Ponteiros

Em C++

O que são ponteiros?

- Um ponteiro é um endereço de memória.
- Seu valor indica onde uma variável está armazenada, e não o que está armazenado.
- Proporciona um modo de acesso a uma variável sem referenciá-la diretamente.
- Variáveis auxiliares que permitem o acesso aos valores de outras variáveis indiretamente.
- Ponteiros, como todas as variáveis, possuem um conteúdo e uma localização. Esta localização pode ser armazenada em outra variável, o que torna então um ponteiro para um ponteiro.

Por que usar ponteiros?

- Manipular elementos de matrizes;
- Receber argumentos em funções que necessitem modificar o argumento original;
- Passar strings de uma função para outra;
- Substituir matrizes;
- Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde um item deve conter referências a outro;
- Alocar e desalocar memória do sistema.

Endereços de memória

- **Endereço**: referência que o computador usa para localizar variáveis.
- **Memória do computador**: dividida em bytes numerados de 0 até o limite de memória da máquina - chamados de endereços de bytes.
- Toda variável ocupa uma certa localização na memória, e seu endereço é do primeiro byte ocupado por ela.
- **Operador de endereços &:** usado para se conhecer o endereço ocupado por uma variável;
- **Operador indireto (*):** operador unário que resulta no valor da variável apontada.

Exemplo

Exibir o endereço de variáveis:

```
#include <iostream>
using namespace std;
int main()
{
   int i, j, k;
   cout << "\n Endereco de i: " << &i;
   cout << "\n Endereco de j: " << &j;
   cout << "\n Endereco de k: " << &k;
}
```

Exemplo de Saída :

```
Endereco de i:  0x6dfefc
Endereco de j:  0x6dfef8
Endereco de k: 0x6dfef4
```

O endereço ocupado por essas variáveis depende, dentre outros fatores:

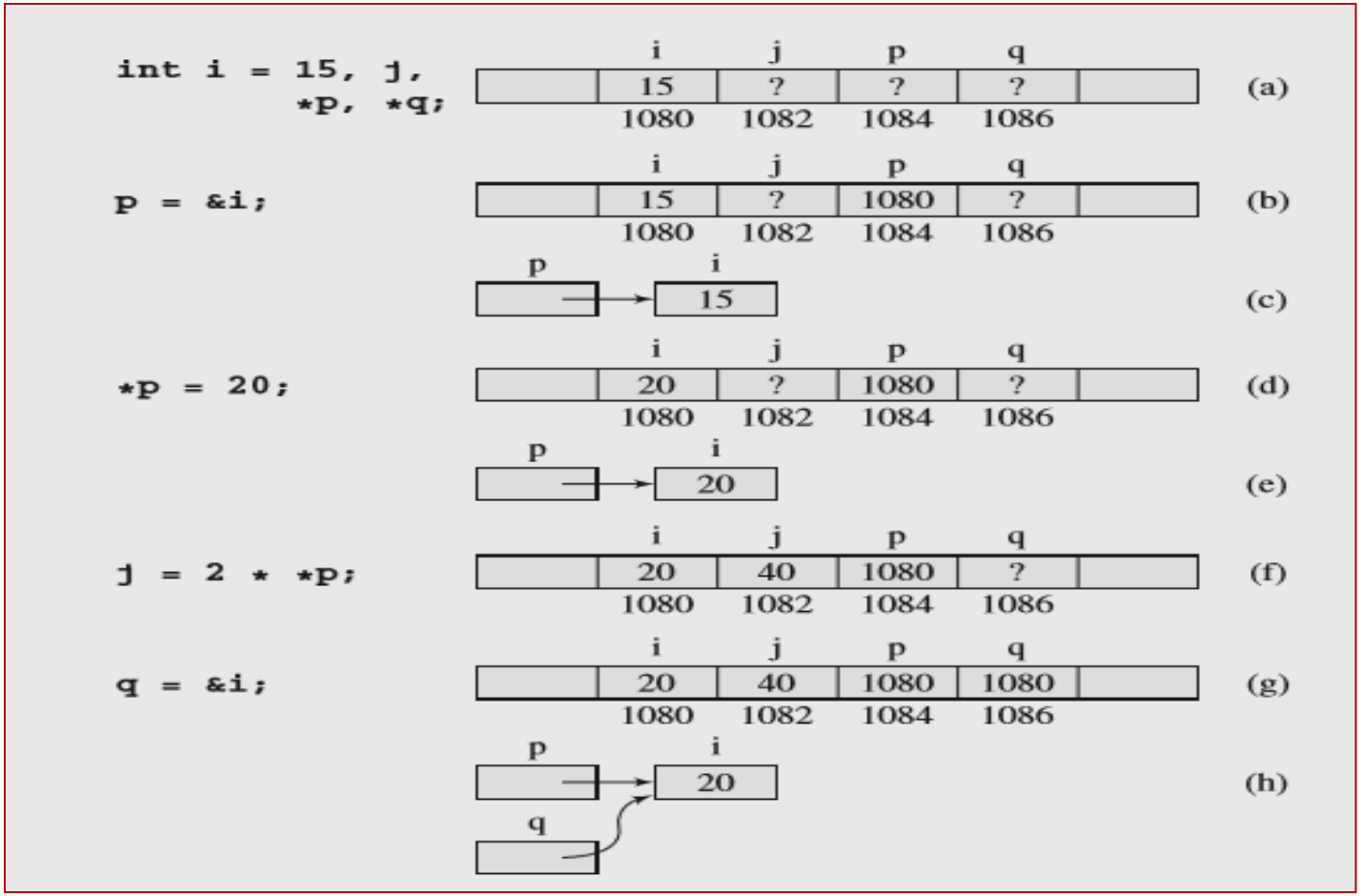
- do tamanho do sistema operacional,
- Se existem outros programas residentes na memória,
- do tamanho do tipo de dados,

Logo, diferentes valores podem ser encontrados.

- No exemplo, cada endereço difere do próximo por 4 bytes (tamanho dos inteiros)

Exemplo de instruções com variáveis ponteiro

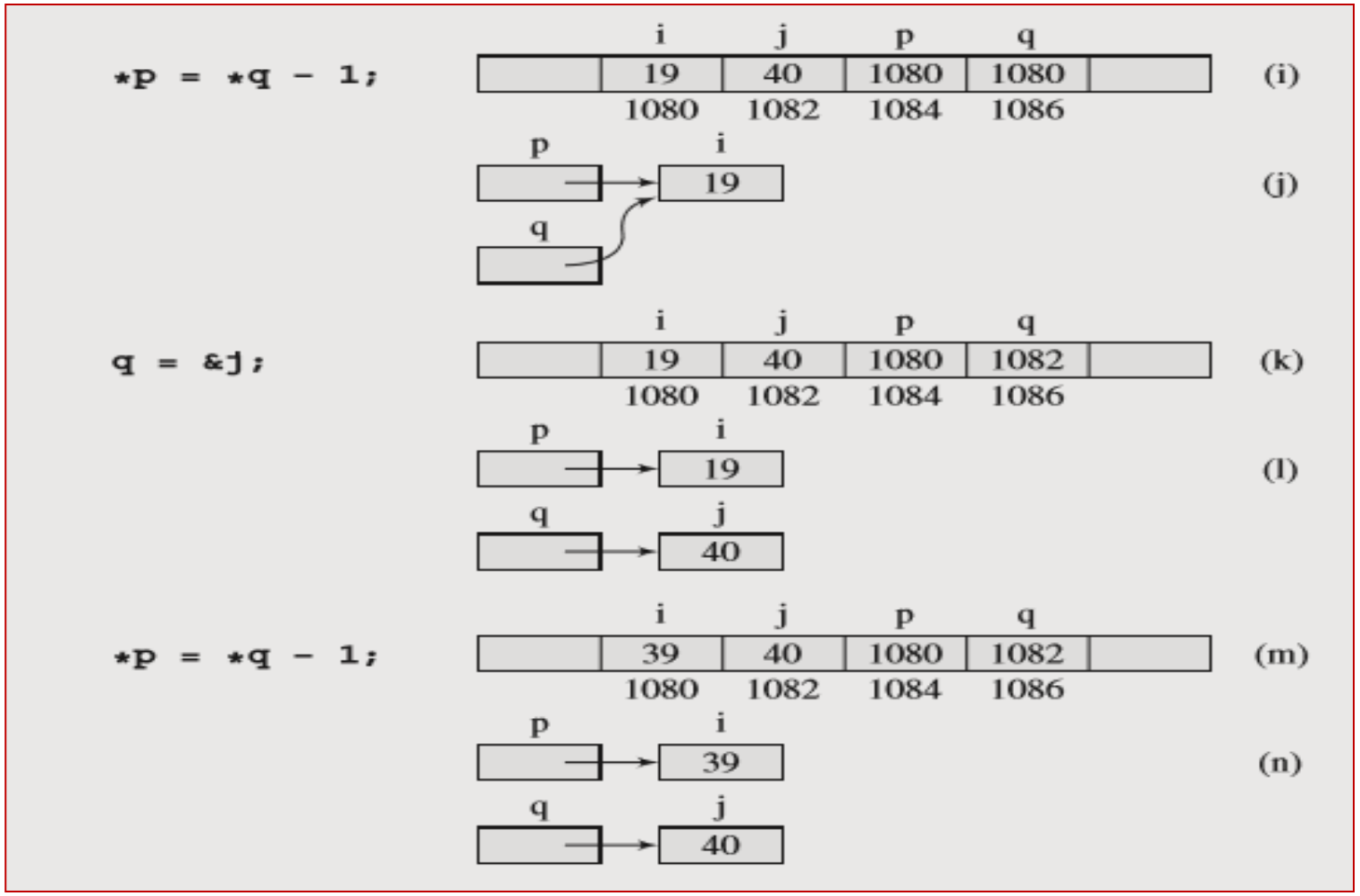
- Valores armazenados na memória:



- Uma variável aponta para outra quando a primeira contém o endereço da segunda.

Exemplo de instruções com variáveis ponteiro

- Valores armazenados na memória:



Ponteiros e variáveis de referência

O operador de referência cria outro nome para uma variável já existente.

```
#include <iostream>
using namespace std;
int main()  {
   int n = 5, *p = &n, &r = n;
   cout << n << ' ' << *p << ' ' << r << endl;  // 5 5 5
   n = 7;  cout << n << ' ' << *p << ' ' << r << endl;  // 7 7 7
   *p = 9; cout << n << ' ' << *p << ' ' << r << endl;  // 9 9 9
   r = 10; cout << n << ' ' << *p << ' ' << r << endl;  // 10 10 10
}
```

- Variável p: do tipo int*, um ponteiro para um inteiro.
- Variável r: do tipo int&, uma variável de referência inteira.
- Variável de referência: deve ser inicializada como uma referência a uma variável particular, e esta referência não pode ser mudada – não pode ser nula!
- **A variável de referência r** pode ser considerada um **nome diferente para a variável n**. Logo, se **n** muda, **r** também muda. Implementada como um ponteiro constante à variável.

Passagem de parâmetros por referência

- O operador de referência cria outro nome para uma variável já existente. Toda operação em qualquer dos nomes tem o mesmo resultado.
- **Uma referência** não é uma cópia da variável a que se refere, e sim, **a mesma variável sob diferentes nomes**.
- **Vantagens**: a função pode acessar as variáveis da função que chamou; uma função pode retornar mais de um valor para a função que chama. Os valores a serem retornados são colocados em referência de variáveis da função chamadora.

- **Argumentos passados por valor**: a função chamada cria novas variáveis do mesmo tipo dos argumentos e copia nelas o valor dos argumentos passados. A função não tem acesso às variáveis originais da função que chamou, portanto, não as pode modificar.

Passagem de parâmetros por referência

```
#include <iostream>
using namespace std;
void reajusta20(float &p, float &r) {
   r = p * 0.2;
   p *= 1.2;
}
int main()    {
   float preco, valor_reajuste;
   do   {
      cout << "\n\n Insira o preco atual:";
      cin >> preco;
      reajusta20 (preco, valor_reajuste);
      cout << "\n preco novo:" << preco;
      cout << "\n Aumento:"    << valor_reajuste;
   } while (preco != 0.0);
}
```

- O operador & só é usado na definição do tipo do argumento;
- A declaração: **float &p, float &r** indica que p e r são outros nomes para as variáveis passadas como argumento pela função que chama;
- A chamada a uma função que recebe uma referência é idêntica à chamada às funções em que os argumentos são passados por valor;

Passando argumentos por referência com ponteiros.

- **A função chamadora**, em vez de passar valores para a função chamada, passa endereços usando o operador de endereços. Estes endereços são de variáveis da função chamadora onde queremos que a função coloque novos valores.

  Ex.: reajusta20(&preco, &valor_reajuste);

- **A função chamada** deve criar variáveis para armazenar os endereços enviados pela função chamadora. Estas variáveis são **ponteiros**.

  *Ex.:void reajusta20(float *p, float *r);*

Passando argumentos por referência com ponteiros.

```
void reajusta20 (float *p, float *r);

void main()   {
   float preco, valor_reajuste;
   do    {
      cout << "\n\n Insira o preco atual:";
      cin >> preco;
      reajusta20(&preco, &valor_reajuste);
      cout << "\n preco novo:" << preco;
      cout << "\n Aumento:"    << valor_reajuste;
   } while (preco != 0.0);
}
void reajusta20(float *p, float *r) {
   *r = *p * 0.2;
   *p *= 1.2;
}
```

- O asterisco (*) faz parte do nome do tipo e indica **Ponteiro para.**
- ***p** e ***r são** do tipo **float** e
- **p** e **r** são ponteiros para variáveis **float.**
- ***p:** maneira indireta de usar a variável **preco** de **main()**. Toda alteração feita em ***p afetará** diretamente **preco;**
- Como **p** contém o endereço da variável **preco**, dizemos que **p** aponta para **preco.**

Ponteiros sem funções

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

   int *px, *py;     //Ponteiros para variáveis
   px = &x;          //Atribui um endereco ao ponteiro
   py = &y;

   cout << "\n px= " << px << "\t *px=" << *px;
   cout << "\n py= " << py << "\t *py=" << *py;
}
```

- Saída (exemplo):

```
&x= 0x6dfef4    x=4
 &y= 0x6dfef0    y=7
 px= 0x6dfef4    *px=4
 py= 0x6dfef0    *py=7
```

Ponteiros sem funções

```
void main()     {
    int x=5, y=6;
    int *px = &x;    int *py= &y;    // Inicializa px com o endereço de x e py com o endereço de y.
    cout << "\n px = " << px         // Nome do ponteiro: valor contido nele - endereço da variável apontada
       << ", *px = " << *px          //Op. Indireto
       << ", &px = " << &px;         //Op. Endereços
    cout << "\n py = " << py        // Nome do ponteiro: valor contido nele - endereço da variável apontada
       << ", *py = " << *py          //Op. Indireto
       << ", &py = " << &py;         //Op. Endereços
    if (px < py)                     //Comparações - sempre entre ponteiros de mesmo tipo
          cout << "\n py-px= " << (py - px);    //Diferença entre ponteiros - expressa em número de tipo
                                                //apontado entre eles
    else    cout << "\n px-py= " << (px - py);

    py++;                          //Incremento de um int.  Movimenta para o próximo tipo apontado.
    cout << "\n\n py++ ";           cout << "\n py = " << py  << ",*py = " << *py  << ",&py = " << &py;
    px = py + 3;                   //Caminha três inteiros adiante de py.
    cout << "\n px=py+3 ";        cout << "\n px = " << px  << ",*px = " << *px  << ",&px = " << &px;
    cout << "\n px-py= " << (px - py);
}
```

Unidade adotada em operações com ponteiros

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

  Exemplo: 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 4 bytes (um int); se somarmos 1 a **pd** estaremos somando 8 bytes (um double),...
- A unidade com ponteiros é o **número de bytes do tipo apontado**.

Saída do exemplo anterior (exemplo):

px = 0x6dfeec, *px = 5, &px = 0x6dfee4

py = 0x6dfee8, *py = 6, &py = 0x6dfee0

px-py= 1

py++

py = 0x6dfeec,*py = 5,&py = 0x6dfee0

px = py+3

px = 0x6dfef8, *px = 7208852, &px = 0x6dfee4

px-py= 3

Ponteiros e matrizes

- **O compilador transforma matrizes em ponteiros na compilação**, pois a arquitetura do microcomputador compreende ponteiros, e não matrizes.
- Qualquer operação feita com índices de uma matriz pode ser feita com ponteiros.
- O **nome de uma matriz** representa seu **endereço de memória** - endereço do primeiro elemento da matriz.
- O **nome de uma matriz** é um **ponteiro** que aponta para o **primeiro elemento da matriz**.

```
void main()    {
   int M[5]={92, 81, 70, 69, 58};
   cout << "\n Imprime os valores dos
                        elementos de uma
matriz.";
   for (int i=0; i<5; i++)
      cout << "\n " << M[i];

   cout << "\n Imprime com  notação
ponteiro";
   for (int i=0; i<5; i++)
      cout << "\n " << *(M+i);
   }
}
```

A expressão ***(M+i)** tem exatamente o mesmo valor de **M[i]**. **M** é um ponteiro para **int** e **aponta** para **M[0]**.

Da aritmética com ponteiros:
se somarmos 1 a M, obteremos o endereço de M[1], **M+2 é o endereço de M[2]**, etc.

Regra geral:
M + i é equivalente a &M[i], logom *(M + i) é equivalente a M[i]

Ponteiro variável: lugar na memória que armazena um endereço.
Ponteiro constante: é um endereço, uma simples referência (nome de matriz).

Exemplo de ponteiro variável

```
#include <iostream>
using namespace std;
//Imprime os valores dos elementos de uma matriz usando um ponteiro variável
void main()     {
   int M[5]={92, 81, 70, 69, 58};
   int *p=M;          //Ponteiro para int inicializado com o nome da matriz
   for (int i=0; i<5; i++)
   {
      cout << "\n " << *(p++);
   }
}
```

O ponteiro p contém inicialmente o endereço do primeiro elemento da matriz M[0].

Para acessar o próximo elemento, basta incrementar **p** de um.

Após o incremento, **p** aponta para o próximo elemento, **M[1]**, e a expressão ***(p++)** representa o conteúdo do segundo elemento.

Os elementos da matriz são acessados em ordem.

Passando matrizes como argumento para funções

```
#include <iostream>
using namespace std;
float media (float *lista, int tamanho);      //float *lista - Declara um ponteiro variável
int main()
{
   const int MAXI=20;       float notas[MAXI];          int i;
   for (i=0; i<MAXI; i++)
   {
      cout << "Digite a nota do aluno: " << (i+1) << ": ";
      cin >> *(notas+i);
      if ( *(notas+i) < 0 ) break;
   }
   float m = media (notas, i);
   cout << "\n\n Média das notas: " << m;
}
float media (float *lista, int tamanho)
{
   float m=0;
   for (int i=0; i<tamanho; i++)
      m+= *lista++;
   return (m/tamanho);
}
```

A instrução *lista++ é interpretada como *(lista++) e o ponteiro é incrementado.

Ponteiros e Strings

Strings são matrizes do tipo char -> A notação ponteiro pode ser aplicada a strings do mesmo modo que é aplicada a matrizes.

```
#include <iostream>
#include <stdio.h>
#include <locale.h>
//Procura um caractere numa cadeia de caracteres
using namespace std;
 char * procura (char *s, char ch);
void main()
{
      setlocale(LC_ALL, "");
      char ch, str[81], *ptr, letra;
      cout << "\n Digite uma frase: ";          gets (str);
      cout << "\n Digite um caractere: ";      cin >> letra;

      ptr = procura (str, letra);
      cout << "\n A frase começa no endereço: " << unsigned (str);

      if (ptr) {
         cout << "\n Primeira ocorrência do caractere " << letra << ": " <<
       unsigned(ptr);
         cout << "\n A sua posição é: " << unsigned(ptr-str);
      }else
         cout << "\n O caractere " << letra << " não existe nesta frase";
}
```

```
char * procura (char *s, char ch)
{
   while (*s != ch && *s != '\0')  s++;
   if (*s != '\0')   return s;
   return (char *)0;
}
```

Exemplo de saída:

Digite uma frase: Eu sou feliz!

Digite um caractere: u

A frase começa no endereço: 2686619

Primeira ocorrência do caractere u: 2686620

A sua posição é: 1

- O endereço da matriz **str** é usado como argumento na chamada à função **procura()**. Este endereço é uma constante, mas, quando passado por valor, uma variável é criada para armazenar uma cópia dele. A variável é um ponteiro **char** de nome **s.**
- A expressão **s++** avança o ponteiro para sucessivos endereços dos caracteres da cadeia.

Função de biblioteca para manipulação de strings

```
#include <iostream>
#include <locale.h>
using namespace std;
//Algumas funções para manipulação de strings.

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

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

int main()
{
   char s1[]="Oi querida";
   char s2[]="O que foi?";
   cout << strlen("João e Maria");
   cout << "\ns1==s2?" << strcmp(s1,s2);
   strcpy(s1, s2);
   cout << "\n" << s1;
   cout << "\ns1==s2?" << strcmp(s1,s2);
}
```

Ponteiros para uma cadeia de caracteres constante

```
#include <iostream>
#include <locale.h>
using namespace std;
//Mostra duas diferentes inicializações de strings
int main() {
   setlocale(LC_ALL,"");
   char s1[] = "Saudações!";      //Ponteiro constante: não pode ser alterado
   char *s2  = "Saudações!";      //Ponteiro variável

   cout << "\n" << s1;
   cout << "\n" << s2;

   //s1++;                              //Erro! Não podemos incrementar uma constante
   s2++;                                //OK! Aponta para o próximo caractere
   cout << "\n" << s2;              //Imprime audações
}
```

Alocação e desalocação de memória.

- **Operador new**: para aquisição de memória em tempo de execução. Obtém memória do sistema operacional e retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado.
- Se não houver memória suficiente para satisfazer a exigência da instrução, **new** devolverá um ponteiro com o valor **zero** (**NULL**).
- Uma vez alocada, esta memória continua ocupada até que seja desalocada explicitamente pelo operador **delete**.
- Antes de liberar a memória alocada por **new**, devemos verificar se essa memória foi realmente alocada.
- **Liberar a memória não libera o ponteiro que aponta para ela**. Não é mudado o endereço contido no ponteiro; simplesmente esse endereço não será mais válido. Assim, não se deve usar um ponteiro para uma memória que foi liberada.

Alocação e desalocação de memória.

```
#include <iostream>
#include <locale.h>
#include <string.h>
using namespace std;
//Mostra o operador new na classe
    String
class String
{
private:
   char *str;
public:
   String()
   {
      str = new char;     //Reserva um
    único byte na memória.
      *str='\0';
   }
```

```
String (char *s) {
//Retorna um ponteiro para um bloco do
//de memória do tamanho exato para
//armazenar a cadeia s mais o '\0'.
      str = new char[strlen(s)+1];
      strcpy(str,s);
 }
  ~String()       { if (str)    delete str;}
   void print()    { cout << str; }
};
void main()   {
   setlocale(LC_ALL,"");
   String s="A vida é para ser vivida";
    String s1;
   cout << "\ns="; s.print();
   cout << "\ns1="; s1.print();
}
```

Ponteiros void

```
#include <iostream>
using namespace std;
 int main()
{
 int i=5, *pi;

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

cout << *pv;       //ERRO: void is not a
    //pointer-to-object type
 pi = (int *)pv;    //Solução:
 cout << *pi;       //OK
}
```

- Quando atribuímos um endereço a um ponteiro, este deve ser do mesmo tipo do ponteiro.
- NÃO devemos atribuir um endereço de uma variável **int** a um ponteiro **float**. Exceção:
- Ponteiro do tipo void: ponteiro de propósito geral que pode apontar para qualquer tipo de dado.
- **Declaração**: void *p;
- Para quando for necessário que uma função retorne um ponteiro genérico e opere independentemente do tipo de dado apontado.
- Qualquer endereço pode ser atribuído a um ponteiro void.
- O conteúdo da variável apontada por um ponteiro **void** não pode ser acessado por meio deste ponteiro. Deve-se criar outro ponteiro e fazer a conversão de tipo na atribuição.

Dimensionando matrizes em tempo de execução

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

float media (float *lista, int tamanho);
int main()
{
   int tamanho;
   float *notas;
   cout << "\n Qual o número de notas? ";
   cin >> tamanho;
   notas = new float (tamanho);
   for (int i=0; i<tamanho; i++)    {
          cout << "Digite a nota do aluno
    " << (i+1) << ":";
      cin >> *(notas+i);
   }
 float m = media (notas, tamanho);
    cout << "\n\nMédia das notas: " <<
      m;
    delete [] notas;
}

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

Ponteiros para objetos

- **Os ponteiros podem apontar para objetos da mesma forma que apontariam para qualquer tipo de dad**o.
- Útil para quando se desconhece o número de objetos a serem criados. Neste caso, usa-se **new** para criar objetos em tempo de execução.
- O operador **new** retorna um ponteiro para um objeto sem nome.
- A forma de acesso aos membros de um objeto é por meio do seu endereço, e não de seu nome.
- **O operador de acesso a membros (->)** conecta um ponteiro para um objeto a um membro dele, enquanto que o **operador ponto (.)** conecta o nome de um objeto a um membro dele.

Ponteiros para objetos

```
#include <iostream>
using namespace std;
class Venda {
private:    int npecas; float preco;
public:
   void getvenda() {
      cout << "\n Insira o no.de pecas:";
     cin>> npecas;
      cout << "\n Insira o preco:";  cin>>
     preco;
   }
   void printvenda() const {
      cout << "\nNo. de pecas:"  << npecas;
      cout << "\nPreco:" << preco;
   }
};
```

```
void main ()
{
   Venda A;
   A.getvenda();
   A.printvenda();

  // Objeto sem nome apontado
   //pelo ponteiro B
   Venda *B;
   B = new Venda;
   B -> getvenda();
   B -> printvenda();
}
```

Usando referências

- Criar uma referência a um objeto definido pelo operador **new**.

(Usando a mesma classe Venda anterior)

```
int main ()
{
   Venda& A = *(new Venda);
   A.getvenda();
   A.printvenda();
}
```

- A expressão **new Venda** retorna um ponteiro para uma área de memória, grande o suficiente para armazenar um objeto **Venda**. O objeto original pode ser referenciado por meio da expressão:***(new Venda); este é o objeto apontado pelo ponteiro.**
- Criamos uma referência a este objeto de nome **A. Assim, A** é o próprio nome do objeto e seus membros podem ser acessados usando o operador ponto em vez do operador **->.**

Matriz de ponteiros para objetos

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;
class Nome
{
private:
   char *str;
public:
   int getnome() {
      char nome[100];
      gets(nome);
      str = new char [strlen(nome)+1];
      strcpy (str, nome);
      return strcmp (str, "");
   }
 void print() { cout << str; }
};

void main()
{
   Nome *p[80];
   int n, i;
   for (n=0; ;  n++)  {
      cout << "\nDigite nome ou
    [ENTER] para fim: ";
      p[n] = new Nome;
      if (p[n] -> getnome()==0)   break;
   }
   cout << "\n\n Lista dos nomes:";
   for (i=0; i<n; i++)
      cout << "\n"; p[i]->print();
}
```

Ponteiros para ponteiros

```
/ORDENAÇÃO DE PONTEIROS
enum Boolean {False, True};
class String    {
private:
   char *str;
public:
   int getname() {
      char nome[100];
      gets(nome);
      str = new char [strlen(nome)+1];
      strcpy (str, nome);
      return strcmp (str,"");
   }
   Boolean operator > (String s)    {
   return (strcmp(str, s.str)>0)? True:False; }
   void print() { cout << str; }
};
```

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

int main()  {
   String *p[100];
   int n, i;
   for (n=0; ;  n++)  {
      cout << "\nDigite nome ou [ENTER] para fim ";
       p[n] = new String;
       if (p[n] -> getname()==0)   break;
 }
 cout << "\n\n Lista original:";
   for (i=0; i<n; i++) {
        cout << "\n "; p[i]->print();
   }
 ordena (p,n);
 cout << "\n\n Lista Ordenada:";
   for (i=0; i<n; i++)  {
      cout << "\n"; p[i]-> print();
   }
}
```

Ponteiros para ponteiros

```
/ORDENAÇÃO DE PONTEIROS (cont)
//Ordena ponteiros para os objetos
void ordena(String **p, int n)
{
   String *temp;
   for (int i=0; i<n; i++)   {
      for (int j=i+1; j<n ; j++)
         if (*(*(p+i)) > *(*(p+j)))
            //ou if (*p[i]>*p[j])
            temp = *(p+i);
            *(p+i) = *(p+j);
            *(p+j) = temp;
         }
   }
}
```

O tipo da variável **p** é **String****. Indica que **p** é um ponteiro duplamente indireto

A função **ordena** não recebe o endereço de um objeto, e sim o endereço de uma matriz de ponteiros para objetos do tipo **String.**

o nome da matriz é um ponteiro que aponta para outro ponteiro.

Ponteiros para 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 de um objeto da classe **String**

```
Digite nome ou [ENTER] para fim: Maria Joao
Digite nome ou [ENTER] para fim: Jose Antonio
Digite nome ou [ENTER] para fim: Ana Carolina
Digite nome ou [ENTER] para fim:

Lista original:
Maria Joao
Jose Antonio
Ana Carolina

Lista Ordenada:
Ana Carolina
Jose Antonio
Maria Joao
```

Ponteiros para funções

Ponteiro que aponta para uma função.

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

void doisbeep(void)   {          //Toca o autofalante duas vezes
   cout << '\x07';
   for (int i=0; i<5000;  i++)
      cout << '\x07';
}
int main()   {
   void (*pf) (void);            //pf: ponteiro para uma função void
   pf = doisbeep;                //Sem parênteses - atribui o endereço da função
   (*pf)();                      //Chama a função - equivalente a doisbeep();
}
```

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

Ponteiros para funções

Um dos atributos de uma variável – seu endereço indicando sua posição na memória do computador.

Um dos atributos de uma função – o endereço indicando a localização do corpo da função na memória.

Após a chamada a função, o sistema transfere controle a essa localização para executar a função. Logo, é possível usar ponteiros para funções! Úteis para 'funcionais' – funções que tomam funções como argumentos, ex. integrais.

Exemplo:

```
double f (double x)  {
        return 2*x;
}
```

f: ponteiro para a função f()

***f**: a própria função

(*f)(7): uma chamada à função

Ponteiros para funções

Exemplo para calcular uma soma: $\sum_{i=n}^{m} f(i)$

Para fazer a soma, devemos fornecer, além dos limites **n** e **m**, uma função **f**. Solução:

```
double soma (double (*f)(double), int n, int m)  {
        double resultado = 0;
        for (int i=n; i<=m; i++)  {
            resultado += f(i);
        return resultado;
}
```

- **A declaração double (*f)(double), significa que f é um ponteiro para uma função com argumento *double* e valor de retorno *double*.**
- Exemplo de chamadas:

```
cout << sum (f,1,5)    << endl;
cout << sum (sin, 3,5) << endl;
```

Exercícios

1) O que significa o operador asterisco em cada um dos seguintes casos:

a) int *p;

b) cout << *p;

c) *p=x*5;

d) cout << *(p+1);

2) Qual o resultado do programa abaixo:

```
#include <iostream>
using namespace std;
int main()
{
   float i=3, j=5;
   float *p=&i, *q=&j;
   cout << "\n"  << *p - *q;
   cout << "\n"  << **&p;
   cout << "\n"  << 3*-*p / *q +7;
}
```

3) Qual a saída do programa?

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

Exercícios

4) Se **i** e **j** são variáveis inteiras e **p** e **q**, 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++;

5) Admitindo a declaração: **int mat[8];** por que a instrução **mat++;** é incorreta

Exercícios

6) Admitindo a declaração: **int mat[8];** quais das seguintes expressões referenciam o valor do terceiro elemento da matriz?

a) *(mat+2);

b) *(mat+3);

c) mat+2;

d) mat+3;

7) O que faz o programa abaixo?

```
#include <iostream.h>
void main()
{
    int mat[]={4,5,6};
    for(int j=0; j<3 ; j++)
        cout << "\n" << *(mat+j);
}
```

8) O que faz o programa abaixo?

```
#include <iostream.h>
void main()
{

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

Exercícios

9) O que faz o programa abaixo?

```
#include <iostream.h>
void main()
{
    int mat[]={4,5,6};
    int *p=mat;
    for(int j=0; j<3 ; j++)
        cout << "\n" << *p++;
}
```

10) Assumindo 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];

11)

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.

12)

O operador **delete**:

a) apaga um programa;

b) devolve memória ao sistema operacional;

c) diminui o tamanho do programa;

d) cria métodos de otimização.

Exercícios

13) Assumindo 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;