FUNÇÕES

A maioria dos programas resolvem problemas muito maiores do que os programas apresentados até agora.

A melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas partes ou componentes, sendo cada uma delas mais fácil de manipular que o programa original.

Essa técnica é chamada **dividir para conquistar**.

Uma das formas de modularização é criação de **funções**.

Uma **função** é um modulo (sequencia de código) criado para resolver uma tarefa simples e bem definida.

Normalmente o próprio nome da função indica a tarefa que será realizada pela mesma. Se escolher um nome que expresse o que a função faz for difícil, é possível que a função esteja tentando realizar muitas tarefas diferentes.

Nesse caso normalmente, é melhor dividir tal função em funções menores.

Em programas que contêm muitas funções, a função **main()** vai ser composta por chamadas de funções e essas funções serão responsáveis pela solução do problema proposto.

As funções que podem ser usadas são:

- as funções criadas pelo programador;
- as funções das bibliotecas disponíveis.

As vantagens de uso das funções:

- reutilização de software;
- facilidade de desenvolvimento e manutenção;

Os operadores que fazem parte do corpo da função são escritos em um único lugar e uma única vez e permanecem ocultos para resto do código.

As funções são “chamadas” ou “ativadas” ao longo da execução do programa.

No momento da chamada é especificado:

- o nome da função
- os argumentos – a informação que é necessária para que a função cumpra seu proposito

Depois da execução a função retorna o resultado para o ponto em qual foi chamada (normalmente isso ocorre usando comando **return**).

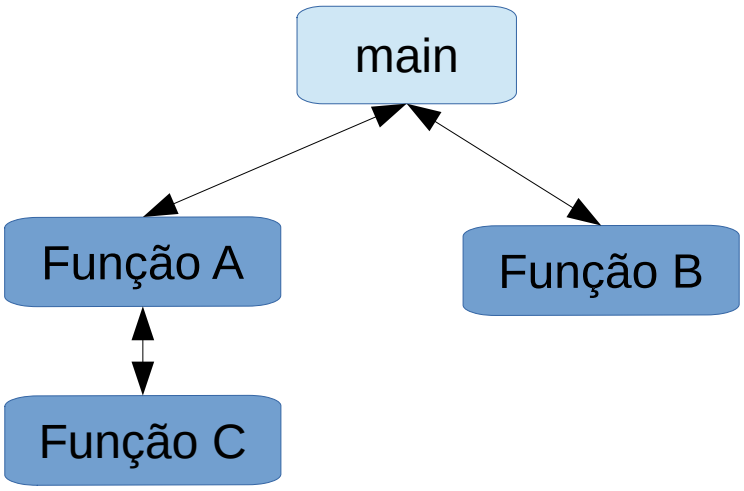

Fig.1: Chamada de funções

Todas as variáveis usadas em uma função são consideradas **locais** – elas existem somente dentro daquela função.

A transferência de informação entre as funções acontece por meio dos argumentos (ou parâmetros). Esses parâmetros também são considerados como variáveis **locais**.

Para criar uma função precisamos definir a declaração da função (ou cabeçalho) e o corpo da função.

Declaração/Cabeçalho:

```
tipo_retornado nome_da_função (lista dos parâmetros);
```

fornece ao compilador as informações sobre como chamar a função:

- o tipo do resultado produzido pela função
- o nome da função
- o tipo dos parâmetros (a função pode não ter nenhum parâmetro) e a sequencia na qual esses parâmetros serão fornecidos

O **corpo** da função pode ser definido separadamente da declaração e deve conter a informação sobre variáveis locais e os comandos de processamento propriamente ditos.

```
tipo_retornado nome_da_função (lista dos parâmetros)
{
        declarações;
        operadores;
        return ;
}
```

As declarações e os comandos entre chaves formam o corpo da função, também

chamado de bloco.

A linguagem C/C++ não permite declaração de uma função dentro da outra função.

Quando o tipo de retorno da função não é declarado, automaticamente assume-se que a função retornará um valor do tipo **int**.

Se a função não deve retornar nenhum dado, o tipo de retorno deve ser declarado como **void**.

Exemplo:

function (int);	recebe um argumento do tipo int retorna um valor do tipo int
int function (int);	recebe um argumento do tipo int retorna um valor do tipo int
void function (int);	recebe um argumento do tipo int não retorna nenhum tipo de dados
void function( );	não recebe nenhum argumento não retorna nenhum tipo de dados

O compilador usa protótipos de funções para validar as chamadas de funções.

A lista de parâmetros é uma lista separada por vírgulas, contendo as declarações dos parâmetros recebidos pela função quando ela é chamada.

Os argumentos da função devem receber nomes próprios: isso deve ser feito junto à definição do corpo da função, mas os mesmos nomes podem aparecer na declaração da função (para ajudar na legibilidade do programa).

O tipo para cada parâmetro na lista de parâmetros de uma função deve ser listado explicitamente.

Exemplo: Uma função recebe dois argumentos do tipo **int**.

int function (int x, int y);	correto
int function (int x, y);	incorreto

Exemplo 1: Função **max** (retorna o maior de dois números)

```c
#include <stdio.h>

int max(int num1, int num2);  // declaração de função
//=====================================================================
int main ()
{
   /* variaveis locais */
   int a, b;
   int num_max;

   printf("\n O programa retorna o maior de dois números\n");
   printf("\n Digite 1o numero: ");
   scanf("%d",&a);
   printf("\n Digite 2o numero: ");
   scanf("%d",&b);

   num_max = max(a, b); /* chamada de função */

   printf( "\n\n O maio número é : %d\n", num_max );

   return 0;
}
//=====================================================================
/* a função retora o maximo de dois numeros */
int max(int num1, int num2)
{
   /* variaveis locais */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Há diferentes maneiras de retornar o controle para o ponto no qual uma função foi chamada:

1. **Explicita –** usando o operador:

return expressão;

retorna o valor de expressão para a função que realizou a chamada, por exemplo:

return (a+b);

return a+1;

return;

2. **Natural** – ocorre depois da execução do último comando da função (quando a chave que indica o término da função é alcançada).

Exemplo 2: Função **square**

```c
#include <stdio.h>

int square(int num);  // declaração de função
void printl();
//=================================================================
int main ()
{
   /* variaveis locais */
   int i;

   printf("\n O programa retorna n * n, [0..10] \n\n");

   for(i = 0; i<=10; i++)
   {
     printf("%i elevado ao quadrado = %i",i, square(i));
     printl();
   }

   return 0;
}
//=================================================================
/* a função retora o num x num */
int square(int num)
{
   return num * num;
}
//-----------------------------------------------------------------
/* a função desenha uma linha */
void printl()
{
  printf("\n ----------------------------------------\n\n");
  return;
}
```

Funções das bibliotecas padrão

Como foi mencionado, o C/C++ oferece várias funções nas próprias bibliotecas padrão.

Para usar as funções disponíveis deve ser incluído no programa o nome da biblioteca específica.

Algumas das funções matemáticas disponíveis em **math.h:**

Função	Descrição	Exemplo
ceil (x)	arredonda o valor de x “pra cima”	ceil (9.2) é 10.0 ceil(-9.7) é -9.0
cos (x)	cosseno de x (x em radianos)	cos (0.0) é 1.0
exp (x)	função exponencial e	exp (1.0) é 2.71727
fabs (x)	valor absoluto de x	fabs(-7.76) é 7.76
floor (x)	arredonda x “pra baixo”	floor (9 .2) é 9 .0 floor(-9.7) é -10.0
fmod(x, y)	resto de x/y como número de ponto flutuante	fmod(13.0, 2.0) é 1 fmod(13.0, 2.1) é 0.4
log (x)	logaritmo natural de x (base e)	log (2.717272) é 1.0
pow(x, y)	x elevado à potência de y	pow(2, 7) é 128
sin (x)	seno de x (x em radianos)	sin (0.0) é 0
sqrt(x)	raiz quadrada de x	sqrt(900.0) é 30.0
tan (x)	tangente de x (x em radianos)	tan (0.0) é 0

Conversão de tipos (Type Casting)

As funções da biblioteca **math.h** geralmente esperam receber um argumento do tipo **double**. Mesmo se passamos uma variável do tipo **int** para uma função dessa biblioteca **math.h**, ela funcionará de forma correta. Isso ocorre por causa da conversão de tipo.

Conversão de tipo permite converter uma variável de um tipo para outro.

Esse procedimento pode ser feito de forma explícita utilizando o operador de conversão de tipo:

(type_name) expression;

Por exemplo:

c = (float) a/b;

A operação de conversão de tipos tem precedência sobre a divisão: primeiro vai ser feita a conversão e depois a divisão.

Exemplo 3: Conversão explicita

```c
#include <stdio.h>

int main()
{
    int a, b;
    double c, d;

    a = 7;
    b = 2;

    c = a/b;
    printf("\n Divisão inteira: %i / %i = %lf", a, b, c);

    d = (double) a / b;
    printf("\n Divisão com conversão de tipo: %i / %i = %lf \n\n", a, b, d);

    return 0;
}
```

Divisão inteira: 7 / 2 = 3.000000
Divisão com conversão de tipo: 7 / 2 = 3.500000

A conversão de tipo pode acontecer de forma implícita, isto é, feita automaticamente pelo próprio compilador.

Boa prática de programação:

Efetuar a conversão de tipos de forma explicita se a tal conversão for necessária.

Normalmente na conversão de tipos o compilador faz a “promoção” do tipo menos preciso para o tipo mais preciso, de acordo com a seguinte sequência:

char → int →
unsigned int →
unsigned long →
long long →
unsigned long long →
float →
double →
long double

Quando variáveis de tipos diferentes participam da expressão, o compilador converte automaticamente a variável do tipo mais baixo (mais impreciso) para o próximo tipo (mais preciso), até colocar todas as variáveis no mesmo nível (o mais alto entre os participantes da expressão).

Exemplo 4: Conversão automática (char to int)

```c
#include <stdio.h>

int main()
{
  int  i = 17;
  char c = 'c'; /* ascii value is 99 */
  int sum_i;
  float sum_f;

  sum_i = i + c;
  printf("Soma (int) = %d \n", sum_i );

  sum_f = i + c;
  printf("Soma (float) = %f \n\n", sum_f );

  return 0;
}
```

```
Soma (int) = 116
Soma (float) = 116.000000
```

Exemplo 5: Conversão automática (int to double)

```c
#include <stdio.h>
#include <math.h>

//=================================================================
int main ()
{
   /* variaveis locais */
   int i;

   printf("\n O programa retorna n * n * n, [0..10] \n\n");

   for(i = 0; i<=10; i++)
   {
     printf("\n %i elevado ao cubo = %.0f \n",i, pow(i, 3));
   }

   return 0;
}
```

A conversão automática não acontece para operadores de atribuição e operadores lógicos (&& e ||).

Os valores limites para cada tipo de dados podem variar. O exemplo a seguir mostra como consultar os limites dos tipos de dados específicos.

Exemplo 6: Valores limites

```c
#include <stdio.h>
#include <float.h>
#include <limits.h>

int main()
{
   printf("\t\tUsing <limits.h> library definitions...\n");

   printf("\n CHAR \n");
   printf("Storage size (bytes): %lu \n", sizeof(char));
   printf("signed char min: %d\n", SCHAR_MIN);
   printf("signed char max: %d\n", SCHAR_MAX);
   printf("unsigned char max: %u\n", UCHAR_MAX); // Note use of u, formatting output

   printf("\n SHORT \n");
   printf("Storage size (bytes): %lu \n", sizeof(short));
   printf("signed short min: %d\n", SHRT_MIN);
   printf("signed short max: %d\n", SHRT_MAX);
   printf("unsigned short max: %ud\n", USHRT_MAX);

   printf("\n INT \n");
   printf("Storage size (bytes): %lu \n", sizeof(int));
   printf("signed int min: %d\n", INT_MIN);
   printf("signed int max: %d\n", INT_MAX);
   printf("unsigned int max: %u\n", UINT_MAX);

   printf("\n LONG \n");
   printf("Storage size (bytes): %lu \n", sizeof(long));
   printf("signed long min: %ld \n", LONG_MIN);
   printf("signed long max: %ld \n", LONG_MAX);
   printf("unsigned long max: %lu\n", ULONG_MAX);

   printf("\n FLOAT \n");
   printf("Storage size (bytes): %lu \n", sizeof(float));
   printf("signed float min: %e\n", FLT_MIN);
   printf("signed float max: %e\n", FLT_MAX);

   printf("\n DOUBLE \n");
   printf("Storage size  (bytes): %lu \n", sizeof(double));
   printf("signed double min: %e\n", DBL_MIN);
   printf("signed double max: %e\n", DBL_MAX);

   return 0;
}
```

Localização

Adaptação do programa às características de um determinado idioma ou de uma região é chamado de localização.

A linguagem C utiliza a biblioteca **locale.h** para implementar a localização de programas.

O programa pode ser localizado para português utilizando função **setlocale()**:

```
setlocale(LC_ALL, “Portuguese”);
```

Onde LC_ALL faz referência à todos os aspectos da localização.

Exemplo 7: Localização

```c
#include <stdio.h>
#include <locale.h> //necessário para usar setlocale

int main(void)
{
  setlocale(LC_ALL, "Portuguese");
  printf("Localização do programa: impressão de caracteres e acentuação \n\n");

  return 0;
}
```

```
Localização do programa: impressão de caracteres e acentuação
```

VARIÁVEIS E FUNÇÕES

Variáveis locais e globais

O escopo (ou a área de visibilidade) em programação é a região onde uma variável é declarada e existe. Fora dessa região essa variável se encontra inacessível.

Em C/C++ uma variável pode ser declarada:

1. Dentro de uma função ou bloco – são chamadas variáveis locais.
2. Fora de qualquer uma das funções – são chamadas variáveis globais.
3. Na declaração dos parâmetros de uma função – parâmetros formais.

Variáveis locais

Variáveis locais são aquelas declaradas dentro de um bloco ou função.

Elas podem ser usadas somente dentro do bloco/função onde foram declaradas.

Essas variáveis podem ser acessadas somente pela própria função e são inacessíveis para outras funções.

Variáveis globais

Variáveis globais são declaradas fora das funções, geralmente no início do programa.

Essas variáveis preservarão seus valores ao longo da execução do programa e poderão ser acessadas pelas funções do programa.

Em um programa podem existir uma variável global e uma variável local com o mesmo nome. Nesse caso vai ser usada a variável local (declarada dentro da função em questão).

Parâmetros formais

Parâmetros da função ou parâmetros formais são tratados como variáveis locais e terão preferência dentro da função em questão.

Exemplo 8: variáveis locais vs globais

```c
#include <stdio.h>

int a = 20; /* variavel global */
int sum(int a, int b);

int main ()
{
  /* variavais locais em função main */
  int a = 10;
  int b = 5;
  int c = 0;

  printf ("\n Valor de a em main() = %d \n\n",  a);
  c = sum( a, b);
  printf ("\n Valor de c= a+b em main() = %d \n\n",  c);

  return 0;
}
//=======================================================
/* soma de dois inteiros */
int sum(int a, int b)
{
    printf("\n____________ Função sum (inicio):_____________________ ");
    printf ("\n Valor de a em sum() = %d\n",  a);
    printf ("\n Valor de b em sum() = %d\n",  b);
    printf("\n____________ Função sum (fim):_____________________ \n\n");

    return a + b;
}
```

Valor de a em main() = 10

____________ Função sum (inicio):_____________________
Valor de a em sum() = 10
Valor de b em sum() = 5
____________ Função sum (fim):_____________________

Valor de c= a+b em main() = 15

Inicialização das variáveis locais e globais.

Quando uma variável local é declarada ela não recebe nenhum valor inicial – a atribuição de valor inicial é responsabilidade do programador.

Variáveis globais recebem valores iniciais de forma automática de acordo com a tabela:

Tipo	Valor inicial
int	**0**
char	**'\0'**
float	**0**
double	**0**
pointer	**NULL**

Exemplo 9: Inicialização de variáveis: variáveis locais vs variáveis globais

```c
#include <stdio.h>

int a; /* = 0, por ser uma variavel global */
void f(int b);

int main ()
{
  /* variavais locais em função main */
  int b; /* = ??, por ser local */

  printf ("\n Valor de a em main() = %d \n\n",  a);
  printf ("\n Valor de b em main() = %d \n\n",  b);

  f(b);

  return 0;
}
//=======================================================
void f(int b)
{
   int c; /* = ??, por ser local */
   printf("\n___________ Função f (inicio):_________________ ");
   printf ("\n Valor de a em f() = %d\n",  a);
   printf ("\n Valor de b em f() = %d\n",  b);
   printf ("\n Valor de c em f() = %d\n",  c);
   printf("\n___________ Função f (fim):____________________ \n\n");

}
```

Valor de a em main() = 0
Valor de b em main() = 0

___________ Função f (inicio):_____________
Valor de a em f() = 0
Valor de b em f() = 0
Valor de c em f() = 32534
___________ Função f (fim):_____________

Classes (tipos) de alocação (armazenamento)

A classe de alocação (armazenamento) da memória define a “visibilidade” e tempo de vida das variáveis e/ou funções.

Os especificadores que definem a classe de armazenamento devem ser colocados antes das variáveis/funções aos quais eles se aplicam.

Em C/C++ existem as seguintes classes de memória:

- **auto**
- **register**
- **static**
- **extern**

Classe de alocação auto

É a classe usada por definição para todas as variáveis locais.

Exemplo: Declaração de variável (dentro de bloco/função)

Declaração da variável local	Expressão equivalente
{ **int m;** **}**	**{** **auto int m;** **}**

A classe **auto** pode ser aplicada somente dentro das funções, i.e. para variáveis locais.

Classe de alocação register

Essa classe é usada para indicar que as variáveis locais devem ser armazenadas em registros e não em memória RAM.

Isso significa que o tamanho máximo permitido para variável é definido pelo tamanho do registro (normalmente **word**) e a operação unária “&” não poderá ser aplicada a essa variável (porque ela não se encontra na memória).

Exemplo:

```
{
        register int  counter;
}
```

Essa classe é recomendada para variáveis que precisam ser acessadas rapidamente e várias vezes, como contadores por exemplo, com intuito de aumento de desempenho.

Porém esse especificador não garante que a variável necessariamente será armazenada num registro.

Na verdade ele significa que a variável poderá ser armazenada num registro. Se isso acontecerá ou não vai depender das características particulares do software e do hardware.

Classe de alocação static

Essa classe indica para o compilador que a variável local deve existir durante a execução de programa inteiro, pois, por definição, o compilador cria uma variável local toda vez que entra numa função, e a variável é automaticamente destruída na saída da função.

Exemplo:

```
static int i;
```

Uma variável desse tipo vai preservar seu valor entre as chamadas de função.

Se esse especificador for aplicado a uma variável global isso restringirá sua existência ao arquivo no qual ela foi declarada.

Classe de alocação extern

Esse especificador é usado (automaticamente) para variáveis/funções globais visíveis (acessíveis) para todos os arquivos do programa.

Quando esse especificador é usado a variável não poderá ser inicializada, pois ela somente serve de referência à uma variável global.

Esse especificador é comumente usado quando dois ou mais arquivos compartilham uma variável/função.

Exemplo:

main.c

```
#include <stdio.h>
#include "support.c"

int counter;

extern void write_extern();

int main()
{
  counter = 5;
  write_extern();

  return 0;
}
```

support.c

```
#include <stdio.h>


extern int counter;

void write_extern(void)
{
  printf("count is %d\n", counter);
}
```

Geração de números aleatórios

A geração de números aleatórios pode ser feita usando a função **rand()**, de acordo com a fórmula:

n = a + rand() % b;

onde

- **a** define o deslocamento, i.e. o inicio do intervalo
- **b** define a “largura” do intervalo

Exemplo 10: Geração de números aleatórios no intervalo [1,6]

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Windows: Sleep()
// #include <windows.h>
// #include <unistd.h>

int main ()
{
  unsigned int seed;
  int i, j, n, r;

  printf("\n Geração de numeros aleatorios de 1 até 6");
  printf("\n ________ função RAND (numeros pseudo aleatorios)__________ \n");
  for(i = 1; i <= 10; i++)
  {
    n = 1 + rand() % 6;
    printf("%i  ", n);
  }
 //-------------------------------------------------------------------------
  printf("\n ________ função SRAND (numeros aleatorios)__________ \n");
  seed = 1;
  while (seed !=0)
  {
   printf("\n Digite um número inteiro ou 0 para sair ");
   scanf("%u", &seed);

   if(seed != 0 )
   {
     srand(seed);

     for(i = 1; i <= 10; i++)
      {
       n = 1 + rand() % 6;
       printf("%i  ", n);
      }
   }
  }
```

```c
//-------------------------------------------------------------------------------
  printf("\n ________ função SRAND + TIME (numeros aleatorios)__________ \n");
  printf("\n Digite quantas sequencias de números aleatorios devem ser geradas ");
  scanf("%d", &r);

  for (j = 1; j<=r; j++ )
  {
   printf("\n Sequencia %i \n ", j);
   srand(time(NULL));
  // getchar();

   for(i = 1; i <= 10; i++)
    {
      n = 1 + rand() % 6;
      printf("%i  ", n);
    }
    sleep(2); // esperar por 2 segundos

  //  Sleep(2000); // Windows
  // getchar();

   }

  return 0;
}
```