

Funções

Exercício:

Escreva um programa que faça uso da função `EDivisivel(int a, int b)`, criada anteriormente. Além desta função o programa deverá usar: `Max(int a, int b)`, que retorna o maior dos parâmetros; `VMedio(int a, int b)`, que retorna o valor médio (inteiro) entre `a` e `b`, você deverá escrever estas funções. Organize o seu programa em três arquivos: o arquivo `exercicio.c`, conterá o programa principal que deve se utilizar das funções descritas; o arquivo `func.c` conterá o corpo das funções; o arquivo `func.h` conterá os protótipos das funções. Compile os arquivos e gere o executável.

```
/*conteúdo do arquivo func.h*/
```

```
int EDivisivel(int a, int b);
```

```
int Max(int a, int b);
```

```
int VMedio(int a, int b);
```

```
/*conteúdo do arquivo exercicio.c*/
#include <stdio.h>
#include "func.h"
main()
{
    int a,b,opcao;
    printf ("\n\nEntre com o valor inteiro para \"a\": ");
    scanf ("%d",&a);
    printf ("\nEntre com o valor inteiro para \"b\": ");
    scanf ("%d",&b);
    do
    {
        printf ("\n\nOpcoes:\n1-Para saber se a eh %s", "divisivel por b");
        printf ("\n2-Para saber qual o maior valor\n");
        printf ("3-Para saber o valor medio\n");
        printf ("4-Para sair\nEntre com sua opcao: ");
        scanf ("%d",&opcao);
    }
```

```
switch (opcao)
```

```
{
```

```
    case 1: if(EDivisivel(a,b))
```

```
        printf("\n\"a\" eh divisivel por \"b\");
```

```
    else
```

```
        printf("\n\"a\" nao eh divisivel por \"b\");
```

```
        break;
```

```
    case 2: printf("\n0 maior valor eh %d",Max(a,b));
```

```
        break;
```

```
    case 3: printf("\n0 valor medio eh %d",VMedio(a,b));
```

```
        break;
```

```
    case 4: break;
```

```
    default: printf("\n0opcao invalida!");
```

```
}
```

```
}while(opcao!=4);
```

```
}
```

```
/*conteudo do arquivo func.c*/
```

```
int EDivisivel(int a, int b)
```

```
{  
    return(a%b?0:1);
```

```
}  
int Max(int a, int b)
```

```
{  
    return(a>b?a:b);
```

```
}  
int VMedio(int a, int b)
```

```
{  
    return((a+b)/2);
```

```
}
```

Funções

linha de comando para compilação no gcc:

gcc exercicio.c func.c -o exercicio

Argumentos argc e argv

Funções

- Os Argumentos argc e argv

A função **main()** pode ter parâmetros formais. Mas, o programador não pode escolher quais serão eles.

A declaração mais completa que se pode ter para a função **main()** é:

```
int main (int argc, char *argv[]);
```

Funções

Os parâmetros **argc** e **argv** dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O **argc** (argument count) é um inteiro e possui o número de argumentos com os quais a função **main()** foi chamada na linha de comando.

O **argv** (argument values) é um vetor de strings. Cada string deste vetor é um dos parâmetros da linha de comando. **argc** é utilizado para sabermos quantos elementos temos em **argv**.

Funções

- Os Argumentos *argc* e *argv* (continuação)

Exemplo: O programa a seguir faz uso dos parâmetros *argv* e *argc*. O programa recebe da linha de comando o dia, mês e ano correntes, e imprime a data em formato apropriado. Veja o exemplo, supondo que o executável se chame “data”:

```
data 19 04 06
```

O programa imprimirá:

```
19 de abril de 2006
```

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int mes;
    char *nomemes [] = {"Janeiro", "Fevereiro", "Março",
    "Abril", "Maio", "Junho", "Julho", "Agosto", "Setembro",
    "Outubro", "Novembro", "Dezembro"};
    if(argc == 4)
    {
        mes = atoi(argv[2]); /*poderia ter usado sscanf()*/
        if (mes<1 || mes>12)
            printf("Erro!\nMes invalido!");
        else
            printf("\n%s de %s de 20%s", argv[1],
            nomemes[mes-1], argv[3]);
    }
    else
        printf("Erro!\nUso: data dia mes ano, todos inteiros");
    return 0;
}
```

Funções

Exercício:

Construa um programa que receba da linha de comando, com a qual o programa é executado, um número natural, e retorne seu fatorial na saída padrão.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int num, fat=1;
    if(argc == 2 && atoi(argv[1])>=0)
    {
        for (sscanf(argv[1], "%d", &num); num>1; num-- )
            fat*=num;
        printf ("\n0 fatorial de %s eh %d\n", argv[1], fat);
    }
    else
        printf("Erro!\nUso: exercicio numero, %s",
            "numero dever representar um natural");
    return 0;
}
```

Recursividade

Recursividade

Alguns problemas são definidos com base nos mesmos, ou seja, podem ser descritos por instâncias do próprio problema.

Para tratar estas classes de problemas, utiliza-se o conceito de recursividade.

Uma função recursiva é uma função que em sua seção de comandos chama a si mesma.

Uma grande vantagem da recursividade é o fato de gerar uma redução no tamanho do algoritmo (programa), permitindo descrevê-lo de forma mais clara e concisa.

Recursividade

Porém, todo cuidado é pouco ao se fazer funções recursivas. A primeira coisa a se providenciar é um critério de parada, o qual vai determinar quando a função deverá parar de chamar a si mesma. Este cuidado impede que a função se chame infinitas vezes.

Um exemplo de um problema passível de definição recursiva é a operação de multiplicação efetuada sobre números naturais. Podemos definir a multiplicação em termos da operação mais simples de adição.

Recursividade

No caso

$$A * B$$

pode ser definido como

$$A + A * (B - 1)$$

precisamos agora especificar um critério de parada. Qual seria?

$$A * 0 = 0$$

Recursividade

De acordo com o que vimos até o momento, podemos definir um laço de repetição que implementaria o cálculo da operação de multiplicação entre valores naturais com base na operação de adição. Por exemplo:

```
...  
int A, B, RES;  
...  
RES=0;  
while (B<>0)  
{  
    RES = RES + A;  
    B = B - 1;  
}  
...
```

Recursividade

Com base no que vimos podemos, também, definir uma função recursiva que implemente a operação de multiplicação com base na operação de adição:

```
int multiplicar (int A, int B)
{
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
```

Recursividade

Para uma melhor compreensão do que foi apresentado, devemos compreender o conceito de “registro de ativação”.

O registro de ativação é uma área de memória que guarda informações referentes ao estado atual de uma função ou do próprio programa:

- valor dos parâmetros (para funções);
- valor das variáveis locais (para funções);
- valor do contador de programa (Program Counter - PC);
- etc.

Recursividade

Sempre que uma função é chamada o registro de ativação de quem a invocou (da função principal, de uma outra função ou da própria função) é salvo e um novo registro de ativação é criado para a função invocada. Estes registros de ativação são empilhados em uma pilha de registros de ativação.

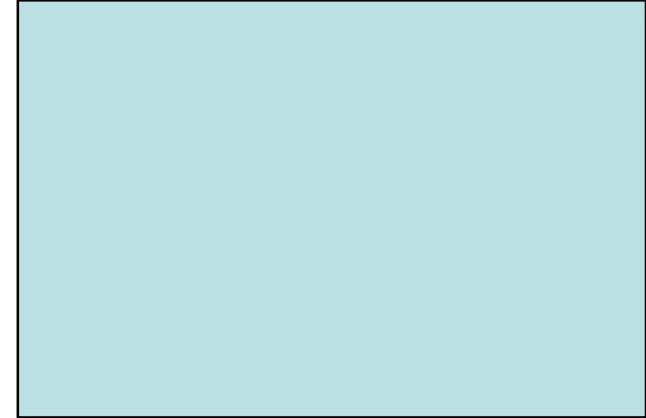
Este processo é conhecido como salvamento e troca de contexto e pode ser melhor compreendido se o aplicarmos sobre um programa que se utilize da função recursiva “multiplicar” definida anteriormente.

```
#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n", A, B, RES);
}
```

```

#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n", A,B,RES);
}

```



Pilha de registros
de ativação

Para melhor contextualizar nossa explicação vamos presumir que o usuário fornecerá o valor 7 para "A" e o valor 3 para "B".

```

#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n", A,B,RES);
}

```

PC = 1 ...

Pilha de registros
de ativação

Inicialmente o registro de ativação da função main é colocado na pilha de registros de ativação.

```

#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n", A, B, RES);
}

```



RES = multiplicar (7,3);

PC = 1	A = 7	B = 3
...		
PC = 10		...

**Pilha de registros
de ativação**

Em nosso exemplo a primeira execução da função multiplicar ocorre na décima instrução da seção de comandos da main. Neste momento é salvo o registro de ativação do algoritmo e introduzido na pilha um novo registro de ativação referente à função chamada.

```

#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n",A,B,RES);
}

```

7 + multiplicar (7,2)
 ?
 RES = multiplicar (7,3);
 ?

PC = 1	A = 7	B = 2
...		
PC = 4	A = 7	B = 3
...		
PC = 9		...

Pilha de registros de ativação

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo um registro de ativação (RA) do invocador e introduzido na pilha um novo RA.



```

#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf ("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf ("\nMultiplicador (valor natural): ");
    8     scanf ("%d", &B);
    9 }while (B<0);
    10 RES = multiplicar(A,B);
    11 printf ("\n%d * %d = %d\n",A,B,RES);
}

```

?
 7 + multiplicar (7,1)
 ?
 7 + multiplicar (7,2)
 ?
 RES = multiplicar (7,3);

PC = 1	A = 7	B = 1	Pilha de registros de ativação
...			
PC = 4	A = 7	B = 2	
...			
PC = 4	A = 7	B = 3	
...			
PC = 9		...	

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo o RA do invocador e introduzido na pilha um novo RA.



Recursividade

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo o RA do invocador e introduzido na pilha um novo RA.

Devido ao valor contido no parâmetro B a segunda instrução da seção de comandos da função multiplicar é executada retornando o valor zero e finalizando as chamadas recursivas. Neste é desempilhado um RA e o contexto do RA da função invocadora é retomado.

PC = 1	A = 7	B = 0
...		
PC = 4	A = 7	B = 1
...		
PC = 4	A = 7	B = 2
...		
PC = 4	A = 7	B = 3
...		
PC = 9		...

0
?
7 + multiplicar (7,0)
?
7 + multiplicar (7,1)
?
7 + multiplicar (7,2)
?
RES = multiplicar (7,3);

Pilha de registros
de ativação

Recursividade

Desta forma uma a uma as camadas às funções vão sendo finalizadas e seus registros de ativação desempilhados.

RES = multiplicar (7,3);

RES = 21

?

7 + multiplicar (7,2)

7 + 14

?

7 + multiplicar (7,1)

7 + 7

?

7 + multiplicar (7,0)

7 + 0

0

PC = 4	A = 7	B = 1
...		
PC = 4	A = 7	B = 2
...		
PC = 4	A = 7	B = 3
...		
PC = 9		...

**Pilha de registros
de ativação**

Recursividade

Com base no que foi exposto, podemos visualizar algumas desvantagens da utilização de recursividade, como:

- O consumo de memória necessário para a troca de contexto.
- Redução do desempenho de execução devido ao tempo para gerenciamento de chamadas.
- Dificuldades na depuração de programas recursivos, especialmente se a recursão for muito profunda.

Exercício: Para uma melhor compreensão do conceito de recursividade faça agora uma função recursiva para calcular o fatorial de um número natural e construa um programa que se utilize de forma adequada da função em questão.

```
#include <stdio.h>
int fatorial (int num)
{
    if (!num)
        return (1);
    else
        return (num * fatorial(num-1));
}
int main()
{
    int n;
    printf ("Digite o numero que voce deseja saber o fatorial: ");
    scanf ("%d", &n);
    if (n>=0)
        printf ("\nO fatorial do numero %d eh %d",n,fatorial(n));
    else
        printf ("\nNao existe fatorial de numeros negativos!");
}
```

Recursividade

Exercício:

Para uma melhor compreensão do conceito de recursividade construa, na linguagem de programação C, uma função recursiva que receba como parâmetro dados referentes a um vetor com elementos inteiros e inverta a ordem de seus elementos.