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 - Classe de string padrão em C++.

Introdução

Arrays (vetores) são como estruturas, pois ambos agrupam vários itens em uma unidade maior. Mas enquanto uma estrutura geralmente agrupa itens de tipos diferentes, uma matriz agrupa itens do mesmo tipo. Mais importante, os itens em uma estrutura são acessados por nome, enquanto aqueles em uma matriz são acessados por um número de índice. Usar um número de índice para especificar um item permite acesso fácil a um grande número de itens.

Os *arrays* existem em quase todas as linguagens de computação. Arrays em C++ são semelhantes aos de outras linguagens e idênticos aos de C. Neste capítulo, veremos primeiro *arrays* de tipos de dados básicos, como *int* e *char*. Em seguida, examinaremos os *arrays* usados como membros de dados em classes e os arrays usados para armazenar objetos. Assim, este capítulo pretende não apenas introduzir *arrays*, mas também aumentar sua compreensão da programação orientada a objetos. Neste capítulo estudar-se-ão duas abordagens diferentes para *strings*, que são usadas para armazenar e manipular texto. O primeiro tipo de *string* é um vetor do tipo *char*, e o segundo é um membro da classe de *string* Standard (padrão) C++.

7.1– Fundamentos de matrizes

Um exemplo de programa simples servirá para introduzir matrizes. Este programa, ***Prog7_1.cpp***, cria uma matriz de quatro inteiros representando as idades de quatro pessoas. Em seguida, ele pede ao usuário para inserir quatro valores, que são colocados na matriz. Finalmente, ele exibe os quatro valores.

Prog7_1.cpp – Lê quatro idades e mostra-as na tela.

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

int main()
{
 int idade[4];              //vetor de idades de 4 posições

 for(int j=0;j<4; j++)      //obtém 4 idades
    {
      cout <<"Digite uma idade: ";
      cin>>idade[j];        //acessa os elementos do vetor
    }
  for(int j=0;j<4; j++)      //mostra as 4 idades
      cout <<"Idade digitada: " <<idade[j]<<endl;
 return 0;
}
```

O primeiro ciclo ***for*** obtém as idades do usuário e as coloca na matriz, enquanto o segundo ciclo ***for*** as lê da matriz e exibe-as.

7.1.1– Definição de matrizes

Como outras variáveis em C++, um *array (vetor, matriz)* deve ser definido antes que possa ser usado para armazenar informações. E, como outras definições, uma definição de matriz especifica um tipo de variável e um nome. Mas inclui outro recurso: um tamanho. *O tamanho especifica quantos itens de dados a matriz conterá.* Ele segue imediatamente o nome e é cercado por colchetes. A Figura 7.1 mostra a sintaxe de uma definição de *array*. No exemplo *Prog7_1*, a matriz é do tipo *int*. O nome do vetor vem em seguida, seguido imediatamente por um colchete de abertura, o tamanho do vetor e um colchete de fechamento. O número entre colchetes deve ser uma constante ou uma expressão que resulta em uma constante e também deve ser um número inteiro. No exemplo usamos o valor 4.

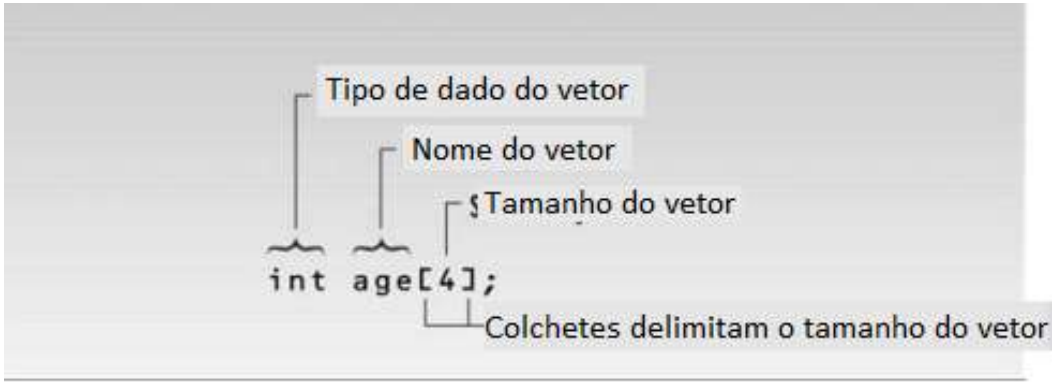

Figura 7.1- Sintaxe para definição de um vetor.

7.1.2– Elementos de matrizes

Os *itens* em uma matriz são chamados de *elementos* (em contraste com os itens de uma *estrutura*, que são chamados de *membros*). Como observamos, todos os elementos em um vetor são do mesmo tipo; apenas os valores variam. A Figura 7.2 mostra os elementos do vetor idade armazenados na memória.

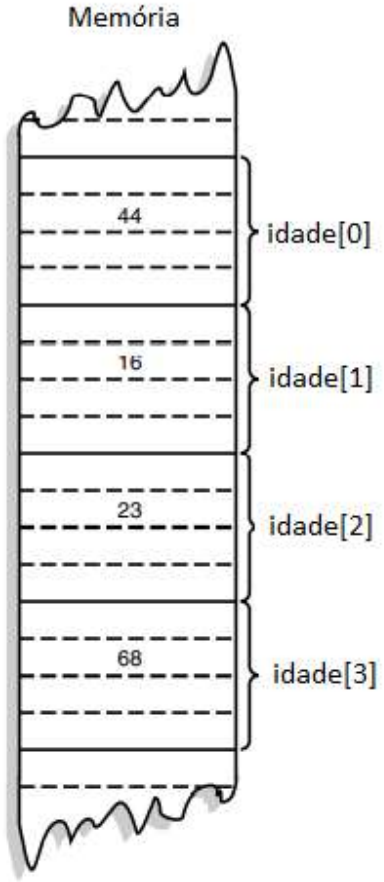

Figura 7.2- Elementos de um vetor.

7.1.3– Acessando os elementos de um vetor

No exemplo **Prog7_1**, acessamos cada elemento do array duas vezes. Na primeira vez, inserimos um valor no vetor , com a linha

cin >> idade[j];

Na segunda vez, lê-se (escreve-se) os elementos do vetor com a instrução

cout << "\n Idade digitada " << idade[j];

Em ambos os casos, a expressão para o elemento do *array* é **idade[j]**. Este consiste no nome do *array*, seguido por colchetes que delimitam uma variável ***j***. Qual dos quatro elementos da matriz é especificado por essa expressão depende do valor de ***j***; idade[0] refere-se ao primeiro elemento, idade[1] ao segundo, idade[2] ao terceiro e idade[3] ao quarto. *A variável* (ou *constante*) entre colchetes é chamada de *índice da matriz*.

Como ***j*** é a variável do ciclo de repetição em ambos os ciclos ***for***, ela começa em ***0*** e é incrementada de um, até atingir 3, acessando assim cada um dos elementos do vetor um por vez.

7.1.4– Média dos elementos de um vetor

Aqui está outro exemplo de funcionamento de um vetor. Este, **Prog7_2**, instrui o usuário a inserir uma série de cinco valores representando as vendas de ferramentas para cada dia da semana (exceto sábado e domingo), e então calcula a média desses valores. Usa-se um array do tipo *double* para que os valores monetários possam ser inseridos.

Prog7_2 .cpp - Calcula a média das vendas de ferramentas em 6 dias.

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

int main()
{
  const int tamanho =5; double total = 0, media;
  double vendas [tamanho]     //define vetor vendas de 5 posições

  cout <<" Insira as vendas para os 5 dias\n ";
  for(int j=0;j<tamanho; j++)
    {
      cin>>vendas[j];          //acessa aos elementos do vetor vendas
      total+=vendas[j];
    }
    media=total/tamanho;
      cout <<"A média =  " << media <<endl;
 return 0;
}
```

7.1.5 – Inicialização dos elementos de um vetor

Pode-se atribuir valores a cada elemento da matriz quando a matriz é definida pela primeira vez. Aqui está um exemplo Prog7_3, que define 12 elementos de matriz na matriz *dias_por_mes* para o número de dias em cada mês.

Prog7_3 .cpp – Mostra os dias desde o inicio do ano até a data especificada.

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

int main()
{
  int dia, mes, total_dias; const int MesAno=12;
  int dias_por_mes[MesAno] = { 31, 28, 31, 30, 31,30,
                               31, 31, 30, 31, 30,31 };
  cout <<"\n Digite o mes (1 a 12): ";              // obter data
  cin >> mes;
  cout <<"\n Digite o dia (1 a 31): ";              // obter data
  cin >> dia;

  total_dias = dia;                                 // dias separados

  for(int j=0; j<mes-1; j++)                        //adicionar dias a cada mes

      total_dias+=dias_por_mes[j];
   cout <<" O total de dias do inicio do ano é: " << total_dias <<endl;

 return 0;
}
```

7.1.5 – Inicialização dos elementos de um vetor

O programa calcula o número de dias desde o início do ano até uma data especificada pelo usuário. (**Cuidado:** não funciona para **anos bissextos**.)
Aqui estão alguns exemplos de interação:
Digite o mes (1 a 12) : 3
Digite o dia (1 a 31): 11
O total de dias desde o início do ano é: 70

Depois de obter os valores do mês e do dia, o programa primeiro atribui o valor do dia à variável *total dias*. Em seguida, ele percorre um *loop*, onde adiciona valores da matriz *dias_por_mes* a *total_dias*. O número desses valores a serem adicionados é um a menos que o número de meses. Por exemplo, se o usuário inserir o mês 5, os valores dos quatro primeiros elementos da matriz (31, 28, 31 e 30) serão adicionados ao total.

Os valores para os quais *dias_por_mes* é inicializado são cercados por chaves e separados por vírgulas. Eles estão conectados à expressão de matriz por um sinal de igual. A Figura 7.3 mostra a sintaxe.

7.1.5 – Inicialização dos elementos de um vetor

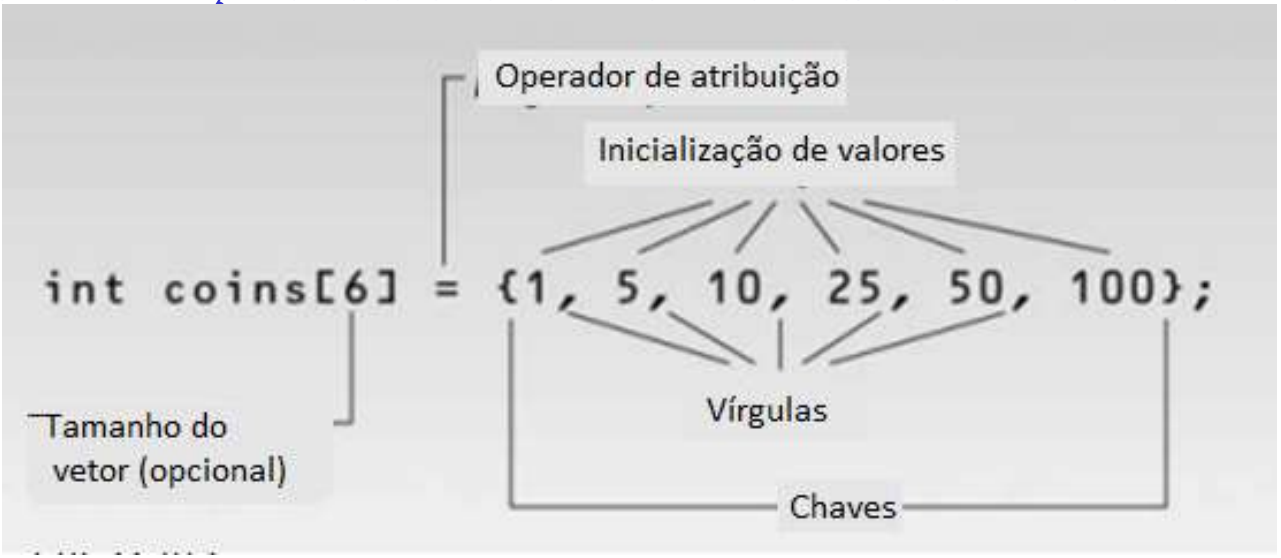

Figura 7.3 – Sintaxe de inicialização de um vetor.

Na verdade, não precisa-se usar o tamanho do vetor quando inicializam-se todos os elementos do vetor, pois o compilador pode descobrir contando as variáveis de inicialização. Assim pode-se escrever
int dias_por_mes[] = { 31, 28, 31, 30, 31, 30,31, 31, 30, 31, 30, 31};

O que acontece se usar-se um tamanho de vetor explícito, mas não concordar com o número de inicializadores? Se houver poucos inicializadores, os elementos ausentes serão definidos como 0. Se houver muitos, um erro será sinalizado.

7.1.6 – Matrizes multidimensionais

Até agora vimos vetores de uma dimensão: uma única variável especifica cada elemento do vetor. Mas matrizes podem ter maiores dimensões. Aqui está um programa, **Prog7_4** , que usa uma matriz para armazenar números de vendas para vários distritos e vários meses.

Prog7_4 .cpp – Exibe o gráfico de vendas usando matriz bidimensional.

7.1.6 – Matrizes multidimensionais

Prog7_4 .cp nsional.

```
#include <iostream>
#include <iomanip>                                  // para definir precisão
using namespace std;
const int ESTADOS = 4;                             // dimensões dos vetores
const int MESES =3;
int main()
{
  int e,m;
  double vendas [ESTADOS][MESES];                    //definição da matriz
bidimensional
   cout <<endl;
  for(e=0, e<ESTADOS; e++)                         //obtém os dados na matriz
      {
      for(m=0;m<MESES; m++)
         {
           cout <<"\n Digite as vendas por estados " <<e+1;
           cout <<", mes "<<m+1 <<":";
           cin >> vendas[e][m];                     //escreve os dados na matriz
         }

  cout <<"\n\n";
  cout <<"                        MES\n";
  cout <<"                 1      2      3";

   for(e=0, e<ESTADOS; e++)                        //obtém os dados na matriz
      {
      cout <<"\nESTADOS "<<e+1;
      for(m=0;m<MESES; m++)                          //mostra os valores na
matriz
         {
           cout << setiosflags(ios::fixed)         //não na forma exponencial
                << setiosflags(ios::showpoint)     //sempre use ponto flutuante
                << setprecision(2)                 //digitos a direita
                << setw(10)                        //largura do campo
                << vendas[e][m];                   //mostra osvalores da matriz
         }
      }     // corresponde end for(e)
 cout <<endl;  return 0;
}
```

7.1.7 – Definição de matrizes multidimensionais

A matriz é definida com dois especificadores de tamanho, cada um entre colchetes:

double vendas [ESTADOS][MESES];

Você pode pensar em *vendas* como uma matriz bidimensional, disposta como um tabuleiro de xadrez. É uma matriz de elementos ESTADOS, cada um dos quais é uma matriz de elementos MESES. A Figura 7.4 mostra como isso se parece. Claro, que pode-se ter matrizes de mais de duas dimensões.

Uma matriz tridimensional é uma matriz de matrizes de matrizes. Ele é acessado com três índices:
elem = dimen3[x][y][z];

Isso é totalmente análogo a matrizes unidimensionais e bidimensionais.

7.1.7 – Definição de matrizes multidimensionais

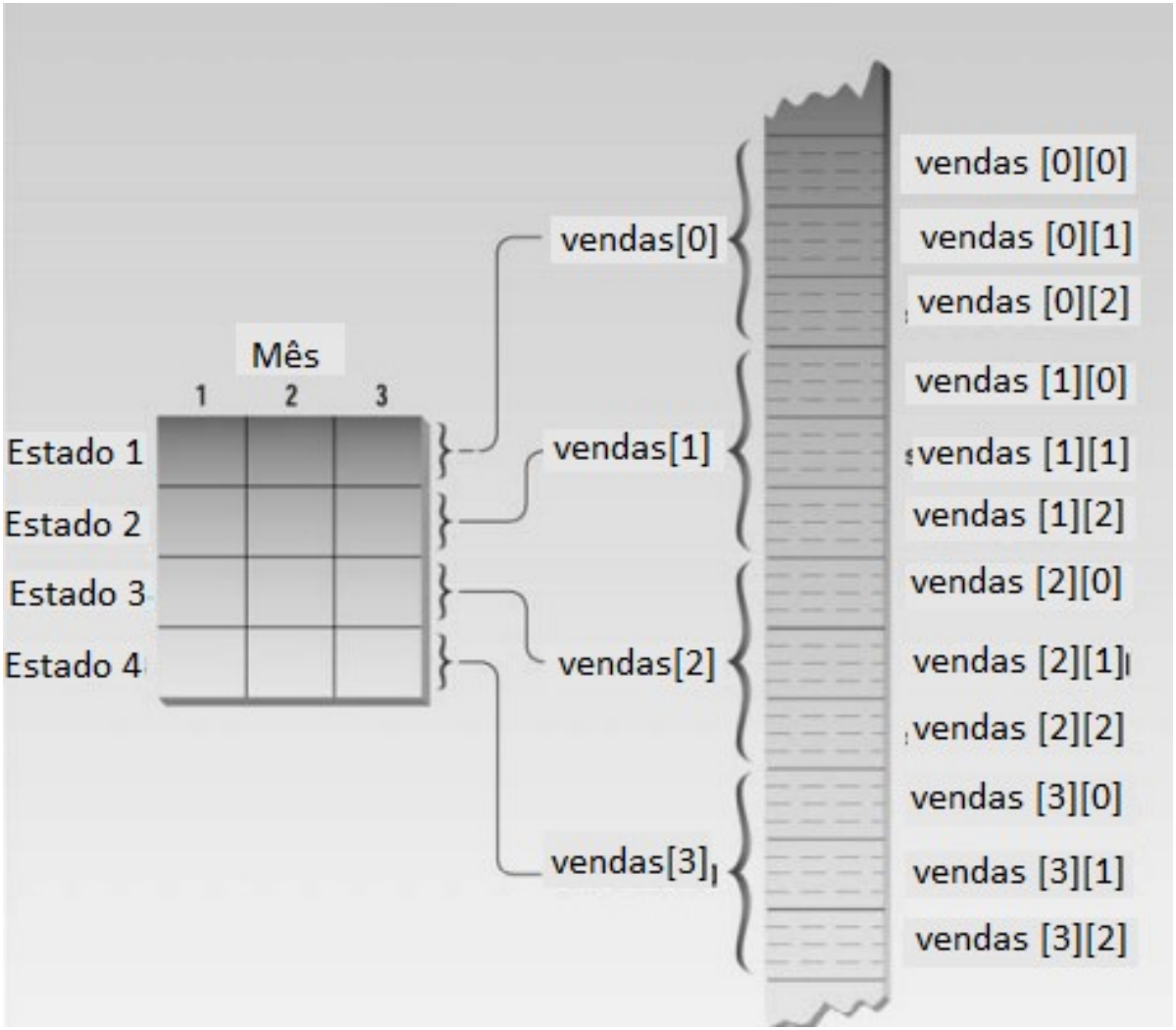

Figura 7.4 – Matriz bidimensional.

7.1.8 – Acesso aos elementos da matriz multidimensional

Elementos de matriz em matrizes bidimensionais requerem dois índices:

vendas[d][m].

Observe que cada índice tem seu próprio conjunto de colchetes. As vírgulas não são usadas. **Não escreva vendas[d,m];** isso funciona em algumas linguagens, mas **não em C++.**

7.1.9 – Inicialização de matrizes multidimensionais

É obvio que pode-se inicializar matrizes multidimensionais. O único pré-requisito é a vontade de digitar um monte de chaves e vírgulas. Aqui **está parte do** programa **Prog7_4.cpp** que usa uma matriz inicializada em vez de solicitar ao usuário a entrada de dados.

```
double vendas [ESTADOS][MESES] = {  {  1432.07,   234.50,   654.01 },
                                    {   322.00, 13838.32, 17589.88 },
                                    {  9328.34,   934.00,  4492.30 },
                                    { 12838.29,  2332.63,    32.93 }  };
```

7.1.10 – Passagem de matrizes como argumento de funções

As matrizes podem ser usados como argumentos para funções. A seguir apresenta-se um exemplo, de um programa, que passa a matriz de números de vendas para uma função cuja finalidade é exibir os dados como uma tabela. Aqui está a listagem do **Prog7_5.cpp**.

Prog7_5.ccp: Passa matriz com parâmetro da função.

```
#include <iostream>
#include <iomanip>                                  // para definir precisão
using namespace std;

const int ESTADOS = 4;
const int MESES =3;
void mostra (double [ESTADOS][MESES]);              // declaração da função
int main()
{
  //inicialização da matriz bidimensional
  double vendas [ESTADOS][MESES]
          = {  {  1432.07,   234.00,     654.09},
               {   332.00, 12234.48,   17654.01},
               {  8902.89,   234.55,    4654.77},
               { 21432.07,  4423.75,      14.96}  };

  mostra(vendas);                                   //chamada da função, usando a
matriz como
                                                    // argumento.
  cout <<endl;
  return 0;
}  // fim da função main
//--------------------------------------------------------------------------
```

7.1.10 – Passagem de matrizes como argumento de funções

Prog7_5.ccp: Passa matriz com parâmetro da função (continuação).

```
// função para mostrar a matriz 2-d passada como argumento da função
void mostra (double funvendas[ESTADOS][MESES])
{
  int e, m;
  cout <<"\n\n";
  cout <<"                          MES\n";
  cout <<"                 1      2      3";

   for(e=0, e<ESTADOS; e++)
      {
      cout <<"\nESTADOS "<<e+1;
      for(m=0;m<MESES; m++)
         {
           cout << setiosflags(ios::fixed) << setw(10)
                << setiosflags(ios::showpoint) << setprecision(2)
                << funvendas[e][m];                    //mostra os elementos da
matriz
         }
      }     // corresponde end(fim) do for(e)
}           // fim da função mostra
```

Declaração de uma função com argumentos de matriz

Em uma declaração de função, os argumentos de matriz são representados pelo tipo de dados e tamanho da matriz. Aqui está a declaração da função ***mostra()***:

void mostra(double [ESTADOS][MESES]); // declaração

7.1.10 – Passagem de matrizes como argumento de funções

Declaração de uma função com argumentos de matriz

Na verdade, há uma informação desnecessária aqui. A seguinte declaração também funciona:

void mostra(double[][MONTHS]); // declaração

Por que a função não precisa do tamanho da primeira dimensão? Novamente, lembre-se de que uma matriz bidimensional é um vetor de vetores. A função primeiro interpreta o argumento como uma matriz de *estados*. Ele não precisa saber quantos distritos existem, mas precisa saber quão grande é cada elemento de *estados*, para que possa calcular onde um determinado elemento está.

Se estivéssemos **declarado** uma função que usasse um vetor unidimensional como argumento, não precisaríamos usar o tamanho do vetor:

void somefunc(int elem[]); // declaração

7.1.10 – Passagem de matrizes como argumento de funções

Chamada de função com argumentos de matriz.

Quando a função é chamada, apenas o nome do vetor é usado como argumento.
display(vendas); // chamada de função
Este nome (*vendas* neste caso) representa na verdade **o endereço de memória do *array***. Não vamos explorar os endereços em detalhes até ao capítulo sobre, "Ponteiros". Usar um endereço para um argumento de matriz é semelhante a usar um argumento de referência, pois os valores dos elementos de matriz não são duplicados (copiados) na função.

Em vez disso, a função trabalha com o vetor original, embora se refira a ele por um nome diferente. Este sistema é usado para vetores porque eles podem ser muito grandes; duplicar uma matriz inteira em cada função que a chamasse consumiria tempo e desperdiçaria memória.

No entanto, um endereço não é o mesmo que uma referência. Nenhum e comercial (**&**) é usado com o nome da matriz na declaração da função. Até discutirmos os ponteiros, acredite que os vetores são passados usando apenas seu nome e que a função acessa o vetor original, não uma duplicata (ou seja, um vetor duplicado).

7.1.10 – Passagem de matrizes como argumento de funções

Definição da função com argumentos de matriz.

Na definição da função, o declarador se parece com isso:

void mostra (double funvendas [ESTADOS][MESES])

O argumento *array* usa o *tipo* de dados, um *nome* e os *tamanhos das dimensões* do *array*. O nome do *array* usado pela função (*funvendas* neste exemplo) pode ser diferente do nome que define o array (*vendas*), mas ambos se referem ao mesmo *array*. Todas as dimensões da matriz deve ser especificadas (exceto em alguns casos o primeiro); a função precisa deles para acessar os elementos da matriz corretamente.

Referências a elementos da matriz na função usam o nome da função para a matriz:

funvendas[d][m]

Mas de todas as outras formas a função pode acessar os elementos do *array* como se o vetor tivesse sido definido em uma função.

7.1.11 – Matrizes de estruturas

Matrizes podem conter estruturas, bem como tipos de dados simples. Aqui está um exemplo baseado na estrutura da peça do Capítulo 4.
O usuário digita o número do modelo, o número da peça e o custo de uma peça. O programa registra esses dados em uma estrutura. No entanto, essa estrutura é apenas um elemento em uma matriz de estruturas. O programa solicita os dados de duas (2) partes diferentes e os armazena nos dois elementos da matriz separada. Em seguida, exibe as informações. Aqui estão alguns exemplos de entrada:

```
Digite o número do modelo: 45
Digite o número da peça: 498
Digite o preço:  127,98

Digite o número do modelo: 66
Digite o número da peça: 4280
Digite o preço: 3387,55

Modelo 45   Peca 498    Custo 127,98
Modelo 66   Peca 4280   Custo 3387,55
```

A matriz de estruturas é definida na instrução:
peca apart [TAMANHO];

Isso tem a mesma sintaxe que as matrizes de tipos de dados simples.

7.1.11 – Matrizes de estruturas

Prog7_6.cpp

```
#include <iostream>
using namespace std;
const int TAMANHO = 4;                              // número de peças

//-------------------------------------------------------------------------
struct peca
{
  int nromodelo;
  int nropeca;
  float custo;
};
//--
int main()
{  int n;
   peca apart[TAMANHO];                             //define a matriz de
estruturas
   for(m=0;m<TAMANHO; m++)                          //obtém os valores para todos
os membros
         {
           cout << endl;
           cout << "Digite o número do modelo:";
           cin  >> apart[m].nromodelo;
           cout << "Digite o número da peça:";
           cin  >> apart[m].nropeca;
           cout << "Digite o preço:";
           cin  >> apart[m].custo;
         }
    cout <<endl;
   for(m=0;m<TAMANHO; m++)                      //mostra os valores de todos os
membros
         {
           cout << "Modelo:" << apart[m].nromodelo;
           cout << "Peça:"   << apart[m].nropeca;
           cout << "Custo:"  << apart[m].custo;
         }
  cout <<endl;
  return 0;
}
```

7.1.11 – Matrizes de estruturas

Apenas o nome do tipo, *peca*, mostra que este é uma matriz de um tipo mais complexo.
Acessar um item de dados que é membro de uma estrutura que é um elemento de uma matriz envolve uma nova sintaxe. Por exemplo:

apart[n].nromodelo

refere-se ao membro da estrutura *nromodelo* que é o elemento *n* da matriz *peca*. A Figura 7.5 mostra como isso se parece.

Matrizes de estruturas são um tipo de dados útil em uma variedade de situações.

Mostrou-se para uma série de peças de carros, mas também podemos armazenar uma série de dados pessoais (nome, idade, salário), uma série de dados geográficos sobre cidades (nome, população, altitude) e muitos outros tipos de dados.

7.1.11 – Matrizes de estruturas

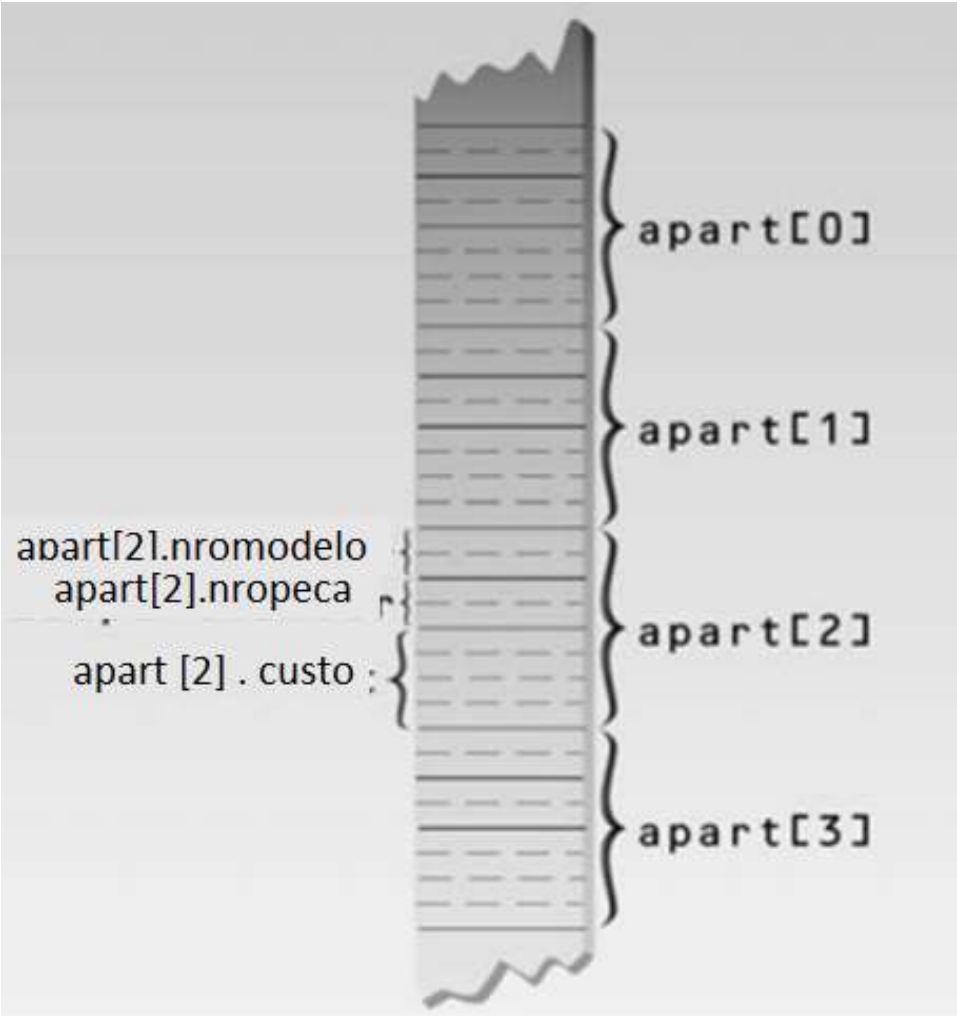

Figura 7.5 –Matriz de estruturas