

Ponteiros e alocação dinâmica em C

Paulo Guilherme de Lima Freire

Supervisor: Prof. Dr. Ednaldo Brigante Pizzolato

Departamento de Computação

Universidade Federal de São Carlos

Agenda

Ponteiros

Alocação dinâmica

Ponteiros

Ponteiros

Introdução

Aplicações

Alocação dinâmica



- Para entender ponteiros, primeiro é preciso entender a memória.

- Para entender ponteiros, primeiro é preciso entender a memória.
- A memória RAM nada mais é do que uma sequência de bytes.

- Para entender ponteiros, primeiro é preciso entender a memória.
- A memória RAM nada mais é do que uma sequência de bytes.
- Os bytes são numerados de forma sequencial.

- Para entender ponteiros, primeiro é preciso entender a memória.
- A memória RAM nada mais é do que uma sequência de bytes.
- Os bytes são numerados de forma sequencial.
 - O número de um byte qualquer é seu *endereço*.

- Para entender ponteiros, primeiro é preciso entender a memória.
- A memória RAM nada mais é do que uma sequência de bytes.
- Os bytes são numerados de forma sequencial.
 - O número de um byte qualquer é seu *endereço*.
- Tudo o que estiver na memória tem um endereço.

- Para entender ponteiros, primeiro é preciso entender a memória.
- A memória RAM nada mais é do que uma sequência de bytes.
- Os bytes são numerados de forma sequencial.
 - O número de um byte qualquer é seu *endereço*.
- Tudo o que estiver na memória tem um endereço.
 - Operador &.

- Um ponteiro armazena endereços.

- Um ponteiro armazena endereços.
 - Ele “aponta” para um local da memória.

- Um ponteiro armazena endereços.
 - Ele “aponta” para um local da memória.
- Com o endereço em mãos podemos acessar seu conteúdo.

- Um ponteiro armazena endereços.
 - Ele “aponta” para um local da memória.
- Com o endereço em mãos podemos acessar seu conteúdo.
 - Operador *.

- Os ponteiros podem apontar para diferentes tipos de dados.

- Os ponteiros podem apontar para diferentes tipos de dados.
 - `int *p`

- Os ponteiros podem apontar para diferentes tipos de dados.
 - `int *p`
 - `char *p`

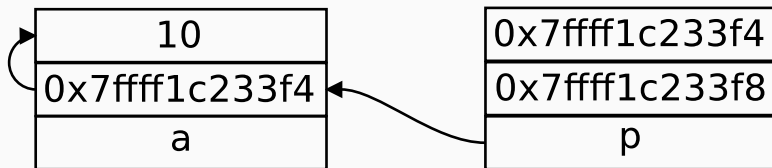
- Os ponteiros podem apontar para diferentes tipos de dados.
 - `int *p`
 - `char *p`
 - `double *p`

- Os ponteiros podem apontar para diferentes tipos de dados.
 - `int *p`
 - `char *p`
 - `double *p`
 - `int **p` (sim, ponteiro para ponteiro!),

Introdução

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p; // declaracao de ponteiro
7     int a = 10;
8     p = &a; // recebe o endereco de a
9     printf("%d %d\n", a, *p); // 10 10
10    return 0;
11 }
```

Introdução



- Tudo o que for alterado em a, vai alterar *p (e vice versa).

Introdução

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p;
7     int a = 10;
8     p = &a;
9     (*p)++; // incrementa o conteúdo
10    printf("%d %d\n", a, *p); // ?
11    return 0;
12 }
```

- Essa liberdade com ponteiros é muito útil.

- Essa liberdade com ponteiros é muito útil.
- Mas deve ser usada com cuidado.

- Essa liberdade com ponteiros é muito útil.
- Mas deve ser usada com cuidado.
 - Você passa a ter controle direto sobre o conteúdo de um endereço de memória.

- Essa liberdade com ponteiros é muito útil.
- Mas deve ser usada com cuidado.
 - Você passa a ter controle direto sobre o conteúdo de um endereço de memória.
- Atenção para onde você está apontando!

Introdução

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p;
7     int a = 10;
8     p = &a;
9     printf("%p\n", p); // 0x7ffff1c233f4
10    p++; // incrementa ...
11    printf("%d %d\n", a, *p); // ?
12    printf("%p\n", p); // ?
13    return 0;
14 }
```

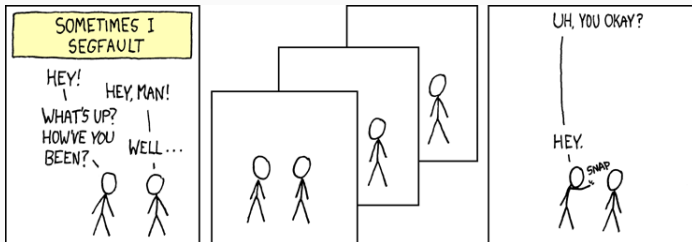
- Um ponteiro pode ainda ter um valor especial: NULL.

- Um ponteiro pode ainda ter um valor especial: NULL.
 - O valor NULL não é endereço de memória.

- Um ponteiro pode ainda ter um valor especial: NULL.
 - O valor NULL não é endereço de memória.
- Todo ponteiro declarado, mas não inicializado, tem esse valor.

Introdução

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *p, a = 10;
7      printf("%p\n", p); // nil
8      *p = a; // errou feio , errou rude
9      return 0;
10 }
```

- Em suma

- Em suma
 - Um ponteiro p tem como valor um endereço de memória.

- Em suma
 - Um ponteiro `p` tem como valor um endereço de memória.
 - Para pegar um endereço, usamos o operador `&`.

- Em suma
 - Um ponteiro `p` tem como valor um endereço de memória.
 - Para pegar um endereço, usamos o operador `&`.
 - Com `p` **inicializado**, podemos acessar seu conteúdo.

- Em suma
 - Um ponteiro `p` tem como valor um endereço de memória.
 - Para pegar um endereço, usamos o operador `&`.
 - Com `p` **inicializado**, podemos acessar seu conteúdo.
 - Para pegar o conteúdo, usamos o operador `*`.

- Em suma
 - Um ponteiro `p` tem como valor um endereço de memória.
 - Para pegar um endereço, usamos o operador `&`.
 - Com `p` **inicializado**, podemos acessar seu conteúdo.
 - Para pegar o conteúdo, usamos o operador `*`.
- Cuidado com o que você altera e onde você aponta.

Ponteiros

Introdução

Aplicações

Alocação dinâmica

- Pense num algoritmo que troca os valores de duas variáveis.

- Pense num algoritmo que troca os valores de duas variáveis.
 - Pra fazer isso, usamos uma variável auxiliar.

- Pense num algoritmo que troca os valores de duas variáveis.
 - Pra fazer isso, usamos uma variável auxiliar.
- Mas e se o programa precisar fazer muitas trocas?

- Pense num algoritmo que troca os valores de duas variáveis.
 - Pra fazer isso, usamos uma variável auxiliar.
- Mas e se o programa precisar fazer muitas trocas?
 - O código vai ficar repetitivo.

- Pense num algoritmo que troca os valores de duas variáveis.
 - Pra fazer isso, usamos uma variável auxiliar.
- Mas e se o programa precisar fazer muitas trocas?
 - O código vai ficar repetitivo.
- Ponteiros ajudam! Mas antes...

- Iremos fazer uso de um procedimento em C.

- Iremos fazer uso de um procedimento em C.
- É um trecho de código que é chamado de alguma função (ou procedimento) e realiza alguma ação.

- Iremos fazer uso de um procedimento em C.
- É um trecho de código que é chamado de alguma função (ou procedimento) e realiza alguma ação.
 - Nesse contexto, a função é a `main()`.

- Iremos fazer uso de um procedimento em C.
- É um trecho de código que é chamado de alguma função (ou procedimento) e realiza alguma ação.
 - Nesse contexto, a função é a `main()`.
- Procedimentos, assim como funções, ajudam a "quebrar" o código.

- Iremos fazer uso de um procedimento em C.
- É um trecho de código que é chamado de alguma função (ou procedimento) e realiza alguma ação.
 - Nesse contexto, a função é a `main()`.
- Procedimentos, assim como funções, ajudam a "quebrar" o código.
 - Mais organização.

- Iremos fazer uso de um procedimento em C.
- É um trecho de código que é chamado de alguma função (ou procedimento) e realiza alguma ação.
 - Nesse contexto, a função é a `main()`.
- Procedimentos, assim como funções, ajudam a "quebrar" o código.
 - Mais organização.
 - Menos repetição.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a = 10, b = 20;
7     printf("%d %d\n", a, b); // 10 20
8     troca(&a, &b);
9     printf("%d %d\n", a, b); // 20 10
10    return 0;
11 }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void troca(int *a, int *b)
5  {
6      int temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10
11 int main()
12 {
13     int a = 10, b = 20;
14     printf("%d %d\n", a, b); // 10 20
15     troca(&a, &b);
16     printf("%d %d\n", a, b); // 20 10
17     return 0;
18 }
```

- Outro ponto útil: ponteiros com endereços de vetores!

- Outro ponto útil: ponteiros com endereços de vetores!
- Um programa pode ler vários vetores diferentes e ordená-los.

- Outro ponto útil: ponteiros com endereços de vetores!
- Um programa pode ler vários vetores diferentes e ordená-los.
 - Usando um único trecho de código.


```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a[5] = {10, 8, 1, 3, 15};
7     int b[5] = {15, 1, 35, 0, 2};
8     selection_sort(&a[0], 5); // 1 3 8 10 15
9     selection_sort(&b[0], 5); // 0 1 2 15 35
10
11     return 0;
12 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void selection_sort(int *a, int n)
5 {
6     int i, j, pos_menor;
7     for(i = 0; i < n; ++i)
8     {
9         pos_menor = i;
10        for(j = i + 1; j < n; ++j)
11            if(a[j] < a[pos_menor])
12                pos_menor = j;
13        if(pos_menor != i)
14            troca(&a[i], &a[pos_menor]);
15    }
16 }
```

Alocação dinâmica

Ponteiros

Alocação dinâmica

Motivação

Funções de alocação

Alocação unidimensional

Alocação bidimensional

- As alocações de vetor feitas até o momento foram estáticas.

- As alocações de vetor feitas até o momento foram estáticas.
 - O compilador sabia o tamanho do vetor antes da execução.

- As alocações de vetor feitas até o momento foram estáticas.
 - O compilador sabia o tamanho do vetor antes da execução.
- Mas isso pode levar a desperdício (ou falta) de memória.

- As alocações de vetor feitas até o momento foram estáticas.
 - O compilador sabia o tamanho do vetor antes da execução.
- Mas isso pode levar a desperdício (ou falta) de memória.
- Há informações que só temos acesso no tempo de execução do código.

- Imagine um vetor de float de 30 posições para armazenar notas.

- Imagine um vetor de float de 30 posições para armazenar notas.
 - E se 5 novos alunos entrarem na turma?

- Imagine um vetor de float de 30 posições para armazenar notas.
 - E se 5 novos alunos entrarem na turma?
 - E se 10 alunos desistirem?

- Imagine um vetor de float de 30 posições para armazenar notas.
 - E se 5 novos alunos entrarem na turma?
 - E se 10 alunos desistirem?
- Solução: alocação dinâmica.

- Imagine um vetor de float de 30 posições para armazenar notas.
 - E se 5 novos alunos entrarem na turma?
 - E se 10 alunos desistirem?
- Solução: alocação dinâmica.
 - Aloca memória em tempo de execução.

Ponteiros

Alocação dinâmica

Motivação

Funções de alocação

Alocação unidimensional

Alocação bidimensional

- Função `malloc(tamanho * sizeof(tipo))`

- Função `malloc(tamanho * sizeof(tipo))`
 - Parâmetros: tamanho do vetor e tipo de dado a ser armazenado.

- Função `malloc(tamanho * sizeof(tipo))`
 - Parâmetros: tamanho do vetor e tipo de dado a ser armazenado.
 - Aloca o espaço necessário.

- Função `malloc(tamanho * sizeof(tipo))`
 - Parâmetros: tamanho do vetor e tipo de dado a ser armazenado.
 - Aloca o espaço necessário.
 - Retorna um ponteiro para o primeiro endereço do bloco (ou NULL se não conseguir alocar).

- Função `free(ptr)`

- Função `free(ptr)`
 - Parâmetro: ponteiro que aponta para o bloco de memória alocado.

- Função `free(ptr)`
 - Parâmetro: ponteiro que aponta para o bloco de memória alocado.
 - Desaloca a memória.

- Regras de ouro para alocação dinâmica:

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.
 - Aloque usando `malloc` (faça um *cast* para o seu tipo de dado!).

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.
 - Aloque usando `malloc` (faça um *cast* para o seu tipo de dado!).
 - Verifique se o ponteiro retornado não é `NULL`.

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.
 - Aloque usando `malloc` (faça um *cast* para o seu tipo de dado!).
 - Verifique se o ponteiro retornado não é `NULL`.
 - Faça as operações desejadas.

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.
 - Aloque usando `malloc` (faça um *cast* para o seu tipo de dado!).
 - Verifique se o ponteiro retornado não é `NULL`.
 - Faça as operações desejadas.
 - Desaloque a memória ao final do uso com `free`.

- Regras de ouro para alocação dinâmica:
 - Saiba o tamanho e tipo de dado que você irá alocar.
 - Aloque usando `malloc` (faça um *cast* para o seu tipo de dado!).
 - Verifique se o ponteiro retornado não é `NULL`.
 - Faça as operações desejadas.
 - Desaloque a memória ao final do uso com `free`.
- Não seja deselegante: devolva sempre aquilo que você pediu.

Ponteiros

Alocação dinâmica

Motivação

Funções de alocação

Alocação unidimensional

Alocação bidimensional

- Iremos ler um inteiro N e criar um vetor com esse tamanho em tempo de execução.

- Iremos ler um inteiro N e criar um vetor com esse tamanho em tempo de execução.
- Novamente:

- Iremos ler um inteiro N e criar um vetor com esse tamanho em tempo de execução.
- Novamente:
 - Aloque usando `malloc`.

- Iremos ler um inteiro `N` e criar um vetor com esse tamanho em tempo de execução.
- Novamente:
 - Aloque usando `malloc`.
 - Verifique se alocou (`ptr != NULL`).

- Iremos ler um inteiro `N` e criar um vetor com esse tamanho em tempo de execução.
- Novamente:
 - Aloque usando `malloc`.
 - Verifique se alocou (`ptr != NULL`).
 - Faça as operações que desejar no vetor.

- Iremos ler um inteiro `N` e criar um vetor com esse tamanho em tempo de execução.
- Novamente:
 - Aloque usando `malloc`.
 - Verifique se alocou (`ptr != NULL`).
 - Faça as operações que desejar no vetor.
 - Devolva a memória usando `free`.

Alocação unidimensional

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int n, i, *vetor;
7     printf("Entre com o tamanho do vetor: ");
8     scanf("%d", &n);
9     vetor = (int*)malloc(n * sizeof(int));
10    if(vetor == NULL)
11        return -1;
12    for(i = 0; i < n; ++i)
13        vetor[i] = i + 1; /*(vetor + i) = i + 1
14    for(i = 0; i < n; ++i)
15        printf("%d ", vetor[i]); /*(vetor + i)
16    printf("\n");
17
18    free(vetor);
19    return 0;
20 }
```

- Note que `vetor[i]` e `*(vetor + i)` fazem a mesma coisa!

- Note que `vetor[i]` e `*(vetor + i)` fazem a mesma coisa!
 - São duas formas distintas de indicar o conteúdo de uma posição de memória.

- Note que `vetor[i]` e `*(vetor + i)` fazem a mesma coisa!
 - São duas formas distintas de indicar o conteúdo de uma posição de memória.
- A aritmética de ponteiros funciona da mesma forma que qualquer outra.

- Note que `vetor[i]` e `*(vetor + i)` fazem a mesma coisa!
 - São duas formas distintas de indicar o conteúdo de uma posição de memória.
- A aritmética de ponteiros funciona da mesma forma que qualquer outra.
 - A diferença é que ela mexe com posições de memória.

- E se quiséssemos ordenar o vetor?

- E se quiséssemos ordenar o vetor?
- O código do selection sort visto mais cedo se mantém o mesmo!

- E se quiséssemos ordenar o vetor?
- O código do selection sort visto mais cedo se mantém o mesmo!
 - Basta passar o ponteiro como parâmetro.

Alocação unidimensional

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main()
6  {
7      int n, i, *vetor;
8      srand(time(NULL));
9      scanf("%d", &n);
10     vetor = (int*)malloc(n * sizeof(int));
11     if(vetor == NULL)
12         return -1;
13     for(i = 0; i < n; ++i)
14         vetor[i] = rand() % 100;
15
16     selection_sort(vetor, n);
17
18     free(vetor);
19     return 0;
20 }
```

Ponteiros

Alocação dinâmica

Motivação

Funções de alocação

Alocação unidimensional

Alocação bidimensional

- Iremos ler dois inteiros N e M e criar uma matriz $N \times M$ em tempo de execução.

- Iremos ler dois inteiros N e M e criar uma matriz $N \times M$ em tempo de execução.
- Há duas formas de fazer isso.

- Iremos ler dois inteiros N e M e criar uma matriz $N \times M$ em tempo de execução.
- Há duas formas de fazer isso.
- Agora é hora de usar ponteiros para ponteiros :-)

Alocação bidimensional

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

- Essa forma de alocação envolve:

- Essa forma de alocação envolve:
 - Criar um vetor de N ponteiros.

- Essa forma de alocação envolve:
 - Criar um vetor de N ponteiros.
 - Para cada posição nesse vetor, alocar outro com tamanho M .

- Essa forma de alocação envolve:
 - Criar um vetor de N ponteiros.
 - Para cada posição nesse vetor, alocar outro com tamanho M .
 - Muitas chamadas ao `malloc`.

- Essa forma de alocação envolve:
 - Criar um vetor de N ponteiros.
 - Para cada posição nesse vetor, alocar outro com tamanho M .
 - Muitas chamadas ao `malloc`.
 - Liberação de memória mais “chata” de ser feita.

- Essa forma de alocação envolve:
 - Criar um vetor de N ponteiros.
 - Para cada posição nesse vetor, alocar outro com tamanho M .
 - Muitas chamadas ao `malloc`.
 - Liberação de memória mais “chata” de ser feita.
- Essa abordagem funciona, mas há outra maneira...

Alocação bidimensional

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *vetor, **matriz;
7      int n_linhas, n_colunas, i;
8      scanf("%d %d", &n_linhas, &n_colunas);
9      vetor = (int*)malloc(n_linhas * n_colunas * sizeof(int));
10     if(vetor == NULL)
11         return -1;
12     matriz = (int**)malloc(n_linhas * sizeof(int*));
13     if(matriz == NULL)
14         return -1;
15     for(i = 0; i < n_linhas; ++i)
16         matriz[i] = &vetor[i * n_colunas];
17     free(vetor);
18     free(matriz);
19     return 0;
20 }
```


- Essa forma de alocação envolve:

- Essa forma de alocação envolve:
 - Criar um vetor com o tamanho $N \times M$ da matriz.

- Essa forma de alocação envolve:
 - Criar um vetor com o tamanho $N \times M$ da matriz.
 - Criar um vetor de tamanho N para armazenar o início de cada linha.

- Essa forma de alocação envolve:
 - Criar um vetor com o tamanho $N \times M$ da matriz.
 - Criar um vetor de tamanho N para armazenar o início de cada linha.
 - Poucas chamadas ao `malloc` (apenas duas).

- Essa forma de alocação envolve:
 - Criar um vetor com o tamanho $N \times M$ da matriz.
 - Criar um vetor de tamanho N para armazenar o início de cada linha.
 - Poucas chamadas ao `malloc` (apenas duas).
 - Liberação de memória feita de maneira direta.

- Essa forma de alocação envolve:
 - Criar um vetor com o tamanho $N \times M$ da matriz.
 - Criar um vetor de tamanho N para armazenar o início de cada linha.
 - Poucas chamadas ao `malloc` (apenas duas).
 - Liberação de memória feita de maneira direta.
- Vantagem: os dados estão em posições contínuas na memória.

Bônus

- C99 é uma versão mais antiga do padrão da linguagem C.

- C99 é uma versão mais antiga do padrão da linguagem C.
- Ele dá suporte a *variable length arrays*.

- C99 é uma versão mais antiga do padrão da linguagem C.
- Ele dá suporte a *variable length arrays*.
- Aceita um vetor com tamanho definido em tempo de execução .

- C99 é uma versão mais antiga do padrão da linguagem C.
- Ele dá suporte a *variable length arrays*.
- Aceita um vetor com tamanho definido em tempo de execução .
- Mas não são todos os compiladores que aceitam esse padrão.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     int n, i;
7     scanf("%d", &n);
8     int vetor[n];
9     for(i = 0; i < n; ++i)
10         vetor[i] = i + 1;
11     for(i = 0; i < n; ++i)
12         printf("%d ", vetor[i]);
13     printf("\n");
14     return 0;
15 }
```

- A função `main` pode aceitar dois parâmetros:

- A função `main` pode aceitar dois parâmetros:
 - `int argc`: argument count.

- A função `main` pode aceitar dois parâmetros:
 - `int argc`: argument count.
 - `char **argv`: argument vector.

- A função `main` pode aceitar dois parâmetros:
 - `int argc`: argument count.
 - `char **argv`: argument vector.
- Parâmetros na linha de comando!

Parâmetros do main

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     int i;
7     if(argc != 2)
8         return -1;
9     for(i = 0; i < argc; ++i)
10         printf("%s ", argv[i]);
11     printf("\n");
12     return 0;
13 }
```

Fim