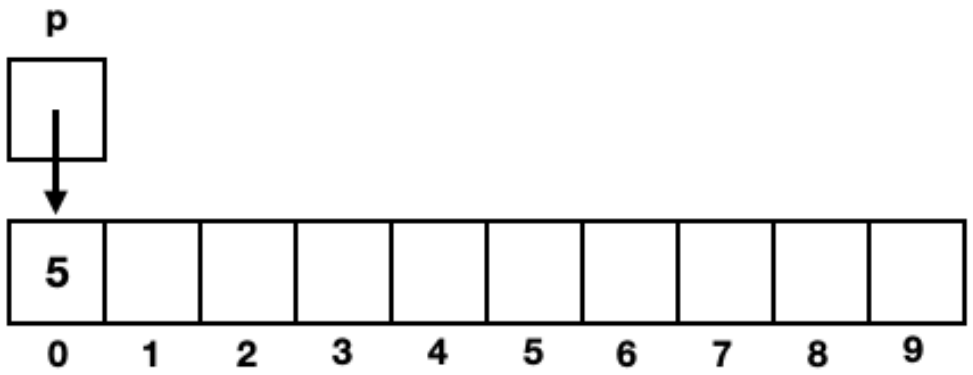
Aritmética de ponteiros

- Ponteiros podem apontar para elementos de um array
- Suponha `int a[10], *p`
- `p = &a[0]`, `p` aponta para `a[0]`



Aritmética de ponteiros

- Agora podemos acessar `a[0]` através de `p`
- `*p = 5` guarda `5` em `a[0]`

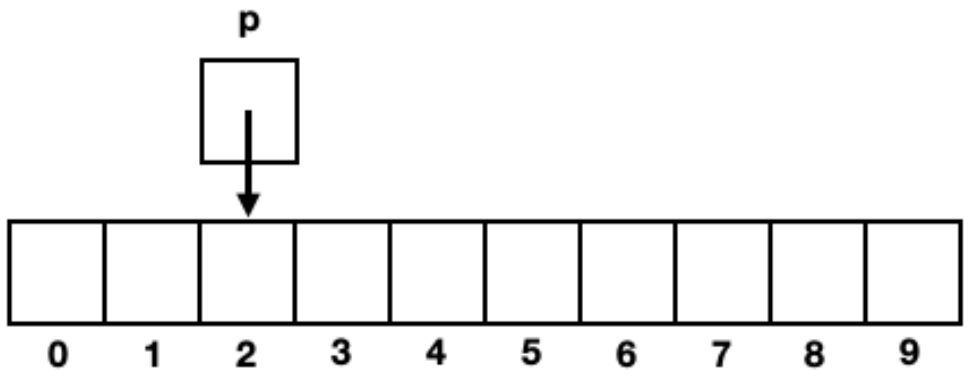


Aritmética de ponteiros

- Utilizando aritmética de ponteiros podemos acessar outros elementos do array `a`
- C suporta três operações aritméticas em ponteiros:
  - Adicionar um inteiro a um ponteiro
  - Subtrair um inteiro de um ponteiro
  - Subtrair um ponteiro de outro ponteiro
- Vamos ver essas três operações
- Suponha: `int a[10], *p, *q, i;`

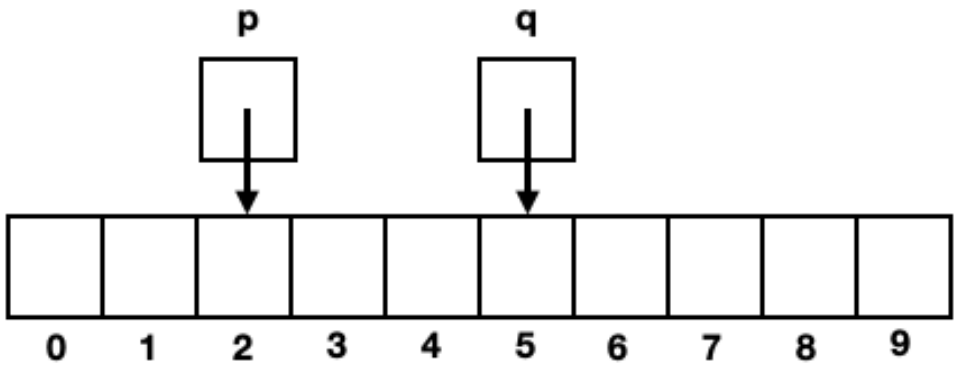
Aritmética de ponteiros

- `p = &a[2]`



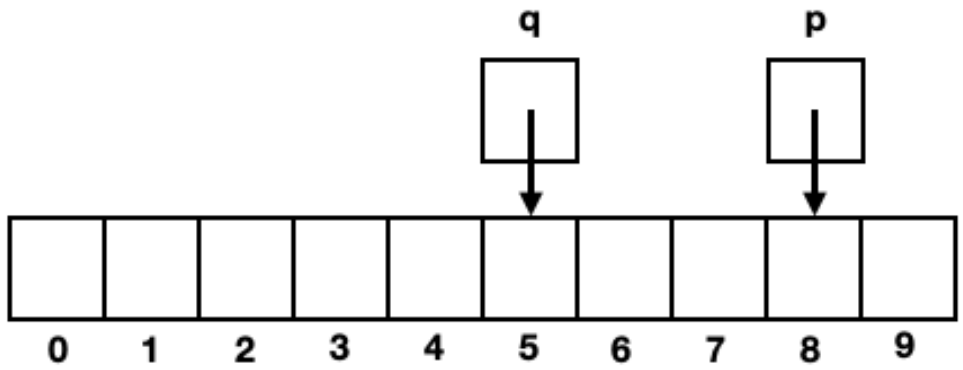
Aritmética de ponteiros

- `q = p+3`



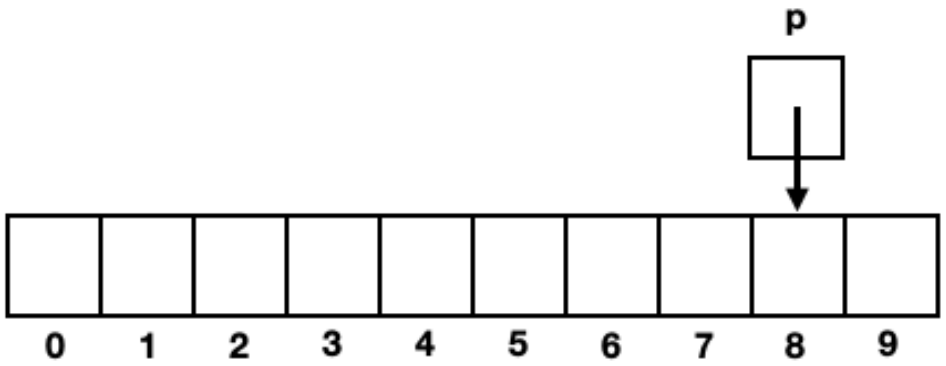
Aritmética de ponteiros

- `p += 6`



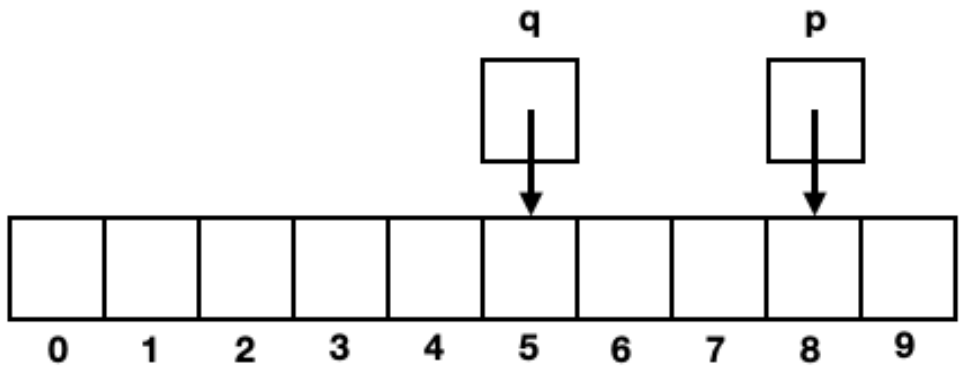
Aritmética de ponteiros

- `p = &a[8]`



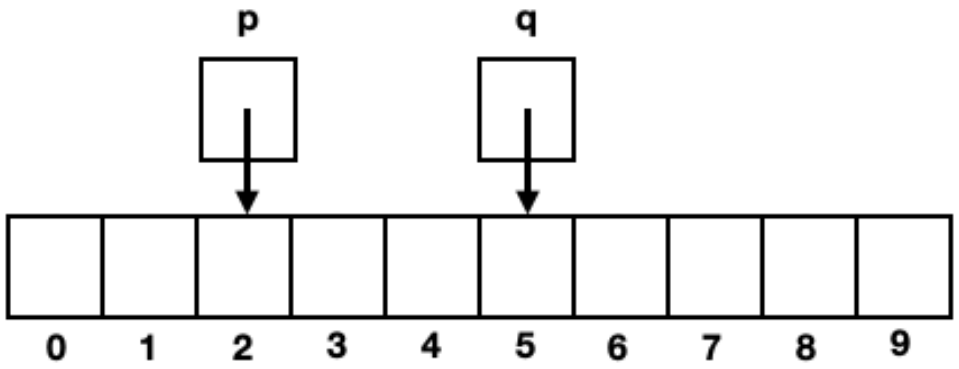
Aritmética de ponteiros

- `q = p-3`



Aritmética de ponteiros

- `p -= 6`

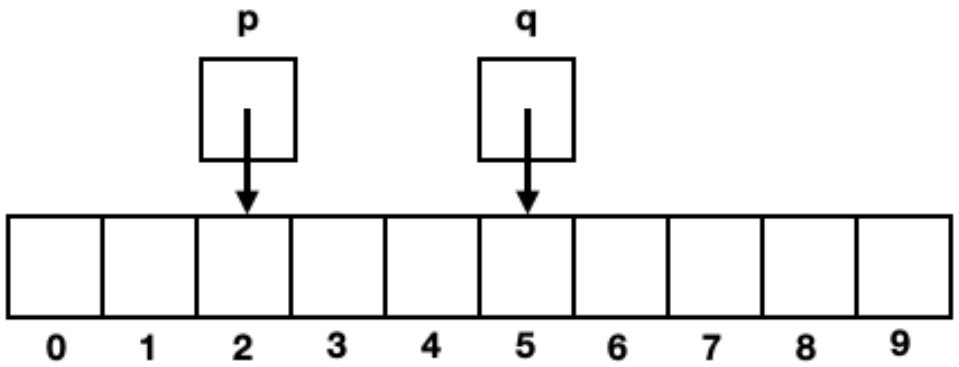


Aritmética de ponteiros

- Subtrair um ponteiro de outro resulta na distância (medida em elementos do array) entre os ponteiros
- Se `p` apontar para `a[i]` e `q` apontar para `a[j]` então `p-q` é igual a `i-j`

Aritmética de ponteiros

- `p = &a[2]`
- `q = &a[5]`



- `q-p` é igual a `3`
- `p-q` é igual a `-3`

Comparando ponteiros

- Podemos comparar ponteiros usando operadores relacionais (`<` , `<=` , `>` , `>=`) e operadores de igualdade (`==` e `!=`)
- Essa comparação faz sentido para ponteiros que apontam para elementos de um mesmo array
- Dados `p = &a[5]` e `q = &a[1]`
- `p <= q` é igual a `0`
- `p >= q` é igual a `1`

Usando ponteiros para processar arrays

- A aritmética de ponteiros nos permite visitar elementos do array incrementando um ponteiro

```
#define N 10

int a[N], sum, *p;

sum = 0;
for (p = &a[0]; p<&a[N]; p++)
  sum += *p;
```

Combinando os operadores `*` e `++`

- Programadores C costumam combinar o operador `*` e o `++` para processar arrays
- Considere o caso de armazenar um valor num elemento do array e incrementar o índice
  - `a[i++] = j`
- Se `p` aponta para um elemento de um array, podemos escrever:
  - `*p++ = j`
- Isto é equivalente a:
  - `*(p++) = j`
- Qual a diferença para `(*p)++`?

Combinando os operadores `*` e `++`

```
for (p = &a[0]; p<&a[N]; p++)
  sum += *p;
```

- É equivalente a:

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

Nome do array como um ponteiro

- O nome do array pode ser usado como um ponteiro para o seu primeiro elemento
- Esta relação simplifica o uso de ponteiros com arrays
- Dado: `int a[10]`
- `*a = 7`
  - 7 é atribuído ao primeiro elemento de `a`
- `*(a+1) = 12`
  - 12 é atribuído ao segundo elemento de `a`

Nome do array como um ponteiro

- No geral, `a+i` é o mesmo que `&a[i]`
- `*(a+i)` é o mesmo que `a[i]`
- Em outras palavras, a indexação de arrays pode ser vista como uma forma de aritmética de ponteiros

Nome do array como um ponteiro

- Utilizando o nome do array como ponteiro, podemos simplificar o nosso código
- Antes:

```
for (p=&a[0]; p<&a[N]; p++)
  sum += *p;
```

- Depois:

```
for (p=a; p<a+N; p++)
  sum += *p;
```

Nome do array como um ponteiro

- O nome do array pode ser utilizado como ponteiro, mas não podemos atribuir um novo valor para esse ponteiro
- Isso é errado:

```
int a[10];
a++;
```

- Entretanto, isso é simples de resolver:

```
int a[10];
int* p;

p = a;
p++;
```

Exemplo

- Vamos escrever um programa que exibe um array de trás para frente

Arrays usados como argumento

- Quando passado para uma função, o nome de um array é tratado como um ponteiro

```c
int encontrar_maior(int a[], int n) {
  int i, max;

  max = a[0];
  for (i=1; i<n; i++) {
    if (a[i] > max)
      max = a[i];
  }

  return max;
}
```

- Ao chamar: `maior = encontrar_maior(b, N);`
- `a` vai apontar para o primeiro elemento de `b`

Arrays usados como argumento

- Um parâmetro array pode ser declarado como um ponteiro
- Por exemplo:

```
int encontrar_maior(int *a, int n) {
  ...
}
```

- Para uma função, se quisermos passar um array, basta passar um ponteiro para o seu primeiro elemento

Arrays usados como argumento

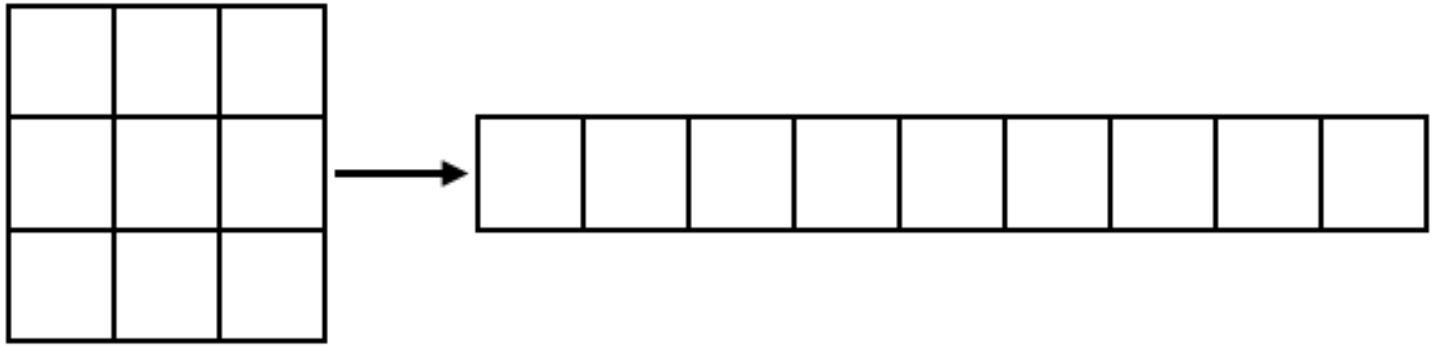
- Podemos passar apenas uma fatia do array para uma função
- `encontrar_maior(&b[5], 10);` vai encontrar o maior elemento começando de `b[5]` até `b[14]`

Ponteiro como nome de array

- Dado: `int a[10], *p = a`
- É válido usar indexação em `p`
- O compilador trata `p[i]` como `*(p+i)`

Ponteiros para arrays multidimensionais

- Ponteiros podem apontar para elementos em arrays multidimensionais
- Vimos anteriormente que C guarda um array bidimensional de forma linear



Ponteiros para arrays multidimensionais

- Se um ponteiro `p` apontar para o primeiro elemento de um array bidimensional, podemos acessar todos os elementos incrementando `p`

```
int a[NUM_LINHAS][NUM_COLS];
int *p;

for (p = &a[0][0]; p <= &a[NUM_LINHAS-1][NUM_COLS-1]; p++)
  *p = 0;
```

Ponteiros para arrays multidimensionais

- Como processar os elementos apenas de uma linha especifica?
- Vamos inicializar o ponteiro para o primeiro elemento de uma linha `i`
  - `p = &a[i][0];` ou `p = a[i];`

```
int a[NUM_LINHAS][NUM_COLS];
int *p, i;

for (p = a[i]; p <= a[i] + NUM_COLS; p++)
  *p = 0;
```

Ponteiros para arrays multidimensionais

- Como `a[i]` aponta para o primeiro elemento de uma linha (que é um array unidimensional)
- Podemos passá-lo para uma função que espera um array unidimensional
    - `encontrar_maior(a[i], NUM_COLS);`

Usando o nome de um array multidimensional como ponteiro

- O nome de qualquer array pode ser usado como ponteiro
- Dado `int a[NUM_LINHAS][NUM_COLS];`
- `a` não é um ponteiro para `a[0][0]`
- `a` é um ponteiro para `a[0]`