

## Capítulo 6

### *Polimorfismo*

A origem da palavra polimorfismo vem do grego: poli (muitos) e morphos (forma) - múltiplas formas. Polimorfismo descreve a capacidade de um código C++ se comportar de diferentes formas dependendo do contexto em tempo de execução.

Este é um dos recursos mais poderosos de linguagens orientadas a objetos (se não o mais), que permite trabalhar em um nível de abstração bem alto ao mesmo tempo que facilita a incorporação de novos pedaços em um sistema já existente. Em C++ o polimorfismo se dá através da conversão de ponteiros (ou referências) para objetos.

### Conversão de ponteiros

Normalmente se usam não objetos de classes isoladas, mas sim objetos em uma hierarquia de classes. Considere as seguintes classes:

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

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

Como *B* é derivado de *A*, todos os membros disponíveis em *A* (função *f*) também estarão disponíveis em *B*. Então *B* é um superconjunto de *A*, e todas as operações que podem ser feitas com objetos da classe *A* também podem ser feitas com objetos do tipo *B*. A classe *B* é uma especialização da classe *A*, e é não só um objeto do tipo *B*, mas também um objeto do tipo *A*. Nada impede que objetos da classe *B* sejam vistos como sendo da classe *A*, pois todas as operações válidas para *A* são também válidas para *B*. Ver um objeto do tipo *B* como sendo do tipo *A* significa convertê-lo para o tipo *A*. Esta conversão pode ser feita, sempre no sentido da classe mais especializada para a mais básica. A conversão inversa não é permitida, pois operações específicas de *B* não são válidas sobre objetos da classe *A*. Conversão aqui não deve ser entendida como cópia. A simples atribuição de um objeto do tipo *B* para um objeto do tipo *A* copia a parte *A* do objeto do tipo *B* para o objeto do tipo *A*. O polimorfismo é feito através da conversão de ponteiros. O exemplo abaixo mostra as várias alternativas:

```
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
}
```

Para tirar qualquer dúvida sobre quais conversões podem ser feitas, o exemplo abaixo mostra o que pode ser feito com este recurso a partir das classes *A* e *B*:

```
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
}

```

Repare que as funções *chamaf* e *chamag* foram escritas para os tipos *A* e *B*, mas podem ser usadas com qualquer objeto que seja derivado destes. Se um novo objeto derivado de *B* for criado no futuro, a mesma função poderá ser usada sem necessidade de recompilação.

Estas conversões só podem ser feitas quando a herança é pública. Se a herança for privada a conversão não é permitida.

## Redefinição de métodos em uma hierarquia

Não existe sobrecarga em uma hierarquia. A definição de um método com mesmo nome de uma classe básica não deixa os dois disponíveis, mesmo que os tipos dos parâmetros sejam diferentes. Os métodos da classe básica de mesmo nome são escondidos. Eles não ficam inacessíveis, mas não podem ser chamados diretamente:

```

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
}

```

É possível também declarar um método com mesmos nome e assinatura (tipo de retorno e tipo dos parâmetros) que um da classe base. O novo método esconde o da classe base, que precisa do operador de escopo para ser acessado. No entanto, esta redefinição merece atenção especial. Considerando o exemplo:

```

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

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

```

```

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

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

```

A função *chamaf* pode ser usada para qualquer objeto do tipo *A* e derivados. No exemplo acima, ela é chamada com um objeto do tipo *B*. O método *f* é chamado no corpo de *chamaf*. Mas qual versão será executada? No exemplo acima o método executado será *A::f*.

Isto é o que acontece, mas será que este é o comportamento desejado? Se o polimorfismo nesse caso for encarado como uma maneira diferente (mais limitada) de ver o mesmo objeto, não seria natural chamar o método *B::f*? Afinal, o objeto é do tipo *B*, apenas está “guardado” em um ponteiro para o tipo *A*. O comportamento ideal pode não estar claro agora. A seção seguinte procura esclarecer este ponto.

## Exemplo: classes *List* e *ListN*

A classe abaixo implementa uma lista encadeada:

```

class List {
public:
    List();
    int add(int); // retorna 0 em caso de erro, 1 ok
    int remove(int); // retorna 0 em caso de erro, 1 ok
};

```

Suponha que esta declaração é parte de uma biblioteca cujo código fonte não está disponível. Ou seja, a declaração acima faz parte de um header file a ser incluído em arquivos que utilizem a biblioteca.

Através do mecanismo de herança, é possível criar uma nova lista que retorna o número de elementos contidos na lista:

```

class ListN : public List {
    int n;
public:
    ListN() { n=0; }
    int nelems() { return n; }
    int add(int i)
    {
        int r = List::add(i);
        if (r) n++;
        return r;
    }
    int remove(int i)
    {
        int r = List::remove(i);
        if (r) n--;
        return r;
    }
};

```

A nova classe foi criada sem nenhum conhecimento sobre a implementação da classe *List*. Para completar o exemplo, falta utilizar as declarações acima:

```

void manipula_lista(List* l)
{
    // insere e remove vários elementos na lista
}

```

```

void main()
{
    List l1;
    ListN l2;
    manipula_lista(&l1);
    manipula_lista(&l2);
    printf("a lista l2 contém %d elementos\n", l2.nelems());
}

```

A função *manipula\_lista* utiliza apenas os métodos *add* e *remove*, portanto pode operar tanto sobre objetos do tipo *List* quanto do tipo *ListN*. Supondo que esta função retorna deixando cinco elementos na lista, qual será o resultado do printf? Assim como na seção anterior, as funções chamadas em *manipula\_lista* serão *List::add* e *List::remove*. Como estas funções não alteram a variável *n*, o resultado do printf será:

```
a lista l2 contém 0 elementos
```

Ou seja, a manipulação deixou o objeto *l2* inconsistente internamente. Para manter a sua consistência, seria preciso que os métodos *List::add* e *List::remove* fossem chamados, o que não está acontecendo.

## Early x late binding

Nos exemplos acima está acontecendo o que se chama de early-binding. Early-binding é a ligação dos identificadores em tempo de compilação. Quando a função *manipula\_lista* foi compilada, o código gerado tem uma chamada explícita à função *A::f*. Com isso, qualquer que seja o objeto passado como parâmetro, a função chamada será sempre a mesma. Este é o processo de ligação utilizado em linguagens de programação convencionais, como C e Pascal.

O problema de early-binding é que o programador precisa saber que objetos serão usados em todas as chamadas de função em todas as situações. Isto é uma limitação. A vantagem é a eficiência; a chamada à função é feita diretamente, pois o código gerado tem o endereço físico da mesma.

Late-binding é um tipo de ligação que deixa a amarração dos nomes para ser feita durante a execução do programa. Isto é, dependendo da situação a amarração pode ser feita a funções diferentes durante a execução do programa. É exatamente o que falta no exemplo da lista encadeada. A função *manipula\_lista* deve chamar ora *List::add* ora *ListN::add*, dependendo do tipo do objeto.

O problema de late-binding é exatamente a eficiência. O código deve descobrir que função chamar durante a execução do programa. Linguagens orientadas a objetos puras, como Smalltalk, usam exclusivamente late-binding. O resultado é uma linguagem extremamente poderosa, mas com algumas penalidades em relação ao tempo de execução. Por outro lado, ANSI C usa somente early-binding, resultando em alta velocidade mas falta de flexibilidade.

C++ não é uma linguagem procedural tradicional como Pascal, mas também não é uma linguagem orientada a objetos pura. C++ é uma linguagem híbrida. C++ usa early e late binding procurando oferecer o melhor de cada um dos métodos. O programador controla quando usar um ou outro. Para um código em que a execução é determinística, pode-se forçar C++ para usar early-binding. Em situações mais complexas, usa-se late-binding. Desta forma, pode-se conciliar alta velocidade com flexibilidade.

## Métodos virtuais

Em C++, late-binding é especificado declarando-se um método como virtual. Late-binding 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* sobre um objeto do tipo *Derivada*, mesmo que via um ponteiro para *Base*, executará *Derivada::f*. A redefinição de um método virtual é também virtual. A especificação virtual nesse caso é redundante.

Este mecanismo pode ser usado com as listas encadeadas para manter a consistência dos objetos do tipo *ListN*. Basta declarar, na classe *List*, os métodos *add* e *remove* como virtuais:

```

class List {
public:
    List();

```

```

        virtual int add(int);    // retorna 0 em caso de erro, 1 ok
        virtual int remove(int); // retorna 0 em caso de erro, 1 ok
    };

```

Agora a função *manipula\_lista* executará as versões corretas de *add* e *remove*.

## Destrutores virtuais

Não faz sentido construtores poderem ser virtuais, já que eles não são chamados a partir de um objeto, mas sim para criar objetos. Destrutores, apesar de serem métodos especiais, podem ser virtuais porque são chamados a partir de um objeto. A chamada se dá em duas situações: quando o objeto sai do escopo e quando ele é destruído com `delete`. Na primeira situação, o compilador sabe o tipo exato do objeto e portanto chama o destrutor correto. Já na segunda situação isso pode não ocorrer, como mostra o exemplo abaixo:

```

class A {
    // ...
};

class B : public A {
    int* p;
public:
    B(int size) { p = new int[size]; }
    ~B() { delete [] p; }
    // ...
};

void destroy(A* a)
{
    delete a; // chama destrutor
}

void main()
{
    B* b = new B(20);
    destroy(b);
}

```

O que está acontecendo é exatamente o que acontecia na primeira versão das listas encadeadas. A função *destroy* chama (com early-binding) diretamente o destrutor `A::~~A`, quando o certo seria chamar `B::~~B` antes, para depois chamar `A::~~A`. Para forçar este funcionamento correto, é preciso que o destrutor seja virtual (`virtual ~A()`). Caso contrário, objetos podem ser destruídos e deixando alguma coisa para trás.

## Tabelas virtuais

Esta seção tem por objetivo esclarecer um pouco o que acontece por trás dos métodos virtuais, ou seja, como late-binding é implementado. Este conhecimento ajuda o entendimento dos métodos virtuais, suas limitações, poderes e eficiência.

Imaginando a seguinte hierarquia de classes:

```

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(); }

```

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*. Ou seja, é como se a função *chamaf* fosse implementada internamente assim:

```

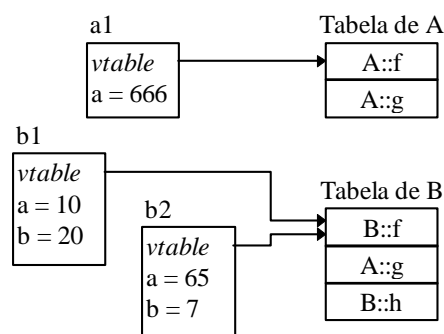
void chamaf(A *a)
{
    switch (tipo de a)
    {
        case A: a->A::f(); break;
        case B: a->B::f(); break;
    }
}

```

Mas isto significa que cada classe derivada de *A* precisa de um case neste switch, o que tem várias desvantagens: esta função não poderia ser utilizada com novas classes que venham a ser criadas no futuro; quanto mais classes na hierarquia pior seria a eficiência e, acima de tudo, o compilador não tem como saber todas as classes derivadas de *A*. Esta análise mostra que esta implementação é irreal, e outra estratégia deve ser usada.

Para resolver este problema, C++ utiliza tabelas virtuais, que são descrições dos métodos de uma determinada classe. Estas tabelas são vetores de funções. O número de entradas da tabela é igual ao número de métodos virtuais da classe e cada posição da tabela tem o ponteiro para uma função virtual. Quando uma classe tem algum método virtual, todos os objetos desta classe terão uma referência para esta tabela, que é o descritor da classe.

No exemplo acima, existem duas tabelas virtuais: uma para a classe *A* e outra para *B*. Quando um objeto é criado, ele leva uma referência para esta tabela. A tabela de *A* tem duas posições, uma para cada método virtual (*f* e *g*). 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. A figura abaixo mostra as possíveis tabelas e objetos para o exemplo de *A* e *B* com alguns objetos na memória:



Considerando que a tabela virtual é um campo de todos os objetos (com nome *vtable* por exemplo), a função *chamaf* pode ser implementada assim:

```

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

```

Esta implementação funcionará com qualquer objeto derivado de *A*, até mesmo de classes futuras, pois não exige que o compilador saiba quais são as classes existentes. A eficiência é sempre a mesma independente de quantas classes existam ou qual a sua altura na árvore de hierarquia. Esta eficiência não é a mesma de uma chamada de função normal (com early-binding) pois envolve uma indireção.

## Classes abstratas - Métodos virtuais nulos

O mecanismo de polimorfismo apresentado até aqui permite a especialização de objetos mantendo sua consistência e sem invalidar códigos que manipulem objetos mais básicos. Esta última característica permite a construção de um sistema inteiro a partir de poucas classes básicas, independente das especializações que vierem a ser feitas.

Se um sistema exige um pilha por exemplo, é possível implementar uma limitada (com vetores por exemplo) e contruir todo o sistema em cima desta. Depois do sistema pronto, pode-se alterar a pilha para uma implementação mais sofisticada (listas encadeadas) sem necessidade de alterar todo o código que manipula a pilha. Basta que a nova pilha seja uma subclasse da pilha original.

Na realidade nem é necessário que a pilha original seja implementada, já que no futuro ela será substituída. Com os recursos vistos até aqui, deve haver uma implementação para todos os métodos da classe, mas C++ permite a declaração de classes sem implementação de alguns métodos. Métodos sem implementação são sempre virtuais e são chamados virtuais puros. Uma classe que tem pelo menos um método virtual puro é chamada de abstrata.

Não é permitido criar objetos de classes abstratas. Estas são feitas apenas para a criação de outras por herança. Classes derivadas de classes abstratas continuam sendo abstratas, a não ser que forneçam implementação para todos os métodos virtuais puros.

Classes abstratas estão em um nível intermediário entre especificação e programa. 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. A partir desta descrição já é possível implementar programas que manipulem estas classes pelo mecanismo de polimorfismo.

Uma aplicação que use uma pilha pode ser totalmente implementada a partir de uma classe abstrata que descreva uma pilha:

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

Apenas com esta declaração já é possível manipular pilhas. Qualquer objeto derivado deste pode ser usado onde se espera um *Stack*. A única restrição é a criação das pilhas. Para criar uma pilha real, é preciso declarar uma classe não abstrata derivada de *Stack*. Não abstrata significa com implementação dos três métodos de *Stack*.

## Herança de tipo x herança de código

Vamos supor que é chegada a hora de implementar a pilha a ser usada na aplicação da seção anterior. Para que os objetos da nova classe possam ser utilizados na a aplicação, a pilha deve herdar de *Stack*:

```
class StackVector : public Stack { ... };
```

Agora vamos imaginar outra situação. É preciso implementar uma classe *Queue* (fila). Para facilitar esta tarefa, será usada uma classe *LinkedList*, que implementa uma lista encadeada. A especificação da fila inclui apenas os métodos *insert*, *remove* e *empty*. A declaração da classe fica assim:

```
class Queue: private LinkedList {
public:
    void insert(int i);
    int remove();
```

```

        int empty();
    };

```

Repare que, para a criação da fila, a herança usada foi a privada. Isto evita que os métodos da lista encadeada possam ser usados diretamente sobre a fila, o que seria um furo na consistência do objeto.

Estas duas situações tem mais diferenças do que uma simples herança pública ou privada. Conceitualmente estas duas heranças estão sendo feitas com objetivos totalmente diferentes. No primeiro caso, o objetivo da herança é simplesmente permitir a utilização da classe *StackVector* como uma *Stack*. Ou seja, aplicações que manipulem *Stacks* também podem ser usadas para manipular *StackVectors*. Nesse caso a herança só é feita para que a classe derivada tenha o tipo da classe base. É a herança de tipo.

No segundo caso, o objetivo não é usar objetos do tipo *Queue* em códigos que manipulem *LinkedList*. A herança está sendo usada para reutilizar o código escrito para implementar listas encadeadas, evitando a re-implementação do que já está pronto. É a herança de código.

Normalmente a herança privada se aplica a uma herança de código, enquanto que a pública se aplica à herança de tipo. Esta diferenciação é importante na hora de projetar sistemas.

### Exemplo: árvore binária para qualquer objeto

Para exemplificar um uso de classes abstratas, esta seção implementa uma árvore binária simples. A primeira versão armazena floats na sua estrutura:

```

class BinTree {
    struct elem {
        elem* right;
        elem* left;
        float val;
        elem(float f) { val=f; right=left=0; }
    } *root;
    int look(elem* no, float f)
    {
        if (!no) return 0;
        if (no->val == f) return 1;
        if (no->val > f) return look(no->left, f);
        return look(no->right, f);
    }
    void put(elem*& no, float f)
    {
        if (!no) no = new elem(f);
        else if (no->val > f) put(no->left, f);
        else put(no->right, f);
    }
public:
    BinTree() { root = 0; }
    int find(float v) { return look(root, v); }
    void insert(float v) { put(root, v); }
};

```

Analisando esta classe, chega-se à conclusão de que a árvore binária em si tem sempre o mesmo comportamento independente do tipo de dado que ela armazena. O tipo float só aparece porque este tipo tem que ser declarado. Se fosse necessária uma árvore binária para armazenar inteiros, a implementação seria exatamente a mesma. A única modificação necessária seria a substituição de todas as palavras float por int.

Com o uso de classes abstratas é possível fazer uma única implementação da árvore binária servir para vários tipos de dados. A idéia é criar uma classe abstrata que represente o tipo de dado a ser armazenado. Qualquer classe derivada desta pode ser usada com a árvore. A implementação da árvore passa a manipular objetos desta classe abstrata. Analisando a implementação acima, nota-se que, para que um tipo de dado seja incluído na árvore binária, ele precisa ter as operações de comparação > e ==. Isto define a classe abstrata:



```

class BinTreeObj {
public:
    virtual int operator==(BinTreeObj&) = 0;
    virtual int operator> (BinTreeObj&) = 0;
};

```

A declaração destes métodos virtuais puros implica que qualquer tipo derivado deste precisa fornecer uma implementação para estes métodos para deixar de ser abstrato. A versão modificada da árvore fica assim:

```

class BinTree {
    struct elem {
        elem* right;
        elem* left;
        BinTreeObj& val;
        elem(BinTreeObj& f) : val(f) { right=left=0; }
    } *root;
    int look(elem* no, BinTreeObj& f)
    {
        if (!no) return 0;
        if (no->val == f) return 1;
        if (no->val > f) return look(no->left, f);
        return look(no->right, f);
    }
    void put(elem*& no, BinTreeObj& f)
    {
        if (!no) no = new elem(f);
        else if (no->val > f) put(no->left, f);
        else put(no->right, f);
    }
public:
    BinTree() { root = 0; }
    int find(BinTreeObj& v) { return look(root, v); }
    void insert(BinTreeObj& v) { put(root, v); }
};

```

Como o polimorfismo se dá pela conversão de ponteiros, a árvore deve armazenar apenas ponteiros ou referências. Para utilizar esta árvore, basta fazer com que o tipo dos objetos a serem armazenados herdem da classe *BinTreeObj*. Para armazenar algum tipo primitivo é necessário criar uma nova classe derivada de *BinTreeObj* que armazene este tipo.

## Conversão de um objeto básico para um derivado

Aproveitando a árvore binária da seção anterior, uma classe *String* será criada com o objetivo de armazenar o tipo *char\** na árvore. A classe deve herdar de *BinTreeObj* e declarar os dois métodos de comparação. Algo do tipo:

```

class String : public BinTreeObj {
    char *str;
public:
    String(char* s) { str = s; }
    int operator==(String& o)
    { return !strcmp(str, o.str); }
    int operator> (String& o)
    { return  strcmp(str, o.str) > 0; }
};

```

No entanto, com a declaração acima, o compilador reclama que a classe *String* continua abstrata quando algum objeto é criado:

```

void main()
{
    BinTree bt;
    bt.insert( * new String("abc") ); // ERRO! String abstrata
}

```

O problema é que os métodos declarados em *String* não estão redefinindo os da classe abstrata, mas escondendo-os. Analisando a assinatura dos dois métodos, percebe-se a diferença:

```

int BinTreeObj::operator==(BinTreeObj& o);
int String      ::operator==(String& o)

```

Se fosse o método de *String* estivesse redefinindo o de *BinTreeObj*, seria possível converter objetos de uma classe mais básica para uma mais específica. Como foi visto com polimorfismo, esta não é uma conversão segura. Na implementação da árvore binária, este método é chamado e o parâmetro é sempre uma referência para um *BinTreeObj*, que pode não ser uma *String*. Para redefinir um método, os parâmetros devem ser ou do mesmo tipo do declarado na classe base ou de um tipo mais básico. Nunca de um tipo derivado. O contrário acontece com o tipo de retorno de uma função.

Chegando à conclusão de que os parâmetros dos métodos de *String* devem ser do tipo *BinTreeObj*, surge outro problema. Como acessar o campo *str* e fazer a comparação das *Strings*? Deve haver alguma maneira de converter um *BinTreeObj* em uma *String*, mesmo que esta conversão não seja segura. Na realidade esta conversão pode ser forçada através de type casts explícitos, assim como em C é possível converter um ponteiro para outro de um tipo completamente diferente. Se o objeto for realmente do tipo desejado, a conversão é feita. Mas se o tipo não for o esperado, os resultados são imprevisíveis. Mas nesse caso, é a única opção:

```

class String : public BinTreeObj {
    char *str;
public:
    String(char* s) { str = s; }
    int operator==(BinTreeObj& o)
    { return !strcmp(str, ((String&)o).str); }
    int operator> (BinTreeObj& o)
    { return  strcmp(str, ((String&)o).str) > 0; }
};

```

Esta com certeza não é a melhor maneira de se implementar uma classe que manipula um tipo qualquer. Existem duas outras maneiras seguras, vistas mais à frente: templates e type casts dinâmicos.

## Herança múltipla

Em C++, a herança não se limita a uma única classe base. Uma classe pode ter vários pais, herdando características de todos eles. Este tipo de herança introduz grande dose de complexidade na linguagem e no compilador, mas os benefícios são substanciais. Considere a criação de uma classe *MesaRedonda*, tendo não só propriedades de mesas, mas também as características geométricas de ser redonda. O código abaixo é uma possível implementação:

```

class Circulo {
    float raio;
public:
    Circulo(float r) { raio = r; }
    float area()      { return raio*raio*3.14159; }
};

class Mesa {
    float ipeso;
    float ialtura;
public:
    Mesa(float p, float a) { ipeso = p; ialtura=a; }
    float peso()           { return ipeso; }
    float altura()         { return ialtura; }
};

```

```

};

class MesaRedonda: public Circulo, public Mesa {
    int icor;
public:
    MesaRedonda(int c, float a, float p, float r);
    int cor() { return icor; }
};

MesaRedonda::MesaRedonda(int c, float a, float p, float r)
    : Mesa(p, a), Circulo(r)
{
    icor = c;
}

void main()
{
    MesaRedonda mesa(5, 1, 20, 3.5 );
    printf("Peso:    %f\n", mesa.peso());
    printf("Altura:  %f\n", mesa.altura());
    printf("Area:    %f\n", mesa.area());
    printf("Cor:     %d\n", mesa.cor());
};

```

Um exemplo natural poderia sair da seção que discute herança de tipo ou código. Para implementar um pilha com listas encadeadas que possa ser usada como *Stack* e aproveitando uma classe já implementada de listas encadeadas, a declaração seria assim:

```

class StackList : public Stack, private LinkedList {
    // ...
};

```

## Ordem de chamada dos construtores e destrutores

Assim como em herança simples, os construtores das classes base são chamados antes do construtor da classe derivada. A ordem de declaração na classe define a ordem de chamada dos construtores. No exemplo acima, a classe foi declarada com uma ordem

```

class MesaRedonda: public Circulo, public Mesa {

```

e o construtor com outra:

```

MesaRedonda::MesaRedonda(int c, float a, float p, float r)
    : Mesa(p, a), Circulo(r)

```

Como a ordem de declaração na classe é a que define, a ordem dos construtores será:

```

Circulo::Circulo
Mesa::Mesa
MesaRedonda::MesaRedonda

```

## Classes básicas virtuais

Classes base virtuais só são usadas com herança múltipla. É uma maneira de o programador controlar como as classes devem ser herdadas. Por exemplo:

```

class A {
public:
    int a;
};

```

```

class B: public A {};
class C: public A {};
class D: public B, public C {
public:
    int valor() { return a; }
};

```

Este código gera uma hierarquia onde a classe *D* tem duas cópias da parte *A*, uma associada a *B* e outra a *C*. Portanto, o código acima gera um erro de compilação:

```
Member is ambiguous: 'A::a' and 'A::a'
```

O problema é que a declaração de *B* herdando de *A* faz com que a classe *B* já tenha uma “parte *A*” incorporada. Nesse caso, *B* já tem a sua variável *a*. O mesmo acontece com *C*. O resultado são duas cópias de *A* na hierarquia. Uma é a “parte *A*” de *B* e outra é a “parte *A*” de *C*. O compilador não sabe que cópia de *a* esta sendo referenciada. O operador de escopo poderia ser utilizado para retirar o erro:

```
int valor() { return C::a; }
```

Às vezes o programador quer montar uma árvore onde só exista uma cópia de *A*. É o caso de usar uma classe base virtual. Declarando uma classe base como virtual faz com que a classe derivada não inclua a classe base. Seria o caso de declarar as heranças de *B* e *C* como virtuais, assim nenhuma das duas teria uma “parte *A*”:

```

class B: public virtual A {};
class C: public virtual A {};
class D: public B, public C {
public:
    int valor() { return a; }
};

```

Agora a função *valor* não precisa mais do operador de escopo, e a árvore gerada terá apenas uma cópia de *A*.

## Chamada de construtores de classes básicas virtuais

Como na herança virtual as classes derivadas não contém a parte da sua classe base, é preciso tomar alguns cuidados na hora de inicializar estas classes básicas. Supondo que a classe *A* tenha um construtor que receba um inteiro:

```

class A {
public:
    int a;
    A(int) {}
};

```

A chamada a este construtor tem que estar explícita no código de *B* e *C*:

```

class B: public virtual A { public: B() : A(1) {} };
class C: public virtual A { public: C() : A(2) {} };
class D: public B, public C {
public:
    int valor() { return a; }
    // erro! compilador não gera construtor vazio
};

```

Se um objeto da classe *B* for criado, sua parte *A* será inicializada com 1. Se for criado um do tipo *C*, a inicialização será com 2. E se o objeto for do tipo *D*? Como a parte *A* é criada diretamente por *D* (herança virtual), o próprio construtor de *D* deve chamar o de *A* diretamente:

```

class D: public B, public C {
public:
    int valor() { return a; }
};

```

```
    D(): A(3) {}  
};
```

### Exercício 10 - Classe List

Implementar uma lista que possa ser utilizada para armazenar qualquer objeto.

A lista só precisa ter um método para inserção de objetos. Além da classe lista, deve ser implementado um mecanismo para percorrer uma lista em dois sentidos: do início para o fim e do fim para o início. Isto não significa que a lista deve ser ordenada, apenas que ela preserva a ordem utilizada na inserção dos elementos. Uma maneira de preservar esta ordem é sempre inserir um elemento novo no início da lista.