Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

31 de janeiro de 2024

Vetores

Sequência do mesmo tipo de tamanho limitado

- Vetores é uma sequência de armazenamento de um mesmo tipo em memória.
- O tipo de dado armazenado pode ser primitivo ou abastrato.
- Para acessar os valores armazenados em um vetor são usados índices.
  - O tamanho definido do vetor representa quantos elementos ele possui
  - Cada elemento é representado por um índice no vetor a partir do 0.
- O tamanho do vetor pode (em algumas linguagens) ser definido na declaração do vetor ao se atribuir valores.

Saída

```
int main()
{
    int i;
    int vint1[10];
    int vint2[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for(i = 0; i < 10; i++)
        vint1[9-i] = vint2[i];
    for(i = 0; i < 10; i++)
        printf("%d..",vint1[i]);
    printf("\n");
    return 0;
}
```

```
9..8..7..6..5..4..3..2..1..0..
```

Exemplo básico (será?) de vetores

```
typedef struct st_complexo
{
    double real;
    double imag;
} complexo;
int main() {
    int i;
    complexo c1[3];
    complexo c2[] = {{2, 3}, {3, 4}, {1, 9}};
    for(i = 0; i < 3; i++)
    {
        c1[2-i].real = c2[i].imag;
        c1[2-i].imag = c2[i].real;
    }
    for(i = 0; i < 3; i++)
        printf("(%f + i%f)..",c1[i].real,c1[i].imag);
    printf("\n");
    return 0;
}
```

```
(9.000000 + i1.000000)..(4.000000 + i3.000000)..(3.000000 + i2.000000)..
```

Manipulando vetores a partir de ponteiros

- Ao declarar um vetor, como no exemplo: `int v[5];`
  - É alocado na memória um espaço para 5 inteiros, em sequência.
  - a variável `v` representa o endereço da primeira posição da memória.
  - este mesmo endereço é obtido diretamente pelo operador endereço & sobre o primeiro elemento do vetor: `&v[0]`.
  - `v` está armazenada exatamente no endereço que ela guarda.
- Podemos usar ponteiros para fazer referências a estes endereços.
- Uma vez declarado um ponteiro para o tipo inteiro (tipo do vetor): `int *p;`
  - Ele guarda o endereço para um elemento do vetor.
  - Incrementar o ponteiro de um, incrementa para o endereço do próximo elemento, automaticamente no tamanho do espaço ocupado por cada elemento.

Exemplo de endereçamento por ponteiros

```
int main() {
    int v[5] = {0, 1, 2, 3, 4};
    int *p;
    p = v;
    printf("Endereço do vetor: %p\n",p);
    p = &v[0];
    printf("Endereço do primeiro elemento: %p\n",p);
    printf("Valor do primeiro elemento: %d\n",*p);
    p++;
    printf("Endereço do segundo elemento: %p\n",p);
    printf("Valor do segundo elemento: %d\n",*p);
    p = (int *) &v;
    printf("Endereço da variável v: %p\n",p);
    return 0;
}
```

```
Endereço do vetor:  0x7fffe4336e30
Endereço do primeiro elemento:  0x7fffe4336e30
Valor do primeiro elemento:  0
Endereço do segundo elemento:  0x7fffe4336e34
Valor do segundo elemento:  1
Endereço da variável v:  0x7fffe4336e30
```

Ponteiros para vetores de tipos abstratos

```
typedef struct racional_s {
    int num, den;
} racional;
int main(int argc, char **args) {
    racional r[5] = {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}};
    racional *pr;
    pr = r;
    printf("Endereço do vetor: %p\n",pr);
    printf("Valor do primeiro elemento: %d/%d\n",pr->num,pr->den);
    pr++;
    printf("Endereço do segundo elemento: %p\n",pr);
    printf("Valor do segundo elemento: %d/%d\n",(*pr).num,(*pr).den);
    return 0;
}
```

```
Endereço do vetor:  0x7fffaedb6460
Valor do primeiro elemento:  1/2
Endereço do segundo elemento:  0x7fffaedb6468
Valor o segundo elemento:  2/3
```

Um pouco de abstração

```
class racional {
  private:
    int num, den;
  public:
    racional() {
      num = 0;
      den = 1;
    }
    racional(int n, int d) {
      num = n;
      den = d;
    }
    int numerador() {
      return num;
    }
    int denominador() {
      return den;
    }
};
```

Usando a classe racional

```
#include <iostream>
    using namespace std;
int main(int argc, char **args) {
    racional r[3];
    racional *pr;

    r[0] = racional(1,2);
    r[1] = racional(2,3);
    r[2] = racional(3,4);
    pr = r;
    cout << "Endereço do vetor: " << r << endl;
    pr = &r[0];
    cout << "Endereço do primeiro elemento: " << pr << endl;
    cout << "Valor do primeiro elemento: " << r[0].numerador() << "/" <<
r[0].denominador() << endl;
    pr++;
    cout << "Endereço do segundo elemento: " << pr << endl;
    cout << "Valor do segundo elemento: " << pr->numerador() << "/" <<
pr->denominador() << endl;
    return 0;
}
```

Resultado do programa:

```
Endereço do vetor:  0x7fffa5dca050
Endereço do primeiro elemento:  0x7fffa5dca050
Valor do primeiro elemento:  1/2
Endereço do segundo elemento:  0x7fffa5dca058
Valor do segundo elemento:  2/3
```

Algumas curiosidades na programação C++:

- O uso de `private` ou `publico` está associado com o escopo de visibilidade dos elementos na orientação a objetos (não é tema desta disciplina)
- Podemos resumir da forma: para acessarmos um elemento interno: "(p-¿num)", por exemplo, este elemento deve estar definido após a declaração explícita de "publico:".

Funções com vetores funciona de forma simples

```
void troca(int[3]);

int main() {
    int i;
    int vint[3] = {0, 1, 2};
    troca(vint);
    for(i = 0; i < 3; i++)
        printf("%d..",vint[i]);
    return 0;
}
void troca(int v[3]) {
    int aux;
    aux = v[2];
    v[2] = v[0];
    v[0] = aux;
    return;
}
```

```
2..1..0..
```

Podemos evitar especificar o tamanho do vetor no parâmetro da função

```
void troca(int*);

int main() {
    int i;
    int vint[3] = {0, 1, 2};
    troca(vint);
    for(i = 0; i < 3; i++)
        printf("%d..",vint[i]);
    return 0;
}
void troca(int *v) {
    int aux;
    aux = v[2];
    v[2] = v[0];
    v[0] = aux;
    return;
}
```

Revendo o conceito de vetores e ponteiros

- A diferença entre ambos exemplos anteriores é que como parâmetro especificamos: `int *v`
- Ou seja, `v` é um ponteiro para inteiro que tem o endereço de `vint`
- Ainda assim usamos `v[0]`.
- Será que funciona somente como parâmetro? Ou sempre?
- É mais interessante que isto. Veja o próximo código.

A ordem das parcelas não altera a soma!

```
int main() {
    int vint[3] = {0, 1, 2};
    int *v;

    v = vint;

    printf("Segundo elemento: %d\n", vint[1]);
    printf("Segundo elemento: %d\n", v[1]);
    printf("Segundo elemento: %d\n", *(v+1));
    printf("Segundo elemento: %d\n", *(1+v));
    printf("Segundo elemento: %d\n", 1[v]);
    return 0;
}
```

```
Segundo elemento:  1
Segundo elemento:  1
Segundo elemento:  1
Segundo elemento:  1
Segundo elemento:  1
```

Considerações finais

- A passagem de parâmetros de tipos abstratos é semelhante aos tipos primitivos
- Para isto é necessário definir um tipo (`typedef`), pois este será especificado como tipo do parâmetro.
- Isto acontece naturalmente no C++ pois a classe em si é um tipo.
- Aqui nos restringimos principalmente à aplicação com estrutura de dados, mas o uso de ponteiros é muito mais poderosa.
- Podemos passar como parâmetro de uma função, outra função (Teremos um exemplo rápido brincando com busca em árvores).

Aplicações de estrutura de vetores

- Veremos nas próximas aulas estruturas mais tradicionais baseadas em vetores.
- Mas os vetores são, em si, já uma estrutura abstrata, e podem ser usados como tal.
- Por exemplo um vetor pode possuir os valores de uma função matemática para um intervalo de pontos.
- Desta forma podemos querer encontrar algo neste vetor, como o maior valor.
- Assim temos os algoritmos de busca que percorrem as estruturas de dados para encontrar algum valor.

Busca Sequencial

- A busca sequencial é uma busca que se faz olhando cada elemento, um por um, em sequência.
- A busca pode se interromper, por exemplo, se um elemento é encontrado.
- No pior dos casos, todos os elementos precisam ser olhados.
- Na busca do maior, se não olharmos todos, pode ser que algum que não olhamos era maior.
- Às vezes a busca é por todos os elementos: quero a soma de todos.
- Dizemos que numa busca sequencial, precisamos percorrer os n elementos, então o algoritmo leva um "tempo" de tamanho $n : O(n)$.

Busca Sequencial: buscando um valor x

Entrada: Um vetor A e um valor x que desejamos em A
Saída: O índice de x em A ou -1, se não estiver em A

Algoritmo BUSCA(A,x)
  $pos \leftarrow -1$.
  $i \leftarrow 1$
  enquanto $i < \text{tamanho}(A)$ **e** $pos = -1$ **faça**
    se $A[i] = x$ **então**
      $pos \leftarrow i$
  retorne pos

- No pior caso, quantos elementos tenho de olhar?

Busca Sequencial: buscando um valor x

Entrada: Um vetor A e um valor x que desejamos em A
Saída: O índice de x em A ou -1, se não estiver em A

Algoritmo BUSCA(A,x)
  $pos \leftarrow -1$.
  $i \leftarrow 1$
  enquanto $i < \text{tamanho}(A)$ **e** $pos = -1$ **faça**
    se $A[i] = x$ **então**
      $pos \leftarrow i$
  retorne pos

- No pior caso, quantos elementos tenho de olhar?

Busca Sequencial: buscando um valor x

Entrada: Um vetor A e um valor x que desejamos em A
Saída: O índice de x em A ou -1, se não estiver em A

Algoritmo BUSCA(A,x)
  $pos \leftarrow -1$.
  $i \leftarrow 1$
  enquanto $i < \text{tamanho}(A)$ **e** $pos = -1$ **faça**
    se $A[i] = x$ **então**
      $pos \leftarrow i$
  retorne pos

- No pior caso, quantos elementos tenho de olhar? n

Busca Sequencial: buscando o maior

Entrada: Um vetor A
Saída: O índice de seu maior elemento

Algoritmo BuscaMaior(A)
    $maior \leftarrow 1$
    para $i \leftarrow 2$ **até** $\text{tamanho}(A)$ **faça**
        se $A[maior] < A[i]$ **então**
            $maior \leftarrow i$
    retorne $maior$

- Quantos elementos precisamos olhar?

Busca Sequencial: buscando o maior

Entrada: Um vetor A
Saída: O índice de seu maior elemento

Algoritmo BuscaMaior(A)
    $maior \leftarrow 1$
    para $i \leftarrow 2$ **até** $\text{tamanho}(A)$ **faça**
        se $A[maior] < A[i]$ **então**
            $maior \leftarrow i$
    retorne $maior$

- Quantos elementos precisamos olhar? n

Busca sequencial: Outros casos

- Buscar o maior e menor simultaneamente.
  Há um truque aqui, varrer de 2 em 2, reduz o número de IFs.
- Somar todos os elementos.
- Encontrar a primeira ou a última posição de um elemento.
- Encontrar, se existe uma subsequência exata (uma palavra, por exemplo).
- Encontrar uma subsequência de maior soma. (Este é divertido, dá para fazer olhando no máximo n elementos.
- ...

Busca sequencial: Considerações finais

- Como o próprio nome diz, a busca é feita em sequência.
- Podemos começar do início e ir buscando até o fim.
- Podemos começar do fim e ir buscando até o início.
- Podemos começar simultaneamente do início e do fim. (Tem um problema legal desta forma na lista).

Busca binária

- Esta busca se aplica em um vetor classificado.
- Iniciamos a busca no vetor a partir do elemento que está no meio do vetor.
- Como os itens estão ordenados, à direita do elemento estão elementos não menores que ele, e à esquerda os não maiores
- Se o valor buscado for menor que este elemento, retomamos a busca pela metade inferior do vetor..
- Da mesma forma, caso contrário, em sua metade superior.

Busca binária - algoritmo

Entrada: Um vetor classificado A, e um valor x
Saída: O índice de uma ocorrência de x em A, ou -1 se não existir

Algoritmo BuscaBinaria(A,x)

```
pos ← −1
ini ← 1
fim ← tamanho(A)
enquanto ini ⩽ fim e pos = −1 faça
    meio ← (ini + fim)/2
    se A[meio] = x então
        pos ← meio
    senão
        se A[meio] > x então
            fim ← meio − 1
        senão
            ini ← meio + 1
retorne pos
```

- Quantos elementos olhamos, no pior caso?

Busca binária - algoritmo

Entrada: Um vetor classificado A, e um valor x
Saída: O índice de uma ocorrência de x em A, ou -1 se não existir

Algoritmo BuscaBinaria(A,x)
  $pos \leftarrow -1$
  $ini \leftarrow 1$
  $fim \leftarrow \text{tamanho}(A)$
  enquanto $ini \leqslant fim$ **e** $pos = -1$ **faça**
    $meio \leftarrow (ini + fim)/2$
    se $A[meio] = x$ **então**
      $pos \leftarrow meio$
    senão
      se $A[meio] > x$ **então**
        $fim \leftarrow meio - 1$
      senão
        $ini \leftarrow meio + 1$
  retorne pos

- Quantos elementos olhamos, no pior caso? $\log n$

Busca binária: Considerações

- A busca binária encontra uma ocorrência do valor, não necessariamente a primeira.
- Se considerarmos a sequência em um espaçamento regular, podemos fazer uma modificação na busca binária: busca por interpolação.
  - O elemento do "meio" é encontrado em uma regra de três entre o valor que queremos encontrar e os valores dos extremos da sequência.
  - Se o valor é um valor próximo de uma das pontas, o elemento do "meio" será um elemento proporcionalmente próximo a este extremo.
- Existe uma versão recursiva do algoritmo de busca binária. Ao invés do while, apenas chamamos o algoritmo recursivamente. Retomamos esta ideia em uma próxima aula.

Ordenação por troca: Bubble Sort

- Buscamos o vetor do início ao fim, se houver um par fora de ordem, trocamos a ordem do par.
- Na primeira busca o maior elemento é posicionado em sua posição final, pois ele sempre vai trocar com um elemento menor que ele que estiver à frente no vetor.
- Repetimos a busca continuamente terminando sempre um elemento antes (pois acabamos de colocar o maior em sua posição final).
- Opcionalmente podemos parar se não houver mais troca. Ou seja, todos já estão em sua posição.

Ordenação por troca: Bubble Sort - algoritmo

Entrada: Vetor A de elementos em ordem qualquer
Saída: Vetor de elementos em ordem não decrescente

Algoritmo BubbleSort(A)
    fim $\leftarrow$ **falso**
    $n \leftarrow \text{tamanho}(A)$
    enquanto $n > 1$ **e** $\neg fim$ **faça**
        fim $\leftarrow$ **verdade**
        para $j \leftarrow 2$ **até** n **faça**
            se $A[j-1] > A[j]$ **então**
                $A[j-1] \leftrightarrow A[j]$
                fim $\leftarrow$ **falso**
        $n \leftarrow n - 1$
retorne A

Ordenação por troca: Bubble Sort - Considerações

- São realizadas em torno de n^2 buscas, dizemos que este algoritmo é $O(n^2)$
- O nome Bubble (bolha) advem do fato do maior elemento sempre subir para a última posição de forma mais rápida.
- Podemos alterar este algoritmo fazendo uma bolha vai-e-vem, na primeira passada levamos o maior elemento para o final, e voltamos trazendo o menor para o início.
- A versão alterada não muda em nada a performance do algoritmo.
- É um algoritmo prático para ordenar um pequeno número de elementos 3.

Ordenação por troca: Bubble Sort - O poder do C++

- Vamos considerar inicialmente um caso simples.
- Um vetor de inteiros.
- Leio uma sequência de números, ordeno em ordem não crescente, e imprimo.
- Apesar de C++, este código é basicamente C simples.
- As entradas e saídas usam C++.

O programa principal e a ordenação

```
int main(int argc, char **args) {
  int v[MAX_N];
  int tamanho;

  tamanho = lerVetor(v);
  ordenar(v,tamanho);
  imprimirVetor(v,tamanho);
  return 0;
}
```

O programa principal e a ordenação

```
void ordenar(int *v, int tamanho) {
  bool fim = false;
  while((tamanho > 0) && !fim) {
    fim = true;
    for(int j = 1; j < tamanho; j++)
      if(v[j-1] < v[j]) {
        int aux = v[j-1];
        v[j-1]=v[j];
        v[j]=aux;
        fim = false;
      }
    tamanho--;
  }
  return;
}
```

Usando um tipo abstrato

- Vamos fazer diferente, não quero trabalhar com inteiros.
- Um exemplo de ordem é a classificação dos países no quadro de medalhas das olimpíadas, vamos trabalhar com uma estrutura que indique o país e suas medalhas recebidas.
- Queremos imprimir um quadro de medalhas do primeiro ao último.
- Este problema é basicamente igual ao anterior:
- Ler os valores, ordenar de forma decrescente, imprimir.

O programa principal e a ordenação

```
int main(int argc, char **args) {
  medalha v[MAX_N];
  int tamanho;

  tamanho = lerVetor(v);
  ordenar(v,tamanho);
  imprimirVetor(v,tamanho);
  return 0;
}
```

O programa principal e a ordenação

```
void ordenar(medalha *v, int tamanho) {
  bool fim = false;
  while((tamanho > 0) && !fim) {
    fim = true;
    for(int j = 1; j < tamanho; j++)
      if(v[j-1] < v[j]) {
        medalha aux = v[j-1];
        v[j-1]=v[j];
        v[j]=aux;
        fim = false;
      }
    tamanho--;
  }
  return;
}
```

O tipo abstrato (classe) medalha

- O tipo medalha é uma classe que contem uma estrutura complexa: pais e número de medalhas.
- Os elementos deste tipo possuem uma ordem de classificação.
- No C++ é possível sobrecarregar operadores para um determinado tipo. Inclusive os operadores de desigualdade.
- Para medalha, vamos sobrecarregar o operador de atribuição para facilmente atribuir um valor indicado por uma variável a outra.
- Vamos também sobrecarregar os operadores de comparação de ordem.
- E, dado o escopo e visibilidade, vamos criar os métodos para construir um objeto e extrair seus valores.

O tipo abstrato (classe) medalha

```
class medalha {
private:
  std::string ps;
  int o, p, b;

public:
  medalha();
  medalha(std::string pais, int ouro, int prata, int bronze);
  std::string pais();
  int ouro();
  int prata();
  int bronze();
  medalha operator=(medalha m);
  bool operator>(medalha m);
  bool operator<(medalha m);
  bool operator==(medalha m);
  bool operator>=(medalha m);
  bool operator<=(medalha m);
  bool operator! =(medalha m);
};
```

O tipo abstrato (classe) medalha

```
medalha medalha::operator=(medalha m) {
  ps = m.ps;
  o = m.o;
  p = m.p;
  b = m.b;
  return *this;
}
bool medalha::operator<(medalha m) {
  bool ret = false;
  if (o < m.o) ret = true;
  else if(o == m.o && p < m.p) ret = true;
  else if(o == m.o && p == m.p && b < m.b) ret = true;
  return ret;
}
```

Algoritmos, classes e modelos

- No algoritmo de ordenação, quando usamos classes (C++), não precisamos reescrever todo o algoritmo.
- Isto se dá pois as classes podem ter os operadores de desigualdade sobrepostos, assim comparar ordem de 2 inteiros é semelhante a comparar ordem de 2 objetos.
- Vamos mais além. O algoritmo independe do tipo de elemento que usamos.
- Então, é possível definir um modelo de tipo (*template*) que pode representar qualquer tipo existente.
- Se o algoritmo consegue executar suas ações com elementos de um modelo de tipo, ele consegue trabalhar com qualquer tipo.
- Isto acontece com os algoritmos genéricos, mas este é um tema para Tópicos Avançados em Algoritmos!!