**11 Funções**

**11.1 Funções: o que são e por que usá-las**

Quando queremos resolver um problema, em geral tentamos dividi-lo em subproblemas mais simples e relativamente independentes, e resolvemos os problemas mais simples um a um. A linguagem C dispõe de construções (abstrações) que auxiliam o projeto de programas de maneira *top-down*. Uma função cria uma maneira conveniente de encapsular alguns detalhes de ``processamento'', ou seja, como algum resultado é obtido. Quando esta ``computação'' é necessária, a função é *chamada*, ou *invocada*. Desta forma, quando uma função é chamada o usuário não precisa se preocupar como a computação é realizada. É importante saber o que a função faz (qual o resultado da execução de uma função) e também como se usa a função. Criando funções, um programa C pode ser estruturado em partes relativamente independentes que correspondem as subdivisões do problema.

Você já viu algumas funções: printf(), scanf(), getchar(), sqrt(). Elas são funções de uma biblioteca padrão (do C ). Você não sabe como elas foram escritas, mas já viu como utilizá-las. Ou seja, você sabe o *nome* das funções e quais informações específicas você deve fornecer a elas (valores que devem ser *passados* para as funções) para que a função produza os resultados esperados.

Quando nos referirmos a uma função neste texto usaremos a maneira frequentemente utilizada que é o nome da função seguido de ().

Tomemos como exemplo o programa abaixo, que calcula o produto de 2 números inteiros positivos apenas se ambos forem primos:

/usr/local/lib/tex/inputs///Programas/primo\_05.c

Observe que o código que verifica se um número é primo teve que ser reproduzido dentro do programa por duas vezes (para testar se os números fornecidos pelo usuário eram primos).

Um dos benefícios mais óbvios de usar funções é que podemos evitar *repetição de código*. Em outras palavras, se você quiser executar uma operação mais de uma vez, você pode simplesmente escrever a função uma vez e utilizá-la diversas vezes ao invés de escrever o mesmo código várias vezes. Outro benefício é que se você desejar alterar ou corrigir alguma coisa mais tarde, é mais fácil alterar em um único lugar.

O exemplo acima poderia ser simplificado pela criação de uma função chamada ehPrimo, que dado um número $n$, dá como resultado 1 se este número é primo, ou 0 (zero) se o número não é primo:

/usr/local/lib/tex/inputs///Programas/ehPrimo.c

O exemplo pode ser então alterado e simplificado com o uso da função ehPrimo():

/usr/local/lib/tex/inputs///Programas/primo\_05-func.c

Como pode ser observado, sejam quais forem os 2 números fornecidos, não precisa escrever um código similar ao mostrado na função ehPrimo acima para cada número.Basta chamar a função ehPrimo(), passar os valores necessários para verificar a primalidade de cada número, e utilizar os resultados.

Evitar repetição de código é a razão histórica que funções foram inventadas (também chamado de procedimento ou subrotinas em outras linguagens de programação). A maior motivação para utilizar funções nas linguagens contemporâneas é a redução da complexidade do programa e melhoria da modularidade do programa. Dividindo o programa em funções, é muito mais fácil projetar, entender e modificar um programa. Por exemplo, obter a entrada do programa, realizar as computações necessárias e apresentar o resultado ao usuário pode ser implementado como diferentes funções chamadas por main() nesta ordem.

Funções podem ser escritas *independentemente* uma da outra. Isto significa que, em geral, variáveis usadas dentro de funções não são compartilhadas pelas outras funções. Assim sendo, o comportamento da função é previsível. Se não for assim, duas funções completamente não relacionadas podem alterar os dados uma da outra. Se as variáveis são *locais* a uma função, programas grandes passam a ser mais fáceis de serem escritos. A comunicação entre funções passa a ser controlada - elas se comunicam somente através pelos valores passados as funções e os valores retornados.

**11.2 Definindo funções**

Um programa C consiste de uma ou mais definições de funções (e variáveis). Há sempre uma função chamada main. Outras funções também podem ser definidas. Cada uma pode ser definida separadamente, mas nenhuma função pode ser definida dentro de outra função. Abaixo, mostramos um exemplo simples de um programa que consiste de duas funções: main() e alo(). Quando executado, este programa imprimirá a mensage Alo! três vezes.

#include <stdio.h>

/\* declaracao (protótipo) da funcao alo() \*/

void alo(void);

/\* definicao da funcao main() \*/

main()

{

int i;

i = 1;

while (i <= 3)

{

alo();

i += 1;

}

}

/\* definicao da funcao alo() \*/

void alo(void)

{

printf("Alo!\n");

}

Todas as funções devem ser declaradas antes de serem usadas. As funções da biblioteca padrão, tais como printf(), scanf() e getchar(), são pré-definidas, mas memsmo assim devem ser declaradas (deve ser anunciado ao compilador que elas existem). É por isso que incluímos a linha #include <stdio.h> no início do código fonte.

O formato geral da definição de uma função é

*tipo-do-resultado* *nome-da função* (*lista-de-argumentos*)

{

*declarações e sentenças*

}

A primeira linha da definição é o cabeçalho da função. Ela têm três partes principais: o nome da função, o tipo do resultado (que é um valor) que a função computa e retorna, e entre parênteses uma lista de *parâmetros* (também chamado de *argumentos formais*). Se a função não retorna nenhum valor, o tipo é chamado de void, e esta palavra é escrita no cabeçalho na frente do nome da função. Se a função não tiver argumentos formais, a palavra void é escrita no lugar da lista de argumentos formais entre os parênteses. Para simplificar a exposição, falaremos sobre o *tipo do retorno* e os *argumentos formais* mais tarde. Eles servem para permitir que as funções troquem informações entre si.

**11.3 Funções simples**

Para começar, vamos utilizar funções na seguinte forma:

void *nome-da-função*(void)

{

*declarações e senteças (corpo da função)*

}

O primeiro void significa que esta função não tem tipo de retorno (não retorna um valor), e o segundo significa que a função não tem argumentos (ela não precisa de nenhuma informação externa para ser executada). Isso não significa que a função não faz nada. Ela pode realizar alguma ação, como imprimir uma mensagem. O exemplo abaixo mostra um programa que usa uma função como essa:

void alo(void);

main()

{

alo();

}

void alo(void)

{

printf("Alo.\n");

}

Neste exemplo, o programa consiste de duas funções, main() e alo(). A ordem em que as funções são definidas não é importante, desde que *protótipos de funções* são usadas. A linha

void alo(void);

no topo do programa é um *protótipo de função* para a função alo(). Um protótipo é usado para declarar uma função. Um protótipo passa ao compilador informações sobre uma função que é definida dentro de um programa ``em algum lugar''. Protótipos são sempre colocados próximo ao início do programa, antes do começo da definição de funções.

A função alo() imprime a mensagem Alo. quando chamada. A sentença printf() é o corpo da função. Dentro da função main() há uma *chamada* a função alo(). A função é chamada pelo seu nome seguido de () (já que a função alo não tem argumentos, nenhuma expressão é escrita dentro dos parênteses). A função alo() não retorna um valor, ela é chamada simplesmente para realizar uma ação (imprimir a mensagem). A chamada de função é uma sentença válida em C , portanto deve ser terminada por ponto e vírgula (;).

alo();

Outra coisa que você deve ter notado é que main() também é uma função. A função main() não difere em nada das demais funções, com a exceção de que contém o programa principal. Além disso, não é necessário declarar o protótipo da função main().

**11.3.1 Argumentos**

Nosso próximo exemplo pede que o usuário digite suas iniciais, e então chama a função cumprimenta() para imprimir a mensagem ``Ola'' junto com as iniciais digitadas. Estas iniciais (seus valores) são passadas para a função cumprimenta(). A função cumprimenta() é definida de forma que ela imprimirá a mensagem incluindo quaisquer iniciais passadas.

#include <stdio.h>

void cumprimenta(char, char);

int main()

{

char primeiro, segundo;

printf("Entre com duas iniciais (sem separacao): ");

primeiro = getchar();

segundo = getchar();

cumprimenta(primeiro, segundo);

}

void cumprimenta(char inic1, char inic2)

{

printf("Ola, %c%c!\n",inic1,inic2);

}

A função main() chama a função cumprimenta(); main() passa para cumprimenta() os valores dos dois caracteres para serem impressos. Veja um exemplo de execução do programa:

Entre com duas iniciais (sem separacao): YK

Alo, YK!

Note que há uma correspondência entre o número e tipo dos valores que main() passa (estes são chamados de *parâmetros reais* ou *argumentos reais*) e os argumentos listados no cabeçalho da função cumprimenta().

**11.4 Funções que retornam um valor**

Funções que não retornam nenhum valor (como alo(), main()) possuem tipo void.

Além de executarem ações (como imprimir) uma função também pode *retornar* um valor para o programa que o chamou. Uma função que retorna um valor tem no cabeçalho o nome do tipo do resultado. O valor retornado pode ser de qualquer tipo, incluindo int, float e char (é claro que uma vez definida, a função só é de um tipo específico). Uma função que retorna um tipo diferente de void executa alguns cálculos, e retorna o resultado (que é um único valor) para quem a chamou. A função chamadora pode então usar o resultado. Para retornar um valor para a função chamadora, a função usa a sentença return.

O formato da sentença return é a seguinte:

return *expressão*;

A *expressão* é avaliada e o seu valor é convertido ao tipo de retorno da função (o tipo da função é dado no cabeçalho da função antes do nome da função).

Considere o seguinte exemplo. O programa consiste de duas funções: main() e quadrado. O programa pede que o usuário digite três números e verifica se eles podem ser os lados de um triângulo reto.

/\* programa que verifica se 3 numeros podem ser os lados de um

\* triangulo reto.

\*/

#include <stdio.h>

int quadrado(int);

int main()

{

int s1, s2, s3;

printf("Entre tres inteiros: ");

scanf("%d %d %d", &s1, &s2, &s3);

if ( s1 > 0 && s2 > 0 && s3 > 0 &&

(quadrado(s1) + quadrado(s2) == quadrado(s3) ||

quadrado(s2) + quadrado(s3) == quadrado(s1) ||

quadrado(s3) + quadrado(s1) == quadrado(s2)) )

printf(" %d %d %d podem formar um triangulo reto\n", s1, s2, s3);

else

printf(" %d %d %d nao podem formar um triangulo reto\n",s1, s2, s3);

}

/\* funcao que calcula o quadrado de um numero \*/

int quadrado(int n)

{

return n \* n;

}

Note que quando chamamos a função quadrado() passamos o valor no qual desejamos executar o cálculo, e também usamos o valor retornado pela função em expressões. O valor de quadrado(s1) é o valor que a função quadrado() retorna quando chamado com o valor do argumento sendo igual ao valor da variável s1.

Os valores retornados pelas chamadas de funções podem ser usados em todos os lugares valores podem ser usados. Por exemplo,

y = quadrado(3);  
Aqui quadrado(3) tem o valor 9, portanto 9 pode ser atribuído a variável y;

x = quadrado(3) + quadrado(4);  
atribuirá 25 a variável x, e

area = quadrado(tamanho);  
atribuirá a variável area o valor da variável tamanho elevado ao quadrado.

O próximo exemplo tem uma função chamada cinco:

int cinco(void);

main()

{

printf("cinco = %d\n", cinco() );

}

int cinco(void)

{

return 5;

}

A saída do programa será

cinco = 5

porque o valor de cinco() dentro da sentença printf() é 5. Olhando na sentença return, 5 é a expressão retornada para o chamador.

Outro exemplo:

int obtem\_valor(void);

main()

{

int a, b;

a = obtem\_valor();

b = obtem\_valor();

printf("soma = %d\n", a + b);

}

int obtem\_valor(void)

{

int valor;

printf("Entre um valor: ");

scanf("%d", &valor);

return valor;

}

Este programa obtém dois inteiros do usuário e mostra a sua soma. Ele usa a função obtem\undvalor() que mostra uma mensagem e obtém o valor do usuário.

Um exemplo de saída deste programa é:

Entre um valor: 15

Entre um valor: 4

soma = 19

**11.5 Mais sobre o return**

Quando uma função return é executada, a função imediatamente acaba - mesmo que haja código na função após a sentença return. A execução do programa continua após o ponto no qual a chamada de função foi feita. Sentenças return podem ocorrer em qualquer lugar na função - não somente no final. Também é válido ter mais de um return dentro de uma função. A única limitação é que return retorna um único valor.

O seguinte exemplo mostra uma função (uma versão para int da função obtem\undvalor) que pede para usuário um valor e se o usuário digitar um valor negativo, imprime uma mensagem e retorna um valor positivo.

int obtem\_valor\_positivo(void)

{

int valor;

printf("Entre um valor: ");

scanf("%d", &valor);

if (valor >= 0)

return valor;

printf("Tornando o valor positivo...\n");

return -valor;

}

Em uma função void, return; (só com ;) pode ser usado para sair de uma função. O exemplo seguinte, pede instruções ao usuário. Se o usuário reponder **nao**, a função termina. Do contrário, ele imprime as instruções e depois termina.

void instrucoes(void)

{

int ch;

printf("Voce quer instrucos? (s/n): ");

ch = getchar();

/\* Termina se resposta for n \*/

if (ch == 'n' || ch == 'N')

return;

/\* Mostra instrucoes \*/

printf("As regras do jogo sao . . . ");

.

.

.

return;

}

O return final (antes de fechar as chaves do corpo da função) na função é opcional. Se omitido, a função atingirá o final da função e retornará automaticamente. Note que o return é opcional somente para funções void.

**11.6 Mais sobre Argumentos**

A comunicação entre uma função e o chamador pode ser nas duas direções. Argumentos podem ser usados pelo chamador para passar dados para a função. A lista de argumentos é definida pelo cabeçalho da função entre parênteses.. Para cada argumento você precisa especificar o tipo do argumento e o nome do argumento. Se houver mais de um argumento, eles são separados por vírgula. Funções que não possuem argumentos tem void como lista de argumento. No corpo da função os argumentos (também chamados de *argumentos formais* ou *parâmetros formais*) são tratados como variáveis. É erro defini-los dentro do corpo da função porque eles já estão definidos no cabeçalho. Antes da execução da função os valores passados pelo chamador são atribuídos aos argumentos da função.

Considere o seguinte programa com a função abs() que calcula o valor absoluto de um número.

int abs(int);

main()

{

int n;

printf("Entre um numero: ");

scanf("%d", &n);

printf("Valor absoluto de %d e' %d", n, abs(n));

}

/\* Definicao da funcao abs \*/

int abs(int x)

{

if (x < 0)

x = -x;

return x;

}

A função abs() tem um argumento do tipo int, e seu nome é x. Dentro da função, x é usado como uma variável x.

Uma vez que abs() tem um único argumento, quando ela é chamada, há sempre um valor dentro do parênteses, como em abs(n). O valor de n é passado para a função abs(), e antes da execução da função, o valor de n é atribuído a x.

Aqui está um exemplo de uma função que converte uma temperatura de Farenheit para Celsius:

float fahr\_para\_cels(float f)

{

return 5.0 / 9.0 \* (f - 32.0);

}

Como você pode ver, esta função tem somente um argumento do tipo float. Um exemplo de chamada desta função poderia ser:

fervura = fahr\_para\_cels(212.0);

O resultado da função fahr\undpara\undcels(212.0) é atribuído a fervura. Portanto, depois da execução desta sentença, o valor de fervura (que é do tipo float) será 100.0.

O exemplo seguinte possui mais de um argumento:

float area(float largura, float altura)

{

return largura \* altura;

}

Esta função possui dois argumentos do tipo float. Para chamar uma função com mais de um argumento, os argumentos devem ser separados por vírgula. A ordem em que os argumentos são passados deve ser na mesma em que são definidos. Neste exemplo, o primeiro valor passado será a largura e o segundo a altura. Um exemplo de chamada seria

tamanho = area(14.0, 21.5);

Depois desta sentença, o valor de tamanho (que é do tipo float) será 301.0.

Quando passar os argumentos, é importante ter certeza de passá-los na ordem correta e que eles são do tipo correto. Se isso não for observado, pode ocorrer erro ou aviso de compilação, ou resultados incorretos podem ser gerados.

Uma última observação. Os argumentos que são passados pelo chamador podem ser expressões em geral e não somente constantes e variávies. Quando a função é chamada durante a execução do programa, estas expressões são avaliadas, e o valor resultante passado para a função chamada.

**11.7 Chamada por valor**

Considere novamente a função quadrado(). Se esta função é chamada de main() como

p = quadrado(x);

somente o valor (não o endereço) de x é passado para quadrado. Por exemplo, se a variável tem valor 5, para a função quadrado(), quadrado(x) ou quadrado(5) são o mesmo. De qualquer forma, quadrado() receberá somente o valor 5. quadrado() não sabe se na chamada da função o 5 era uma constante inteira, o valor de uma variável do tipon int, ou alguma expressão como 625/25 - 4 \* 5. Quando quadrado() é chamado, não interessa qual a expressão entre parênteses, ela será avaliada e o valor passado para quadrado().

Esta maneira de passar argumentos é chamada de *chamada por valor*. Argumentos em C são passados por valor. Portanto, a função chamada não pode alterar o valor da variável passada pelo chamador como argumento, porque ela não sabe em que endereço de memória o valor da variável está armazenado.

**11.8 Variáveis locais**

Como você provavelmente já reparou em alguns exemplos, é possível definir variáveis dentro de funções, da mesma forma que temos definido variáveis dentro da função main(). A declaração de variáveis é feita no início da função.

Estas variáveis são *restritas* a função dentro da qual elas são definidas. Só esta função pode ``enxergar'' suas próprias variáveis. Por exemplo:

void obtem\_int(void);

main()

{

obtem\_int();

/\* \*\*\*\* Isto esta' errado \*\*\*\* \*/

printf("Voce digitou %d\n", x);

}

void obtem\_int(void)

{

int x;

printf("Entre um valor: ");

scanf("%d", &x);

printf("Obrigado!\n");

}

A função main() usou um nome x, mas x não é definido dentro de main; ele é uma variável local a get\undint(), não a main(). Este programa gera erro de compilação.

Note que é possível ter duas funções que usam variáveis locais com o mesmo nome. Cada uma delas é restrita a função que a define e não há conflito. Analise o seguinte programa (ele está correto):

int obtem\_novo\_int(void);

main()

{

int x;

x = obtem\_novo\_int();

/\* \*\*\*\*Isto nao esta errado !! \*\*\*\* \*/

printf("Voce digitou %d\n", x);

}

int obtem\_novo\_int(void)

{

int x;

printf("Entre um valor: ");

scanf("%d", &x);

printf("Obrigado!\n");

return x;

}

A função obtem\undnovo\undint() usa uma variável local chamada x para armazenar o valor digitado e retorna como resultado o valor de x. main() usa outra variável local, também chamada de x para receber o resultado retornado por obtem\undnovo\undint(). Cada função tem sua própria variável x.

**11.9 Protótipos**

Os protótipos servem para dar ao compilador informações sobre as funções. Isso para que você possa chamar funções antes que o compilador tenha a definição (completa) das funções. O protótipo de uma função é idêntico ao cabeçalho da função, mas o nome dos argumentos podem ser omitidos e ele é terminado com uma vírgula. Protótipos *declaram* uma função ao invés de *defini-las*. O formato dos protótipos é:

*tipo-de-retorno* *nome-da-função*(*lista-dos-tipos-dos-argumentos*);

Definindo protótipos, você não precisa se preocupar com a ordem em que define as funções dentro do programa. A principal vantagem de definir protótipos é que erros de chamada de funções (como chamar uma função com o número incorreto de argumentos, ou com argumentos de tipo errado) são detectados pelo compilador. Sem protótipos, o compilador só saberia que há erro depois de encontrar a definição da função. Em versões antigas do compilador C , programas com tais erros compilariam sem erros, o que tornava a depuração de erros mais difícil.

Abaixo, mostramos duas funções e seus protótipos:

float volume(float, float, float);

float dinheiro(int, int, int, int);

float volume(float comprimento, float largura, float altura)

{

return comprimento \* largura \* altura;

}

float dinheiro(int c25, int c10, int c5, int c1)

{

return c25 \* 0.25 + c10 \* 0.10 +

c5 \* 0.05 + c1 \* 0.01;

}

**11.10 Documentação de funções**

Você deve documentar as funções que escreve. Na documentação você deve especificar as seguintes informações:

**Ação** - o que a função faz

**Entrada** - descrição dos argumentos passados para a função

**Saída** - descrição do valor retornado pela função

**Suposições** - o que você assume ser verdade para que a função funcione apropriadamente

**Algoritmo** - como o problema é resolvido (método)

Estas informações devem ser colocadas como comentário antes da definição da função.

**11.11 Comentários**

Você pode colocar comentários no seu programa para documentar o que está fazendo. O compilador ignora completamente o que quer esteja dentro de um comentário.

Comentários em C começam com um /\* e terminam com um \*/. Alguns exemplos:

/\* Este e' um comentario sem graca \*/

/\* Este e'

um comentario

que usa

diversas linhas

\*/

/\* Este e'

\* um comentario

\* de diversas linhas

\* mais bonito

\*/

Note que não podemos aninhar comentários dentro de comentários. Um comentário termina no primeiro \*/ que encontrar. O comentário abaixo é ilegal:

/\* Este e' um comentario /\* illegal \*/ ilegal \*/

**Regras para comentário**

É sempre uma boa idéia colocar comentários em seu programa das coisas que não são claras. Isto vai ajudar quando mais tarde você olhar o programa que escreveu já há algum tempo ou vai ajudar a entender programas escritos por outra pessoa.

Um exemplo de comentário útil:

/\* converte temperatura de farenheit para celsius \*/

celsius = (fahrenheit - 32) \* 5.0 / 9.0;

O comentário deve ser escrito em português e não em C . No exemplo abaixo

/\* usando scanf, obter valor de idade e multiplicar por 365 para

\* obter dias \*/

scanf("%d", &idade);

dias = idade \* 365;

o comentário é basicamente uma transcrição do código do programa. Em seu lugar, um comentário como

/\* obtem idade e transforma em numero de dias \*/

seria mais informativo neste ponto. Ou seja, você deve comentar o código, e não codificar o comentário.

Você também deve evitar comentários inúteis. Por exemplo:

/\* Incrementa i \*/

i++;

Não há necessidade de comentários já que i++ já é auto explicativo.

E abaixo está um exemplo de como você deve comentar uma função.

/\* funcao instrucoes()

\* acao: mostra instrucoes do programa

\* entrada: nenhuma

\* saida: nenhuma

\* suposicoes: nenhuma

\* algoritmo: imprime as instrucoes

\*/

void instrucoes(void)

{

/\* mostra instrucoes \*/

printf("O processo de purificacao do Uranio-235 e' . . . ");

.

.

}