```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Alocação Dinâmica

Revisão: Vetores (arranjos, *arrays*)

```
tipo nome[capacidade];
tipo nome[] = {e1, ..., eN};
```

Reserva bloco contíguo de memória (volátil), suficientes para alocar dado número de cópias de um tipo de dado nativo (`int`, `char`, ...)

- `capacidade` é constante do tipo `unsigned int`
- `nome` é ponteiro para primeira variável da sequência


```c
int i;
int u[] = {1, 2, 3, 4}; /* vetor de inteiros */
char *v[5] = "MAC\n"; /* string */
```

Revisão: Memória Volátil

- É abstraída como uma sequência de bytes
- Cada byte é acessado através de seu endereço
- Endereço de variável contém byte inicial (acessado por `&`)

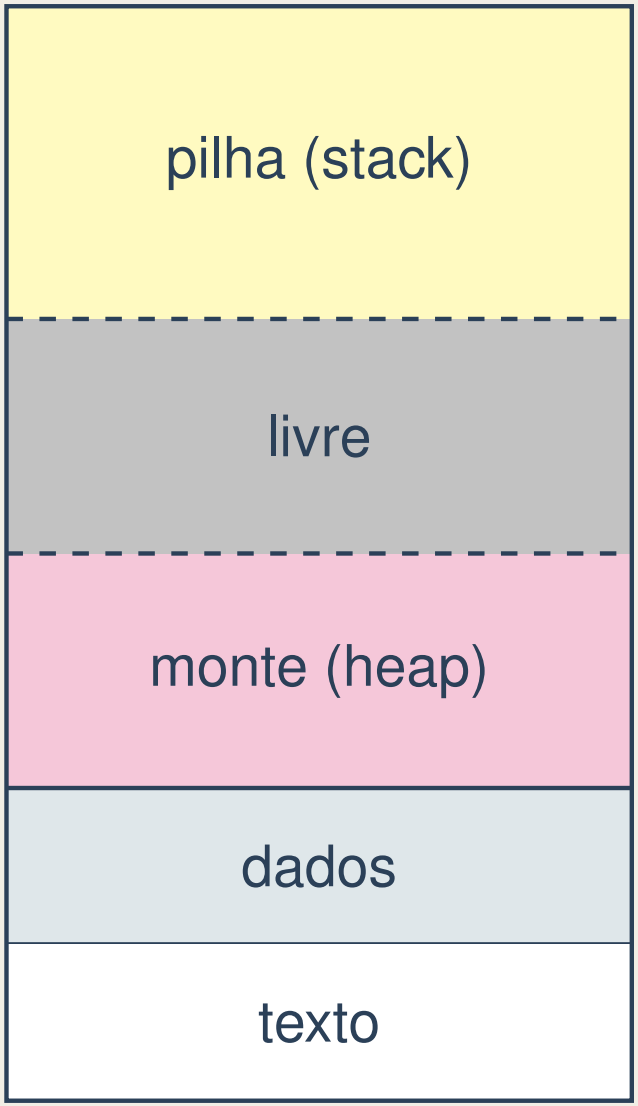
```
int a = 2;
char c;
struct { int x,y } ponto;
int v[2] = {-2, -4};
int *p = &v;
```

variável	endereço
`a`	89420
`c`	89424
`ponto`	89425
`v[0]`	89433
`v[1]`	89437
`p`	89441

`&v = 89433; &p = 89441; p = 89433; *p = -2`

Organização da memória para um programa em C

- Texto: instruções do programa
- Dados e Pilha (Stack): variáveis globais, estáticas e locais (local e tamanho não muda durante execução)
- Monte (Heap): variáveis alocadas dinamicamente; local e tamanho podem mudar durante execução (aula de hoje)

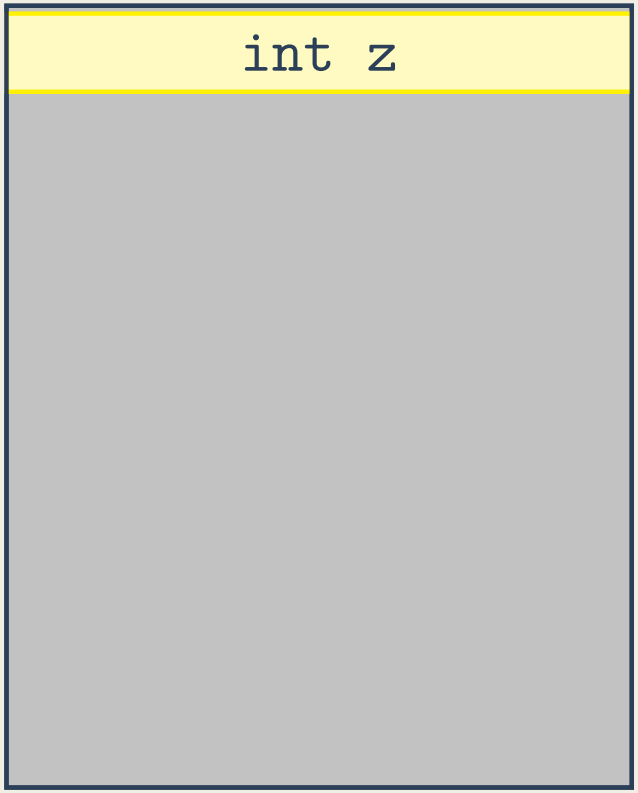


Alocação estática de memória

`int n; char c, int v[30], char str[100];`

Aloca memória na pilha do programa (stack) para cada variável antes do começo do programa ou função; variáveis locais são *automaticamente desalocadas* quando chamada da função termina

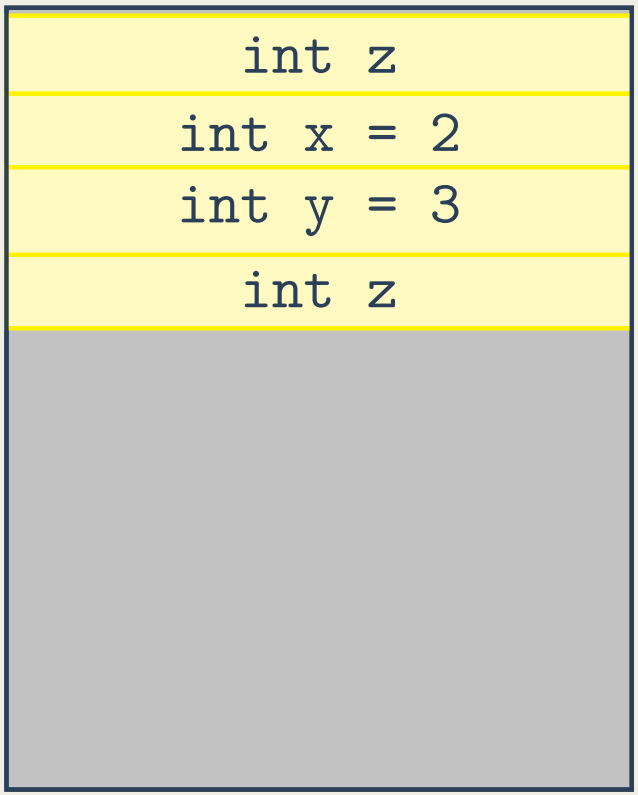
```
1  int soma(int x, int y) {
2    int z = x + y;
3    return z;
4  }
5
6  int main() {
7    int z; ⟸
8    z = soma(2, 3);
9    return 0;
10 }
```



`int n; char c, int v[30], char str[100];`

Aloca memória na pilha do programa (stack) para cada variável antes do começo do programa ou função; variáveis locais são *automaticamente desalocadas* quando chamada da função termina

```
1  int soma(int x, int y) {  ⟸
2    int z = x + y;
3    return z;
4  }
5
6  int main() {
7    int z;
8    z = soma(2, 3);  ⟸
9    return 0;
10 }
```

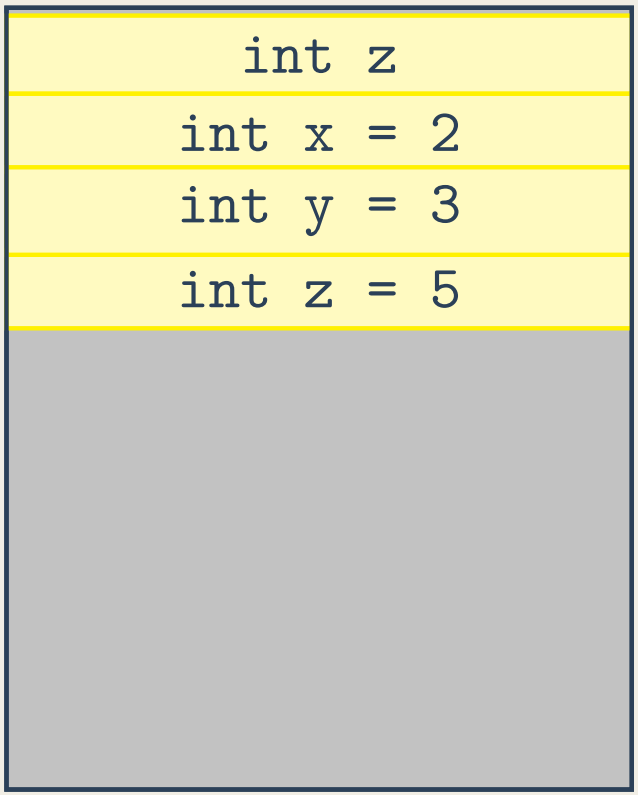


Alocação estática de memória

`int n; char c, int v[30], char str[100];`

Aloca memória na pilha do programa (stack) para cada variável antes do começo do programa ou função; variáveis locais são *automaticamente desalocadas* quando chamada da função termina

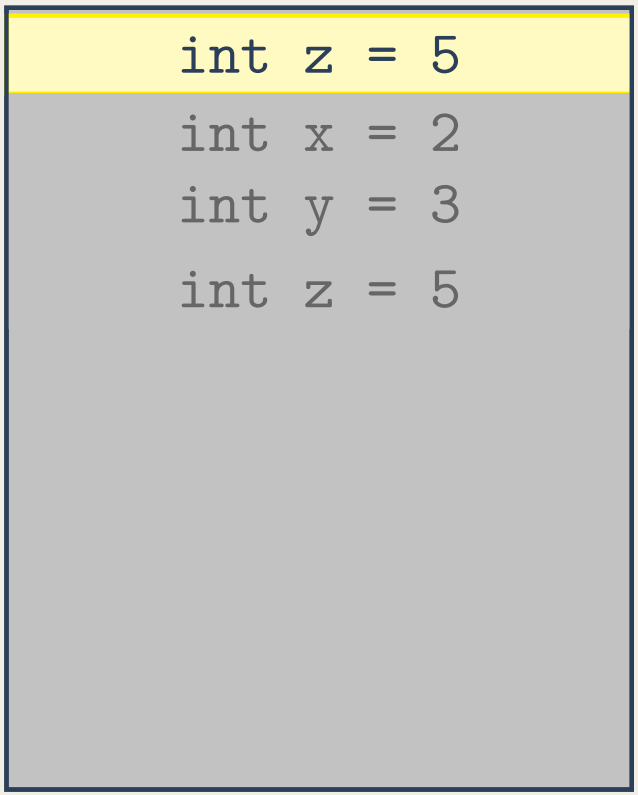
```
1  int soma(int x, int y) {
2    int z = x + y; ⟸
3    return z;
4  }
5
6  int main() {
7    int z;
8    z = soma(2, 3); ⟸
9    return 0;
10 }
```



`int n; char c, int v[30], char str[100];`

Aloca memória na pilha do programa (stack) para cada variável antes do começo do programa ou função; variáveis locais são *automaticamente desalocadas* quando chamada da função termina

```
1  int soma(int x, int y) {
2    int z = x + y;
3    return z; ⟸
4  }
5
6  int main() {
7    int z;
8    z = soma(2, 3); ⟸
9    return 0;
10 }
```



Alocação estática de memória

`int n; char c, int v[30], char str[100];`

Aloca memória na pilha do programa (stack) para cada variável antes do começo do programa ou função; variáveis locais são *automaticamente desalocadas* quando chamada da função termina

```
1  int soma(int x, int y) {
2    int z = x + y;
3    return z;
4  }
5
6  int main() {
7    int z;
8    z = soma(2, 3);
9    return 0; ⟸
10 }
```

```
int z = 5
int x = 2
int y = 3
int z = 5
```

`int n; char c, int v[30], char str[100];`

Aloca memória para cada variável antes do começo do programa ou função; tamanho a ser alocado é determinado no código-fonte

```
1 /* Criar string s de tamanho n definido por usuario */
2 int n;
3 scanf ("Quantas letras? %d", &n);
4 char s[n+1]; /* erro de compilação */
```

Solução simplista: definir vetor com `char s[CAP]` capacidade suficiente

- Uso ineficiente de memória
- Não permite encapsulamento de tipos de dados de usuário

Alocação estática (na pilha do programa/stack)

```c
/* vetor.c */
int* cria_vetor(int valor, int tamanho)
{
    int vetor[100];
    for (int i=0; i < tamanho; i++) vetor[i] = valor;
    return vetor;
}
int main() {

 int *v1 = cria_vetor(1,10);
 int *v2 = cria_vetor(-1,5); /* v2 = v1 */

 /*exibe   -1 -1 -1 -1 -1  1  1  1  1  1 */
 for (int i=0; i < 10; i++) printf(" %d", v1[i]);
 printf("\n");

 return 0;
}
```

`void *malloc (unsigned int n);`

Aloca um bloco de `n` *bytes* consecutivos e retorna um ponteiro para o espaço alocado ou `NULL` em caso de erro

`void free (void *ptr);`

Libera o espaço de memória endereçado por `ptr`

Definidos na biblioteca `stdlib.h`

```
1 /* criar/ler string de tamanho n definido por usuario */
2 int n; char *s;
3 scanf ("%d", &n);
4 s = (char *) malloc (n + 1);  /* n caracteres + '\0' */
5                 /* valor de string s e' ainda indefinido */
6 fgets (s, n, stdin) ;
7 printf ("%s", s);
8 free (s);
```

Use `sizeof` para determinar o número de *bytes* necessários para alocar um tipo diferente de `char`

```
1 /* criar vetor de tamanho n definido por usuario  */
2 int n, *v;
3 scanf ("%d", &n);
4 v = malloc (n * sizeof (int));          /* n ints */
5                 /* valor de v[0..n-1] e' indefinido */
6 free (v);
```

Vetores criados dessa forma são chamados vetores dinâmicos

Alocação estática

```
1  /* vetor2.c */
2  int* cria_vetor(int valor, int tamanho) {
3     int *vetor = malloc(tamanho*sizeof(int));
4     if (vetor != NULL) for (int i=0; i < tamanho; i++) vetor
       [i] = valor;
5     return vetor;
6  }
7  int main() {
8   int *v1 = cria_vetor(1,10);
9   int *v2 = cria_vetor(-1,5); /* v2 != v1 */
10  /*exibe   1 1 1 1 1  1  1  1  1  1 */
11  for (int i=0; i < 10; i++) printf(" %d", v1[i]);
12  printf("\n");
13
14  free(v1); free(v2);
15
16  return 0;
17 }
```

Matrizes estáticas:

```
1 /* alocar vetor de vetores de ints */
2 #define numLin 10
3 #define numCol 20
4 int matriz[numLin][numCol];
5
6 for (i = 0; i < numLin; i++)
7   for (j = 0; j < numCol; j++)
8       matriz[i][j] = numCol*i + j;
```

Matrizes dinâmicas:

```
1  /* matriz.c */
2  /* cria vetor de vetores de ints */
3  int **matriz;
4  int numLin, numCol;
5  scanf("%d", &numLin); scanf("%d", &numCol);
6  /* alocar vetor de ponteiros de ints para cada coluna */
7  matriz = malloc (numLin * sizeof (int *));
8  for (i = 0; i < numLin; i++)
9     /* alocar vetor de ints para cada linha */
10    matriz[i] = malloc (numCol * sizeof (int));
11
12 for (i = 0; i < numLin; i++)
13   for (j = 0; j < numCol; j++)
14      matriz[i][j] = numCol*i + j;
```

Matrizes dinâmicas:

```
1 /* matriz.c */
2 /* desalocar linhas */
3 for (i = 0; i < numLin; i++)
4     free (matriz[i]);
5 /* desalocar matriz */
6 free (matriz);
```

Alocação estática de registros

```
1 typedef struct { int num, den; } racional;
2 racional a; /* registro racional */
3 a.num = 1; b.den = 3; /* a = 1/3 */
```

ALOCAÇÃO DINÂMICA DE REGISTROS

```
1 typedef struct { int num, den; } racional;
2 racional *a; /* ponteiro para racional */
3 a = malloc (sizeof (racional)); /* aloca 1 racional */
4                                 /* valor de a e' indefinido */
5 a->num = 1; b->den = 3;
6                                                 /* a = 1/3 */
7 free (a);  /* libera memória */
```

Vetor estático de registros alocados dinâmicamente:

```
1  typedef struct { int num, den; } racional;
2  racional *a[10]; /* vetor de ponteiros para racional */
3                   /* (alocação estática) */
4  for (i = 0; i < 10; i++)
5   a[i] = malloc (sizeof (racional));/* aloca racional */
6                                    /* alocação dinâmica */
7                /* valores de a[0..9] são indefinidos */
8  for (i = 0; i < 10; i++)
9   free (a[i]);/* libera memória alocada dinamicamente */
10 /* vetor de ponteiros para racional continua alocado */
```

Vetor dinâmico de registros dinâmicos:

```
1 typedef struct { int num, den; } racional;
2 racional **a;  /* ponteiro para ponteiro de racionais */
3 a = malloc (10 * sizeof (racional*));  /* 10 pont. rac. */
4 for (i = 0; i < 10; i++)
5    a[i] = malloc (sizeof (racional));    /* 1 racional */
6
7 for (i = 0; i < 10; i++)
8    free (a[i]);/* libera memoria alocada para racional */
9 free (a);      /* libera memoria de vetor de ponteiros */
```

Alocação dinâmica de memória

- Programa usa espaço de memória chamado heap para alocar novas variáveis
- Quando o heap está cheio, `malloc` retorna ponteiro `NULL`
- Deve-se sempre testar se alocação foi bem sucedida

```
1 void *p;
2 p = malloc (10); /* tenta alocar 10 bytes */
3 if (p == NULL) printf("Erro ao alocar memoria!");
```

`void *calloc (unsigned int n, unsigned int size);`
Aloca um bloco de `n` *bytes* consecutivos e retorna um ponteiro para o espaço alocado ou `NULL` em caso de erro; preenche o espaço alocado de zeros

```
1 /* criar vetor de tamanho n definido por usuario */
2 int n, *v;
3 scanf ("%d", &n);
4 v = calloc (n, sizeof (int)); /* n int */
5 /* v e' um vetor de n zeros */
6 free (v);
```

- Menos eficiente que `malloc`
- Mais eficiente que `malloc` seguido de laço que zera valores

Tipo de dado: vetor dinâmico

Objetivo: Permitir que capacidade de vetor varie conforme uso

```c
/* vetor3.c */

/* Capacidade inicial do vetor */
#define CAP0 1024

/* Estrutura de dados para vetor */
struct vetor_st {
  int *dados; /* ponteiro para vetor */
  int tamanho; /* num. elementos */
  int cap; /* qtd. de memória alocada */
};

/* tipo de dado definido por usuário */
typedef struct vetor_st *vetor;
```

Tipo de dado: vetor dinâmico

Objetivo: Permitir que capacidade de vetor varie conforme uso

- Tamanho do vetor é limitado apenas pela quantidade de memória disponível ao programa
- Capacidade é alterada para uso eficiente:
  - Se vetor estiver cheio, dobre capacidade
  - Se vetor estiver 3/4 vazio, meie capacidade
- Implementação é escondida de usuário/cliente

Vetor dinâmico: Interface

```c
/* vetor_t.h */

/* Capacidade inicial do vetor */
#define CAP0 1024

/* Estrutura de dados para vetor */
struct vetor_st {
  int *dados; /* ponteiro para vetor */
  int tamanho; /* num. elementos */
  int cap; /* qtd. de memória alocada */
};

/* tipo de dado definido por usuário */
typedef struct vetor_st *vetor;

/* (continua...) */
```

Vetor dinâmico: Interface

```
1  /* (cont.) vetor_t.h */
2
3  /* Cria vetor vazio: Aloca espaço */
4  vetor CriaVetor (void);
5  /* Devolve tamanho do vetor */
6  int Tamanho (vetor v);
7  /* Destroi vetor: Libera espaço */
8  void DestroiVetor (vetor v);
9  /* Insere x no final; dobra capacidade se cheio. */
10 void  InsereNoFim (vetor v, int x)
11 /* Remove e devolve último elemento; meia capacidade se 3/4
       vazio */
12 int  RemoveDoFim (vetor v);
```

Vetor dinâmico: Implementação

```c
/* Cria vetor vazio: Aloca espaço */
vetor CriaVetor (void) {
  vetor v = malloc (sizeof(struct vetor_st));
  v->dados = malloc (sizeof(int)*CAP0);
  v->tamanho = 0;
  v->cap = CAP0;
  return v;
}
/* Devolve tamanho do vetor */
int Tamanho (vetor v) {
  return v->tamanho;
}
/* Destrói vetor: Libera espaço */
void DestroiVetor (vetor v) {
  free (v->dados);
  free (v);
}
```

Vetor dinâmico: Implementação

```
1  /* Insere x no final; dobra capacidade se cheio. */
2  void  InsereNoFim (vetor v, int x) {
3    if (v->tamanho == v->cap) {
4      int *v2; /* cria novo vetor com dobro de capacidade */
5      v2 = malloc ((v->cap * 2) * sizeof(int));
6      /* copia itens de vetor de dados atual */
7      for (int i = 0; i < v->cap; i++)  v2[i] = v->dados[i];
8      /* libera memória */
9      free (v->dados);
10     /* redireciona ponteiro */
11     v->dados = v2;
12     v->cap = v->cap * 2; /* nova capacidade */
13   } /* aumenta tamanho e insere elemento */
14   v->dados[v->tamanho++] = x;
15 }
```

Vetor dinâmico: Implementação

```
1  /* Remove e devolve último final; meia capacidade se 3/4
       vazio */
2  int  RemoveDoFim (vetor v) {
3    int x;
4    if (!v->tamanho) return 0; /* nada para remover */
5    x = v->dados[--(v->tamanho)];
6    if (v->tamanho < v->cap/4) { /* liberar espaço */
7      int *v2;
8      v2 = malloc ((v->cap / 2) * sizeof(int));
9      for (int i = 0; i < v->tamanho; i++)
10           v2[i] = v->dados[i];
11     free (v->dados);
12     v->dados = v2;
13     v->cap = v->cap / 2;
14   }
15   return x;
16 }
```

```
void *realloc (void *ptr, unsigned int n);
```

Realoca (altera tamanho de) um bloco de `n` *bytes* consecutivos previamente alocados; retorna um ponteiro para o espaço alocado.
O conteúdo do bloco é inalterado, dentro do novo tamanho.
Se não conseguir alocar mais espaço, o espaço é deixado inalterado e `NULL` é devolvido.

Definido na biblioteca `stdlib.h`

```
1  /* realloc.c */
2    int n = 1024; char *s;
3    /* Aloca memória suficiente */
4    s = (char *) malloc (n + 1);  /* n caracteres + '\0' */
5    /* le string s (tamanho pode ser menor que n-1 */
6    printf("String: "); scanf("%s", s);
7    /* reduz para somente o necessário */
8    n = strlen(s);  s = realloc(s,n+1);
9    /* String é a mesma */
10   printf("String: %s", s);
```

Vetor dinâmico: Implementação

```c
/* Insere x no final; dobra capacidade se cheio. */
void  InsereNoFim (vetor v, int x) {
  if (v->tamanho == v->cap) {
    /* dobra capacidade */
    v->cap = v->cap * 2; /* nova capacidade */
    v->dados = realloc (v->dados, v->cap*sizeof(int));
  } /* aumenta tamanho e insere elemento */
  v->dados[v->tamanho++] = x;
}
```

Vetor dinâmico: Implementação

```
1  /* Remove e devolve último final; meia capacidade se 3/4
       vazio */
2  int  RemoveDoFim (vetor v) {
3    int x;
4    if (!v->tamanho) return 0; /* nada para remover */
5    x = v->dados[--(v->tamanho)];
6    if (v->tamanho < v->cap/4) { /* liberar espaço */
7  ^^I  /* reduz capacidade */
8      v->cap = v->cap / 2;
9      v->dados = realloc (v->dados, v->cap*sizeof(int));
10   }
11   return x;
12 }
```

Vetor dinâmico: Exemplo de uso

```c
/* usa_vetor_t2.c
Exibe lista de números em ordem reversa
Compile com gcc -std=c99 -o uso usa_vetor_t2.c vetor_t.c
*/
#include "vetor_t.h"
int main() {
    vetor v = CriaVetor(); // cria vetor vazio
    int x, n = 10; // Insere 10 num. aleatórios em [0,9]
    for (i=0; i < n; i++) {
        printf("Numero? "); scanf("%d", &x);
        InsereNoFim(v,x);
    }
    n = Tamanho(v);
    while (n > 0) { /* remove elementos */
      x = RemoveDoFim(v); n = Tamanho(v);
    }
    DestroiVetor(v); // libera memória
    return 0;
}
```

Para casa

Exercícios: 6A-6C