Vetores

Implementação de um contêiner `vector` em C

Prof. Edson Alves - UnB/FGA

2018

Sumário

1. Definição do contêiner

2. Operações de inserção

3. Operações de remoção

Definição do contêiner

Definição das operações

- O primeiro passo a ser feito é definir quais as operações que serão suportadas pelo tipo abstrado de dados
- Por motivos didáticos, serão implementadas apenas as operações listadas no próximo *slide*
- Uma implementação real teria um número muito maior de operações e variantes delas
- A página do Cppman para `vector` contém 33 operações, sem contar as variantes de cada operação
- Também por motivos didáticos, o contêiner a ser implementado armazenará apenas elementos do tipo inteiro
- Uma implementação que permitiria armazenar elementos de qualquer tipo é mais complexa e será deixada como exercício

Operações a serem implementadas

Operação	Complexidade	Descrição
`create(N)`	$O(1)$	Cria uma nova instância com capacidade para `N` elementos
`free(v)`	$O(1)$	Desaloca a memória utilizada pela instância `v`
`element_at(v, i)`	$O(1)$	Retorna o valor do elemento de `v` armazenado na posição `i`
`size(v)`	$O(1)$	Retorna o número de elementos armazenados em `v`
`push(v, x, i)`	$O(N)$	Insere o valor `x` na posição `i` de `v`
`push_back(v, x)`	$O(1)$	Anexa o valor `x` ao final `v`
`pop(v, i)`	$O(N)$	Remove o elemento que ocupa a posição `i` de `v`
`pop_back(v)`	$O(1)$	Remove o último elemento de `v`
`clear(v)`	$O(1)$	Remove todos os elementos de `v`

Cabeçalho vector.h

```c
#ifndef C_VECTOR_H
#define C_VECTOR_H

typedef struct _vector {
    int *data;
    int size;
    int capacity;
} vector;

extern vector * create_vector(int capacity);
extern void free_vector(vector *v);
extern int element_at(vector *v, int index);
extern int size(vector *v);

extern int push(vector *v, int value, int index);
extern int push_back(vector *v, int value);
extern int pop(vector *v, int index);
extern int pop_back(vector *v);
extern void clear(vector *v);

#endif
```

Funções de criação e destruição

- Para criar uma instância da estrutura, são necessários dois passos: alocação de memória para a estrutura em si e alocação de memória para os elementos armazenados
- Uma vez alocado o espaço para a estrutura, os campos devem ser inicializados corretamente antes do retorno
- Como o único parâmetro da função `create_vector()` é a capacidade, o campo `size` inicialmente será igual a 0 (zero)
- Um erro na criação será reportado através do retorno de um ponteiro nulo
- Na função `free_vector()` é necessário desalocar tanto a memória para a estrutura quanto a memória para os elementos armazenados nela

Implementação da função create_vector()

```c
#include <stdlib.h>
#include "vector.h"

vector * create_vector(int capacity)
{
    if (capacity < 1) return NULL;

    vector *v = (vector *) calloc(1, sizeof(vector));

    if (!v) return NULL;

    v->capacity = capacity;
    v->data = (int *) malloc(capacity * sizeof(int));

    if (!v->data) {
        free(v);
        return NULL;
    }

    return v;
}
```

Implementação da função free_vector()

```
23 void free_vector(vector *v)
24 {
25     if (!v)
26         return;
27
28     if (v->data)
29         free(v->data);
30
31     free(v);
32 }
```

Operações de acesso

- As estruturas de C não tem suporte aos conceitos de interfaces públicas e privadas
- Deste modo, é possível acessar tanto os elementos quanto o tamanho da estrutura diretamente a partir dos seus campos
- Contudo, este acesso viola dois conceitos importantes
- Primeiramente, uma ADT é definida por suas operações, e não por sua implementação. O acesso direto à implementação viola o conceito de ADT
- Em segundo lugar, a modificação destes campos diretamente pode comprometer o estado da estrutura, inclusive podendo corrompê-la
- Para evitar este acesso direto seria necessário definir a estrutura como um ponteiro `void *`
- Na implementação seria necessária a coerção para a estrutura correta, a qual seria definida apenas no arquivo de implementação
- A esta técnica chamamos de *private implementation (PIMPL)*

Interface com a técnica do PIMPL

```c
#ifndef C_VECTOR_ADT_H
#define C_VECTOR_ADT_H

typedef void * vector;

extern vector create_vector(int capacity);
extern void free_vector(vector v);
extern int element_at(const vector v, int index);
extern int size(const vector v);

extern int push(vector v, int value, int index);
extern int push_back(vector v, int value);
extern int pop(vector v, int index);
extern int pop_back(vector v);
extern int clear(vector v);

#endif
```

Implementação com a técnica do handler

```c
#include <stdlib.h>
#include "vector_adt.h"

typedef struct _vector {
    int *data;
    int size;
    int capacity;
} pvector;

vector create_vector(int capacity)
{
    if (capacity < 1) return NULL;

    pvector *pv = (pvector *) calloc(1, sizeof(pvector));

    if (!pv) return NULL;

    pv->capacity = capacity;
    pv->data = (int *) malloc(capacity * sizeof(int));

```

Implementação com a técnica do handler

```
21      if (!pv->data) {
22          free(pv);
23          return NULL;
24      }
25
26      return (vector) pv;
27  }
28
29  void free_vector(vector v)
30  {
31      pvector *pv = (pvector *) v;
32
33      if (!pv)
34          return;
35
36      if (pv->data)
37          free(pv->data);
38
39      free(pv);
40  }
```

Funções de acesso

- As funções de acesso tem sempre como parâmetro um ponteiro do tipo `vector`
- Deste modo, a primeira etapa da implementação é sempre checar se este ponteiro é nulo ou não
- Isto é necessário porque tentar acessar um campo a partir de um ponteiro nulo gerar um erro de segmentação
- Além disso, há outro erro a ser considerado: o índice passado por estar fora do intervalo $[0, N-1]$
- Uma forma de reportar o erro ao usuário da função é retornar um código erro númerico
- Veja que esta técnica funciona bem na função `size()`, mas pode ser ambígua no caso da função `element_at()`, se o vetor puder armazenar valores inteiros negativos

Códigos de erros

```
1  #ifndef VECTOR_ERRORS_H
2  #define VECTOR_ERRORS_H
3
4  #define VECTOR_OK                          0
5  #define VECTOR_ERROR_OUT_OF_MEMORY        -1
6  #define VECTOR_ERROR_NULL_PARAMETER       -2
7  #define VECTOR_ERROR_INDEX_OUT_OF_BOUNDS  -3
8  #define VECTOR_ERROR_EMPTY_VECTOR         -4
9
10 #endif
```

Implementação das funções de acesso

```
42 #include "vector_errors.h"
43
44 int element_at(const vector v, int index)
45 {
46     const pvector *pv = (const pvector *) v;
47
48     if (!pv)
49         return VECTOR_ERROR_NULL_PARAMETER;
50
51     if (index < 0 || index >= pv->size)
52         return VECTOR_ERROR_INDEX_OUT_OF_BOUNDS;
53
54     return pv->data[index];
55 }
56
57 int size(const vector v)
58 {
59     const pvector *pv = (const pvector *) v;
60
61     return pv ? pv->size : VECTOR_ERROR_NULL_PARAMETER;
62 }
```

Operações de inserção

Anexar ao final do vetor

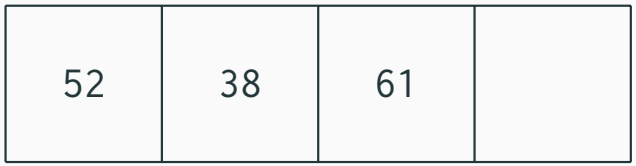
- Dentre as duas operações de inserção definidas, a operação push_back() é a que tem menor complexidade assintótica
- Se ainda houver espaço disponível no vetor, basta inserir o valor indicado na posição correspondente e atualizar o valor do campo `size`
- Se o vetor estiver cheio (isto é, se o tamanho for igual a capacidade), há duas estratégias possível
- A mais simples, e que mantém o pior caso em $O(1)$, é retornar um código de erro
- A segunda seria ampliar a capacidade do vetor para comportar novos elementos
- A complexidade do pior caso seria então $O(N)$, mas o custo amortizado seria $O(1)$

Visualização da inserção

Elemento a ser inserido: 14

Capacidade: 4

Tamanho: 3

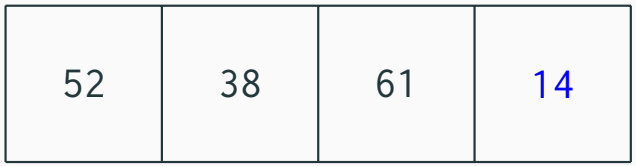


Visualização da inserção

Elemento a ser inserido: 14

Capacidade: 4

Tamanho: 4

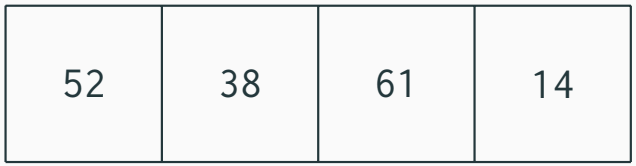


Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 4

Tamanho: 4

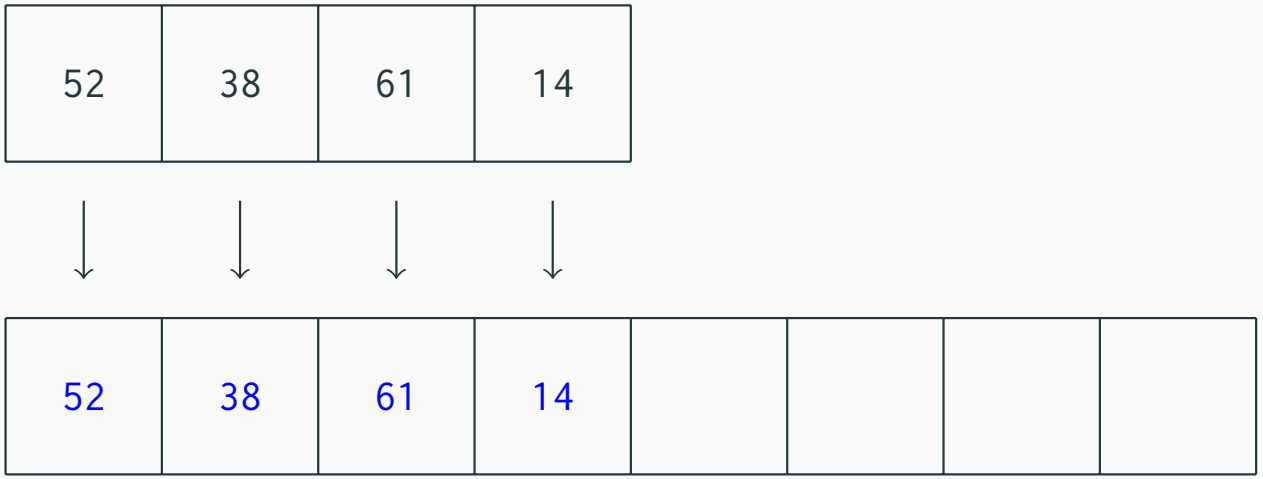


Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 4

Tamanho: 4



Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 8

Tamanho: 4

Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 8

Tamanho: 5

Implementação da função push_back()

```
64 int push_back(vector v, int value) {
65     pvector *pv = (pvector *) v;
66     int i, *p;
67
68     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
69
70     if (pv->size == pv->capacity) {
71         if (!(p = (int *) malloc(2 * pv->capacity * sizeof(int))))
72             return VECTOR_ERROR_OUT_OF_MEMORY;
73
74         for (i = 0; i < pv->size; ++i)
75             p[i] = pv->data[i];
76
77         free(pv->data);
78         pv->data = p;
79         pv->capacity *= 2;
80     }
81
82     pv->data[pv->size++] = value;
83     return VECTOR_OK;
84 }
```

Inserção em posição arbitrária

- A inserção em uma posição arbitrária `i` do vetor em complexidade $O(N)$
- Assim como a inserção ao final do vetor, pode ser necessário ampliar a capacidade do vetor caso este esteja cheio
- Todos os elementos, a partir da posição `i` em diante, devem ser deslocados uma posição para a direita
- Uma vez realizado o deslocamento, o valor indicado pode ser escrito na posição `i`
- O tamanho do vetor também deve ser atualizado

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 9

Implementação da função push()

```
86  int push(vector v, int value, int index) {
87      pvector *pv = (pvector *) v;
88      int i;
89
90      if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
91
92      if (index < 0 || index >= pv->size)
93          return VECTOR_ERROR_INDEX_OUT_OF_BOUNDS;
94
95      if (pv->size == pv->capacity) {
96          if ((i = push_back(v, value)) != VECTOR_OK) return i;
97          pv->size--;
98      }
99
100     for (i = pv->size; i >= index; --i)
101         pv->data[i] = pv->data[i - 1];
102
103     pv->data[index] = value;
104     pv->size++;
105     return VECTOR_OK;
106 }
```

Exemplo de inserções

```c
#include <stdio.h>
#include "vector_adt.h"
#include "vector_errors.h"

int main()
{
    vector v = create_vector(10);
    int N, rc;

    if (!v) {
        fprintf(stderr, "Falha na criação do vetor!\n");
        return -1;
    }

    while (1) {
        printf("Informe o número de passageiros na viagem (Ctrl-D para"
            " encerrar): ");

        if (scanf("%d", &N) < 1)
            break;

```

Exemplo de inserções

```
22          rc = push_back(v, N);
23
24          if (rc != VECTOR_OK)
25          {
26              fprintf(stderr, "Erro na inserção de passageiros: %d\n", rc);
27              free_vector(v);
28              return rc;
29          }
30      }
31
32      printf("\n\nRelatório de passageiros a cada viagem: ");
33
34      for (int i = 0; i < size(v); ++i)
35          printf("%d%c", element_at(v, i), " \n"[i + 1 == size(v)]);
36
37      free_vector(v);
38
39      return 0;
40  }
```

Operações de remoção

Remoção do final e limpeza

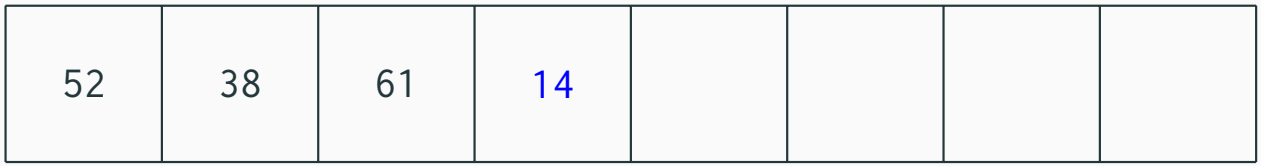
- A remoção do final do vetor tem complexidade $O(1)$
- O procedimento é simples: basta decrementar o campo `size`
- Observe que não é necessário modificar ou alterar o valor do último elemento
- Embora o valor permaneça inalterado na memória, ele se torna inacessível
- A função `clear()` é semelhante, com a única diferença que o campo `size` é zerado

Visualização da remoção ao final

Elemento a ser removido: 14

Capacidade: 8

Tamanho: 4



Visualização da remoção ao final

Elemento a ser removido: 14

Capacidade: 8

Tamanho: 3

Implementação das funções pop_back() e clear()

```
08 int pop_back(vector v) {
09     pvector *pv = (pvector *) v;
10
11     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
12
13     if (pv->size == 0) return VECTOR_ERROR_EMPTY_VECTOR;
14
15     pv->size--;
16
17     return VECTOR_OK;
18 }
19
20 int clear(vector v) {
21     pvector *pv = (pvector *) v;
22
23     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
24
25     pv->size = 0;
26
27     return VECTOR_OK;
28 }
```

Remoção em posição arbitrária

- Assim como na inserção, a remoção em posição arbitrária tem complexidade $O(N)$
- Novamente o deslocamento é necessário, mas em sentido oposto
- Os elementos à direita da posição indicada devem ser deslocados uma posição para a esquerda
- O campo `size` deve ser atualizado
- Constitui um erro remover um elemento de uma posição inválida ou de um vetor vazio

Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

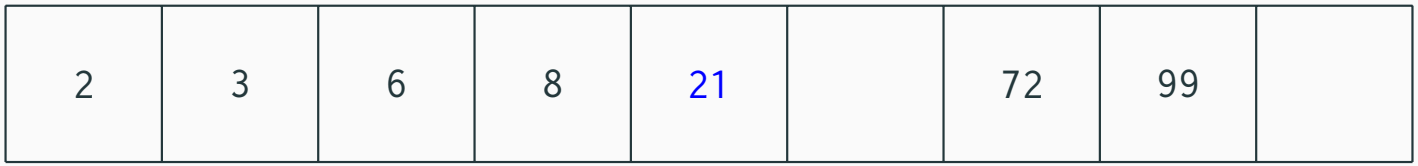
Tamanho: 8

Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

Tamanho: 8



Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

Tamanho: 8

Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

Tamanho: 7

2	3	6	8	21	72	99		

Implementação da função pop

```
30 int pop(vector v, int index)
31 {
32     pvector *pv = (pvector *) v;
33     int i;
34
35     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
36
37     if (pv->size == 0) return VECTOR_ERROR_EMPTY_VECTOR;
38
39     if (index < 0 || index >= pv->size)
40         return VECTOR_ERROR_INDEX_OUT_OF_BOUNDS;
41
42     for (i = index + 1; i < pv->size; ++i)
43         pv->data[i - 1] = pv->data[i];
44
45     --pv->size;
46
47     return VECTOR_OK;
48 }
```

Exemplo de remoção em posição arbitrária

```c
#include <stdio.h>
#include "vector_adt.h"
#include "vector_errors.h"

void inserir(vector v) {
    int N, rc;

    printf("Insira o número de passageiros da viagem %d: ", size(v) + 1);

    if (scanf("%d", &N) < 1) {
        fprintf(stderr, "Quantidade inválida!\n");
        return;
    }

    rc = push_back(v, N);

    if (rc == VECTOR_OK)
        printf("Informação inserida com sucesso!\n");
    else
        fprintf(stderr, "Erro na inserção: %d\n", rc);
}
```

Exemplo de remoção em posição arbitrária

```
23 void remover(vector v) {
24     int N, rc;
25
26     printf("Insira o número da viagem a ser removida: ");
27
28     if (scanf("%d", &N) < 1 || N < 1 || N > size(v))
29         fprintf(stderr, "Número inválido!\n");
30     else if ((rc = pop(v, N - 1)) == VECTOR_OK)
31         printf("Remoção bem sucedida!\n");
32     else
33         fprintf(stderr, "Erro na remoção: %d\n", rc);
34 }
35
36 void imprimir(vector v) {
37     int i, total = 0;
38
39     for (i = 0; i < size(v); i++) {
40         printf("Viagem %02d: %d passageiros\n", i + 1, element_at(v, i));
41         total += element_at(v, i);
42     }
43
```

Exemplo de remoção em posição arbitrária

```
44      printf("Total de passageiros: %d\n", total);
45  }
46
47  int main() {
48      vector v = create_vector(10);
49      int opcao;
50
51      if (!v) {
52          fprintf(stderr, "Falha na criação do vetor!\n");
53          return -1;
54      }
55
56      while (1) {
57          printf("\n\nDigite a opcao desejada: \n");
58          printf("1. Inserir informações sobre viagem\n");
59          printf("2. Remover informações sobre viagem\n");
60          printf("3. Relatório de viagens\n");
61          printf("4. Sair\n");
62
63          if (scanf("%d", &opcao) < 1 || opcao < 1 || opcao > 3)
64              break;
```

Exemplo de remoção em posição arbitrária

```
65
66          switch (opcao) {
67          case 1:
68              inserir(v);
69              break;
70          case 2:
71              remover(v);
72              break;
73          case 3:
74              imprimir(v);
75              break;
76          }
77      }
78
79      printf("Encerrando o programa...\n");
80      free_vector(v);
81
82      return 0;
83  }
```

Referências

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
3. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.