Universidade de Brasília

Programação Orientada a Objetos

1º semestre - 2022

Prof. Luís Filomeno

Conteúdo desse capítulo

1 Fundamentos de matrizes.

2- Matrizes como dados de membros de classe.

3- Matrizes de objetos.

4 – A classe de string padrão em C++ padrão.

5 -

7.2– Matrizes como dados de membros de classe.

Matrizes podem ser usados como itens de dados em classes. Vejamos um exemplo que modela uma estrutura de dados de computador comum: a pilha.

Uma pilha funciona como os dispositivos de mola que seguram as bandejas nas cafeterias. Quando você coloca uma bandeja em cima, a pilha afunda um pouco; quando você tira uma bandeja, ela reaparece. A última bandeja colocada na pilha é sempre a primeira bandeja removida.

As pilhas são um dos pilares da arquitetura dos microprocessadores usados na maioria dos computadores modernos. Como vimos anteriormente, as funções passam seus argumentos e armazenam seu endereço de retorno na pilha. Esse tipo de pilha é implementado parcialmente em *hardware* e é mais acessado em linguagem *assembly (assembler)*. No entanto, as pilhas também podem ser criadas completamente em *software*. As pilhas de *software* oferecem um dispositivo de armazenamento útil em determinadas situações de programação, como ao analisar (analisar) expressões algébricas. Nosso programa de exemplo, **Prog7_7.cpp**, cria uma classe de pilha simples.

7.2– Matrizes como dados de membros de classe.

Prog7_7.cpp - Uma pilha como uma classe.

```
#include <iostream>
using namespace std;
//--------------------------------------------------------------------------
class Pilha
{
  private:
    enum {MAX = 10};                          //sintaxe não padronizada
    int st[MAX];                              //pilha: vetor de inteiros
    int top;                                  // número do topo da pilha

 public:
    :Pilha()                                  //construtor
      { top = 0;}
    void push(int var)                        //para colocar o nro na pilha
      {st[++top] = var;}
    int pop()                                 //tirar o número da pilha
      {return st[top--];}
};
//--------------------------------------------------------------------------
int main()
{  Pilha p1;
   p1.push(11); p1.push(22);
   cout << "1: " <<p1.pop() << endl;          //escreve 22
   cout << "2: " <<p1.pop() << endl;          //escreve 11
   p1.push(33);
   p1.push(44);
   p1.push(55);
   p1.push(66);
   cout << "3: " <<p1.pop() << endl;          //escreve 66
   cout << "4: " <<p1.pop() << endl;          //escreve 55
   cout << "5: " <<p1.pop() << endl;          //escreve 44
   cout << "6: " <<p1.pop() << endl;          //escreve 33

  return 0;
}
```

7.2– Matrizes como dados de membros de classe.

O membro importante da pilha é vetor *st*. Uma variável *int*, *top*, indica o índice do último item colocado na pilha; a localização deste item é o topo da pilha. O tamanho do vetor usado para a pilha é especificado por MAX, na instrução:
enum { MAX = 10 };

Esta definição de MAX é incomum. De acordo com a filosofia de encapsulamento, é preferível definir constantes que serão usadas inteiramente dentro de uma classe, como MAX está aqui, dentro da classe. Assim, o uso de variáveis *const* **globais** para esse propósito não é ideal. O padrão C++ exige que possamos declarar MAX dentro da classe como
static const int MAX = 10;

Isso significa que MAX é constante e se aplica a todos os objetos da classe. Infelizmente, alguns compiladores, incluindo a versão atual do Microsoft Visual C++, não permitem essa construção recém-aprovada.

Como solução alternativa, podemos definir tais constantes como enumeradores (descritos no Capítulo 4).

7.2– Matrizes como dados de membros de classe.

Não precisamos nomear a enumeração e precisamos apenas de um enumerador:

enum { MAX = 10 };

Isso define MAX como um inteiro com o valor 10 e a definição está contida inteiramente na classe. Essa abordagem funciona, mas é estranha. Se o seu compilador suportar a abordagem *static const* , deve-se usá-la para definir constantes dentro da classe.

A Figura 7.6 mostra uma pilha. Como a memória cresce para baixo na figura, o topo da pilha fica na parte inferior da figura. Quando um item é adicionado à pilha, o índice e, *top* é incrementado para apontar para o novo topo da pilha. Quando um item é removido, o índice em *top* é diminuído. Para colocar um item na pilha - um processo chamado *push* do item - você chama a função de membro *push()* com o valor a ser armazenado como um argumento.

Para recuperar (ou *pop*) um item da pilha, você usa a função de membro *pop()*, que retorna o valor do item. O programa *main()* em ***Prog7_7.cpp*** exercita a classe de pilha criando **um objeto, *p1*, da classe**.

7.2– Matrizes como dados de membros de classe.

Ele empurra dois itens para a pilha e os remove e os exibe. Então empurra mais quatro itens na pilha, e os retira e os exibe. Aqui está a saída:

1: 22
2: 11
3: 66
4: 55
5: 44
6: 33

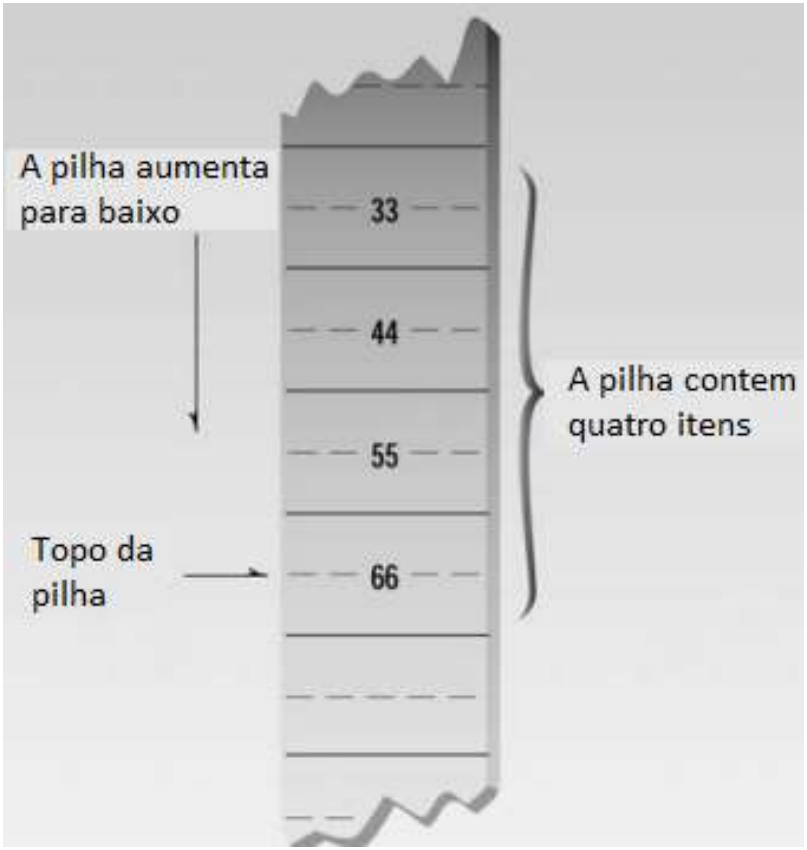

Figura 7.6 – Uma representação da pilha.

7.2– Matrizes como dados de membros de classe.

Prog7_7.cpp - Uma pilha como uma classe.

Como pode-se ver, os itens são retirados da pilha na ordem inversa; a última coisa empurrada é a primeira coisa que estourou (foi retirada). Observe o uso sutil de notação de *prefixo* e *postfix* nos operadores de incremento e decremento. A declaração:

st[++top] = var;

na função de membro *push()* primeiro incrementa *top* para que aponte para o próximo elemento de array disponível-um após o último elemento. Em seguida, atribui *var* a esse elemento, que se torna o novo topo da pilha. A declaração:

return st[top--];

primeiro retorna o valor encontrado no topo da pilha e, em seguida, diminui o *top* para que aponte para o elemento anterior (precedente).

A classe pilha é um exemplo de um recurso importante da programação orientada a objetos: usar uma classe para implementar mecanismo de armazenamento de dados, tais como: filas, conjuntos, listas vinculadas etc.

7.3– Matriz de objetos

Vimos como um objeto pode conter um vetor. Também pode-se reverter essa situação e criar um vetor de objetos. Analisaremos uma situação: uma série de distâncias . No Capítulo 6, “Objetos e Classes”, mostramos vários exemplos de classe que incorporou *metros* e *cms* em um objeto representando um novo tipo de dados. No próximo programa, **Prog7_8.cpp**, demonstra uma matriz de tais objetos.

Prog7_8.cpp – Objetos usando medições inglesas.

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class distancia
{
  private:
    int metros;
    float cms;

 public:
   void obtemdist()              // obtem os dados de distancia do usuário
     {
       cout << "\n Digite metros: " ;  cin>> metros;
       cout << "\n Digite cms   : " ;  cin>> cms;
     }
   void  mostradist()           // mostra os dados da distancia
     {
       cout << " metros << "\'-" << cms << "'\" ;  }
     }
};
////////////////////////////////////////////////////////////////
```

7.3– Matriz de objetos

Prog7_8.cpp – Objetos usando medições inglesas (continuação).

```
int main()
{
   distancia dist [20];                          //vetor de distâncias;
   int n = 0;                                    //contador de distâncias
   char resp;                                    //resposta ao usuário ('y'ou'n')
   do{
      cout << "Digite o nro da distância: " << n+1;
      dist[n++].obtemdist();                     //armazena as distancias num
vetor
      cout <<" Digite outra distancia(Y/N)?:";
      cin  >> resp;
     }while((resp != 'n')||(resp != 'N'));      //sair se o usuário digitar n ou
N

   for(int j=0; j<n; j++)
     {
      cout << "\n Nro da distância " << j+1 << " é ";
      dist[j].mostradist();
     }
  cout << endl;
  return 0;
}
```

Neste programa o usuário digita quantas distâncias desejar. Após cada distância ser inserida, o programa pergunta se o usuário deseja inserir outra distância.

7.3– Matriz de objetos

Caso contrário, ele termina e exibe todas as distâncias inseridas até o momento.
Aqui está um exemplo de interação quando o usuário insere duas distâncias:

```
Digite o nro da distância 1
Digite metros: 5
Digite cms:     4
Digite outra distância (y/n)? y
Digite o nro da distância 2
Digite metros: 6
Digite cms:     10.75

Digite outra distância (y/n)? n
Distância número 1 é 5 - 4
Distância número 2 é 6 – 10.75
```

7.3.1– Acessando objetos em um vetor

A declaração da classe *distancia* neste programa é similar àquela utilizada em programas anteriores. Assim, no programa *main()* definimos um vetor de objetos tal:

distancia dist[MAX];

Aqui, o tipo de dados do vetor *dist* é *distancia* e possui MAX elementos. A Figura 7.7 mostra como isso se parece.

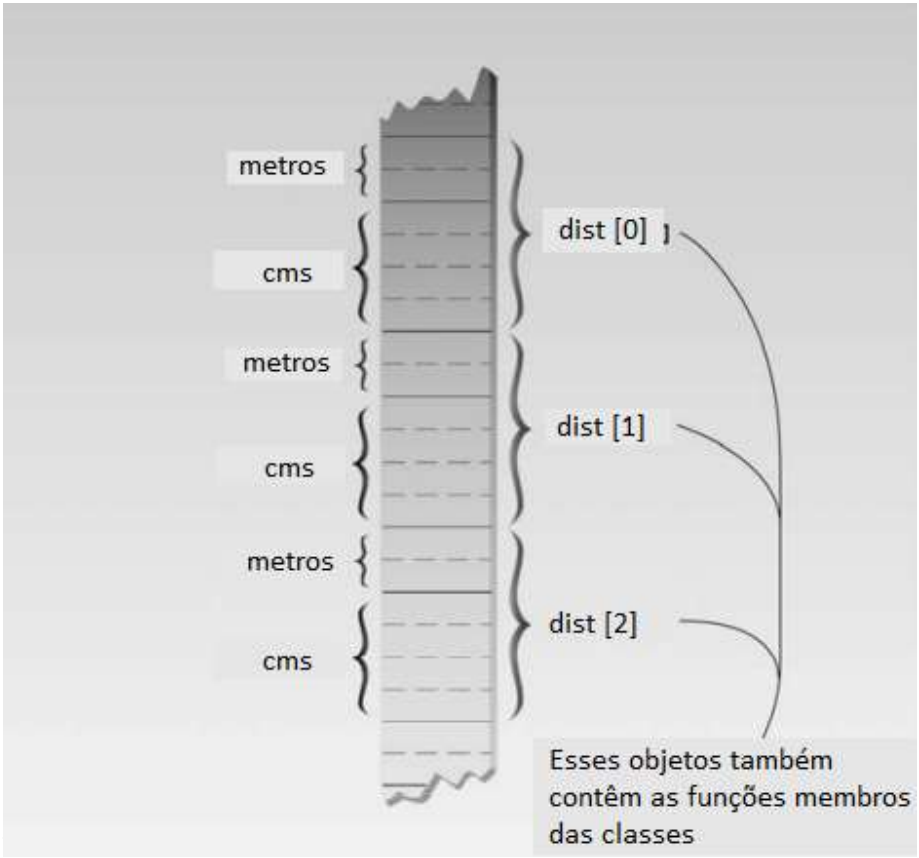

Figura 7.7 – Vetor de objetos

7.3.1– Acessando objetos em um vetor

Uma função de membro de classe que é um elemento de matriz é acessada de maneira semelhante a um membro de estrutura que é um elemento de matriz, como no exemplo **Prog7_6.cpp**. Veja como a função-membro *mostradist()* do *j-ésimo* elemento do vetor *dist* é invocada:

dist[j].mostradist();

Como pode-se ver, uma função membro de um objeto que é um elemento vetor é acessada usando o **operador ponto.** O nome do vetor seguido pelo índice entre colchetes, usando o operador ponto, ao nome da função membro seguido por parênteses. Isso é semelhante a acessar um membro de dados de estrutura (ou classe), exceto que o nome da função e os parênteses são usados em vez do nome dos dados.

Observe que quando chamamos a função membro *obtemdist()* para colocar uma distância no vetor, aproveitamos para incrementar o índice *n* do vetor :

dist[n++].obtemdist()

7.3.1– Acessando objetos em um vetor

Desta forma o próximo grupo de dados obtidos do usuário será colocado na estrutura do próximo elemento do vetor em *dist*. A variável *n* deve ser incrementada manualmente porque usamos um ciclo de repetição ***do*** em vez de um ciclo ***for***. No ciclo (laço, *loop*) ***for***, a variável do ciclo - que é incrementada automaticamente — pode servir como índice do vetor.

7.4– Classe de string padrão em C++.

Vetores de objetos são amplamente utilizados na programação C++. Veremos outros exemplos à medida que avançarmos.

7.4.1 *String* padrão em C

Observamos no início deste capítulo que dois tipos de *strings* são comumente usados em C++: *strings* em C e *strings* que são objetos da *classe string*. Nesta seção descreveremos o primeiro tipo, que se encaixa no tema do capítulo Strings em C são *arrays* (vetores) do tipo *char*. Chamamos essas *strings de C-strings*, ou *strings de estilo C*, porque elas eram o único tipo de *string* disponível na linguagem C (e nos primórdios do C++). Eles também podem ser chamados de ***char* strings***, porque eles podem ser representados como ponteiros para tipo *char*. (O * indica um ponteiro, como veremos adiante).

Embora as *strings* criadas com a classe *string*, que examinaremos na próxima seção, tenham substituído as *strings* em C em muitas situações, as *strings* C ainda são importantes por vários motivos:

1) Eles são usados em muitas funções da biblioteca C;
2) Eles continuarão a aparecer no código legado nos próximos anos;

7.4– Classe de string padrão em C++.

3), para estudantes de C++, as strings C são mais primitivas e, portanto, mais fáceis de entender em um nível fundamental.

7.4.2 Variáveis *String* em C

Assim como outros tipos de dados, as *strings* podem ser *variáveis* ou *constantes*. Veremos essas duas entidades antes de examinarmos as operações de *string* mais complexas. Aqui está um exemplo que define um variável de cadeia única. (Nesta seção, vamos supor que a palavra *string* se refere a uma *string* C.) que o usuário insira uma *string* e coloca essa *string* na variável *string*. Em seguida, exibe o fragmento. Aqui está a listagem de Prog7_9.cpp.

7.4– Classe de string padrão em C++.

7.4.2 Variáveis *String* em C

Prog7_9.cpp – Variável de matriz de caracteres (*string*) simples.

```
#include <iostream>
using namespace std;

int main()
{
   const int MAX  = 40;                    //máximo de caracteres na string
   char str[MAX];                          //str - variável do tipo string

   cout << "Digite uma string(matriz de caracteres): " ;
   cin  >> str;                             //insira os caracteres da string

   //exiba os caracteres da variável str
   cout << "\n Você digitou: " <<str;
   cout << endl;
   return 0;
}
```

A definição da variável *string* **str** se parece (e é) a definição de um vetor do tipo char:

char str[MAX];

Usa-se o operador de extração >> para ler uma *string* a partir do teclado e colocá-la na variável *string* **str**. Este operador sabe como lidar com *strings*; entende que são matrizes de caracteres.

7.4– Classe de string padrão em C++.

7.4.2 Variáveis *String* em C

Se o usuário digitar a *string* "*Amanuensis*" (uma empregada para copiar manuscritos) neste programa, o vetor **str** ficará parecido com a Figura 7.8.

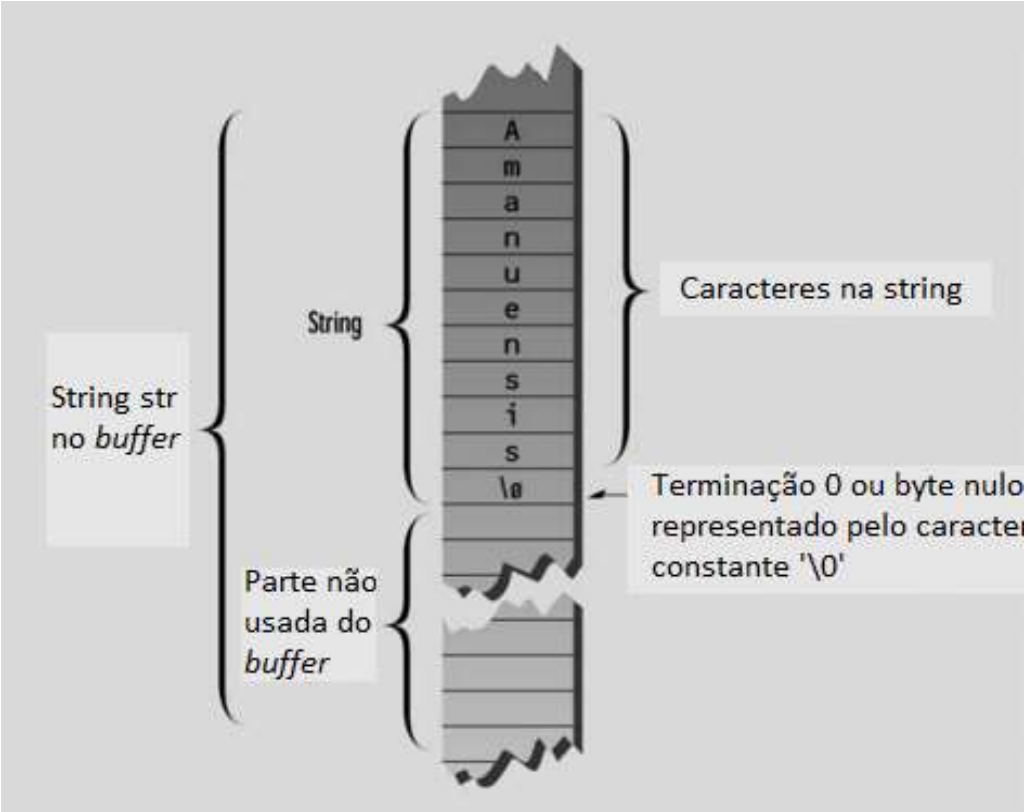

Figura 7.8- Matriz de caracteres armazenada na variável ***str***.

Cada caractere ocupa 1 byte de memória. As *strings* C++ é que elas devem terminar com constante de caractere '\0', que equivalente ao valor ASCII de 0. Quando o

7.4– Classe de string padrão em C++.

7.4.3-Evitando estouro do *buffer*

O que acontece se o usuário inserir uma *string* maior do que a matriz usada para mantê-la? Viu-se anteriormente, não há mecanismo interno em C++ para impedir que um programa insira elementos do vetor além do seu limite. Assim, um digitador entusiasmado pode acabar travando o sistema. No entanto, é possível dizer ao operador >> para limitar o número de caracteres que ele coloca em uma matriz de caracteres. O programa **Prog7_10.cpp** demonstra esta abordagem.

Prog7_10.cpp – Evita o estouro do buffer com *cin.width*.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int MAX  = 15;                    //máximo de caracteres na string
   char str[MAX];                          //str - variável do tipo string

   cout << "Digite uma string: " ;
   cin  >> setw(MAX) >>str;                //não mais do que max caracteres

   //exibe os caracteres da variável str
   cout << "\n Você digitou: " <<str <<endl;
   return 0;
}
```

7.4– Classe de string padrão em C++.

7.4.4- Lendo espaços em branco incorporados

Numa *string* que contenha mais de uma palavra, ocorrerá algo inesperado. Ao ler-se a *string* com o operador de extração >> considera um espaço como um caractere de terminação. Assim, ele lerá *strings* que consistem em uma única palavra, mas qualquer coisa digitada após um espaço é descartada. Para ler texto contendo espaços em branco, usamos outra função, ***cin.get().*** Essa sintaxe significa uma função membro da classe ***get()*** da qual ***cin*** é um objeto. O exemplo **Prog7_11.cpp** mostra como é usado.

Prog7_11.cpp – Lê *strings* de caracteres com espaços em branco incorporados.

```
#include <iostream>
using namespace std;

int main()
{
   const int MAX  = 90;
   char str[MAX];

   cout << "\nDigite uma string: " ;
   cin.get(str,MAX);

   cout << "\n Você digitou: " <<str <<endl;
   return 0;
}
```

7.4– Classe de string padrão em C++.

7.4.5- Lendo múltiplas linhas

Resolveu-se o problema de ler *strings* com espaços em branco embutidos, mas e quando a strings tem várias linhas? A função **cin::get()** pode receber um terceiro argumento para ajudar nessa situação. Este argumento especifica o caractere que diz à função para parar de ler. O valor padrão para este argumento é o caractere de nova linha (**'\n'**), mas se chamar-se a função com algum outro caractere para este argumento, o padrão será substituído pelo caractere especificado, isso mostra-se no **Prog7_12.cpp.**

Prog7_12.cpp – Lê múltiplas linhas e termina com o caractere *'$'*.

```
#include <iostream>
using namespace std;
const int MAX  = 300;
char str[MAX];

int main()
{
   cout << "\nDigite uma string:\n " ;
   cin.get(str,MAX,'$');                  // a string termina com $


   cout << "\n Você digitou: " <<str <<endl;
   return 0;
}
```

7.4– Classe de string padrão em C++.

7.4.6- Copiar *strings* de modo fácil

Para copiar uma string pode-se usar um *loop* que é uma aproximação complexa. Mas em C++ existe uma função de biblioteca ***strcpy()*** que o faz de modo mais elegante. Eis a seguir uma versão revisada de um programa, que copia uma *string* em outra *string*.

Prog7_13.cpp – Copia uma string usando a função *strcpy()*.

```
#include <iostream>
#include <cstring>              //biblioteca para strcpy
using namespace std;

int main()
{
  char str1[]="Cidade, municipio e estado\n"
              "País localizado na América";

  const int MAX =100;
  char str2[MAX];

   strcpy(str2,str1)
   cout <<str2 <<endl;
   return 0;
}
```

7.4– Classe de string padrão em C++.

7.4.7- Matrizes de *strings*

Se houver matriz de matriz, é claro que pode haver matriz de *strings*. Isso é realmente uma construção bastante útil. Aqui está um exemplo, **Prog7_14**, que coloca os nomes dos dias da semana em uma matriz:

Prog7_14.cpp – Matriz de *strings*.

```
#include <iostream>
using namespace std;

int main()
{
  const int DIAS = 7;            //número de dias de cada string
  const int MAX  = 10;           //tamanho máximo de cada string

  char str1[DIAS][MAX] = { "Domingo", "Segunda-feira", "Terça-feira",
                           "Quarta-feira", "Quinta-feira", "Sexta-feira",
                           "Sábado" };

  for(int i=0; i<DIAS; i++) cout <<str1[i]<<endl;  //exibe cada string

  return 0;
}
```

7.4– Classe de string padrão em C++.

7.4.7- Matrizes de *strings*

No código anterior, a primeira dimensão do vetor DIAS, informa quantos caracteres estão no vetor. A segunda dimensão, MAX, especifica o comprimento máximo das strings (9 caracteres para "*Wednesday*" mais o nulo final faz 10). A Figura 7.9 mostra o arranjo

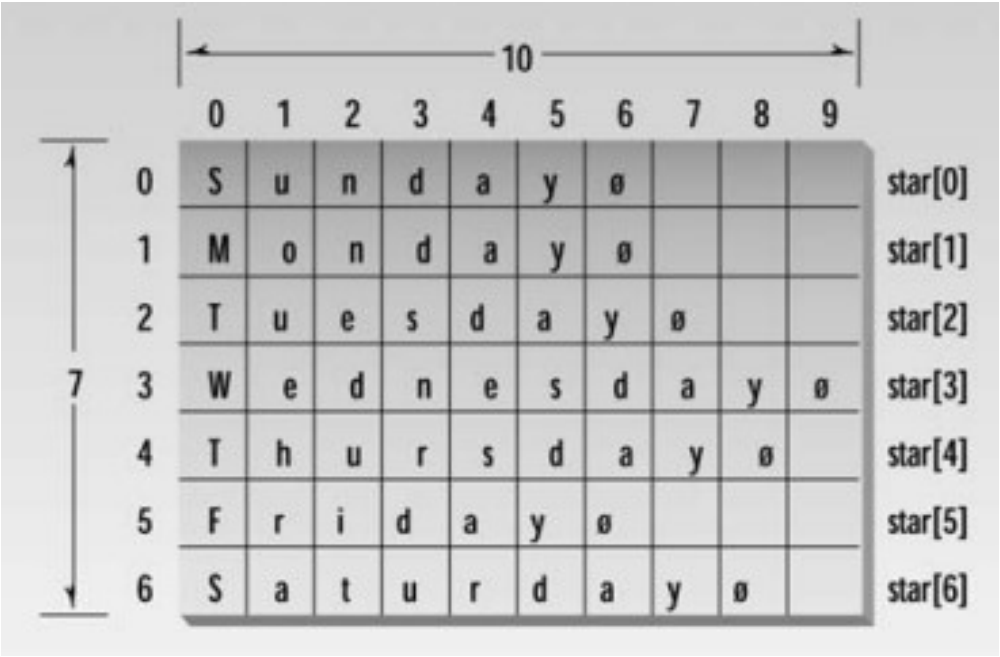

Figura 7.9- Matriz de strings.

A sintaxe para acessar uma determinada *string* pode parecer surpreendente:

star[j];

7.4– Classe de string padrão em C++.

7.4.7- Matrizes de *strings*

e estamos lidando com uma matriz bidimensional, onde está o segundo índice? Como uma matriz bidimensional é um vetor de vetor, pode-se acessar elementos do vetor "externo", cada um dos que é um vetor (neste caso uma *string*), individualmente.

Para fazer isso, não precisa-se do segundo índice. Então **star[j]** é o número da *string j* na matriz de *strings*.

7.4.8– *String* como membros de classe.

As *strings* frequentemente aparecem como membros de classes. O próximo exemplo, uma variação de um programa do Capítulo 6, usa uma *string* C para contar o nome da parte do *widget*.

Este programa define dois objetos da classe *peca* e dá a eles valores com a função membro ***define_peca()***. Em seguida, ele os exibe com a função de membro ***mostra_peca().***

7.4.8– *String* como membros de classe.

Prog7_15.cpp – *Strings* usada como objeto de ***peca*** de ferramenta.

```
#include <iostream>
#include <cstring>
using namespace std;
//-------------------------------------------------------------
class peca
{
 private:
    char pecanome[40];              //nome da peça
    int  pecanro;                   //numero ID da peça
    double custo;                   //custo da peça
 public:
    void define_peca(char pnome[], int pn, double c)
    {
     strcpy(pecanome,pnome);
     pecanro = pn;
     custo   = c;
    }
    void mostra_peca()
    {
     cout<<"\nNome: " <<pecanome;
     cout<<", Numero: " <<pecanro;
     cout<<", Custo=$ " <<custo;
    }
};
```

7.4.8– *String* como membros de classe.

Prog7_15.cpp – *Strings* usada como objeto de ***peca*** da ferramenta (continuação).

```
int main()
{
  peca p1, p2;                          //define os objetos

  p1.define_peca("parafuso de punho", 245, 217.55)    //define as peças
  p2.define_peca("alavanca de partida", 923, 4091.25)

  cout <<"\n Primeira parte: "; p1.mostra_peca();     //exibe as peças
  cout <<"\n Segunda parte: ";  p2.mostra_peca();
  cout <<endl;
  return 0;
}
```

A saída:
Primeira parte:
Nome = parafuso de punho, numero = 245, custo = $ 217,55
Segunda parte:
Nome = alavanca de partida, número = 923, custo = $ 4091,25

Na função de membro ***define_peca()***, usa-se a função de biblioteca de strings ***strcpy()*** para copiar o *strin*g do argumento *pecanome* para o membro de dados da classe *pecanome*. Assim, esta função serve ao mesmo propósito com variáveis de *string* que uma instrução de atribuição faz com variáveis simples.

7.5– A classe de *String* C++ padrão

O C++ padrão inclui uma nova classe chamada *string*. Esta classe melhora a *string* C tradicional de várias maneiras. Por um lado, você não precisa mais se preocupar em criar uma matriz de tamanho certo para armazenar variáveis de *string*. A classe *string* assume toda a responsabilidade pelo gerenciamento da memória. Além disso, a classe *string* permite o uso de operadores sobrecarregados, para que você possa concatenar objetos string com **o operador +:**

s3 = s1 + s2

Existem outros benefícios também. Esta nova classe é mais eficiente e segura de usar, do que as *strings* eram em C . Na maioria das situações, é a abordagem preferida. Nesta seção, examinaremos a classe *string* e suas várias funções de membro e operadores.

7.5.1– Definição e atribuição de objetos de *string*

Pode-se definir um objeto *string* de várias maneiras:

- Usar um construtor sem argumentos, criando uma *string* vazia. Você também pode
- Usar um construtor de um argumento, onde o argumento é uma constante C-*string*; ou seja, caracteres delimitados por aspas duplas.

Como em nossa classe *String* caseira, objetos da classe *string* podem ser atribuídos uns aos outros com um simples operador de atribuição.

7.5.1– Definição e atribuição de objetos de string

Prog7_16.cpp – Definindo e atribuindo objetos de strings.

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  string s1(" Carro");                    //inicializa a string
  string s2 = "bonito";                   //inicializa a string
  string s3;

  s3 = s1;                                //atribuição
  cout <<"s3= "<< s3 <<endl;

  s3 = " Nem " + s1 + " nem feio";        //concatenação
  s3 += s2;                               //concatenação
  cout <<"s3= "<< s3 <<endl;

  s1.swap(s2);                            //troca s1 e s2
  cout << s1 <<"nem" << s2 << endl;
  return 0;
}
```

O operador sobrecarregado + concatena um objeto *string* com outro.
A declaração *s3* = “Nem “ + s1 + “ nem feio, “;coloca a *string* “Nem Carro nem feio“ na variável *s3*.

7.5.1– Definição e atribuição de objetos de *string*

Pode-se também pode usar o operador += para anexar uma string ao final de uma string existente. A declaração

s3 += s2;

acrescenta *s2*, que é "bonito", ao final de *s3*, produzindo a string "Nem Carro nem bonito" e atribuindo-a a *s3*.

Apresenta-se a primeira função membro da classe *string*: ***swap()***, que troca os valores de dois objetos *string*. É a chamada para um objeto com o outro como argumento. Aplica-se a **s1 ("Carro")** e **s2 ("bonito")** e, em seguida, exibimos seus valores para mostrar que **s1** agora é "**bonito**" e **s2** é agora **"Carro".**

O **operador +** sobrecarregado concatena um objeto string com outro. A declaração

s3 = "Nem " + s1 + " nem feio ";coloca a string "Nem Carro nem feio " na variável s3.

7.5.2– Entrada e saída com objetos de *string*

A entrada e a saída são tratadas de maneira semelhante à das *strings* C. Os operadores << ***and*** >> são sobrecarregado para lidar com objetos *string*, e uma função ***getline()*** lida com entradas que contêm espaços em branco ou várias linhas incorporadas. O exemplo Prog7_17.cpp mostra como isso se parece.

Prog7_17- Entrada e saída de classe *string*.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string nome_completo, apelido, endereco;
  string saudacao("Olá");
  cout << "Digite o seu nome completo"
  getline(cin, nome_completo);
  cout << "Seu nome completo é: " << nome_completo << endl;

  cout << "Digite o seu apelido: ";
  cin >> apelido;

  saudacao+=apelido;
  cout << saudacao << endl;

  cout <<"Digite o seu endereço em linhas separadas\n";
  cout <<"Terminando com  '$'\n";
  getline(cin, endereco, '$');

  cout << "Seu endereço é:" << endereco << endl;
  return 0;
}
```

7.5.3– Encontrar objetos de *string*

A classe *string* inclui uma variedade de funções de membro para localizar *strings* e *substrings* em objetos *string*. O exemplo Prog7_18.cpp mostra alguns deles.

Prog7_18- Entrada e saída de classe string.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int ;
  string  s1 = "O rei de Kuala publicou um decreto que determina a felicidade";
  n =s1.find ("Kuala");
  cout << "Encontrado Kuala em " << n << endl;

  n =s1.find_first_of ("qdaf");
  cout << "Primeiro de qdaf " << n << endl;

  n =s1.find_first_of ("AEIOUaeiou");
  cout << "Primeiras consoantes em " << n << endl;

  return 0;
}
```

A **função *find_first_of()*** procura qualquer um de um grupo de caracteres e retorna a posição do primeiro que encontrar. Aqui ele procura por qualquer um dos grupos ‘*q*', ‘*d*', ‘*a*' ou ‘*f*

7.5.3– Encontrar objetos de *string*

Uma função similar **find_first_not_of()** encontra o primeiro caractere em sua string que não pertence a um grupo especificado.
Aqui o grupo consiste em todas as vogais, maiúsculas e minúsculas, então a função encontra a primeira consoante, que é a segunda letra.

A saída do programa é:
Encontrado *Kuala* em 9
Primeiro de *qdaf* em 6
Primeira *consoante* em 1

Existem variações em muitas dessas funções que não demonstramos aqui, como**:**
rfind(), que varre sua string para trás;
find_last_of(), que encontra o último caractere que corresponde a um grupo de caracteres, e
find_last_not_of()**,** o oposto do anterior.

Todas essas funções retornam **–1** se o objetivo não for alcançado.

7.5.4– Modificar objetos de string

Existem várias maneiras de modificar objetos *string*. Nosso próximo exemplo mostra as funções-membro tais como: ***erase()***, ***replace()*** e ***insert()*** .

A função ***erase()*** remove uma *substring* de uma *string*. Seu primeiro argumento é a posição do primeiro caractere na *substring* e o segundo é o comprimento da *substring*.

A função ***insert()*** insere uma *string* especificada por seu segundo argumento no local especificado por seu primeiro argumento.

7.5.4– Modificar objetos de string

Prog7_19- Modifica partes de um objeto *string*.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int ;
  string  s1 = "Rápido! Enviado para Conde Feliciano.";
  string  s2("Senhor");
  string  s3("Não");
  s1.replace(0,8);                          //remove "Rápido! "
  s1.replace(10,5,s2);                      //substitui "Conde" por "Senhor"
  s1.replace(0, 1, "s");                    //substitui 'E' por 'e'
  s1.insert(0, s3);                         //insere "Não " no início
  s1.erase(s1.size()-1,1);                  //remove '.'
  s1.append(3, '!');                        //adiciona '!'

  int n =s1.find (' ');                     //encontre um espaço
  while(n<s1.size())                        //executa o laço enquanto
      {                                     //existirem espaços
        s1.replace(n, 1, "/");              //substitui por /
        n = s1.find(' ');                   //encontre o próximo espaço
      }
  cout << "s1: " << s1 << endl;
  return 0;
}
```

7.5.4– Modificar objetos de string

No exemplo, ***erase()***, ele remove "Rápido" do início da *string*. A função ***replace()*** substitui parte da string por outra *string*. O primeiro argumento é a posição onde a substituição deve começar, o segundo é o número de caracteres na string original a ser substituída e o terceiro é a string de substituição. Aqui "Conde" é substituído por "Senhor".
No exemplo, ***replace()***, aqui ele insere "Não" no início de s1.

O segundo uso de ***erase()*** emprega a função de membro *size()*, que retorna o número de caracteres no objeto string. A expressão *size()-1* é a posição do último caractere, o ponto, que é apagado. A função ***append()*** instala três pontos de exclamação no final da frase. Nesta versão da função, o primeiro argumento é o número de caracteres a serem anexados e o segundo é o caractere a ser anexado.

No final do programa, mostra-se, em um laço ***while***, procura-se o caractere de espaço ' ' usando ***find()*** e substitua cada um por uma barra usando ***replace()***.

A saída: **s1: Não /enviado/para/Senhor/Feliciano!!!**

7.5.5– Acessar caracteres em objetos de string

Pode-se acessar caracteres individuais dentro de um objeto *string* de várias maneiras. Também pode usar o operador sobrecarregado [ ], que faz com que o objeto *string* pareça um vetor. No entanto, o operador [ ] não avisa se você tentar acessar um caractere que está fora dos limites (além do final da string, por ex.). O operador [ ] se comporta dessa maneira com vetores reais e é mais eficiente. No entanto, pode levar a erros de programa difíceis de diagnosticar. É mais seguro usar a **função *at()*,** que faz com que que o programa pare se você usar um índice fora dos limites.

Neste programa **Prog7_20** usa-se ***at()*** para exibir todos os caracteres em um objeto *string*, caractere por caractere. O argumento para ***at()*** é a localização do caractere na *string*. Em seguida, mostra-se como pode-se usar a função membro ***copy()*** para copiar um objeto *string* em um vetor do tipo char, transformando-o efetivamente em uma string C. Após a cópia, um caractere nulo('\0') deve ser inserido após o último caractere na matriz para concluir a transformação em uma *string* C.

A função membro ***length()*** de string retorna o mesmo número que ***size()***.

7.5.5– Acessar caracteres em objetos de string

Prog7_20 – Acessando caracteres em objetos de string

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  char chvetor[80];
  string  palavra;
  cout <<"Digite uma palavra";
  cin >> palavra;
  int pcomp = palavra.length();       //comprimento do objeto string

  cout<<"Um caracter por vez: ";
  for(int i=0; i<pcomp; i++)
      cout <<palavra.at(i);          //excepção se fora dos limites
//    cout <<palavra.at[i];          //sem aviso se fora dos limites

  palavra.copy(chvetor, pcomp, 0);   //copia o objeto string para o array
  chvetor[pcomp] = 0;                //termina com '\0'
  cout << "\n O array contem: " << chvetor << endl;
  return 0;
}
```

7.5.5.1– Outras funções de string

Viu-se que ***size()*** e ***length()*** retornam o número de caracteres atualmente em um objeto *string*.

A quantidade de memória ocupada por uma string é geralmente um pouco maior do que realmente precisava para os caracteres. (Embora, se não tiver sido inicializado, use 0 bytes para caracteres.). A função de membro ***capacity()*** retorna a memória real ocupada.

Pode-se adicionar caracteres à string sem fazer com que ela expanda sua memória até que esse limite seja atingido.

A função membro ***max_size()*** retorna o tamanho máximo possível de um objeto *string*. Essa quantidade corresponde ao tamanho das variáveis ***int*** em seu sistema, menos 3 bytes.

Em sistemas Windows de 32 bits, são 4.294.967.293 bytes, mas o tamanho de sua memória provavelmente restringir esse valor.