

Programação Estruturada

Ponteiros — Parte 2

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



Ponteiros

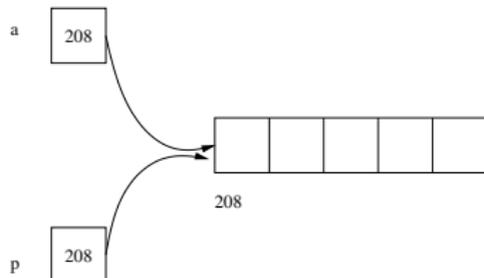
Ponteiros

Lembre-se que uma variável vetor possui um endereço, e que podemos atribuí-la para uma variável ponteiro:

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

E podemos então usar p como se fosse um vetor:

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



- Em aulas anteriores, ao trabalhar com vetores e matrizes, assumíamos que estes tinham dimensões máximas.

```
1  #define MAX 100
2  .
3  .
4  .
5  int vet [MAX] ;
6  int m [MAX] [MAX] ;
```

- O que acontece se o usuário precisar trabalhar com vetores ou matrizes maiores?
- Temos que mudar o valor de MAX e recompilar o programa?

- Alocação dinâmica refere-se à possibilidade de alocar mais memória durante a execução de um programa conforme haja necessidade.
- Pode-se alocar dinamicamente uma quantidade de memória contígua e associá-la com um ponteiro, por exemplo.
- Este ponteiro então será usado como um vetor.
- Desta forma, podemos criar programas sem saber a priori o número de dados a ser armazenado.

Ponteiros e alocação dinâmica

Na biblioteca `stdlib.h` do C existem duas funções para se fazer alocação dinâmica de memória: `malloc` e `calloc`.

Função **malloc**: O seu único parâmetro é o número de **bytes** que deve ser alocado.

Ela devolve o **endereço de memória** do início da região que foi alocada ou **NULL** caso aconteça algum erro.

Exemplo de alocação dinâmica de um espaço para armazenar 100 inteiros:

```
1 int *p, i;
2 p = malloc(100 * sizeof(int));
3 for (i = 0; i < 100; i++)
4     p[i] = 2*i;
```

Ponteiros e alocação dinâmica

Função **calloc**: Seus parâmetros são o número de “blocos de memória” a serem alocados e o tamanho **em bytes** de cada bloco.

Ela devolve o **endereço de memória** do início da região que foi alocada ou **NULL** caso aconteça algum erro.

Exemplo de alocação dinâmica de espaço para armazenar 100 inteiros:

```
1 int *p, i;
2 p = calloc(100, sizeof(int));
3 for (i = 0; i < 100; i++)
4     p[i] = 2*i;
```

Diferença entre malloc e calloc

A função **calloc** “zera” todos os bits da memória alocada enquanto que a **malloc** não.

Logo, se não for necessária uma inicialização (com zeros) da memória alocada, o **malloc** é preferível por ser um pouco mais rápido.

Ponteiros e alocação dinâmica

Juntamente com estas funções, está definida a função `free` na biblioteca `stdlib.h`.

Ela recebe como parâmetro um ponteiro, e libera a memória previamente alocada e apontada pelo ponteiro.

Exemplo:

```
1 int *p;  
2 p = calloc(100, sizeof(int));  
3 ....  
4 free(p);
```

REGRA para uso correto de alocação dinâmica: toda memória alocada durante a execução de um programa e que não for mais utilizada **deve ser desalocada com o `free`!**

Exemplos de alocação dinâmica de vetores

Exemplo 1: produto escalar de 2 vetores

Problema

Calcular o produto escalar de 2 vetores.

Exemplo 1

O programa lê inicialmente a dimensão dos vetores e em seguida faz a alocação dos mesmos.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      double *v1, *v2, prodEsc;
5      int n, i;
6
7      scanf("%d", &n);
8      v1 = malloc(n*sizeof(double));
9      v2 = malloc(n*sizeof(double));
10     ...
11 }
```

Exemplo 1

Em seguida o programa faz a leitura dos dados dos dois vetores.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      ...
6      for (i = 0; i < n; i++)
7          scanf("%lf", &v1[i]);
8      for (i = 0; i < n; i++)
9          scanf("%lf", &v2[i]);
10     ...
11 }
```

Exemplo 1

Finalmente, o programa calcula o produto e imprime o resultado. Note que, no final, os dois vetores têm suas memórias liberadas.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      ...
5      prodEsc = 0;
6      for (i = 0; i < n; i++)
7          prodEsc = prodEsc + (v1[i] * v2[i]);
8
9      printf("Resposta: %.2lf\n", prodEsc);
10
11     free(v1);
12     free(v2);
13     return 0;
14 }
```

Exemplo 1

Solução completa:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      double *v1, *v2, prodEsc;
5      int n, i;
6
7      scanf("%d", &n);
8      v1 = malloc(n * sizeof(double));
9      v2 = malloc(n * sizeof(double));
10
11     for (i = 0; i < n; i++)
12         scanf("%lf", &v1[i]);
13     for (i = 0; i < n; i++)
14         scanf("%lf", &v2[i]);
15
16     prodEsc = 0;
17     for (i = 0; i < n; i++)
18         prodEsc = prodEsc + (v1[i] * v2[i]);
19
20     printf("Resposta: %.2lf\n", prodEsc);
21
22     free(v1);
23     free(v2);
24     return 0;
25 }
```

Exemplo 1

Problema

Criar uma função que recebe duas strings de tamanhos quaisquer e que devolve a concatenação delas.

Lembre-se que uma função não pode devolver um vetor (uma string é um vetor de caracteres), mas ela pode devolver um ponteiro.

Assim, o protótipo da função será:

```
1 char *concatena(char *s1, char *s2);
```

Exemplo 2

Primeiramente devemos alocar a string resposta sres com tamanho suficiente para armazenar a concatenação de s1 com s2.

```
1 char *concatena(char *s1, char *s2) {
2     char *sres = NULL;
3     int t1, t2, i;
4
5     t1 = strlen(s1);
6     t2 = strlen(s2);
7
8     sres = malloc((t1+t2+1) * sizeof(char));
9     ...
10 }
```

Exemplo 2

Depois fazemos a cópia de s1 e s2 para sres e devolvemos o valor do ponteiro sres.

```
1 char *concatena(char *s1, char *s2) {
2     char *sres = NULL;
3     int t1, t2, i;
4     t1 = strlen(s1);
5     t2 = strlen(s2);
6     sres = malloc((t1+t2+1) * sizeof(char));
7
8     for (i = 0; i < t1; i++)
9         sres[i] = s1[i];
10    for (i = 0; i < t2; i++)
11        sres[i + t1] = s2[i];
12
13    sres[t1 + t2] = '\0';
14
15    return sres;
16 }
```

Exemplo 2

Considere esta versão onde fazemos a liberação de memória alocada. Ela está correta?

```
1 char *concatena(char *s1, char *s2) {
2     char *sres = NULL;
3     int t1, t2, i;
4     t1 = strlen(s1);
5     t2 = strlen(s2);
6     sres = malloc((t1+t2+1) * sizeof(char));
7
8     for (i = 0; i < t1; i++)
9         sres[i] = s1[i];
10    for (i = 0; i < t2; i++)
11        sres[i + t1] = s2[i];
12    sres[t1+t2]='\0';
13
14    free(sres); /* Libera memória */
15    return sres;
16 }
```

Exemplo 2

Exemplo de implementação e uso correto da função.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char *concatena(char *s1, char *s2); /* já implementamos */
6
7  int main() {
8      char s1[100], s2[100], *s3;
9
10     fgets(s1, 100, stdin);
11     if (s1[strlen(s1)-1] == '\n')
12         s1[strlen(s1)-1] = '\0'; /* Remove '\n' */
13
14     fgets(s2, 100, stdin);
15     if (s2[strlen(s2)-1] == '\n')
16         s2[strlen(s2)-1] = '\0';
17
18     s3 = concatena(s1, s2);
19     printf("%s\n", s3);
20
21     free(s3); /* aqui podemos liberar a memória */
22     return 0;
23 }
```

Erros comuns ao usar alocação dinâmica

Erros comuns ao usar alocação dinâmica

- Você pode fazer ponteiros distintos apontarem para uma mesma região de memória.
 - Mas tome cuidado para não utilizar um ponteiro se a sua região de memória foi desalocada!

```
1 double *v1, *v2;
2
3 v1 = malloc(100 * sizeof(double));
4 v2 = v1;
5 free(v1);
6
7 for (i = 0; i < n; i++)
8     v2[i] = i*i;
```

- O código acima está errado e pode causar erros durante a execução, já que v2 está acessando posições de memória que foram liberadas!

Erros comuns ao usar alocação dinâmica

O programa abaixo imprime resultados diferentes dependendo se comentamos ou não o comando `free(v1)`. Por quê?

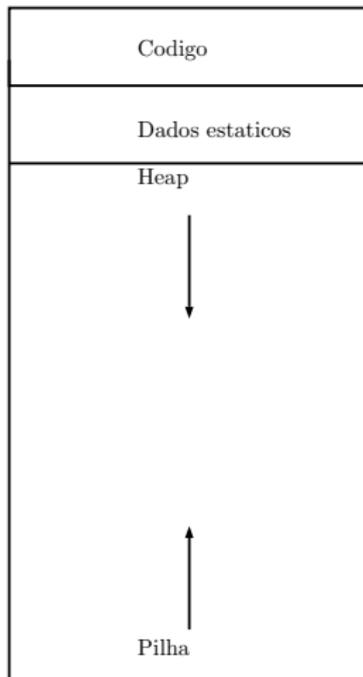
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      double *v1, *v2, *v3;
5      int i;
6      v1 = malloc(100 * sizeof(double));
7      v2 = v1;
8
9      for (i = 0; i < 100; i++)
10         v2[i] = i;
11     free(v1); /* Comente e descomente este comando */
12
13     v3 = calloc(100, sizeof(double));
14     for (i = 0; i < 100; i++)
15         printf("%.2lf\n", v2[i]);
16     free(v3);
17     return 0;
18 }
```

Organização da memória do computador

A memória do computador na execução de um programa é organizada em quatro segmentos:

- **Código executável:** Contém o código binário do programa.
- **Dados estáticos:** Contém variáveis globais e estáticas que existem durante toda a execução do programa.
- **Pilha:** Contém as variáveis locais que são criadas na execução de uma função e depois são removidas da pilha ao término da função.
- **Heap:** Contém as variáveis criadas por alocação dinâmica.

Organização da memória do computador



Organização da memória do computador

Em C99 podemos declarar vetores de tamanho variável em tempo de execução usando o valor de uma variável.

No exemplo abaixo declaramos o vetor `v` com tamanho igual ao valor da variável `n`, que foi lida do teclado.

```
1  #include <stdio.h>
2  int main() {
3      long int n, i;
4
5      scanf("%ld", &n);
6      double v[n]; /* Vetor alocado com tamanho n não pré-estabelecido */
7
8      for (i = 0; i < n; i++)
9          v[i] = i;
10     for (i = 0; i < n; i++)
11         printf("%.2lf\n", v[i]);
12     return 0;
13 }
```

Organização da memória do computador

Porém, a criação de vetores desta forma faz a alocação de memória na **pilha**, que possui um limite máximo.

Execute o programa a seguir digitando 1000000 e depois 2000000.

```
1  #include <stdio.h>
2  int main() {
3      long int n, i;
4
5      scanf("%ld", &n);
6      double v[n]; /* Vetor alocado com tamanho n não pré-estabelecido */
7
8      for (i = 0; i < n; i++)
9          v[i] = i;
10     for (i = 0; i < n; i++)
11         printf("%.2lf\n", v[i]);
12     return 0;
13 }
```

- O programa anterior será encerrado (*segmentation fault*) se for usado um valor grande o suficiente para n .
- Isto se deve ao fato de que o SO limita o que pode ser alocado na pilha na execução de uma função.
- Este limite não existe para o Heap (com exceção do limite de memória do computador).

Organização da memória do computador

Utilizando alocação dinâmica não temos o problema de erro do programa anterior.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      long int n = 2000000, i;
6      double *v = malloc(n * sizeof(double));
7      for (i = 0; i < n; i++)
8          v[i] = i;
9      for (i = 0; i < n; i++)
10         printf("%.2lf\n", v[i]);
11     free(v);
12     return 0;
13 }
```

Exemplos de ponteiros e alocação dinâmica

Vamos criar uma aplicação que usa vetores dinâmicos e implementa as seguintes operações:

- Inclusão de um elemento no final do vetor.
- Exclusão da primeira ocorrência de um elemento no vetor.
- Impressão do vetor.

Exemplo de ponteiros e alocação dinâmica

- O tamanho do vetor deve se ajustar automaticamente: se elementos são inseridos, devemos “aumentar” o tamanho do vetor para inclusão de novos elementos, e se elementos forem removidos devemos “diminuir” o tamanho vetor.
- Temos duas variáveis associadas ao vetor:
 - `tam`: denota quantos elementos estão armazenados no vetor.
 - `max_tam`: denota o tamanho alocado do vetor.

Temos as seguintes regras para ajuste do tamanho alocado do vetor:

- O vetor deve ter tamanho alocado de pelo menos 4.
- Se o vetor ficar cheio, então devemos alocar um novo vetor com o dobro do tamanho atual.
- Se o número de elementos armazenados no vetor for menor do que $1/4$ do tamanho alocado do vetor, então devemos alocar um novo vetor com metade do tamanho atual.

Exemplo de ponteiros e alocação dinâmica

Implementaremos as seguintes funções:

```
1 int *cria_vet(int *tam, int *max_tam);
```

Aloca um vetor inicial de tamanho 4, inicializando `tam` com valor 0, `max_tam` com valor 4, e devolvendo o endereço do vetor alocado.

```
1 void imprime_vet(int *v, int tam, int max_tam);
```

Imprime o conteúdo e tamanhos associados ao vetor `v`.

```
1 int *adiciona(int *v, int *tam, int *max_tam, int elem);
```

Adiciona o elemento `elem` no final do vetor `v`. Caso não haja espaço, um novo vetor com o dobro do tamanho deve ser alocado. A função sempre retorna o endereço do vetor, sendo o novo alocado ou o atual. Além disso, os valores de `tam` e `max_tam` devem ser atualizados.

Exemplo de ponteiros e alocação dinâmica

Implementaremos as seguintes funções:

```
1 int busca(int *v, int tam, int elem);
```

Determina se o elemento `elem` está presente ou não no vetor `v`. Caso esteja presente, retorna a posição da primeira ocorrência de `elem` em `v`. Caso contrário, retorna `-1`.

```
1 int *remove(int *v, int *tam, int *max_tam, int elem);
```

Remove a primeira ocorrência do elemento `elem` do vetor `v`, caso este esteja presente. O valor de `tam` deve ser decrementado de 1. Caso o número de elementos armazenados seja menor do que $\frac{1}{4}max_tam$, então um novo vetor de tamanho $\frac{1}{2}max_tam$ deve ser alocado no lugar de `v`. A função sempre retorna o endereço inicial do vetor, sendo um novo vetor alocado ou não.

Exemplo de ponteiros e alocação dinâmica

```
1  /* Cria vetor com tamanho total 4. Devolve o endereço do vetor criado.
   ↪ */
2  int *cria_vet(int *tam, int *max_tam) {
3      int *v = malloc(4 * sizeof(int));
4      *tam = 0;
5      *max_tam = 4;
6      return v;
7  }
8
9  /* Imprime o vetor. */
10 void imprime_vet(int *v, int tam, int max_tam) {
11     int i;
12     printf("Vetor de tamanho %d (max. alocado %d):\n", tam, max_tam);
13     printf("%d", v[0]);
14     for (i = 1; i < tam; i++)
15         printf(", %d", v[i]);
16     printf("\n");
17 }
```

Exemplo de ponteiros e alocação dinâmica

```
1  int *adiciona(int *v, int *tam, int *max_tam, int elem) {
2      if (*tam < *max_tam) {
3          /* Tem espaço para o novo elemento. */
4          v[*tam] = e;
5          (*tam)++;
6          return v;
7      } else {
8          /* Precisamos alocar um espaço maior. */
9          ...
10     }
11 }
```

Exemplo de ponteiros e alocação dinâmica

```
1  int *adiciona(int *v, int *tam, int *max_tam, int elem) {
2      if (*tam < *max_tam) {
3          /* Tem espaço para o novo elemento. */
4          ...
5      } else {
6          /* Precisamos alocar um espaço maior. */
7          int *vaux = malloc(2 * (*max_tam) * (sizeof(int)));
8          int i;
9          for (i = 0; i < *tam; i++) /* Salva dados de v em vaux. */
10             vaux[i] = v[i];
11         vaux[*tam] = e; /* Adiciona elemento no fim. */
12         (*tam)++;
13         *max_tam = 2 * (*max_tam); /* Atualiza dados de tamanho. */
14
15         free(v); /* Libera memória não mais necessária. */
16         return vaux;
17     }
18 }
```

Exemplo de ponteiros e alocação dinâmica

```
1  /* Retorna posição da primeira ocorrência de elem ou -1 caso elem não  
   ↪ seja encontrado. */  
2  int busca(int *v, int tam, int elem) {  
3     int i;  
4     for (i = 0; i < tam; i++)  
5         if (v[i] == elem)  
6             return i;  
7     return -1;  
8 }
```

Exemplo de ponteiros e alocação dinâmica

```
1  int *remove(int *v, int *tam, int *max_tam, int elem) {
2      int i;
3      i = busca(v, *tam, elem);
4      if (i != -1) {
5          /* O elemento está em v. */
6          /* Copia dados a partir da posição i+1 uma posição para trás. */
7          for (; i < (*tam)-1; i++)
8              v[i] = v[i+1];
9          (*tam)--;
10
11         /* Se tamanho do vetor for > 4 e ele estiver menos de 1/4 ocupado
12         ↪ devemos diminuir tamanho do vetor pela metade. */
13         if (*tam < (0.25 * (*max_tam)) && *max_tam > 4) {
14             ...
15             /* Exercício. */
16         }
17     }
18     return v;
19 }
```

Exemplo de ponteiros e alocação dinâmica

Com essas funções podemos executar o seguinte exemplo:

```
1  int main() {
2      int *vet, tam, max_tam, i;
3      vet = cria_vet(&tam, &max_tam);
4
5      for (i = 0; i < 20; i++)
6          vet = adiciona(vet, &tam, &max_tam, i);
7      imprime_vet(vet, tam, max_tam);
8
9      vet = remove(vet, &tam, &max_tam, 14);
10     imprime_vet(vet, tam, max_tam);
11     for (i = 5; i < 15; i++)
12         vet = remove(vet, &tam, &max_tam, i);
13     imprime_vet(vet, tam, max_tam);
14
15     for (i = 0; i < 20; i++)
16         vet = adiciona(vet, &tam, &max_tam, i);
17     imprime_vet(vet, tam, max_tam);
18
19     free(vet);
20     return 0;
```

Informações extras: ponteiros para ponteiros e alocação dinâmica de matrizes

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

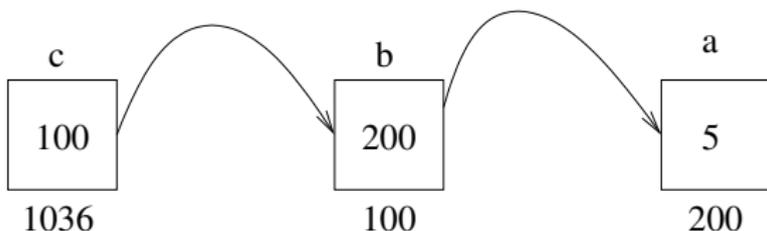
Informações extras: ponteiros para ponteiros

- Uma variável ponteiro está alocada na memória do computador como qualquer outra variável.
- Portanto, podemos criar um ponteiro que contém o endereço de memória de um outro ponteiro.
- Para criar um ponteiro para ponteiro: tipo `**nomePonteiro`;

```
1 int main() {
2     int a = 5, *b, **c;
3     b = &a;
4     c = &b;
5     printf("%d\n", a);
6     printf("%d\n", *b);
7     printf("%d\n", *(*c));
8     return 0;
9 }
```

Informações extras: ponteiros para ponteiros

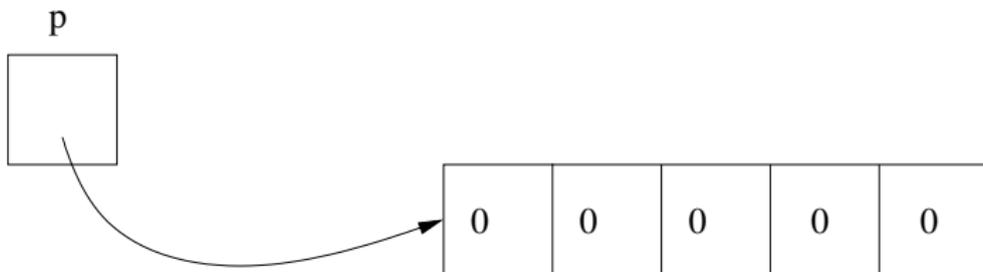
O programa imprime 5 três vezes, mostrando as três formas de acesso à variável a: a, *b, **c.



Informações extras: ponteiros para ponteiros

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

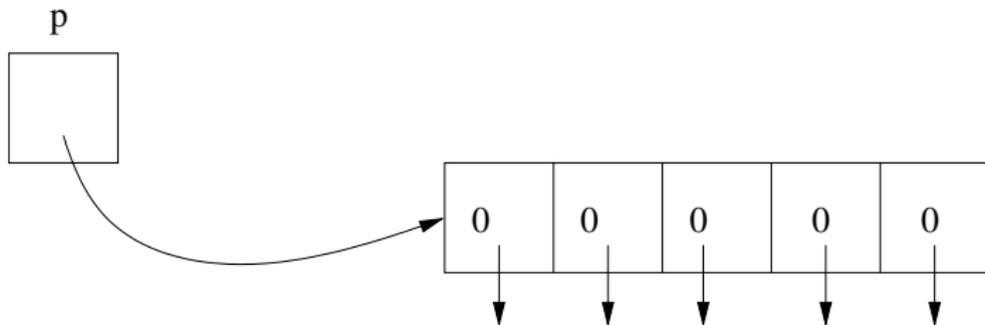
```
1 int *p;  
2 p = calloc(5, sizeof(int));
```



Informações extras: ponteiros para ponteiros

A mesma coisa acontece com um ponteiro para ponteiro, só que neste caso o vetor alocado é de ponteiros.

```
1 int **p;  
2 p = calloc(5, sizeof(int *));
```

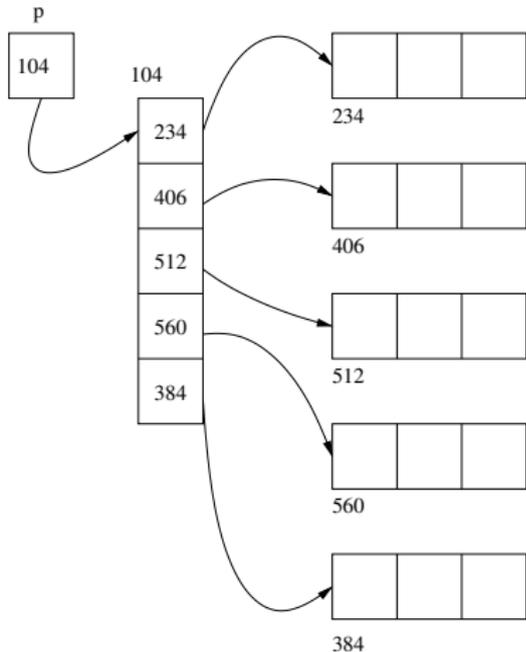


Note que cada posição do vetor acima é do tipo `int *`, ou seja, um ponteiro para inteiro!

Informações extras: ponteiros para ponteiros

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

```
1 int **p;  
2 int i;  
3 p = calloc(5, sizeof(int *));  
4 for (i = 0; i < 5; i++)  
5     p[i] = calloc(3, sizeof(int));
```



Esta é uma forma de se criar matrizes dinamicamente:

- Crie um ponteiro para ponteiro.
- Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor é o número de linhas da matriz.
- Cada posição do vetor será associada com um outro vetor do tipo a ser armazenado. Cada um destes vetores é uma linha da matriz (portanto possui tamanho igual ao número de colunas).

OBS: No final você deve desalocar toda a memória alocada!!

Informações extras: alocação dinâmica de matrizes

```
1  int main() {
2      int **p, i, j;
3
4      p = calloc(5, sizeof(int *));
5      for (i = 0; i < 5; i++)
6          p[i] = calloc(3, sizeof(int));
7      /* Alocou matriz 5x3 acima */
8
9      printf("Digite os valores da matriz\n");
10     for (i = 0; i < 5; i++)
11         for (j = 0; j < 3; j++)
12             scanf("%d", &p[i][j]);
13
14     printf("Matriz lida\n");
15     for (i = 0; i < 5; i++) {
16         for (j = 0; j < 3; j++)
17             printf("%d, ", p[i][j]);
18         printf("\n");
19     }
20
21     /* Desalocando a memória usada: */
22     for (i = 0; i < 5; i++)
23         free(p[i]);
24     free(p);
25     return 0;
26 }
```

Exercícios

Exercício 1

Qual o resultado da execução do programa abaixo? Ocorre algum erro?

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *misterio(int n) {
5      int i, *vet;
6
7      vet = malloc(n * sizeof(int));
8
9      vet[0] = 1;
10     for (i = 1; i < n; i++)
11         vet[i] = i * vet[i-1];
12
13     return vet;
14 }
15
16 int main() {
17     int i, n, *v;
18     scanf("%d", &n);
19     v = misterio(n);
20     for (i = 0; i < n; i++)
21         printf("%d\n", v[i]);
22     free(v);
23     return 0;
24 }
```

Exercício 2

Faça um programa que lê a dimensão n de um vetor, em seguida aloca dinamicamente dois vetores do tipo *double* de dimensão n , faz a leitura de cada vetor e, finalmente, imprime o resultado da soma dos dois vetores.

Exercício 3

Faça uma função que recebe como parâmetro dois vetores de inteiros representando conjuntos de números inteiros e devolve um outro vetor com o resultado da união dos dois conjuntos.

O vetor resultante deve ser alocado dinamicamente.

O protótipo da função é

```
1 int *uniao(int *v1, int n1, int *v2, int n2);
```

onde **n1** e **n2** indicam o número de elementos em **v1** e **v2**, respectivamente.

Exercício 4

Vimos uma aplicação que aumenta e diminui o tamanho do vetor conforme necessário durante a execução.

Implemente a função de remoção de um elemento do vetor.

Informações extras: alocação dinâmica de matrizes

Vimos que alocar ponteiros para ponteiros é uma forma de se fazer alocação dinâmica de matrizes.

Mas a forma mais eficiente de criar matrizes é:

- Para uma matriz de dimensões $n \times m$, crie dinamicamente um vetor *unidimensional* deste tamanho.
- Use linearização de índices para trabalhar com o vetor como se fosse uma matriz.
- Desta forma, tem-se um melhor aproveitamento da cache pois a matriz inteira está sequencialmente em memória.

No final você deve desalocar toda a memória alocada!!

- Podemos sempre usar vetores simples para representar matrizes (na prática o compilador faz isto por você).
- Ao declarar uma matriz como `int mat[3][4]`, sabemos que serão alocadas 12 posições de memória associadas com a variável `mat`.
- Poderíamos simplesmente criar `int mat[12]` então.
- Apenas perderíamos a simplicidade de uso dos índices em forma de matriz.
 - Você não poderá escrever `mat[1][3]`, por exemplo.

- A *linearização de índices* é justamente a representação de matrizes usando-se um vetor simples.
- Mas devemos ter um padrão para acessar as posições deste vetor como se sua organização fosse na forma de matriz.

- Considere o exemplo: `int mat[12]; /* ao invés de int mat[3][4] */`
- Fazemos a divisão por linhas como segue:
 - Primeira linha: `mat[0]` até `mat[3]`
 - Segunda linha: `mat[4]` até `mat[7]`
 - Terceira linha: `mat[8]` até `mat[11]`
- Para acessar uma posição “[i][j]”, usamos:
 - `mat[i*4 + j]`; onde $0 \leq i \leq 2$ e $0 \leq j \leq 3$.

- De forma geral, seja matriz $\text{mat}[n*m]$, representando $\text{mat}[n][m]$.
- Para acessar a posição correspondente à $[i][j]$ usamos:
 - $\text{mat}[i*m + j]$; onde $0 \leq i \leq n - 1$ e $0 \leq j \leq m - 1$.
- Note que i pula de blocos de tamanho m , e j indexa a posição dentro de um bloco.

Linearização de índices

```
1  int main() {
2      int mat[40]; /* representando mat[5][8] */
3      int i, j;
4
5      for (i = 0; i < 5; i++)
6          for (j = 0; j < 8; j++)
7              mat[i*8 + j] = i*j;
8
9      for (i = 0; i < 5; i++) {
10         for (j = 0; j < 8; j++)
11             printf("%d, ", mat[i*8 + j]);
12         printf("\n");
13     }
14
15     return 0;
16 }
```