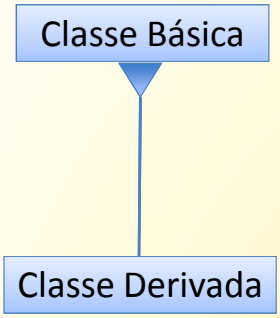
POO e C++: Herança e Polimorfismo

Márcio Santi
Luiz Fernando Martha

Conceito de Herança em POO

- Recurso que torna o conceito de classe mais poderoso;
- Permite que se construa e estenda continuamente classes desenvolvidas, mesmo por outras pessoas;
- Projeto OO: o objetivo é desenvolver classes que modelem e resolvam um determinado problema;
- Com o mecanismo de herança essas classes podem ser desenvolvidas incrementalmente a partir de classes básicas simples.

Conceito de Herança em POO



Conceito de Herança em POO

- Cada vez que se deriva uma nova classe, a partir de uma já existente, pode-se herdar algumas ou todas as características da classe pai (básica);
- É comum projetos em POO apresentarem até centenas de classes, sempre derivadas de um subconjunto pequeno de classes básicas;

Conceito de Herança em C++

```
class Caixa {
 public:
   int altura, largura;
   void Altura(int a)  { altura=a; }
   void Largura(int l) { largura=l; }
};

class CaixaColorida : public Caixa {
 public:
   int cor;
   void Cor(int c) { cor=c; }
};
```

```
void main()
{
  CaixaColorida cc;
  cc.Cor(5);
  cc.Largura(3);  // herdada
  cc.Altura(50);  // herdada
}
```

Conceito de Herança em C++

- Os métodos herdados são usados exatamente como os não herdados;
- Em nenhum trecho do código foi necessário mencionar que os métodos *Caixa::Altura* e *Caixa::Largura* foram herdados
- O uso de um recurso de uma classe não requer saber se este foi ou não herdado;
- Essas características podem ser traduzidas por flexibilidade para o programador;

Conceito de Herança em C++

- O que não é herdado:
  - Construtores;
  - Destrutores;
  - Operadores *new*;
  - Operadores de atribuição;
  - Relacionamentos *friend*;
  - Atributos privados.

Membros de Classe *protected*

- Além dos especificadores de acesso já apresentados: *public* e *private*, existe um outro especificador relacionado estritamente ao conceito de herança: *protected*
- Um atributo *protected* funciona como *private* sob o ponto de vista externo a classe;
- A diferença é que atributos *protected* são visíveis pelas classes derivadas, enquanto os *private* não o são;
- Essas características podem ser traduzidas por flexibilidade para o programador;

Membros de Classe Protected

```
class A {
 private:
   int a;
 protected:
   int b;
 public:
   int c;
};
```

```
void main()
{
   A ca;
   B cb;

   ca.a = 1; // ERRO! a não é visível (private)
   ca.b = 2; // ERRO! b não é visível de fora (protected)
   ca.c = 3; // válido (c é public)

   cb.a = 4; // ERRO! a não é visível nem internamente em B
   cb.b = 5; // ERRO! b continua protected em B
   cb.c = 6; // válido (c continua public em B)
}
```

```
class B : public A {
 public:
   int geta() { return a; }  // ERRO!! a não é visível
   int getb() { return b; }  // válido (b protected)
   int getc() { return c; }  // válido (c public)
};
```

Herança: Construtores e Destrutores

- Quando uma classe é instanciada, o seu construtor é chamado. Se a classe for derivada de alguma outra, o construtor da classe base é chamado anteriormente;
- Se a classe base também é derivada de outra, o processo é repetido recursivamente até que uma classe não derivada seja alcançada;
- Isso é fundamental para se manter consistência para o objeto recém criado;

Herança: Construtores e Destrutores

```
class Primeira {};
class Segunda: public Primeira {};
class Terceira: public Segunda {};
```

Quando a classe Terceira é instanciada, os construtores são chamados da seguinte maneira:

```
Primeira::Primeira();
Segunda::Segunda();
Terceira::Terceira();
```

Herança: Construtores e Destrutores

```
class Primeira {
   int a, b, c;
 public:
   Primeira(int x, int y, int z) { a=x; b=y; c=z; }
};

class Segunda : public Primeira {
   int valor;
 public:
   Segunda(int d) : Primeira(d, d+1, d+2) { valor = d; }
   Segunda(int d, int e);
};

Segunda::Segunda(int d, int e) : Primeira(d, e, 13)
{
   valor = d + e;
}
```

Herança: Construtores e Destrutores

- Se uma classe base não possui um construtor sem parâmetros, a classe derivada tem que, obrigatoriamente, declarar um construtor, mesmo que esse construtor seja vazio:

```
class Base {
 protected:
   int valor;
 public:
   Base(int a) { valor = a; }
   // esta classe não possui um construtor
   // sem parâmetros
};
```

Herança: Construtores e Destrutores

```
class DerivadaErrada : public Base{
 public:
   int pegaValor() { return valor; }
   // ERRO! classe não declarou construtor, compilador não
   // sabe que parâmetro passar para Base
};
```

```
class DerivadaCerta: public Base {
 public:
   int pegaValor() { return valor; }
   DerivadaCerta() : Base(0) {}
   // CERTO: mesmo que não haja nada a fazer
   // para inicializar a classe,
   // é necessário declarar um construtor
   // para dizer com que parâmetro
   // construir a classe Base
};
```

Herança: Pública x Privada

- Por *default* as heranças são *private*. É por isso que nos exemplos acima especificou-se sempre as classes em herança da seguinte forma: *class B : public A {...}*
- Herança Privada: todos os atributos herdados tornam-se *private* na classe derivada;
- Herança Pública: Os atributos *public* e *protected* assim permanecem na classe derivada;

Conceito de Polimorfismo em POO

- Polimorfismo = ***poli*** + ***morphos***
- Polimorfismo descreve a capacidade de um código de programação comportar-se de ***diversas formas*** dependendo do contexto;
- É um dos recursos mais poderoso de linguagens orientadas a objetos:
  - Permite trabalhar em um nível alto de abstração;
  - Facilita a incorporação de novos pedaços em um sistema existente;

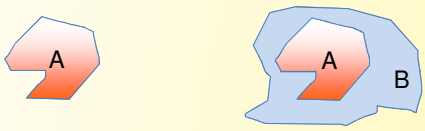
Conceito de Polimorfismo em C++

- EM C++ polimorfismo se dá através da conversão de ponteiros (ou referências)
- Utiliza-se objetos em hierarquia de classes:

```
class A {
   public: void f();
};

class B: public A {
   public: void g();
};
```

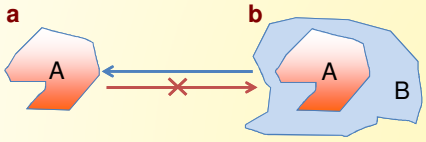
Conceito de Polimorfismo em C++

- Como B é derivado de A, todos os membros disponíveis em A também estarão em B;
- B é um super-conjunto de A: todas as operações que podem ser feitas com objetos de A também o podem através de objetos de B;
- Um objeto da classe B também é um objeto da classe A: isso significa a possibilidade de se converter um objeto de B para A



Conceito de Polimorfismo em C++

```
void main()
{
   A a, *pa; // pa pode apontar para objetos do tipo A e derivados
   B b, *pb; // pb pode apontar para objetos do tipo B e derivados

   a = b;    // copia a parte A de b para a (não é conversão)
   b = a;    // erro! a pode não ter todos elementos para a cópia
   pa = &a; // ok
   pa = &b; // ok, pa aponta para um objeto do tipo B
   pb = pa; // erro! pa pode apontar para um objeto do tipo A
   pb = &b; // ok
   pb = &a; // erro! pb não pode apontar para objetos do tipo A
}
```



Polimorfismo em C++

```
void chamaf(A* a) // pode ser chamada para A e derivados
{
   a->f();
}

void chamag(B* b) // pode ser chamada para B e derivados
{
   b->g();
}

void main()
{
   A a;
   B b;
   chamaf(&a); // ok, a tem a função f
   chamag(&a); // erro! a não tem a função g
               // (a não pode ser convertido para o tipo B)
   chamaf(&b); // ok, b tem a função f
   chamag(&b); // ok, b tem a função g
}
```

```
class A {
   public: void f();
};

class B: public A {
   public: void g();
};
```

Polimorfismo: Redefinição de Métodos em uma Hierarquia

- E se definíssemos dois métodos com mesmo nome nas classes A e B ?
- Não existe sobrecarga em uma hierarquia;
- A definição de métodos com mesmo nome em classes básica e derivada não os deixa disponíveis, mesmo com assinaturas distintas;
- A última definição esconde a anterior;
- Continuam acessíveis, mas não de forma direta;

Polimorfismo: Redefinição de Métodos em uma Hierarquia

```
class A {
   public: void f();
};

class B : public A {
  public:
    void f(int a);      // f(int) esconde f()
    void f(char* str);
};

void main()
{
    B b;
    b.f(10);      // ok, função f(int) de B
    b.f("abc");   // ok, função f(char*) de B
    b.f();        // erro! f(int) escondeu f()
    b.A::f();     // ok
}
```

Polimorfismo: Redefinição de Métodos em uma Hierarquia

- É possível também declarar um método com mesmos nome e assinatura nas classes base e derivada:

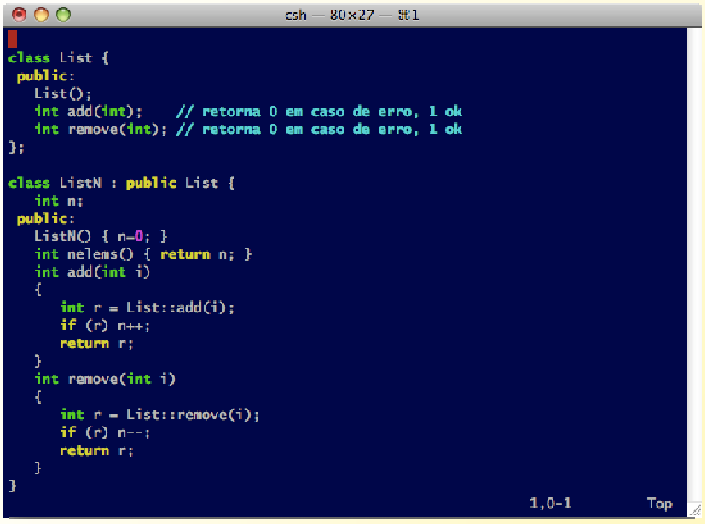
```
class A {
public: void f();
};
class B : public A {
public: void f();
};

void chamaf(A* a) { a->f(); }

void main()
{
   B b;
   chamaf(&b);
}
```

Polimorfismo: Redefinição de Métodos em uma Hierarquia

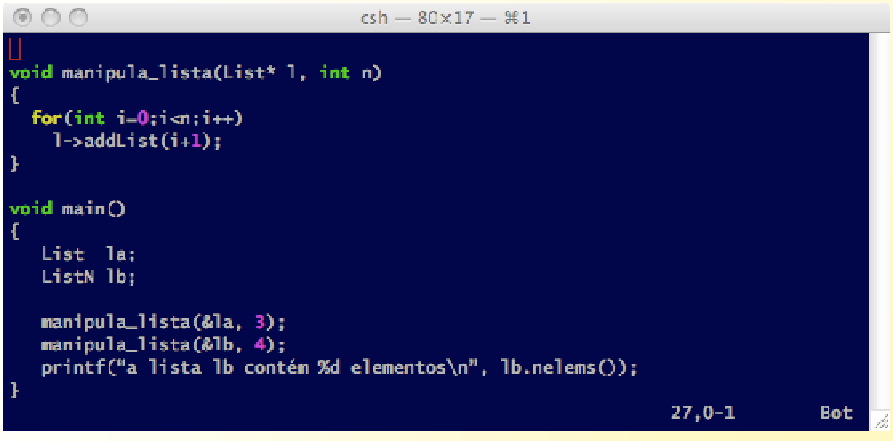
- A função *chamaf* pode ser usada para qualquer objeto da classe **A** e derivados;
- No exemplo acima ela é chamada com um objeto do tipo **B**. O método f é chamado no corpo de *chamaf*;
- Qual versão será executada ?
  - A::f();
  - B::f();

Era esse o comportamento desejado ?

Polimorfismo: Redefinição de Métodos em uma Hierarquia



Polimorfismo: Redefinição de Métodos em uma Hierarquia



Polimorfismo: Redefinição de Métodos em uma Hierarquia

- A função *manipula_lista* utiliza apenas os métodos *add* e *remove*;
- Qual o resultado que este programa vai imprimir na tela ?
  - ***a lista lb contém 0 elementos.***
- Os métodos chamados serão *List::Add* e *List::Remove*, que não alteram a variável *n*;
- A manipulação deixou o objeto inconsistente internamente;
- Isso só seria possível caso os métodos chamados fossem *ListN::Add* e *ListN::Remove*

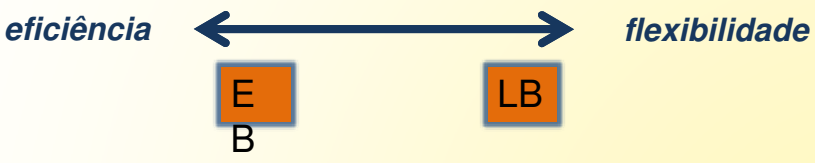
Early Binding / Late Binding

- *Early Binding* (**EB**): ligação dos identificadores em tempo de compilação;
- *Late Binding* (**LB**): ligação que permite a amarração dos identificadores em tempo de execução do programa;
- Para todos os exemplos apresentados até aqui, todas as amarrações de funções e métodos foram feitas em tempo de compilação (*early binding*);

Early Binding / Late Binding

- O problema de **EB** é que o programador precisa saber quais objetos serão usados para as chamadas dos seus métodos;
- Linguagens convencionais como C, FORTRAM, PASCAL só utilizam **EB**;
- A vantagem de **EB** é a eficiência computacional;

Early Binding / Late Binding



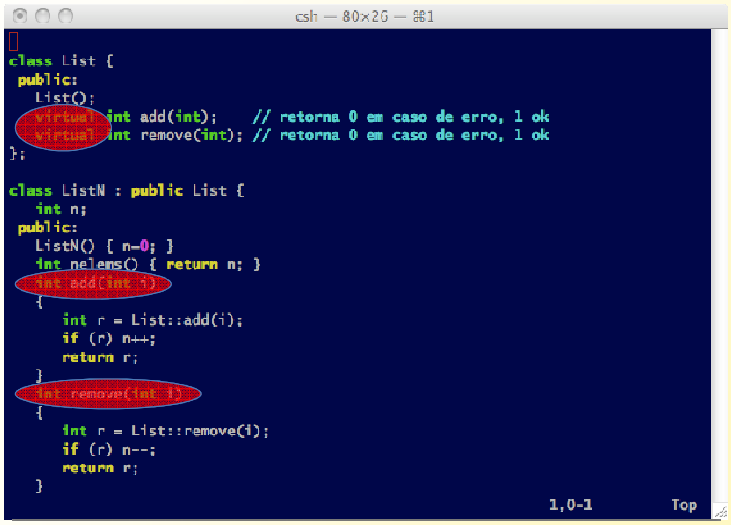
Linguagem C++: Híbrida

- C++ não é uma linguagem procedural tradicional, mas também não é uma linguagem orientada a objetos pura: é uma linguagem híbrida;
- C++ oferece **EB** e **LB**: o programador controla quando usa um ou o outro;
- A vantagem de **EB** é a eficiência computacional;

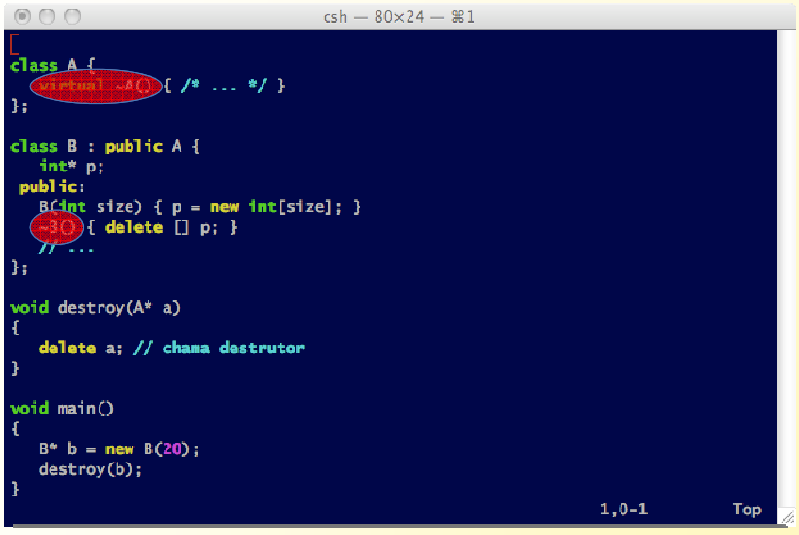
Polimorfismo: Métodos Virtuais

- Em C++, **LB** é especificado declarando-se um método como virtual. **LB** só faz sentido para objetos que fazem parte de uma hierarquia de classes;
- Se um método *f* é declarado virtual em uma classe *Base* e redefinido na classe *Derivada*, qualquer chamada a *f* a partir de um objeto do tipo *Derivada*, mesmo via um ponteiro para *Base*, executará *Derivada::f*;
- A redefinição de um método virtual é também virtual (implícita e explicitamente);

Polimorfismo: Métodos Virtuais



Polimorfismo: Destrutores Virtuais



Classes Abstratas

- Classes abstratas estão em um nível intermediário entre especificação e código de programação;
- Uma classe abstrata é quase uma especificação; ao mesmo tempo é um elemento da linguagem de programação;
- Estas classes permitem a definição das interfaces dos objetos sem entrar em detalhes de implementação;

Classes Abstratas em C++

- Em C++, classes abstratas são aquelas que apresentam ao menos um método virtual puro;
- C++ permite que uma classe apresente métodos sem implementação;
- Métodos sem implementação são sempre virtuais
- Métodos definidos em uma classe sem implementação, apenas com a definição de sua assinatura, é denominado ***método virtual puro***;

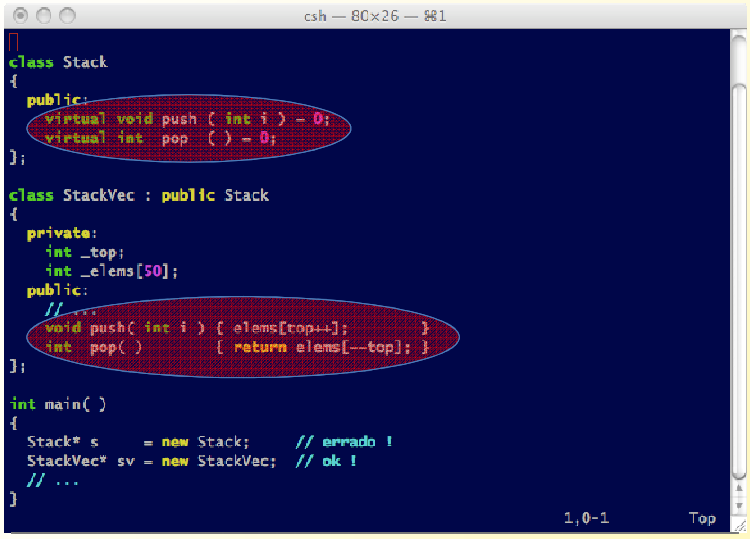
Classes Abstratas: métodos virtuais puros

- um método virtual puro em C++ é definido definindo-o como virtual e atribuindo-lhe o valor zero:

```
class Stack {
 public:
   virtual void push(int) = 0;
   virtual int  pop() = 0;
   virtual int  empty() = 0;
};
```

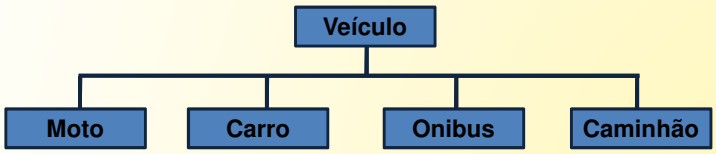
- Uma classe abstrata não pode ter um objeto instanciado diretamente. É necessária a definição de uma classe derivada com a implementação de todos os métodos definidos na classe abstrata como virtuais puros.

Classes Abstratas: métodos virtuais puros



Classes Abstratas: Motivação

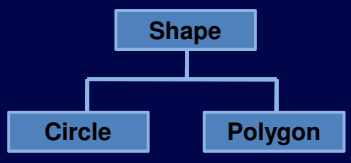
- Especificação de interface, ou herança de tipo;
- Manipulação de objetos utilizando o recurso de polimorfismo;
- Nesse caso pode-se definir classes genéricas para representar um super-conjunto de sub-classes, que por sua vez vão representar os objetos instanciados;



Classes Abstratas: Trabalho

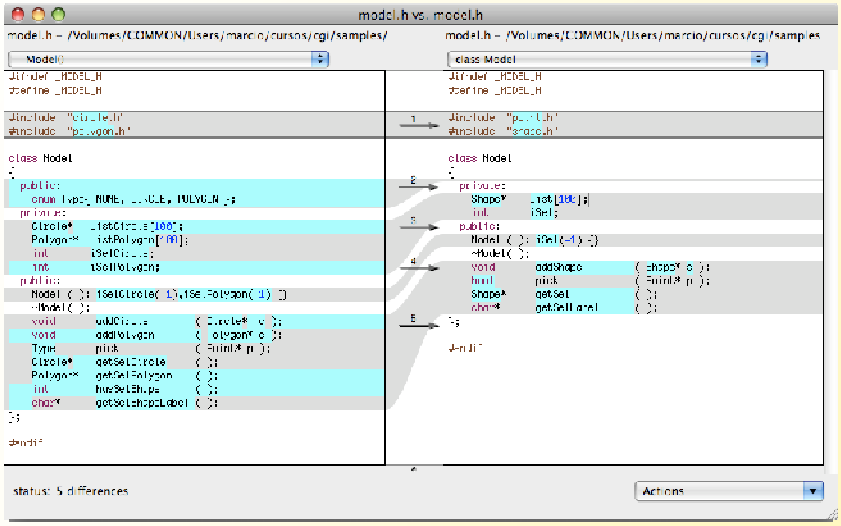
csh — 80×25 — ⌘1

```
class Shape
{
  public:
    enum Type { CIRCLE=1, POLYGON=2 };
  protected:
    char* label;
    Type  t;
    static int nObjs;
  public:
    Shape( );
    virtual ~Shape( );

    void  setLabel( char* lab );
    virtual void move( float dx, float dy ) = 0;

    Type          getType( );
          char* getLabel( );
    virtual float getPerimeter ( ) = 0;
    virtual float getArea      ( ) = 0;
    virtual int   isPointIn    ( Point* p ) = 0;

    static int getNumObjects( );
};
```



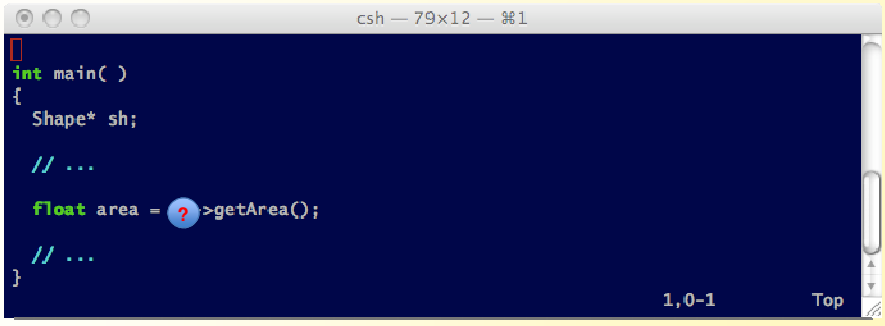
1,0-1 Top

Classes Abstratas: Trabalho (Classe Model)

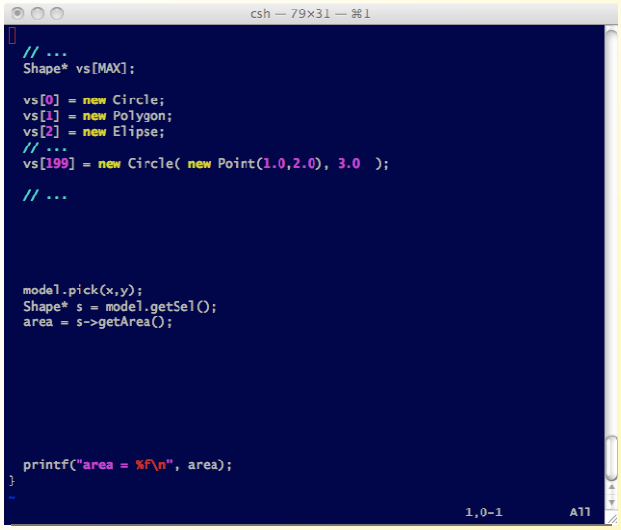


Classes Abstratas: Exemplos

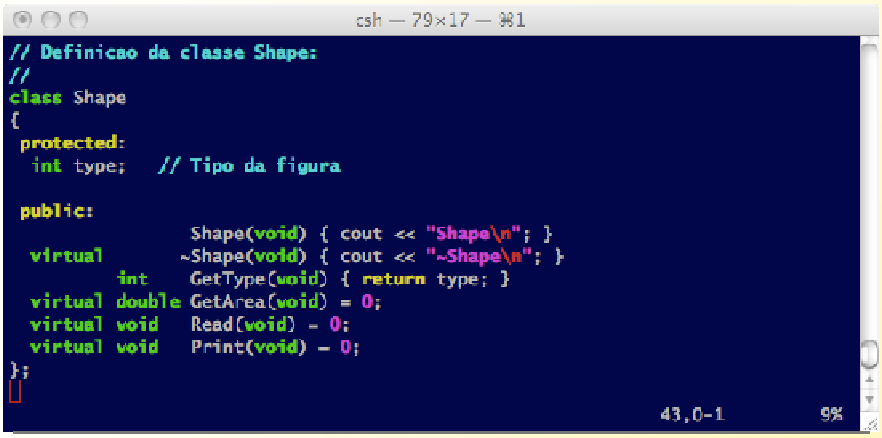
- Os objetos podem ser acessados sempre partindo-se do ponteiro para objetos da classe básica e os recursos de polimorfismo e LB permitem que o programador faça chamadas dos métodos sem saber para que tipo este endereço de memória está apontando;



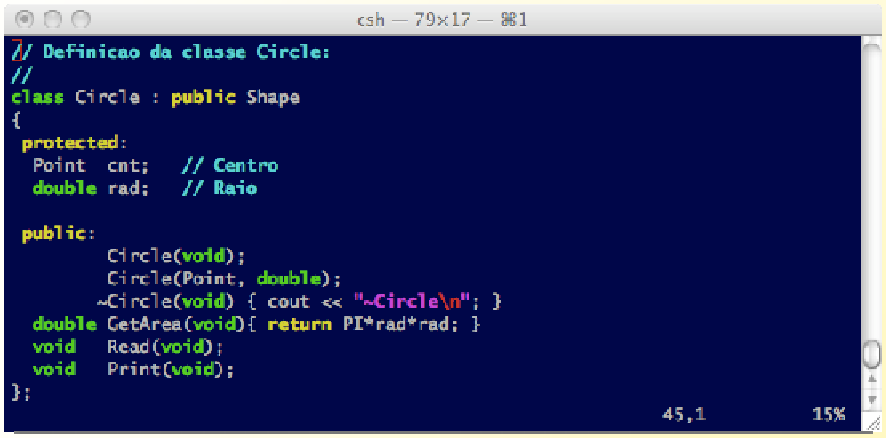
Classes Abstratas: Exemplos



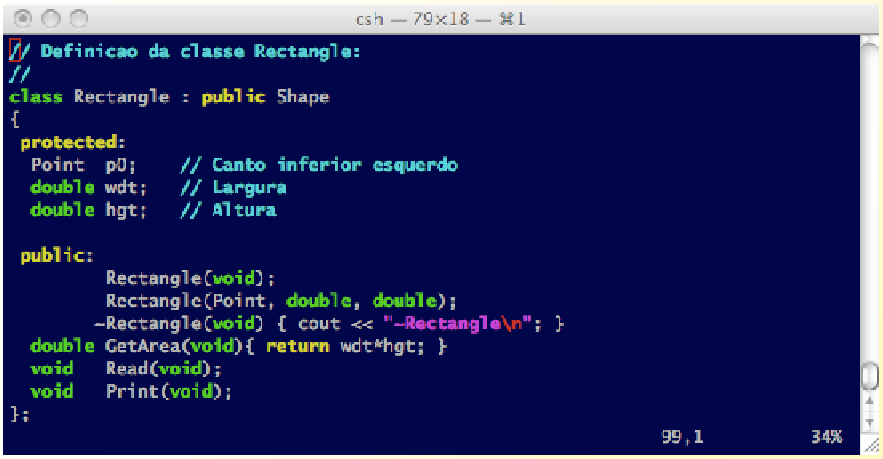
Classes Abstratas: Exemplos



Classes Abstratas: Exemplos



Classes Abstratas: Exemplos



Classes Abstratas: Exemplos

Mar 02, 02 12:34 **shape.cpp** Page 1/5

```
// ------------------------------------------------------------------------
// shape.cpp - classe para representacao de figuras geometricas planas.
// ------------------------------------------------------------------------
// Criado:      31-Out-1999     Marcio Santi
//
// Modificado:
// ------------------------------------------------------------------------
#include <math.h>
#include <iostream.h>

// ------------------------------------------------------------------------
// Simbolos auxiliares:
//
enum
{
  CIRCLE = 1,
  RECTANGLE,
};

typedef struct
{
  double x,y;
} Point;

const double PI = 3.14159265358979323846;

// ------------------------------------------------------------------------
// Definicao da classe Shape:
//
class Shape
{
 protected:
  int type;   // Tipo da figura

 public:
         Shape(void) { cout << "Shape\n"; }
 virtual ~Shape(void) { cout << "~Shape\n"; }
         int  GetType(void) { return type; }
 virtual double GetArea(void) = 0;
 virtual void   Read(void) = 0;
 virtual void   Print(void) = 0;
};

// ------------------------------------------------------------------------
// Definicao da classe Circle:
//
class Circle : public Shape
{
 protected:
  Point  cnt;   // Centro
  double rad;   // Raio

 public:
         Circle(void);
         Circle(Point, double);
        ~Circle(void) { cout << "~Circle\n"; }
  double GetArea(void){ return PI*rad*rad; }
  void   Read(void);
  void   Print(void);
};

// ================================ Circle ================================

Circle :: Circle(void)
{
  cout << "Circle\n";
  type = CIRCLE;
  cnt.x = cnt.y = 0.0;
  rad = 1.0;
}
```

Mar 02, 02 12:34 **shape.cpp** Page 2/5

```
}
// ================================ Circle ================================

Circle :: Circle(Point c, double r)
{
  cout << "Circle\n";
  type = CIRCLE;
  cnt  = c;
  rad  = r;
}

// ================================ Read ================================

void Circle :: Read(void)
{
  cin >> cnt.x >> cnt.y >> rad;
}

// ================================ Print ================================

void Circle :: Print(void)
{
  cout << "Circle\n";
  cout << "Center=(" << cnt.x << "," << cnt.y << ")\n";
  cout << "Radius=" << rad << "\n";
}

// ------------------------------------------------------------------------
// Definicao da classe Rectangle:
//
class Rectangle : public Shape
{
 protected:
  Point  p0;    // Canto inferior esquerdo
  double wdt;   // Largura
  double hgt;   // Altura

 public:
         Rectangle(void);
         Rectangle(Point, double, double);
        ~Rectangle(void) { cout << "~Rectangle\n"; }
  double GetArea(void){ return wdt*hgt; }
  void   Read(void);
  void   Print(void);
};

// ================================ Rectangle ================================

Rectangle :: Rectangle(void)
{
  cout << "Rectangle\n";
  type = RECTANGLE;
  p0.x = p0.y = 0.0;
  wdt = hgt = 1.0;
}

// ================================ Rectangle ================================

Rectangle :: Rectangle(Point p, double w, double h)
{
  cout << "Rectangle\n";
  type = RECTANGLE;
  p0   = p;
  wdt = w;
  hgt = h;
}

// ================================ Read ================================
```

Classes Abstratas: Exemplos

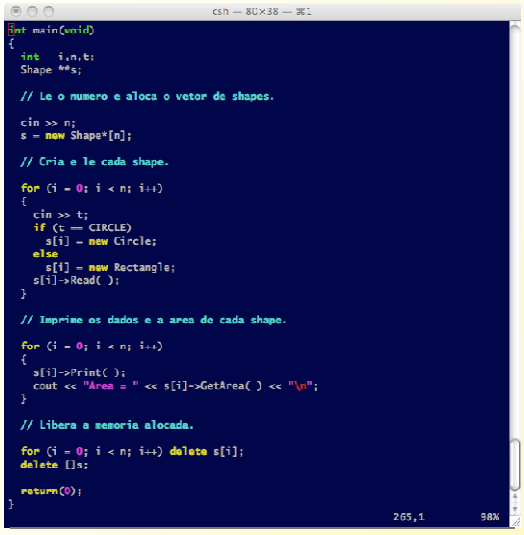
Mar 02, 02 12:34 **shape.cpp** Page 3/5

```
void Rectangle :: Read(void)
{
  cin >> p0.x >> p0.y >> wdt >> hgt;
}

// ================================ Print ================================

void Rectangle :: Print(void)
{
  cout << "Rectangle\n";
  cout << "Point=(" << p0.x << "," << p0.y << ")\n";
  cout << "Width=" << wdt << "\n";
  cout << "Height=" << hgt << "\n";
}

#if 0
// ------------------------------------------------------------------------
// Funcao principal:
//
int main(void)
{
  Circle c;
  cout << "type=" << c.GetType( ) << "\n";
  cout << "area=" << c.GetArea( ) << "\n";

  Rectangle r;
  cout << "type=" << r.GetType( ) << "\n";
  cout << "area=" << r.GetArea( ) << "\n";

  return(0);
}
#endif

// ------------------------------------------------------------------------
// Saida:
//
// Shape
// Circle
// type = 1
// area = 3.14159
// Shape
// Rectangle
// type = 2
// area = 1
// ~Rectangle
// ~Shape
// ~Circle
// ~Shape
//

#if 0
// ------------------------------------------------------------------------
// Funcao principal:
//
int main(void)
{
  Shape *s[2];

  s[0] = new Circle;
  s[1] = new Rectangle;
  for (int i = 0; i < 2; i++)
  {
    cout << "type=" << s[i]->GetType( ) << "\n";
    cout << "area=" << s[i]->GetArea( ) << "\n";
  }

  return(0);
}
```

Mar 02, 02 12:34 **shape.cpp** Page 4/5

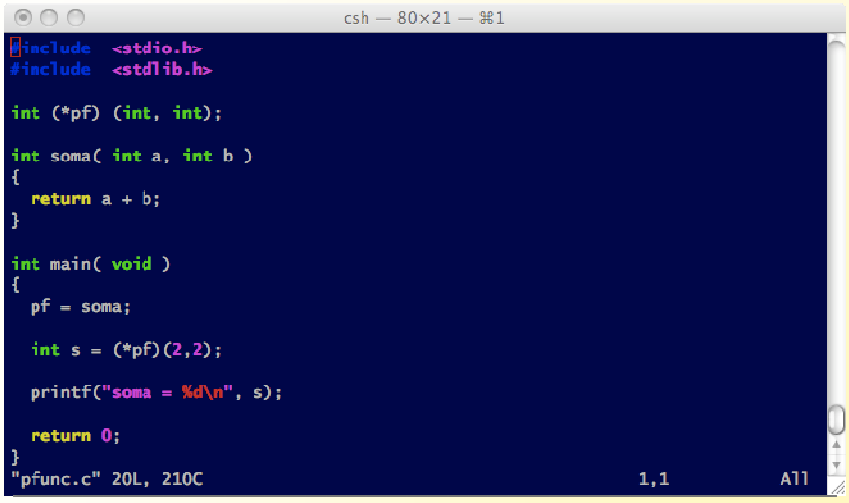
```
#endif

// ------------------------------------------------------------------------
// Saida:
//
// Shape
// Circle
// Shape
// Rectangle
// type = 1
// area = 3.14159
// type = 2
// area = 1
//

#if 0
// ------------------------------------------------------------------------
// Funcao principal:
//
int main(void)
{
  Shape *s[2];

  s[0] = new Circle;
  s[1] = new Rectangle;
  for (int i = 0; i < 2; i++)
  {
    cout << "type=" << s[i]->GetType( ) << "\n";
    cout << "area=" << s[i]->GetArea( ) << "\n";
    delete s[i];
  }

  return(0);
}
#endif

// ------------------------------------------------------------------------
// Saida:
//
// Shape
// Circle
// Shape
// Rectangle
// type = 1
// area = 3.14159
// ~Circle
// ~Shape
// type = 2
// area = 1
// ~Rectangle
// ~Shape
//

// ------------------------------------------------------------------------
// Funcao principal:
//
int main(void)
{
  int   i,n,t;
  Shape **s;

  // Le o numero e aloca o vetor de shapes.

  cin >> n;
  s = new Shape*[n];

  // Cria e le cada shape.

  for (i = 0; i < n; i++)
```

Classes Abstratas: Exemplos



```
int main(void)
{
  int   i,n,t;
  Shape **s;

  // Le o numero e aloca o vetor de shapes.

  cin >> n;
  s = new Shape*[n];

  // Cria e le cada shape.

  for (i = 0; i < n; i++)
  {
    cin >> t;
    if (t == CIRCLE)
      s[i] = new Circle;
    else
      s[i] = new Rectangle;
    s[i]->Read( );
  }

  // Imprime os dados e a area de cada shape.

  for (i = 0; i < n; i++)
  {
    s[i]->Print( );
    cout << "Area = " << s[i]->GetArea( ) << "\n";
  }

  // Libera a memoria alocada.

  for (i = 0; i < n; i++) delete s[i];
  delete []s;

  return(0);
}
```

Tabela de Métodos Virtuais
Ponteiro de Funções



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

int (*pf) (int, int);

int soma( int a, int b )
{
  return a + b;
}

int main( void )
{
  pf = soma;

  int s = (*pf)(2,2);

  printf("soma = %d\n", s);

  return 0;
}
"pfunc.c" 20L, 210C                                  1,1           All
```

Tabela de Métodos Virtuais
Ponteiro de Funções

csh — 80×21 — ⌘1

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

int (*vpf[5])(int, int);

int soma( int a, int b )
{
  return a + b;
}

int main( void )
{
  vpf[3] = soma;

  int s = vpf[3](2,2);

  printf("soma = %d\n", s);

  return 0;
}
                                                    1,1           All
```

Tabela de Métodos Virtuais
Ponteiro de Funções

csh — 80×29 — ⌘1

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

class Svpf
{
   public: int (*vpf[5])(int, int);
};

int soma( int a, int b )
{
  return a + b;
}

int main( void )
{
  Svpf*  var = new Svpf;

  var->vpf[3] = soma;

  int s = var->vpf[3](2,2);

  printf("soma = %d\n", s);

  delete var;

  return 0;
}
~
                                                    1,1           All
```

Tabela de Métodos Virtuais

- Como C++ implementa o **LB** ?

```
class A {
   int a;
 public:
   virtual void f();
   virtual void g();
};
class B : public A {
   int b;
 public:
   virtual void f(); // redefinição de A::f
   virtual void h();
};
void chamaf(A *a) { a->f(); }
```

Tabela de Métodos Virtuais

- A função *chamaf* executará o método *f* do objeto passado como parâmetro;
- Dependendo do tipo do objeto, a mesma linha executará *A::f* ou *B::f*. É como se a função fosse implementada assim:

```
void chamaf(A *a)
{
   switch (tipo de a)
   {
      case A: a->A::f(); break;
      case B: a->B::f(); break;
   }
}
```

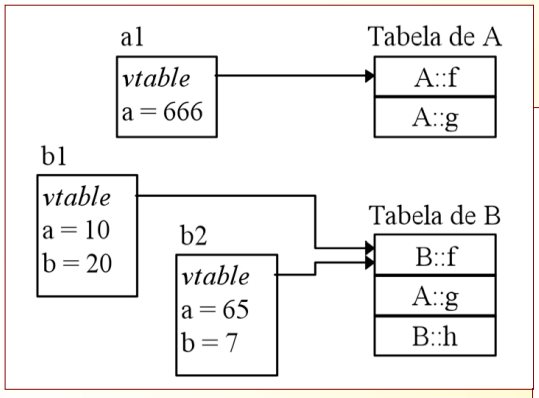
Tabela de Métodos Virtuais

- Mas se na realidade fosse assim não teríamos o comportamento desejado:
  - Cada classe derivada de **A** precisaria de um *case* dentro do *switch*;
  - Esta função não poderia ser utilizada com as futuras classes derivadas;
  - Quanto mais classes, pior seria a eficiência;
  - O compilador não tem como saber o que seria derivado de **A** no futuro
- Conclusão: ***essa implementação é irreal.***

Tabela de Métodos Virtuais

- Na realidade o que C++ utiliza é uma tabela de métodos virtuais (TMV);
- TMV são nada mais que ***vetores de ponteiros de funções***;
- O número de entradas da tabela é igual ao número de métodos virtuais da classe e cada posição guarda o ponteiro para uma função virtual;
- Quando uma classe contém algum método virtual, todos os seus objetos conterão uma referência para essa tabela;

Tabela de Métodos Virtuais




```
class A {
   int a;
 public:
   virtual void f();
   virtual void g();
};
class B : public A {
   int b;
 public:
   virtual void f();
   virtual void h();
};
```

Tabela de Métodos Virtuais

- No exemplo em questão, existem duas tabelas virtuais: uma para a classe A e outra para B;
- A tabela de A tem duas posições, uma para cada método virtual. A tabela de B tem uma posição a mais para o método h.
- A posição dos métodos na tabela é sempre a mesma, ou seja, se na tabela de A a primeira posição apontar para o método f, em todas as classes derivadas a primeira posição será de f.
- Na tabela de A, este ponteiro aponta para A::f, enquanto que em B ele aponta para B::f.
- Quando acontece alguma chamada no código, a função não é chamada pelo nome, e sim por indexação a esta tabela.
- Em qualquer tabela de classes derivadas de A o método f estará na mesma posição, no caso, a primeira:

```
void chamaf(A *a)
{
   a->vtable[0](); // posição 0 corresponde ao método f
}
```