

Funções

Uma função é um bloco autocontido dentro de um programa (ou de uma biblioteca) que pode operar na modalidade caixa preta (isto é, seu usuário, não precisa saber como ela funciona, apenas o que faz e qual é a sua interface).¹

O conceito de função é chave para quebrar a complexidade crescente dos programas de computador em unidades autônomas e manejáveis. A última vez em que soube, o Microsoft Word andava na casa dos 4.5 milhões de linhas de código, há coisa de mais de 10 anos. Imagine escrever e sobretudo manter um programa deste tamanho se o mesmo for atacada em bloco, o que se chama de programação espagueti. Uma função é um pedaço de programa que:

- Tem um nome único e possivelmente auto-explicativo
- Realiza uma única computação
- Recebe um conjunto de dados de entrada necessários para fazer a computação acima
- Devolve um ou mais resultados a quem o chamou
- Pode ser construída, testada e mantida de maneira independente em relação aos programas que a usam
- Pode ser usada em diversos locais do mesmo programa, com isto evitando duplicar linhas de código
- Garante que todos os usuários usem o mesmo código, já que ele está escrito em um único local (a função)

Note que todos estamos acostumados com este conceito desde que tenhamos estudado matemática além da 6 a série do fundamental. Quando se fala lá na matemática na função raiz quadrada, está se falando de uma função única (calcular a raiz quadrada), que recebe um valor (aquele de quem se quer obter a raiz) e devolve um resultado (a raiz quadrada calculada). Tudo isto opera na modalidade de caixa preta, já que bem poucas pessoas conhecem como se calcula uma raiz. Ao escrever $\sqrt{x}$, todos reconhecem o que é isso e não perdem um segundo sequer ao estudar como essa função opera. Na programação, a coisa é parecida, mas mais flexível, já que se a lista de funções da matemática embora grande, é mais ou menos estável, na programação é completamente flexível e opera ao gosto do arquiteto de sistemas, ou em casos mais simples, do programador. Aliás, quebrar um programa grande em um conjunto coerente e integrado de funções é habilidade importante e convenientemente valorizada pela indústria de software. Quem programa em C (ou C++) já usa e abusa do conceito de função. Todo programa C precisa ter uma função de nome `main()` e essa função é que recebe o controle quando o programa que a contém é chamado. Programas mais simples têm apenas a função `main`, mas nada impede que um programa mais complexo tenha dezenas ou centenas de funções umas chamando as outras, cedendo e recebendo dados. A regra acima entretanto permanece válida. Todo programa C tem que ter uma função principal chamada `main`. A anatomia de uma função é:

```
tipo-resposta nome-função(parâmetros) {  
... computação realizada pela função ...  
return variavel-resposta  
}
```

Neste formato cabem as seguintes definições:

tipo-resposta indica qual o tipo do dado que a função deverá retornar quando for encerrada. Pode ser qualquer um dos tipos já estudados (como por exemplo `int` ou `char...`)

nome-função um nome qualquer, sujeito às regras de formação de nomes da linguagem. Espera-se que seja autoexplicativo.

() ao lado do nome indica que este nome é uma função. Este formalismo existe para impedir que este nome seja confundido com uma variável, já que a regra de formação de nomes é a mesma. Se os parênteses aparecem sem nada dentro, como em `main()` isto significa que esta função não recebe parâmetros.

parâmetros tipos e nomes dos diversos parâmetros que a função deverá receber quando for chamada. Se houver mais de um, deverão estar separados por uma vírgula.

...computação... tudo o que você já aprendeu e sabe sobre a linguagem C++ pode e deve ser usado aqui.

return variável-resposta Este comando faz várias coisas:

- Retorna o valor que existe na variável citada para quem chamou esta função
- Encerra esta função, mesmo que haja mais comandos abaixo do `return` que, portanto, jamais serão executados
- Libera todos os recursos anteriormente alocados à execução desta função

¹Interface é uma palavra onnipresente na ciência da computação e ela sugere como objetos de software se relacionam entre si, isto é, o que eles cedem e recebem. A título de exemplo (meio pobre), pode-se dizer que a interface entre um carro e um motorista seriam a direção, os pedais, as alavancas bem como os marcadores e indicadores do painel

O ciclo de vida de uma função sugere 2 eventos separados e imprescindíveis. O primeiro é a definição da função, através do formalismo visto aí acima. Mas, só definir uma função, não causa nenhum processamento. Este ocorre num segundo momento, quando a função é chamada dentro do programa (ou dentro de outra função), quando finalmente a computação descrita dentro da função é executada. A regra aqui é que antes de executar ou chamar uma função, a mesma terá que ter sido definida. Vale, entretanto, a lembrança de que esta definição eventualmente poderá estar em outro módulo e até ser meio “invisível” para quem a está chamando. Não é exatamente o caso, mas temos algo parecido para dar como exemplo. Os “comandos” `cout` e `cin` não são funções, mas podem ser entendidos como se o fossem. Notem que eles precisam estar definidos antes de serem usados, mas a definição não é feita por nós e sim apenas agregada ao nosso programa. Quem faz isso é o comando `#include<iostream>`. A esse respeito, se você usar um `cout` no seu programa sem colocar o `#include` no programa receberá um erro dizendo que `cout` não está definida. Depois que a função foi corretamente definida, ela pode ser usada quantas vezes você quiser, sempre realizando a mesma computação para a qual foi escrita operando a cada vez sobre os parâmetros passados a ela naquela execução. Veja-se um exemplo do que se diz

```
#include<iostream>
#include<cmath>
using namespace std;
float pitagoras(float cateto1, float cateto2) {
    float a,b;
    a = cateto1 * cateto1;
    b = cateto2 * cateto2;
    return sqrt(a+b);
}
int main() {
    float c1,c2,hip;
    cout<<"Calculo de uma hipotenusa "<<endl;
    cin>>c1;
    cin>>c2;
    hip = pitagoras(c1,c2);
    cout<<"Hipotenusa "<<hip;
}
```

Veja-se quem é quem neste exemplo

pitagoras	é uma função definida pelo usuário que recebe 2 parâmetros float, calcula alguma coisa e devolve um float.
cateto1	é o primeiro parâmetro que deve ser um float.
cateto2	o segundo parâmetro, também float.
a,b	variáveis internas à função pitagoras só começam a existir no momento em que a função começa a operar e deixam de existir após a execução do return.
operações envolvendo os parâmetros	os parâmetros citados no cabeçalho são usados extensivamente no corpo da função. Deve-se notar que os parâmetros não existem fisicamente na memória da máquina. Só vão existir quando a função for chamada.
sqrt(a+b)	calcula o resultado final desta função
return	devolve o resultado final para quem chamou esta função e encerra a função. main a função principal deste programa
c1,c2 e hip	variáveis dentro do programa. Existem fisicamente na memória do computador <code>cout</code> e <code>cin</code> obtenção de dados externos e oferecimento dos mesmos
hip	a variável onde o resultado da função vai ser guardado
pitagoras	a função acima definida sendo chamada. É só agora que ela executa alguma computação
c1,c2	os valores que vão ser repassados à função. A partir deste ponto é como se <code>cateto1</code> recebesse (ele recebe) uma cópia do conteúdo de <code>c1</code> . Igualmente <code>cateto2</code> recebe uma cópia do conteúdo de <code>c2</code>

Observação: note que tanto na definição quanto na chamada, `pitagoras` não tem acento. É uma boa estratégia. Depois que foi definida, `pitagoras` funciona como se fosse uma função primitiva do C++, tal como `sqrt`.

Passagem por referência

Quando se escreve uma função em C++, normalmente os parâmetros são passados por valor e o retorno idem. Assim, em

```
int dobro(int N){
```

```

    return N*2;
}
int main(){
    int x;
    x=55;
    cout<<dobro(x);
}

```

No exemplo acima, a função `dobro` desconhece a variável `x`, enquanto que a função principal desconhece uma possível variável `N`. Assim, se a função for chamada com 55, ela vai retornar 110, como era esperado. Vamos elaborar mais um pouco este exemplo, como em

```

int dobro(int N){
    N=N+10;
    return N*2;
}
int main(){
    int x;
    x=55;
    cout<<dobro(x)<<endl;
    cout<<x<<endl;
}

```

Note que a função `dobro`, antes de dobrar o valor do parâmetro recebido, soma 10 unidades a ele. E retorna o dobro deste novo valor. Quando esta nova função for chamada, ela fará isso corretamente, mas o valor da variável original (aquela que estava definida na função chamadora, no caso, a função `main`, NÃO É ALTERADA. Chamando este trecho com 55, a função `dobro` vai somar 10 (obtendo 65) e depois vai retornar 130. A função principal vai imprimir 130, mas depois ao imprimir o valor de `x`, teremos 55, e não 65. A soma de 10 unidades ficou restrita à função “dobro” e não teve seu efeito propagado para a função chamadora. Esta é a melhor maneira de trabalhar minimizando os efeitos colaterais e garantindo que cada trecho de código tem encapsulados os seus dados. Entretanto, eventualmente, pode ser necessário abrir esta regra (a que permite que funções alterem os dados usados para chamar a função). Esta ação pode ser necessária também quando mais do que um valor precisa ser retornado. Agora, ao invés de `return`, poderemos retornar os resultados diretamente sobre os dados passados (que agora podem ser alterados pela função chamada). A chave para fazer isto é PASSAGEM DE VALORES POR REFERÊNCIA. Se na definição de uma função, colocarmos um `&` entre o tipo e o nome do parâmetro, estamos avisando ao compilador que quando a função for chamada, devem ser passados a ela não os valores da chamada, mas sim o endereço real das variáveis na memória. Logo, neste caso, eventuais alterações feitas pela função chamada serão preservadas para a função chamadora. Veja-se o exemplo

```

int dobro(int & N){
    N=N+10;
    return N*2;
}
int main(){
    int x;
    x=55;
    cout<<dobro(x)<<endl;
    cout<<x<<endl;
}

```

Agora, chamando a função com 55, a variável `x` passa a valer 55. Quando a função `dobro` é chamada, é a variável `x` que é instanciada dentro da função `dobro`. Quando a variável `N` (na realidade `x`) recebe o valor original (55) mais 10 ela passa a valer 65. A função retorna o dobro disso, mas a variável `x` original passa a valer 65. Esta habilidade é frequentemente necessária quando uma função precisa retornar mais do que um valor (quando então usaria o comando `return`). Agora, se for necessário devolver 2 valores após a chamada de uma função, será preciso:

- passar essas 2 variáveis para a função por referência (usando `&`)
- garantir que a função altere essas duas variáveis corretamente
- Não é necessário devolver nada via `return`
- Garantir que a função chamadora busque os resultados da aplicação da função diretamente nas variáveis chamadas

Por exemplo, vamos escrever uma função que receba um valor em reais e informe quantas notas de 50, 20 e 10 reais devem ser usadas para compor o valor original. Se não for possível compor o valor com essas 3 notas, deve retornar -1 nas 3 notas. Veja no exemplo:

140 deve retornar 2,2,0; significando 2 notas de 50, 2 notas de 20 e nenhuma nota de 10 reais.

200 deve retornar 4,0,0; significando 4 notas de 50.

33 deve retornar -1,-1,-1; significando que com notas de 50,20 e 10 não é possível compor o valor 33.

Eis como ficaria

```
int troco(float valor,int & n50, int & n20, int & n10){
    n50=0;
    n20=0;
    n10=0;
    while (valor>=50){
        n50=n50+1;
        valor=valor-50;
    }
    while (valor>=20){
        n20=n20+1;
        valor=valor-20;
    }
    while (valor>=10){
        n10=n10+1;
        valor=valor-10;
    }
    if (valor !=0) {
        n50=-1;
        n20=-1;
        n10=-1;
    }
}

int main(){
    float t;
    int a,b,c;
    t=200.0;
    troco(t,a,b,c);
    cout<<t<<' '<<a<<' '<<b<<' '<<c<<endl;
}
```

Aqui a resposta será 200 4 0 0 mostrando que as variáveis a, b e c tiveram seus valores corretamente estabelecidos.

Mais um exemplo: Você tem inúmeros terrenos retangulares em uma determinada cidade, Curitiba, por exemplo. Cada terreno é descrito por 2 medidas, a largura e o comprimento. Você gostaria de gerar uma tabela mostrando para cada terreno sua área e seu perímetro. Como você aprendeu recentemente a usar funções, gostaria de criar uma função que achasse esses 2 parâmetros. Eis como ficaria o relatório

```
20,100 ----- area 2000 perimetro 240
100,100 ----- area 10000 perimetro 400
1,400 ----- area 400 perimetro 802
```

Eis como ficaria

```
int areaper(float & larg, float & compr, float & area, float & perim){
    area = larg * compr;
    perim = (2*larg)+(2*compr);
}

int main() {
    float a,b,c,d;
    a=20;
    b=100;
    areaper(a,b,c,d);
    cout<<a<<','<<b<<" ----- area "<<c<<" perimetro "<<d<<endl;
    a=100;
    b=100;
    areaper(a,b,c,d);
}
```

```

cout<<a<<','<<b<<" ----- area "<<c<<" perimetro "<<d<<endl;
a=1;
b=400;
areaper(a,b,c,d);
cout<<a<<','<<b<<" ----- area "<<c<<" perimetro "<<d<<endl;
}

```

Mais um exemplo: Suponha que você ganhou um termômetro que mostra a temperatura ambiente em graus Celsius (centígrados). Você gostaria de disponibilizar outras 2 temperaturas: a absoluta que também é centígrada, mas cujo ponto inicial é diferente da temperatura Celsius e também a temperatura Fahrenheit. Para isso gostaria de criar uma função C++ que recebesse a temperatura Celsius e devolvesse a absoluta e a Fahrenheit. Use as seguintes fórmulas:

$$A = C + 273.15$$

$$F = (C * 1.8) + 32$$

Eis como ficaria:

```

int celfar(float celsius, float & absol, float & faren){
    absol = celsius+273.15;
    faren = (celsius*1.8)+32;
}
int main() {
    float a,f;
    celfar(100,a,f);
    cout<<"Ebulicao: 100 "<<a<< ' '<<f<<endl;
    float c = 37;
    celfar(c,a,f);
    cout<<"Corpo humano: "<<c<< ' '<<a<< ' '<<f<<endl;
}

```

Um exemplo um pouco mais elaborado: Suponha que na sua cidade e no seu país há 2 impostos: o de consumo (federal) que corresponde a 17% do valor da venda, com isenção de valores de venda menores que ou iguais a 100,00 reais. Há também o imposto de serviços (municipal) que é calculado sempre como sendo de 10% ou de 20% do valor bruto da venda a depender do valor da venda: maiores que 1000 reais pagam 20% e menores do que isso pagam 10%. Para calcular o imposto municipal, considera-se o valor bruto da venda, desconsiderando neste caso o imposto federal. Por exemplo, uma venda de 1500 reais, pagará 17% de consumo (porque 1500 > 100) ou 255,00 de imposto federal e também 300 reais de imposto municipal (já que 1500 > 1000). Escreva uma função que receba o valor da venda e devolva os dois impostos: o federal e o municipal.

```

int fedemuni(float valor, float & fede, float & muni){
    fede=0;
    if (valor > 100) {
        fede = valor * 0.17;
    }
    if (valor > 1000) {
        muni = valor*0.2;
    }
    else {
        muni = valor*0.1;
    }
}
int main() {
    float v,f,m;
    v=1500;
    fedemuni(v,f,m);
    cout<<"Venda: "<<v<<" Federal: "<<f<<" municipal: "<<m<<endl;
}

```