Subprogramas

→ **Subprogramas** são chamados de funções, procedimentos, sub-rotinas, tarefas, métodos, etc.., conforme a linguagem usada.

→ Técnica de particionar o programa em partes menores = modelar o problema em subproblemas menores, com resolução mais simples, as quais seriam executadas quando fossem necessárias.

→ Subprogramas são chamados pelo programa principal.

→ Programa tem o subprograma, especializado na execução de uma determinada tarefa, que é chamado no momento em que a execução da tarefa é necessária.

→ O programa fornece ao subprograma todas as informações exigidas para a sua execução, deixando-o então trabalhar até que a tarefa seja concluída.

→ Nesse momento o subprograma devolve, junto com os resultados de sua operação, o controle da execução ao programa, que iria então concluir sua atividade.

Benefícios dos Subprogramas

*** Divisão para a conquista**: construção de programas a partir de partes ou componentes menores.

*** Módulos**: maior facilidade de gestão de cada módulo do que do programa original. → reduzir a complexidade de um programa → são uma ferramenta essencial ao desenvolvimento estruturado de aplicações.

*** Abstração**: ocultação de detalhes internos (funções de biblioteca)

*** Repetição de código evitada**.

Funções: são módulos em C.

→ Todo subprograma em C é uma função, incluindo-se a função *main*, com a qual se inicia a execução do programa.

→ Em C ⇒ possibilidade de combinação de **funções definidas pelo usuário** com **funções das bibliotecas** nos programas.

⇒ **Funções na biblioteca padrão de C**

Exemplos:

1- *printf*, *scanf*, getchar, etc... – que fazem parte da biblioteca *standard* de C → são fornecidas quando adquirimos qualquer compilador de C.

2- *Bibliotecas de funções matemáticas* (*#include "math.h"*)

```
printf("%.2f", sqrt( 900.0) );
```

⇒ **Funções definidas pelo usuário**: servem para a modularização de um programa.

Exemplo:

Suponha que se queira escrever um programa que coloque na tela a seguinte saída, escrevendo a linha de 20 asteriscos através de um laço **for**:

```
********************
Numeros entre 1 e 5
********************
1
2
3
4
5
********************
```

```
#include <stdio.h>
main()
{
    int i;
    for (i=1;  i<=20;  i++)
        putchar('*');
    putchar('\n');
    puts("Numeros entre 1 e 5");
    for (i=1;  i<=20;  i++)
        putchar('*');
    putchar('\n');
    for (i=1;  i<=5;  i++)
        printf("%d\n",i);
    for (i=1;  i<=20;  i++)
        putchar('*');
    putchar('\n');
}
```

→ Repetimos três vezes o mesmo conjunto de código para escrever uma linha de asteriscos na tela ⇒ o ideal seria escrever esse trecho de código apenas uma única vez e invocá-lo sempre que necessário.

→ **Solução:** dividir o programa em pequenos fragmentos de código, cada um ficando responsável por determinada tarefa.

→ Poderíamos escrever um programa que coloque uma linha com 20 asteriscos na tela.

→ Como a função desse programa é escrever uma linha, em vez de chamá-lo ***main***, vamos chamá-lo de ***linha***.

```
#include <stdio.h>
linha()
{      int i;
       for (i=1;  i<=20;  i++)
            putchar('*');
       putchar('\n');
}
```

→ Se tentarmos criar um executável com esse código, vamos obter um **erro**: a <u>função main não se encontra presente no programa</u>.

Programa em C tem que possuir SEMPRE a *função main()* escrita no seu código, independentemente do número e da variedade de funções que o programa contenha.

→ É a **função main()**, ou outra qualquer função invocada pela função main(), que terá que solicitar os serviços da **função linha()**.

```
#include <stdio.h>
linha()
{
       int i;
       for (i=1;  i<=20;  i++)
            putchar('*');
       putchar('\n');
}
main()
{
       int i;
       linha();
       puts("Numeros entre 1 e 5");
       linha();
       for (i=1;  i<=5;  i++)
            printf("%d\n",i);
       linha();
}
```

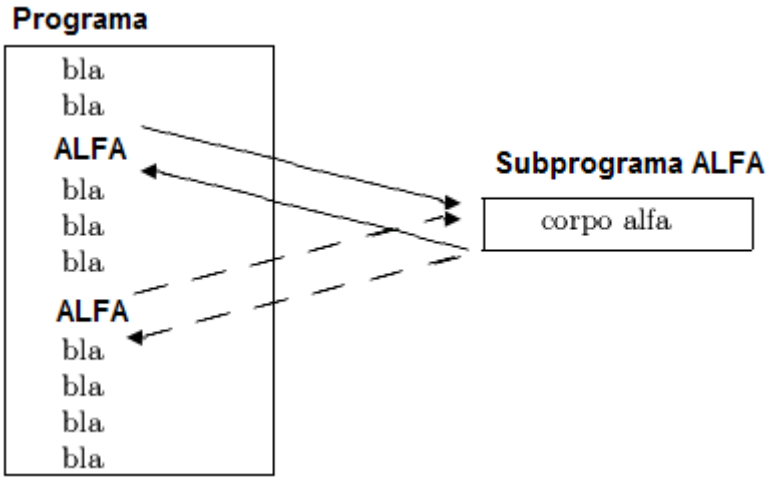
→ Este programa possui 2 funções:

- **função main():** responsável por iniciar o programa e executar todas as instruções presentes em seu interior.

- **função linha():** responsável por escrever uma linha na tela. Sempre que pretendermos escrever uma linha na tela bastará invocar a função linha(), evitando escrever sempre todo o código que esta executa.

→ **Uma das principais vantagens desse tipo de abordagem** = se for necessário alterar a linha da tela, bastará alterar apenas uma vez o código respectivo na função linha.

Como Funciona uma Função = Como o computador usa subprogramas?



Visão de fluxos da execução de subprogramas

→ Código de uma função: só é **executado** quando esta é invocada em alguma parte do programa.

→ Quando função é invocada ⇒ o programa que a invoca é "suspenso" temporariamente. Em seguida, são executadas as instruções presentes no corpo da função. Uma vez terminada a função, o controle de execução do programa volta ao local em que esta foi invocada.

→ O programa que invoca uma função pode enviar **ARGUMENTOS**, que são recebidos pela função. Estes são recebidos e armazenados em variáveis locais, que são automaticamente iniciadas com os valores enviados. A essas variáveis dá-se o nome de **PARÂMETROS**.

→ Depois de terminar o seu funcionamento, uma função pode devolver um valor para o programa que a invocou.

Exemplo: Escreva um programa que escreva na tela a seguinte saída:

```
***
*****
*******
*****
***
```

```
#include <stdio.h>
linha3x()          //função responsável por escrever 3 asteriscos na tela
{       int i;
        for (i=1;  i<=3;  i++)
             putchar('*');
        putchar('\n');
}
linha5x()          //função responsável por escrever 5 asteriscos na tela
{       int i;
        for (i=1;  i<=5;  i++)
             putchar('*');
        putchar('\n');
}
linha7x()          //função responsável por escrever 7 asteriscos na tela
{       int i;
        for (i=1;  i<=7;  i++)
             putchar('*');
        putchar('\n');
}
main()             //função que invoca as funções
{       linha3x();
        linha5x();
        linha7x();
        linha5x();
        linha3x();
}
```

→ Código das 3 funções: é em tudo igual, exceto nas seguintes linhas:

```
for (i=1;  i<=3;  i++)
for (i=1;  i<=5;  i++)
for (i=1;  i<=7;  i++)
```

que corresponde ao número de asteriscos a serem apresentados na tela.

→ ***Ideal*** = escrever uma ÚNICA função *linha*(), com um número de asteriscos específico em cada chamada ⇒ indicar à função o numero de caracteres a serem colocados na tela.

Se quisermos escrever 3 asteriscos, invocamos a função *linha(3)*.
Se quisermos escrever 5 asteriscos, invocamos a função *linha(5)*.
Se quisermos escrever 123 asteriscos, invocamos a função *linha(123)*.

⇒ A função é sempre a mesma, função *linha*, o que muda é o número de caracteres para serem colocados na tela.

```
#include <stdio.h>
linha (int num)
{
        int i;
        for (i=1;  i<=num;  i++)
             putchar('*');
        putchar('\n');
}
main()
{
        linha(3);
        linha(5);
        linha(7);
        linha(5);
        linha(3);
}
```

→ A função *linha* recebe dentro de parênteses um valor do tipo inteiro, o qual terá que ser colocado numa variável.

→ Depois de armazenado o valor, o laço da função *linha* terá que executar o número de vezes que está armazenado nessa variável.

→ Dados da função:

- **nome**: *linha()*
- **números de parâmetros**: 1
- **tipo de parâmetro**: inteiro
- **nome da variável que vai armazenar esse parâmetro**: *num*
- **cabeçalho da função**: *linha (*int *num)*
- **corpo da função**: será alterado para *for (i=1; i<=**num**; i++)*

Formato de Definição de uma Função

```
TipoFunção  NomeFunção (Lista_de_parâmetros)
{
   //declarações e atribuições
   Variáveis internas da função

   Corpo da função
}
```

→ **NomeFunção**: nome simbólico pelo qual outros trechos do programa poderão ativar (e fazer uso) da função.

***** Escolha do nome de uma função obedece às regras apresentadas para as variáveis.

***** Nome de uma função deve ser único: não pode ser igual ao nome de outra função ou de uma variável.

***** Deve especificar aquilo que a função faz, e deve ser de fácil leitura e interpretação.

→ **Lista de parâmetros**: declaração de uma série de parâmetros.

***** *Parâmetros* são valores que as funções recebem da função que a chamou.

↳ permitem que uma função passe valores para outra → são os dados que devem ser passados como informações necessárias ao seu funcionamento.

* Comunicação com uma função = através dos argumentos que lhe são enviados e dos parâmetros presentes na função que os recebe.

* Qualquer tipo de dados da linguagem pode ser enviado como parâmetro para uma função, mesmo o tipo de dados que venham a ser definidos pelo programador.

* Os parâmetros de uma função são separados por vírgula → é absolutamente necessário que para cada um deles seja indicado o seu tipo.

funcao (**int** x, **char** y, **float** k, **double** xpto)

funcao (**int** x, y, k, xpto) //exemplo incorreto

* Um parâmetro não é nada mais do que uma **variável local** à função a que pertence. Um parâmetro é automaticamente iniciado com o valor enviado pelo programa invocador.

* A **passagem de argumentos** para uma função deve ser realizada colocando-se dentro de parênteses, separados por vírgulas, imediatamente após o nome da função.

* Quando se faz a chamada de uma função, **o numero e o tipo** dos argumentos enviados devem ser ***coincidentes*** com os parâmetros presentes no cabeçalho da função.

Exemplo:

```
main()
{   ....
    funcao ('A', 123, 23.456);
    ...
}
funcao (char ch, int n, float x)
{
      ...
}
```

* É comum chamar parâmetro tanto aos argumentos de invocação de uma função como aos verdadeiros parâmetros da função.

* Qualquer expressão válida em C pode ser enviada como argumento para uma função.

* o nome das variáveis (parâmetros) presentes no cabeçalho de uma função é totalmente independente do nome das variáveis que lhe serão enviadas pelo programa que a invoca.

* Os parâmetros podem ainda não existir, caso a função não necessite de parâmetros. *Ex*: função *rand*(), que gera números aleatórios.

Exemplo: Alterando o programa anterior de forma que a função *linha* escreva qualquer caractere, e não apenas o caractere asterisco.

```
#include <stdio.h>
linha (int num, char ch)
{
        int i;
        for (i=1;  i<=num;  i++)
             putchar(ch);
        putchar('\n');
}
main()
{
        linha(3,'+');
        linha(5,'+');
        linha(7,'-');
        linha(5,'*');
        linha(3,'*');
}
```

→ **Corpo da Função**: corresponde ao código efetivo da função, contendo todas as ações necessárias para que se realize a sua tarefa.

* é constituído por instruções de C de acordo com a sintaxe da linguagem.

* tem que se seguir imediatamente ao cabeçalho da função, e é escrito entre chaves.

* o cabeçalho de uma função NUNCA deve ser seguido de ponto-e-virgula (;)

* sempre que uma função é invocada pelo programa, o corpo da função é executado, instrução a instrução, até terminar o corpo da função ou até encontrar a instrução *return*, voltando imediatamente ao programa em que foi invocada.

* dentro do corpo de uma função pode-se escrever qualquer instrução ou conjuntos de instruções da linguagem C. Em C **não se pode** definir funções dentro de funções.

* qualquer instrução é admissível dentro de uma função (ex: atribuições, *if*, *for*, *switch*, ... invocar outras funções, etc.)

* o numero de instruções que pode estar presente dentro de uma função não tem qualquer limite, deve, no entanto, ser relativamente pequeno e responsável por realizar uma única tarefa.

→ **TipoFunção**: refere-se ao tipo do valor retornado após a execução da função.

* Função: responsável por realizar uma determinada tarefa e é possível que, terminada essa tarefa, devolva UM ÚNICO resultado.
⇩ esse resultado poderá ser armazenado numa variável ou aproveitado por qualquer instrução.

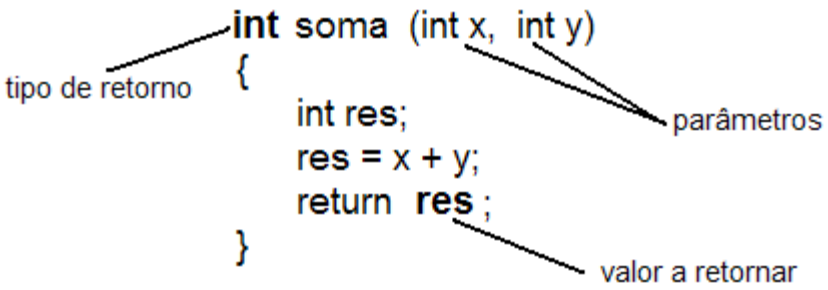
* A devolução de um resultado é feita através da instrução *return*, seguida do valor a ser devolvido: **return** expressão***;*** ou **return** (valor);

* A instrução *return* permite terminar a execução de uma função e voltar ao programa que a invocou.

Exemplo: suponha que quiséssemos calcular a soma de 2 números inteiros.

```
n = soma(3,4);
printf("%d\n",n);
```

Repare que a função soma terá que receber dois inteiros e terá que devolver um resultado do tipo inteiro, que corresponderá à soma dos dois parâmetros recebidos pela função.



⇒ Escreva um programa em C que solicite dois números ao usuário e calcule, utilizando funções distintas, o resultado da sua soma e o dobro de cada um deles.

* Uma função pode ser invocada dentro de outra função. O resultado é o mesmo que se obteria se, em vez da chamada à função, aí estivesse o resultado devolvido por esta.

EX: Qual a saída da linha abaixo?

```
printf("%d", dobro(som(dobro(2),3)));
```

* Uma função pode conter várias instruções ***return***. No entanto, apenas uma instrução ***return*** é executada na função.

EX:

```
int max (int n1, int n2)
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}
```

⇒ Sempre que no cabeçalho de uma função não é colocado o tipo de retorno, este é substituído pelo tipo *int* ⇒ ***linha (int n)*** ≡ ***int linha (int n)***

* FUNÇÃO = pode ser **invocada** de 3 formas distintas:

1- dentro de uma atribuição, para armazenar o valor dentro de uma variável:

```
x    = soma(23,y) + dobro(k+2);
alfa = func_beta(tg(x), y, 2);
```

2- dentro de uma função, em que o valor de retorno é aproveitado como parâmetro para outra função:

```
printf("%d %d", dobro(5), soma(dobro(2), 3+2));
        ou
if (soma(x,y) > 0)
```

3- sem o valor de retorno.

```
puts("Uso da função puts");   // chama a função puts sem uso de resposta.
getchar();                    // para parar a tela. Não interessa qual o char digitado.
```

* O tipo de uma função pode ser *int*, *float*, *char*, *double* ou qualquer outro tipo declarado para o programa, incluindo o tipo ***void*** quando não se retorna valor de forma explícita.

* É habitual, também, encontrar a palavra reservada ***void*** para indicar que uma função não recebe qualquer parâmetro.

```
void linha ()
{
    int i;
    for (i=1;  i<=20;  i++)
         putchar('*');
    putchar('\n');
}
```

```
void linha (void)
{
    int i;
    for (i=1;  i<=20;  i++)
         putchar('*');
    putchar('\n');
}
```

Onde colocar as Funções

→ podem ser colocadas em qualquer local, antes ou depois de serem invocadas, antes ou depois da função *main*. Existe uma restrição que deve ser levada em conta!!

Ex:

```
#include <stdio.h>
main()
{       linha();
        printf("Hello\n");
        linha();
}
void linha ()
{       int i;
        for (i=1;  i<=20;  i++)
             putchar('*');
        putchar('\n');
 }
```

Embora o código esteja formalmente bem escrito, vamos obter um **erro de compilação** semelhante a ***"function linha: redefinition"***.

- *Como será possível que a função linha esteja redefinida, se ela é escrita apenas uma vez?*

- *Como então resolver esse problema?*

```
#include <...>
void f1(int n, char ch);       //Protótipos das funções
int  max(int n1, int n2);
main()
{       .... }
void f1(int n, char ch)
{   ...       }
int  max(int n1, int n2)
{   ...       }
```

→ **Variáveis internas da função**: são as variáveis declaradas *dentro* dos blocos, sendo portanto de escopo local a ela;

* variáveis podem ser declaradas dentro do corpo de uma função. Essas variáveis são **visíveis** (isto é, conhecidas) apenas dentro da própria função.

* as variáveis declaradas dentro de uma função só são conhecidas dentro dessa função. São, por isso, denominadas **variáveis locais**. Essas variáveis só podem ser utilizadas dentro da própria função.

* se uma mesma variável for declarada em duas funções distintas, não haverá qualquer tipo de problema, pois o compilador sabe qual utilizar em cada uma delas. Apesar de terem o mesmo nome, são variáveis distintas sem qualquer relação.

* depois de terminada a execução de uma determinada função, todas as suas variáveis locais são destruídas.

* sempre que possível recorra a variáveis locais, evitando assim os efeitos colaterais que ocorrem quando se utilizam variáveis globais.

* funções *não* podem ser definidas *dentro* de outras funções.

Características de uma Função

→ Cada função tem que ter um nome único, o qual serve para a sua invocação em algum lugar no programa a que pertence.

→ Uma função pode ser invocada a partir de outras funções.

→ Uma função (como o seu nome indica) deve realizar UMA ÚNICA TAREFA bem definida.

→ Uma função deve comportar-se como uma caixa preta. Não interessa como funciona, o que interessa é que o resultado final seja o esperado, sem efeitos colaterais.

→ O código de uma função deve ser o mais independente possível do resto do programa, e deve ser tão genérico quanto possível, para poder ser reutilizado em outros projetos.

→ Uma função pode receber parâmetros que alterem o seu comportamento de forma a adaptar-se facilmente a situações distintas.

→ Uma função pode retornar, para a entidade que a invocou, um valor como resultado do seu trabalho.