Ministério da Agricultura, Pecuária e Abastecimento

Documentos 2

Novembro, 2001

ISSN 1677-9274

Programação Orientada a Objetos Utilizando C++

Embrapa

República Federativa do Brasil

Fernando Henrique Cardoso
Presidente

Ministério da Agricultura, Pecuária e Abastecimento

Marcus Vinicius Pratini de Moraes
Ministro

Empresa Brasileira de Pesquisa Agropecuária - Embrapa

Conselho de Administração

Márcio Fortes de Almeida
Presidente

Alberto Duque Portugal
Vice-Presidente

Dietrich Gerhard Quast
José Honório Accarini
Sérgio Fausto
Urbano Campos Ribeiral
Membros

Diretoria Executiva da Embrapa

Alberto Duque Portugal
Diretor-Presidente

Bonifácio Hideyuki Nakasu
Dante Daniel Giacomelli Scolari
José Roberto Rodrigues Peres
Diretores-Executivos

Embrapa Informática Agropecuária

José Gilberto Jardine
Chefe-Geral

Tércia Zavaglia Torres
Chefe-Adjunto de Administração

Kleber Xavier Sampaio de Souza
Chefe-Adjunto de Pesquisa e Desenvolvimento

Álvaro Seixas Neto
Supervisor da Área de Comunicação e Negócios

Embrapa

Novembro, 2001

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 2

ISSN 1677-9274

Programação Orientada a Objetos Utilizando C++

Evandro Bacarin

Campinas, SP
2001

Embrapa Informática Agropecuária
Área de Comunicação e Negócios (ACN)
Av. Dr. André Tosello s/nº
Cidade Universitária "Zeferino Vaz" – Barão Geraldo
Caixa Postal 6041
13083-970 – Campinas, SP
Telefone/Fax: (19) 3789-5743
URL: http://www.cnptia.embrapa.br
Email: sac@cnptia.embrapa.br

Comitê de Publicações

Amarindo Fausto Soares
Francisco Xavier Hemerly (Presidente)
Ivanilde Dispato
José Ruy Porto de Carvalho
Marcia Izabel Fugisawa Souza
Suzilei Almeida Carneiro

Suplentes
Fábio Cesar da Silva
João Francisco Gonçalves Antunes
Luciana Alvin Santos Romani
Maria Angélica de Andrade Leite
Moacir Pedroso Júnior

Supervisor editorial: *Ivanilde Dispato*
Normalização bibliográfica: *Marcia Izabel Fugisawa Souza*
Capa: *Intermídia Publicações Científicas*
Editoração eletrônica: *Intermídia Publicações Científicas*

1ª edição

Todos os direitos reservados

Bacarin, Evandro.
Programação orientada a objetos utilizando C++ / Evandro Bacarin. — Campinas : Embrapa Informática Agropecuária, 2001.
37 p. — (Documentos / Embrapa Informática Agropecuária ; 2)

ISSN 1677-9274

1. Programação orientada a objetos. 2. Linguagem C++. I. Título. II. Série.

CDD – 005.117 (21.ed.)

© Embrapa 2001

Autor

Evandro Bacarin

M.Sc. em Ciência da Computação, Pesquisador da Embrapa Informática Agropecuária, Caixa Postal 6041, Barão Geraldo - 13083-970 - Campinas, SP.

Apresentação

As atividades de pesquisa da Embrapa Informática Agropecuária, executadas através dos seus núcleos temáticos de Modelagem e Simulação, Infra-estrutura Computacional para Transferência de Informação para o Agronegócio e Bioinformática, exigem o acompanhamento e a adoção de processos de produção de software no estado da arte, para garantir a alta qualidade dos produtos de software desenvolvidos nesses núcleos.

Nesta concepção, é fundamental, para a execução das atividades desta Unidade, que ela dissemine entre o seus pesquisadores informações técnicas que possam ser facilmente assimiladas e utilizadas no desenvolvimento dos seus produtos de software. Este documento, com uma descrição clara e objetiva do modelo de programação orientado a objetos, se encaixa perfeitamente nesta concepção.

José Gilberto Jardine
Chefe-Geral

Sumário

Introdução 9

Objetivos e Classes 10

Visibilidade de Membros de Classes 18

Funções 19

Funções Amigas 20

Sobrecarga de Operadores 21

Hierarquia de Classes 22

Classes Abstratas 26

Criação e Destruição de Objetos 27

Alocação e Liberação Dinâmica de Objetos 30

Atribuição e Iniciação de Objetos 31

Modelos de Classes (Templates) 33

Conclusões 36

Referências Bibliográficas 37

Programação Orientada a Objetos Utilizando C++

Evandro Bacarin

Introdução

O modelo de programação orientado a objetos foi desenvolvido no Centro de Pesquisas da Xerox em Palo Alto durante a década de 70. Estes conceitos são exercitados na linguagem de programação *Smalltalk* apresentados em Goldberg & Robson (1985). O surgimento desta linguagem, entretanto, não simbolizou uma drástica ruptura com os modelos de programação de outras linguagens contemporâneas, outrossim, foi resultado do processo natural de evolução dos paradigmas de linguagens de programação.

Durante a década de 80 a orientação a objetos influenciou várias áreas da Ciência da Computação, mais notadamente as linguagens de programação. O termo "Orientado a Objetos" passou a adjetivar muitos sistemas comerciais.

As próximas seções apresentam uma introdução aos conceitos de orientação a objetos e à linguagem de programação c++ (Stroustrup, 1986). É necessário observar que este resumo não pretende esgotar a discussão sobre orientação a objetos nem substituir os manuais da linguagem c++. Outrossim, pressupõe que o leitor conheça a linguagem c e possua alguma experiência de programação em computadores.

Objetos e Classes

Imagine a seguinte situação: uma biblioteca conta com cerca de duzentos freqüentadores. Como estas pessoas são de plena confiança, aboliu-se o controle de retirada de livros. Convencionou-se que qualquer pessoa poderia retirar o livro de seu interesse e sugeriu-se que em, no máximo, duas semanas fosse recolocado no lugar de onde foi retirado. Em pouco tempo dois fatos ocorrerão: algumas pessoas atrasarão a devolução de seus livros dificultando a utilização por outras pessoas ou estes livros serão recolocados em lugares inadequados, dificultando que sejam encontrados. Em poucos meses a situação da biblioteca poderá ser descrita como caótica.

Em termos de programação, situação semelhante pode ocorrer. Em um programa grande (algumas poucas centenas de linhas), constituído por alguns módulos, se várias variáveis globais forem manipuladas por várias rotinas sem um controle muito rigoroso, a manutenção do programa pode tornar-se bastante difícil. A alteração de uma rotina pode influenciar o comportamento de uma outra.

Como tentativa para evitar estas situações caóticas, algumas linguagens de programação incorporaram o conceito de objetos. Intuitivamente, *objetos* são "indivíduos" que possuem um comportamento bem conhecido e que reagem quando estimulados[1]. O comportamento do objeto pode ser determinado por seu estado interno, porém, este estado interno não pode ser diretamente manipulado ou consultado por outros indivíduos (objetos) que não o próprio.

Por exemplo, um liquidificador, em geral, possui quatro comportamentos bem definidos: estar desligado, estar ligado na velocidade 1, estar ligado na velocidade 2 e estar ligado na velocidade 3. Um outro indivíduo (objeto), um ser humano, pode interagir com o liquidificador estimulando-o a trabalhar na velocidade 2 através da utilização do interruptor

---

[1] Neste caso os objetos são denominados passivos. Existem linguagens de programação que definem o conceito de objetos ativos. Este conceito está fora do escopo deste contexto.

adequado. O estado interno do liquidificador determinará a velocidade em que está trabalhando, mas não é necessário, muito menos recomendável, que a pessoa que o utilizar abra sua carcaça, corte fios, troque resistências ou troque o motor para impor a velocidade desejada.

O liquidificador é estimulado através do toque em algum interruptor. Um objeto é estimulado através da ativação de um método. O *método* é, intuitivamente, como uma mensagem, uma requisição que um indivíduo solicita a outro.

"- José, por favor, corte a grama do jardim."
...
"- João, a grama está cortada!"

No diálogo, João está solicitando a José que corte a grama do jardim. Ou seja, João está ativando o método "cortar grama" que José sabe desempenhar. Note que João não se preocupou e, em geral, não tem domínio da forma que a grama foi aparada, nem quais foram os instrumentos utilizados, nem se José estava com sede, com fome ou cansado. João não conhece nem tem domínio do "estado interno" de José. Novamente, o importante é que João sabe que se pedir a José que apare a grama, o serviço será bem executado.

Outro exemplo. Como José é um excelente jardineiro, outras pessoas quiseram contratá-lo para cuidar de seus jardins. O fato é que existiam mais jardins do que José podia cuidar. José, então, resolveu ensinar seu ofício para outras pessoas. Ele fez uma lista de instrumentos que todo bom jardineiro deve possuir e fez outra lista de todas as tarefas que todo jardineiro deve saber desempenhar. José criou a "classe dos jardineiros".

Exercício

Descreva os intrumentos que você deve usar e atividades que você deve desempenhar na sua profissão.

Uma *classe de objetos* é, portanto, a descrição de um conjunto de objetos que são similares em sua forma e em seu comportamento.

O momento é oportuno para formalizar os conceitos de classes e objetos e tratá-los em termos de linguagens de programação, mais especificamente, c++.

Um *objeto* é uma estrutura de dados, um registro (`struct` em c). Cada componente deste registro é um atributo aos quais podem ser atribuídos valores. Os componentes de um objeto **não** podem ser diretamente manipulados ou consultados por outros objetos. O *estado interno* do objeto é determinado pelo conjunto dos valores de cada um de seus componentes. Os métodos são funções autorizadas para manipular os atributos de objetos.

Objetos são instâncias de classes. *Classes* descrevem (declaram) os atributos que cada uma de suas instâncias deve possuir e os métodos aos quais seus objetos respondem. A classe é o *tipo* do objeto. Considere o exemplo a seguir:

```
#include <assert.h>
#include <iostream.h>


const TAMANHO = 10;

class PilhaInt     // classe de pilha de inteiros
{
  public:
                                // construtores
    PilhaInt ();
    PilhaInt (int tamanho);
                                // destrutor
    ~Pilha ();
                                // metodos
     void empilha (int i);
     //
     // Empilha o valor i.

     int desempilha ();
     //
     // Desempilha o valor inteiro do topo da pilha e
     // retorna-o.

     int vazia ();
     //
     // Retorna 1 se a pilha esta´ vazia; 0, caso
     // contrario.
```

```
      int cheia ();
      //
      // Retorna 1 se a pilha esta´ cheia; 0, caso
      // contrario.

   private:

      void init (int t);

      int   vi[];
      int   topo;  // aponta topo da pilha
      int   tam;    // tamanho maximo da pilha

};


//
//    Implementacao dos metodos, construtores e
//    destrutores de PilhaInt
//

PilhaInt::PilhaInt ()
{
  init(TAMANHO);
}


PilhaInt::PilhaInt (int t)
{
  init(t);
}


PilhaInt::~PilhaInt ()
{
  delete [] vi;
}


void PilhaInt::init (int t)
{
  vi = new int [t];
  assert(vi != NULL);
  tam = t;
  topo = 0;     // topo aponta primeira posicao
             // livre dapilha
}


void PilhaInt::empilha (int i)
```

```
{
  assert(topo < tam);
  vi[topo] = i;
  topo++;
}


int PilhaInt::desempilha ()
{
  assert(topo > 0);
  topo--;
  return vi[topo];
}


int PilhaInt::cheia()
{
  if (topo >= tam)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}


/*
 *   Main
 */

int main ()
{
  PilhaInt p1(20);    // invocado construtor
     //   PilhaInt::PilhaInt(int)
  PilhaInt p2;        // invocado construtor
     //   PilhaInt::PilhaInt()
  char          oper;      // operacao: <e>mpilha ou
     //           <d>esempilha
  int      i;


  cout << "Qual a operacao desejada? "
       << "(<e>mpilha ou <d>esempilha): ";
  cin >> oper;

  oper = tolower(oper);

  while ( (oper == 'e') || (oper == 'd') )
  {
```

```
    if (oper == ´e´)
    {
      // empilha o valor inteiro se a pilha nao esta´
      // cheia.

      if (p1.cheia())
      {
        cout << "PILHA CHEIA: Impossivel empilhar."
             << endl;
      }
      else
      {
        // Ok, posso empilhar mais um numero.

        cout << "empilhar: ";
        cin >> i;

        p1.empilha(i);
      }
    }     // "then"
    else
    {
      // desempilha um valor inteiro se a pilha nao
      // esta´ vazia.

      if (p1.vazia())
      {
        cout << "PILHA VAZIA: Impossivel desempilhar."
             << endl;
      }
      else
      {
        // Ok, posso desempilhar mais um numero.

        i = p1.desempilha(i);

        cout << "desempilhado: " << i << endl;
      }
    }     // else
  }     // while

  return 0;

  // destrutor e´ invocado automaticamente.
}
```

O programa citado declara uma pilha de valores inteiros e a função `main` utiliza esta pilha. Algumas diferenças podem ser notadas. Em lugar de `stdio.h`, é incluído o arquivo `iostream.h`; em lugar de de `printf` é

utilizada a variável `cout` e o operador <<[2]. A formatação das cadeias de caracteres a serem impressas é feita automaticamente.

A linguagem c++ define também comentários de linha que começam por // e terminam ao final da linha, além do tradicional comentário de bloco (/ * */).

Um objeto do tipo `PilhaInt` possui um número máximo de valores inteiros que podem ser armazenados simultaneamente, isto é, o tamanho da pilha é limitado.

A declaração do tipo `PilhaInt` é constituída pela declaração de construtores, destrutores, de métodos e de atributos.

Os *construtores* são rotinas invocadas automaticamente no momento da criação do objeto a fim de deixar o estado interno do objeto em uma configuração consistente. Neste momento, valores podem ser atribuídos a atributos, memória pode ser alocada, arquivos podem ser abertos, etc[3]. O nome de construtores, bem como o nome de qualquer rotina podem ser sobrecarregados, isto é, vários construtores podem possuir o mesmo nome, desde que o número ou tipo de seus parâmetros sejam diferentes.

Construtores são rotinas declaradas dentro do escopo da declaração da classe e possuem o mesmo nome de sua classe.

Destrutor é uma rotina da classe invocada na destruição de um objeto desta classe (item "Criação e Destruição de Objetos"). Não possui argumentos e seu nome é o mesmo nome da classe precedido por "`~`" (`~PilhaInt`, no exemplo).

Os protótipos de `empilha, desempilha, vazia, cheia e init` dentro do escopo da declaração da classe determinam os *métodos* aplicáveis a objetos da classe `PilhaInt`.

O *estado interno* de instâncias de `PilhaInt` é determinado pelos atributos vi, topo e tam, semelhantes a campos de registros (`struct`). Ou seja, cada objeto da classe `PilhaInt` é um registro com três campos. Estes

---

[2] O arquivo stdio.h ainda pode ser incluído e suas funções podem ser utilizadas.
[3] Ver item Criação e Destruição de Objetos.

campos são *privados* (`private`) ao objeto, uma vez que só os métodos da classe `PilhaInt`, quando aplicados, podem diretamente consultá-los e alterá-los. O meio externo (outros objetos) interage com uma instância desta classe através da invocação de seus métodos públicos[4] (`empilha`, `desempilha`, ...)[5].

A implementação de métodos, construtores e destrutores pode ser efetuada no próprio escopo da declaração da classe (método `vazia`, por exemplo) ou após a declaração da classe. No primeiro caso as funções são consideradas "inline" e, por isso, são expandidas no local da invocação. No segundo caso, como diferentes classes podem possuir métodos com o mesmo nome, o nome do método deve ser qualificado com o nome da classe através do operador :: (`PilhaInt::empilha`, por exemplo).

Na função `main` são declaradas duas pilhas. Quando a função inicia sua execução, antes que a primeira linha do programa seja executada, os construtores dos objetos declarados são invocados. Nesta função, as variáveis `p1` e `p2` denotam instâncias da classe `PilhaInt`. No caso de `p1`, é invocado o construtor `PilhaInt:: PilhaInt(int)` que recebe como argumento um valor inteiro que indica o tamanho máximo da pilha.

Na declaração de `p2` é invocado o construtor `PilhaInt::PilhaInt()`. Este construtor não aceita nenhum argumento e cria uma pilha com o tamanho padrão definido pela constante TAMANHO. Ao final da execução da função, o destrutor é invocado duas vezes. Uma para liberar a área alocada para a pilha p1 e outra para liberar a memória reservada a `p2`.

A invocação `p1.empilha(10)` faz com que seja empilhado o valor 10 na pilha `p1`. O comando `y = p2. desempilha()` significa que deve ser desempilhado um valor da pilha p2 e este valor deve ser atribuído à variável `y`.

Exercício

Considere uma garagem na qual carros são estacionados um atrás do outro, isto é, um carro só pode ser retirado se não houver outro carro

[4] Public.

[5] A visibilidade de membros de classes é discutida na próximo item.

atrás impedindo sua saída. Os carros são identificados pelo número de suas placas (sem considerar as letras). Eventualmente o dono do carro X, que não está necessariamente no fim da fila, solicita que seu carro seja retirado da garagem. Escreva um programa, utilizando a classe `PilhaInt`, que simule o funcionamento desta garagem.

Visibilidade de Membros de Classes

Construtores, destrutores, métodos e atributos são denominados membros da classe. Podem ser públicos (`public`), privados (`private`) ou protegidos (`protected`). Um membro *público* significa que seu nome pode ser referenciado por outras funções ou métodos de outros objetos. Membros *privados* são acessíveis para outras funções membros ou funções amigas (item "Funções Amigas") da mesma classe. Membros *protegidos* apenas podem ser referenciados por outras funções membros e funções amigas de sua classe ou de classes derivadas[6] (subclasses). Considere o exemplo que segue:

```
class X
{
  public:
    void f1();
    int  i1;
  private:
    void f2();
    int  i2;
};

void X::f1()
{
  i2 = 0; // ok
  f2();   // ok
  i1 = 1; // ok
}

void X::f2()
{
  i1 = 0; // ok
```

[6] Item Hierarquia de Classes.

```
  i2 = 0; // ok
  f1();   // ok
}

void p1()
{
  X x1;

  x1.i1 = 0;         // ok
  x1.f1();           // ok
  x1.i2 = 0;         // ERRO !!
  x1.f2();           // ERRO !!
}
```

Funções

Em relação a linguagem c, a declaração de funções em c++ apresentam algumas características extras. Sempre deve ser declarado o protótipo da função antes de sua utilização. O protótipo não precisa declarar o nome de seus parâmetros, apenas seus tipos, por exemplo, `int f1 (int, char, char *)`. Duas ou mais funções podem possuir o mesmo nome, desde que suas assinaturas (número e tipo dos parâmetros) sejam diferentes:

```
// prototipos

void f1 (int);
void f1 (double);
void f1 (int,double);

// implementacoes

void f1 (int i)                { ... }      // *1*
void f1 (double j)             { ... }   // *2*
void f1 (int i, double j)      { ... }   // *3*

void f ()
{
  f1(10);      // invoca *1*
  f1(3.14);         // invoca *2*
  f1(10,3.14); // invoca *3*
}
```

Além disso, os últimos parâmetros da função podem possuir valores "default" e seus respectivos argumentos podem ser omitidos na invocação. Se o argumento é omitido, o valor do parâmetro é iniciado com este

valor. Se um argumento é omitido, os argumentos subseqüentes deverão também ser omitidos:

```
void f1 (int x, char y = ´a´,double z = 0.0)    { ... }

void f ()
{
  f(10);         // x = 10, y = ´a´, z = 0.0
  f(10,'x');          // x = 10, y = ´x´, z = 0.0
  f(10,'x',20.0);     // x = 10, y = ´x´, z = 20.0
  f(10,20.0);         // ERRO !!
}
```

Ao contrário de c, a linguagem c++ permite a passagem de parâmetros por referência:

```
void troca (int &x, int &y);

void main ()
{
  int x = 10;
  int y = 20;

  troca(x,y);
}

void troca (int &x, int &y)
{
  int z;
  z = x; x = y; y = z;
}
```

A passagem de apontadores por referência pode ser declarada como, por exemplo,

```
void troca (int *&p1, int *&p2)
```

Funções Amigas

Uma função não-membro pode ser declarada *amiga* (`friend`) de uma classe. Esta função pode, então, manipular os nomes privados (`private`) e protegidos (`protected`) da classe como se eles tivessem sido declarados públicos (`public`). Por exemplo:

```
class X
{
  public:
    int f1 ();
    char c1;
  private:
    int x2;

  friend void func_amiga(X x1);
};

void func_amiga (X x1)
{
  x1.x2 = 0;    // ok
}

void outra_funcao (X x1)
{
  x1.x2 = 0;    // ERRO !!
}
```

Sobrecarga de Operadores

Podem ser declaradas funções que definem o significado dos operadores da tabela a seguir. Os quatro últimos são, respectivamente, operador de indexação, operador de invocação de função, operador de alocação dinâmica de memória e operador para liberação de memória alocada dinamicamente.

```
+     -     *     /     %     ^     &     |     ^     !
=     <     >     +=    -=    *=    /=    %=    ^=    &=
|=    <<    >>    >>=   <<=   ==    !=    <=    >=    &&
||    ++    --    []    ()    new   delete
```

Os nomes destas funções são compostos pela palavra `operator` seguido pelo operador (`operator+`, `operator-`, `operator[]`, por exemplo). Não é possível, entretanto, alterar a ordem de precedência dos operadores. No exemplo abaixo, são declarados os operadores de soma e multiplicação para números complexos. Note que estes operadores são funções amigas da classe `Complexo` e não membros:

```
class Complexo
{
  public:
    Complexo ()            { re = 0; img = 0; }
    Complexo(double r, double i)  { re = r; img = i; }
  private:
    double re,img;

  friend Complexo operator+ (Complexo,Complexo);
  friend Complexo operator* (Complexo,Complexo);
};

Complexo operator+ (Complexo c1, Complexo c2)
{
  // ...
}

Complexo operator* (Complexo c1, Complexo c2)
 {
  // ...
 }

void f ()
{
  Complexo      x(1, 3.1);
  Complexo      y(1.2, 1.3);
  Complexo      z;

  z = x + y;    // z = operator+(x,y);
  y = x * z;    // y = operator*(x,z);
}
```

Hierarquia de Classes

Uma classe C (*subclasse*) pode tirar proveito de uma outra classe S (*superclasse*) de maneira que membros (métodos e atributos) da superclasse são herdados pela subclasse e podem ser utilizados como se realmente tivessem sido declarados na classe C. A subclasse pode redefinir alguns dos métodos herdados e, possivelmente, acrescentar outros. Diz-se que C *especializa* S.

Considere o exemplo:

```
double abs (double x)
{
  return x < 0 ? -r: r;
}

class Figura
{
  public:
    Figura (double x1,double y1,double x2,double y2);

    virtual double area ()
    { cout << "Nao sei calcular area de Figura"; }

  protected:
    double     x,y,xx,yy;
};

class Retangulo: public Figura
{
  public:
    Retangulo (double x1, double y1, double x2,
               double y2);
    double lado1 () { return abs(x2 - x1); }
    double lado2 () { return abs(y2 - y1); }
    double area ()  { return lado1() * lado2(); }
}

class Circulo: public Figura
{
  public:
    Circulo (double x1, double y1, double r)
    { x = x1; y = y1; raio = r; }

    double area ()
    { return 3.14 * raio * raio; }

  protected:
    double raio;
}
```

A palavra reservada `virtual` significa que o método poderá ser redefinido em suas subclasses. Note que atributos não podem ser redefinidos, apenas métodos. Considere a função a seguir:

```
void f ()
{
  Figura       fig(1,1,10,10);
  Retangulo    ret(10,10,20,20);
  Circulo circ(100,100,20);

  cout << "area figura"    << fig.area()  << endl;
  cout << "area retangulo" << ret.area()  << endl;
  cout << "area circulo"   << circ.area() << endl;
  cout << "area figura"    << fig.area()  << endl;
}
```

A declaração `class Circulo: public Figura` significa que os membros públicos da superclasse `Figura` serão também públicos na classe derivada `Circulo` (subclasse) e o membros protegidos da superclasse também serão protegidos na subclasse.

Uma declaração `class D: private S` indica que os membros públicos e protegidos da classe `S` serão considerados como membros privados na classe derivada `D`.

Uma classe derivada pode, em c++, herdar atributos diretamente de várias superclasses (classes base). Este fenômeno é denominado *herança múltipla*, em contraposição a *herança simples*.

A declaração `class D: public S1, public S2, private S3` significa que a classe derivada `class D` herda publicamente os membros de `class S1` e `S2` e de forma privada os membros de `S3`.

A herança múltipla deve ser utilizada com muito critério. Dizer que uma classe D herda uma classe S (por exemplo, `class Poltrona: public Cadeira`) significa que um objeto da classe D também é uma instância da classe S (uma poltrona é uma cadeira) com alguns atributos ou funcionalidades a mais. Isto é, uma poltrona é uma cadeira com um estofamento mais confortável e com a capacidade de reclinar. Um indivíduo pode sentar-se com mais ou menos conforto na cadeira ou na poltrona.

Por outro lado, afirmar que uma classe derivada `D` possui como superclasses `S1`, `S2` e `S3` (`class Computador:public CPU,public Memoria,public Teclado`) significa que a classe `D` é simultaneamente um objeto da classe `S1`, também da classe `S2` e, também, representa uma instân-

cia da classe S3, isto é, um computador é uma forma de teclado[7] e, ao mesmo tempo, um computador é uma forma de CPU. Neste caso, o correto seria dizer que um computador é constituído por um teclado, por uma memória e por uma CPU, ou seja, CPU, memória e teclado são atributos e não classes base. Note a diferença no exemplo que segue.

```
class Pai { ... };
class Mae { ... };

class Filho: public Pai, public Mae
{ ... };

class Coracao  { ... };
class Cabeca         { ... };
class Perna          { ... };
class Braco          { ... };

class SerHumano
{
  private:
    Coracao          cor;
    Cabeca           cab;
    Perna      pdir, pesq;
    Braco      bdir, besq;
};
```

Um filho herda simultaneamente características genéticas de seu pai e de sua mãe: um traço físico lembra seu pai, outro, lembra sua mãe, etc. De certa forma, um filho representa simultaneamente seu pai e sua mãe.

No segundo modelo, um ser humano é constituído por um coração, por uma cabeça, por duas pernas e por dois braços. Um ser humano não é um coração. O coração é parte do indivíduo. O coração pode ser transplantado, a CPU pode ser trocada, a memória pode ser substituída, mas a carga genética do pai não pode ser retirada do filho.

Para retomar a discussão sobre herança simples, considere as classes `Figura, Circulo e Retangulo` descritas anteriormente e o procedimentos a seguir:

---

[7] É possível colocar um computador inteiro no lugar do teclado de uma máquina de escrever?

```
void imprime_area (Figura f)
{
  cout << "Area da figura: " << f.area() << endl;
}

void g ()
{
  Circulo c(10,11,2);
  Retangulo    r(5,6,20,21);

  imprime_area(c);
  imprime_area(r);
}
```

Quando uma classe `D` é subclasse de `S`, uma instância de `D` também é considerada uma instância de `S`. As classes `Circulo` e `Retangulo` são subclasses de `Figura`. Na função g, c é uma instância de `Circulo` e também de `Figura`. Por isso, `c` e `r` podem ser argumentos da função `imprime_area`.

Na função `imprime_area`, o argumento f denota uma instância de `Figura` (que responde ao método área). As classes `Circulo` e `Retangulo` redefinem a implementação deste método e, por isso, quando o método area é invocado, é executado `Circulo::area`, se `f` corresponde a um círculo ou `Retangulo::area`, se `f` corresponde a um retângulo. Entretanto, dentro de `imprime_area` não é permitido invocar o método `lado1`, pois `f` é da classe `Figura`, mesmo que, em algum momento, `f` corresponda a um objeto da classe `Retangulo`.

Classes Abstratas

Classes abstratas são aquelas que possuem pelo menos um método virtual puro, isto é, sem implementação. Uma classe abstrata somente pode ser usada como superclasse de outras classes. Uma classe abstrata não possui instâncias (por isso abstrata), exceto os objetos de suas classes derivadas, como no exemplo que segue:

```
class Ponto { ... };

class FiguraGeometrica
{
  public:
```

```
    Ponto onde ()        { return centro; }
    void  move (Ponto p) { centro = p; desenhe(); }
    virtual void rotacao (int) = 0; // metodo virtual puro
    virtual void desenhe () = 0;    // metodo virtual puro
  private:
    Ponto    centro;
};
```

Uma classe abstrata não pode ser usada como tipo de argumentos ou de valores de retorno de funções. Porém, apontadores e referências a classes abstradas são permitidas[8].

Classes abstratas permitem que seja expresso um conceito geral, por exemplo, figuras geométricas, e que sejam criadas e utilizadas representações concretas para cada figura geométrica de interesse (círculos, retângulos, elipses, etc.).

Criação e Destruição de Objetos

Objetos possuem um ciclo de vida bem determinado. Em algum momento são criados (instanciados) e, a partir deste instante, podem responder a invocações de seus métodos. Algum tempo depois não são mais necessários e, portanto, são destruídos liberando recursos que por ventura utilizavam. Vale lembrar que, se a classe possui um construtor, este é invocado sempre que um objeto é instanciado e, se a classe possui um destrutor, ele é invocado no momento da destruição de um objeto desta classe. Os momentos de criação e destruição de objetos durante a execução do programa são:

- um objeto é criado quando é encontrada uma declaração de variável e destruído ao final da execução do bloco desta declaração (*objetos automáticos*);
- *objetos estáticos* (`static`) são criados no início da execução do programa e destruídos ao seu final;
- *objetos dinâmicos* são criados através do operador `new` e destruídos pela aplicação do operador `delete`.

[8] Entretanto, referências que requerem objetos temporários em sua iniciação são ilegais.

Durante a criação de vetores de objetos (automática, estática ou dinâmica), o construtor adequado é invocado para cada um dos elementos do vetor. Analogamente, na destruição de um vetor, o destrutor é invocado para cada um de seus elementos.

Exercício

Execute o programa a seguir e observe a seqüência de invocações aos construtores e destrutores:

```
#include <iostream.h>
#include <stdio.h>

class C1
{
  public:
    C1 (char *msg)
    {
      sprintf(m,"%s",msg);
      cout << "Executando construtor C1::C1 => "
           << msg << endl;
    }

    ~C1 ()
    {
      cout << "Executando destrutor C1::~C1 => "
           << m << endl;
    }

  private:
    char  m[128];
};

class C2
{
  public:
   C2 (char *msg)
    {
      sprintf(m,"%s",msg);
      cout << "Executando construtor C2::C2 => "
           << msg << endl;
    }

    ~C2 ()
    {
      cout << "Executando destrutor C2::~C2 => "
           << m << endl;
    }
```

```
  private:
    char  m[128];
    C1         o1("C2::o1");
    C1         o2("C2::o2");
};

C1        g1("variavel global g1");
C2        g2("variavel global g2");

void f ();
void g ();
void h (C1);
void i (C1 &);

void f ()
{
  C1 o1("funcao f, o1");
  C2 o2("funcao f, o2");
}

 void g ()
{
  static C1    o1("funcao g, o1");
  static C2    o2("funcao g, o2");
}

void h (C1 c)
{}

void i (C1 &c)
{}

void main ()
{
  C1 c("funcao main, c);

  cout << "*** Iniciando a execucao do programa *** "
       << endl << endl;
  cout << "  invocando f" << endl;
  f();
  cout << "  invocando g" << endl;
  g();
  cout << "  invocando h" << endl;
  h(c);
  cout << "  invocando i" << endl;
  i(c)
  cout << "** Finalizando a execucao do programa ** "
       << endl;
}
```

Alocação e Liberação Dinâmica de Objetos

Objetos podem ser criados dinamicamente através do operador new e liberados através do operador `delete`[9]. Considere o exemplo da figura que segue:

```
PilhaInt   *p1_p;
Complexo   *c1_p = new Complexo (10.0, 1.2);
Complexo   *c2_p;

pi_p = new PilhaInt;    // aloca memoria para uma
                        // pilha de inteiros
c2_p = new [10] Complexo; // vetor de 10 numeros complexos

pi_p->empilha(10);            // (*pi_p).empilha(10)
*c1_p = c2_p[5] + c2_p[6];

delete c1_p;
delete [] c2_p;        // desaloca vetor
delete pi_p;
```

Quando um vetor de objetos é alocado dinamicamente, o apontador referencia o primeiro elemento do vetor. Na liberação de vetores, o operador [] é colocado após `delete`. Opcionalmente, dentro de [] , pode ser especificada uma expressão que informa o número de elementos do vetor. Em geral, isto é redundante e desnecessário. O operador new retorna 0 se não conseguir alocar a memória solicitada.

Exercício

O tipo `FILE` da biblioteca de c permite que arquivos de caracteres sejam abertos e que caracteres sejam lidos e consumidos um a um.

Implemente uma classe denominada `ArquivoParaCuriosos` que permita que o i-ésimo caractere a frente do próximo caractere a ser lido ($0 \leq i \leq n$) seja consultado sem que nenhum caractere seja consumido. Pelo menos as seguintes operações devem ser implementadas:

- `int abra (char *n_arq, int n)`: abre para leitura o arquivo de nome 'n_arq'. Permite que até o 'n'-ésimo caractere a frente da posição corrente do arquivo seja consultado sem consumação. Retorna 0, se o arquivo não pode ser aberto; 1, caso contrário.

---

[9] Utilizados em lugar das funções malloc e free de c.

- `char leia_carac ()`: retorna o próximo caractere do arquivo. Se o arquivo já foi inteiramente lido ou se está fechado, retorna EOF.
- `char olhe (int i)`: retorna o 'i'-ésimo caractere a frente do próximo caractere a ser lido, sem consumi-lo. Em especial, para i = 0, retorna o mesmo caractere que seria resultado da invocação leia (sem comsumi-lo). Aborta a execução do programa se i > n ou se o arquivo está fechado.
- `void feche ()`: fecha o arquivo.

Não esqueça de implementar o destrutor e os construtores.

Atribuição e Iniciação de Objetos

Considere a classe `String` a seguir:

```
class String
 {
   public:
     String (int tam)
     {
       s = new char [tam];
       t = tam;
     }

     ~String()
     {
       delete s;
     }

        void put (char *p)
     {
       sprintf(s,"%s",p);
     }

   private:
     char     *s;
     int      t;
 };

 void f ()
 {
   String  s1(30);
   String  s2(40);

   s1 = s2;
 }
```

Quando a função `f` for executada, o construtor `String::String(int)` é invocado duas vezes: uma para alocar memória para o objeto s1 e outra para alocar outro bloco de memória para s2. A atribuição s1 = s2 faz com que o valor de cada um dos campos de s2 seja copiado para os

respectivos campos de s1 (*cópia rasa*). Em particular, s1.s contém o endereço do bloco de memória alocado para s2. Este valor (o endereço) é copiado para s1.s e, desta forma, o bloco alocado pelo construtor para s1 não é mais referenciado por nenhum apontador e, por conseqüência, não pode ser liberado e reutilizado. Além disso, o bloco de s2 passa a ser apontado tanto por s1.s quanto por s2.s. Ao final da execução de f, este bloco será liberado na destruição de s1 e na destruição de s2, podendo causar problemas.

Para evitar esta situação, pode ser definido o operador de atribuição para a classe String. Esta implementação tem por finalidade garantir que, durante a atribuição, o bloco de memória alocado para s2 seja duplicado e copiado.

```
void String::operator= (String &a)
{
   char    *p;

   p = new char [a.t + 1];
   strcpy(p,a.s);
   delete s;
   s = p;
   t = a.t;
}
```

A implementação desta função resolve o problema nas atribuições, porém, em:

```
void f ()
{
   String  s1(10);
   String  s2 = s1;
}
```

`String s2 = s1` é uma *iniciação* e não uma atribuição. Assim, a implementação do operador = não é ativada neste caso e, apenas um objeto é construído e dois são destruídos.

Vale enfatizar que atribuição e iniciação são operações distintas em c++. O construtor X(X &) (por exemplo, `String::String(String &)`), denominado *construtor de cópia*, é responsável pela iniciação de um objeto a partir de outro do mesmo tipo. Para a classe `String` poderia ser implementado o seguinte construtor:

```
String::String (String &a)
{
   s = new char [a.t+1];
   t = a.t;
   sprintf(s,"%s",a.s);
}
```

A construção por cópia, também denominada *cópia profunda*, é utilizada em outros dois casos: na passagem de parâmetros por valor (quando o parâmetro formal é iniciado com uma cópia do argumento) e no retorno de funções.

Exercício

Execute o programa a seguir e observe o comportamento dos construtores:

```
#include <iostream.h>

 class X
 {
   public:
     X ()  { cout << "   => X::X()" << endl;   }
     X (X &z)   { cout << "   => X::(X &)" << endl; }

     operator= (X &z)
     {
       cout << "   => X::operator=(X &)" << endl;
       return 0;
     }
 };

 void f (X z)
 {
   cout << " Entrei em f " << endl;
 }

 void g (X &z)
 {
   cout << " Entrei em g" << endl;
 }

 int main ()
 {
   X x1;
   X x2 = x1;

   cout << "Atribuicao x1 = x2" << endl;
   x1 = x2;
   cout << "Invocacao de f" << endl;
   f(x1);
   cout << "Invocacao de g" << endl;
   g(x1);
   return 0;
 }
```

Modelos de Classes (Templates)

No item *Objetos e Classes* foi implementada uma classe de pilhas de valores inteiro (`PilhaInt`). Instâncias desta classe não podem armazenar números reais (`double`). Para fazê-lo, bastaria copiar a declaração desta classe, mudar seu nome e trocar as ocorrências de `int` por `double`.

A linguagem c++ provê um mecanismo para a descrição de modelos de declarações de classes.

```
const TAMANHO = 20;

template <class T> class Pilha
{
  public:
    Pilha ()         { init(TAMANHO); }
    Pilha (int t)    { init(t);       }

    int empilha (T d);
    T   desempilha ();

  private:
    void init (int t);
    T          v[];
    int   topo;
    int   tam;
};

template <class T> void Pilha::init (int t)
{
  v = new T [t];
  assert (v != NULL);
  tam = t;
  topo = 0;
}

template <class T> void Pilha::empilha(T d)
{
  assert(topo < tam);
  v[topo] = d;
  topo++;
}

template <class T> T Pilha::desempilha()
{
  /* ... */
}

struct S { /* ... */ };

int main ()
{
  Pilha<int>        pi, pi2;
  Pilha<S>          ps(50), ps2;
  Pilha<double>    pd(128),pd2;
  S            s;

  pi.empilha(10);
  ps.empilha(s);
  pd.empilha(3.14);
}
```

No exemplo, foi declarado um modelo (*template*) de classe, a partir do qual podem ser criadas automaticamente declarações de diferentes classes para diferentes tipos `T`. A declaração `Pilha<int> pi` significa que `pi` é uma pilha de inteiros. Esta declaração faz com que uma nova declaração de classe seja criada automaticamente no programa de forma que o parâmetro `T` é substituído por `int` e compilado como se tivesse sido escrita pelo programador.

Neste exemplo existem, portanto, a declaração de um modelo de classe, de três classes diferentes (`Pilha<int>`, `Pilha<S>` e `Pilha<double>`) baseadas neste modelo e de seis objetos (`pi`, `pi2`, `ps`, `ps2`, `pd` e `pd2`), instâncias destas classes.

Note que também existem três versões distintas do método `empilha`: para empilhar um valor inteiro em pilhas de inteiros, outro para empilhar um valor real (`double`) em pilhas de reais e, finalmente, mais um método para empilhar uma estrutura do tipo `S`.

Vale observar que, dependendo do compilador utilizado, o mecanismo de templates pode levar a problemas de ligação de arquivos-objeto. Suponha que o modelo de pilha seja declarado no arquivo pilha.h. Este arquivo é importado (incluído) por m1.cxx e por m2.cxx. Em m1.cxx é declarada a variável `Pilha<int> p1` e em m2.cxx é declarada `Pilha<int> p2`. Os arquivos m1.cxx e m2.cxx são compilados separadamente e seus respectivos arquivos-objeto são ligados posteriormente. Quando m1 foi compilado, foi criada uma declaração para `Pilha<int>`, isto é, foram criados e compilados novos métodos. O mesmo ocorre com m2. No momento da ligação, o ligador encontrará duas declarações do método `empilha` para inteiros e, possivelmente, acusará a multiplicidade da declaração.

Existem três formas de abordar este problema. Na primeira, o programador organiza o código a fim de evitar que a mesma expansão do modelo ocorra em arquivos diferentes. Na segunda, o compilador deve ser "esperto" o suficiente para evitar que a mesma expansão seja executada várias vezes e, por fim, o ligador deve ser inteligente para não se aborrecer com esta multiplicidade.

Como a solução deste problema depende do compilador utilizado, este assunto não será aprofundado, pois foge ao escopo deste documento.

Conclusões

O documento apresentou os conceitos básicos de orientação a objetos e os mecanismos básicos para que sejam utilizados na linguagem de programação c++.

Embora não tenha sido esgotado nenhum dos dois assuntos, espera-se que o leitor seja capaz de escrever seus primeiros programas utilizando o paradigma OO e a linguagem c++. Para vôos mais altos recomenda-se leitura de manuais da linguagem c++ e de publicações sobre orientação a objetos.

O modelo de programação orientada a objetos foi apresentado na linguagem Smalltalk. A leitura de Goldberg & Robson (1985) é bastante interessante. Os conceitos deste paradigma são apresentados de uma forma pura e bastante detalhados.

A linguagem c++ incorporou alguns destes conceitos, entretanto, ela não pode ser denominada orientada a objetos no sentido mais estrito do termo, quando comparada à Smalltalk. Em conseqüência disso, o programador c++ deverá ser mais cuidadoso no exercício da orientação a objetos, a fim de não cometer o que seriam consideradas verdadeiras heresias pelos mais puristas.

Vale observar que a linguagem c++ é bastante complexa e possui meandros que apenas podem ser desvendados a custa de bastante suor. São aconselhadas as leituras de Stroustrup (1986) e Ellis & Stroustrup (1993). O primeiro é o manual de referência da linguagem, escrito por seu autor. O segundo, também apresenta a linguagem c++, enriquecido com detalhes de projeto e de implementação de seus mecanismos, é útil para a exploração dos “meandros” da linguagem. Manuais de Referência e do Usuário de compiladores comerciais apresentam, em geral, uma boa introdução aos conceitos de orientação a objetos e à linguagem c++.

Referências Bibliográficas

ELLIS, M. A.; STROUSTRUP, B. **C++ manual de referência comentado**. Rio de Janeiro: Campus, 1993.

GOLDBERG, A.; ROBSON, D. **Smalltalk - 80:** the language and its implementation. Reading: Addison-Wesley, 1985. 714 p. (Addison-Wesley Series in Computer Science).

STROUSTRUP, B. **The C++ programming language**. Reading: Addison-Wesley, 1986. 328 p.

Informática Agropecuária

MINISTÉRIO DA AGRICULTURA,
PECUÁRIA E ABASTECIMENTO