11

Ponteiros

Os ponteiros são vistos pela maioria dos programadores como um dos tópicos mais difíceis de C++. Existem várias razões para isso. Primeiro, os conceitos embutidos em ponteiros podem ser novos para muitos programadores, visto que não são comuns em linguagens de alto nível. Segundo, os símbolos usados para notação de ponteiros em C++ não são tão claros quanto poderiam ser. Por exemplo, o mesmo símbolo é usado para duas finalidades diferentes. Conceitualmente, os ponteiros podem ser difíceis, mas nada tão complicado.

Ponteiros em C++ são similares a ponteiros em C. Nosso objetivo neste capítulo é o de expor tão claramente quanto possível como trabalhar com ponteiros.

O QUE SÃO PONTEIROS?

Ponteiro é um endereço de memória. Seu valor indica em que parte da memória do computador uma variável está alocada, não o que está nela armazenado. Um ponteiro proporciona um modo de acesso a uma variável sem referi-la diretamente (modo indireto).

POR QUE OS PONTEIROS SÃO USADOS?

Para dominar a linguagem C++, é essencial dominar ponteiros. Eles são utilizados em situações em que o uso do nome de uma variável não é permitido ou é difícil ou indesejável. Algumas razões para o uso de ponteiros são:

- manipular elementos de matrizes;
- receber argumentos em funções que necessitem modificar o argumento original;
- passar strings de uma função para outra; usá-los no lugar de matrizes;
- criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro;
- alocar e desalocar memória do sistema.
- passar para uma função o endereço de outra.

Ponteiros Variáveis

Em C++ há um tipo especial de variável concebida para conter o endereço de outra variável; é o *ponteiro variável.*

O ponteiro variável armazena um endereço de memória. Esse endereço é a localização de uma outra variável. Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda.

Endereços de Memória

A memória de seu computador é dividida em bytes, e os bytes são numerados de zero até o limite de memória de sua máquina. Esses números são chamados de *endereços* de bytes. Um endereço é a referência que o computador usa para localizar variáveis.

Toda variável ocupa uma certa localização na memória, e seu endereço é o do primeiro byte ocupado por ela.

Nossos programas, quando carregados para a memória, ocupam uma certa parte dela. Dessa maneira, toda variável e toda função de nosso programa começam em um endereço particular, denominado endereço da variável ou da função.

O Operador de Endereços &

Para conhecermos o endereço ocupado por uma variável, usamos o operador de endereços &.

Ele é um operador unário, e seu operando deve ser o nome de uma variável. O resultado é o seu endereço. Veja um pequeno programa que mostra o seu uso:

```
//Mostra o uso do operador de endereços
#include <iostream>
using namespace std;

int main()
{
      int i,j,k;
      cout <<  "Endereço de i = " << &i << endl;
      cout <<  "Endereço de j = " << &j << endl;
      cout <<  "Endereço de k = " << &k << endl;
      system("PAUSE");
      return 0;
}
```

Veja a saída:

```
Endereço de i = 0012FED4
Endereço de j = 0012FEC8
Endereço de k = 0012FEBC
```

Esse programa declara três variáveis inteiras e imprime o endereço de cada uma delas. O endereço ocupado por essas variáveis dependerá de vários fatores como o tamanho do sistema operacional, se há ou não outros programas residentes na memória etc. Por essas razões, você pode encontrar números diferentes quando executar o programa. O operador << imprime endereços em hexadecimal.

Não confunda o operador de endereços, o qual precede o nome de uma variável, com o operador de referências (&), que segue o nome do tipo de uma variável, usado para criar referências. A falta de símbolos, ocasionada pela grande quantidade de operadores em C++, faz vários operadores diferentes terem o mesmo símbolo.

O operador de endereço (&) só pode ser usado com nomes de variáveis. Construções tais como:

```
&(i+1)        //ERRADO
&5            //ERRADO
```

são incorretas.

Revisão: Passando Argumentos por Referência

Antes de começarmos, vamos rever o uso do operador de referência e a passagem de argumentos para funções por referência.

O operador de referência cria outro nome para uma variável já existente. Toda operação em qualquer dos nomes tem o mesmo resultado. A referência não é uma cópia da variável à que se refere. É a mesma variável sob diferentes nomes.

Quando argumentos são passados por valor, a função chamada cria novas variáveis e copia nelas os valores passados. Dessa maneira, a função não tem acesso às variáveis originais da função chamadora, portanto não poderá modificar seus conteúdos.

Quando argumentos são passados por referência, a função recebe um outro nome para as variáveis originais e poderá modificar os seus conteúdos. Esse mecanismo possibilita que uma função retorne mais de um valor para a função chamadora. Os valores a serem retornados são colocados em referências de variáveis da função chamadora.

Verifique novamente o exemplo do uso simples de referência como argumento de uma função:

```
//Referencias.Cpp
//Mostra passagem de argumentos por referência
void reajusta20( float& preco, float& reajuste); //protótipo
#include <iostream>
using namespace std;

int main()
{
      float val_preco, val_reaj;
      do
```

```
        {
                cout << "Insira o preço atual: ";
                cin  >> val_preco;
                reajusta20(val_preco,val_reaj);//função recebe por referência
                cout << "\n0 preço novo é " << val_preco << endl;
                cout << "0 aumento foi de " << val_reaj << endl;
        } while( val_preco != 0.0);
        system("PAUSE");
        return 0;
}

//reajusta20()
//Reajusta o preço em 20%
void reajusta20(float& preco, float& reajuste)
{
        reajuste = preco * 0.2;
        preco *= 1.2;
}
```

O programa solicita ao usuário que insira o preço atual de uma certa mercadoria. Em seguida, modifica o preço para um valor reajustado em 20% e calcula o valor do aumento.

Funções que recebem argumentos por referência utilizam o operador & somente na declaração do tipo do argumento.

Observe que a chamada a uma função que recebe uma referência é idêntica à chamada às fúnções em que os argumentos são passados por valor.

A declaração

```
float& preco, float& reajuste
```

indica que **preco** e **reajuste** são outros nomes para as variáveis passadas como argumento pela função **main()**. Em outras palavras, quando usamos **preco** e **reajuste**, estamos realmente usando **val_preco** e **val_reaj** de **main()**.

Passando Argumentos por Referência com Ponteiros

Há três maneiras de passar argumentos para uma função: por valor, por referência e por ponteiro. Se uma função deseja alterar variáveis da função chamadora, as variáveis não podem ser passadas por valor.

Em situações como essa, podemos usar referências ou ponteiros. Para usarmos ponteiros como argumentos, devemos seguir dois passos. Primeiro, a função chamadora, em vez de passar valores para a função chamada, passa endereços, usando o operador de endereços. Os endereços são de variáveis que poderão ser alteradas, onde queremos colocar novos valores. Segundo, a função chamada deverá criar variáveis para armazenar os endereços que estiver recebendo, enviados pela função chamadora. Essas variáveis são ponteiros.

Vamos modificar o programa anterior para utilizar ponteiros em vez de referências. Veja a alteração:

```
//Ponteiros.Cpp
//Mostra passagem de argumentos por ponteiros
void reajusta20( float *preco, float *reajuste); //protótipo
#include <iostream>
using namespace std;

int main()
{
      float val_preco, val_reaj;
      do
      {
            cout << "Insira o preço atual: ";
            cin  >> val_preco;
            reajusta20(&val_preco, &val_reaj); //Enviando endereços
            cout << "\n0 preço novo é " << val_preco << endl;
            cout << "O aumento foi de " << val_reaj << endl;
      } while( val_preco != 0.0);
      system("PAUSE");
      return 0;
}

//reajusta20()
//Reajusta o preço em 20%
void reajusta20(float *preco, float *reajuste) //Recebendo ponteiros
{
      *reajuste = *preco * 0.2;
      *preco *= 1.2;
}
```

O programa faz exatamente o mesmo que a versão original.

A função **reajusta20()** recebe como argumento dois ponteiros para **float**. O seu protótipo é o seguinte:

```
void reajusta20( float *preco, float *reajuste); //protótipo
```

VARIÁVEIS QUE ARMAZENAM ENDEREÇOS

Você já usou variáveis para armazenar números inteiros, caracteres, números em ponto flutuante etc. Os endereços são armazenados de modo semelhante em variáveis com um tipo específico para esse fim. A variável que armazena um endereço é chamada de ponteiro.

Observe que um ponteiro para um **float** não é do tipo **float**, mas sim do tipo **float** *. O asterisco faz parte do nome do tipo e indica Ponteiro para.

A declaração dos argumentos da função **reajusta20()** indica que ***preco** e ***reajuste** são do tipo **float** e que **preco** e **reajuste** são ponteiros para variáveis **float**.

Na verdade, usar ***preco** é uma maneira indireta de usar a variável **val_preco** de **main()**. Toda alteração feita em ***preco** afetará diretamente **val_preco**. Como **preco** contém o endereço da variável **val_preco**, dizemos que **preco** aponta para **val_preco**.

O Operador Indireto (*)

O operador indireto (*) é unário e opera sobre um endereço ou ponteiro. O resultado da operação é o nome da variável localizada nesse endereço (apontada). O nome da variável representa seu valor ou conteúdo. Em outras palavras, resulta o valor da variável apontada.

Passando Endereços para a Função

A nossa função **main()** chama a função **reajusta20()** por meio da seguinte instrução:

```
reajusta20(&val_preco, &val_reaj); //Enviando endereços
```

Os endereços das variáveis **val_preco** e **val_reaj** são acessados por meio do operador de endereços e enviados como argumentos para a função **reajusta20()**.

A função **reajusta20()** recebe esses valores em ponteiros (tipos de variáveis para armazenar endereços). Para que a função possa acessar as variáveis apontadas, deverá não somente conhecer seus endereços, como também seus tipos. O endereço informa o primeiro byte ocupado pela variável; é necessário saber quantos bytes estão ocupados por ela e qual é a forma de armazenamento para se ter acesso ao seu conteúdo.

O tipo da variável apontada é dado na declaração do ponteiro:

```
float *preco, float *reajuste
```

Assim, **float** é a variável apontada por **preco**. A função pode então usar instruções como:

```
*reajuste = *preco * 0.2;
*preco *= 1.2;
```

que modificam indiretamente as variáveis **val_preco** e **val_reaj**. É indireto porque a função não tem acesso direto aos nomes dessas variáveis, e sim aos seus endereços. A conclusão é que **main()** acessa diretamente as variáveis **val_preco** e **val_reaj**, enquanto **reajusta20()** as acessa via operador indireto.

Uma vez conhecidos os endereços e os tipos das variáveis do programa chamador, a função pode não somente colocar valores nessas variáveis como também ler o valor já armazenado nelas. Ponteiros podem ser usados não apenas para que a função passe valores para o programa chamador, mas também para que o programa chamador passe valores para a função.

Na instrução

```
*reajuste = *preco * 0.2;
```

estamos usando o valor original da variável **val_preco**, por meio de ***preco**, multiplicando este valor por 0.2 e atribuindo o resultado à variável **val_reaj** indiretamente por meio de ***reajuste**.

PONTEIROS SEM FUNÇÕES

O exemplo que apresentamos usou ponteiros como argumentos de funções. O programa seguinte cria ponteiros como variáveis automáticas dentro de funções.

```
//PtrVar.Cpp
//Mostra o uso de ponteiros declarados dentro da função
#include <iostream>
using namespace std;

int main()
{
      int x=4, y=7;
      cout << "&x = " << &x << "\t  x = " << x << endl;
      cout << "&y = " << &y << "\t  y = " << y << endl;

      int *px,*py;
      px = &x;
      py = &y;
      cout << "px = " << px << "\t*px = " << *px << endl;
      cout << "py = " << py << "\t*py = " << *py << endl;

      system("PAUSE");
      return 0;
}
```

Eis a saída:

```
&x = 0012FED4      x = 4
&y = 0012FEC8      y = 7
px = 0012FED4    *px = 4
py = 0012FEC8    *py = 7
```

A instrução

```
int *px,*py;
```

declara px e py ponteiros para variáveis **int**. Quando um ponteiro não é inicializado na instrução de sua declaração, o compilador inicializa-o com o endereço zero (NULL). C++ garante que NULL não é um endereço válido, então antes de usar os ponteiros devemos atribuir a eles algum endereço válido; isto é feito pelas instruções:

```
px = &x;
py = &y;
```

Um ponteiro pode ser inicializado na mesma instrução de sua declaração:

```
int *px = &x, *py = &y;
```

Observe que estamos atribuindo &x e &y a px e py, respectivamente, e não a *px e *py. O asterisco, na declaração, faz parte do nome do tipo.

C++ permite ponteiros de qualquer tipo, e as sintaxes de declaração possíveis são as seguintes:

- Operador indireto junto do nome da variável.

```
int  *p;     //Ponteiro int
char *p;     //Ponteiro char
String *p;   //Ponteiro para um tipo
             //definido pelo usuário
```

- Operador indireto junto do nome do tipo.

```
int*   p;    //Ponteiro int
char*  p;    //Ponteiro char
String* p;   //Ponteiro para um tipo
             //definido pelo usuário
```

- Se vários ponteiros são declarados em uma mesma instrução, o tipo deve ser inserido somente uma única vez; o asterisco, todas as vezes.

```
int * p, * p1, * p2;
int *p, *p1, *p2;
```

- Inicializando ponteiros.

```
int i;
int *pi=&i;
int *pj, *pi=&i, *px;
```

- Ponteiros e variáveis simples declarados em uma única instrução.

```
int *p, i, j, *q;
int *px=&x, i, j=5, *q;
```

PONTEIROS E VARIÁVEIS APONTADAS

Você pode usar ponteiros para executar qualquer operação na variável apontada.

```
//PtrVar1.Cpp
//Mostra o uso de ponteiros declarados dentro da função
#include <iostream>
using namespace std;

int main()
{
      int x,y;
      int *px=&x;  //Inicializa px com o endereço de x

      *px=14;      //O mesmo que x=14
      y = *px;     //O mesmo que y=x

      cout << "y = " << y << endl;

      system("PAUSE");
      return 0;
}
```

Nesse programa, usamos o ponteiro para atribuir um valor à variável x; em seguida, usamos novamente o ponteiro para atribuir este valor a y. O operador indireto, precedendo o nome do ponteiro em instruções como

```
      y = *px;     //O mesmo que y=x
```

resulta o valor da variável apontada.

OPERAÇÕES COM PONTEIROS

C++ permite várias operações básicas com ponteiros. Nosso próximo exemplo mostra essas possibilidades. O programa imprime os resultados de cada operação, o valor do ponteiro, o valor da variável apontada e o endereço do próprio ponteiro.

```
//PtrOperacoes.Cpp
//Mostra as operações possíveis com ponteiros
#include <iostream>
using namespace std;

int main()
{
      int x=5, y=6;
      int *px, *py;

      px = &x;     //Atribuições
```

```
        py = &y;

        if( px < py)//Comparações
              cout << "py-px= " << (py-px) << endl;//Subtração
        else
              cout << "px-py= " << (px-py) << endl;

        cout << "px = " << px;
        cout << ", *px = " << *px;                    //Operador Indireto
        cout << ", &px = " << &px << endl;     //Operador de Endereços

        cout << "py = " << py;
        cout << ", *py = " << *py;
        cout << ", &py = " << &py << endl;

        py++; //Incremento

        cout << "py = " << py;
        cout << ", *py = " << *py;
        cout << ", &py = " << &py << endl;

        px = py + 5; //Somar inteiros

        cout << "px = " << px;
        cout << ", *px = " << *px;
        cout << ", &px = " << &px << endl;

        cout << "px-py= " << (px-py) << endl;

        system("PAUSE");
        return 0;
}
```

A saída é:

```
px-py= 3
px = 0012FED4, *px = 5,  &px = 0012FEBC
py = 0012FEC8, *py = 6,  &py = 0012FEB0
py = 0012FECC, *py = 28, &py = 0012FEB0
px = 0012FEE0, *px = 46, &px = 0012FEBC
px-py= 5
```

ATRIBUIÇÃO

Um endereço pode ser atribuído a um ponteiro desde que o ponteiro seja do mesmo tipo do endereço. Geralmente fazemos isso usando o operador de endereços (&) junto ao nome de uma variável. No nosso exemplo, atribuímos a px o endereço de x e a py o endereço de y.

```
px = &x;        //Atribuições
py = &y;
```

Operação Indireta

O operador indireto (*) precedendo o nome do ponteiro resulta no nome da variável apontada.

Trazendo o Endereço do Ponteiro

Como todas as variáveis, os ponteiros têm um endereço e um valor. O operador &, precedendo o nome do ponteiro, resulta na posição de memória onde o ponteiro está localizado.

O nome do ponteiro indica o valor contido nele, isto é, o endereço para o qual ele aponta (o endereço da variável apontada).

Incrementando um Ponteiro

Podemos incrementar um ponteiro por meio de adição regular ou do operador de incremento. Incrementar um ponteiro acarreta sua movimentação para o próximo tipo apontado; se **px** é um ponteiro para uma variável **int**, depois de executar a instrução

```
px++;
```

o valor de **px** será incrementado de um **int** (quatro bytes em ambientes de 32 bits). Cada vez que **px** é incrementado, apontará para o próximo **int** da memória. A mesma idéia é verdadeira para decrementos. Toda vez que **px** é decrementado de uma unidade, será decrementado de um tipo apontado, neste caso **int**.

Você pode subtrair ou adicionar números de e para ponteiros. A instrução

```
px = py + 3;
```

fará que **px** caminhe três inteiros adiante de **py**.

Diferença

Você pode encontrar a diferença entre dois ponteiros. Esta diferença será expressa em número de variáveis apontadas entre eles. Então, se **py** tem o valor 4000 e **px** o valor 3996, a expressão

```
py-px
```

resultará 1 quando **px** e **py** forem ponteiros para **int** em ambientes de 32 bits.

Comparações entre Ponteiros

Testes relacionais com >, <, >=, <=, == ou != são aceitos somente entre ponteiros do mesmo tipo. Cuidado, se você comparar ponteiros que apontam para variáveis de tipos diferentes, obterá resultados sem sentido.

Testes com **NULL** ou zero também podem ser feitos, contanto que você use o modificador de tipo:

```
if(px == (int *)0)
```

A Unidade Adotada em Operações com Ponteiros

Quando declaramos um ponteiro, o compilador necessita conhecer o tipo da variável apontada para poder executar corretamente operações aritméticas.

```
int *pi;
double *pd;
float *pf;
```

O tipo declarado é entendido como o tipo da variável apontada. Assim, se somarmos 1 a **pi**, estaremos somando o tamanho de um **int**; se somarmos 1 a **pd**, somaremos o tamanho de um **double** (oito bytes) e assim por diante.

A unidade com ponteiros é o número de bytes do tipo apontado.

Ponteiros no Lugar de Matrizes

Em C++, o relacionamento entre ponteiros e matrizes é tão estreito que ponteiros e matrizes deveriam ser realmente tratados juntos.

O compilador transforma matrizes em ponteiros, pois a arquitetura do microcomputador compreende ponteiros, e não matrizes. Qualquer operação que possa ser feita com índices de uma matriz pode ser realizada com ponteiros.

No Capítulo 6 do livro *Treinamento em linguagem C++, Módulo 1*, aprendemos que o nome de uma matriz representa seu endereço de memória. Esse endereço é o do primeiro elemento da matriz. Em outras palavras, o nome de uma matriz é um ponteiro que aponta para o primeiro elemento da matriz.

Para esclarecermos a relação entre ponteiros e matrizes, vamos examinar um simples programa escrito primeiramente com matrizes e depois com ponteiros.

```
//Matriz.Cpp
//Imprime os elementos de uma matriz
#include <iostream>
using namespace std;
```

```
int main()
{
      int M[5]={92,81,70,69,58};

      for(int i=0; i<5; i++)
            cout << M[i] <<endl; //Notação matriz

      system("PAUSE");
      return 0;
}
```

O próximo exemplo usa a notação ponteiro:

```
//PMatriz.Cpp
//Imprime os elementos de uma matriz usando notação ponteiro
#include <iostream>
using namespace std;

int main()
{
      int M[5]={92,81,70,69,58};

      for(int i=0; i<5; i++)
            cout << *(M + i) <<endl; //Notação ponteiro

      system("PAUSE");
      return 0;
}
```

A expressão *(M+i) tem exatamente o mesmo valor de M[i]. Você já sabe que M é um ponteiro **int** e aponta para M[0], e conhece também a aritmética com ponteiros. Assim, se somarmos 1 a M, obteremos o endereço de M[1]; M+2 é o endereço de M[2] e assim por diante. Em regra geral, temos que:

M + i	é equivalente a	**&M[i]**,	portanto
***(M + i)**	é equivalente a	**M[i]**	

PONTEIROS CONSTANTES E VARIÁVEIS

Analisando o exemplo anterior, você poderia perguntar: será que a instrução

```
cout << *(M + i) <<endl; //Notação ponteiro
```

não poderia ser simplificada e substituída por

```
cout << *(M ++) <<endl; //Erro
```

A resposta é não. A razão disso é que não podemos incrementar uma constante. Da mesma forma que existem inteiros constantes e inteiros variáveis, existem ponteiros constantes e ponteiros variáveis.

O nome de uma matriz é um ponteiro constante.

Isso vale para qualquer constante. Você não poderia escrever:

```
x = 3++;      //ERRADO
```

O compilador apresentaria um erro.

O nome de uma matriz é um ponteiro constante e não pode ser alterado. Um ponteiro variável é um lugar na memória que armazena um endereço. Um ponteiro constante é um endereço, uma simples referência.

Vamos reescrever **PMatriz.Cpp** usando um ponteiro variável em vez do nome da matriz.

Veja a listagem:

```
//PMatriz.Cpp
//Imprime os elementos de uma matriz usando ponteiro variável
#include <iostream>
using namespace std;

int main()
{
      int M[5]={92,81,70,69,58};
      int *p=M; //Cria e inicializa o ponteiro variável

      for(int i=0; i<5; i++)
            cout << *(p++) <<endl; //Notação ponteiro

      system("PAUSE");
      return 0;
}
```

Nessa versão, definimos um ponteiro para um **int** e o inicializamos com o nome da matriz:

```
int *p=M;
```

Agora podemos usar **p** em todo lugar do programa que usa **M**, e, como **p** é um ponteiro variável e não uma constante, podemos usar expressões como:

```
*(p++)
```

O ponteiro p contém inicialmente o endereço do primeiro elemento da matriz &M[0]. Para acessar o próximo elemento, basta incrementar p de 1. Após o incremento, p aponta para o próximo elemento, &M[1], e a expressão *(p++) representa o conteúdo desse segundo elemento. O laço for faz a expressão acessar cada um dos elementos em ordem.

PASSANDO MATRIZES COMO ARGUMENTO PARA FUNÇÕES

No Capítulo 6 do livro *Treinamento em linguagem C++, Módulo 1*, apresentamos numerosos exemplos de como matrizes são passadas como argumentos para funções e como as funções podem acessar seus elementos.

Quando uma função recebe o endereço de uma matriz como argumento, ela o declara usando o nome do tipo e colchetes ([]). Essa notação declara ponteiros constantes, e não ponteiros variáveis. Entretanto, é mais conveniente usar a notação ponteiro no lugar da notação matriz. A notação ponteiro declara um ponteiro variável.

O próximo exemplo modifica o programa **Media.Cpp** para que use ponteiros no lugar de matriz:

```
//PMedia.Cpp
//Mostra passagem de matrizes para funções usando ponteiros
#include <iostream>
#include <iomanip>
using namespace std;
const TAMANHO=50;
float media(float *lista, int tamanho);//protótipo
int main()
{
      float notas[TAMANHO];
      int i=0;
      cout << setprecision(4); //número de dígitos a serem impressos
      do
      {
            cout << "Digite a nota do aluno " << (i + 1) << " : ";
            cin  >> *(notas+i);
      } while( *(notas + i++) >= 0.0);

      i--;

      float m = media( notas, i );
      cout << "Média das notas: " << m << endl;

      system("PAUSE");
      return 0;
}

//Calcula a média dos valores da matriz
```

```
float media(float *lista, int tamanho)
{
      float m=0.0;
      for(int i=0; i < tamanho ; i++) m += *(lista++);
      return  m/tamanho ;
}
```

Observe primeiramente o protótipo da função **media()**.

```
float media(float *lista, int tamanho);//protótipo
```

A declaração:

```
float *lista
```

é equivalente à original

```
float lista[]
```

A primeira declara um ponteiro variável; a segunda, um ponteiro constante.

Como o nome da matriz é um endereço, não usamos o operador de endereços (&) na instrução de chamada à função:

```
float m = media( notas, i );
```

PONTEIROS E STRINGS

Strings são matrizes do tipo **char**. Dessa forma, a notação ponteiro pode ser aplicada.

Como primeiro exemplo, vamos escrever uma função que procura um caractere em uma cadeia de caracteres. Esta função retorna o endereço da primeira ocorrência do caractere, se este existir, ou o endereço zero, caso o caractere não seja encontrado.

```
//StrProcura.Cpp
//Procura um caractere numa cadeia de caracteres
#include <iostream>
#include <cstdio>
using namespace std;

char * procura(char *s, char ch);//protótipo

int main()
{
      char str[81], *ptr;

      cout << "Digite uma frase:\n" << endl;
      gets(str);

      ptr = procura(str, 'h');
```

```
    cout << "\nA frase começa no endereço "
         << reinterpret_cast<unsigned *>(str) << endl;

    if(ptr)
    {
        cout << "\nPrimeira ocorrência do caractere 'h': "
             << reinterpret_cast<unsigned *>(ptr) << endl;
        cout << "\nA sua posição é: "
             << (ptr-str) << endl;
    } else
        cout << "O caractere 'h' não existe nesta frase."<< endl;

    system("PAUSE");
    return 0;
}

//Procura um caractere numa frase
char *procura(char *s, char ch)
{
    while( *s != ch && *s != '\0') s++;
    if(*s != '\0') return s;
    return (char *)0;
}
```

Veja a saída:

```
Digite uma frase:

O seu aniversário será comemorado hoje à noite.

A frase começa no endereço 0012FE7C

Primeira ocorrência do caractere 'h': 0012FE9E

A sua posição é: 34
```

Primeiramente vamos analisar o protótipo da função **procura**().

```
char * procura(char *s, char ch);//protótipo
```

A função retorna um ponteiro. As funções podem tanto receber ponteiros como argumentos, e podem também retornar um ponteiro, desde que declarados de acordo.

O Operador reinterpret_cast<>

Como fazer para imprimir o endereço da matriz str? Quando o objeto cout recebe como argumento um ponteiro char, imprime o texto armazenado com base nesse endereço. Ou seja,

```
cout << str;
```

imprime a frase digitada.

Poderíamos pensar em mudar o tipo apontado para unsigned. O operador static_cast<> não opera com ponteiros.

```
cout << static_cast<unsigned *>(str);//ERRO, não opera ponteiros
```

O operador reinterpret_cast<> pode ser usado para modificar o tipo apontado por um ponteiro:

```
cout << reinterpret_cast<unsigned *>(str); //Ok.
```

Funções de Biblioteca para Manipulação de Strings

Você já usou várias funções de biblioteca para manipular strings. Essas funções utilizam ponteiros. Apesar de existirem prontas na biblioteca C++, escreveremos algumas delas para que você aprenda e crie outras conforme suas necessidades.

```
//Protótipos
int strlen(char *);
void strcpy(char *dest, char *orig);
int strcmp(char *s, char *t);
```

Veja a listagem das funções:

```
//Retorna o tamanho da cadeia
int strlen(char *s)
{
      int i=0;
      while(*(s++)) i++;
      return i;
}

//Copia a cadeia origem na cadeia destino
void strcpy(char *dest, char *orig)
{
      while(*(dest++) = *(orig++));
}

//Compara a cadeia s com a cadeia t
```

```
//Retorna a diferença ASCII:
//                  um número positivo se s > t
//                  um número negativo se s < t
//                  zero se s == t
int strcmp(char *s, char *t)
{
      while(*s==*t && *s && *t)
      {
            s++;
            t++;
      }
            return *s - *t;
}
```

PONTEIROS PARA UMA CADEIA DE CARACTERES CONSTANTES

Vamos analisar duas maneiras de inicializar cadeias de caracteres constantes: usando um ponteiro constante e usando um ponteiro variável. Observe o exemplo:

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

int main()
{
      char s1[] = "Saudações!";
      char *s2  = "Saudações!";

      cout << s1 << endl;
      cout << s2 << endl;

//    s1++; Erro. Não podemos incrementar uma constante
      s2++; //OK

      cout << s2 << endl; //Imprime: audações

      system("PAUSE");
      return 0;
}
```

O ponteiro constante poderia ser usado em situações em que não queremos alterar o conteúdo da cadeia de caracteres, como, por exemplo, para imprimir, enviar como argumento para uma função etc.

MATRIZES DE PONTEIROS

Matrizes de ponteiros são essencialmente utilizadas para substituir matrizes de duas dimensões em que cada elemento é uma cadeia de caracteres. Esse uso permite uma grande economia de memória, já que não teremos a desvantagem de dimensionar todos os elementos com o tamanho da maior cadeia.

Vamos modificar o programa **DiaSemana.Cpp**, do Capítulo 6 do livro *Treinamento em linguagem C++, Módulo 1*, para que utilize uma matriz de ponteiros.

```
//PDiaSemana.Cpp
//Imprime o dia da semana com base em uma data
//Mostra o uso de uma matriz de ponteiros
#include <iostream>
#include <conio.h> //para getche()
using namespace std;
int dsemana(int dia, int mes, int ano);
int main()
{
      char *diasemana[7]= {"Domingo",
                           "Segunda-feira",
                           "Terça-feira",
                           "Quarta-feira",
                           "Quinta-feira",
                           "Sexta-feira",
                           "Sábado"
                          };
      int dia, mes, ano;
      const char ESC = 27;
      do
      {
            cout << "Digite a data na forma dd mm aaaa: ";
            cin >> dia >>  mes >> ano;
            cout << diasemana [ dsemana(dia,mes,ano)] << endl;
            cout << "ESC para terminar ou ENTER para recomeçar" << endl;
      } while (getch() != ESC);

      system("PAUSE");
      return 0;
}

//Encontra o dia da semana com base em uma data
//Retorna 0 para domingo, 1 para segunda-feira etc.
int dsemana(int dia, int mes, int ano)
{
      int dSemana = ano + dia + 3 * (mes - 1) - 1;
      if( mes < 3)
            ano--;
```

```
    else
        dSemana -= int(0.4*mes+2.3);
    dSemana += int(ano/4) - int((ano/100 + 1)*0.75);
    dSemana %= 7;
    return dSemana;
}
```

Na versão matriz, as cadeias de caracteres são guardadas na memória em 7 posições de 14 bytes cada uma, ou seja, ocupando 98 bytes de memória. Na nova versão, as cadeias são guardadas de forma que ocupem somente o número de bytes necessários para o seu armazenamento. Cada elemento da matriz é um ponteiro, e não mais uma outra matriz.

MATRIZ DE STRINGS E A MEMÓRIA ALOCADA

A versão matriz aloca 98 bytes de memória da seguinte maneira:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	D	o	m	i	n	g	o	\0						
2	S	e	g	u	n	d	a	-	f	e	i	r	a	\0
3	T	e	r	ç	a	-	f	e	i	r	a	\0		
4	Q	u	a	r	t	a	-	f	e	i	r	a	\0	
5	Q	u	i	n	t	a	-	f	e	i	r	a	\0	
6	S	e	x	t	a	-	f	e	i	r	a	\0		
7	S	á	b	a	d	o	\0							

MATRIZ DE PONTEIROS E A MEMÓRIA ALOCADA

A versão ponteiros aloca 79 bytes de memória da seguinte maneira:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	D	o	m	i	n	g	o	\0	S	e	g	u	n	d
1	a	-	f	e	i	r	a	\0	T	e	r	ç	a	-
2	f	e	i	r	a	\0	Q	u	a	r	t	a	-	f
3	e	i	r	a	\0	Q	u	i	n	t	a	-	f	e
4	i	r	a	\0	S	e	x	t	a	-	f	e	i	r
5	a	\0	S	á	b	a	d	o	\0					

A versão ponteiros aloca uma matriz de 7 ponteiros e inicializa cada um deles com o endereço de cada uma das cadeias de caracteres constantes.

ÁREA DE ALOCAÇÃO DINÂMICA: heap

A área de alocação dinâmica — também denominada **heap** — consiste em toda memória disponível que não foi usada para outro propósito. Em outras palavras, o **heap** é simplesmente o resto da memória.

C++ oferece dois operadores que permitem a alocação ou a liberação dinâmica de memória do **heap**: **new** e **delete**.

Alocando e Desalocando a Memória do heap

Suponhamos, por exemplo, que você vá escrever um programa interativo e não conheça de antemão quantas entradas de dados serão fornecidas. A classe **String** é um bom exemplo. Você pode criar uma matriz para armazenar a cadeia de caracteres e reservar uma quantidade de memória que pensa ser razoável, como, por exemplo, para armazenar 80 caracteres. Nesse caso, o compilador alocará a memória necessária para armazenar 80 caracteres toda vez que um objeto da classe **String** for criado.

Isso se torna ineficiente: se não ocuparmos todo o espaço reservado, muita memória poderá ser desperdiçada. Por outro lado, poderá ser necessário armazenar mais caracteres que os 80 reservados, e a classe não atenderá a esse requisito.

A solução para esse tipo de problema é solicitar memória toda vez que se fizer necessário. O mecanismo para aquisição de memória em tempo de execução é por meio do operador **new**.

Antes de exemplificarmos o uso de **new**, vamos primeiro ver como ele opera.

O operador **new** solicita memória ao sistema operacional e retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado. Uma vez alocada, a memória continuará ocupada até que seja desalocada explicitamente pelo operador **delete**. Em outras palavras, uma variável criada com o operador **new** existirá e poderá ser acessada em qualquer parte do programa enquanto não for destruída pelo operador **delete** e seu espaço de memória não for retornado ao sistema operacional.

O próximo exemplo mostra a classe **String** modificada.

Veja a listagem:

```
//NewStr.Cpp
//Mostra o operador new na classe string
#include <iostream>
#include <cstring>
using namespace std;

class String
{
      private:
            char *str;
      public:
            String() //Construtor default
            {
                  str = new char;
                  *str='\0';
            }
```

```
        String(char *s) //Construtor - converte matriz em String
        {
              str = new char[strlen(s) + 1];
              strcpy(str,s);
        }

        ~String()
        {
              if(str) delete[] str;
        }

        void Print() const
        {
              cout << str << endl;
        }
};

int main()
{
      String s = "A vida é uma longa estrada na qual "
                 "corremos contra o tempo.";
      String s1;

      s.Print();
      s1.Print();

      system("PAUSE");
      return 0;
}
```

Veja a saída:

```
A vida é uma longa estrada na qual corremos contra o tempo.
```

A instrução

```
str = new char[strlen(s) + 1];
```

retorna um ponteiro para um bloco de memória do tamanho exato para armazenar a cadeia s mais o '\0'.

A instrução

```
str = new char;
```

reserva um único byte de memória.

Se não houver memória suficiente para satisfazer a exigência da instrução, new devolverá um ponteiro com o valor zero (NULL).

O OPERADOR delete

Quando a memória reservada pelo operador **new** não for mais necessária, deve ser liberada pelo operador **delete**.

Nossa classe devolve a memória alocada ao sistema operacional toda vez que um objeto é destruído. Para tanto, criamos o seguinte destrutor:

```
~String()
{
      if(str) delete[] str;
}
```

Observe que, antes de liberarmos a memória alocada por **new**, devemos verificar se ela foi realmente alocada. Se **str** aponta para o endereço zero, nada é liberado.

OBSERVAÇÕES PARA OS PROGRAMADORES C

Se você é um programador C, reconhecerá, no modo de trabalhar dos operadores **new** e **delete**, uma similaridade com as funções de biblioteca **malloc()** e **free()**. Entretanto, os operadores **new** e **delete** de C++ são superiores em seu modo de operar.

A função **malloc()** retorna um ponteiro genérico (**void**). O tipo desse ponteiro deve ser modificado, por meio do operador de modificação de tipo, para acessar a variável apontada.

O operador **new** retorna um ponteiro já apropriado para o tipo de dado solicitado.

Em C++, **malloc()** não deve ser usada para alocação dinâmica de um objeto, pois supõe-se que um construtor da classe seja chamado na sua criação. Se **malloc()** for usada na criação de um objeto, obteremos um ponteiro para um bloco não inicializado de memória e perderemos o benefício proporcionado por construtores. A função **malloc()** não conhece nada sobre o tipo da variável sendo alocada. Ela toma o tamanho em bytes como argumento e retorna um ponteiro **void**.

Por outro lado, os operadores **new** e **delete** foram concebidos para compreender classes e objetos. Assim, construtores e destrutores serão chamados.

PONTEIROS void

Antes de prosseguirmos a explanação de ponteiros, vamos apresentar um ponteiro peculiar. Quando queremos atribuir um endereço a um ponteiro, esse deve ser do mesmo tipo do endereço. Por exemplo, não podemos atribuir um endereço de uma variável **int** a um ponteiro **float**. Entretanto, há uma exceção. Há um tipo de ponteiro de propósito geral que pode apontar para qualquer tipo de dado. Ele é do tipo **void**, podendo ser declarado pela seguinte instrução:

```
void *p;//p aponta para qualquer tipo de dado
```

Ponteiros do tipo **void** são usados em situações em que seja necessário que uma função retorne um ponteiro genérico e opere independentemente do tipo de dado apontado.

Observe que o conceito de ponteiros **void** não tem absolutamente nada a ver com o tipo **void** para funções.

Qualquer endereço pode ser atribuído a um ponteiro void:

```
int i=5;
float f=3.2;

void *pv;     //Ponteiro genérico
pv = &i;      //Endereço de um int

cout << *pv;//ERRO
pv = &f;     //Endereço de um float

cout << *pv;//ERRO
```

O conteúdo da variável apontada por um ponteiro **void** não pode ser acessada por este ponteiro. É necessário criar outro ponteiro e fazer a conversão de tipo na atribuição:

```
int i=5, *pi;

void *pv;     //Ponteiro genérico

pv = &i;      //Endereço de um int

pi = reinterpret_cast<int *>(pv);

cout << *pi; //OK
```

SOBRECARGA DO OPERADOR DE ATRIBUIÇÃO COM STRINGS

Suponhamos que você use a classe **String** anterior e escreva o seguinte trecho de programa:

```
int main()
{
      String s1("Feliz Aniversário! "),s2;

      s2 = s1;

      ...

}
```

O programa cria o objeto s2 e então atribui a ele o objeto s1. Quando você atribui um objeto a outro, o compilador copia byte a byte todo o conteúdo de memória ocupado pelo objeto no outro.

A classe **String** tem somente um único membro de dado, um ponteiro **str**. O valor desse membro de s1 será copiado no correspondente de s2. Em outras palavras, ao membro str de s2 será atribuído o endereço contido em str de s1 e, como resultado, s2.str e s1.str apontarão para a mesma localização de memória.

Dessa maneira, qualquer modificação na cadeia de caracteres de um dos objetos afetará diretamente o outro. Certamente, não é isso que desejamos.

Um problema mais sério ocorre quando os objetos envolvidos têm escopos diferentes. Por exemplo:

```
int main()
{
      String s1 = "A vida é uma longa estrada na qual "
                  "corremos contra o tempo.";
      {
            String s2;
            s2 = s1;
            s2.Print();
      }

      s1.Print();

      system("PAUSE");
      return 0;
}
```

Quando o bloco mais interno encerrar sua execução, o destrutor será chamado para liberar a memória alocada para s2. Mas **s2.str** aponta para a mesma localização de memória que **s1.str** e o destrutor destruirá a memória alocada e apontada por **s1**.

Esse problema ocorre com qualquer classe que contém um ponteiro como membro e aloca memória dinamicamente.

A solução é impedir que seja executada a atribuição-padrão implementando a nossa sobrecarga do operador de atribuição.

```
void String::operator=(const String& s)
{
      int tamanho=strlen(s.str);
      delete [] str;
      str = new char[tamanho+1];
      strcpy(str,s.str);
}
```

Essa sobrecarga deverá, obrigatoriamente, receber o argumento por referência, pois, caso contrário, ao término da função, será executado o destrutor para o argumento, e o objeto original **s1** será destruído.

Nosso operador de atribuição ainda tem alguns pequenos problemas que você verifica quando deseja executar atribuições múltiplas:

```
s3 = s4 = s1; //ERRO. Atribuição é void
```

A função não poderia ser do tipo **void**.

Outro problema ocorrerá se acidentalmente atribuir um objeto a si mesmo.

```
s1 = s1;
```

Durante a operação de atribuição, nossa função operadora primeiramente libera a memória alocada e apontada pelo ponteiro **str**, em seguida aloca uma nova memória e atribui o endereço dessa nova memória ao ponteiro **str**. Essa nova memória contém inicialmente valores denominados lixo, então a função copia o conteúdo da nova memória nela mesma. Essa operação causa resultados absurdos.

Para reescrevermos o operador de modo consistente, apresentaremos o ponteiro this.

O PONTEIRO this

Para que a função **operator=()** opere de maneira satisfatória, deverá verificar atribuições de um objeto a si mesmo e retornar o próprio objeto apontado. Isso requer o uso de um ponteiro **this**.

O ponteiro **this** pode ser acessado por todos os métodos não estáticos do objeto. Ele aponta sempre para o próprio objeto do qual o método é membro. Ou seja, aponta para o objeto por meio do qual o método foi chamado.

Quando um método é chamado por um objeto, é atribuído o endereço do objeto ao ponteiro **this**.

O método pode usar o ponteiro this para testar quando um objeto passado como argumento é o mesmo do qual a função é membro. Além disso, poderá retornar o objeto apontado por **this**.

A nova versão da função é a seguinte:

```
String& String::operator=(const String& s)
{
      if ( &s == this) return *this;
      int tamanho=strlen(s.str);
      delete [] str;
      str = new char[tamanho+1];
      strcpy(str,s.str);
      return *this;
}
```

this é um Ponteiro Constante

O ponteiro **this** é um ponteiro constante; seu conteúdo não pode ser alterado para que aponte para outro objeto qualquer.

O copy constructor

A nossa classe **String** ainda tem um problema. Observe o seguinte trecho de código:

```
int main()
{
      String s1("Feliz Aniversário! ");
      ChamaFunc( s1); //Chama uma função que recebe String
      ...
}

void ChamaFunc( String x)
{
      ...
}
```

Na chamada à função **ChamaFunc()**, será executado o construtor default de cópia para criar o objeto **x**. Quando a função encerrar, o destrutor será chamado para liberar a memória alocada para **x**. Mas **x.str** aponta para a mesma localização de memória que **s1.str** e o destrutor destruirá a memória alocada e apontada por **s1**.

Para resolvermos esse problema, devemos escrever o nosso próprio **copy constructor**.

```
String::String(const String& s) //Copy constructor
{
      int tamanho=strlen(s.str);
      str = new char[tamanho+1];
      strcpy(str,s.str);
}
```

O Argumento do copy constructor Deve Ser uma Referência

Se você escrever o **copy constructor** com o argumento passado por valor, obterá uma mensagem de "out of memory" ao executar. Quando um argumento é passado por valor, uma cópia dele é construída usando o próprio **copy constructor**. Assim, ele chamará a si próprio novamente e novamente até atingir o fim de memória e o erro ser apresentado. Portanto, o argumento deve ser passado por referência.

O programa de teste completo é o seguinte:

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

```
class String
{
      private:
            char *str;
      public:
            String() //Construtor default
            {
                  str = new char;
                  *str='\0';
            }

            String(char *s) //Construtor - converte matriz em String
            {
                  str = new char[strlen(s) + 1];
                  strcpy(str,s);
            }
            String(const String& s) //Copy constructor
            {
                  int tamanho=strlen(s.str);
                  str = new char[tamanho+1];
                  strcpy(str,s.str);
            }
            String& operator=(const String& s)
            {
                  if ( &s == this) return *this;
                  int tamanho=strlen(s.str);
                  delete [] str;
                  str = new char[tamanho+1];
                  strcpy(str,s.str);
                  return *this;
            }

            ~String()
            {
                  if(str) delete[] str;
            }

            void Print() const
            {
                  cout << str << endl;
            }
};

void ChamaFunc(String);

int main()
```

```
{
      String s1 = "A vida é uma longa estrada na qual "
                  "corremos contra o tempo.";
      {
            String s2; //Objeto com escopo interno
            s2=s1;
            s2.Print();
      }

      ChamaFunc(s1); //Chamada à função

      String s3,s4;

      s3=s4=s1; //Atribuições múltiplas

      s3.Print();
      s1.Print();

      system("PAUSE");
      return 0;
}

void ChamaFunc( String x)
{
    . x.Print();
}
```

DEFININDO O copy constructor E O OPERADOR DE ATRIBUIÇÃO COMO PRIVADOS

Em que situações deveríamos definir o copy constructor e a sobrecarga do operador de atribuição como métodos privados de uma classe?

Imagine uma classe em que seja definido um membro que deve ser individual para cada objeto, não podendo haver dois objetos com o mesmo valor para esse membro.

Nesse caso, não gostaríamos de permitir a cópia de objetos. A solução é escrever os métodos que permitem a cópia como privados, assim o compilador apresentará um erro se o programa tentar inicializar o objeto com outro da mesma classe ou tentar atribuir um objeto a outro da mesma classe.

A definição privada desses métodos também é usada nos casos em que classes alocam memória dinamicamente e não querem permitir a atribuição do ponteiro que aponta para o buffer alocado. Os erros que podem ocorrer com essa atribuição foram descritos anteriormente, na definição da classe String.

```
class String
{
      private:
```

```
        char *str;
        String(const String&);//Copy constructor privado
        String& operator=(const String&);//Atribuição privado
        . . .
};
```

Se você tentar copiar um objeto com instruções como

```
String s1,s2;
String s3(s1);//Erro copy constructor privado

s1=s2;//Erro operador de atribuição privado
```

o compilador informará que as funções estão inacessíveis. Nesse caso, não é necessário definir essas funções, já que nunca serão chamadas.

Operador de Atribuição não Pode Ser Herdado

O operador de atribuição é o único que não pode ser herdado. Se for definida uma sobrecarga do operador de atribuição dentro de uma classe-base, ele não poderá ser utilizado em nenhuma classe derivada desta.

Alocando Tipos Básicos Por Meio de Memória Dinâmica

Os operadores **new** e **delete** podem ser usados para criar variáveis de tipos básicos como inteiros ou caracteres. Por exemplo:

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

int main()
{
      int *pi;

      pi = new int;

      cout << "Digite um número: ";
      cin >> *pi;
      cout << "\nVocê digitou o número " << *pi << endl;

      delete pi;

      system("PAUSE");
      return 0;
}
```

DIMENSIONANDO MATRIZES EM TEMPO DE EXECUÇÃO

A determinação do tamanho de uma matriz pode ser feita em tempo de execução. Vamos modificar o programa **PMedia.Cpp** para que o usuário indique quantas notas serão inseridas.

```
//PDMedia.Cpp
//Alocação dinâmica da matriz
#include <iostream>
#include <iomanip>
using namespace std;
float media(float *lista, int tamanho);//protótipo
int main()
{
      int tamanho;

      cout << "Qual é o número de notas? ";
      cin >> tamanho;

      float * notas;
      notas = new float[tamanho];

      int i;

      cout << setprecision(4); //número de dígitos a serem impressos

      for(i=0; i < tamanho; i++)
      {
            cout << "Digite a nota do aluno " << (i + 1) << " : ";
            cin  >> *(notas+i);
      }

      float m = media( notas, tamanho );
      cout << "Média das notas: " << m << endl;
      delete [] notas;
      system("PAUSE");
      return 0;
}

//Calcula a média dos valores da matriz
float media(float *lista, int tamanho)
{
      float m=0.0;
      for(int i=0; i < tamanho ; i++) m += *(lista++);
      return  m/tamanho ;
}
```

Observe a sintaxe da instrução de liberação da memória alocada: incluímos um par de colchetes vazios antes do nome do ponteiro. O compilador ignora qualquer número colocado dentro dos colchetes.

```
delete [] notas;
```

Dimensionando Matrizes de Duas Dimensões com new

Suponhamos que você queira criar um programa que calcule a média das notas de cada aluno de uma escola. Você sabe que cada aluno tem três notas, mas desconhece o número de alunos da escola. O seguinte trecho de programa mostra como alocar memória em tempo de execução:

```
int (*notas)[3];
int tamanho;

cout << "\nQual é o número de alunos? ";
cint >> tamanho;

notas = new int[tamanho][3];

...

delete [] notas;
```

Você pode alocar uma matriz multidimensional com new, mas somente a primeira dimensão pode ser definida em tempo de execução; as outras devem ser constantes.

Ponteiros para Objetos

Os ponteiros podem apontar para objetos da mesma maneira que apontariam para qualquer tipo básico.

Em muitas situações, no instante em que você está escrevendo um programa, desconhece o número de objetos a serem criados. Nesses casos, você usa new para criar objetos em tempo de execução. O operador new retorna um ponteiro para um objeto. Vamos rever a classe Venda, descrita no Capítulo 8. O objetivo é comparar dois modos de criar objetos.

```
//PtrVenda.Cpp
//Mostra o acesso a funções membros por ponteiros
#include <iostream>
#include <iomanip>
using namespace std;

class Venda
{
```

```
        private:
                int npecas;
                float preco;
        public:
                void GetVenda();
                void PrintVenda() const;
};

void Venda::GetVenda()
{
        cout << "Insira No.Peças: "; cin >> npecas;
        cout << "Insira Preço   : "; cin >> preco;
}

void Venda::PrintVenda() const
{
        cout << setiosflags(ios::fixed)  //não notação científica
             << setiosflags(ios::showpoint)    //ponto decimal
             << setprecision(2)   //duas casas
             << setw(10) << npecas;            //tamanho 10
        cout << setw(10) << preco << endl;
}


int main()
{
        Venda A;
        Venda *B;

        B = new Venda;

        A.GetVenda();
        B->GetVenda();

        A.PrintVenda();
        B->PrintVenda();

        system("PAUSE");
        return 0;
}
```

A função **main()** cria dois objetos da classe **Venda**: o objeto **A** e um segundo objeto apontado pelo ponteiro **B**.

A novidade desse programa é a forma de acesso aos membros de um objeto por meio de seu endereço, e não de seu nome.

Acessando Membros Por Meio de Ponteiros

Você já aprendeu que, se o nome de um objeto for conhecido, podemos acessar seus membros usando o operador ponto (.).

Será que uma construção análoga, usando um ponteiro em vez do nome do objeto, poderia ser escrita? Ou seja, seria possível escrever a construção seguinte?

```
B.GetVenda();//ERRO
```

A resposta é não, pois B não é um objeto, e sim um ponteiro para um objeto, e o operador ponto (.) opera sobre o nome de um objeto.

C++ oferece dois métodos para resolver esse problema: o primeiro, menos elegante, é obter o nome da variável apontada por B por meio do operador indireto (*):

```
(*B).GetVenda();//OK
```

Entretanto, a expressão é de visualização complexa por causa dos parênteses. Os parênteses são necessários, pois o operador (.) tem precedência sobre o operador (*).

O segundo método, de uso mais comum, é pelo operador de acesso a membros (->) que consiste no sinal de "menos" (-) seguido do sinal de "maior que" (>). Esse operador opera sobre o endereço de um objeto, e não sobre seu nome.

```
B->GetVenda();//Mais usado
```

Um ponteiro para um objeto, seguido pelo operador (->) e pelo nome de um membro, trabalha da mesma maneira que o nome de um objeto seguido pelo operador (.) e pelo nome do membro.

O operador de acesso a membros (->) conecta o endereço de um objeto a um membro dele; o operador ponto (.) conecta o nome de um objeto a um membro dele.

Usando Referências

Você pode criar uma referência a um objeto definido pelo operador new. Veja um exemplo:

```
int main()
{
    Venda& A= *(new Venda);
    A.GetVenda();
    A.PrintVenda();
    system("PAUSE");
    return 0;
}
```

A expressão

```
new Venda
```

reserva uma área de memória grande o suficiente para armazenar um objeto da classe **Venda** e retorna o seu endereço. O nome do objeto pode ser referido pela expressão:

```
*(new Venda)
```

Esse é o objeto apontado pelo ponteiro. Criamos uma referência ao objeto de nome A. Desse modo, **A** é o próprio nome do objeto e seus membros podem ser acessados usando o operador ponto em vez do operador ->.

UMA MATRIZ DE PONTEIROS PARA OBJETOS

Quando um programa necessita manipular um grupo de objetos, é mais flexível criar uma matriz de ponteiros para objetos do que uma matriz para agrupar os próprios objetos. Por exemplo, é mais rápido indexar uma matriz de ponteiros para objetos do que indexar os próprios objetos.

Nosso próximo exemplo cria uma matriz de ponteiros para a classe **Venda**.

```
//MPtrVenda.Cpp
//Mostra uma matriz de ponteiros para objetos
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;

class Venda
{
      private:
            int npecas;
            float preco;
      public:
            void GetVenda();
            void PrintVenda() const;
};

void Venda::GetVenda()
{
      cout << "Insira No.Peças: "; cin >> npecas;
      cout << "Insira Preço   : "; cin >> preco;
}

void Venda::PrintVenda() const
{
```

```
    cout << setiosflags(ios::fixed)  //não notação científica
         << setiosflags(ios::showpoint)    //ponto decimal
         << setprecision(2)   //duas casas
         << setw(10) << npecas;            //tamanho 10
    cout << setw(10) << preco << endl;
}


int main()
{
    Venda *p[80];
    int i=0;

    do
    {
        p[i] = new Venda;
        p[i++]->GetVenda();
        cout << "Deseja entrar mais uma venda (s/n)?" << endl;
    } while (getch() != 'n');

    cout << "\nRELATÓRIO DE VENDAS" << endl;
    int j;
    for( j=0 ; j < i ; j++ )
    {
        p[j]->PrintVenda();
        delete p[j];
    }
    system("PAUSE");
    return 0;
}
```

A função **main()** cria uma matriz de 80 ponteiros do tipo **Venda**. O primeiro laço cria um objeto usando **new** e solicita a entrada dos dados de uma venda.

```
p[i] = new Venda;
p[i++]->GetVenda();
```

Em seguida pergunta se o usuário deseja adicionar outra venda ou terminar. O segundo laço imprime os dados dos objetos, um a um, e libera a memória alocada.

```
for( j=0 ; j < i ; j++ )
{
    p[j]->PrintVenda();
    delete p[j];
}
```

CRIANDO UMA LISTA LIGADA

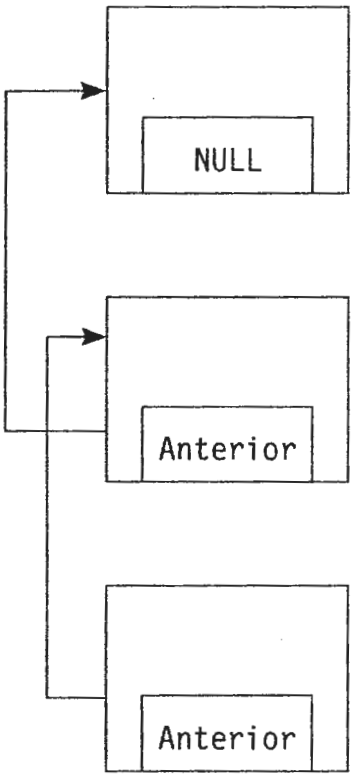
Lista ligada é um algoritmo de armazenamento de dados que muitas vezes supera o uso de uma matriz, ou de uma matriz de ponteiros.

A lista ligada assemelha-se a uma corrente em que os registros de dados estão pendurados seqüencialmente. O espaço de memória para cada registro é obtido pelo operador new, conforme surge a necessidade de adicionar itens à lista. Cada registro é conectado ao anterior por meio de um ponteiro. O primeiro registro contém um ponteiro com o valor NULL, e cada registro sucessor contém um ponteiro apontando para o anterior.

Nosso exemplo implementa uma lista ligada em uma classe. A lista é um objeto da classe **ListaLigada**. Cada registro é representado por uma variável estrutura do tipo Livro. Cada estrutura contém um conjunto de membros para armazenar os dados de um livro: título, autor, número do registro e preço. Há um membro a mais — um ponteiro — para armazenar o endereço do registro anterior.

A lista, como um todo, armazena um ponteiro para o último registro.

ESQUEMA DE UMA LISTA LIGADA



A CLASSE ListaLigada E O PROGRAMA Lista.Cpp

```
//Lista.Cpp
//Mostra a implementação de uma lista ligada
```

```
#include <iostream>
#include <cstdio>  //para gets()
#include <conio.h> //para getch()
using namespace std;

struct Livro
{
      char Titulo[30];
      char Autor[30];
      int  NumReg;
      double Preco;
      Livro *Anterior;
};

class ListaLigada
{
      private:
            Livro *Fim;
      public:
            ListaLigada() //Construtor default
            {
                  Fim=(Livro *)NULL;
            }
            void GetLivro();
            void PrintLivro();
            ~ListaLigada();
};

void ListaLigada::GetLivro()
{
      Livro *NovoLivro = new Livro;
      cout << "\nDigite titulo: ";
      gets(NovoLivro->Titulo);
      cout << "Digite autor: ";
      gets(NovoLivro->Autor);
      cout << "Digite o número do registro: ";
      cin >> NovoLivro->NumReg;
      cout << "Digite o preço: ";
      cin >> NovoLivro->Preco;
      //Limpar teclado após uso de cin
      cin.ignore(10,'\n');//Limpa teclado
      NovoLivro->Anterior=Fim;
      Fim=NovoLivro;
}

void ListaLigada::PrintLivro()
{
```

```
        Livro *Atual = Fim;
        while(Atual != NULL)
        {
                cout << endl;
                cout << "Título: " << Atual->Titulo << endl;
                cout << "Autor : " << Atual->Autor << endl;
                cout << "No.Reg: " << Atual->NumReg << endl;
                cout << "Preço : " << Atual->Preco << endl;
                Atual = Atual->Anterior;
        }
}

ListaLigada::~ListaLigada()
{
        Livro *Atual = Fim, *Anterior;
        do
        {
                Anterior = Atual->Anterior;
                delete Atual;
                Atual = Anterior;
        }while(Atual != NULL);
}

int main()
{
        ListaLigada Lista;

        do
        {
                Lista.GetLivro();
                cout << "\nInserir outro livro (s/n)? " << endl;
        } while( getch() != 'n');

        cout << "\nLISTA DOS LIVROS CADASTRADOS";
        cout << "\n============================";

        Lista.PrintLivro();
        system("PAUSE");
        return 0;
}
```

Veja a saída:

```
Digite título: Helena
Digite autor: Machado de Assis
Digite o número do registro: 102
Digite o preço: 70.5
```

```
Inserir outro livro? s

Digite título: Iracema
Digite autor: José de Alencar
Digite o número do registro: 321
Digite o preço: 63.25

Inserir outro livro? s

Digite título: Macunaíma
Digite autor: Mário de Andrade
Digite o número do registro: 543
Digite o preço: 73.3

Inserir outro livro? n

LISTA DOS LIVROS CADASTRADOS
============================

Título: Macunaíma
Autor : Mário de Andrade
No.Reg: 543
Preço : 73.3

Título: Iracema
Autor : José de Alencar
No.Reg: 321
Preço : 63.25

Título: Helena
Autor : Machado de Assis
No.Reg: 102
Preço : 70.5
```

Adicionando um Livro à Lista

O método GetLivro() adiciona um livro à lista na instrução:

```
Livro *NovoLivro = new Livro;
```

Em seguida, os membros da nova estrutura são preenchidos pelo usuário, com exceção do membro ponteiro. Para acessar os membros da estrutura, usamos o operador -> da mesma forma que o usamos com ponteiros para objetos de classes.

Ao membro, atribuímos o endereço da estrutura anterior; a instrução que faz a atribuição é a seguinte:

```
NovoLivro->Anterior=Fim;
```

Em seguida, o endereço do novo livro é atribuído ao ponteiro **Fim**. Assim, esse ponteiro sempre apontará para o último livro inserido na lista:

```
Fim = NovoLivro;
```

O novo livro é inserido no final da lista.

Imprimindo os Dados da Lista

Para imprimirmos os dados dos livros cadastrados, devemos seguir a cadeia de ponteiros, de uma estrutura para a anterior, até encontrar **NULL**, começando pelo último livro cadastrado.

O método **PrintLivro()** inicia atribuindo o valor do ponteiro **Fim** ao ponteiro **Atual**, criado para varrer a lista.

```
Livro *Atual = Fim;
```

Em seguida, analisa se a lista não está vazia, verificando se o conteúdo do ponteiro **Atual** é ou não **NULL**.

```
while(Atual != NULL)
```

Se não for **NULL**, a função entra no laço e imprime os valores dos membros da estrutura apontada por **Atual**, em seguida, acerta o ponteiro **Atual** para que aponte para a estrutura anterior:

```
Atual = Atual->Anterior;
```

O laço termina quando a primeira estrutura é atingida.

Liberando a Memória

Nossa classe tem um destrutor que caminha pela lista usando **delete** para liberar um por um dos registros. O código é semelhante ao do método **PrintLivro()**.

```
ListaLigada::~ListaLigada()
{
      Livro *Atual = Fim, *Anterior;
      do
      {
            Anterior = Atual->Anterior;
            delete Atual;
            Atual = Anterior;
      }while(Atual != NULL);
}
```

Implementações Adicionais

Observe que nossa lista é impressa no sentido inverso ao da entrada. Você pode reescrever a classe **ListaLigada** para que a lista seja manipulada no mesmo sentido da entrada de dados. Essa alteração envolve uma programação um pouco mais complexa.

Vários outros refinamentos podem ser implementados. Por exemplo, você pode escrever métodos para adicionar ou remover registros de qualquer posição da lista. Eliminar um item da lista é razoavelmente fácil. Se o registro a ser eliminado é B, então o ponteiro da estrutura anterior A é copiado para a estrutura seguinte C.

Ponteiros para Ponteiros

O programa **StrSort.Cpp**, descrito no Capítulo 9, mostra a ordenação de uma matriz de strings. Vamos alterar esse programa para criar uma matriz de ponteiros para objetos e mostrar como ordenar esses ponteiros baseados nos dados contidos nos objetos.

A modificação envolve o uso de ponteiros que apontam para outros ponteiros.

Nosso programa cria uma matriz de ponteiros para objetos da classe **String** e modifica a função **ordena()** para que receba um ponteiro para ponteiro.

Veja a listagem:

```
//PStrSort.Cpp
//Mostra o uso de ponteiros para ponteiros
#include <iostream>
#include <cstring>
using namespace std;

class String
{
      private:
            char *str;
      public:
            int Get()
            {
                  char nome[100];
                  gets(nome);
                  str = new char[strlen(nome)+1];
                  strcpy(str,nome);
                  return strcmp(str,"");
            }

            void Print() const
            {
                  cout << str << endl;
            }
```

```
            bool operator < (String& s) const ;
};

bool String::operator < (String& s)const
{
      return (strcmp(str,s.str) < 0 ) ? true : false;
}

void ordena(String **p,int n)
{
      String *temp;
      int i,j;

      for(i=0;i<n;i++)
      {
            for(j=i+1; j<n;j++)
                  if( *(*(p+j)) < *(*(p+i)))
                  {
                        temp = *(p+i);
                     *(p+i) = *(p+j);
                     *(p+j) = temp;
                  }
      }
}

int main()
{
      String *p[100];
      int n;
      for( n=0;;n++)
      {
            cout << "Digite nome ou [ENTER] para fim: ";
            p[n] = new String;
            if(p[n]->Get()==0) break;
      }

      cout << "\n\nLista original:" << endl;
      int i;
      for(i=0;i<n;i++)
            p[i]->Print();

      ordena(p,n);

      cout << "\n\nLista ordenada:" << endl;
      for(i=0;i<n;i++)
      {
            p[i]->Print();
```

```
        delete p[i];
    }
    system("PAUSE");
    return 0;
}
```

Eis a saída:

```
Digite nome ou [ENTER] para fim: Regiane Ferreira
Digite nome ou [ENTER] para fim: Ana Maria de Paula
Digite nome ou [ENTER] para fim: Denise Silveira
Digite nome ou [ENTER] para fim: André Victor Mesquita
Digite nome ou [ENTER] para fim: Lucy Coelho
Digite nome ou [ENTER] para fim:

        Lista original:

        Regiane Ferreira
        Ana Maria de Paula
        Denise Silveira
        André Victor Mesquita
        Lucy Coelho

        Lista ordenada:

        Ana Maria de Paula
        André Victor Mesquita
        Denise Silveira
        Lucy Coelho
        Regiane Ferreira
```

O programa declara uma matriz de 100 ponteiros para objetos do tipo **String**; em seguida cria um objeto por vez por meio do operador **new**, conforme o usuário digita os nomes. Quando o usuário digitar uma string vazia, o processo será interrompido. Toda vez que um objeto é criado, seu endereço é atribuído a um elemento da matriz de ponteiros.

Os nomes são impressos duas vezes: a primeira, na mesma ordem em que foram digitados, e a segunda, em ordem alfabética.

ORDENANDO PONTEIROS

A novidade deste programa está em como os objetos são ordenados. Na realidade, a função **ordena()** não ordena objetos, e sim os ponteiros para os objetos. A ordenação é muito mais rápida que a dos próprios objetos, pois estaremos movimentando ponteiros pela memória, e não objetos. Os ponteiros são variáveis que ocupam pouco lugar de memória; entretanto, os objetos podem ocupar muita memória.

Observe a declaração do primeiro argumento de ordena():

```
void ordena(String **p,int n)
```

O tipo da variável p é **String****. Essa notação indica que p é um ponteiro duplamente indireto. Quando o endereço de um objeto da classe **String** é passado para uma função como argumento, você já sabe que o seu tipo é **String***. Um único asterisco é usado para indicar o endereço de um objeto.

A função **ordena**() não recebe o endereço de um objeto, recebe o endereço de uma matriz de ponteiros para objetos do tipo **String**. O nome de uma matriz é um ponteiro para um elemento dela. No caso da nossa matriz, um elemento é um ponteiro. Portanto, o nome da matriz é um ponteiro que aponta para outro ponteiro.

Dois asteriscos são usados para indicar um ponteiro para ponteiro.

Notação Ponteiro para Matrizes de Ponteiros

Cada elemento da matriz **p** é um ponteiro para um objeto **String**. Assim **p[i]** é o endereço de um objeto **String**. Portanto, ***p[i]** é o nome desse objeto.

Você já sabe que um elemento de uma matriz pode ser escrito em notação ponteiro. Desse modo, podemos escrever a expressão **p[i]** como

```
*(p+i)
```

então a expressão ***p[i]** que indica o nome do objeto pode ser escrita como:

```
*(*(p+i))
```

em notação ponteiro.

Argumentos da Linha de Comando

Se você já trabalhou com os sistemas operacionais Unix, Linux ou DOS, com certeza já deve estar acostumado a usar argumentos da linha de comando.

Por exemplo:

```
C:\>FORMAT A: /s /u
```

Nesse exemplo, A: é um argumento da linha de comando, /s e /u são outros dois. Os itens digitados na linha de comando do DOS são chamados de *argumentos da linha de comando.*

Usuários do Windows podem digitar argumentos da linha de comando após o nome do programa na opção Executar do menu Iniciar.

Como podemos escrever programas em C++ que acessem esses argumentos?

Os argumentos digitados na linha de comando são enviados pelo sistema operacional como argumentos da função main(). Para que ela possa reconhecê-los, é necessário declará-los, como é feito em qualquer função C++.

Exemplo:

```
//LinCom.Cpp
//Mostra argumentos da linha de comando
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
      cout << "Número de argumentos: " << argc << endl;

      for(int i=0; i<argc; i++)
            cout<< "Argumento número "<< i <<": "<< argv[i] << endl;

      system("PAUSE");
      return 0;
}
```

Veja uma simples execução:

```
C:\>LinCom José 555 João 246 Maria 987

Número de argumentos: 7
Argumento número 0: C:\CppProjetos\LinCom.exe
Argumento número 1: José
Argumento número 2: 555
Argumento número 3: João
Argumento número 4: 246
Argumento número 5: Maria
Argumento número 6: 987
```

A função **main**() recebe dois argumentos: argc e argv.

```
int main(int argc, char **argv)
```

O primeiro argumento representa o número de argumentos digitados na linha de comando; neste caso, 7.

O segundo argumento corresponde a uma matriz de ponteiros para strings, em que cada string representa um dos argumentos da linha de comando. As cadeias de caracteres podem ser acessadas por meio da notação matriz, argv[0], argv[1] etc. ou por meio da notação ponteiro, *(argv+0), *(argv+1) etc.

A primeira cadeia, argv[0], é sempre o nome do programa em execução e o seu caminho de localização no disco.

Os nomes argc (ARGument Count) e argv (ARGument Values) são tradicionalmente usados para esse fim, mas qualquer outro poderia ser usado.

PONTEIROS PARA FUNÇÕES

Terminaremos este capítulo mostrando um tipo de ponteiro especial: ponteiro que aponta para uma função. Ou seja, uma variável que conterá o endereço de uma função. A função poderá ser executada por meio do ponteiro.

Nosso primeiro exemplo mostra um ponteiro para a função **doisbeep()**.

Veja a listagem:

```
//PtrFunc.Cpp
//Mostra o uso de ponteiro para função
#include <iostream>
using namespace std;
void doisbeep(void);
int main()
{
      void (*pf)(void);//ponteiro para função void que recebe void

      pf = doisbeep; //nome da função sem os parênteses

      (*pf)();//chama a função

      system("PAUSE");
      return 0;
}

//doisbeep()
//toca o alto-falante duas vezes
void doisbeep(void)
{
      cout << '\a';
      for(int i=0; i<80000 ; i++); //dar um tempo
      cout << '\a';
}
```

DECLARANDO O PONTEIRO PARA FUNÇÃO

A função **main()** começa declarando **pf** como um ponteiro para uma função void. É claro que o tipo **void** é uma das possibilidades. Se a função a ser apontada é do tipo float, por exemplo, o ponteiro para ela deve ser declarado como tal.

```
void (*pf)(void);
```

Observe os parênteses envolvendo *pf. Eles são realmente necessários, pois, se omitidos

```
void *pf(void); //ERRO
```

estaríamos declarando pf como uma função que retorna um ponteiro void; em outras palavras, estaríamos escrevendo o protótipo da função pf.

Endereços e Funções

O nome de uma função, desacompanhado de parênteses, é o seu endereço. A instrução

```
pf = doisbeep; //nome da função sem os parênteses
```

atribui o endereço da função doisbeep() a pf. Observe que não colocamos parênteses junto do nome da função. Se eles estivessem presentes, como em

```
pf = doisbeep();    //ERRO
```

estaríamos atribuindo a pf o valor de retorno da função, e não o seu endereço.

Executando a Função Por Meio do Ponteiro

Da mesma maneira que podemos substituir o nome de uma variável usando um ponteiro acompanhado do operador indireto (*), podemos substituir o nome da função usando o mesmo mecanismo. A instrução

```
(*pf)();//chama a função
```

é equivalente a

```
doisbeep();
```

e indica uma chamada à função doisbeep().

Operações Ilegais com Ponteiros para Funções

A aritmética com ponteiros para funções não é definida em C++. Por exemplo, você não pode incrementar ou decrementar ponteiros para funções.

Ponteiros para Funções como Argumentos

O exemplo a seguir cria um ponteiro para armazenar o endereço da função de biblioteca gets(). Ela tem o seguinte protótipo:

```
char *gets(char *);
```

definido no arquivo cstdio. A função retorna um ponteiro para a cadeia de caracteres lida do teclado e armazenada no endereço recebido por ela como argumento.

```
//PtrGets.Cpp
//Mostra o uso de ponteiros como argumentos de função
#include <iostream>
#include <cstdio>
using namespace std;

void func( char * (*p)(char *));

int main()
{
      char * (*p)(char *);
      p = gets;
      func(p);

      system("PAUSE");
      return 0;
}

void func(char * (*p)(char *))
{
      char nome[80];

      cout << "Digite seu nome: ";

      (*p)(nome);  //chama a função gets()

      cout << "Seu nome é: " << nome << endl;
}
```

A declaração do ponteiro é a seguinte:

```
char * (*p)(char *);
```

Essa instrução indica que **p** é um ponteiro para uma função do tipo **char*** e recebe um **char*** como argumento.

A função **main()** envia o ponteiro **p** como argumento para a função **func()**.

MATRIZES DE PONTEIROS PARA FUNÇÕES

Os ponteiros para funções oferecem uma maneira eficiente de executar uma entre uma série de funções, com base em alguma escolha dependente de parâmetros conhecidos somente em tempo de execução.

Por exemplo, suponhamos que você queira escrever um programa no qual é apresentado um "menu" de opções ao usuário e, para cada escolha, o programa deve executar uma chamada a determinada função. Em vez de utilizar estruturas tradicionais de programação como **switch** ou **if-else** ou qualquer outra estrutura de controle para decidir qual função

deve ser chamada, você simplesmente cria uma matriz de ponteiros para funções e executa a função correta por meio de seu ponteiro.

Veja um esqueleto do programa:

```
//FPtrMatriz.CPP
//Mostra uma matriz de ponteiros para função
#include <iostream>
using namespace std;

void func0(void), func1(void), func2(void);//Protótipos

int main()
{
      void (*ptrf[3])(void);//Matriz de ponteiros para funções

      ptrf[0] = func0;
      ptrf[1] = func1;
      ptrf[2] = func2;

      do
      {
            int i;
            cout << "0 - ABRIR" << endl;
            cout << "1 - FECHAR" << endl;
            cout << "2 - SALVAR" << endl;
            cout << "\nEscolha um item: ";
            cin >> i;
            if(i < 0 || i > 2) break;
            (*ptrf[i])();
      } while(true);
      system("PAUSE");
      return 0;
}

void func0()
{
      cout << "\n*** Estou em func0() ***" << endl;
}

void func1()
{
      cout << "\n*** Estou em func1() ***" << endl;
}

void func2()
{
      cout << "\n*** Estou em func2() ***" << endl;
}
```

A instrução

```
void (*ptrf[3])(void);//Matriz de ponteiros para funções
```

declara uma matriz de três ponteiros para funções **void**. O laço **do-while** em **main**() chama uma das três funções, dependendo da escolha do usuário. O programa termina se um número menor que zero ou maior que dois for digitado.

Matrizes de ponteiros para funções fornecem um mecanismo alternativo de mudar o controle do programa sem utilizar estruturas de controle convencionais.

INICIALIZANDO UMA MATRIZ DE PONTEIROS PARA FUNÇÕES

O programa anterior poderia ter inicializado a matriz **ptrf** na mesma instrução de sua declaração. Observe a modificação:

```
//FPtrMatriz.CPP
//Mostra uma matriz de ponteiros para função inicializada
#include <iostream>
using namespace std;
void func0(void), func1(void), func2(void);//Protótipos

int main()
{
      void (*ptrf[3])(void)={ func0, func1, func2};//Matriz inicializada

      do
      {
            int i;
            cout << "0 - ABRIR" << endl;
            cout << "1 - FECHAR" << endl;
            cout << "2 - SALVAR" << endl;
            cout << "\nEscolha um item: ";
            cin >> i;
            if(i < 0 || i > 2) break;
            (*ptrf[i])();
      } while(true);
      system("PAUSE");
      return 0;
}

void func0()
{
      cout << "\n*** Estou em func0() ***" << endl;
}

void func1()
```

```
{
      cout << "\n*** Estou em func1() ***" << endl;
}

void func2()
{
      cout << "\n*** Estou em func2() ***" << endl;
}
```

USANDO typedef PARA DECLARAR UM PONTEIRO PARA FUNÇÃO

É comum definir um nome para um tipo de dado que seja ponteiro para função. Fazemos isso por meio de typedef.

```
//FPtrMatriz.CPP
//Mostra uso de typedef
#include <iostream>
using namespace std;
void func0(void), func1(void), func2(void);//Protótipos

typedef void (*PFunc)(void); //O tipo PFunc é ponteiro para a função void

int main()
{
      PFunc ptr[3] = { func0, func1, func2};//Matriz do tipo PFunc

      do
      {
            int i;
            cout << "0 - ABRIR" << endl;
            cout << "1 - FECHAR" << endl;
            cout << "2 - SALVAR" << endl;
            cout << "\nEscolha um item: ";
            cin >> i;
            if(i < 0 || i > 2) break;

            (ptr[i])(); //Chama função

      } while(true);
      system("PAUSE");
      return 0;
}

void func0()
{
      cout << "\n*** Estou em func0() ***" << endl;
```

```
}

void func1()
{
      cout << "\n*** Estou em func1() ***" << endl;
}

void func2()
{
      cout << "\n*** Estou em func2() ***" << endl;
}
```

Ponteiros para Métodos de Classes

Vamos mostrar um exemplo de como criar uma matriz de ponteiros para métodos de uma classe. Observe a sintaxe:

```
//FPtrClass.CPP
//Mostra ponteiros para métodos de classe
#include <iostream>
using namespace std;

class Calculadora
{
private:
      float a,b;
public:
      Calculadora() : a(1),b(1){}
      Calculadora(float x, float y) : a(x),b(y){}

      float adicao(){ return a+b;}
      float subtracao(){ return a-b;}
      float multiplicacao(){ return a*b;}
      float divisao (){ return a/b;}
};

typedef float (Calculadora::*PFunc)();

int main()
{
      Calculadora x(2.8,3.1);

      PFunc ptrf[4] ={Calculadora::adicao,
                      Calculadora::subtracao,
                      Calculadora::multiplicacao,
                      Calculadora::divisao};
```

```
do
{
        int i;
        cout << "0 - Adicao" << endl;
        cout << "1 - Subtracao" << endl;
        cout << "2 - Multiplicacao" << endl;
        cout << "3 - Divisao" << endl;
        cout << "\nEscolha um item: ";
        cin >> i;
        if(i < 0 || i > 3) break;

        cout << (x.*ptrf[i])() << endl;//Chama método

} while(true);
system("PAUSE");
return 0;
}
```

Revisão

1. Um ponteiro é um endereço de memória. Qualquer coisa armazenada na memória do computador tem um endereço e este endereço é um *ponteiro constante.*
2. *Ponteiro variável* é um lugar na memória que armazena o endereço de outra variável.
3. Podemos encontrar o endereço de variáveis usando o *operador de endereços* (&).
4. Se um ponteiro variável **p** contém o endereço de uma variável **var**, dizemos que **p** aponta para **var**, ou que **var** é a variável apontada por **p**.
5. Os ponteiros variáveis são declarados usando um asterisco (*) que significa **ponteiro para**. O tipo da variável apontada deve sempre ser especificado na declaração do ponteiro para que o compilador possa executar operações aritméticas corretamente.
6. O *operador indireto* (*) pode ser usado com ponteiros para obter o conteúdo da variável apontada por ele ou para executar qualquer operação na variável apontada. Em outras palavras, o operador indireto junto do nome do ponteiro substitui o nome da variável apontada.
7. Em operações aritméticas com ponteiros, a unidade adotada é o número de bytes do tipo apontado.
8. Os elementos de matrizes podem ser acessados pela notação matriz, com colchetes, ou pela notação ponteiro, com um asterisco. O nome da matriz é um ponteiro constante.
9. Se o endereço de uma variável é passado como argumento para uma função, a função terá acesso à variável original. A passagem por ponteiros oferece os mesmos benefícios da passagem por *referência.* Em alguns casos, os ponteiros oferecem maior flexibilidade.
10. Uma cadeia de caracteres constante pode ser definida como uma matriz ou podemos atribuir o seu endereço a um ponteiro. Usar ponteiros, nesse caso, facilita a manipulação dos caracteres da cadeia.

11. Definir uma *matriz de ponteiros para cadeias de caracteres constantes* aloca menos memória que a definição por meio de uma matriz de duas dimensões.

12. O operador reinterpret_cast<> é usado para modificar o tipo apontado por um ponteiro. O operador static_cast<> não trabalha com ponteiros.

13. O operador **new** obtém uma quantidade específica de memória do sistema e retorna um ponteiro para essa memória. Ele é usado para criar variáveis durante a execução do programa.

14. O operador **delete** devolve ao sistema a memória alocada por **new**.

15. Os ponteiros **void** podem apontar para qualquer tipo de dado e são usados em situações onde necessitamos de um ponteiro genérico, independentemente do tipo de dado apontado.

16. O ponteiro **this** sempre aponta para o objeto do qual o método é membro. Qualquer membro da classe poderá ter acesso a ele. O ponteiro **this** é útil quando o método deve retornar o objeto do qual é membro.

17. Métodos **static** não têm acesso ao ponteiro **this**, pois podem ser executados mesmo se nenhum objeto existir.

18. Membros de um objeto podem ser acessados por meio de um ponteiro que aponta para o objeto. Para tanto, deve ser usado o *operador de acesso a membros* (->). A mesma sintaxe é usada para acessar membros de uma estrutura.

19. Uma classe ou uma estrutura pode conter um ponteiro que aponta para o seu próprio tipo, mas não poderá conter um objeto dela mesma. Por meio desse conceito, podemos criar algoritmos complexos de armazenamento e acesso a dados como *listas ligadas*.

20. Um ponteiro pode apontar para outro ponteiro. Esse tipo de variável é declarada usando duas vezes o asterisco (**).

21. Um programa C++ pode acessar os *argumentos da linha de comando* por meio do uso de dois argumentos na definição da função **main()**. O primeiro, argc, é um número inteiro cujo valor indica o número de argumentos fornecidos na linha de comando. O segundo, argv, é uma matriz de ponteiros para cadeias de caracteres que permite o acesso aos dados fornecidos.

22. Um *ponteiro para uma função* contém o endereço da localização da função na memória e é usado sempre que o nome da função for desconhecido, ou como argumento de funções que chamam outras funções por intermédio de seu endereço.

23. Uma *matriz de ponteiros para funções* é usada em situações em que o programa deve escolher a função a ser executada dependendo de uma condição conhecida somente durante a execução.

Exercícios

1. Um ponteiro é:
   a) o endereço de uma variável;
   b) uma variável que armazena endereços;
   c) o valor de uma variável;
   d) um indicador da próxima variável a ser acessada.

2. Escreva uma instrução que imprima o endereço da variável **var**.

3. Indique: (1) operador de endereços (2) operador de referência
   a) `p = &i;`
   b) `int &i=j;`
   c) `cout << &i;`
   d) `int *p=&i;`
   e) `int& func(void);`
   f) `void func(int &i);`
   g) `func(&i);`

4. A instrução:
   ```
   int *p;
   ```
   a) cria um ponteiro com valor indefinido;
   b) cria um ponteiro do tipo **int**;
   c) cria um ponteiro com valor zero;
   d) cria um ponteiro que aponta para uma variável do tipo **int**.

5. Qual o significado do operador asterisco em cada um dos seguintes casos:
   a) `int *p;`
   b) `cout << *p;`
   c) `*p = x*5;`
   d) `cout << *(p+1);`

6. Quais das seguintes instruções declaram um ponteiro para uma variável **float**?
   a) `float *p;`
   b) `*float p;`
   c) `float* p;`
   d) `float *p=&f;`
   e) `*p;`
   f) `float& p=q;`
   g) `*float &p;`

7. O que é do tipo **int** na instrução a seguir?
   ```
   int *p;
   ```
   a) a variável **p**;
   b) o endereço de **p**;
   c) a variável apontada por **p**;
   d) o endereço da variável apontada por **p**.

8. Se o endereço de **var** foi atribuído a um ponteiro variável **pvar**, quais das seguintes expressões são verdadeiras?
   a) `var == &pvar;`
   b) `var == *pvar;`
   c) `pvar == *var;`
   d) `pvar == &var;`

9. Considere as declarações abaixo e indique qual o valor das seguintes expressões:
   ```
   int i=3,j=5;
   int *p=&i, *q=&j;
   ```

a) `p==&i;`
b) `*p - *q;`
c) `**&p;`
d) `3*-*p/ *q+7;`

10. Qual a saída deste programa?

```
#include <iostream>
int main()
{
        int i=5,*p;
        p=&i;
        cout << p << '\t' << (*p+2) << '\t' << **&p
             << '\t' << (3**p) << '\t' << (**&p+4) << endl;
        return 0;
}
```

11. Se **i** e **j** são variáveis inteiras e **p** e **q** são ponteiros para **int**, quais das seguintes expressões de atribuição são incorretas?

a) `p=&i;`
b) `*q=&j;`
c) `p=&*&i;`
d) `i=(*&)j;`
e) `i=*&*&j;`
f) `q=&p;`
g) `i=(*p)++ + *q;`
h) `if(p == i) i++;`

12. Explique cada uma das seguintes declarações e identifique quais são incorretas:

a) `int *const p=&x;`
b) `const int &p=*x;`
c) `int &const p=*x;`
d) `int const *p=&x;`

13. O seguinte programa é correto?

```
#include <iostream>
const VAL=987;
int main()
{
        int *p=VAL;
        cout << *p;
        return 0;
}
```

14. O seguinte programa é correto?

```
#include <iostream>
const VAL=987;
int main()
{
          int i=VAL;
```

```
        int *p;
        cout << *p;
        return 0;
}
```

15. Qual a diferença entre: mat[3] e *(mat+3)?

16. Admitindo a declaração: **int mat[8];** por que a instrução **mat++;** é incorreta?

17. Admitindo a declaração: **int mat[8];** quais das seguintes expressões se referem ao valor do terceiro elemento da matriz?
    a) *(mat+2);
    b) *(mat+3);
    c) mat+2;
    d) mat+3;

18. O que faz o programa seguinte?

```
#include <iostream>
int main()
{
        int mat[]={4,5,6};
        for(int j=0; j<3 ; j++)
              cout << *(mat+j) << endl;
        return 0;
}
```

19. O que faz o programa seguinte?

```
#include <iostream>
int main()
{

        int mat[]={4,5,6};
        for(int j=0; j<3 ; j++)
                cout << (mat+j) << endl;
        return 0;
}
```

20. O que faz o programa seguinte?

```
#include <iostream>
int main()
{
        int mat[]={4,5,6};
        int *p=mat;
        for(int j=0; j<3 ; j++)
                cout << *p++ << endl;
        return 0;
}
```

21. Qual a diferença entre as duas instruções seguintes?

```
char s[]="Brasil";
char *s ="Brasil";
```

22. Considerando a declaração

```
char *s = "Eu não vou sepultar Cesar";
```

o que imprimirão as instruções seguintes?

a) `cout << s;`
b) `cout << &s[0];`
c) `cout << (s+11);`
d) `cout << s[0];`

23. Escreva a expressão **mat[i][j]** em notação ponteiro.

24. Qual a diferença entre os seguintes protótipos de funções:

```
void func(char *p);
void func(char p[]);
```

25. Considerando a declaração

```
char *items[5] = {  "Abrir",
                    "Fechar",
                    "Salvar",
                    "Imprimir",
                    "Sair"
                 };
```

para poder escrever a instrução **p=items;** a variável **p** deve ser declarada como:

a) `char p;`
b) `char *p;`
c) `char **p;`
d) `char ***p;`
e) `char *p[];`
f) `char **p[][];`

26. O operador **new**:
    a) cria uma variável de nome new;
    b) retorna um ponteiro **void**;
    c) aloca memória para uma nova variável;
    d) informa a quantidade de memória livre.

27. O operador **delete**:
    a) apaga um programa;
    b) devolve memória ao sistema operacional;
    c) diminui o tamanho do programa;
    d) cria métodos de otimização.

28. Explique o significado da palavra **void** em cada uma das seguintes instruções:
    a) `void *p;`
    b) `void p();`

c) `void p(void);`
d) `void (*p)();`

29. Qual o erro deste trecho de programa:

```
float x = 333.33;
void *p = &x;

cout << *p;
```

30. Qual o significado do ponteiro **this** e quando ele é usado?

31. Se p é um ponteiro para um objeto da classe Data, então quais das seguintes instruções executam o método **PrintData**()?
a) `p.PrintData();`
b) `*p.PrintData();`
c) `p->PrintData();`
d) `*p->PrintData();`

32. Se p é uma matriz de ponteiros para objetos da classe **Data**, escreva uma instrução que execute o método **PrintData**() do objeto apontado pelo terceiro elemento da matriz p.

33. Em uma lista ligada:
a) cada item contém um ponteiro para o próximo item;
b) cada item contém dados ou ponteiros para os dados;
c) cada item contém uma matriz de ponteiros;
d) os itens são armazenados em uma matriz.

34. Observe a classe **String** do programa **PStrSort.Cpp**. Por que motivo o programa não funcionará se incluirmos o seguinte destrutor na classe **String**?

```
~String()
{ delete str;}
```

35. O que declara cada uma destas instruções?
a) `int (*ptr)[10];`
b) `int *ptr[10];`
c) `int (*ptr)();`
d) `int *ptr();`
e) `int (*ptr[10])();`

36. Escreva uma função que inverta a ordem dos caracteres de uma cadeia de caracteres que ela recebe como argumento. Use ponteiros.

Exemplo: "Saudações" resulta "seõçaduaS".

37. Escreva um programa que solicite ao usuário o número de notas a serem digitadas, crie uma matriz, com a dimensão especificada, para armazenar as entradas, solicite as notas e chame uma função que retorne a média aritmética das notas. Após imprimir a média, o programa libera a memória alocada para a matriz.

38. Adicione um destrutor à classe **ListaLigada** que destrua todos os itens da lista quando um objeto é liberado da memória.

39. Crie uma estrutura para descrever restaurantes. Os membros devem armazenar o nome, o endereço, o preço médio e o tipo de comida. Crie uma lista ligada que apresente os restaurantes de um certo tipo de comida indexados pelo preço. O menor preço deve ser o primeiro da lista. Escreva um programa que peça o tipo de comida para o usuário e imprima os restaurantes que oferecem esse tipo de comida.

40. Escreva a função **RetornaPFunc()** que retorne um ponteiro para função correspondente ao inteiro recebido como argumento. Declare o seu protótipo e teste o programa seguinte:

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

void func0(void), func1(void), func2(void);//Protótipos

typedef void (*PFunc)(void); //O tipo PFunc é ponteiro para a função void

int main()
{
    PFunc ptr;

    do
  {
            int i;
            cout << "0 - ABRIR" << endl;
            cout << "1 - FECHAR" << endl;
            cout << "2 - SALVAR" << endl;
            cout << "\nEscolha um item: ";
            cin >> i;
            if(i < 0 || i > 2) break;

           ptr = RetornaPFunc(i);
           (ptr)(); //Chama função

      } while(true);
      system("PAUSE");
      return 0;
}
```

41. Escreva uma classe chamada **Fruta**. Não defina nenhum membro privado. Inclua três métodos públicos do tipo **void** que não recebam nada como argumento. Os nomes dos métodos são **Pera**, **Banana** e **Goiaba**. Cada método tem uma única instrução que imprime o nome dele.

    A função **main()** declara uma matriz de três ponteiros para os métodos da classe Fruta e inicializa a matriz com os endereços dos métodos. O usuário digita um número entre 0 e 2, e o programa executa o método correspondente.