

Vetores

Implementação de um contêiner vector em C

Prof. Edson Alves - UnB/FGA

2020

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

Cabeçalho vector.h

```
1  #ifndef C_VECTOR_H
2  #define C_VECTOR_H
3
4  typedef struct _vector {
5      int *data;
6      int size;
7      int capacity;
8  } vector;
9
10 extern vector * create_vector(int capacity);
11 extern void free_vector(vector *v);
12 extern int element_at(vector *v, int index);
13 extern int size(vector *v);
14
15 extern int push(vector *v, int value, int index);
16 extern int push_back(vector *v, int value);
17 extern int pop(vector *v, int index);
18 extern int pop_back(vector *v);
19 extern void clear(vector *v);
20
21 #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()

```
1 #include <stdlib.h>
2 #include "vector.h"
3
4 vector * create_vector(int capacity)
5 {
6     if (capacity < 1) return NULL;
7
8     vector *v = (vector *) calloc(1, sizeof(vector));
9
10    if (!v) return NULL;
11
12    v->capacity = capacity;
13    v->data = (int *) malloc(capacity * sizeof(int));
14
15    if (!v->data) {
16        free(v);
17        return NULL;
18    }
19
20    return v;
21 }
```


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
- Esta técnica é chamada *private implementation (PIMPL)*

Interface com a técnica do PIMPL

```
1 #ifndef C_VECTOR_ADT_H
2 #define C_VECTOR_ADT_H
3
4 typedef void * vector;
5
6 extern vector create_vector(int capacity);
7 extern void free_vector(vector v);
8 extern int element_at(const vector v, int index);
9 extern int size(const vector v);
10
11 extern int push(vector v, int value, int index);
12 extern int push_back(vector v, int value);
13 extern int pop(vector v, int index);
14 extern int pop_back(vector v);
15 extern int clear(vector v);
16
17 #endif
```

Implementação com a técnica do handler

```
1 #include <stdlib.h>
2 #include "vector_adt.h"
3
4 typedef struct _vector {
5     int *data;
6     int size;
7     int capacity;
8 } pvector;
9
10 vector create_vector(int capacity)
11 {
12     if (capacity < 1) return NULL;
13
14     pvector *pv = (pvector *) calloc(1, sizeof(pvector));
15
16     if (!pv) return NULL;
17
18     pv->capacity = capacity;
19     pv->data = (int *) malloc(capacity * sizeof(int));
20 }
```

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 gera um erro de segmentação
- Além disso, há outro erro a ser considerado: o índice passado pode estar fora do intervalo $[0, N - 1]$
- Uma forma de reportar o erro ao usuário da função é retornar um código erro numérico
- 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 campo `size`
- Se o vetor estiver cheio (isto é, se o tamanho for igual a capacidade), há duas estratégias possíveis
- 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

52	38	61	
----	----	----	--

Visualização da inserção

Elemento a ser inserido: 14

Capacidade: 4

Tamanho: 4

52	38	61	14
----	----	----	----

Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 4

Tamanho: 4

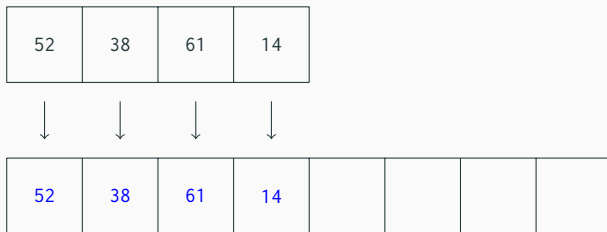
52	38	61	14
----	----	----	----

Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 4

Tamanho: 4



Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 8

Tamanho: 4

52	38	61	14				
----	----	----	----	--	--	--	--

Visualização da inserção

Elemento a ser inserido: 77

Capacidade: 8

Tamanho: 5

52	38	61	14	77			
----	----	----	----	----	--	--	--

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

2	3	6	8	14	21	72	99	
---	---	---	---	----	----	----	----	--

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

2	3	6	8	14	21	72		99
---	---	---	---	----	----	----	--	----

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

2	3	6	8	14	21		72	99
---	---	---	---	----	----	--	----	----

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 8

2	3	6	8	14		21	72	99
---	---	---	---	----	--	----	----	----

Visualização da inserção em posição arbitrária

Elemento a ser inserido: 19

Posição: 5

Tamanho: 9

2	3	6	8	14	19	21	72	99
---	---	---	---	----	----	----	----	----

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

```
1 #include <stdio.h>
2 #include "vector_adt.h"
3 #include "vector_errors.h"
4
5 int main()
6 {
7     vector v = create_vector(10);
8     int N, rc;
9
10    if (!v) {
11        fprintf(stderr, "Falha na criação do vetor!\n");
12        return -1;
13    }
14
15    while (1) {
16        printf("Informe o número de passageiros na viagem (Ctrl-D para"
17              " encerrar): ");
18
19        if (scanf("%d", &N) < 1)
20            break;
21    }
```

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 tentativa de remoção de elemento em um vetor vazio constitui um erro, o qual deve ser tratado
- 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

52	38	61	14				
----	----	----	----	--	--	--	--

Visualização da remoção ao final

Elemento a ser removido: 14

Capacidade: 8

Tamanho: 3

52	38	61					
----	----	----	--	--	--	--	--

Implementação das funções pop_back() e clear()

```
108 int pop_back(vector v) {  
109     pvector *pv = (pvector *) v;  
110  
111     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;  
112  
113     if (pv->size == 0) return VECTOR_ERROR_EMPTY_VECTOR;  
114  
115     pv->size--;  
116  
117     return VECTOR_OK;  
118 }  
119  
120 int clear(vector v) {  
121     pvector *pv = (pvector *) v;  
122  
123     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;  
124  
125     pv->size = 0;  
126  
127     return VECTOR_OK;  
128 }
```

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

2	3	6	8	14	21	72	99	
---	---	---	---	----	----	----	----	--

Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

Tamanho: 8

2	3	6	8	21		72	99	
---	---	---	---	----	--	----	----	--

Visualização da remoção em posição arbitrária

Elemento a ser removido: 14

Posição: 5

Tamanho: 8

2	3	6	8	21	72		99	
---	---	---	---	----	----	--	----	--

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

```
130 int pop(vector v, int index)
131 {
132     pvector *pv = (pvector *) v;
133     int i;
134
135     if (!pv) return VECTOR_ERROR_NULL_PARAMETER;
136
137     if (pv->size == 0) return VECTOR_ERROR_EMPTY_VECTOR;
138
139     if (index < 0 || index >= pv->size)
140         return VECTOR_ERROR_INDEX_OUT_OF_BOUNDS;
141
142     for (i = index + 1; i < pv->size; ++i)
143         pv->data[i - 1] = pv->data[i];
144
145     --pv->size;
146
147     return VECTOR_OK;
148 }
```

Exemplo de remoção em posição arbitrária

```
1 #include <stdio.h>
2 #include "vector_adt.h"
3 #include "vector_errors.h"
4
5 void inserir(vector v) {
6     int N, rc;
7
8     printf("Insira o número de passageiros da viagem %d: ", size(v) + 1);
9
10    if (scanf("%d", &N) < 1) {
11        fprintf(stderr, "Quantidade inválida!\n");
12        return;
13    }
14
15    rc = push_back(v, N);
16
17    if (rc == VECTOR_OK)
18        printf("Informação inserida com sucesso!\n");
19    else
20        fprintf(stderr, "Erro na inserção: %d\n", rc);
21 }
```

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 }
```

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.