CAPÍTULO 4

FUNÇÕES

4.1 Definição de Função

Funções servem para dividir um grande programa em diversas partes menores. Além disso, permitem que sejam utilizadas partes de programa desenvolvidas por outras pessoas, sem que se tenha acesso ao código-fonte. Como exemplo, em capítulos anteriores foi utilizada a função printf() sem que fossem conhecidos detalhes de sua programação.

Programas em C geralmente utilizam diversas pequenas funções, ao invés de poucas e grandes funções. Ao dividir um programa em funções, diversas vantagens são encontradas, como impedir que o programador tenha que repetir o código diversas vezes, facilitar o trabalho de encontrar erros no código.

Diversas funções são fornecidas juntamente com os compiladores, e estão presentes na norma ANSI, como funções matemáticas (seno, coseno, potência, raiz quadrada etc.), funções de entrada e saída (scanf(), printf() e outras), entre outras.

Uma função tem a sintaxe

```
tipo nome(argumentos)
{
    declarações de variáveis
    comandos
}
```

onde:

tipo determina o tipo do valor de retorno (se omitido, é assumido int);
nome representa o nome pelo qual a função será chamada ao longo do programa;
argumentos são informações externas transmitidas para a função (podem não existir).

Todo programa é composto de funções, sendo iniciada a execução pela função de nome main().

4.1.1 Variáveis Locais

A declaração das variáveis, em C, deve vir no início da função, antes de qualquer comando. Uma variável declarada dentro do corpo de uma função é local, ou seja, só existe dentro da função. Ao ser iniciada a função, a variável é criada. Quando a função termina, a variável é apagada, sendo liberado seu espaço ocupado na memória.

4.1.2 Chamando Funções

Para executar uma função, ela deve ser chamada no corpo de uma outra função (à exceção da função main(), que é executada no início do programa). Uma chamada de função é feita escrevendo-se o nome da função seguido dos argumentos fornecidos, entre parênteses. Se não houver argumentos, ainda assim devem ser mantidos os parênteses, para que o compilador diferencie a chamada da função de uma variável. O comando de chamada de uma função deve ser seguido de ponto-e-vírgula.

As funções só podem ser chamadas depois de terem sido declaradas. Caso sejam chamadas sem que tenham sido declaradas, um erro de compilação ocorre.

4.1.3 Programa Exemplo

O programa abaixo é composto de duas funções: main() e linha().

```
linha()
{
    printf("------------------------------\n");
}

main()
{
    linha();
    printf("Programa exemplo de funcoes \n");
    linha();
}
```

```
SAIDA
------------------------------
Programa exemplo de funcoes
------------------------------
```

A função linha(), apresentada neste exemplo, escreve uma linha na tela. Esta função chama uma outra função, printf(), da biblioteca C. Uma função pode conter em seu corpo chamadas a outras funções.

A função main() apresenta duas chamadas à função linha().

4.2 Argumentos

Argumentos são utilizados para transmitir informações para a função. Já foram utilizados anteriormente nas funções printf() e scanf().

Uma função pode receber qualquer número de argumentos, sendo possível escrever uma função que não receba nenhum argumento. No caso de uma função sem argumentos pode-se escrevê-la de duas formas: deixando a lista de argumentos vazia (mantendo entretanto os parênteses) ou colocando o tipo *void* entre parênteses.

O quinto tipo existente em C, *void* (vazio, em inglês), é um tipo utilizado para representar o nada. Nenhuma variável pode ser declarada como sendo do tipo void. A função main(), já utilizada em capítulos anteriores, é um exemplo de função sem argumentos.

Exemplo: o programa abaixo utiliza a função EscreveCaractere(). Esta função recebe como argumento uma variável caractere (ch) e uma variável inteira (n) e faz com que o caractere *ch* seja impresso *n* vezes.

```
EscreveCaractere(char ch, int n)
{
    int i;

    for(i = 0; i < n; i++)
    {
        printf("%c", ch);
    }
}
```

```
main()
{
    EscreveCaractere('-', 27);
    printf("\nPrograma exemplo de funcoes\n");
    EscreveCaractere('-', 27);
    EscreveCaractere('\n', 3);
    printf("Teste concluido\n");
}
```

Nota-se que no exemplo inicialmente é definida a função EscreveCaractere() para, somente depois, ser definida a função main(), que acessa a função EscreveCaractere(). Caso a função main() venha primeiro, quando o compilador tentar compilar a linha que chama a função EscreveCaractere(), o compilador não reconhecerá a função.

Caso deseje-se definir a função EscreveCaractere() antes da função main(), deve-se inicialmente declarar a função EscreveCaractere(). A declaração de uma função consiste em escrevê-la da mesma forma que na definição, sem o corpo da função e seguida por ponto-e-vírgula.

O exemplo anterior ficaria da seguinte maneira:

```
EscreveCaractere(char ch, int n);

main()
{
    EscreveCaractere('-', 27);
    printf("\nPrograma exemplo de funcoes\n");
    EscreveCaractere('-', 27);
    EscreveCaractere('\n', 3);
    printf("Teste concluido\n");
}

EscreveCaractere(char ch, int n)
{
    int i;

    for(i = 0; i < n; i++)
        printf("%c", ch);
}
```

Quando o compilador encontra a primeira linha do código, ele entende que a função EscreveCaractere() existe e tem a forma apresentada na declaração, mas ainda não está definida; será definida em algum lugar do código. Portanto, ele consegue compilar as chamadas à esta função no resto do programa.

4.3 Valor de Retorno

Valor de retorno é o valor que uma função retorna para a função que a chamou. Seu tipo é fornecido antes do nome da função na declaração.

Exemplo: a função Quadrado() recebe como argumento uma variável real (a) e retorna o quadrado dela.

```
float Quadrado(float a)
{
    return a * a;
}
```

Na função, são fornecidos o tipo do valor de retorno (float), o nome da função (Quadrado) e o argumento (a, do tipo float). No corpo da função, é encontrado o comando **return**. Este comando fornece o valor de retorno da função, fazendo com que ela termine. Uma função pode ter diversos comando **return**.

Exemplo: a função Maximo() retorna o maior valor entre dois números.

```
int Maximo(int a, int b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

Esta outra função possui dois argumentos inteiros (*a* e *b*) sendo o valor de retorno também inteiro. Este é um exemplo de função que possui mais de um comando **return**.

A função EscreveCaractere(), do exemplo fornecido na seção 4.2, é um exemplo de função que não possui retorno. No caso, se for omitido o valor de retorno de uma função, este valor é assumido como int. Se uma função não retorna nada, seu tipo de retorno deve ser definido como **void**.

O comando **return** pode ser utilizado numa função com tipo de retorno **void**. neste caso, o comando não deve retornar nenhum valor, sendo chamado simplesmente seguido do ponto-e-vírgula.

4.4 Recursividade

Como foi visto nas seções anteriores, uma função pode conter em seu corpo chamadas a outras funções. Nesta seção veremos um caso particular em que uma função é chamada por ela própria. A este caso, dá-se o nome de *recursividade*.

Exemplo: Função fatorial(), utilizando recursividade.

Sabe-se que N! = N * (N - 1)! e que 0! = 1.

```
int fatorial(int n)
{
    if(n > 0)
        return n * fatorial(n - 1);
    else
        return 1;
}
```

Note que existe dentro da função recursiva uma possibilidade de o programa sair dela sem que ocorra a chamada à própria função (no caso de n menor ou igual a zero a função retorna 1). Se não existisse esta condição, a função se chamaria infinitamente, causando o travamento do programa.

4.5 Classes de Armazenamento

Todas as variáveis e funções C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos, como visto anteriormente, são cinco: int, long, float, double e void. As classes de armazenamento são quatro: auto (automáticas), extern (externas), static (estáticas) e register (em registradores).

4.5.1 Classe de Armazenamento - auto

As variáveis declaradas até agora nos exemplos são acessíveis somente na função a qual ela pertence. Estas variáveis são automáticas caso não seja definida sua classe de armazenamento. Ou seja, o código

```
main()
{
    auto int n;
    . . .
}
```

é equivalente a

```
main()
{
    int n;
    . . .
}
```

4.5.2 Classe de Armazenamento - extern

Todas as variáveis declaradas fora de qualquer função são externas. Variáveis com este atributo serão conhecidas por todas as funções declaradas depois delas.

```
/* Testa o uso de variaveis externas */
int i;

void incrementa(void)
{
    i++;
}

main()
{
    printf("%d\n", i);
    incrementa();
    printf("%d\n", i);
}
```

4.5.3 Classe de Armazenamento - static

Variáveis estáticas possuem dois comportamentos bem distintos: variáveis estáticas locais e estáticas externas.

Quando uma variável estática é declarada dentro de uma função, ela é local àquela função. Uma variável estática mantém seu valor ao término da função. O exemplo a seguir apresenta uma função contendo uma variável estática.

```
void soma(void)
{
    static int i = 0;
    printf("i = %d\n", i++);
}

main()
{
    soma();
    soma();
    soma();
}
```

A saída será:

```
i = 0
i = 1
i = 2
```

Como a variável i é estática, não é inicializada a cada chamada de soma().

O segundo uso de **static** é associado a declarações externas. Declarar uma variável estática externa indica que esta variável será global apenas no arquivo fonte no qual ela foi declarada, e só a aprtir de sua declaração. Esta declaração é usada como mecanismo de privacidade e impede que outros arquivos tenham acesso à variavél.

4.5.4 Classe de Armazenamento - register

A classe de armazenamento **register** indica que a variável será guardada fisicamente numa memória de acesso ,uito mais rápido chamada registrador. Somente podem ser armazenadas no registrador variáveis do tipo **int** e **char**. Basicamente, variáveis **register** são usadas para aumentar a velocidade de processamento.

4.6 O Pré-processador C

O pré-processador C é um programa que examina o programa-fonte em C e executa certas modificações nela, baseado em instruções chamadas diretivas. É executado automaticamente antes da compilação.

4.6.1 A Diretiva #define

A diretiva **#define** substitui determinados códigos predefinidos pelo usuário. Possui a forma

```
#define nome xxxxx
```

onde *nome* é o símbolo a ser trocado e *xxxxx* é qualquer expressão. O comando pode continuar em mais de uma linha utilizando-se a barra invertida (\) para indicar a continuação na linha de baixo.

Ao ser criado um **#define**, o precompilador substitui todas as ocorrências de *nome* no código fonte pela expressão *xxxxx* fornecida.

Exemplo: a seguir definimos algumas constantes simbólicas.

```
#define PI 3.14159
#define ON 1
#define OFF 0
#define ENDERECO 0x378

void main()
{
...
}
```

No exemplo acima, definimos PI como 3.14159. Isto significa que em todas as ocorrências do texto, PI será trocado por 3.14159. Assim, a instrução

```
area = PI * raio * raio;
```

será interpretada pelo compilador como se fosse escrita assim:

```
area = 3.14159 * raio * raio;
```

Não é utilizado ponto-e-vírgula no comando #define. Se for utilizado, o precompilador incluirá o ponto-e-vírgula na substituição.

#define pode ter argumentos. Eles são fornecidos imediatamente após o nome:

```
#define nome(arg1,arg2,...) xxxxx
```

Dado:

```
#define MAIS_UM(x) x+1
```

então:

```
MAIS_UM(y)
```

será substituído por:

```
y+1
```

#defines podem referenciar **#define**s anteriores. Por exemplo:

```
#define PI    3.141592653589793
#define PI_2  2*PI
```

Um uso típico de **#define** e substituir algumas chamadas a funções por código direto. Vamos supor, por exemplo, que quiséssemos criar uma macro equivalente a uma função que multiplique dois números. A macro

```
#define PRODUTO(a,b) a*b
```

parece atender ao problema. Vamos testar:

Expressão original	**Expressão pré-compilada**
resposta = PRODUTO(a,b);	resposta = a*b;
resposta = PRODUTO(x,2);	resposta = x*2;
resposta = PRODUTO(a+b,2);	resposta = a+b*2;

Para as duas primeiras expressões, a macro funcionou bem. Já para a terceira, a pricípio desejaríamos o produto de a + b por 2. A expressão pré-compilada não fornece isso, já que o operador * é processado antes do operador +. A solução para evitar este problema é por os argumentos na definição entre parênteses, ou seja:

```
#define PRODUTO(a,b) (a)*(b)
```

Novamente, parece atender ao problema. Vamos testar:

Expressão original	**Expressão pré-compilada**
resposta = PRODUTO(a,b);	resposta = (a)*(b);
resposta = PRODUTO(x,2);	resposta = (x)*(2);
resposta = PRODUTO(a+b,2);	resposta = (a+b)*(2);
resposta = PRODUTO(2,4) / PRODUTO(2,4);	resposta = (2)*(4)/(2)*(4);

Esta macro continua atendendo às duas primeiras expressões, passa a atender à terceira. Entretanto, para a quarta expressão, não obtivemos o resultado desejado. A princípio, desejaríamos efetuar os dois produtos, que são idênticos, dividir um pelo outro, obtendo como resposta 1`. Ao utilizar a macro, como os operadores * e / possuem a mesma precedência, eles são executados na ordem em que são encontrados. Ou seja, a divisão é feita antes do segundo produto ter sido efetuado, o que fornece o resultado 16 para a expressão. A solução para este problema é ser redundante no uso de perênteses ao se construir macros. A macro PRODUTO é definida então por

```
#define PRODUTO(a,b) ((a)*(b))
```

Ainda assim, colocar os parênteses em macros não atende a todos os casos. Por exemplo, a macro

```
#define DOBRO(a) ((a)+(a))
```

apesar de conter todos os parênteses necessários, se for utilizada na expressão

```
resposta = DOBRO(++x)
```

criará o código pré-compilado

```
resposta = (++x)+(++x)
```

que incrementará a variável x duas vezes, ao invés de uma.

Ou seja, **MACROS NÃO SÃO FUNÇÕES**. Tenha sempre cuidado em sua utilização.

4.6.2 A Diretiva #undef

A diretiva **#undef** tem a seguinte sintaxe:

```
#undef nome
```

Esta diretiva anula uma definição de **#define**, o que torna *nome* indefinido.

4.6.3 A Diretiva #include

A instrução **#include** inclui outro arquivo fonte dentro do arquivo atual. Possui duas formas:

```
#include "nome-do-arquivo"
#include <nome-do-arquivo>
```

A primeira inclui um arquivo que será procurado no diretório corrente de trabalho. A segunda incluirá um arquivo que se encontra em algum lugar predefinido.

Os arquivos incluídos normalmente possuem **#define**s e declarações de funções. Os compiladores fornecem uma série de funções que são definidas em arquivos de cabeçalho, com extensão .h, que posem ser incluídos no código-fonte escrito pelo programador. São alguns exemplos a biblioteca matemática *math.h* e a biblioteca de entrada e saída padrão *stdio.h*.

4.6.4 Outras Diretivas

O preprocessador C possui ainda outras diretivas de pré-compilação. Alguns trechos do código podem ser ignorados pelo compilador, dependendo de alguns parâmetros de pré-compilação.

#if *expr*	testa para verificar se a expressão com constantes inteiras *exp* é diferente de zero
#ifdef *nome*	testa a ocorrência de *nome* com **#define**
#ifndef *nome*	testa se *nome* não está definido (nunca foi definido com **#define** ou foi eliminado com **#undef**)

Se os resultados destes testes forem avaliados como verdadeiros, então as linhas seguintes são compiladas até que um **#else** ou um **#endif** seja encontrado.

#else	sinaliza o começo das linhas a serem compiladas se o teste do **if** for falso
#endif	finaliza os comandos **#if**, **#ifdef** ou **#ifndef**

Exemplo: a seqüência seguinte permite que diferentes códigos sejam compilados, baseado na definição de COMPILADOR.

```
#define COMPILADOR TC
  /* Esta definicao precisa ser trocada para cada compilador*/

#if COMPILADOR == TC
  /* Codigo para o Turbo C
#endif

#if COMPILADOR == GCC
  /* Codigo para o Gnu C
#endif
```