4. FUNÇÕES

Uma função consiste num conjunto de comandos agrupados em um bloco que recebe um nome e através deste pode ser ativado. Em C não se usa a nomenclatura de "procedimento", mesmo que a função não retorne nada. Funções são uma das características mais importantes de C [SCH 96].

Funções são utilizadas para:

- permitir o reaproveitamento de código (por você ou por outros programadores);
- evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa;
- permitir a alteração de um trecho de código de uma forma mais rápida, pois com o uso de uma função é preciso alterar o código apenas dentro da função;
- facilitar a leitura de um programa, pois com elas os blocos do programa não ficam grandes demais, o que normalmente dificulta o entendimento;
- separar o programa em partes(blocos) que possam ser logicamente compreendidos de forma isolada [PIN 00].

A forma geral de uma função é:

```
<especificador_de_tipo_da_funcao>  <nome_da_funcao> (<lista_de_parâmetros>)
{
      // corpo da função
}
```

Se nenhum tipo é especificado, o compilador assume que a função devolve um resultado inteiro. A lista de parâmetros é uma lista de nomes de variáveis separados por vírgulas e seus tipos associados que recebem os valores dos argumentos quando a função é chamada. Torna-se importante ressaltar que os parênteses ao lado do nome da função são obrigatórios mesmo que a lista de parâmetros esteja vazia. A forma geral para lista de parâmetros é:

```
void f (<tipo> <nomevar1>, <tipo> <nomevar2>, ..., <tipo> <nomevarn> )
```

Um valor deve ser retornado de uma função que não é declarada como *void*, da mesma maneira que um valor não pode ser retornado de uma função *void*. O valor de retorno é especificado pelo comando *return*. Por exemplo:

```
int f1() { }              //erro: nenhum valor é retornado
void f2() { }             // correto
int f3() { return 1; }    // correto
void f4() { return 1; }   // erro: valor é retornado em uma função void
int f5() { return; }      //erro: falta valor de retorno
void f6() { return;  }    // correto
```

É interessante comentar que pode haver mais de um comando *return* em uma função, como ilustra o próximo exemplo, que apresenta uma função recursiva (função que chama a si mesma).

```
int fat (int n) {
      if (n>1) return n*fac2(n-1);
      return 1;
}
```

4.1. Funções Pré-Definidas (Bibliotecas Padrão)

Depois que as funções são escritas é possível:

1. Deixá-las no mesmo arquivo da função *main*
   Neste caso, deve-se cuidar apenas para o arquivo não ficar muito grande. Não há uma regra, mas o ideal é que o arquivo-fonte não seja maior do que 10000 ou 15000 *bytes*.

2. Colocá-las num arquivo separado
   Neste caso, para facilitar o trabalho, deve-se organizar quais funções serão armazenadas em quais arquivos. É interessante que todas as funções que estão conceitualmente relacionadas fiquem num mesmo arquivo.

3. Colocá-las em uma biblioteca
   Tecnicamente uma biblioteca de funções é diferente de um arquivo de funções compilado separadamente. Quando as rotinas em uma biblioteca são linkeditadas com o restante do programa, apenas as funções que o programa realmente usa são carregadas e linkeditadas. Em um arquivo compilado separadamente, todas as funções são carregadas e linkeditadas com o programa.

Para a maioria dos arquivos que cria, você provavelmente estará interessado em ter todas as funções no arquivo. No caso de uma biblioteca C padrão, você nunca iria querer todas as funções linkeditadas com seu programa, porque o código-objeto seria enorme.

Todo compilador C vem com uma biblioteca C padrão de funções que realizam as tarefas necessárias mais comuns. O padrão C ANSI especifica o conjunto mínimo de funções que estará contido na biblioteca. No entanto, cada compilador provavelmente conterá muitas outras funções. Por exemplo, o padrão C ANSI não define nenhuma função gráfica, mas os compiladores em geral incluem alguma [SCH 96].

A maioria das funções de propósito geral que são usadas no desenvolvimento de um programa já foram escritas e estão na biblioteca padrão. Cada arquivo-fonte que faz referência a uma função da biblioteca padrão, por exemplo, deve conter a linha
#include <stdio.h>
no seu início. O arquivo *stdio.h* define certas macros e variáveis usadas pela biblioteca de entrada e saída. O uso dos sinais < e > ao invés de aspas, dirigem o compilador a pesquisar o arquivo *stdio.h* num diretório contendo informações padrão (tipicamente diretório "*include*") [KER 88].

4.2. Funções de E/S

Nesta seção estão descritas as funções de E/S através do console (entrada pelo teclado, saída pela tela).

- ***putchar()***
  - Escreve um caracter na tela a partir da posição atual do cursor.
  - Protótipo: *int putchar(int c);*
  - Apesar de ser declarada como pegando um parâmetro inteiro, geralmente é chamada usando um caractere (variável do tipo *char* ou 'c').
  - Valor retornado é o caracter, ou EOF se ocorreu erro (macro definida em *stdio.h* cujo valor é geralmente -1).

- ***puts()***
  - Escreve uma seqüência de caracteres (=*string*) até que seja pressionado *enter*.
  - Protótipo: *int puts(const char *s);*

- ***getchar()***
  - Espera uma tecla ser pressionada e devolve seu valor. Tecla pressionada é mostrada na tela.
  - Protótipo: *int getchar(void);*
  - Valor retornado usualmente é armazenado numa variável do tipo *char*. Se ocorreu erro retorna EOF.

- ***getch()***
  - Espera até que uma tecla seja pressionada e não mostra o caracter na tela.
  - Protótipo: *int getch(void);*

- ***gets()***
  - Lê uma seqüência de caracteres (=*string*) até que seja pressionado *enter*.
  - Protótipo: *char gets(char *s);*

- ***printf()***
  - Realiza saída formatada.
  - Protótipo: *int printf(const char *string_de_controle, ...);*
  - Retorna o número de caracteres escritos ou um valor negativo se ocorrer um erro.
  - *string_de_controle* consiste em dois tipos. O primeiro tipo é formado por caracteres que serão exibidos na tela. O segundo tipo contém comandos de formato que definem a maneira pela qual os argumentos subseqüentes serão mostrados.
  - Comando de formato começa por % e é seguido pelo código de formato.
  - Principais comandos de formato:
    %c - caracter
    %d - números inteiros decimais
    %f - números reais do tipo *float*
    %lf - números reais do tipo *double*
    %s - seqüência de caracteres (=*string*)
  - Comandos de formato podem ser alterados, por exemplo, para especificação da largura mínima de campo ou do número de casas decimais. Por exemplo, *printf("%.2f \n", 45.6231)* produz como saída 45.62.

- ***scanf()***
  - Rotina de entrada.
  - Lê todos os tipos de dados (inverso de *printf*).
  - Protótipo: *int scanf(const char *string_de_controle, ...);*
  - Retorna o número de caracteres que foi atribuído ou EOF se ocorrer um erro.
  - *string_de_controle* determina como os valores são lidos para as variáveis apontadas na lista de argumentos.

Torna-se importante comentar que C possui constantes especiais de caractere de barra invertida para caracteres que não podem ser imprimidos, como por exemplo o retorno de carro. Os principais deles são [SCH 96]:

Código	Significado
\f	Salto de Página (*Form Feed*)
\n	Linha Nova (*Line Feed*)
\r	Retorno do Carro (cr)
\t	Tabulação Horizontal (TAB)

O próximo exemplo lê caracteres do teclado e inverte-os de maiúsculas para minúsculas e vice-versa. Para parar o programa deve-se digitar um ponto.

```
#include <stdio.h>
#include <ctype.h>
void main(void)
{
        char ch;
        printf("Entre com algum texto (digite um ponto para sair).\n");
        do
        {
                ch = getchar();
                if(islower(ch)) ch = toupper(ch);
                else ch = tolower(ch);
                putchar(ch);
        } while (ch!='.');
}
```

A utilização das funções *printf* e *scanf* está exemplificada abaixo.

```
#include <stdio.h>
main()
{
        int idade;
        char nome[30];
        printf("Digite sua Idade: ");
        scanf("%d",&idade);
        printf("Seu Nome: ");
        scanf("%s",nome);    /* Strings não utiliza '&' na leitura */
        printf("%s Sua idade e' %d anos. \n", nome, idade);
}
```

4.3. Passagem de parâmetro

A fim de tornar mais amplo o uso de uma função, a linguagem C permite o uso de parâmetros. Este parâmetros possibilitam definir sobre quais dados a função deve operar. Os parâmetros da função na sua declaração são chamados parâmetros formais. Na chamada da função os parâmetros são chamados parâmetros atuais. Para definir os parâmetros de uma função o programador deve explicitá-los como se estive declarando uma variável, entre os parênteses do cabeçalho da função. Caso precise declarar mais de um parâmetro, basta separá-los por vírgulas.

Os parâmetros são passados para uma função de acordo com a sua posição. Ou seja, o primeiro parâmetro atual (da chamada) define o valor o primeiro parâmetro formal (na definição da função, o segundo parâmetro atual define o valor do segundo parâmetro formal e assim por diante. Os nomes dos parâmetros na chamada não tem relação com os nomes dos parâmetros na definição da função [PIN 00].

Os parâmetros formais de uma função correspondem a variáveis que se comportam como quaisquer outras variáveis locais dentro da função (são criadas na entrada e destruídas na saída). A declaração de parâmetros ocorre após o nome da função.

Em geral, podem ser passados argumentos para funções de duas maneiras:

- **Chamada por valor**: copia o valor do argumento no parâmetro formal, de maneira que alterações feitas nos parâmetros da função não têm efeito nas variáveis usadas para chamá-la.
- **Chamada por referência**: como o endereço de um argumento é copiado no parâmetro, alterações feitas no parâmetro afetam a variável usada para chamar a função.

Exemplo de chamada por valor:

```
sqr (int x)
{
        x = x * x;
        return (x);
}
main()
{
        int t=10;
        printf("%d   %d", sqr(t), t); // saída do programa: 100   10
}
```

Para criar uma chamada por referência é necessário declarar os parâmetros como do tipo ponteiro. Por exemplo:

```
void swap (int *x, int *y) {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
main() {
        int i, j;
        i = 10;
        j = 20;
        swap(&i, &j); // passa os endereços de i e j
}
```

Quando uma matriz é usada como um argumento para uma função, apenas o endereço da matriz é passado como parâmetro, não uma cópia da matriz inteira. Em C, um nome de matriz sem qualquer índice é um ponteiro para o primeiro elemento na matriz. Quando é chamada uma função com um nome de matriz, um ponteiro para o seu primeiro elemento é passado para a função. Isso significa que a declaração de parâmetros deve ser de um tipo de ponteiro compatível.

Existem três maneiras de declarar um parâmetro que receberá um ponteiro para matriz. Primeiro, ele pode ser declarado como uma matriz, como mostra o exemplo a seguir.

```
// Imprime alguns números
#include <stdio.h>
void display(int num[10]);
main () {
        int t[10], i;
        for (i=0; i<10; i++)
                t[i] = i;
        display(t);
}
void display(int num[10]) {        //compilador C converte num para um ponteiro de inteiros,
        int i;                     // pois nenhum parâmetro pode receber uma matriz inteira
        for (i=0; i<10; i++)
                printf("%d ", num[i]);
}
```

A segunda forma de declarar um parâmetro de matriz é especificá-lo como uma matriz sem dimensão, conforme mostra o próximo exemplo.

```
void display(int num[]) { // num é uma matriz de inteiros de tamanho desconhecido
        int i;            // C não fornece nenhuma verificação de limites em matrizes
        for (i=0; i<10; i++)
                printf("%d ", num[i]);
}
```

A última forma de declaração, a mais comum em programas escritos profissionalmente em C, é como um ponteiro, como exemplificado a seguir [SCH 96].

```
void display(int *num) {  // Permitido porque qualquer ponteiro pode ser indexado usando [],
        int i;                  // como se fosse uma matriz
        for (i=0; i<10; i++)
                printf("%d ", num[i]);
}
```

4.4. Parâmetros da função main()

Algumas vezes é útil passar informações para um programa quando ele é executado. Geralmente, você passa informações para a função *main* através de argumentos da linha de comando (= informação que segue o nome do programa na linha de comando do sistema operacional).

Há dois argumentos internos especiais, ***argc*** e ***argv***, que são usados para receber os argumentos da linha de comando. O parâmetro *argc* contém o número de argumentos da linha de comando e é um inteiro. Ele é sempre pelo menos 1 porque o nome do programa é qualificado como um primeiro argumento. O parâmetro *argv* é um ponteiro para uma matriz de ponteiros para caracteres. Cada elemento nessa matriz aponta para um argumento da linha de comando. Todos os argumentos da linha de comando são *strings* – quaisquer números terão de ser convertidos pelo programa no formato interno apropriado. Exemplo (nome do programa = "ola"):

```
C:\> ola Joao
```

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[]) { // [] indica que a matriz é de tamanho indeterminado
        if(argc!=2) {
                printf("Voce esqueceu de digitar seu nome. \n");
                exit(1);
        }
        printf ("Ola %s \n", argv[1]);
}
```

Cada argumento da linha de comando deve ser separado por um espaço ou um caractere de tabulação. Vírgulas, pontos-e-vírgulas, etc. não são considerados separadores. Alguns ambientes permitem que se coloque entre aspas uma *string* contendo espaços. Isso faz com que a *string* inteira seja tratada como um único argumento.

4.5. Protótipo de funções

A princípio podemos tomar com regra a afirmativa de que toda função deve ser declarada antes de ser usada. Na definição da função está implícita a sua declaração, mas a linguagem C permite que se declare uma função, antes de defini-la. Esta declaração é feita através do protótipo da função. O protótipo da função nada mais é do que o trecho de código que especifica o nome e os parâmetros da função. A forma geral de uma definição de protótipo de função é:

```
<tipo_funcao>  <nome_funcao> (<tipo> <nome_param1>, <tipo> <nome_param2>, ...,
                                                        <tipo> <nome_paramN> );
```

O uso dos nomes dos parâmetros é opcional, porém eles habilitam o compilador a identificar qualquer incompatibilidade de tipos por meio do nome quando ocorre um erro, de forma que é uma boa idéia incluí-los.

Para ilustrar a diferença entre a utilização ou não de protótipos, observe os próximos exemplos [PIN 00].

```
//////////////////////////////////////////////
// Sem utilização de protótipo
#include <conio.h>
#include <dos.h>
#include <stdio.h>

void EsperaEnter()       // Definição da função "EsperaEnter"
{
        int tecla;
        printf("Pressione ENTER\n");
        do {
                tecla = getch();
                if (tecla !=13) // Se não for ENTER
                {
                        sound(700); // Ativa a emissão de um BEEP
                        delay(10);  // Mantém a emissão do som por 10 ms
                        nosound();  // Para de emitir o som
                }
        } while(tecla != 13); // 13 e' o codigo ASCII do ENTER
}

main()
{
        EsperaEnter();     // Chamada da função definida antes
        // .....
        EsperaEnter();     // Chamada da função definida antes
}

//////////////////////////////////////////////
// Com utilização de protótipo
#include <conio.h>
#include <dos.h>
#include <stdio.h>

void EsperaEnter();      // Protótipo da função

main()
{
        EsperaEnter();     // Chamada da função definida antes
        // .....
        EsperaEnter();     // Chamada da função definida antes
}

void EsperaEnter()       // Definição da função "EsperaEnter"
{
        int tecla;
        printf("Pressione ENTER\n");
        do {
                tecla = getch();
                if (tecla !=13) // Se não for ENTER
                {
                        sound(700); // Ativa a emissão de um BEEP
                        delay(10);  // Mantém a emissão do som por 10 ms
                        nosound();  // Para de emitir o som
                }
        } while(tecla != 13); // 13 e' o codigo ASCII do ENTER
}
```

Protótipos permitem que C forneça uma verificação mais forte de tipos, pois quando são utilizados o C pode encontrar e apresentar quaisquer conversões de tipos ilegais entre o argumento

usado para chamar uma função e a definição de seus parâmetros. C também encontra diferenças entre o número de argumentos usados para chamar a função e o número de parâmetros da função [SCH 96].

4.6. Arquivos de Cabeçalhos

Cada função definida na biblioteca C padrão tem um arquivo de cabeçalho associado a ela. Os arquivos de cabeçalho que relacionam as funções que são utilizadas no programa devem estar incluídos (usando *#include*) em seu programa. Há duas razões para isso. Primeira, muitas funções da biblioteca padrão trabalham com seus próprios tipos de dados específicos, aos quais o programa deve ter acesso. Esses tipos de dados são definidos em **arquivos de cabeçalho** fornecidos para cada função. Um dos exemplos mais comuns é o arquivo *stdio.h*, que fornece o tipo *FILE* necessário para operações com arquivos em disco.

A segunda razão para incluir arquivos de cabeçalho é obter os protótipos da biblioteca padrão. Se os arquivos de cabeçalho seguem o padrão C ANSI, eles também contêm os protótipos completos para as funções relacionadas com o arquivo de cabeçalho. Isso fornece um método de verificação mais forte que aquele anteriormente disponível ao programador C. Incluindo os arquivos de cabeçalho que correspondem às funções padrões utilizadas pelo programa, pode-se encontrar erros potenciais de inconsistências de tipos. A próxima tabela mostra os arquivos de cabeçalho padrões.

Arquivo de Cabeçalho	Finalidades
assert.h	Define a macro *assert()*
ctype.h	Manipulação de caracteres
errno.h	Apresentação de erros
float.h	Define valores em ponto flutuante dependentes da implementação
limits.h	Define valores em ponto flutuante dependentes da implementação
locale.h	Suporta localização
math.h	Diversas definições usadas pela biblioteca de matemática
setjmp.h	Suporta desvios não-locais
signal.h	Suporta manipulação de sinal
stdarg.h	Suporta listas de argumentos de comprimento variável
stddef.h	Define algumas constantes normalmente usadas
stdio.h	Suporta E/S com arquivos
stdlib.h	Declarações miscelâneas
string.h	Suporta funções de *strings*
time.h	Suporta as funções de horário do sistema

Torna-se interessante comentar que o padrão C ANSI reserva nomes de identificadores, começando com um caracter de sublinha (_) e seguido por um caracter de sublinha ou uma letra maiúscula, para uso em arquivos de cabeçalho [SCH 96].