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

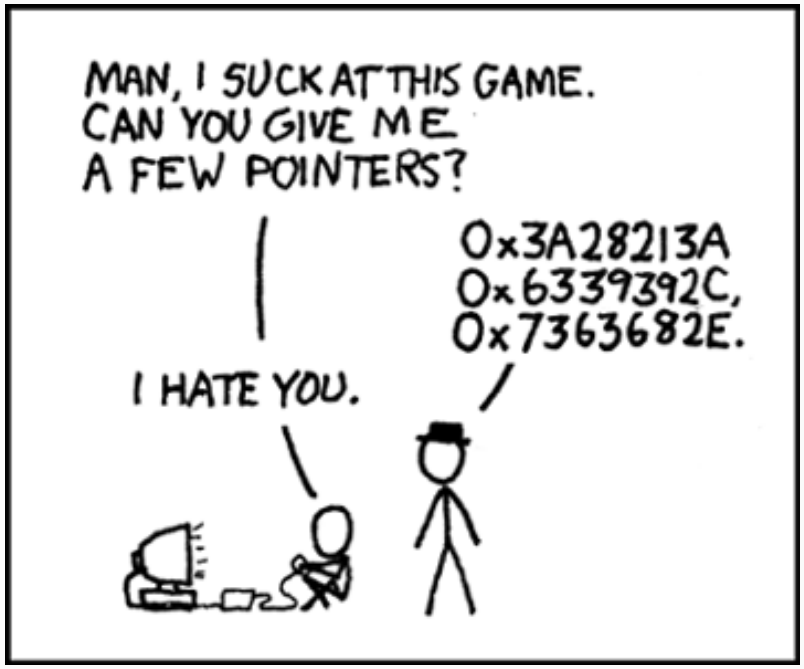
Agenda

Ponteiros

- Introdução
- Aplicações

Alocação dinâmica

Introdução


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Introdução

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

Introdução

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

Introdução

- 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.

Introdução

- 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*.

Introdução

- 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.

Introdução

- 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 `&`.

Introdução

- Um ponteiro armazena endereços.

Introdução

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

Introdução

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

Introdução

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

Introdução

- Os ponteiros podem apontar para diferentes tipos de dados.

Introdução

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

Introdução

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

Introdução

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

Introdução

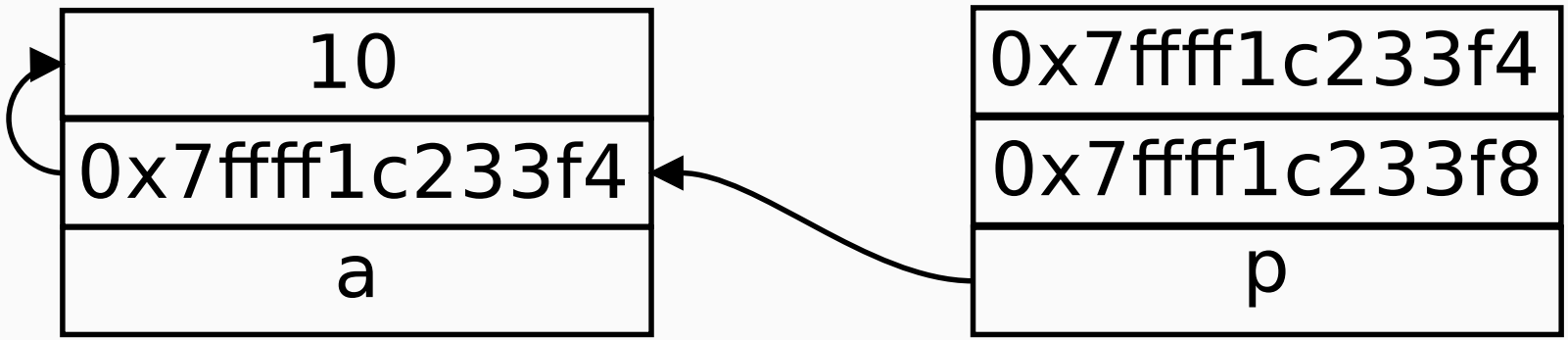
- Os ponteiros podem apontar para diferentes tipos de dados.
  - `int *p`
  - `char *p`
  - `double *p`
  - `int **p` (sim, ponteiro para ponteiro!),

Introdução

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

int main()
{
        int *p; // declaracao de ponteiro
        int a = 10;
        p = &a; // recebe o endereco de a
        printf("%d %d\n", a, *p); // 10 10
        return 0;
}
```

Introdução



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

Introdução

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

int main()
{
        int *p;
        int a = 10;
        p = &a;
        (*p)++; // incrementa o conteudo
        printf("%d %d\n", a, *p); // ?
        return 0;
}
```

Introdução

- Essa liberdade com ponteiros é muito útil.

Introdução

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

Introdução

- 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.

Introdução

- 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

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

int main()
{
        int *p;
        int a = 10;
        p = &a;
        printf("%p\n", p); // 0x7ffff1c233f4
        p++; // incrementa...
        printf("%d %d\n", a, *p); // ?
        printf("%p\n", p); // ?
        return 0;
}
```

Introdução

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

Introdução

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

Introdução

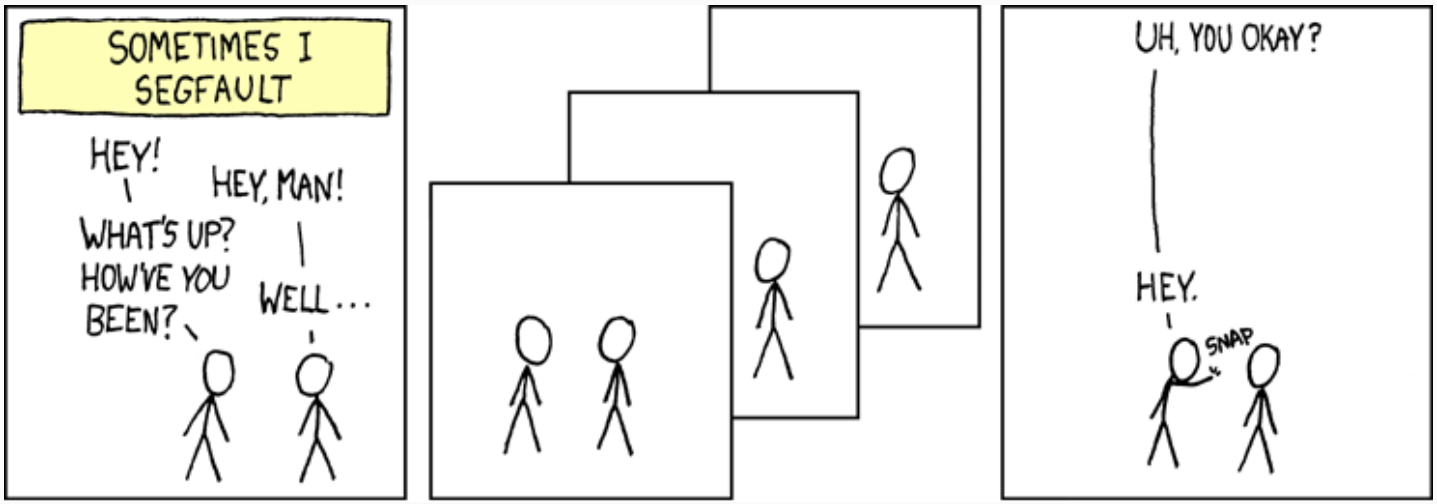
- Um ponteiro pode ainda ter um valor especial: `NULL`.
  - O valor `NULL` não é endereço de memória.
- Todo ponteiro declarado, mas não inicializado, tem esse valor.

Introdução

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

int main()
{
        int *p, a = 10;
        printf("%p\n", p); // nil
        *p = a; // errou feio, errou rude
        return 0;
}
```

Introdução


SOMETIMES I SEGFAULT
HEY!
HEY, MAN!
WHAT'S UP? HOW'VE YOU BEEN?
WELL...
UH, YOU OKAY?
HEY.
SNAP

Introdução

- Em suma

Introdução

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

Introdução

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

Introdução

- 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.

Introdução

- 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 `*`.

Introdução

- 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.

Agenda

Aplicações

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

Aplicações

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

Aplicações

- 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?

Aplicações

- 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.

Aplicações

- 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...

Aplicações

- Iremos fazer uso de um procedimento em C.

Aplicações

- 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.

Aplicações

- 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()`.

Aplicações

- 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.

Aplicações

- 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.

Aplicações

- 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.

Aplicações

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

int main()
{
        int a = 10, b = 20;
        printf("%d %d\n", a, b); // 10 20
        troca(&a, &b);
        printf("%d %d\n", a, b); // 20 10
        return 0;
}
```

Aplicações

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

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

int main()
{
        int a = 10, b = 20;
        printf("%d %d\n", a, b); // 10 20
        troca(&a, &b);
        printf("%d %d\n", a, b); // 20 10
        return 0;
}
```

Aplicações

- Outro ponto útil: ponteiros com endereços de vetores!

Aplicações

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

Aplicações

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

Aplicações

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

int main()
{
        int a[5] = {10, 8, 1, 3, 15};
        int b[5] = {15, 1, 35, 0, 2};
        selection_sort(&a[0], 5); // 1 3 8 10 15
        selection_sort(&b[0], 5); // 0 1 2 15 35

        return 0;
}
```

Aplicações

```
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

Agenda

Motivação

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

Motivação

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

Motivação

- 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.

Motivação

- 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.

Motivação

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

Motivação

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

Motivação

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

Motivação

- 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.

Motivação

- 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.

Agenda

Função de alocação

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

Função de alocação

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

Função de alocação

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

Função de alocação

- 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).

Funções de alocação

- Função `free(ptr)`

Funções de alocação

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

Funções de alocação

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

Funções de alocação

- Regras de ouro para alocação dinâmica:

Funções de alocação

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

Funções de alocação

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

Funções de alocação

- Regras de ouro para alocação dinâmica:
  - Saiba o tamanho e tipo de dado que você irá alocar.
  - Aloque usando `malloc`.
  - Verifique se o ponteiro retornado não é `NULL`.

Funções de alocação

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

Funções de alocação

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

Funções de alocação

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

Agenda

Alocação unidimensional

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

Alocação unidimensional

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

Alocação unidimensional

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

Alocação unidimensional

- 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`).

Alocação unidimensional

- 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.

Alocação unidimensional

- 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

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

int main()
{
        int n, i, *vetor;
        printf("Entre com o tamanho do vetor: ");
        scanf("%d", &n);
        vetor = malloc(n * sizeof(int));
        if(vetor == NULL)
                return -1;
        for(i = 0; i < n; ++i)
                vetor[i] = i + 1; //*(vetor + i) = i + 1
        for(i = 0; i < n; ++i)
                printf("%d ", vetor[i]); //*(vetor + i)
        printf("\n");

        free(vetor);
        return 0;
}
```

Alocação unidimensional

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

Alocação unidimensional

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

Alocação unidimensional

- 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.

Alocação unidimensional

- 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.

Alocação unidimensional

- E se quiséssemos ordenar o vetor?

Alocação unidimensional

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

Alocação unidimensional

- 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

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

int main()
{
        int n, i, *vetor;
        srand(time(NULL));
        scanf("%d", &n);
        vetor = malloc(n * sizeof(int));
        if(vetor == NULL)
                return -1;
        for(i = 0; i < n; ++i)
                vetor[i] = rand() % 100;

        selection_sort(vetor, n);

        free(vetor);
        return 0;
}
```

Agenda

Alocação bidimensional

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

Alocação bidimensional

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

Alocação bidimensional

- 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

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

int main()
{
        int **matriz, n_linhas, n_colunas, i;
        scanf("%d %d", &n_linhas, &n_colunas);
        matriz = malloc(n_linhas * sizeof(int*));
        if(matriz == NULL)
                return -1;
        for(i = 0; i < n_linhas; ++i)
        {
                matriz[i] = malloc(n_colunas * sizeof(int));
                if(matriz[i] == NULL)
                        return -1;
        }
        for(i = 0; i < n_linhas; ++i)
                free(matriz[i]);
        free(matriz);
        return 0;
}
```

Alocação bidimensional

- Essa forma de alocação envolve:

Alocação bidimensional

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

Alocação bidimensional

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

Alocação bidimensional

- 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`.

Alocação bidimensional

- 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.

Alocação bidimensional

- 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

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

int main()
{
        int *vetor, **matriz;
        int n_linhas, n_colunas, i;
        scanf("%d %d", &n_linhas, &n_colunas);
        vetor = malloc(n_linhas * n_colunas * sizeof(int));
        if(vetor == NULL)
                return -1;
        matriz = malloc(n_linhas * sizeof(int*));
        if(matriz == NULL)
                return -1;
        for(i = 0; i < n_linhas; ++i)
                matriz[i] = &vetor[i * n_colunas];
        free(vetor);
        free(matriz);
        return 0;
}
```

Alocação bidimensional

- Essa forma de alocação envolve:

Alocação bidimensional

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

Alocação bidimensional

- 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.

Alocação bidimensional

- 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).

Alocação bidimensional

- 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.

Alocação bidimensional

- 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

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

C99

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

C99

- 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

- 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.

C99

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

int main(int argc, char **argv)
{
        int n, i;
        scanf("%d", &n);
        int vetor[n];
        for(i = 0; i < n; ++i)
                vetor[i] = i + 1;
        for(i = 0; i < n; ++i)
                printf("%d ", vetor[i]);
        printf("\n");
        return 0;
}
```

Parâmetros do main

- A função `main` pode aceitar dois parâmetros:

Parâmetros do main

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

Parâmetros do main

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

Parâmetros do main

- 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