Orientação a objetos e C++

Professora: Sílvia Campos

IPRJ/UERJ

Programação Orientada a Objetos (POO)

Conceitos:

1- Encapsulamento

2- Classe como tipos de dados

3- Objetos

4- Herança

5- Polimorfismo

Programação orientada a objetos

Facilita o desenvolvimento e entendimento de programas de grande porte.

Possibilita a reutilização do código, reduz custos de desenvolvimento.

Conceitos: classes, objetos, herança, encapsulamento e polimorfismo.

Classes e Objetos

- Possibilidade de combinar num único registro campos que conterão dados e campos que são funções para operar os campos de dados do registro. Uma unidade assim definida é chamada classe.
- Classe: tipo de dado como os tipos que existem predefinidos em compiladores de diversas linguagens de programação. Assim como é possível declarar várias variáveis do tipo int, pode-se também declarar várias variáveis de uma classe já definida.
- **Uma variável de uma classe é chamada objeto e conterá campos de dados e funções.**
- As funções de um objeto são chamadas **funções-membro** ou **métodos** e, de modo geral, são o único meio de acesso aos campos de dados também chamados **variáveis de instância**.

Classes e objetos

Objetos

- Enquanto uma instância de um estrutura ou de uma união é chamada variável, uma instância de uma **classe** é chamada **objeto**.
- Uma **variável de um tipo classe** é dita um **objeto**.

Definindo a Classe

- Antes de criar um objeto de uma certa classe, é necessário que ela esteja definida.
- Esse conceito é o mesmo que usamos com estruturas e uniões: antes de criar uma variável de um tipo estrutura, é necessário definir a estrutura.

- Um programa em C++ consiste em um conjunto de objetos que se comunicam por meio de chamadas às funções-membro.
- A frase "**chamar uma função-membro de um objeto**" pode ser dita como "**enviar uma mensagem a um objeto**".

Encapsulamento

- Se o programa necessita atribuir um valor a alguma **variável de instância**, deve chamar uma **função membro** que recebe o valor como argumento e faz a alteração. **Não podemos acessar variáveis de instância diretamente.**

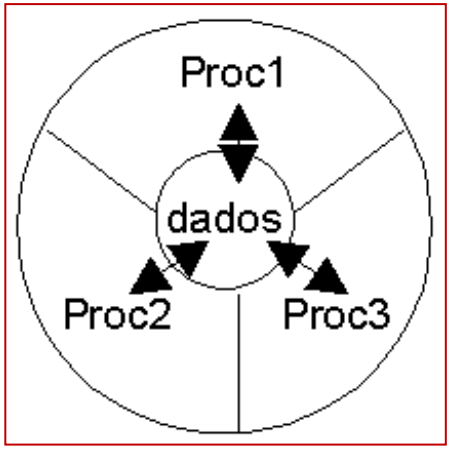
- Dessa forma, os **campos de dados** estarão **escondidos**, o que previne alterações acidentais. Dizemos então que os campos de dados e suas funções estão **encapsuladas** numa única entidade.

- Se alguma modificação ocorrer em variáveis de instância de um objeto, sabemos exatamente quais funções interagiram com elas: são as funções-membro do objeto. Nenhuma outra função pode acessar esses dados. Isso simplifica a escrita, manutenção e alteração de programas.

Encapsulamento

- **Objetos: tipos abstratos de dados** nos quais encontramos tanto variáveis, usadas para armazenamento das informações, quanto o código que pode atuar sobre as mesmas.

- As variáveis encontram-se encapsuladas no objeto, e a única forma de acessá-las é através das chamadas às funções do próprio objeto.



Objeto composto por dados e procedimentos necessários à descrição de um elemento no sistema.

Herança

- Conceito de subclasse ou processo de classes derivadas.
- Relacionamento entre classes por meio de hierarquias.
- Quando dividimos classes em subclasses, cada subclasse herda as características da classe da qual foi derivada. Além das características herdadas, cada subclasse tem suas características particulares.
- Em C++, é definida uma **classe-base** quando são identificadas características comuns em um grupo de **classes derivadas**.
- Uma classe que é derivada de uma classe-base pode, por sua vez, ser a classe-base de outra classe.

Herança

Simples ou **múltipla**.

- **Herança simples**: uma classe pode herdar características de apenas uma outra classe.

- **Herança múltipla:** extensão da herança simples. Uma classe pode herdar características de duas ou mais classes, logicamente relacionadas, ou não.

O uso de uma biblioteca de classes oferece uma grande vantagem sobre o uso de uma biblioteca de funções:

→o programador pode criar classes derivadas de classe-base de biblioteca - sem alterar a classe-base, é possível adicionar a ela características diferentes que a tornarão capaz de executar exatamente o que se deseja.

Herança

Vantagens:

- Possibilidade de se criar classes especializadas sem ter que alterar a implementação das classes básicas, aumentando, desta forma, a possibilidade de **reutilização do código** e, consequentemente, obtendo uma **otimização** do processo de desenvolvimento dos programas.

- Facilidade de se obter uma melhor **organização** do programa, simplificando, consequentemente, a evolução do sistema, já que facilita o processo de especialização e modificação do código, além de **reduzir a duplicação de código e de dados no sistema.**

Polimorfismo

- Possibilidade de um mesmo nome poder ser usado para identificar diferentes procedimentos.
- Nomes usados para representar operações genéricas, e não procedimentos específicos.

Em C++, chamamos de polimorfismo a criação de uma família de funções que compartilham do mesmo nome, mas cada uma com código independente.

Polimorfismo

- O resultado da ação de cada uma das funções da família é o mesmo. A maneira de se atingir esse resultado é distinta.
- **Exemplo**: calcular o salário líquido de todos os funcionários de uma empresa. Nessa empresa há horistas, mensalistas, os que ganham por comissão etc.
- Criamos um conjunto de funções em que cada uma tem um método diferente de calcular o salário.
- Quando a função é chamada, C++ saberá identificar qual é a função com o código adequado ao contexto.

Sobrecarga

- Tipo particular de polimorfismo.

- A utilização de um mesmo símbolo para a execução de operações distintas.

- **Exemplo**: o operador aritmético +.

O computador executa operações completamente diferentes para somar números inteiros e números reais.

- Em C++, além das sobrecargas embutidas na linguagem, podemos definir outras com capacidades novas.

O polimorfismo e a sobrecarga são habilidades únicas em linguagens orientadas ao objeto.

Classes e objetos

Uma definição de classe em C++ sempre começa pela palavra-chave **class seguida do nome da classe**, e seu corpo delimitado por chaves e finalizado com um ponto-e-vírgula.

```
//Ex.: retangulo.cpp
#include <iostream.h>
class Retangulo {              //Define a classe
    private:
        int bas,alt;           //Dados membro
    public:
        //Funções-membro para inicializar os dados
        void init (int b, int h)       {bas = h;   alt = h;}
        void printdata() {
                cout << "\nBase = " <<bas<<  "Altura = "<<alt;
                cout << "\nArea= "  <<(bas*alt);
        }
};
```

```
void main ()
{
    Retangulo x,y;          //Declara dois objetos: x e y
    x.init(5,3);            //Chama função-membro que inicializa
    y.init(10,6);                   //os dados dos dois objetos
    x.printdata();          //Chama função-membro que imprime a
    y.printdata();                  // área do retângulo x e do retângulo y
}
```

- A classe Retangulo é composta por dois itens de dados e duas funções.
- Estas funções são o único meio de acesso aos itens de dados.
- A primeira atribui valores aos itens de dados;
- A segunda calcula e imprime a área do retângulo.

Membros privados e públicos

- O corpo da definição da classe contém as palavras **private** e **public** seguidas de dois-pontos. Elas especificam a **visibilidade dos membros**.
- Os membros definidos após **private:** são chamados de parte privada da classe e podem ser acessados pela classe inteira, mas não fora dela.
- **Um membro privado não pode ser acessado por meio de um objeto** – conceito equivalente à relação existente entre uma função e as suas variáveis locais; elas não estarão visíveis fora da função.
- Geralmente, na parte privada da classe são colocados os membros que conterão dados. Diz-se que esses dados estarão "escondidos", ou seja, não poderão ser alterados por funções que não pertençam à classe.
- Se uma função-membro é declarada **private**, somente outras funções-membro da mesma classe poderão acessá-la.

Membros privados e públicos

- A seção pública da classe é formada pelos membros definidos após **public:** e pode ser acessada por qualquer função-membro e por qualquer outra função do programa onde um objeto foi declarado.

- Os membros públicos fazem a interface entre o programador e a classe.

- Geralmente, na parte pública de uma classe, são colocadas as funções que irão operar os dados.

Membro protegido

- Além de privado ou público, um membro pode ser declarado como protegido, através da palavra-chave ***protected***.

- O motivo para a declaração de um membro como protegido está relacionado ao mecanismo de herança de classes.

- **Uma classe pode herdar características de outras classes**. Quando isto ocorre, a nova classe não pode acessar os membros declarados como privados à classe herdada, a não ser que, em vez de serem declarados como privados, sejam declarados como protegidos.

Membros declarados como protegidos continuam só podendo ser acessados por funções-membros, mas o direito de acesso passa a poder ser herdado pelas funções membros das classes derivadas

Definição de classe

```
class nome_da_classe
{
    public:
        variáveis e funções públicas
    private:
        variáveis e funções privadas
    protected:
        variáveis e funções protegidas
};
```

As funções-membro de uma classe

- As funções-membro, ou métodos, são as que estão dentro da classe. Na classe Retangulo, definimos dois **métodos**: init() e printdata(). Estas funções executam operações comuns em classes: atribuem valores aos dados-membro e imprimem certos resultados.

Definição:

→ 1ª. Forma: definir a função dentro da própria classe:

```
class nome_da_classe
{
    tipo-retorno nome_da_função (lista de parâmetros)
    {
        corpo da função
    }
};
```

As funções membro de uma classe

→ 2ª. Forma: definir a função fora do escopo da definição da classe

Uso do **operador de identificação de escopo**, representado pelo símbolo (**::**):

```
class nome_da_classe
{
    tipo_retorno   nome_da_função (lista de parâmetros);
};

tipo_retorno    nome_da_classe :: nome_da_função (lista de parâmetros)
{
    corpo da função
}
```

Uma vez definida, a função pode ser chamada a partir de um método da mesma classe, ou no caso do método ser público, a partir de qualquer outra função dentro do escopo de um objeto deste tipo de classe.

```
#include <iostream.h>
class Retangulo {            //Define a classe
    private:
        int bas,alt;         //Dados membro
    public:
        //Funções-membro para inicializar os dados
        void init (int b, int h);
        void printdata();
};

void Retangulo:: init (int b, int h)  { bas = h;   alt = h;}
void Retangulo:: printdata()
 {
        cout << "\nBase = " <<bas<<  "Altura = "<<alt;
        cout << "\nArea= "  <<(bas*alt);
}
```

A chamada a uma função membro

- Os membros públicos de um objeto são acessados por meio do operador ponto. Sintaxe similar à de acesso aos membros de uma estrutura.

- **Uma função-membro age sobre um objeto particular e não sobre a classe como um todo.**

A instrução: x.init(5,3);

- executa a função-membro **init()** do objeto **x**.
- Esta função atribui o valor 5 à variável **bas** e 3 à variável **alt** do objeto **x**.

A instrução: y.init(10,6);

- executa a mesma tarefa sobre o objeto y.

A chamada a uma função membro

As instruções:

```
x.printdata();
y.printdata();
```

imprimem os dados de cada retângulo e as suas áreas.

Mensagem:

- instrução de chamada a uma função-membro

A instrução: `x.printdata();`

- envia uma mensagem ao objeto x solicitando que ele imprima os dados e a área do retângulo.

A instrução: `y.printdata();`

- envia uma mensagem ao objeto y solicitando que ele imprima os dados e a área do retângulo.

Dados-membro static

- Muitas vezes é necessário que todos os objetos de uma mesma classe acessem um mesmo item de dado. Este item faria o mesmo papel de uma variável externa, visível a todos os objetos daquela classe.

- Quando um dado membro é declarado como **static**, é criado um único item para a classe como um todo, não importando o número de objetos declarados. A informação de um membro **static** é compartilhada por todos os objetos da mesma classe.

- Um membro **static** tem características similares às variáveis estáticas normais. É visível somente dentro da classe e é destruído somente quando o programa terminar.

Funções construtoras

- Usadas para **inicializar um objeto quando da sua criação**.
- Tem o **mesmo nome** da classe da qual é membro, sendo automaticamente chamada quando for criado um objeto desta classe.
- O seu uso é opcional, sendo útil quando um objeto, ao ser criado, precise executar algum processo de inicialização.
- Um construtor **não deve retornar nenhum valor**. O motivo é ele ser chamado diretamente pelo sistema e portanto não há como recuperar um valor de retorno.
- Um construtor pode perfeitamente chamar uma função-membro. O seu código é igual ao de qualquer outra função, exceto pela **falta de tipo**.

Funções construtoras

Exemplo:

- Considerar a definição de uma classe que modela uma estrutura de dados tipo pilha.

- A classe contém, na sua parte privada, uma matriz para armazenamento das informações e uma variável para contagem do número de elementos inseridos.

- Na parte pública encontramos as funções-membro e a função construtora, responsável por inicializar a variável de contagem quando for criado um objeto desta classe.

Exemplo:

```
#define SUCESSO 1
#define FALHA 0

//definição da classe pilha
class pilha
{
    int dados[100];
    int conta;
public:
     //função construtora:
          pilha (void);
    //funções-membro:
    int insere (int valor);
    int remove (int *valor);
};
```

```
pilha::pilha()  {
    conta = 0;
}
int pilha::insere(int valor)  {
    if (conta < 100) {
          dados[conta++] = valor;
          return SUCESSO;
    } else
          return FALHA;
}
int pilha::remove(int *valor) {
    if (conta == 0)
          return FALHA;
    else {
          *valor = dados [--conta];
          return SUCESSO;
    }
}
```

Funções destrutoras

- Além de ser criado, **um objeto pode também ser destruído (liberado da memória)**, por exemplo, quando a execução do programa desloca-se para um bloco de código fora do escopo do objeto.

- Uma função destruidora é chamada **quando uma variável da classe deixa de existir.**

- A identificação de uma função destrutora é feita pelo **nome da classe precedido pelo símbolo (~).**

- Da mesma forma que um construtor, os destrutores não podem ter valor de retorno.

- O destrutor também não pode receber argumentos nem pode ser chamado explicitamente pelo programador.

Definição de um construtor:

```
class nome_da_classe
{
public:
        nome_da_classe (void);    //função construtora
};
```

Definição de um destrutor:

```
class nome_da_classe
{
public:
        ~nome_da_classe (void); //função destrutora
};
```

Funções construtoras e destrutoras

Destrutor da classe REC: decrementa o contador de objetos sempre que um objeto é liberado da memória.

```
class REC
{
    private:
        static int n;
    public:
        //construtor
        REC () { n++; }
        //destrutor
        ~REC () { n-- ; }
        int getrec() const {return n; }
};

int REC :: n = 0;
```

```
void main()
{
    REC r1, r2, r3;
    cout  << "\nNumero de objetos: "
          <<r1.getrec();
    {
          REC r4, r5;
          cout <<"\nNumero de objetos:"
               <<r1.getrec();
    }
    cout << "\nNumero de objetos: "
          <<r1.getrec();
}
```

Membros static Públicos

Um membro **static** é criado na definição da classe e existe independentemente de qualquer objeto. E não pode ser inicializado por um construtor da classe, visto que o construtor é chamado toda vez que um objeto é criado.

Acessado quando precedido do **nome da classe** seguido do **operador (::)**.

```
class facil {
    private:
        int x, y, z;
    public:
        static int p;
        facil() { }
        ~facil() { }
};
```

para se atribuir 5 ao membro p: **facil :: p=5;**

Objetos e Alocação de Memória

- Para cada objeto declarado, é reservado um espaço de memória em separado para armazenamento de seus dados-membro.

- Como todos os objetos da classe utilizam as mesmas funções, é armazenado espaço de memória para armazenar somente uma cópia de cada função-membro de uma classe particular, independente de quantos objetos tiverem sido criados!!!

- **Funções-membro são criadas e colocadas na memória somente uma vez para a classe toda.**

- As funções-membro são compartilhadas por todos os objetos da classe. Os itens de dado, por sua vez, armazenam diferentes valores para cada objeto, então é necessário que sejam criados novos membros de dados sempre que um objeto for declarado.

Objetos como argumentos de funções-membro

```
#include <iostream.h>
#include <iomanip.h>
class venda
{
private:
    int npecas;
    float preco;
public:
    venda() { } //construtor sem args
    venda(int np, float p) //construtor
                           //com args
    {
        npecas = np;
        preco = p;
    }
    void getvenda()
    {
        cout << "Insira No. Pecas: ";
        cin >> npecas;
        cout << "Insira Preco: ";
        cin >> preco;
    }
    void printvenda() const;

    void add_venda(venda v1, venda v2)
    {
        npecas = v1.npecas+v2.npecas;
        preco = v1.preco + v2.preco;
    }
};
```

```
void venda::printvenda() const
{
    cout << setiosflags(ios::fixed)
         //nao notacao científica
         <<
    setiosflags(ios::showpoint)
         //ponto decimal
         << setprecision(2)
         //duas casas
         << setw(10) << npecas;
         //tamanho 10
    cout << setw(10) << preco <<
    "\n";
}
```

```
void main()
{
    venda A(58,12734.53), B,
    Total;
    B.getvenda();
    Total.add_venda(A,B);


    cout << "Venda A.......";
    A.printvenda();


    cout << "Venda B.......";
    B.printvenda();


    cout << "Totais....... ";
    Total.printvenda();
}
```

Funções que retornam um objeto

```
#include <iostream.h>
#include<iomanip.h>
class venda
{
private:
      int npecas;
      float preco;
public:
      venda() { } //construtor sem args
      venda(int np, float p); //construtor
                              //com args
      void getvenda();
      void printvenda() const;

      venda add_venda(venda v) const;
};
```

```
venda venda::add_venda(venda v)
                  const
{
      venda temp; //variavel temporaria
      temp.npecas = npecas + v.npecas;
      temp.preco = preco + v.preco;
      return temp;
}

void main()
{
      venda A(58,12734.53), B, Total;
      B.getvenda();
      Total = A.add_venda(B);

      cout << "Venda A....."; A.printvenda();
      cout << "Venda B....."; B.printvenda();
      cout << "Totais..... "; Total.printvenda();
}
```

Construtores de corpo vazio

- Muitas vezes, pode-se querer ter um construtor para inicializar os dados e também definir algum objeto sem que este seja inicializado. Para tanto, deve-se criar um construtor de corpo vazio.

- O corpo vazio indica que a função não faz nada e pertence à classe somente para que o outro construtor não seja executado automaticamente.

```
//Exemplo: Mostra acesso público, privado e protegido à classe-base
#include <iostream.h>

class BASE {
    protected:    int secreto;
    private:      int super-secreto;
    public:       int publico;
};

class DERIV1 : public BASE {
    public:
        int a = secreto;                    //OK
        int b = super-secreto;              //ERRO: não acessível
        int c = publico;                    //OK
};
class DERIV2 : private BASE {
    public:
        int a = secreto;                    //OK
        int b = super-secreto;              //ERRO: não acessível
        int c = publico;                    //OK
};
```

```
void main()
{
    int x;

    DERIV1 obj1;            //DERIV1 é public
    x = obj1.a;                     //ERRO: não acessível
    x = obj1.b;                     //ERRO: não acessível
    x = obj1.c;                     //OK

    DERIV2 obj2;            //DERIV2 é private
    x = obj2.a;                     //ERRO: não acessível
    x = obj2.b;                     //ERRO: não acessível
    x = obj2.c;                     //ERRO: não acessível
}
```

Reescrevendo Funções membro da classe-base

- Quando se usa uma classe predefinida e pré-compilada como classe-base, nem sempre as funções-membros dessa classe atendem completamente ao que se espera. Muitas vezes, é preciso modificar uma das funções para acrescentar alguma nova característica.

- **É possível criar funções-membro de uma classe derivada que tenham o mesmo nome de funções membro da classe-base**. Assim, a sintaxe da chamada a elas, por meio de um objeto, é a mesma, independentemente de tratar-se de um objeto da classe-base ou da classe derivada. A nova função pode chamar a função da classe-base ou da classe derivada. **A nova função pode chamar a função da classe-base por meio do operador de resolução de escopo (::)**.

- Se duas funções de mesmo nome existem, uma na classe-base e outra na classe derivada, a função da classe derivada será executada se for chamada por meio de um objeto da classe derivada.

- Se um objeto da classe-base é criado, usará sempre funções da própria classe-base pois não conhece nada da classe derivada.

Reescrevendo Funções membro da classe-base

```
#include <iostream.h>
//definição da classe base
class classe_1 {
    public:
        int x;
        classe_1 (void) { x = 10; }
};

//definição da classe derivada
class classe_2 : public classe_1 {
    public:
      int x;
      void atribui (int valor) { x = valor; }
      int soma (void);
};
```

```
int classe_2::soma(void)
{
    return (classe_1::x + x);
}

main()
{
    classe_2 obj;
    obj.atribui (15);
    cout << "soma dos valores = "
         <<obj.soma();
}
```

Reescrevendo Funções membro da classe-base

- Neste exemplo, acessamos a variável x, membro da classe base classe_1, a partir da função soma(), membro da classe derivada classe_2.

- Caso a classe base contenha funções construtoras, ou destrutoras, estas funções serão chamadas quando da criação, ou destruição, de um objeto com o tipo da classe derivada. Além disto, caso ambas as classes contenham estes tipos de funções, quando da criação, ou da destruição, de um objeto do tipo da classe derivada, será executada, tanto a função membro da classe derivada, quanto a função membro da classe base.

- Ao ser criado um objeto com o tipo da classe derivada, a função construtora da classe derivada será executada após a execução da função construtora da classe base. Ao ser destruído um objeto com o tipo da classe derivada, a função destrutora da classe derivada será executada antes da execução da função destrutora da classe base.

Hierarquia de Classes

Exemplo:

- Classe-base conta: usada para criar as classes derivadas: conta simples, contaEspecial e Poupança.

  A base de dados armazena o nome do cliente, a identificação numérica da conta e o saldo, independentemente de sua categoria. Entretanto, contas especiais necessitam de um dado adicional: o limite; e contas-poupança necessitam da taxa de reajuste.

Hierarquia de Classes

```
#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
const MAX=80;

class conta
{
      private:
            char nome[MAX];
            int Nconta;
            float saldo;
      public:
            void getdata()
            {
                  cout << "\n  Nome:  "   << endl;  gets(nome);
                  cout << "  No.conta:  "<< endl;  cin >> Nconta;
                  cout << "  Saldo:  ";  << endl;  cin >> saldo;
            }
            void putdata()
            {
                  cout << "\n  Nome:  " << nome;
                  cout << "\n  No.Conta:  " << Nconta;
                  cout << "\n  Saldo:  " << setiosflags (ios::fixed)
                                        << setprecision(2) << saldo;
            }
            float Saldo()  { return saldo; }
};
```

```
class contaSimples : public conta
{ };

class contaEspecial : public conta
{
      private:
            float limite;
      public:
            void getdata()
            {
                  conta::getdata();
                  cout << "  Limite:  ";  cin >> limite;
            }
            void putdata()
            {
                  conta::putdata();
                  cout << "\n Limite: " << limite;
                  cout << "\n Saldo Total: "
                        << setiosflags(ios::fixed)
                        << setprecision(2) << (Saldo() + limite);
            }
};
```

```
class Poupanca  : public conta
{
      private:
            float taxa;
      public:
            void getdata()
            {
                  conta::getdata();
                  cout << "  Taxa:  "; cin >> taxa;
            }
            void putdata()
            {
                  conta::putdata();
                  cout << "\n  Taxa:  " << taxa;
                  cout << "\n  Saldo Total:  "
                              << setiosflags(ios::fixed)
                              << setprecision(2) << (Saldo() * taxa);
            }
};
```

```
void main()
{
      contaSimples c1, c2;
      contaEspecial c3;
      Poupanca c4;

      cout << "\nDigite os dados da conta simples 1. " << endl;
      c1.getdata();

      cout << "\nDigite os dados da conta simples 2. " << endl;
      c2.getdata();

      cout << "\nDigite os dados da conta especial.  " << endl;
      c3.getdata();

      cout << "\nDigite os dados da conta poupanca.  " << endl;
      c4.getdata();

      cout <<"\n\n Conta Simples 1. "; c1.putdata();
      cout <<"\n\n Conta Simples 2. "; c2.putdata();
      cout << "\n\n Conta Especial. "; c3.putdata();
      cout << "\n\n Conta Poupanca. "; c4.putdata();
}
```

Conversões de tipos entre classe-base e classe derivada

- Conversão implícita de um objeto da classe derivada num objeto da classe-base. Exemplo:

```
conta C;
contaEspecial CE;
CE.getdata();
C = CE;      //converte contaEspecial em conta
```

- Todos os membros do objeto C da classe-base recebem os valores dos membros correspondentes do objeto CE da classe derivada. Entretanto, a atribuição inversa é proibida:

```
CE = C;                    //erro! Não pode ser convertido
```

Como CE tem membros que C não tem, seus valores ficariam indefinidos!!

Herança Múltipla

- Uma classe pode ser derivada de outra classe, que, por sua vez, é também uma classe derivada.
- A construção de hierarquias de herança múltipla envolve mais complexidade do que as hierarquias de herança simples. Esta complexidade diz respeito ao desenho da construção das classes e não a sintaxe de uso. A sintaxe de múltiplas heranças é similar à de uma única herança.

```
class X
{ };
class Y
{ };
class Z : public X, public Y          //Z é derivada de X e de Y
{ };
```

- As classes-base das quais Z é derivada devem sem escritas após os dois-pontos e separadas por vírgula.

Referência bibliográficas

- Treinamento em Linguagem C++, Módulo 2, Victorine Viviane, Makron Books.

- Estrutura de dados e Algoritmos em C++.
  Autor: Adam Drosdek.
  Editora: Cengage Learning.

- Algoritmos – Teoria e Prática
  Autores: Cormen, Leiserson, Rivest, Stein
  Editora: Campus