Funções

Gonzalo Travieso

2019

1 Funções em C++

O termo **função** é usado em C++ para designar um trecho de código ao qual (normalmente) se dá um nome e que pode ter sua execução realizada quando conveniente.[1] Existem duas razões principais para se realizar isso:

1. **Para reuso de código**, se um mesmo código aparece repetidamente num programa, sua definição como uma função permite que ele seja escrito apenas uma vez, facilitando o desenvolvimento e a depuração.
2. **Para organização do código**, visto que trechos muito longos de código são difíceis de entender, quebramos esses trechos em funções distintas, que podem ser desenvolvidas separadamente.

2 Definição e uso de funções

Para definir uma função, precisamos especificar seu nome, os parâmetros que ela aceita e o tipo do valor de retorno. Parâmetros e valores de retorno são opcionais. A função mais simples possível, e que não faz nada, seria definida da seguinte forma:

```
void nada()
{

}
```

O `void` diz que essa função não retorna valores, os parêntesis têm dentro deles os parâmetros; neste caso, não há nenhum parâmetro. As chaves delimitam o bloco de comandos a serem executados pela função; neste caso, nenhum comando é executado.

Para indicar que queremos que o código da função seja executado, usamos a sintaxe (no ponto onde queremos a execução):

```
nada();
```

Neste caso, nada será executado. Podemos executar a função quantas vezes quisermos:

[1] Estou desconsiderando a possibilidade de definir trechos de código através de macros, pois isso não é recomendado em C++ moderno.

```
nada();
nada();
for (int i = 0; i < 100000; ++i) {
    nada();
}
```

A função seguinte já faz alguma coisa:

```
void calcula_10_fatorial()
{
    int fat{1};
    for (int i = 2; i <= 10; ++i) {
        fat *= i;
    }
}
```

Como o nome diz, ela calcula o fatorial de 10. Infelizmente, não há como usar o valor calculado na sua chamada, e portanto o trabalho realizado é inútil.

Já a seguinte função realiza algum trabalho que tem efeito na execução do programa (além de gastar tempo):

```
void saudacao()
{
    std::cout << "Estou na funcao saudacao\n";
}
```

O seguinte código fará com que a mensagem acima seja mostrada 10 vezes:

```
for (int i = 0; i < 10; ++i) {
    saudacao();
}
```

3 Parametros e valores de retorno

Se queremos que a função que calcula o fatorial de 10 tenha alguma utilidade, precisamos fazer com que ela retorne o valor calculado:

```
int fatorial_de_10()
{
    int fat{1};
    for (int i = 2; i <= 10; ++i) {
        fat *= i;
    }
    return fat;
}
```

O `int` antes do nome da função indica que essa função retorna um valor do tipo `int`. Para retornar o valor, o código da função deve executar o comando `return`, passando em seguida o valor a retornar, como mostrado na última linha. A execução do comando `return` termina a execução da função.

Ao usar essa função, a chamada da função é considerada como uma expressão, que tem o valor indicado pelo valor retornado:

```
int dez_fat = fatorial_de_10();
int dez_fat_menos_2 = fatorial_de_10() - 2;
```

Isso já é mais útil do que a versão anterior; entretanto, ainda é bastante rígido: temos apenas o fatorial de 10. Se quisermos que a função fique mais flexível, precisamos fazer com que ela calcule o fatorial de qualquer inteiro positivo. Isso pode ser conseguido especificando o número para o qual queremos o fatorial como *parâmetro* da função:

```
int fatorial(int n)
{
    int fat = 1;
    for (int i = 2; i <= n; ++i) {
        fat *= i;
    }
    return fat;
}
```

O `int n` entre parêntesis após o nome da função diz que essa função tem como parâmetro um inteiro que receberá o nome de `n` no código da função. Esse `n` é uma **variável** do tipo `int`, que terá seu valor **inicializado** com o valor passado na chamada da função.

```
int fat_3 = fatorial(3);
int fat_fat_3 = fatorial(fat_3);
```

Na primeira chamada acima, a variável (parâmetro) `n` da função `fatorial` será inicializada com 3, provocando o cálculo do fatorial de 3 (que é 6); na segunda chamada, a variável `n` será inicializada com o conteúdo da variável `fat_3`, isto é, 6, realizando então o cálculo do fatorial de 6.

Uma função pode ter qualquer número de parâmetros (os tipos dos parâmetros também podem ser distintos). Por exemplo, uma função para calcular o número de combinações de n elementos m a m pode ser:

```
int combinacoes(int n, int m)
{
    return fatorial(n) /
        (fatorial(m) * fatorial(n-m));
}
```

Note como essa função faz chamadas repetidas à função `fatorial` previamente definida.

Os valores retornados e passados como parâmetro não precisam ser tipos simples de C++, mas podem ser tipos mais complexos, definidos em alguma biblioteca ou pelo usuário. Por exemplo, se queremos calcular os n primeiros elementos da sequência de Fibonacci (começando em 1, 1), podemos definir:

```
std::vector<int> fibonacci(int n)
{
    std::vector<int> sequencia{1, 1};
    // sequencia[0] == 1 && sequencia[1] == 1
    for (int i = 2; i < n; ++i) {
        auto novo = sequencia[i-1] + sequencia[i-2];
```

```
        sequencia.push_back(novo);
    }

    return sequencia;
}
```

Note como o tipo definido para o valor de retorno é `std::vector<int>`, que contém os valores da sequência como seus elementos. Em situações normais, o código gerado pelo compilador C++ (com otimização) para esse tipo de retorno é bastante eficiente.

Um modo alternativo de especificar o valor de retorno de uma função é demonstrado no código abaixo:

```
auto fibonacci(int n) -> std::vector<int>
{
    std::vector<int> sequencia{1, 1};
    // sequencia[0] == 1 && sequencia[1] == 1
    for (int i = 2; i < n; ++i) {
        auto novo = sequencia[i-1] + sequencia[i-2];
        sequencia.push_back(novo);
    }

    return sequencia;
}
```

Note que neste caso, colocamos a palavra-chave `auto` no tipo de retorno, e então especificamos o tipo de retorno depois da lista de argumentos, em seguida de um `->`. Isso é denominado *trailing return type*, e é especialmente útil em conjunto com *templates* (a serem estudados mais tarde) quando o tipo de retorno depende do tipo dos argumentos. Ademais neste caso, o tipo específico de retorno pode ser omitido, se ele puser ser deduzido pela análise das expressões `return` na função:

```
auto fibonacci(int n)
{
    std::vector<int> sequencia{1, 1};
    // sequencia[0] == 1 && sequencia[1] == 1
    for (int i = 2; i < n; ++i) {
        auto novo = sequencia[i-1] + sequencia[i-2];
        sequencia.push_back(novo);
    }

    return sequencia;
}
```

Neste código o compilador deduz que o tipo de retorno é `std::vector<int>`, pois esse é o tipo da variável `sequencia` usada no `return`. Isto é útil quando o tipo de retorno é especialmente complexo, mas pode ser utilizado em qualquer situação. Note que, neste caso, se a função tem mais do que um `return`, todos eles devem especificar o mesmo tipo de dados. Por exemplo, o seguinte código é inválido:

```
auto trunca_zero(double x)
{
    if (x > 0) {
        return x;
    }
    else {
        return 0;
    }
}
```

O problema é que no primeiro `return` é especificado o valor de `x`, que é uma variável `double`, enquanto no segundo é especificado o literal `0`, que é do tipo `int`. Para acertar o código temos duas possibilidades: Uma é especificar um `double` em todos os `return`:

```
auto trunca_zero(double x)
{
    if (x > 0) {
        return x;
    }
    else {
        return 0.0;
    }
}
```

Neste caso, os dois `return` especificam o mesmo tipo e o compilador decide que a função retorna `double`. Uma outra opção é especificarmos o tipo explicitamente:

```
auto trunca_zero(double x) -> double
{
    if (x > 0) {
        return x;
    }
    else {
        return 0;
    }
}
```

Neste caso, como o compilador sabe que o tipo a ser retornado é `double`, ele irá converter o `0` para `0.0` automaticamente.

4 Parâmetros de referência

Às vezes precisamos não apenas ler os valores de um parâmetro, mas também alterá-los. Por exemplo, suponha que queremos realizar o cálculo do produto de todos os elementos de um vetor por um valor fixo, colocando o resultado no vetor original. Num código sem funções, podemos fazer:

```
std::vector<double> v(N);
double fator_escala{3};
```

```
// inicializa os v[i] de alguma forma

for (int i = 0; i < N; ++i) {
    v[i] *= fator_escala;
}
```

No entanto, se esse tipo de código for frequente, podemos querer criar uma função. Uma opção seria:

```
std::vector<double> escala_v0(std::vector<double> v, double fator)
{
    int N = static_cast<int>(v.size());
    std::vector<double> novo_v{N};
    for (int i = 0; i < N; ++i) {
        novo_v[i] = v[i] * fator;
    }
    return novo_v;
}
```

E então usamos a função da seguinte forma:

```
std::vector<double> v(N);
double fator_escala{3};

// inicializa os v[i] de alguma forma

v = escala_v0(v, fator_escala);
```

Esse esquema não é muito eficiente pelos seguintes fatores:

- O parâmetro `v` da função `escala_v0` será uma nova variável do tipo `std::vector<double>`, inicializada com o valor da variável `v` do ponto da chamada, o que implica uma *cópia de todos os elementos de um vetor no outro*.
- Os novos elementos são calculados em um novo vetor `novo_v`, *duplicando a quantidade de memória necessária*.
- No retorno, o vetor retornado pela função deve ser transferido para a variável `v`. Essa transferência é feita de forma eficiente, mas implica *liberar a memória anteriormente usada pelo vetor* `v`, uma operação não presente no código original.

Para contornar esse tipo de problemas, podemos definir um parâmetro de função como sendo uma **referência**. Deste modo, o parâmetro é apenas um *sinônimo* para a variável passada na chamada. O código fica:

```
void escala(std::vector<double> &v, double fator)
{
    for (auto &vi: v) {
        vi *= fator;
    }
}
```

(Aqui fizemos uso do `for auto` com uma variável de referência `vi`, para poder alterar o valor dos elementos do vetor.) O ponto importante aqui é que, ao colocarmos o `&` no parâmetro `v`, estamos dizendo que ele é apenas uma referência. Portanto, ao usá-lo da seguinte forma:

```
std::vector<double> v(N);
double fator_escala{3};

// inicializa os v[i] de alguma forma

escala(v, fator_escala);
```

O `v` referenciado no código da função `escala` será um **sinônimo** para o `v` do ponto de chamada, e portanto qualquer alteração será realizada no vetor original. Uma vantagem adicional é que, como estamos lidando apenas com uma referência, *não existe cópia dos elementos do vetor*. Também *não precisamos retornar o vetor*, pois os valores foram alterados no vetor original.

Como dito acima, o uso de referência em um parâmetro de função evita que o valor seja copiado para uma nova variável ao ser realizada a chamada. Isso é muito útil quando o valor passado é grande, por exemplo no caso de vetores ou cadeias grandes de caracteres. Por essa razão, nessas situações usamos referência **mesmo quando não vamos fazer alteração no valor**. Para deixar claro que não vai haver alteração, e que a referência está sendo usada apenas para evitar uma cópia, marcamos o parâmetro como constante. Veja o exemplo seguinte:

```
double soma(std::vector<double> const &v)
{
    double resultado{0};
    for (auto vi: v) {
        resultado += vi;
    }
    return resultado;
}
```

Esta função recebe um vetor e retorna a soma de seus valores. Os valores armazendos no vetor não são alterados, mas se o vetor fosse passado por valor, haveria uma cópia de todos os elementos. A passagem por referência evita essa cópia, e o `const` garante que os elementos do vetor não serão alterados.

5 Protótipos

Ao contrário de C, a linguagem C++ exige que a interface de uma função seja conhecida antes que uma chamada a essa função seja realizada. Se a definição da função aparece antes de seu uso (na ordem do arquivo de código fonte), então isso não é problema. Mas em algumas situações queremos usar uma função cuja definição não foi apresentada no código. As duas principais razões são:

- Estamos usando uma função que não é implementada por nosso código (da biblioteca ou implementada em outro arquivo).
- Queremos apresentar o código que usa a função antes de apresentar o código da função, por uma questão de organização lógica do arquivo. De

fato, em geral apresentamos primeiro as partes mais gerais do código, e deixamos as partes mais específicas para depois.

Estes dois casos são resolvidos com o uso de **protótipos**, que declaram a interface de uma função sem apresentar seu código. No caso do primeiro item acima, o protótipo é dado em um arquivo de cabeçalho, introduzido em nosso código por um `#include`. Já no segundo caso, precisamos escrever o protótipo explicitamente. O protótipo consiste na declaração de nome, tipo de retorno e parâmetros da função, mas sem a declaração do corpo (bloco de comandos) a ser executado. Por exemplo, para a função `soma` do exemplo anterior, o protótipo será:

```
double soma(std::vector<double> const &v);
```

Note como o protótipo é terminado com um `;`.

6 Parâmetros com valores assumidos

Ao realizarmos a chamada de uma função, precisamos especificar todos os seus parâmetros. Em algumas situações, certos parâmetros têm valores que são usados na maioria dos casos. Neste caso, seria útil não precisar especificar esse valor comum. Isso pode ser feito especificando um valor assumido (*default*) para esses parâmetros.

Por exemplo, suponha que queremos escrever uma função que dado um número inteiro positivo n calcula a somatória

$$\sum_{k=1}^{n} k^2$$

mas queremos deixar a opção do usuário especificar um outro valor inicial, ao invés de 1, isto é, queremos calcular

$$\sum_{k=i}^{n} k^2$$

sendo que sabemos que i normalmente será 1. Podemos escrever:

```
int soma_quadrados(int n, int i = 1)
{
    int soma{0};
    for (int k = i; k <= n; ++k) {
        soma += k * k;
    }
    return soma;
}
```

A declaração de parâmetro `int i = 1` indica para o compilador que o parâmetro `i` pode ser omitido na chamada da função, e nesse caso ele valerá 1.

```
auto a = soma_quadrados(10, 2); // Soma dos quadrados de 2 a 10
auto b = soma_quadrados(10, 1); // Soma dos quadrados de 1 a 10
auto c = soma_quadrados(10); // Soma dos quadrados de 1 a 10
```

O uso de valores assumidos para parâmetros tem algumas restrições:

- Na declaração da função, os parâmetros com valor assumido devem ser os últimos, isto é, não se pode declarar parâmetro sem valor assumido depois de declarar um parâmetro com valor assumido.
- Na chamada da função, se assumimos o valor para um parâmetro, devemos assumir o valor para todos os parâmetros seguintes, isto é, não há como usar o valor assumido para um parâmetro mas fornecer um valor diferente do assumido para um parâmetro posterior.
- Quando usamos protótipos, o valor a assumir deve ser especificado ou no protótipo ou na definição da função, mas não nos dois. O recomendado é especificar no protótipo.

Vejamos o caso da função seguinte:

```
// Prototipo
std::vector<double> comb_lin(std::vector<double> const &v1,
                             std::vector<double> const &v2,
                             double a = 1, double b = 1);

// Definicao
std::vector<double> comb_lin(std::vector<double> const &v1,
                             std::vector<double> const &v2,
                             double a, double b)
{
    auto n = static_cast<int>(std::min(v1.size(), v2.size()));
    std::vector<double> resultado(n);
    for (int i = 0; i < n; ++i) {
        resultado[i] = a * v1[i] + b * v2[i];
    }

    return resultado;
}
```

A função calcula uma combinação linear $a\mathrm{v}1 + b\mathrm{v}2$ de dois vetores, e assume que normalmente os coeficientes da combinação são 1. Note como os valores assumidos para os coeficientes a e b são apresentados apenas no protótipo (que normalmente está num arquivo de cabeçalho), enquanto na definição da função não apresentamos o valor assumido.

Essa função pode ser chamada de 3 formas:

```
std::vector<double> x(10, 2); // x[i] == 2 para i em 0..9
std::vector<double> y(10, 3); // y[i] == 3 para i em 0..9

auto t = comb_lin(x, y, 5, 4); // Calcula t = 5*x + 4*y
auto w = comb_lin(x, y, 7); // Calcula w = 7*x + y
auto z = comb_lin(x, y); // Calcula z = x + y
```

Veja que não é possível especificar um valor diferente de 1 para o parâmetro `b` se assumimos 1 para o parâmetro `a`. A saída nesse caso é especificar o 1 do `a` explicitamente:

```
auto s = comb_lin(x, y, 1, 8); // Calcula t = x + 8*y
```

7 Retorno de múltiplos valores

C++ permite o retorno de apenas um valor da função. Se quisermos retornar mais do que um valor, devemos retornar um valor composto, constituido dos diversos valores que desejamos retornar. As principais opções para fazer isso são:

- Se todos os valores a retornar são do mesmo tipo, podemos retornar um `std::array<tipo, n>` (onde `tipo` é o tipo dos dados e `n` é o número de valores a retornar; o número de valores precisa ser fixo, não pode ser calculado).
- Uma outra opção é retornar uma tupla com todos os valores desejados.
- Por fim, podemos também criar uma `struct` (a ser estudada posteriormente) para carregar os valores a retornar.

Vejamos essas opções num exemplo específico. Suponha que fazemos uma função para calcular o quociente e o resto da divisão de dois número inteiros. Usando o método do `array` declaramos:

```
std::array<int, 2> div_mod_array(int n, int m)
{
    auto div = n / m;
    auto mod = n % m;
    return {div, mod};
}
```

Usando uma tupla, podemos declarar:

```
std::tuple<int, int> div_mod_tupla(int n, int m)
{
    auto div = n / m;
    auto mod = n % m;
    return {div, mod};
}
```

Para o caso de `struct`, fazemos:

```
// Declara um tipo struct apropriado
struct div_mod_t
{
    int quociente, resto;
};

div_mod_t div_mod_struct(int n, int m)
{
    auto div = n / m;
    auto mod = n % m;
    return {div, mod};
}
```

O uso dessas funções seria:

```
auto res_array = div_mod_array(10, 3);
std::cout << "Quociente: " << res_array[0]
          << ", resto: " << res_array[1] << std::endl;
auto res_tupla = div_mod_tupla(10, 3);
std::cout << "Quociente: " << std::get<0>(res_tupla)
          << ", resto: " << std::get<1>(res_array) << std::endl;
auto res_struct = div_mod_struct(10, 3);
std::cout << "Quociente: " << res_struct.quociente
          << ", resto: " << res_struct.resto << std::endl;
```

As vantagens e desvantagens de cada opção são:

- O uso de `array` permite uma sintaxe fácil de acesso aos elementos retornados, e também facilita se o número de elementos for aumentar (por exemplo, se estamos retornando coordenadas em 2D e depois queremos passar para 3D). Por outro lado, este método só funciona se todos os elementos retornados forem do mesmo tipo, o que não é sempre o caso; além disso, os valores serem colocados em um `array` sugere que eles são partes (componentes) de uma coisa maior (o que não é o caso no nosso exemplo)
- O uso de `tuplas` permite tratar com quaisquer mistura de tipos. As desvantagens são o modo de acesso aos valores retornados (com `std::get` e sem possibilidade de usar repetições) e a maior dificuldade de expansão (precisamos alterar a tupla retornada).
- O uso de `struct` tem a mesma vantagem de flexibilidade quanto a tipos da tupla, e tem uma forma de acesso mais adequada (é mais fácil de entender o que cada valor retornado significa). Por outro lado, é necessário definir um tipo para ser usado especialmente para o retorno da função.

Para facilitar o uso dos valores retornados, o padrão C++17 inclui a possibilidade de usar esses múltiplos valores diretamente para a inicialização de variáveis, como exemplificado nos códigos abaixo:

```
auto [q1, r1] = div_mod_array(11, 4);
std::cout << "Quociente: " << q1
          << ", resto: " << r1 << std::endl;
auto [q2, r2] = div_mod_tupla(11, 4);
std::cout << "Quociente: " << q2
          << ", resto: " << r2 << std::endl;
auto [q3, r3] = div_mod_struct(11, 4);
std::cout << "Quociente: " << q3
          << ", resto: " << r3 << std::endl;
```

Neste código, as variáveis `q1` e `r1` são criadas e têm valores iniciais dados pelos valores retornados pela função `div_mod_array`, na ordem. O mesmo ocorre nos outros exemplos. Note como esta sintaxe elimina a principal desvantagem do uso de tuplas para o retorno de múltiplos valores, e porisso a recomendação é **usar tuplas para retornar múltiplos valores**, a menos que exista uma indicação importante do uso de outro método (por exemplo, todos os valores são do mesmo tipo e prevemos que pode aumentar o número de valores em nova versão, neste caso `array` pode ser mais adequado).

8 Sobrecarga de nome de funções

Em C++, a identidade de uma função não é determinada apenas por seu nome, mas pelo nome em conjunto com o número e tipo de cada um dos parâmetros (mas não pelo tipo do valor de retorno). Objetivamente isto significa que podemos definir diferentes funções com o mesmo nome se elas atuam sobre tipos de dados distintos ou têm números distintos de parâmetros.

Como exemplo, suponhamos que estamos trabalhando num código muito antigo que opera com vetores bidimensionais de forma inconsistente. Em alguns lugares são usadas duas variáveis distintas para as coordenadas x e y, em outros lugares usa-se um `std::array<double, 2>`, em outros lugares usa-se uma `std::tuple<double, double>` e ainda em outros uma `struct` denominada `Coord` com campos `x` e `y`. Se desejamos ter uma função denominada `modulo` que calcula o módulo do vetor e que funciona para todos os casos, podemos definir diversas funções com esse nome, como abaixo:

```
double modulo(double x, double y)
{
    return sqrt(x * x + y * y);
}

double modulo(std::array<double, 2> v)
{
    return modulo(v[0], v[1]);
}

double modulo(std::tuple<double, double> v)
{
    return modulo(std::get<0>(v), std::get<1>(v));
}

double modulo(Coord v)
{
    return modulo(v.x, v.y);
}
```

Apesar de todas as funções terem o mesmo nome, o C++ as distingue pelo tipo do parâmetro passado. Isto é, quando uma função denominada `modulo` for chamada, o C++ vai verificar o tipo do argumento passado e encontrar uma função que tenha parâmetro compatível.

```
double x{3}, y{4};
// Inicializa a_v[0] == 3.0, a_v[1] == 4.0
std::array<double, 2> a_v{3.0, 4.0};
// Inicializa get<0>(t_v) == 3.0 e get<1>(t_v) == 4.0
std::tuple<double, double> t_v{3.0, 4.0};
// Inicializa s_v.x = 3.0 e s_v.y = 4.0
Coord s_v{3.0, 4.0};

// mais codigo
```

```
auto xymod = modulo(x, y); // Chama modulo(double, double)
auto amod  = modulo(a_v);  // Chama modulo(array<double, 2>)
auto tmod  = modulo(t_v);  // Chama modulo(tuple<double, double>)
auto smod  = modulo(s_v);  // Chama module(Coord)
```

8.1 Sobrecarga e valores assumidos

Existe uma relação entre sobrecarga de nome de função e valores assumidos para os argumentos: Para o compilador C++, uma definição de função com valores assumidos é o mesmo que múltiplas definições com sobrecarga em todas as combinações possíveis de parâmetros omitidos na chamada. Por exemplo, uma declaração:

```
void funcao_maluca(int a, double b = 0, int c = 1, float d = 0.5);
```

equivale ao seguinte conjunto de declarações (mas usando apenas um código para a função e assumindo os valores especificados):

```
void funcao_maluca(int a, double b, int c, float d);
void funcao_maluca(int a, double b, int c);
void funcao_maluca(int a, double b);
void funcao_maluca(int a);
```

Isso significa que as duas declarações abaixo são incompatíveis:

```
int escolhe(int a, int b, int qual = 0)
{
    int retorno = -1;
    if (qual == 0) {
        retorno = a;
    }
    else if (qual == 1) {
        retorno = b;
    }
    return retorno;
}

int escolhe(int a, int b, int c, int qual = 0)
{
    int retorno = -1;
    if (qual == 0) {
        retorno = a;
    }
    else if (qual == 1) {
        retorno = b;
    }
    else if (qual == 2) {
        retorno = c;
    }
    return retorno;
}
```

A razão para a incompatibilidade é que em chamadas como a abaixo, o compilador não saberia qual das duas funções chamar:

```
auto res = escolhe(2, 10, 1);
```

Afinal, o usuário quer chamar a primeira função especificando `qual==1` ou a segunda função assumindo `qual==0`?

9 Funções anônimas e *closures*

A declaração de funções como apresentada acima pode ser inconveniente e limitada, pois as funções somente podem ser declaradas no escopo global, isto é, não podemos por exemplo definir uma função dentro de outra função. Existem duas razões principais pelas quais gostaríamos de poder fazer isso:

- Em diversas situações, queremos uma função simples que terá utilidade apenas local. Neste caso, não é necessário definir uma função que possa ser acessada por qualquer parte do programa. Isto é frequente no uso de algoritmos da biblioteca padrão de C++ que aceitam uma função como parâmetro: muitas vezes a função é bastante simples.
- Ocasionalmente (também em conjunto com o caso anterior), queremos definir uma função que tem acesso a variáveis locais, sem que essas variáveis sejam passadas como argumentos. Para isso, a nova função precisa ser definida dentro do escopo onde essas variáveis que ela vai utilizar são definidas, e não no escopo global como vimos até agora.

Isso pode ser conseguido pela definição de *funções anônimas* ou *funções lambda*, usando uma sintaxe com três elementos:

```
[ ] () {}
```

Entre `[` e `]` temos a *lista de captura*, entre `(` e `)` a *lista de argumentos* e entre `{` e `}` o bloco de código a executar quando a função for chamada. Vejamos alguns exemplos para clarificar.

```
[] (int x) { return 2 * x; }
```

define uma função que retorna o dobro do valor do argumento passado, sendo que o argumento precisa ser inteiro.

```
[] (int &x) { x *= 2; }
```

duplica o valor da variável inteira passada como argumento (por referência).

```
[] () { return exp(-1.0); }
```

retorna sempre e^{-1} (não muito útil).

```
[] (double x, double y) {
    if (x > y) return x*x - y*y;
    else return y*y - x*x;
}
```

Note que funções lambda usam dedução de tipo de retorno, como discutido acima, e as mesmas regras se aplicam.

Para usar uma função lambda, podemos colocá-la numa variável e então usar essa variável como se fosse uma função com o número e tipo de argumentos da função lambda:

```
// Supondo que todos_elementos é um std::vector<int> que já teve seu
// valor inicializado.
auto duplica = [] (int &x) { x *= 2; };
for (auto &elem: todos_elementos) {
    duplica(elem);
}
```

Uma forma comum de uso é definir a função lambda no ponto onde ela é usada, normalmente quando ela será um parâmetro para uma função. Por exemplo, o código anterior pode ser simplificado se lembramos que a biblioteca de C++ tem uma função denominada `for_each` que percorre cada um dos elementos de um conteiner (como por exemplo um `std::vector`) e aplica uma função a cada um deles. O código fica:

```
std::for_each(begin(todos_elemento), end(todos_elementos),
              [] (int &x) { x *= 2; });
```

Veja como a função lambda é definida imediatamente ao passarmos como argumento para a função. As função `begin()` e `end()` indicam o começo e o final de um conteiner (veremos mais detalhes mais tarde).

Até agora não usamos a parte do `[]`, porque nossas funções usam apenas os argumentos e valores literais. Suponha por exemplo que, ao invés de multiplicar por 2 todos os valores de um vetor, como no caso anterior, queiramos multiplicar por um fator de escala calculado na parte anterior do código, e presente numa variável denominada `beta`. Neste caso, poderíamos tentar fazer:

```
// ATENÇÃO: Código errado!
std::for_each(begin(todos_elemento), end(todos_elementos),
              [] (int &x) { x *= beta; });
```

mas isso não funciona, pois o código `x *= beta` será executado no contexto da função `for_each` que não conhece a variável `beta`. Para que o código funcione, precisamos **capturar** a variável `beta` dentro da função lambda criada:

```
std::for_each(begin(todos_elemento), end(todos_elementos),
              [beta] (int &x) { x *= beta; });
```

A presença do nome de variável `beta` na lista de captura faz com que essa variável seja acessível durante a execução da função lambda. Note que `beta` precisa ser capturada: ela não pode ser passada como parâmetro para a função lambda:

```
// ATENÇÃO: Código errado!
std::for_each(begin(todos_elemento), end(todos_elementos),
              [] (int &x, int beta) { x *= beta; });
```

isto não funciona porque a função `for_each` espera como terceiro parâmetro uma função de apenas um argumento!

Quando uma função lambda faz captura de variáveis, dizemos que ela é uma *closure*.

Como indicado, a variável capturada pode ser acessada apenas para leitura. Se queremos também alterar o valor da variável capturada, ela precisa ser capturada por referência, colocando o símbolo `&` antes de seu nome. Para um exemplo artificial, suponhamos que, além de multiplicar todos os valores por `beta`, queremos contar quantos valem zero. Podemos fazer isso com apenas uma chamada de `for_each` da seguinte forma:

```
int nzeros{0};
std::for_each(begin(todos_elementos), end(todos_elementos),
              [beta, &nzeros] (int &x) {
                  if (x == 0) ++nzeros;
                  x *= beta;
              });
```

Existem ainda duas especificações generalizadas de captura: se a lista de capturas é da forma `[=]`, então todas as variáveis existentes no escopo estão sendo capturadas para leitura; se a lista é da forma `[&]`, então todas as variáveis estão sendo capturadas para leitura e escrita.