

Fundamentos de Programação

Vetores e alocação dinâmica de memória

Dainf - UTFPR

Profa. Leyza B. Dorini
Prof. Bogdan T. Nassu

Tamanho constante!

Temos usado vetores de tamanho constante.

```
1 int foo [100];  
2 char bar [BUFSIZE];
```

E quando o tamanho não é conhecido em tempo de compilação?

Exemplo: no Projeto 4, o conteúdo de um arquivo de áudio é lido e colocado em um vetor.

- O tamanho do vetor depende do arquivo...
- ... mas só vai ser conhecido quando o programa for executado!

Tamanho não é conhecido em tempo de compilação?

Até aqui, usamos um artifício: criamos o vetor com um tamanho máximo, e usamos uma variável para guardar o número de posições usadas de fato.

```
1      char vetor [TAM_MAX];
2      int i, n;
3
4      scanf ("%d", &n);
5
6      if (n > TAM_MAX)
7          n = TAM_MAX;
8
9      for (i = 0; i < n; i++)
10         vetor [i] = 0;
```

Esta pode não ser a melhor opção...

- TAM_MAX maior que o necessário → desperdício de espaço.
- TAM_MAX menor que o necessário → precisamos recompilar o programa com outro valor para TAM_MAX.

Alocação de memória?

Alocação de memória — variáveis locais

Alocar (dicionário): *destinar ou reservar para determinado fim.*

Variáveis locais (automáticas) (ex: `int foo; char bar;`):

- O computador aloca memória para cada variável.
- O espaço total é conhecido em tempo de compilação.
- A memória é reservada na *stack* (pilha).
 - Cada processo em execução tem sua própria *stack*.
- O espaço é desalocado automaticamente no final da função.

```
1  int func (int a, int b)
2  {
3
4      int x = a+b;
5      int vetor [20];
6
7      // (...)
8
9      return x;
10 }
```

Tudo será desalocado
após o `return`, e
recriado a cada chamada
da função.

Vetores locais

Vetores locais...

- ... precisam ter tamanho conhecido em tempo de compilação.
- ... não podem ser retornados por funções, porque ao final da função o vetor é desalocado.
- ... podem ser grandes demais para a *stack* do programa, causando *stack overflow*.

Todos estes problemas podem ser resolvidos usando *alocação dinâmica* de memória.

Alocação dinâmica?

Alocação dinâmica de memória

Alocação dinâmica: permite reservar o espaço que queremos explicitamente, em tempo de execução.

- É possível criar vetores de tamanhos que só são conhecidos em tempo de execução.
- É possível alocar uma quantidade de memória e depois alocar mais, caso necessário.
- O espaço pode ser desalocado quando não for mais necessário.

A memória é reservada em um espaço chamado *heap* (“monte”).

Alocação dinâmica de memória: passos

Como fazer uma alocação dinâmica? Em resumo:

- ① Declaramos uma variável que irá apontar para o *buffer* alocado.
- ② Chamamos uma função que aloca o espaço desejado.
- ③ Usamos o espaço como um vetor.
- ④ Chamamos uma função que libera o espaço.

Alocação dinâmica de memória: passo 1

Declaramos uma variável que irá apontar para o *buffer* alocado.

Exemplos:

```
int* foo; // Um vetor de int.  
float* bar; // Um vetor de float.  
char* foostring; // Uma string.
```

Alocação dinâmica de memória: passo 2

Chamamos uma função que aloca o espaço desejado. Ela faz parte da biblioteca-padrão `stdlib`.

Protótipo

```
void* malloc (size_t size);
```

`size_t size`¹: quantos *bytes* queremos reservar. A função aloca uma região contígua de memória.

O valor de retorno é o endereço do espaço reservado — um ponteiro para a primeira posição.

- O tipo é `void*` porque a função não sabe como o espaço reservado será interpretado.
- Se não houver espaço suficiente, retorna `NULL` (um ponteiro nulo, com endereço 0).

¹`size_t` é como um “apelido” para um tipo *unsigned*.

Alocação dinâmica de memória: passo 2

Exemplos:

Observe que é realizado um *casting* de `void*` para o tipo do vetor (neste caso, `int*`)

Usamos o comando `sizeof` para obter o tamanho em bytes de cada posição. Multiplicamos pela quantidade de posições desejadas para calcular o total de bytes que devem ser alocados.

```
1      (...)
2      int* foo; // Um vetor de int.
3      foo = (int*) malloc (sizeof (int) * 10);
4      if (foo == NULL)
5          // Erro.
6
7      (...)
8      float* bar; // Um vetor de float.
9      bar = (float*) malloc (n * sizeof (float));
10     if (!bar)
11         // Erro.
```

Alocação dinâmica de memória: passo 3

Usamos o espaço como um vetor.

Importante: o acesso é **idêntico** a um vetor qualquer! Também podemos passar para uma função.

Exemplos:

```
// Coloca 10 na 1a posição de foo.
```

```
foo [0] = 10;
```

```
// Coloca em x o conteúdo da posição i de bar.
```

```
x = bar [i];
```

```
// Passando para uma função.
```

```
// Protótipo: void func (int* v, int n);
```

```
func (vetor_alocado_dinamicamente, tam);
```

Alocação dinâmica de memória: passo 4

Chamamos uma função que libera o espaço. Ela faz parte da biblioteca-padrão `stdlib`.

Protótipo

```
void free (void* ptr);
```

`ptr` é o ponteiro para o endereço do espaço que alocamos.

Exemplo:

```
free (foo); // Desaloca o vetor foo.
```

Exemplo simples.

```
1  int main () {
2      float* vetor;
3      int tam, i;
4
5      scanf ("%d", &tam);
6      vetor = (float*) malloc (sizeof (float) * tam);
7
8      if (vetor == NULL)
9          return (1); /* ERRO. */
10
11     for (i = 0; i < tam; i++)
12         scanf ("%f", &(vetor [i]));
13
14     for (i = 0; i < tam; i++)
15         printf (-> "%f\n", vetor [i]);
16
17     free (vetor);
18
19     return (0);
20 }
```

Declaramos uma variável que irá apontar para o *buffer* alocado.

Chamamos uma função que aloca o espaço desejado.

Usamos o espaço como um vetor.

Chamamos uma função que libera o espaço.

Passando vetores alocados dinamicamente por parâmetro para funções

Como discutido no Passo 3, é exatamente da mesma forma que fazíamos com vetores estáticos!

Exemplo: alocando e passando vetores dinâmicos por parâmetro [1/2]

```
1  int main() {  
2  
3      int *vetor, n;  
4      scanf("%d", &n);  
5  
6      vetor = (int *) malloc (n * sizeof(int));  
7  
8      imprime(vetor, n);  
9      free(vetor);  
10  
11     return 0;  
12 }
```

Passamos como parâmetro o endereço da primeira posição de memória alocada (que está armazenada na “variável vetor”)

Exemplo: alocando e passando vetores dinâmicos por parâmetro [2/2]

Na função, podemos optar por utilizar o operador de indexação ou ponteiros...

```
1 void imprime(int vet[], int c){ //ou (int *vet, int n)
2     int i;
3     for (i=0; i<c; i++)
4         printf("%d ", vet[i]); //ou printf("%d ", *(vet+i));
5 }
```

Funções que retornam vetores

Exemplo: função que aloca e retorna um vetor (int)

Utilizando alocação dinâmica, conseguimos fazer com que uma função retorne um vetor!

Observe o tipo da função (ie., do retorno)!

```
1 int* alocaVetor (int n){  
2     int *v;  
3     v = (int*)malloc(n*sizeof(int));  
4     return v;  
5 }
```

Um ponteiro recebe o retorno da função
(a qual vai retornar o endereço da primeira
posição de memória que foi alocada)

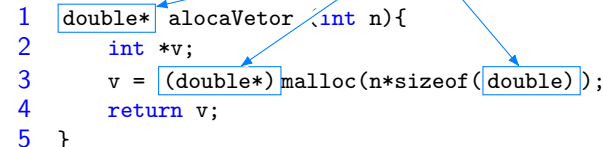
```
6  
7 int main(){  
8     int n, *v;  
9  
10    scanf("%d", &n);  
11    v = alocaVetor(n);  
12    //agora podemos acessar as n posições do vetor v  
13  
14    free(v); //não se esqueça do free  
15    return 0;  
16 }
```

Exemplo: função que aloca e retorna um vetor (double)

Atenção para a compatibilidade de tipos! Por exemplo:

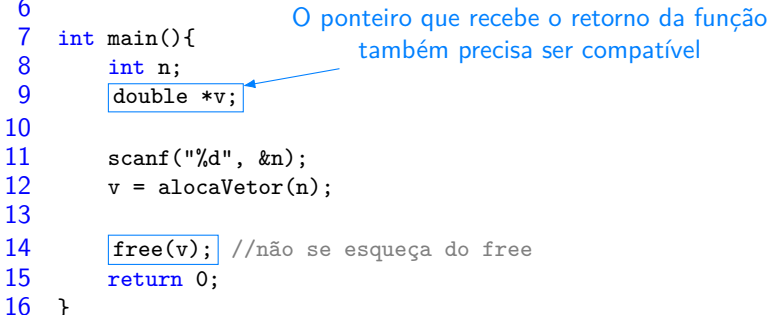
Observe o tipo da função e o malloc!

```
1 double* alocaVetor (int n){  
2     int *v;  
3     v = (double*) malloc(n*sizeof(double));  
4     return v;  
5 }
```



```
6  
7 int main(){  
8     int n;  
9     double *v;  
10  
11     scanf("%d", &n);  
12     v = alocaVetor(n);  
13  
14     free(v); //não se esqueça do free  
15     return 0;  
16 }
```

O ponteiro que recebe o retorno da função também precisa ser compatível



Exemplo: função que aloca e inicializa com zeros um vetor.

```
1  int* criaVetorZeroado (int tamanho)
2  {
3      int* vetor_interno;
4      int i;
5
6      vetor_interno = (int*) malloc (sizeof (int) * tamanho);
7      for (i = 0; i < tamanho; i++)
8          vetor_interno [i] = 0;
9
10     return (vetor_interno);
11 }
12
13 int main ()
14 {
15     int* vetor_do_main;
16
17     vetor_do_main = criaVetorZeroado (10);
18     free (vetor_do_main);
19
20     return (0);
21 }
```

Observe o tipo de retorno.

Não se esqueça do free.

Exemplo: função que copia apenas os pares

```
1  int* copiaPares (int input[], int n, int *contPares){
2
3      int *output, i;
4
5      output = (int*)malloc(n*sizeof(int));
6
7      *contPares = 0;
8      for(i=0; i<n; i++)
9          if(input[i]%2 == 0)
10             output[( *contPares )++] = input[i];
11
12     output = realloc(output, *contPares);
13
14     return output;
15 }
```

Copia elemento a elemento

É preciso saber a quantidade de pares, tanto para realocar o vetor output para o tamanho correto quanto para devolver essa informação para quem invocou a função!

Exemplo: retornando vetor (passando por referência).

```
1 void criaVetorZerado (int** vetor, int tamanho)
2 {
3     int* vetor_interno;
4     int i;
5
6     vetor_interno = (int*) malloc (sizeof (int) * tamanho);
7     for (i = 0; i < tamanho; i++)
8         vetor_interno [i] = 0;
9     *vetor = vetor_interno;
10 }
11
12 int main ()
13 {
14     int* vetor_do_main;
15
16     criaVetorZerado (&vetor_do_main, 10);
17     free (vetor_do_main);
18
19     return (0);
20 }
```

Um ponteiro para um ponteiro.

Acessa indiretamente a variável do main.

Passagem por referência.

Exemplo: trocando vetores.

```
1 float* vetor1;
2 float* vetor2;
3 float* vetor_aux;
4
5 vetor1 = (float*) malloc (sizeof (float) * tamanho);
6 vetor2 = (float*) malloc (sizeof (float) * tamanho);
7
8 // (...)
9 vetor_aux = vetor1;
10 vetor1 = vetor2;
11 vetor2 = vetor_aux;
12
13 free (vetor1);
14 free (vetor2);
```

Vazamento de memória

CUIDADO!!! Vazamento de memória!

Memory leak / vazamento de memória: é um dos bugs mais comuns e mais difíceis de detectar.

TODA memória alocada precisa ser liberada.

- Garanta que **sempre** um free será executado para cada vetor alocado!
- **Nunca** perca a referência para um vetor alocado!
- Ao fim do programa, toda memória alocada para o programa é desalocada automaticamente, mas **sempre** desaloque tudo explicitamente mesmo assim.

Quais as consequências de um vazamento de memória?

- O programa pode ir consumindo toda a memória disponível.
- O computador pode ir ficando cada vez mais lento.
- O programa (ou o computador) pode “travar”.

CUIDADO!!! Vazamento de memória!

```
1 // Aloca o vetor.
2 scanf ("%d", &tamanho);
3 um_vetor = (int*) malloc (sizeof (int) * tamanho);
4
5 // Faz alguma coisa...
6
7 // Aloca de novo.
8 scanf ("%d", &tamanho);
9 um_vetor = (int*) malloc (sizeof (int) * tamanho);
10
11 // Faz outra coisa...
12
13 // Libera.
14 free (um_vetor);
```

Só está liberando a memória da segunda alocação.

CUIDADO também com a localização do free!

Ao usar free a memória é desalocada! Qual o problema para o código abaixo?

```
1  int* alocaVetor (int n)
2  {
3      int *v;
4      v = (int*) malloc (n * sizeof (int));
5      free(v);
6      return v;
7  }
8
9  int main ()
10 {
11     int n, *v;
12
13     scanf("%d", &n);
14     v = alocaVetor (n);
15     //....
16
17     return 0;
18 }
```

Vai retornar um ponteiro para uma região que não está mais alocada!

Outras funções

Outras funções

```
void* calloc (size_t num, size_t size);
```

Aloca num posições, cada uma com size bytes, e “zera” o vetor alocado.

```
void* realloc (void* ptr, size_t size);
```

Realoca um vetor — talvez em outra região de memória, o que exige a cópia do conteúdo. A função faz isso sozinha, mas pode consumir tempo (ou seja, use com cautela).

A função calloc: exemplo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (){
5
6      int *v, tam;
7
8      scanf("%d", &tam),
9
10     v=(int *)calloc(tam, sizeof(int));
11     if (v == NULL)
12         printf ("** Erro: Memoria Insuficiente **");
13
14     return 0;
15 }
```

Observe que é realizado um casting de void* para o tipo do vetor (no caso do exemplo, int*)

Uso da função sizeof para calcular o tamanho em bytes de cada posição

A função realloc: exemplo

O exemplo abaixo copia apenas os pares em outro vetor.

```
1  #define N 5
2  int main(){
3      int i, contPares=0,
4          v1[N] = {1, 2, 3, 4, 5},
5          *v2;
6
7      v2 = (int*)malloc(N * sizeof(int));
8      for(i=0; i<N; i++){
9          if(v1[i]%2 == 0){
10             v2[contPares] = v1[i];
11             contPares++;
12         }
13
14         v2 = realloc(v2, contPares);
15         imprimeVetor(v2, contPares);
16
17         return 0;
18     }
```

Aloca inicialmente com
o tamanho total ...

... e depois realoca segundo
a quantidade de pares

Quais as implicações em declarar `int v[n]`?

Organização da memória do computador

A memória do computador na execução de um programa é organizada em quatro segmentos, os quais contêm:

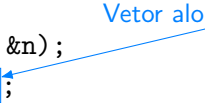
- Código executável: o código binário do programa
- Dados estáticos: variáveis globais e estáticas (que existem durante toda a execução do programa)
- Pilha: variáveis locais, criadas na execução de uma função (ao término da sua execução, elas são removidas da pilha)
- *Heap*: variáveis criadas por alocação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

Exemplo

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

```
1  #include <stdio.h>
2  int main()
3  {
4      int n, i;
5
6      scanf("%d", &n);
7      double v[n];
8
9      return 0;
10 }
```

Vetor alocado com tamanho n



Entretanto, os vetores criados desta forma tem memória alocada na pilha, que possui um limite máximo.

Exemplo

Por exemplo, para $n = 2000000$, o programa abaixo resulta em segmentation fault!

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

Vetor alocado com tamanho n

Erro ao retornar vetor!

Qual a inconsistência no código abaixo?

```
1  int* alocaVetor (int n){
2      int v[n];
3      return v;
4  }
5
6  int main(){
7      int n, *v;
8
9      scanf("%d", &n);
10     v = alocaVetor(n);
11     return 0;
12 }
```

Para pensar: quando a memória reservada aqui será desalocada?

Portanto, quando precisar alocar memória para um vetor dentro de uma função (cujo tamanho só é definido em tempo de execução), use `malloc()/calloc()`!

Não há diferença de desempenho entre acessar um vetor alocado dinamicamente e outro alocado automaticamente...

... mas alocar memória muitas e muitas vezes (por ex, dentro de um *loop*) pode prejudicar o desempenho do programa.

Em certas situações vale mais a pena alocar um vetor grande e usar apenas parte dele, como vínhamos fazendo, mas por enquanto, esta é uma preocupação secundária!