MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2014

Roteiro

Ponteiros

- Ponteiros (também chamados de apontadores) são tipos especiais de dados que armazenam endereços de memória.
- Uma variável do tipo ponteiro deve ser declarada da seguinte forma:

  `tipo *nome_variável;`
- A variável ponteiro armazenará um endereço de memória de uma outra variável do tipo especificado.

Exemplos:

```
int *mema;
float *memb;
```

- Neste exemplos temos que:
  - `mema` armazena o endereço de memória de variáveis do tipo `int`.
  - `memb` armazena o endereço de memória de variáveis do tipo `float`.

Operadores de ponteiros

- Existem dois operadores relacionados a ponteiros:
  - O operador & retorna o endereço de memória de uma variável:

```
int *mema;
int a = 90;
mema = &a;
```

  - O operador $*$ retorna o conteúdo do endereço indicado pelo ponteiro:

```
printf("%d", *mema);
```

Operadores de ponteiros

```
#include <stdio.h>

int main() {
  int b, *c;

  b = 10;
  c = &b;
  *c = 11;

  printf("%d\n", b);

  return 0;
}
```

O que será impresso por este programa?
11

Operadores de ponteiros

```
#include <stdio.h>

int main() {
  int num, q = 1, *p;

  num = 100;
  p = &num;
  q = *p;

  printf("%d\n", q);

  return 0;
}
```

O que será impresso por este programa?
100

Operadores de ponteiros

```
#include <stdio.h>

int main() {
  int a = 3, b = 2;
  int *p, *q;

  p = &a;
  q = p;
  *q = *q + 1;
  q = &b;
  b = b + 1;

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

  return 0;
}
```

O que será impresso por este programa?

3, 4

Cuidados com uso de ponteiros

- Não se deve atribuir um valor ao endereço apontado por um ponteiro, sem antes ter certeza de que o endereço é válido:

```
int a, b, *c;

b = 10;
*c = 13;  /* onde o valor 13 sera armazenado? */
```

- O correto seria, por exemplo:

```
int a, b, *c;

b = 10;
c = &a;
*c = 13;
```

Cuidados com uso de ponteiros

Como o operador $*$ de ponteiros é igual ao operador $*$ utilizado na multiplicação, deve-se ter cuidado no uso desses operadores.

```
#include <stdio.h>

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

  b = 10;
  c = &a;
  *c = 11;
  a = b * c; /* erro: operacao invalida */

  printf("%d\n", a);

  return 0;
}
```

Ocorre um erro de compilação na linha indicada acima, pois o $*$ é interpretado como operador de ponteiro sobre a variável c.

Cuidados com uso de ponteiros

Neste caso, o correto seria algo como:

```
#include <stdio.h>

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

  b = 10;
  c = &a;
  *c = 11;
  a = b * (*c);

  printf("%d\n", a);

  return 0;
}
```

Cuidados com uso de ponteiros

Um ponteiro armazena o endereço de um tipo específico.

```
#include <stdio.h>

int main() {
  double b, a;
  int *c;

  b = 10.89;
  c = &b;  /* erro: tipos diferentes */
  a = *c;
  printf("%f\n", a);
  return 0;
}
```

Além do compilador alertar que a atribuição pode causar problemas, um valor diferente do desejado será impresso.

Operações com ponteiros

Podemos comparar ponteiros ou os conteúdos apontados por estes:

```
#include <stdio.h>

int main() {
  double *a, *b, c, d;
  a = &d;
  b = &c;
  scanf("%lf %lf", &c, &d);

  if (b < a)
    printf("Endereco apontado por b eh menor: %p e %p\n", b, a);
  else
    if (a < b)
      printf("Endereco apontado por a eh menor: %p e %p\n", a, b);
    else if (a == b)
      printf("Mesmo endereco\n"); /* impossivel neste exemplo */
  if (*a == *b)
    printf("Mesmo conteudo: %f\n", *a);

  return 0;
}
```

Note que, para imprimir um ponteiro, usamos %p.

Operações com ponteiros

- Quando um ponteiro não está associado a nenhum endereço válido é comum atribuir o valor `NULL` para este (definido na biblioteca `stdlib.h` como zero).
- `NULL` é usado em comparações com ponteiros para saber se um determinado ponteiro possui um endereço válido ou não.

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

int main() {
  double *a = NULL, *b, c = 5;
  a = &c;

  if (a != NULL) {
    b = a;
    printf("Numero: %f\n", *b);
  }

  return 0;
}
```

Passagem de parâmetros

- Passagem de parâmetro é o mecanismo utilizado para fornecer informações para uma função.
- Há dois tipos de passagem de parâmetros:
  - Passagem por valor.
  - Passagem por referência.

Passagem de parâmetros por valor

- Quando passamos parâmetros para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função.
- Este processo é idêntico a uma atribuição e é chamado de passagem por valor.
- Desta forma, alterações nos parâmetros dentro da função não alteram os valores que foram passados.

```
#include <stdio.h>

void nao_troca(int a, int b) {
  int aux = a;
  a = b;
  b = aux;
}

int main() {
  int x = 4, y = 5;
  nao_troca(x, y);
  printf("%d %d\n", x, y);
  return 0;
}
```

Passagem de parâmetros por referência

- Em C, só existe passagem de parâmetros por valor.
- Em algumas linguagens, há construções para se passar *parâmetros por referência*.
  - Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
  - No exemplo anterior, se `x` e `y` fossem passados por referência, seus conteúdos seriam trocados.

Passagem de parâmetros por referência

- Podemos obter algo semelhante em C utilizando ponteiros.
- O artifício corresponde em fornecer como parâmetro para uma função o *endereço* de uma variável e não o seu valor.
- Desta forma, podemos alterar o conteúdo da variável como se fizéssemos passagem por referência.

```
#include <stdio.h>

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

int main() {
  int x = 4, y = 5;
  troca(&x, &y);
  printf("%d %d\n", x, y);
  return 0;
}
```

Passagem de parâmetros por referência

- O uso de ponteiros para passar parâmetros que devem ser alterados dentro de uma função é útil em certas situações como, por exemplo, funções que precisam retornar mais do que um valor.
- Suponha que queremos criar uma função que recebe um vetor como parâmetro e precisa retornar o menor e o maior elemento do vetor.
  - Uma função só pode retornar um único valor.
  - Podemos passar ponteiros para variáveis que armazenarão o menor e o maior elemento.

Passagem de parâmetros por referência

```
#include <stdio.h>
void min_and_max(int vet[], int tam, int *min, int *max);

int main() {
  int v[] = {10, 80, 5, -10, 45, -20, 100, 200, 10};
  int min, max;
  min_and_max(v, 9, &min, &max);
  printf("Menor valor: %d\n", min);
  printf("Maior valor: %d\n", max);
  return 0;
}

void min_and_max(int vet[], int tam, int *min, int *max) {
  int i;
  *min = vet[0];
  *max = vet[0];
  for (i = 1; i < tam; i++) {
    if (vet[i] < *min)
      *min = vet[i];
    if (vet[i] > *max)
      *max = vet[i];
  }
}
```

Aritmética de ponteiros

- Os operadores + e - podem ser utilizados com ponteiros.
- Seja `p` um ponteiro para um inteiro (num computador de 32 bits) com um valor atual de 3000.
- A expressão:

  `p++;`

  faz com que o conteúdo de `p` seja alterado para 3004 (e não 3001).
- A cada incremento de `p`, ele apontará para o próximo endereço de um tipo inteiro, cujo tamanho é de 4 bytes para o computador neste exemplo.

Aritmética de ponteiros

- O mesmo é válido para decrementos.
- Por exemplo:

  `p--;`

  fará com que `p` assuma o valor 2996 (considerando que o endereço anterior era 3000 e o tamanho de um inteiro é de 4 bytes).

Ponteiros e vetores

- Quando declaramos uma variável do tipo vetor, aloca-se uma quantidade de memória contígua cujo tamanho é especificado na declaração (e também depende do tipo do vetor).
  - `int v[5]; /* aloca 5 * 4 = 20 bytes de memoria para o vetor v, supondo que cada inteiro ocupe 4 bytes */`
- Uma variável vetor, assim como um ponteiro, armazena um endereço de memória: o endereço do início do vetor.
  - `int v[5]; /* variavel v contem o endereco de memoria do inicio do vetor */`
- Por este motivo, quando passamos um vetor como parâmetro para uma função, seu conteúdo pode ser alterado dentro da função, pois estamos passando, na realidade, o endereço do início do espaço alocado para o vetor.

Ponteiros e vetores

```
#include <stdio.h>

void zeraVetor(int vet[], int tam) {
  int i;
  for (i = 0; i < tam; i++)
    vet[i] = 0;
}

int main() {
  int vetor[] = {1, 2, 3, 4, 5}, i;

  zeraVetor(vetor, 5);

  for (i = 0; i < 5; i++)
    printf("%d\n", vetor[i]);

  return 0;
}
```

Ponteiros e vetores

```
#include <stdio.h>

void zeraVetor(int *vet, int tam) {
  int i;
  for (i = 0; i < tam; i++)
    vet[i] = 0;
}

int main() {
  int vetor[] = {1, 2, 3, 4, 5}, i;

  zeraVetor(vetor, 5);

  for (i = 0; i < 5; i++)
    printf("%d\n", vetor[i]);

  return 0;
}
```

Ponteiros e vetores

- De fato, como uma variável vetor possui um endereço, podemos atribuí-la a uma variável ponteiro:

```
int a[] = {1, 2, 3, 4, 5};
int *p;
p = a;
```

- E podemos então usar `p` como se fosse um vetor:

```
for (i = 0; i < 5; i++)
  p[i] = i * i;
```

Ponteiros e vetores

- Uma variável vetor, diferentemente de um ponteiro, possui um endereço fixo.
- Não podemos atribuir um outro endereço a uma variável vetor.

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5}, b[5], i;

  b = a; /* erro: atribuicao invalida */

  for (i = 0; i < 5; i++)
    printf("%d ", b[i]);
  printf("\n");

  return 0;
}
```

Ponteiros e vetores

- Entretanto, se b for declarado como ponteiro, não há problemas:

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5}, *b, i;

  b = a;

  for (i = 0; i < 5; i++)
    printf("%d ", b[i]);
  printf("\n");

  return 0;
}
```

Ponteiros e vetores

- Aritmética de ponteiros pode ser utilizada com vetores. Exemplo:

```
char str[80], *p, c;
p = str;
```

- O ponteiro `p` foi definido como o endereço do primeiro elemento do vetor (string) `str`.
- Para fazer acesso ao quinto elemento de `str`, pode-se escrever:

```
c = str[4];
```

ou

```
c = *(p + 4);
```

Ponteiros e vetores

O resultado deste programa...

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5};
  int *b, i;

  b = a;

  for (i = 0; i < 5; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

Ponteiros e vetores

... é igual a este...

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5};
  int *b, i;

  b = a;

  for (i = 0; i < 5; i++)
    printf("%d ", *(a + i));
  printf("\n");

  return 0;
}
```

Ponteiros e vetores

... e a este...

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5};
  int *b, i;

  b = a;

  for (i = 0; i < 5; i++)
    printf("%d ", *(b + i));
  printf("\n");

  return 0;
}
```

Ponteiros e vetores

... ou mesmo este.

```
#include <stdio.h>

int main() {
  int a[] = {1, 2, 3, 4, 5};
  int *b, i;

  b = a;

  for (i = 0; i < 5; i++)
    printf("%d ", b[i]);
  printf("\n");

  return 0;
}
```

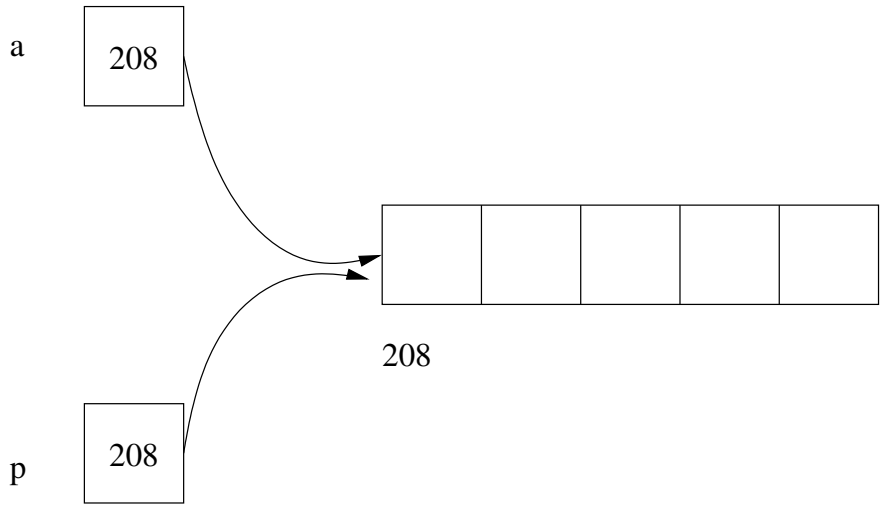
Alocação dinâmica de memória

- Como vimos, uma variável vetor possui um endereço, que podemos atribuí-lo para uma variável ponteiro:

```
int a[] = {1, 2, 3, 4, 5};
int *p;
p = a;
```

- E podemos então usar p como se fosse um vetor:

```
for (i = 0; i < 5; i++)
  p[i] = i * i;
```



Alocação dinâmica de memória

- Em aulas anteriores, ao trabalhar com matrizes, por exemplo, assumíamos que estas tinham dimensões máximas conhecidas:

```
#define MAX 100
...
int m[MAX][MAX];
```

- O que fazer se o usuário precisar trabalhar com matrizes maiores?
- Será que é possível alocar apenas a quantidade de memória necessária para o programa, evitando assim o desperdício de recursos computacionais?

Alocação dinâmica de memória

A biblioteca `stdlib.h` possui funções que permitem manipular memória dinamicamente.

- `malloc`: esta função aloca uma região de memória contígua (número de bytes recebido como parâmetro), retornando um ponteiro para a primeira posição da memória alocada.
  - Exemplo: alocando memória para 100 números inteiros.

```
int *p;
p = malloc(100 * sizeof(int));
```

- `free`: esta função recebe como parâmetro um ponteiro e libera a memória previamente alocada e apontada pelo ponteiro.
  - Exemplo: liberando memória previamente alocada.

```
free(p);
```

- Importante: toda memória alocada dinamicamente durante a execução de um programa (com `malloc`) deve ser desalocada (com o `free`) quando não for mais necessária.

Exemplo - produto interno de vetores

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

int main() {
  double *v1, *v2, produto;
  int i, n;

  printf("Qual a dimensao dos vetores?\n");
  scanf("%d", &n);

  v1 = malloc(n * sizeof(double));
  v2 = malloc(n * sizeof(double));

  printf("Entre com os valores do primeiro vetor: ");
  for (i = 0; i < n; i++)
    scanf("%lf", &v1[i]);

  ...
```

Exemplo - produto interno de vetores

```
  ...

  printf("Entre com os valores do segundo vetor: ");
  for (i = 0; i < n; i++)
    scanf("%lf", &v2[i]);

  produto = 0;
  for (i = 0; i < n; i++)
    produto = produto + (v1[i] * v2[i]);

  printf("Produto interno dos dois vetores: %f\n", produto);

  free(v1);
  free(v2);

  return 0;
}
```

Ponteiros e alocação dinâmica

- Podemos fazer ponteiros distintos apontarem para uma mesma região de memória.
- Neste caso, precisamos tomar cuidado para não acessar uma região de memória (através de um ponteiro) que foi previamente desalocada. Exemplo:

```
double *v1, *v2;

v1 = malloc(100 * sizeof(double));
v2 = v1;
free(v1);

for (i = 0; i < n; i++)
  v2[i] = i;
```

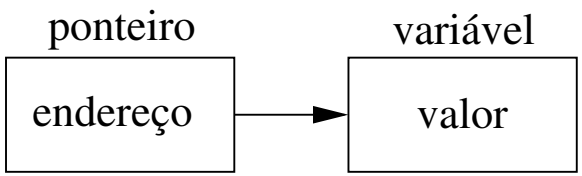
- O código acima está errado e irá causar um erro de execução já que v2 está acessando posições de memória que não estão mais reservadas para o programa.

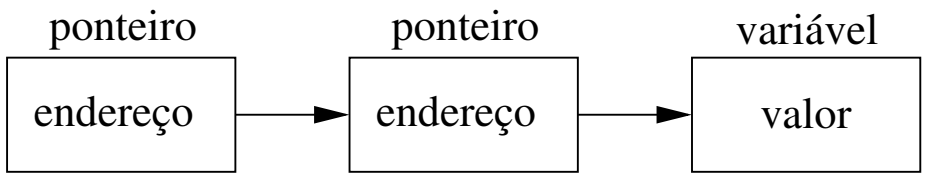
Alocação dinâmica de matrizes

- Em aplicações científicas e de engenharias, é muito comum a realização de diversas operações sobre matrizes.
- Como vimos, em situações reais o ideal é alocar memória suficiente para conter os dados a serem tratados, sem usar nem mais e nem menos memória do que o necessário.
- Como alocar vetores multidimensionais dinamicamente?

Ponteiros de ponteiros

- Uma variável ponteiro é alocada na memória do computador como qualquer outra variável.
- Portanto, podemos criar um ponteiro que contém o endereço de memória de um outro ponteiro.
- O ponteiro de um ponteiro é uma forma de endereçamento encadeado.
- Na figura à esquerda, o valor do ponteiro é o endereço da variável que contém o valor desejado.
- Na figura à direita, o primeiro ponteiro contém o endereço de um segundo ponteiro, que aponta para a variável que tem o valor desejado.





Ponteiros de ponteiros

O que o programa abaixo irá imprimir quando executado?

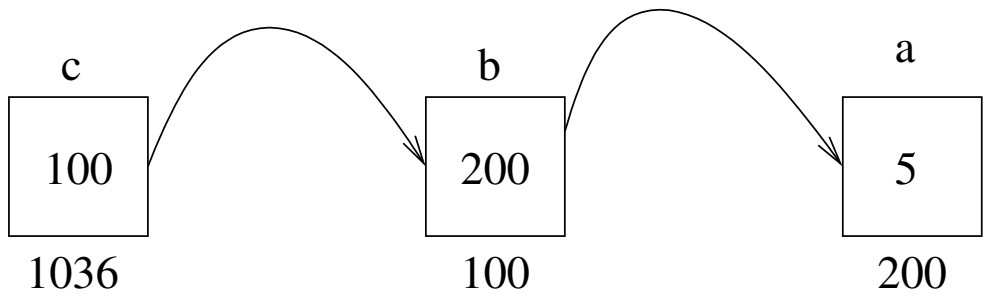
```
#include <stdio.h>

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

  a = 5;
  b = &a;
  c = &b;
  printf("%d\n", *(*c));

  return 0;
}
```

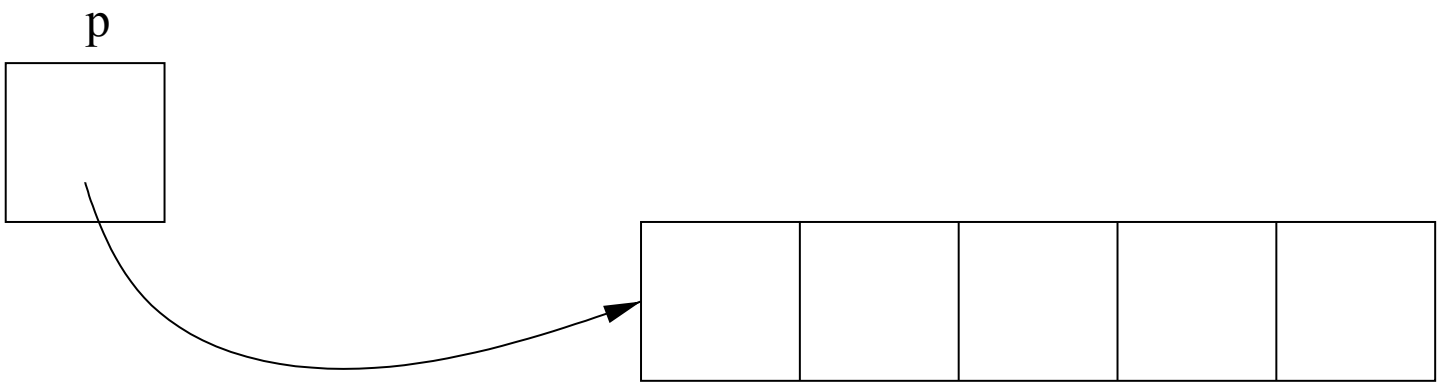
O programa imprimirá o valor de a, ou seja, 5.



Ponteiros para ponteiros

Pela nossa discussão anterior sobre ponteiros, sabemos que um ponteiro pode ser usado para referenciar um vetor alocado dinamicamente.

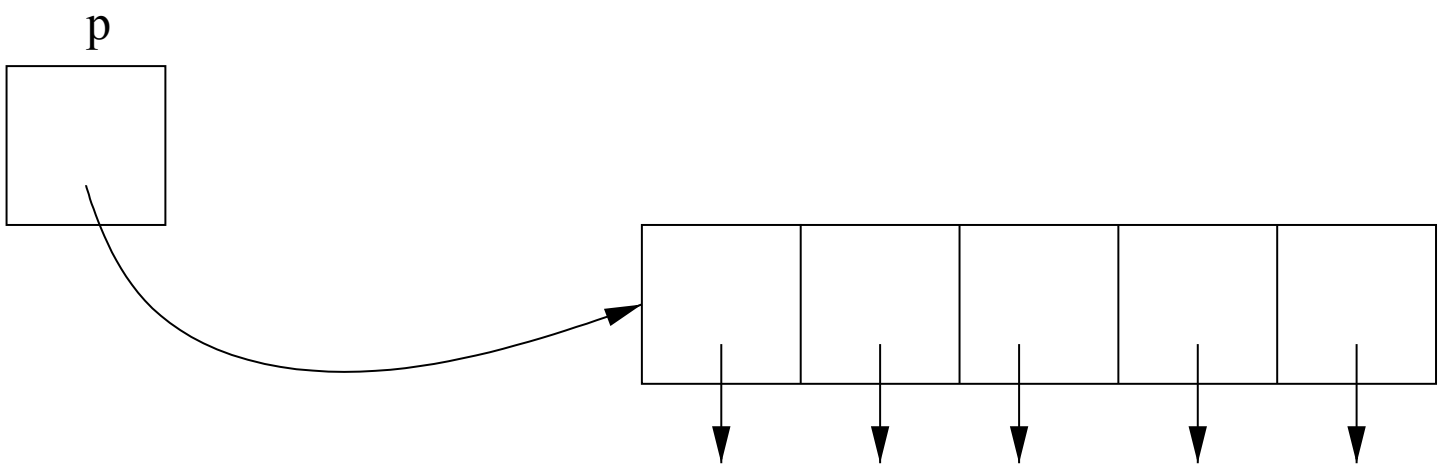
```
int *p;
p = malloc(5 * sizeof(int));
```



Ponteiros para ponteiros

Da mesma forma, podemos usar um ponteiro de ponteiro para referenciar um vetor de ponteiros alocado dinamicamente.

```
int **p;
p = malloc(5 * sizeof(int *));
```



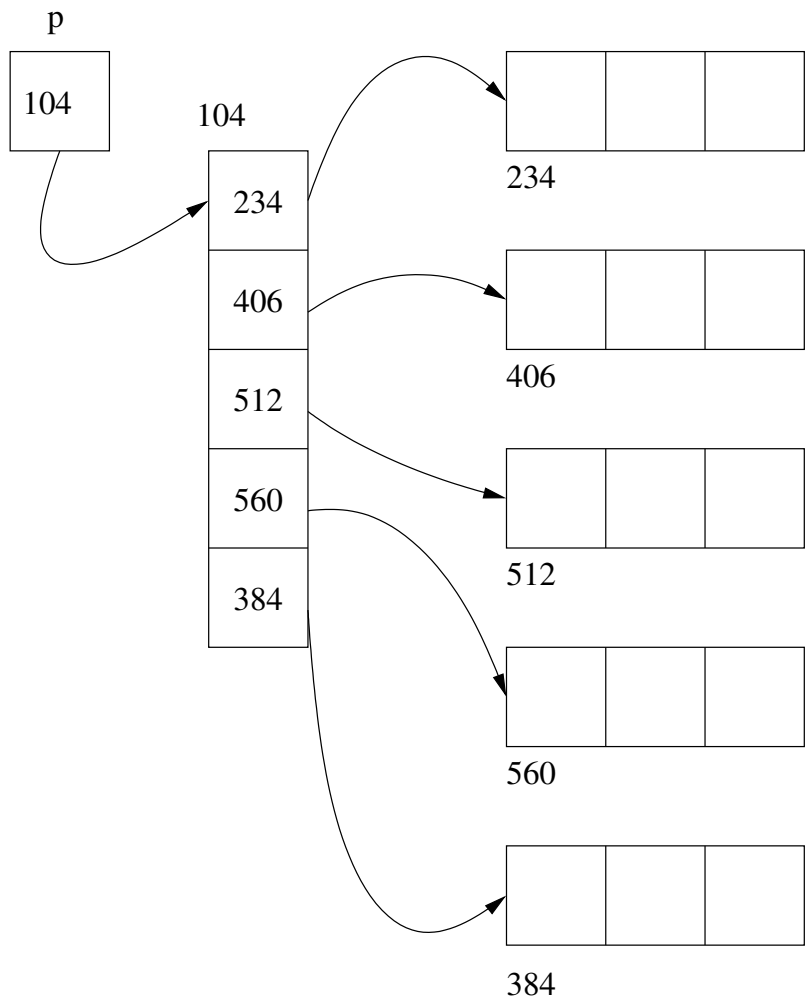
Note que cada posição do vetor acima é do tipo `int *`, ou seja, um ponteiro para inteiro.

Ponteiros para ponteiros

Como cada posição do vetor é um ponteiro para inteiro, podemos associar cada posição dinamicamente com um vetor de inteiros.

```
int **p, i;
p = malloc(5 * sizeof(int *));

for (i = 0; i < 5; i++)
  p[i] = malloc(3 * sizeof(int));
```



Alocação dinâmica de matrizes

- Podemos alocar matrizes dinamicamente da seguite forma:
  - Crie um ponteiro para ponteiro.
  - Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor será o número de linhas da matriz.
  - Cada posição do vetor será associado com um outro vetor do tipo a ser armazenado. Cada um destes vetores será uma linha da matriz (portanto, possuirá tamanho igual ao número de colunas).
- Lembre que devemos desalocar toda a memória alocada por este processo assim que ela não for mais necessária.

Exemplo - alocação dinâmica de matrizes

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

int main() {
  int **matriz, linhas, colunas, i, j;

  printf("Entre com o numero de linhas: ");
  scanf("%d", &linhas);

  printf("Entre com o numero de colunas: ");
  scanf("%d", &colunas);

  printf("Alocando a matriz...\n");
  matriz = malloc(linhas * sizeof(int *));
  for (i = 0; i < linhas; i++)
    matriz[i] = malloc(colunas * sizeof(int));
  ...
```

Exemplo - alocação dinâmica de matrizes

```
  ...
  printf("Obtendo os valores da matriz...\n");
  for (i = 0; i < linhas; i++)
    for (j = 0; j < colunas; j++)
      scanf("%d", &matriz[i][j]);

  printf("Imprimindo a matriz...\n");
  for (i = 0; i < linhas; i++) {
    for (j = 0; j < colunas; j++)
      printf("%d ", matriz[i][j]);
    printf("\n");
  }

  printf("Desalocando a matriz...\n");
  for (i = 0; i < linhas; i++)
    free(matriz[i]);
  free(matriz);

  return 0;
}
```

Exemplo - alocação dinâmica de matrizes usando funções

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

int ** aloca_matriz(int linhas, int colunas) {
  int i, **matriz;

  matriz = malloc(linhas * sizeof(int *));

  for (i = 0; i < linhas; i++)
    matriz[i] = malloc(colunas * sizeof(int));

  return matriz;
}

void desaloca_matriz(int **matriz, int linhas) {
  int i;

  for (i = 0; i < linhas; i++)
    free(matriz[i]);

  free(matriz);
}
```

Exemplo - alocação dinâmica de matrizes usando funções

```
void obtem_matriz(int **matriz, int linhas, int colunas) {
  int i, j;

  for (i = 0; i < linhas; i++)
    for (j = 0; j < colunas; j++)
      scanf("%d", &matriz[i][j]);
}

void imprime_matriz(int **matriz, int linhas, int colunas) {
  int i, j;

  for (i = 0; i < linhas; i++) {
    for (j = 0; j < colunas; j++)
      printf("%d ", matriz[i][j]);
    printf("\n");
  }
}
```

Exemplo - alocação dinâmica de matrizes usando funções

```
int main() {
  int **matriz, linhas, colunas;

  printf("Entre com o numero de linhas: ");
  scanf("%d", &linhas);

  printf("Entre com o numero de colunas: ");
  scanf("%d", &colunas);

  printf("Alocando a matriz...\n");
  matriz = aloca_matriz(linhas, colunas);

  printf("Obtendo os valores da matriz...\n");
  obtem_matriz(matriz, linhas, colunas);

  printf("Imprimindo a matriz...\n");
  imprime_matriz(matriz, linhas, colunas);

  printf("Desalocando a matriz...\n");
  desaloca_matriz(matriz, linhas);

  return 0;
}
```

Exercícios

- Escreva uma função `length(s)` que recebe como parâmetro uma string `s` e retorna seu tamanho (equivalente a função `strlen(s)` da biblioteca `string.h`).
- Escreva uma função `copy(s,t)` que recebe como parâmetro duas strings e copia o conteúdo da string `t` na string `s` (equivalente a função `strcpy(s,t)` da biblioteca `string.h`).
- Escreva uma função `compare(s,t)` que recebe como parâmetro duas strings e compara `s` e `t`, retornando um valor negativo, zero ou positivo se `s` é lexicograficamente menor, igual ou maior que `t`, respectivamente (equivalente a função `strcmp(s,t)` da biblioteca `string.h`).
- Escreva uma função `concatenate(s,t)` que recebe como parâmetro duas strings e concatena `t` em `s` (equivalente a função `strcat(s,t)` da biblioteca `string.h`).

Exercícios

- Escreva funções que, dados dois vetores A e B, representando conjuntos com n e m números inteiros respectivamente, calcule:
  - $C = A \cup B$
  - $C = A \cap B$
  - $C = A - B$
  - $C = A \triangle B = (A - B) \cup (B - A)$
- Escreva um programa que, dadas duas matrizes A e B de números inteiros, de dimensões $p \times q$ e $q \times r$ respectivamente, calcule a matriz produto $C = A \times B$, de dimensões $p \times r$. Seu programa deve alocar as 3 matrizes dinamicamente.

Length

```
/* Versao com vetores */
int length(char s[]) {
  int i = 0;

  while (s[i])
    i++;

  return i;
}
```

Length

```
/* Versao com ponteiros */
int length(char *s) {
  int i = 0;

  while (*s) {
    s++;
    i++;
  }

  return i;
}
```

Copy

```
/* Versao com vetores */
void copy(char s[], char t[]) {
  int i = 0;

  do {
    s[i] = t[i];
  } while (t[i++]);
}
```

Copy

```
/* Versao com ponteiros */
void copy(char *s, char *t) {
  while (*t) {
    *s = *t;
    s++;
    t++;
  }

  *s = '\0';
}
```

Compare

```
/* Versao com vetores */
int compare(char s[], char t[]) {
  int i = 0;

  while (s[i] == t[i])
    if (s[i++] == '\0')
      return 0;

  return (s[i] - t[i]);
}
```

Compare

```
/* Versao com ponteiros */
int compare(char *s, char *t) {
  while (*s == *t) {
    if (*s == '\0')
      return 0;
    s++;
    t++;
  }

  return (*s - *t);
}
```

Concatenate

```
/* Versao com vetores */
void concatenate(char s[], char t[]) {
  int i = 0, j = 0;

  while (s[i])
    i++;

  while (t[j])
    s[i++] = t[j++];

  s[i] = '\0';
}
```

Concatenate

```
/* Versao com ponteiros */
void concatenate(char *s, char *t) {

  while (*s)
    s++;

  while (*t)
    *(s++) = *(t++);

  *s = '\0';
}
```