**Aula 6 – Funções**

Posted in [Curso gratuito de programação em C](https://www.paulotrentin.com.br/category/programacao/curso-gratuito-de-programacao-em-c/) By [Paulo Marcos Trentin](https://www.paulotrentin.com.br/author/admin/) On fevereiro 24, 2011

**A Forma Geral de uma Função**

Sua forma geral é:

especificador\_de\_tipo nome\_da\_função(lista de parâmetros)
{

corpo da função

}

Por padrão uma função retornará um resultado de tipo inteiro. Porém qualquer tipo de valor válido é aceito em especificador\_de\_tipo. A lista de parâmetros  é uma lista de nomes de variáveis separadas por vírgulas e seus tipos associados. Uma função pode não usar parâmetros, mas em C, de qualquer forma precisaremos usar os parênteses.

**Regras de Escopo de Funções**

Regras de escopo definem se uma dada porção de código conhece ou tem acesso a outra porção de código ou dados. Em C, o código de uma função é privado, ou seja, nenhuma outra parte do programa pode acessar aquele código, exceto pela chamada da função.

Variáveis definidas dentro de funções são chamadas **locais** e só existem durante a chamada da função. Após seu término estas variáveis são destruídas. Somente podem manter seus valores entre chamadas da função se definirmos elas como **static.**Uma variável **static** funciona como uma variável global para armazenamento, porém só pode ser vista de dentro do escopo que foi definida.

Em C, não podemos definir uma função dentro de outra, por este motivo C não é tecnicamente uma linguagem estruturada em blocos.

**Argumentos de Funções**

Se uma função aceita argumentos, ela deve declarar variáveis para receberem estes argumentos, isso ocorre na definição da função e estas variáveis são chamadas de parâmetros formais da função. Como qualquer outra variável local, elas são destruídas na saída da função.

|  |  |
| --- | --- |
| 123456789 | // devolve 1 se c é parte da string s; 0 caso contráriois\_in(char \*s, char c){    while (\*s){        if (\*s == c) return 1;        else s++;    }    return 0;} |

A função acima **is\_in** tem os parâmetros **s** e **c**. Ela devolve 1 caso c exista em s e 0 do contrário.
Devemos nos assegurar que os argumentos usados na chamada da função sejam compatíveis com o tipo de seus parâmetros. O compilador C não acusar erro se os dados forem incompatíveis, porém resultados inesperados irão ocorrer.

Os parâmetros formais são variáveis locais, e podemos usá-las em qualquer expressão C permitida.

***Chamada por Valor, Chamada por Referência***

As duas chamadas são usadas para envio de argumentos à subrotinas (funções). Na **chamada por valor** ocorre a cópia do argumento para o parâmetro, assim quaisquer alterações no parâmetro não afetam os dados externos da chamada. Já na **chamada por referência** é enviado apenas o endereço do argumento, a função então trabalha com o ponteiro do valor real, ou seja, qualquer alteração interna irá alterar os dados usados na chamada.

Vejamos um exemplo de chamada por valor:

|  |  |
| --- | --- |
| 1234567891011121314 | #include <stdio.h>int sqr(int x); void main(void){    int **t** = 10;    printf("%d %d", sqr(**t**), **t**); // chamada} sqr(int x){    x = x \* x;    return (x);} |

No exemplo acima, o valor do argumento para **sqr**, 10, é copiado no parâmetro **x**. Na atribuição x = x \* x; apenas o valor da variável local **x**é modificado, ou seja a variável **t**, usada para chamar **sqr** continua ainda com valor 10. Assim a saída será **100  10**.

Lembrando, em **chamadas por valor** a função não tem efeito algum sobre a variável usada na chamada, pois apenas trabalha com uma cópia dos valores.

**Criando uma Chamada por Referência**

Apesar do padrão C usar chamadas por valor, podemos fazer chamadas por referência, onde apenas o ponteiro para o valor real é passado à função. Logo em **chamadas por referência** a função altera diretamente a variável usada na chamada, pois está trabalhando com o ponteiro que aponta para ela.

Vamos à um exemplo:

|  |  |
| --- | --- |
| 12345678910 | void swap(int \*x, int \*y){    int temp;    // Salva localmente o valor no endereço x    temp = \*x;    // Põem y em x    \*x = \*y;    // Pões x em y    \*y = temp;}OUTRO EXEMPLO:#include <stdio.h>#include <stdlib.h>int soma(int \*a, int \*b);int resultado = 0;int main(int argc, char \*argv[]){ int valor1 = 30, valor2 = 20;  resultado = soma(&valor1, &valor2); printf( "\n\tO resultado e\'%d", resultado ); system("PAUSE > NULL");  return 0;}int soma(int \*a, int \*b){ return \*a + \*b; } |

O operador **\***acessa a variável apontada por seu operando, veja na aula2 uma discussão completa de \*. A aula 5 trata exclusivamente de ponteiros.

Na função acima os conteúdos das variáveis usadas para chamar a função, são trocados. Lembrando que para fazer a chamada à uma função que recebe endereços de variáveis, precisamos fazer a chamada com os endereços dos arguemntos, por exemplo, a chamada da função **swap()** acima seria:

|  |  |
| --- | --- |
| 123456789 | void swap(int \*x, int \*y);void main(void){    int x, y;    x = 10;    y = 20;    // Usamos os endereços de x e y para fazer a chamada à função swap    swap(&x, &y);} |

O operador unário **&** é usado para produzir o endereço das variáveis. Assim na chamada de **swap** estamos enviando os endereços de **x** e **y** e não seus valores.

**Chamando Funções com Matrizes**

Ao passarmos uma matriz como argumento sempre passaremos seu endereço, ou seja, sempre haverá uma **chamada por referência**, lembrando que em C, o nome da matriz sem qualquer índice, corresponde ao endereço do primeiro elemento. Existem três maneiras de declarar um parâmetro que irá receber um ponteiro para matriz.

A primeira, ele pode ser declarado como uma matriz:

|  |  |
| --- | --- |
| 1234567891011121314 | #include <stdio.h>void display(int num[10]);void main(void){    int t[10], i;    for (i = 0; i < 10; ++i) t[i] = i;    display(t);} void display(int num[10]){    int i;    for (i = 0; i < 10; i++) printf("%d ", num[i]);} |

Apesar de **num** ter cido declarado como uma matriz, o compilador C irá convertê-lo automaticamente para um ponteiro de inteiros. Isso ocorre pois, nenhum parâmetro pode receber uma matriz inteira, assim o ponteiro da matriz é passado, e um parâmetro de ponteiro deverá estar lá para recebê-lo.

A segunda forma de especificar o parâmetro é como uma matriz sem dimensão:

|  |  |
| --- | --- |
| 12345 | void display(int num[]){    int i;    for (i = 0; i < 10; i++) printf("%d ", num[i]);} |

Aqui **num** é declarado como uma matriz de inteiros de tamanho desconhecido. Como C não faz nenhuma verificação de limites em matrizes, para o parâmetro, não importa o tamanho da matriz. Esse método de declaração define **num** como um ponteiro de inteiros.

O último método para **num** ser declarado é como um ponteiro, esta é a forma mais comum:

|  |  |
| --- | --- |
| 12345 | void display(int \*num){    int i;    for (i = 0; i < 10; i++) printf("%d ", num[i]);} |

Isto funciona, pois qualquer ponteiro pode ser indexado usando [] como se fosse uma matriz. Por isso é comum ouvirmos que em C, ponteiros e matrizes estão intimamente ligados.

Devemos sempre lembrar que ao passar uma matriz para uma função, estamos passando seu endereço inicial de memória, ou seja, a função trabalha e pode alterar diretamente os valores da matriz. Abaixo um exemplo da função **print\_upper()** que imprime seu argumento string em maiúsculas:

|  |  |
| --- | --- |
| 123456789101112131415161718192021 | #include <stdio.h>#include <ctype.h>void print\_upper(char \*string); void main(void){    char s[80];    scanf("%s", s);    print\_upper(s);} // Imprime uma string em maiúsculasvoid print\_upper(char \*string){    register int t;    for (t = 0; string[t]; ++t){        string[t] = toupper(string[t]);        putchar(string[t]);    }    putchar('\n');} |

No programa acima, após **print\_upper()** ser chamada, a matriz **s** em **main()** será modificada. Se não é esse o comportamento desejado podemos implementar o programa sem alterar o valor do ponteiro:

|  |  |
| --- | --- |
| 1234567891011121314151617181920 | #include <stdio.h>#include <ctype.h>void print\_upper(char \*string); void main(void){    char s[80];    scanf("%s", s);    print\_upper(s);} // Imprime uma string em maiúsculasvoid print\_upper(char \*string){    register int t;    for (t = 0; string[t]; ++t){        putchar(toupper(string[t]));    }    putchar('\n');} |

A função **scanf** usada acima é um exemplo clássico de passagem de matrizes para funções, veremos aqui uma função simples que nos dá uma idéia do funcionamento dela.

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627282930313233 | // Uma versão muito simples da função scanf#include <stdio.h>;char \*xscanf(char \*);void main() {    char matrix[80];    //  Podemos fazer uma chamada enviando o endereço do primeiro endereço da matriz    printf("Você digitou: %s\n", xscanf(&matrix[0]));    //  Ou podemos simplesmente chamar enviando o nome da matriz que também corresponde ao primeiro endereço    printf("Você digitou: %s\n", xscanf(matrix));} char \*xscanf(char \*s) {    char ch, \*p;    int t;    p = s;    for (t = 0; t < 80; ++t) {        ch = getchar();        switch (ch) {            //  Se for término de linha, encerra a string            case '\n':                s[t] = '\0';                return p;            //  Se for um retrocesso, decrementa t            case '\b':                if (t > 0) t--;                break;            default:                s[t] = ch;        }    }    s[80] = '\0';    return p;} |

A função **xscan** deve receber uma matriz de pelo menos 80 campos, se mais de 80 caracteres forem digitados no teclado a função retorna os valores. Obviamente a função original C não tem tais limitações, mas é interessante entendermos como esta funciona, observe com bastante atenção as duas chamadas feitas em **main()**.

**argc e argv – Argumentos para main()**

Em certos casos precisamos passar informações para o programa quando o executamos. Para tal, na linha de comando, o que fazemos é chamar o programa e então após ele inserimos os argumentos separados por espaços. Para compilar um programa C em Linux normalmente utilizamos o compilador **cc** e em seguida o nome do arquivo fonte a ser compilado. O que estamos fazendo é passando um argumento (nome do arquivo) para o programa **cc** que então trabalha com este arquivo.

Quem recebe o argumento? Simples, a função **main()** através de dois argumentos especiais, **argc** contém o número de argumentos sempre começando a partir do 1, pois o nome do programa é qualificado como primeiro argumento. Já **argv** é um ponteiro para uma matriz de ponteiros para caractere. Cada elemento desta matriz aponta para um argumento da linha de comandos. Vejamos o programa abaixo que exibe o nome e idade do usuário:

|  |  |
| --- | --- |
| 12345678910111213 | #include <stdio.h>#include <stdlib.h>void main(int argc, char \*argv[]){    if (argc == 1){        printf("Você esqueceu de digitar seu nome e idade\n");        exit(1);    }else if (argc == 2){        printf("Você esqueceu de digitar sua idade\n");        exit(1);    }    printf("Olá %s, você tem %s anos, como eu sou sabido!\n", argv[1], argv[2]);} |

Abaixo a execução do programa, note a chamada de argc\_argv que é o fonte acima compilado e em seguida os dois argumentos separados por espaçamentos.



Observando os parâmetros formais de **main()** vemos que \***argv[]** indica que iremos receber uma matriz de tamanho indefinido, sabemos isso por causa dos colchetes.

O programa abaixo conta regressivamente a partir do valor especificado na linha de comandos e avisa quando chega a 0. Podemos observar aqui o uso da função padrão **atoi()** que serve para converter uma string em um inteiro.

|  |  |
| --- | --- |
| 1234567891011121314151617181920 | #include <stdio.h>#include <stdlib.h>#include <ctype.h>#include <string.h>void main(int argc, char \*argv[]){    int disp, count;    if (argc < 2){        printf("Você deve digitar o valor a contar\n");        printf(" na linha de comandos. Tente novamente.\n");        exit(1);    }    if (argc == 3 && !strcmp(argv[2], "display")) disp = 1;    else disp = 0;    for (count = atoi(argv[1]); count; --count)        if (disp) printf("%d\n", count);    //  Na maioria dos computadore irá tocar a capainha    putchar(7);    printf("Terminei.\n");} |

Se quiséssemos acessar um caractere individual em uma das strings de comando, basta acrescentar um segundo índice à **argv.**O programa abaixo, mostra todos os argumentos quais foi chamada, um caractere de cada vez:

|  |  |
| --- | --- |
| 1234567891011121314 | include <stdio.h>void main(int argc, char \*argv[]){    int t, i;    for (t = 0; t < argc; ++t){        printf("Argumento %d: ", t);        i = 0;        while(argv[t][i]){            putchar(argv[t][i]);            ++i;        }        putchar('\n');    }} |

Aqui a execução do programa acima:



Observações importantes: Teoricamente poderíamos usar 32.767 argumentos, porém a maioria dos sistemas operacionais nos limita à muito menos que isso. É uma prática comum quando não usarmos parâmetros de linha de comando em nossos programas declararmos **main()** usando a palavra chave **void**porém podemos simplesmente deixar os parênteses vazios.  Os nomes **argc** e **argv** são tradicionalmente usados, mas poderíamos usar quaisquer outros nomes de variáveis para receber os parâmetros.

**O comando return**

O comando **return** pode ser utilizado para causar uma saída imediata da função que o contém, fazendo com que a execução do programa retorne ao código chamador. E também pode ser usado para devolver um valor.

**Retornando de uma Função**

As funções podem derminar sua execução de duas maneiras. A primeira é quando o código encerra e é encontrada a chave final do programa (}), que claro no código objeto não existe, mas podemos imaginar desta maneira. Segue um exemplo de um programa que escreve a string “Eu gosto de C” de trás para frente na tela.

|  |  |
| --- | --- |
| 12345678910111213 | #include <stdio.h>#include <string.h>void pr\_reverse(char \*s);void main(void){    pr\_reverse("Eu gosto de C");} void pr\_reverse(char \*s){    register int t;    for (t = strlen(s) -1; t >= 0; t--) putchar(s[t]);} |

Poucas funções utilizam este método de retorno, a maioria usa o comando **return** que inclusive, pode estar presente várias vezes dentro de uma mesma função. A função abaixo **find\_substr()** returna a posição inicial de uma substring dentro de uma string, ou se não encontrar nada, retorna -1:

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627282930 | #include <stdio.h>int find\_substr(char \*s1, char \*s2); void main(void){    int pos = find\_substr("C é legal", "é");     if (pos != -1)        printf("Encontrada substring na posicao %d\n", pos);    else{        printf("A substring não foi encontrada\n");    }} //  Devolve o índice de s1 em s2find\_substr(char \*string, char \*find){    register int t;    char \*p, \*p2;    for (t = 0; string[t]; t++){        p = &string[t];        p2 = find;        while(\*p2 && \*p2 == \*p){            p++;            p2++;        }        if (!\*p2) return t;    }    return -1;} |

**Retornando Valores**

Qualquer função que não seja do tipo **void** retorna algum valor. E para retorná-lo usamos o comando **return** dentro da função. Uma vez que a função não seja do tipo **void** ela pode ser usada como operando em qualquer expressão válida de C. As expressões abaixo são válidas:

|  |  |
| --- | --- |
| 123 | x = poower(y);if(max(x, y) > 100) printf("maior");for (cha = getchar(); isdigit(ch); ) ...; |

Porém uma função não pode ser destino de uma atribuição:

|  |  |
| --- | --- |
| 12 | //  Está erradoswap(x y) = 100; |

Uma questão comum sobre retorno de funções é: se uma função retorna algum valor, não somos obrigados a usá-lo. O programa abaixo mostra um exemplo disso:

|  |  |
| --- | --- |
| 12345678910111213141516 | #include <stdio.h>int mul(int a, int b); void main(void){    int x, y, z;    x = 10; y = 20;    z = mul(x, y);    printf("%d", mul(x, y));    mul(x, y);} mul(int a, int b){    return a \* b;} |

Na linha 8 o valor retornado por **mul()** é atribuído a **z**. Na linha 9 o valor não é atribuído, porém é usado pela função **printf()**. E por último na linha 10, o valor é perdido porque não é atribuído a uma variável e nem usado como parte de uma expressão.

**Funções que Devolvem Valores Não-Inteiros**

Quando o programador não define o valor de retorno de uma função, o compilador C assume o default que é o tipo **int.** Quando não desejamos retornar um **int**da função, primeiro precisamos explicitar que tipo de dados desejamos que seja retornado e segundo, o tipo de retorno da função deve ser identificado antes da chamada desta.

As ufnções podem ser declaradas como retornando qualquer tipo de dado válido em C. Semelhante as variáveis, o especificador de tipo precede o nome da função, e ele informa ao compilador qual tipo de dado a função devolverá. Essa informação é crítica para o correto funcionamento do programa.

Se nossa função retorna um valor **não inteiro** e não for declarada antes de ser chamada, o compilador C irá gerar um código de chamada à função errado. Para evitar isso devemos usar uma forma especial de declaração perto do início do programa, informando ao compilador que tipo de dado a função retornará.

Existem duas maneiras de declarar uma função antes de usá-la: a forma tradicional e o novo método de **protótipos de funções** recomendado pelo padrão ANSI. Para garantir compatibilidade o padrão ANSI permite o uso tradicional de declaração de funções, porém desencoraja-o em novos projetos.

O código abaixo faz o uso da abordagem tradicional de identificação da função:

|  |  |
| --- | --- |
| 12345678910111213141516 | #include <stdio.h>//  Identifica a funçãofloat sum();float first, second; void main(void){    first = 123.23;    second = 99.09;    printf("%f", sum());} float sum(){    return first + second;} |

Com a declaração no início do programa, o compilador saberá que **sum()** retorna um valor em ponto flutuante. Sem essa declaração (comentando a linha 3) o compilador indicaria um erro de incompatibilidade de dados como vemos abaixo.



**Protótipos de Funções**

Originalmente C não tinha protótipos de funções, talvez este seja o acréscimo mais importante que o padrão ANSI trouxe. Os protótipos permitem que informemos a quantidade e tipos de argumentos das funções e isso garante a C uma verificação mais forte de tipos. Isso é bom, pois assim C pode avisar quando uma conversão de tipo é ilegal na passagem de argumentos para funções.

A forma geral para protótipos de funções é:

tipo nome\_func(tipo nome\_param1, tipo nome\_param2, … , tipo nome\_paramN);

Apesar de facultativo é sempre importante declarar uma lista de parâmetros para a função, isso garante a verificação na passagem de valores. O programa abaixo tenta passar um inteiro para a função **sqr\_it** quando esta espera receber um ponteiro para inteiro:

|  |  |
| --- | --- |
| 1234567891011121314151617 | //  Este programa usa um protótipo de função para forçar uma verificação forte de tipos //  Protótipo da função sqr\_itvoid sqr\_it(int \*i); void main(void){    int x;    x = 10;    //  Aqui ocorre a incompatibilidade de tipos    sqr\_it(x);} void sqr\_it(int \*i){    \*i = \*i + \*i;} |

Uma dúvida comum é, em uma função que não recebe nenhum argumento, como ficaria seu protótipo? Simples, apenas usamos o **void** dentro dos parênteses:

float generate\_float(void);

Isso diz ao compilador que qualquer chamada à função **generate\_float** com algum parâmetro é inválida. Precisamos ter em mente que apesar de não ser obrigatório (pelo fato do padrão ANSI manter compatibilidade), precisamos **sempre** usar protótipos de funções.

**Retornando Ponteiros**

Devemos sempre lembrar que ponteiro não é uma variável, tampouco inteiros sem sinal. Um ponteiro é um endereço de memória para um certo tipo de dado. É importante saber qual tipo de dado, pois no uso da aritmética de ponteiros isso é crucial, pois ao incrementar o ponteiro, ele irá apontar para a próxima ocorrência do valor e não o próximo byte.

A função **match()** abaixo retorna um ponteiro para a primeira ocorrência de **c**em **s**. Se não encontrar retornará um ponteiro para o terminador nulo ‘\0’. Ela não funcionaria se **s** fosse declarada como um ponteiro para **float** por exemplo.

|  |  |
| --- | --- |
| 12345678910111213141516171819202122 | #include <stdio.h>//  Protótipo da funçãochar \*match(char c, char \*s); void main(int argc, char \*argv[]){    char \*p, find = 'a';    char \*word = argv[1];     p = match(find, word);     if (\*p)        printf("encontrei o caracter 'a' na sua frase.\n");    else        printf("Não encontrei o caracter 'a'.\n");} char \*match(char c, char \*s){    while(c != \*s && \*s) s++;    return (s);} |

**Funções do Tipo void**

O tipo **void** pode ser usado para declarar explicitamente funções que não retornam nada. Isso é útil para garantir que tal função não seja utilizada em expressões afastando o mau uso acidental. Como qualquer função não inteira, funções com o tipo de retorno **void** devem ser declaradas usando o **protótipo de função**. O programa abaixo imprime valores na vertical usando uma função que não devolve nenhum valor.

|  |  |
| --- | --- |
| 1234567891011121314 | #include <stdio.h>//  Protótipo da funçãovoid print\_vertical(char \*str); void main(int argc, char \*argv[]){    if (argc) print\_vertical(argv[1]);} void print\_vertical(char \*str){    while(\*str)        printf("%c\n", \*str++);} |

**O que main() Devolve?**

O padrão ANSI estabelece que a função **main()** devolve um inteiro para o processo chamador, que geralmente é o sistema operacional. Chamar a função **exit()** é o mesmo que devolver um valor em **main()**. Se main não devolver explicitamente um valor, o processo que a chamou receberá um valor tecnicamente indefinido. Na prática a maioria dos compiladores C devolvem 0, porém não conte com isso se houver interesse em portabilidade.

**Recursão**

Recursão é a capacidade de uma função chamar a si mesma. Um exemplo simples é uma função que calcula o fatorial de um número. Abaixo temos uma versão recursiva e uma normal da função **factr()**:

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627 | #include <stdio.h>//  Protótipo das funçõesint factr(int);int factr\_normal(int); void main(void){    printf("Chamada recursiva para 5: %d\n", factr(5));    printf("Chamada normal para 5: %d\n", factr\_normal(5));} factr(int n){    int answer;    if (n == 1) return(1);    answer = factr(n -1) \* n;    return (answer);} factr\_normal(int n){    int t, answer;    answer = 1;    for (t = 1; t <= n; t++)        answer \*= t;    return (answer);} |

O funcionamento de **factr()**é que ao ser chamada com arguemnto de 1, devolve 1, do contrário devove o produto de **factr(n -1) \* n**. Isso acontece até que **n** se iguale a 1 e as chamadas da função comecem a retornar.

Para calcular o fatorial de 2, a primeira chamda de **factr()** provoca uma segunda chamada com o argumento **n -1** que é 1. Essa segunda chamada irá então retornar 1 que é então multiplicado por **n** neste caso 2. Logo teremos 2 x 1 que é o valor fatorial de 2. Usar comandos **printf()** entre as chamadas é interessante para observar melhor o funcionamento da recursividade.Para fixar melhor fiz um video mostrando o debugger do programa acima que executa o fatorial de 5:

Quando uma função chama a si mesma, não é feita uma nova cópia desta função. O que ocorre é que novos parâmetros e variáveis locais são alocados na pilha, e o código da função então é executado com estes novos parâmetros. Por este motivos devemos sempre lembrar que uma função recursiva deve **sempre** ter um **ponto de parada** como um comando **if** em algum lugar que force a função a retornar o valor em vez de chamar a si mesma novamente. Do contrário ela causaria um estouro na pilha e a possível sobreescrita de dados do programa.

Curiosidade: Um microcontrolador PIC16F628 contém 8 níveis de pilha. Logo se uma função chamar a si mesma mais que 7 vezes, irá perder-se os dados da primeira chamada à função, causando resultados inesperados na execução do programa.

A maioria das funções recursivas não trazem melhoras significativas em tamanho de código, e muitas vezes são executadas mais lentamente que suas equivalentes iterativas. Logo, a principal vantagem do uso de recursividade é deixar algoritmos mais claros e simples. Algoritmos de ordenação como QuickSort seriam muito difíceis de serem implementados sem recursão.

**Declarando uma Lista de Parâmetros de Extensão Variável**

As vezes precisamos receber uma lista desconhecida de argumentos em uma função. É o caso de **printf()**onde são passados sempre dois ou mais argumentos. Para informar ao compilador que um número desconhecido de parâmetros será passado para a função devemos terminar sua declaração com três pontos. A função abaixo terá pelo menos dois parâmetros inteiros e um número desconhecido (incluindo 0) de parâmetros após eles.

func(int a, int b, …);

Em um protótipo de função também podemos usar a forma de declaração acima.

Devemos observar que uma função que recebe argumentos variáveis, deve ao menos ter um argumento verdadeiro. O exemplo abaixo está incorreto:

func(…);

**Declaração de Parâmetros de Funções Moderna**versus**Clássica**

Originalmente C usava um método de declaração de parâmetros diferente, também conhecido como forma *clássica*. O livro “C Completo e Total” segue o padrão ANSI e recomenda o uso da forma *moderna*. Porém é importante que saibamos como é a forma clássica afinal existe muito software feito em C que utiliza desta forma seja por ter sido desenvolvido a muito tempo, seja para manter compatibilidade com compiladores antigos.

A declaração clássica consiste em duas partes: Primeiro definimos o nome dos parâmetros dentro dos parênteses que seguem o nome da função. E segundo, após os parênteses e antes da chave que inicia o escopo da função, fazemos as declarações reais dos parâmetros.

A forma geral da declaração clássica é:

tipo nome\_funcao(param1, param2, … paramN)

tipo param1;

tipo param2;

….

tipo paramN

{

código da função

}

abaixo um exemplo de uma forma clássica de declaração de parâmetros:

|  |  |
| --- | --- |
| 123456 | float funcao(a, b, ch)int a, b;char ch;{    ...} |

**Questões Sobre a Implementação**

Alguns poucos pontos que afetam a eficiência e usabilidade, precisam ser observados em relação a criação de funções em C.

**Parâmetros e Funções de Propósito Geral**

Uma função usada para propósito geral é aquela usada em várias situações, e geralmente por muitos programadores. Tipicamente, não devemos usar dados globais neste tipo de funções, é mais interessante passarmos todos os dados necessários para ela via parâmetros. Caso ainda assim seja necessária uma variável que mantenha seu valor entre chamadas, devemos recorrer ao uso de **variáveis estáticas**.

**Eficiência**

Funções são os blocos construcionais em C, e são usadas em qualquer programa. Porém, em certas aplicações especializadas quando se deseja muito desempenho, talvez precisemos substituir uma função por um código *em linha (inline).*Código em linha é o equivalente ao da função porém são usados sem uma chamada de função. Isso torna-o mais rápido devido ao não uso da instrução CALL e também da pilha, visto que não ocorre a passagem de argumentos.

Na maioria das aplicações este aumento muito pequeno de tempo não fará diferença, porém quando se deseja desempenho cada chamada de função pode ser crítica. Abaixo exemplos de um programa em linha e outro com chamada à função:

|  |  |
| --- | --- |
| 12345678910111213141516171819202122 | //  Versão em linhamain (void){    int x;     for (x = 1; x < 11; ++x)        printf("%d", x \* x);} //  Versão com chamada à funçãomain (void){    int x;     for (x = 1; x < 11; ++x)        printf("%d", sqr(x));} sqr(int a){    return a \* a;} |

**Bibliotecas e Arquivos**

Uma vez escrita a função pode ser armazenada no mesmo arquivo da função **main(),** podemos também colocá-la em um arquivo separado com outras funções ou podemos colocá-la em uma biblioteca.

**Arquivos Separados**

Em um grande programa, procurar por vários arquivos em busca de uma função pode ser muito frustrante. Uma organização eficiente evita este tipo de problema.

Primeiro, devemos agrupar funções que estejam conceitualmente relacionadas, por exemplo em um programa de tratamento de imagens, as funções que redimensionam a imagem podem ficar em um arquivo juntas.

Segundo, colocamos as funções de uso geral juntas. Em um banco de dados, funções de formatação de entrada/saída são usadas por outras funçõe e devem estar em um arquivo separado.

Terceiro, devemos agrupar funções de nível mais alto em um arquivo separado ou se houver espaço no arquivo **main().**

**Bibliotecas**

Tecnicamente uma biblioteca de funções é diferente de um arquivo de funções compiladas separadamente. Em um arquivo de fontes criado para nosso programa provavelmente iremos utlizar todas as funções, logo é viável linkeditar todas as funções com o programa. Já em uma biblioteca geralmente não utilizaremos todas as suas funções. No caso de uma biblioteca C padrão, nunca iríamos querer todas as funções linkeditadas com nosso programa, isso geraria um código-objeto muito grande.

**De Que Tamanho Deve Ser um Arquivo de Programa?**

O tempo de compilação está diretamente relacionado ao tamanho do arquivo que está sendo compilado. O tamanho de um arquivo varia de usuário, compilador e ambiente operacional. Porém a regra geral é que nenhum arquivo fonte deve ser maior que 15000 bytes.