Programação do Lado do Servidor

Geralmente os SGBDs permitem a criação de funções ou procedimentos definidos pelo usuário.

Estes procedimentos podem ser utilizados para implementar parte da lógica de um aplicativo ou criar regras para as atualizações no banco de dados.

Cada SGBD tem sua própria relação de linguagens que podem ser utilizadas nesses procedimentos.

O PostgreSQL permite a criação de funções nas seguintes linguagens:

- SQL
- C/C++
- Linguagens Procedurais
  - PL/pgSQL
  - PL/Tcl
  - PL/Perl
  - PL/Python

CREATE LANGUAGE

O comando **CREATE LANGUAGE** define uma linguagem procedural no banco de dados.

```
CREATE [ PROCEDURAL ] LANGUAGE name
```

Exemplo:

```
CREATE LANGUAGE plpgsql;
```

A partir da versão 9.0 a linguagem PL/pgSQL é instalada por padrão em todos os novos bancos de dados, não sendo mais necessário criar a linguagem nesses bancos.

Estrutura da Linguagem PL/pgSQL

A linguagem PL/pgSQL é estruturada em blocos. O texto completo da definição da função deve ser um bloco. Um bloco é definido como:

```
[ <rótulo> ]
[ DECLARE
    declarações ]
BEGIN
    instruções
END;
```

Todas as declarações e instruções dentro do bloco devem ser terminadas por ponto-e-vírgula. Um bloco contido dentro de outro bloco deve conter um ponto-e-vírgula após o **END**, entretanto, o **END** final que conclui o corpo da função não requer o ponto-e-vírgula.

CREATE FUNCTION

O comando **CREATE FUNCTION** cria uma função no banco de dados.

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ [ nome_do_argumento ]
tipo_do_argumento [, ...] ] ) RETURNS tipo_retornado
AS 'definição' | AS 'arquivo_objeto', 'símbolo_de_vinculação'
LANGUAGE nome_da_linguagem;
```

O nome da nova função não deve corresponder ao nome de uma função existente no mesmo esquema com argumentos dos mesmos tipos. Entretanto, funções com argumentos de tipos diferentes podem ter o mesmo nome, o que é chamado de sobrecarga (overload).

Para atualizar a definição de uma função existente deve ser usado o comando **CREATE OR REPLACE FUNCTION**. Não é possível mudar o nome ou os tipos dos argumentos da função desta maneira; se for tentado, na verdade será criada uma nova função distinta. O comando **CREATE OR REPLACE FUNCTION** também não permite mudar o tipo de dado retornado por uma função existente; para fazer isto a função deve ser removida e criada novamente.

CREATE FUNCTION

Se uma função não retorna nenhum valor, deve ser declarada como **VOID**.

Se a função for removida e recriada, a nova função não será mais a mesma entidade que era antes. Será necessário remover as regras, visões, gatilhos, etc. que fazem referência à função antiga. O comando **CREATE OR REPLACE FUNCTION** é utilizado para mudar a definição de uma função sem invalidar os objetos que fazem referência à função.

Exemplo:

```
CREATE FUNCTION add(INTEGER, INTEGER) RETURNS INTEGER
  AS 'select $1 + $2;'
LANGUAGE SQL;
```

Teste:

```
SELECT add(1,3);
 add
-----
   4
```

CREATE FUNCTION

Geralmente é útil utilizar outro delimitador como **$$** para envolver a cadeia de caracteres que define a função, em vez de usar aspas simples. Se a definição da função estiver envolvida por aspas simples, toda aspas simples ou contrabarra presente na definição da função deverá receber um escape duplicando os mesmos.

Exemplo:

```
CREATE OR REPLACE FUNCTION incrementar(INTEGER) RETURNS
INTEGER AS $$
  BEGIN
    RETURN $1 + 1;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT incrementar( 5 );
 incrementar
-------------
           6
```

DROP FUNCTION

O comando **DROP FUNCTION** remove uma função.

```
DROP FUNCTION [ IF EXISTS ] name ( [ [ argmode ]
[ argname ] argtype [, ...] ] ) [ CASCADE | RESTRICT ]
```

Exemplo:

```
DROP FUNCTION add(INTEGER,INTEGER);
```

Parâmetros

Os parâmetros passados para as funções recebem como nome os identificadores **$1**, **$2**, etc. Opcionalmente, para melhorar a legibilidade do código, podem ser declarados aliases para os nomes dos parâmetros.

Aliases podem ser criados fornecendo um nome para o parâmetro na declaração da função.

Exemplo:

```
CREATE FUNCTION taxa_de_venda(subtotal REAL) RETURNS REAL
AS $$
  BEGIN
    RETURN subtotal * 0.06;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT taxa_de_venda( 200 );
 taxa_de_venda
---------------
            12
```

Parâmetros

Outra forma para criar aliases é declarando explicitamente.

Exemplo:

```
CREATE FUNCTION taxa_de_venda(REAL) RETURNS REAL AS $$
  DECLARE
    subtotal ALIAS FOR $1;
  BEGIN
    RETURN subtotal * 0.06;
  END;
$$ LANGUAGE plpgsql;
```

Parâmetros

Pode ser determinado um valor default para os parâmetros da função.

Exemplo:

```
CREATE FUNCTION potencia(a REAL, b REAL DEFAULT 2)
RETURNS REAL AS $$
  BEGIN
    RETURN a ^ b;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT potencia( 5, 3 );
 potencia
----------
      125
SELECT potencia( 5 );
 potencia
----------
       25
```

Parâmetro de Saída

Pode ser determinado um parâmetro para receber o valor de saída da função.

Não é obrigatório especificar que um parâmetro recebe valores de entrada apenas.

Exemplo:

```
CREATE FUNCTION taxa_de_venda(IN subtotal REAL, OUT taxa
REAL ) AS $$
  BEGIN
    taxa = subtotal * 0.06;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT taxa_de_venda( 200 );
 taxa_de_venda
---------------
            12
```

Parâmetro de Saída

Um mesmo parâmetro pode ser usado como valor de entrada e de saída da função.

Exemplo:

```
CREATE FUNCTION taxa_de_venda(INOUT valor REAL ) AS $$
  BEGIN
    valor = valor * 0.06;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT taxa_de_venda( 200 );
 taxa_de_venda
---------------
            12
```

Tipos Polimórficos

O tipo dos argumentos e o tipo de retorno de uma função pode ser declarado como um tipo polimórfico que aceita diferentes tipos de dados:

anyelement – aceita qualquer tipo de dados

anyarray – aceita array de qualquer tipo de dado

anynonarray – aceita qualquer tipo de dados não array.

anyenum – aceita qualquer tipo enumerado.

anyrange – aceita qualquer tipo de dado de range.

Os tipos polimórficos podem ser usados várias vezes na declaração da função, porém devem representar o mesmo tipo de dados em uma mesma execução da função.

Tipos Polimórficos

Exemplo:

```
CREATE FUNCTION potencia(a anyelement, b anyelement)
RETURNS anyelement AS $$
  BEGIN
    RETURN a ^ b;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT potencia( 5, 2 );
 potencia
----------
       25

SELECT potencia( 5.5, 2.5 );
      potencia
--------------------
 70.942538367329372

SELECT potencia( 5, 2.5 );
ERROR:  function potencia(integer, numeric) does not
exist
```

Declarações

Todas as variáveis utilizadas em um bloco devem ser declaradas na seção de declarações do bloco. As exceções são a variável de laço do **FOR** interagindo sobre um intervalo de valores inteiros, ou interagindo sobre um cursor, que são automaticamente declaradas como sendo do tipo correspondente.

A sintaxe geral para declaração de variáveis é:

```
nome [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := }
expressão ];
```

As variáveis da linguagem PL/pgSQL podem possuir qualquer tipo de dado da linguagem SQL, como **INTEGER**, **VARCHAR** e **CHAR**.

Declarações

A cláusula **DEFAULT**, se for fornecida, especifica o valor inicial atribuído à variável quando o processamento entra no bloco. Se a cláusula **DEFAULT** não for fornecida, então a variável é inicializada com o valor nulo do SQL. A opção **CONSTANT** impede que seja atribuído valor a variável e, portanto, seu valor permanece constante pela duração do bloco. Se for especificado **NOT NULL**, uma atribuição de valor nulo resulta em um erro em tempo de execução. Todas as variáveis declaradas como **NOT NULL** devem ter um valor padrão não nulo especificado.

Cópia de Tipo

variável%TYPE

A expressão **%TYPE** fornece o tipo de dado da variável ou da coluna da tabela. Pode ser utilizada para declarar variáveis que armazenam valores do banco de dados.

Exemplo:

```
aluno_nome aluno.nome%TYPE;
```

Utilizando **%TYPE** não é necessário conhecer o tipo de dado da estrutura sendo referenciada e, ainda mais importante, se o tipo de dado do item referenciado mudar no futuro, não será necessário mudar a definição na função.

Tipo-Linha

```
nome nome_da_tabela%ROWTYPE;
nome nome_do_tipo_composto;
```

Uma variável de tipo composto é chamada de variável linha (ou variável tipo-linha). Este tipo de variável pode armazenar toda uma linha de resultado de um comando **SELECT** ou **FOR**, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável. Os campos individuais do valor linha são acessados utilizando a notação usual de ponto como, por exemplo, variável_linha.campo.

Uma variável-linha pode ser declarada como tendo o mesmo tipo de dado das linhas de uma tabela ou de uma visão existente, utilizando a notação **nome_da_tabela%ROWTYPE** ou pode ser declarada especificando o nome de um tipo composto pois todas as tabelas possuem um tipo composto associado, que possui o mesmo nome da tabela.

Tipo-Linha

Exemplo:

```
CREATE FUNCTION cliente_nome( t cliente ) RETURNS TEXT AS
$$
  BEGIN
    RETURN t.sobrenome || ' ' || t.nome;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT cliente_nome( cliente.* ) FROM cliente;
  cliente_nome
---------------
 Santos Jose
 Cunha Paulo
 Alves Maria
 Silveira Joana
 Batista Luis
 Ramalho Marcia
```

Tipo Registro

```
nome RECORD;
```

As variáveis registro são semelhantes às variáveis tipo-linha, mas não possuem uma estrutura pré-definida. Assumem a estrutura da linha para a qual são atribuídas pelo comando **SELECT** ou **FOR**. A subestrutura da variável registro pode mudar toda vez que é usada em uma atribuição.

Tipo Registro

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS TEXT AS $$
  DECLARE
    t RECORD;
  BEGIN
    SELECT INTO t * FROM cliente WHERE codigo=cod;
    RETURN t.sobrenome || ' ' || t.nome;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT cliente_nome( codigo ) FROM cliente;
  cliente_nome
---------------
 Santos Jose
 Cunha Paulo
 Alves Maria
 Silveira Joana
 Batista Luis
 Ramalho Marcia
```

IF

A instrução **IF** permite a execução condicional dos comandos.

```
IF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
    ...]]
[ ELSE
    instruções ]
END IF;
```

Pode ser utilizado **ELSEIF** no lugar de **ELSIF**.

IF

Exemplo:

```
CREATE FUNCTION sexo_extenso( sexo CHAR(1) ) RETURNS TEXT
AS $$
  DECLARE
    ret TEXT;
  BEGIN
    IF sexo = 'M' THEN
      ret = 'Masculino';
    ELSIF sexo = 'F' THEN
      ret = 'Feminino';
    ELSE
      ret = 'Valor Incorreto';
    END IF;
    RETURN ret;
  END;
$$ LANGUAGE plpgsql;
```

IF

Teste:

```
SELECT sexo_extenso( 'F' );
 sexo_extenso
--------------
 Feminino
```

CASE

A instrução **CASE** permite também permite a execução condicional dos comandos.

Essa instrução pode ser utilizada de duas formas.

Pode ser usada com a verificação de condições independentes:

```
CASE
  WHEN expressão_booleana THEN
    instruções
[ WHEN expressão_booleana THEN
    instruções
[ WHEN expressão_booleana THEN
    instruções
    ...]]
[ ELSE
    instruções ]
END CASE;
```

CASE

Exemplo:

```
CREATE FUNCTION sexo_extenso( sexo CHAR(1) ) RETURNS TEXT
AS $$
  DECLARE
    ret TEXT;
  BEGIN
    CASE
      WHEN sexo = 'M' THEN
        ret = 'Masculino';
      WHEN sexo = 'F' THEN
        ret = 'Feminino';
      ELSE
        ret = 'Valor Incorreto';
    END CASE;
    RETURN ret;
  END;
$$ LANGUAGE plpgsql;
```

CASE

Teste:

```
SELECT sexo_extenso( 'F' );
 sexo_extenso
--------------
 Feminino
```

CASE

A instrução **CASE** também pode ser usada para comparar o valor de uma expressão.

```
CASE expressão
  WHEN expressão [, expressão [ ... ]] THEN
    instruções
[ WHEN expressão [, expressão [ ... ]] THEN
    instruções
    ... ]
[ ELSE
    instruções ]
END CASE;
```

CASE

Exemplo:

```
CREATE FUNCTION grau_experiencia( funcao funcao.funcao
%TYPE ) RETURNS TEXT AS $$
  DECLARE
    ret TEXT;
  BEGIN
    CASE funcao
      WHEN 1,4 THEN
        ret = 'experiente';
      WHEN 3 THEN
        ret = 'intermediario';
      WHEN 2,5 THEN
        ret = 'iniciante';
      ELSE
        ret = 'Valor Incorreto';
    END CASE;
    RETURN ret;
  END;
$$ LANGUAGE plpgsql;
```

CASE

Teste:

```
SELECT grau_experiencia(funcao) FROM funcionario WHERE
matricula='12342';
 grau_experiencia
------------------
 experiente
```

SELECT INTO

O resultado de um comando **SELECT** que retorna várias colunas (mas apenas uma linha) pode ser atribuído a uma variável registro, a uma variável tipo-linha, ou a uma lista de variáveis escalares. É feito através de

```
SELECT INTO destino expressões_de_seleção FROM ...;
```

onde destino pode ser uma variável registro, uma variável linha, ou uma lista separada por vírgulas de variáveis simples e campos de registro/linha. A expressões_de_seleção e o restante do comando são os mesmos que no SQL comum.

Se a consulta não retornar nenhuma linha, são atribuídos valores nulos aos destinos. Se a consulta retornar várias linhas, a primeira linha é atribuída aos destinos e as demais são desprezadas.

A variável especial **FOUND** pode ser verificada imediatamente após a instrução **SELECT INTO** para determinar se a atribuição foi bem-sucedida, ou seja, foi retornada pelo menos uma linha pela consulta.

SELECT INTO

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS text AS $$
DECLARE
  c_nome cliente.nome%TYPE;
  c_sobrenome cliente.sobrenome%TYPE;
BEGIN
    SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
    IF FOUND THEN
      RETURN c_sobrenome || ' ' || c_nome;
    ELSE
      RETURN 'nao encontrado';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

SELECT INTO

Teste:

```
SELECT cliente_nome( '01' );
 cliente_nome
--------------
 Santos Jose
SELECT cliente_nome( '07' );
  cliente_nome
----------------
 nao encontrado
```

SELECT INTO

A expressão de seleção pode ser colocada antes da cláusula **INTO**. Dessa forma, a linha:

```
SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
```

É equivalente a:

```
SELECT nome,sobrenome INTO c_nome,c_sobrenome FROM
cliente WHERE codigo=cod;
```

Recursividade

As funções em PL/pgSQL são recursivas.

Exemplo:

```
CREATE FUNCTION fatorial(n INTEGER) RETURNS INTEGER AS $$
  DECLARE
    ret INTEGER;
  BEGIN
    IF ( n<=0 ) THEN
      ret = 1;
    ELSE
     ret = n*fatorial(n-1);
    END IF;
    RETURN ret;
  END;
$$ LANGUAGE plpgsql;
```

Teste:

```
SELECT fatorial( 5 );
 fatorial
----------
      120
```

RAISE

O comando **RAISE** gera mensagens informativas ou erros na execução das funções.

```
RAISE nível 'formato' [, variável [, …]] [ USING opcao = expressao [, ... ] ];
```

Os níveis possíveis são **DEBUG**, **LOG**, **INFO**, **NOTICE**, **WARNING**, e **EXCEPTION**. O nível **EXCEPTION** causa um erro que interrompe a transação corrente. Os outros níveis apenas geram mensagens com diferentes níveis de prioridade.

Se as mensagens de uma determinada prioridade são informadas ao cliente, escritas no log do servidor, ou as duas coisas, é controlado por variáveis de configuração do servidor.

RAISE

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS text AS $$
  DECLARE
    c_nome cliente.nome%TYPE;
    c_sobrenome cliente.sobrenome%TYPE;
  BEGIN
    SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
    IF FOUND THEN
      RAISE NOTICE 'Cliente encontrado';
      RETURN c_sobrenome || ' ' || c_nome;
    ELSE
      RAISE EXCEPTION 'Codigo % nao encontrado', cod;
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

RAISE

Teste:

```
SELECT cliente_nome( '01' );
NOTA:  Cliente encontrado
 cliente_nome
--------------
 Santos Jose

SELECT cliente_nome( '07' );
ERROR:  Codigo 07 nao encontrado
```

RAISE

É possível adicionar informações sobre o erro utilizando a cláusula USING.

As opções possíveis são:

MESSAGE – especifica o texto da mensagem de erro, não pode ser usado se existe uma string de formatação da mensagem de erro antes do USING.

DETAIL – especifica uma mensagem mais detalha sobre o erro

HINT – especifica uma dica sobre o erro

ERRCODE – especifica o código para o erro

RAISE

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS text AS $$
  DECLARE
    c_nome cliente.nome%TYPE;
    c_sobrenome cliente.sobrenome%TYPE;
  BEGIN
    SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
    IF FOUND THEN
      RETURN c_sobrenome || ' ' || c_nome;
    ELSE
      RAISE EXCEPTION USING MESSAGE = 'Codigo ' || cod ||
' nao encontrado', DETAIL = 'Nao existe um registro com o
codigo informado no cadastro de clientes', HINT =
'Verique o codigo do cliente';
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

RAISE

Teste:

```
SELECT cliente_nome( '07' );
ERROR:  Codigo 07 nao encontrado
DETAIL:  Nao existe um registro com o codigo informado no
cadastro de clientes
HINT:  Verique o codigo do cliente
```

RAISE

O erro pode ser especificado tanto pelo nome da condição ou pelo código de erro (**SQLSTATE**).

Existe uma tabela padrão para códigos de erros. Os códigos de erro devem ter 5 caracteres, sendo que os dois primeiros indicam a classe do erro e os demais indicam a condição dentro da classe.

Não é obrigatório que os **SGBDs** utilizem exatamente a tabela de erros padrão, podendo tanto não utilizar alguns códigos da tabela, quanto utilizar códigos próprios. A tabela com os nomes de condições e códigos de erro devem ser verificadas na documentação do **PostgreSQL**.

Se não for especificada uma mensagem de erro, será usado o nome da condição ou o código do erro como mensagem de erro.

RAISE

O nome da condição deve ser um nome utilizado pelo SGBD.

```
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
```

RAISE

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS text AS $$
  DECLARE
    c_nome cliente.nome%TYPE;
    c_sobrenome cliente.sobrenome%TYPE;
  BEGIN
    SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
    IF FOUND THEN
      RETURN c_sobrenome || ' ' || c_nome;
    ELSE
      RAISE NO_DATA_FOUND USING DETAIL = 'Codigo ' || cod
|| ' nao encontrado';
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

RAISE

Teste:

```
SELECT cliente_nome( '07' );
ERROR:  no_data_found
DETAIL:  Codigo 07 nao encontrado
```

RAISE

O **SQLSTATE** pode ser um código do erro utilizado pelo **SGBD** ou um código próprio com 5 caracteres, diferente de **00000**.

Não é recomendado utilizar códigos de erro que terminem em **000** pois são considerados códigos de categoria.

Se não for especificado um nome de condição ou código de erro para um **RAISE EXCEPTION**, será utilizado o valor padrão **RAISE_EXCEPTION** (**P0001**).

```
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
```

RAISE

Exemplo:

```
CREATE FUNCTION cliente_nome( cod cliente.codigo%TYPE )
RETURNS text AS $$
  DECLARE
    c_nome cliente.nome%TYPE;
    c_sobrenome cliente.sobrenome%TYPE;
  BEGIN
    SELECT INTO c_nome,c_sobrenome nome,sobrenome FROM
cliente WHERE codigo=cod;
    IF FOUND THEN
      RETURN c_sobrenome || ' ' || c_nome;
    ELSE
      RAISE EXCEPTION SQLSTATE 'MY001' USING DETAIL =
'Codigo ' || cod || ' nao encontrado';
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

RAISE

Exemplo:

```
SELECT cliente_nome( '07' );
ERROR:  MY001
DETAIL:  Codigo 07 nao encontrado
```

Exercícios

1) Crie uma função que receba como parâmetro o código da revenda e retorne o lucro médio das vendas feitas pela revenda.

2) Crie uma função que receba como parâmetro a matricula do funcionário e retorne o nome da diretoria a qual o funcionário esta subordinado ou "nao alocado" se o funcionário não pertencer a nenhuma diretoria. A função deve gerar um erro caso não exista funcionário com a matricula indicada, com uma mensagem indicando o erro ocorrido.

3) Monte o script para criar a seguinte tabela:

ProdutoLote(produto,lote,quantidade)
- **produto – 2 caracteres, chave estrangeira para a tabela produto**
- **lote – 10 caracteres**
- **quantidade - inteiro**

Exercícios

4) Crie uma função que receba como parâmetros o código do produto, numero do lote e quantidade e acrescente essa quantidade na tabela produtolote. Se o produto não existir, a função deve gerar um erro indicando que o código não foi encontrado. Se a quantidade do estoque se tornar negativa, a função deve emitir uma mensagem avisando que a quantidade em estoque do lote esta negativa. A função deve retornar o saldo do produto/lote.

5) Crie uma função que receba um nome como parâmetro e retorne o superior máximo desse nome na organização.

6) Crie uma função que receba o código de uma conta, código do grupo, data e valor de um lançamento e insira esse lançamento na tabela se a conta e o grupo existirem e se o valor corresponder ao tipo do grupo (grupos de receita só podem ter valores positivos e grupos de despesa só podem ter valores negativos). Se não for possível inserir o lançamento, a função deve gerar um erro com uma mensagem explicativa sobre o erro ocorrido. A função não deve retornar nenhum valor.