Programar em C++/Herança

< Programar em C++

Programar em C++ Encapsulamento Polimorfismo

Índice [esconder]

Conceito [editar | editar código-fonte]

Herança é um dos pontos chave de programação orientada a objetos (POO). Ela fornece meios de promover a extensibilidade do código, a reutilização e uma maior coerência lógica no modelo de implementação. Estas características nos possibilitam diversas vantagens, principalmente quando o mantemos bibliotecas para uso futuro de determinados recursos que usamos com muita frequência.

Uma classe de objetos "veiculo", por exemplo, contém todas as características inerentes aos veículos, como: combustível, autonomia, velocidade máxima, etc. Agora podemos dizer que "carro" é uma classe que têm as características básicas da classe "veículo" mais as suas características particulares. Analisando esse fato, podemos concluir que poderíamos apenas definir em "carro" suas características e usar "veículo" de alguma forma que pudéssemos lidar com as características básicas. Este meio chama-se herança.

Agora podemos definir outros tipos de veículos como: moto, caminhão, trator, helicóptero, etc, sem ter que reescrever a parte que está na classe "veículo". Para isso define-se a classe "veículo" com suas características e depois cria-se classes específicas para cada veículo em particular, declarando-se o parentesco neste instante.

Outro exemplo: Imagine que já exista uma classe que defina o comportamento de um dado objeto da vida real, por exemplo, animal. Uma vez que eu sei que o leão **é** um animal, o que se deve fazer é aproveitar a classe animal e fazer com que a classe leão derive (herde) da classe animal as características e comportamentos que a mesma deve apresentar, que são próprios dos indivíduos classificados como animais.

Ou seja, herança acontece quando duas classes são próximas, têm características mútuas mas não são iguais e existe uma especificação de uma delas. Portanto, em vez de escrever todo o código novamente é possível poupar algum tempo e dizer que uma classe herda da outra e depois basta escrever o código para a especificação dos pontos necessários da classe derivada (classe que herdou).

Sintaxe [editar | editar código-fonte]

Para declarar uma classe derivada de outra já existente, procedemos de forma a declarar o parentesco e o grau de visibilidade (acesso) que a classe derivada terá dos membros de sua classe base. Para isso seguimos o seguinte código sintático:

```
class classe_derivada : [<acesso>] classe_base {
 //corpo da classe derivada
}
```

Repare que temos o operador ":" (dois pontos) como elo entre as duas classes. Este operador promove o "parentesco" entre as duas classes quando é usado na declaração de uma classe derivada.

O termo [<acesso>] é opcional, mas se estiver presente deve ser public, private ou protected. Ele define o grau de visibilidade dos membros da classe base quando a classe derivada precisar acessá-los.

Exemplo de implementação:

```
// Demonstra herança.
#include <iostream>
using namespace std;
class veiculo_rodoviario // Define uma classe base veículos.
   {
       int rodas;
       int passageiros;
   public:
       void set_rodas(int num) { rodas = num; }
       int get_rodas() { return rodas; }
       void set_pass(int num) { passageiros = num; }
       int get_pass() { return passageiros; }
   };
class caminhao : public veiculo_rodoviario // Define um caminhao.
   {
         int carga;
   public:
          void set_carga(int size) { carga = size; }
          int get_carga() { return carga; }
          void mostrar();
   };
enum tipo {car, van, vagao};
class automovel : public veiculo_rodoviario // Define um automovel.
   {
       enum tipo car_tipo;
   public:
       void set_tipo(tipo t) { car_tipo = t; }
       enum tipo get_tipo() { return car_tipo; }
       void mostrar();
   };
void caminhao::mostrar()
       {
       cout << "rodas: " << get_rodas() << "\n";
       cout << "passageiros: " << get_pass() << "\n";
       cout << "carga (capacidade em litros): " << carga << "\n";
       }
void automovel::mostrar()
       {
       cout << "rodas: " << get_rodas() << "\n";
       cout << "passageiros: " << get_pass() << "\n";
       cout << "tipo: ";
       switch(get_tipo())
               {
               case van: cout << "van\n";
```

```
                    break;
                case car: cout << "carro\n";
                    break;
                case vagao: cout << "vagao\n";
                }
        }
 int main()
 {
    caminhao t1, t2;
    automovel c;
    t1.set_rodas(18);
    t1.set_pass(2);
    t1.set_carga(3200);
    t2.set_rodas(6);
    t2.set_pass(3);
    t2.set_carga(1200);
    t1.mostrar();
    cout << "\n";
    t2.mostrar();
    cout << "\n";
    c.set_rodas(4);
    c.set_pass(6);
    c.set_tipo(van);
    c.mostrar();
#ifdef WIN32
    system ("pause");
#endif
    return 0;
 }
```

Na implementação acima temos a classe base veiculo_rodoviario e duas classes derivadas ":" caminhao e automovel. Podemos notar que as características comuns a todos os tipos de veículos, rodas e passageiros, estão na classe base, enquanto as características exclusivas de cada tipo de veículo estão nas classes derivadas. Desta forma podemos definir procedimentos especializados para cada classe, fazendo com que todas as eventuais modificações feitas ao longo da implementação na classe base sejam estendidas a todos os objetos criados a partir das classes derivadas no programa.

Repare ainda um pormenor: tanto a classe "caminhao" quanto a automovel têm como função membro o método mostrar(), mas uma não interfere com a outra. Isto ilustra um outro aspecto da orientação a objeto, o polimorfismo. Este será exposto em mais detalhe nos capítulos subsequentes.

Controle de acesso à classe base [editar | editar código-fonte]

Quando uma classe herda outra, os membros da classe base são incorporados como membros da classe derivada. Devido à separação das classes e do controle de acesso às variáveis em cada classe, devemos pensar como as restrições de acesso são gerenciadas em classes diferentes, principalmente o acesso a membros da classe base a partir das classes derivadas.

O acesso dos membros da classe base à classe derivada é determinado pelo especificador de acesso: **public**, **private** e **protected**. Por "defaut" (padrão) temos o private, ou seja, como temos a opção de não explicitar o especificador de acesso, se este não estiver presente o compilador usará "private" durante a interpretação do código.

Assim ficamos com as possíveis combinações

- Classe base herdada como **public**:
  - Membros públicos (public) da classe base:
    - É como se copiássemos os membros da classe base e os colocássemos como "public" na classe derivada. No final, eles permanecem como públicos.
  - Membros privados (private) da classe base:
    - Os membros estão presentes na classe, porém ocultos como privados. Desta forma as informações estão presentes, mas só podem ser acessadas através de funções publicas ou protegidas da classe base.
  - Membros protegidos (protected) da classe base:
    - Se tivermos membros protegidos (protected) na classe derivada, eles se comportam como se tivessem sido copiados para a classe derivada como protegidos (protected).

- Classe base herdada como **private**:
  - Membros públicos (public) da classe base:
    - Os membros se comportam como se tivessem sido copiados como privados (private) na classe derivada.
  - Membros privados (private) da classe base:
    - Os membros estão presentes na classe, porém ocultos como privados. Desta forma as informações estão presentes, mas não poderão ser acessadas, a não ser por funções da classe base que se utilizem delas.
  - Membros protegidos (protected) da classe base:
    - Os membros se comportam como se tivessem sido copiados como privados (private) na classe derivada.

- Classe base herdada como **Protected**:
  - Membros públicos (public) da clase base:
    - Se comportam como se tivéssemos copiado-os como protegidos (protected) na classe derivada
  - Membros privados (private) da classe base:
    - Os membros estão presentes na classe, porém ocultos como privados. Desta forma as informações estão presentes, mas não poderão ser acessadas, a não ser por funções da classe base que se utilizem delas.
  - Membros protegidos (protected) da classe base:

- Se comportam como se estivéssemos copiado-os como protegidos (protected) na classe derivada.

Em suma, estas regras podem ser sintetizadas em uma regra muito simples: *Prevalece o atributo mais restritivo*. Para isto basta-nos listar os atributos por ordem de restrições decrescente:

1. ***private***
2. ***protected***
3. ***public***

Assim, temos todas as combinações definidas quando colocamos um atributo combinado com o outro, bastando para isto escolher sempre o mais restritivo na combinação. Por exemplo: Quando temos uma variável pública na base e a herança é privada, a combinação resulta em uma variável privada na classe derivada.

Aqui está um exemplo muito simples:

```
#include <iostream>

using namespace std;

class base
{
     int i, j;
public:
      void set(int a, int b) { i = a; j = b; }
      void show() { cout << i << " " << j << "\n"; }
};

class derived : public base
{
     int k;
public:
      derived(int x) { k = x; }
      void showk() { cout << k << "\n"; }
};

int main()
{
   derived ob(3);
   ob.set(1, 2); // acesso a membro da base
   ob.show(); // acesso a membro da base
   ob.showk(); // uso de membro da classe derivada
#ifdef WIN32
   system ("pause");
#endif
   return 0;
}
```

Conseguimos acessar as funções set() e show() porque são heradadas como publicas.

Agora modifiquemos o atributo de acesso na declaração da herança da classe base:

```
#include <iostream>

using namespace std;
```

```
class base
{
     int i, j;
public:
      void set(int a, int b) { i = a; j = b; }
      void show() { cout << i << " " << j << "\n"; }
};

class derived : private base
{
     int k;
public:
      derived(int x) { k = x; }
      void showk() { cout << k << "\n"; }
};

int main()
{
   derived ob(3);
   ob.set(1, 2); // Erro, não é possível acessar set()
   ob.show(); // Erro, não é possível acessar show()
   ob.showk(); // uso de membro da classe derivada
#ifdef WIN32
   system ("pause");
#endif
   return 0;
}
```

Agora já não podemos acessar as funções porque estão privadas.

Heranças múltiplas [editar | editar código-fonte]

Podemos ter a situação em que uma classe derivada possa herdar membros de várias classes base. Esta característica é uma distinção entre C++ e outras linguagens orientadas a objeto. Este recurso dá mais poder de modelagem ao programador, mas vale a pena lembrar que *mais poder* exige *mais cautela* no uso.

```
// Um exemplo de múltiplas classes base.
#include <iostream>

using namespace std;

class base1
   {
   protected:
             int x;
   public:
          void showx() { cout << x << "\n"; }
   };

class base2
   {
   protected:
             int y;
   public:
          void showy() { cout << y << "\n"; }
```

```
    };

class derived: public base1, public base2 // Inherit multiple base classes.
    {
    public:
           void set(int i, int j) { x = i; y = j; }
    };

int main()
{
    derived ob;
    ob.set(10, 20); // Disponível pela classe "derived"
    ob.showx(); // Pela classe base1
    ob.showy(); // Pela classe base2
#ifndef WIN32
    system ("pause");
#endif
    return 0;
}
```

Repare que utilizamos o operador vírgula para dizer ao compilador que a classe derivada herda mais de uma classe. Com efeito, temos uma lista de classes separadas por vírgulas depois do operador ":" (dois pontos).

Quando queremos que a classe derivada herde uma classe como pública e outra como privada ou protegida basta preceder a classe com o seu especificador de acesso. Da mesma forma, a omissão do especificador leva o compilador a usar o padrão que é privado (private).

Construtores e destrutores [editar | editar código-fonte]

Temos uma série de classes que mantém relações de parentesco conforme mostramos nas seções anteriores. Em termos genéricos, classes que herdam características de uma base precisam de regras claras quando forem criadas e destruídas. Precisamos definir a sequência em que os construtores e destrutores serão chamados, uma vez que cada classe tem pelo menos um construtor e um destrutor.

Agora temos a questão: Quando é que os construtores são chamados quando eles são herdados?

- Quando um objeto da classe derivada é instanciado, o construtor da classe base é chamado primeiro seguido do construtor das classes derivadas, em sequência da base até a última classe derivada.
- Quando o objeto da classe derivada é destruído, o seu destrutor é chamado primeiro seguido dos destrutores das outras classes derivadas logo abaixo, em sequência até a base.

Vamos testar e ver como isto funciona.

No caso em que termos herança sequencial A-B-C, teremos:

```
#include <iostream>

using namespace std;

class base
    {
    public:
           base() { cout << "Construindo base" << endl; }
           ~base() { cout << "Destruindo base" << endl; }
```

```
    };

class derivada1 : public base
    {
    public:
           derivada1() { cout << "Construindo derivada1" << endl; }
           ~derivada1() { cout << "Destruindo derivada1" << endl; }
    };

class derivada2: public derivada1
    {
    public:
           derivada2() { cout << "Construindo derivada2\n"; }
           ~derivada2() { cout << "Destruindo derivada2\n"; }
    };

int main()
{
    derivada2 ob; // constrói e destrói o objeto ob
#ifdef WIN32
    system ("pause");
#endif
    return 0;
}
```

Caso de múltipla herança A - B e C

```
#include <iostream>

using namespace std;

class base1
        {
        public:
              base1() { cout << "Construindo base1\n"; }
              ~base1() { cout << "Destruindo base1\n"; }
       };

class base2
       {
       public:
              base2() { cout << "Construindo base2\n"; }
              ~base2() { cout << "Destruindo base2\n"; }
       };

class derivada: public base2,public base1
       {
       public:
              derivada() { cout << "Construindo derivada\n"; }
              ~derivada() { cout << "Destruindo derivada\n"; }
       };

int main()
{
    derivada ob;// construindo e destruindo o objeto.
```

```
#ifdef WIN32
    system ("pause");
#endif
    return 0;
 }
```

Neste caso a sequência de inicialização segue ordem estabelecida na lista de herança. Mais explicitamente, temos a construção das bases: "base2" e "base1", nesta ordem respectivamente e depois a derivada. O que, automaticamente, nos revela a sequência de destruição na ordem inversa, ou seja: destroi-se a "derivada", depois "base1" e, finalmente, a estrutura da "base2".

Passando parâmetros para construtores da classe base [editar | editar código-fonte]

Agora imaginemos que temos um conjunto de bases para uma classe que queiramos derivar, então podemos ter um construtor em cada base que precise de parâmetros para que possa ser invocado pela nossa classe. Como poderemos passar os parâmetros, uma vez que os mesmos só podem ser passados durante a inicialização da classe?

Para que possamos passar os parâmetros para as classes bases durante a inicialização do objeto da classe derivada temos o recurso de passagem de parâmetros pelo construtor. Basicamente, ele funciona como se passássemos valores para variáveis membro. Chamamos cada construtor na lista de passagem de valores, a sintax para declarar o corpo do construtor é a seguinte:

```
class Classe_derivada : public Base1, public Base2, ..., public BaseN
{ // Membros...
  public:
     Classe_derivada(lista_de_argumentos);
  // Outras funções...
};


 Classe_derivada::Classe_derivada(lista_de_argumentos) : Base1(lista_de_argumentos),
Base2(lista_de_argumentos), ...BaseN(lista_de_argumentos);
 {
  //Corpo do construtor da classe derivada
 }
```

Este exemplo é um pouco mais complexo, atenção!

```
 #include <iostream>

 using namespace std;

 class base
        {
        protected:
                  int i;
        public:
               base(int x) { i = x; cout << "Construindo base\n"; }
               ~base() { cout << "Destruindo base\n"; }
        };

 class derivada: public base
```

```
        {
              int j;
        public:
               derivada(int x, int y): base(y) { j = x; cout << "Construindo
derivada\n"; }// derivada usa x; y é passada em lista para a base.
               ~derivada() { cout << "Destruindo derivada\n"; }
               void mostrar() { cout << i << " " << j << "\n"; }
        };

 int main()
 {
    derivada ob(3, 4);
    ob.mostrar(); // mostra 4 3
#ifdef WIN32
    system ("pause");
#endif
    return 0;
 }
```

No exemplo, a o construtor da classe derivada é declarado com 2 argumentos (x e y). no entanto a função derivada() usa apenas um para inicializar a variável interna da classe, o segundo argumento é usado para passar o valor de inicialização para a classe base.

Vejamos mais um exemplo:

```
 #include <iostream>

 using namespace std;

 class base1
    {
    protected:
              int i;
    public:
           base1(int x) { i = x; cout << "Construindo base1\n"; }
           ~base1() { cout << "Destruindo base1\n"; }
    };

 class base2
    {
    protected:
              int k;
    public:
           base2(int x) { k = x; cout << "Construindo base2\n"; }
           ~base2() { cout << "Destruindo base2\n"; }
    };

 class derivada: public base1, public base2
    {
         int j;
    public:
           derivada(int x, int y, int z): base1(y), base2(z)
                      { j = x; cout << "Construindo derivada\n"; }
           ~derivada() { cout << "Destruindo derivada\n"; }
           void mostrar() { cout << i << " " << j << " " << k << "\n"; }
    };
```

```
 int main()
 {
    derivada ob(3, 4, 5);
    ob.mostrar(); // mostra 4 3 5
#ifdef WIN32
    system ("pause");
#endif
    return 0;
 }
```

Superposição de funções [editar | editar código-fonte]

Muitas vezes temos classes derivadas que executam uma determinada ação de forma distinta da mesma ação definida na classe base. Por exemplo, se temos uma classe "animal" e declaramos uma função chamada "mover" e depois declaramos duas derivadas: "ave" e "peixe" com a mesma função "mover" teremos uma incoerência devido ao fato de que peixes se movem de forma totalmente diferente de aves. Uma vez que peixes devem "nadar" e "aves" podem "voar" ou "andar" nosso modelo de objetos está incorreto.

Por questões de coerência semântica, porém, precisamos manter o mesmo nome para as funções das classes base e derivada em algumas construções. Isto é essencial devido a necessidade de criarmos objetos generalistas, por exemplo se tivermos classes "ave" e "peixe" abstraídas em uma base "animal", como vimos acima. Havendo estas condições, como poderemos criar comportamentos diferentes usando o mesmo nome para as funções?

A resposta está em uma das características que será muito útil quando quisermos usar de polimorfismo, que iremos abordar em capítulo específico mais adiante: a ocultação e superposição de funções da classe base a partir de uma classe derivada (conhecida como "overriding" em manuais de compiladores). Com este recurso podemos declarar em uma classe derivada uma função com nome e parâmetros idênticos a uma existente em uma classe base, porém com conteúdo diferente.

Vejamos o exemplo de código e teremos uma noção mais concreta do que foi explanado:

```
#include <iostream>

using namespace std;

 class animal
 {
   public:
      void comer();
      void mover();
      void dormir() { cout << "Dormindo..." << endl; }
 };
...
...
 class ave : public animal
 {
   public:
      void comer(){ cout << "Bicando..." << endl; }
      void mover(){ cout << "Voando..." << endl; }
 };
 ...
```

```
 ...
 class peixe : public animal
 {
   public:
      void comer(){ cout << "Mordendo..." << endl; }
      void mover(){ cout << "Nadando..." << endl; }
 };

int main()
{
  ave passarinho;
  peixe sardinha;

  passarinho.mover();
  sardinha.mover();

#ifdef WIN32
 system("pause");
#endif
 return 0;
}
```

Ao executar o programa gerado por este código percebemos que a mesma função: mover(), terá comportamento diferente quando invocada por objetos de classes diferentes. O programa mostrará a mensagem "Nadando..." para a função invocada pelo objeto sardinha e "Voando..." para a invocada pelo objeto passarinho. Aqui, o mecanismo é bem simples de se entender, quando cada objeto tem uma versão diferente para a mesma função é fácil para o compilador relacionar o objeto à classe que ele pertence e invocar a função apropriada.

Acessando funções superpostas da classe base [editar | editar código-fonte]

O mecanismo para obter acesso às classes base a partir da classe derivada é intuitivo. Para isto usamos o operador de resolução de escopo, composto por um par de dois pontos "::", usando a seguinte sintaxe:

```
<CLASSE>::<FUNÇÃO>(lista_de_parâmetros);
```

Ou seja, basta invocar a função informando qual é a versão específica que se deseja utilizar. Se tivermos uma classe "A" e outra "B" com uma função "Print()", por exemplo, e quisermos usar a função "Print()" da classe "B" fazemos:

```
B::Print();
```

Talvez seja melhor visualizar um exemplo no código mais completo:

```
class B
{
  public:
        void Print()
             {
              cout << "Chamando Print() da classe B." << endl;
```

```
                }
};

class A : public B
{
  public:
         void Print()
              {
               cout << "Chamando Print() da classe A." << endl;
               B::Print();
              }
};

int main()
{ A ca;

  ca.Print();

  return 0;
}
```