

Cap. VI – SUBPROGRAMAÇÃO

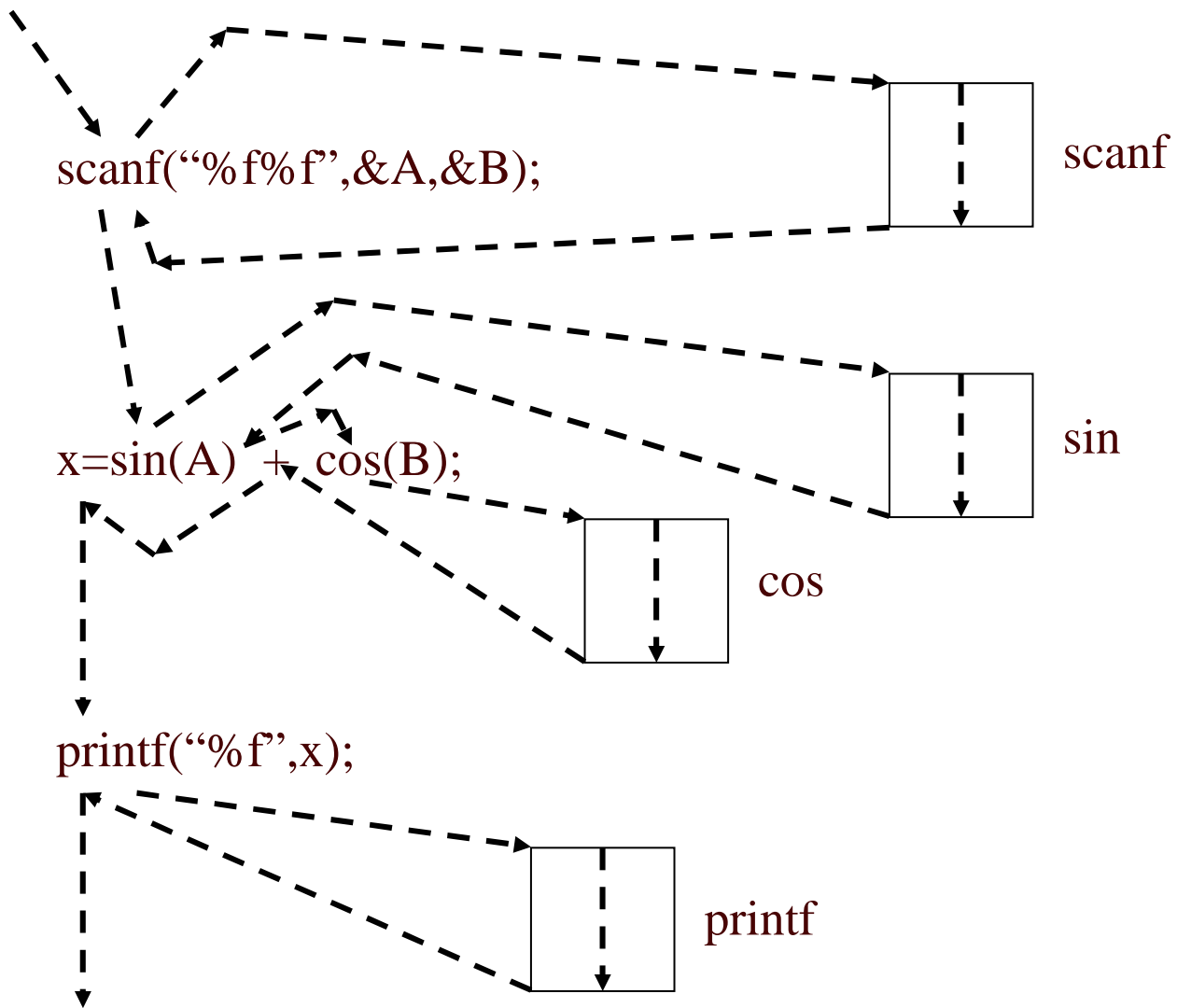
6.1 – Conceitos Básicos

6.1.1 - Definição de subprograma

- **Subprograma:** programa ao qual está associado um nome (identificador), cuja execução está subordinada a um outro programa.
- Subprogramas podem receber os seguintes nomes:
 - Função, Subrotina, Rotina, Procedimento, Segmento, Módulo, etc.
- Em C o único tipo de subprograma é a Função.
- Até agora já foram vistas funções da biblioteca da Linguagem C:
 - printf, scanf, pow, sin, cos, isdigit, etc.
- As funções necessárias a um programa, que não constam dessa biblioteca, têm de ser programadas.

6.1.2 – Fluxo de controle de um programa com subprogramas

- Seja o seguinte código:



6.1.3 – Definição de *function* em C

- Estrutura geral de um programa (sugerido na pg 2.14):

- Declarações globais
 - Subprogramas
 - Programa principal

} Funções

- Forma geral de uma função:

Tipo Nome (Lista de parâmetros)
{ *Declarações Comandos* }

- **Cabeçalho da função:** fora das chaves ‘{ }’.
Corpo da função: o que está dentro das chaves.
- **Exemplo 6.1:** função fatorial.

```
long fat (int n) {  
    int i; long fat;  
    if (n < 0 || n > 12) fat = -1;  
    else  
        for (i = 2, fat = 1; i <= n; i++)  
            fat *= i;  
    return fat;  
}
```

- O tipo da função pode ser:
 - um tipo primitivo (**int**, **float**, **char**, etc),
 - um tipo enumerativo,
 - uma estrutura (**struct**, **union**),
 - um ponteiro (próximo capítulo).
- Não pode ser uma variável indexada.

6.1.4 – Retorno de uma função

- **Retorno natural:** quando a chave de fechamento ‘}’ da função for encontrada.
- **Retorno explícito:** execução do comando *return*; pode haver mais de um numa função.
- O comando *return* pode ou não incluir uma expressão:

return; **return ++a;** **return (a * b);**

- Comando *return* com expressão: seu valor é enviado ao programa que chamou a função.
- Toda função cujo valor é usado no programa que a chamou deve conter pelo menos um *return* com expressão.

- O tipo de uma função que não retorna valor pode ser ***void***;
Exemplo:

```
void prt_ender() {  
    printf (“Rua Alfa 387, Centro, São Jose, SP”);  
}
```

É uma função que realiza uma tarefa, mas não produz um valor a ser usado no programa que a chamou.

- Em outras linguagens, subprogramas que não retornam valor são diferentes de funções (***Function***):
 - ***Procedure*** em **Pascal**
 - ***Subroutine*** em **Fortran**

6.2 – Utilidades da Subprogramação

6.2.1 – Evitar a repetição de código

- **Exemplo 6.2:** Cálculo do número de combinações de ***m*** elementos tomados ***n*** a ***n***:

Fórmula:
$$C_n^m = \frac{m!}{(m - n)! * n!} \quad (3 \text{ cálculos de fatorial})$$

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

```
long fat (int n) {  
    int i; long f;  
    if (n < 0 || n > 12) f = -1;  
    else  
        for (i = 2, f = 1; i <= n; i++)  
            f *= i;  
    return f;  
}
```

```
void main ( ) {  
    char c; int m, n; long comb;  
    do {  
        printf ("Combinacao de m elementos tomados n a n? (s/n): ");  
        do scanf ("%c", &c); while (c!='s' && c!='n');  
        if (c == 's') {  
            printf ("\n\tm: "); scanf ("%d", &m);  
            printf ("\tn: "); scanf ("%d", &n);  
            if (m <= 0 || m > 12 || n <= 0 || m < n)  
                printf ("\n\tDados incompativeis\n\n");  
            else {  
                comb = fat (m) / (fat (m - n) * fat (n));  
                printf ("\n\tNum. de combinacoes: %ld\n\n", comb);  
            }  
        }  
    } while (c == 's');  
}
```

6.2.2 – Refinamento passo a passo

- **Exemplo 6.3:** Análise de textos

- Ler um texto de um arquivo, verificar o número de ocorrências de cada palavra do texto e gerar uma tabela com essas palavras em ordem alfabética ao lado de seus respectivos números de ocorrências.
- Por exemplo, o texto: *voce pode enganar algumas pessoas em todo o tempo e todas as pessoas em algum tempo mas não pode enganar todas as pessoas em todo o tempo* deve gerar a seguinte tabela:

| palavra | nº de ocorr |
|---------|-------------|
| algum | 1 |
| algumas | 1 |
| as | 2 |
| e | 1 |
| em | 3 |
| enganar | 2 |
| mas | 1 |
| não | 1 |
| o | 2 |
| pessoas | 3 |
| pode | 2 |
| tempo | 3 |
| todas | 2 |
| todo | 2 |

| | |
|------|---|
| voce | 1 |
|------|---|

/* Diretivas de preprocessamento */

```
#include <stdio.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define N1 31
#define N2 50
```

/* Declaracoes de tipos e variaveis globais */

```
typedef char cadeia[N1];
```

```
struct entr_tab {
    cadeia nome;
    int n_ocorr;
};
```

```
typedef struct entr_tab entr_tab;
```

```
FILE *fl;
entr_tab tabela[N2];
int ntab;
```


/*

Funcao procura: procura uma palavra numa tabela de palavras

Se a procura tiver sucesso, retorna a posicao da palavra na tabela. Caso contrario, retorna a posicao em que deveria estar somada de 10 e tudo multiplicado por -1

*/

```
int procura (cadeia palavra) {  
    int inf, sup, med, posic; char achou;  
  
    achou = FALSE; inf = 0; sup = ntab-1;  
    while (!achou && sup >= inf) {  
        med = (inf + sup) / 2;  
        if (strcmp (palavra, tabela[med].nome) == 0)  
            achou = TRUE;  
        else if (strcmp (palavra, tabela[med].nome) < 0)  
            sup = med - 1;  
        else inf = med + 1;  
    }  
    if (achou) posic = med;  
    else posic = - (10 + inf);  
    return posic;  
}
```

```
/*
```

```
Funcao inserir: insere uma palavra numa determinada  
posicao da tabela de palavras e inicializa com 1 o  
seu numero de ocorrencias
```

```
*/
```

```
void inserir (cadeia palavra, int posic) {  
    int i;  
    ntab++;  
    for (i = ntab - 1; i > posic; i--)  
        tabela[i] = tabela[i-1];  
    strcpy (tabela[posic].nome, palavra);  
    tabela[posic].n_ocorr = 1;  
}
```

```
/*
```

```
Funcao exibir_tabela: exhibe na tela a tabela de palavras
```

```
*/
```

```
void exibir_tabela () {  
    int i;  
    printf ("%10s%-20s|%17s\n    ",  
            " ", "Palavra", "Num. de ocorr.");  
    for (i = 1; i <= 47; i++) printf("-");  
    for (i = 0; i < ntab; i++)  
        printf ("\n%10s%-20s|%10d",  
                " ", tabela[i].nome, tabela[i].n_ocorr);  
}
```

```
/*  
    Funcao main: funcao principal da analise de texto  
*/  
  
void main ( ) {  
    cadeia palavra;  
    int posic;  
    ntab = 0;  
    fl = fopen ("infile.cpp", "r");  
    while (fscanf(fl, "%s", palavra) == 1) {  
        posic = procura (palavra);  
        if (posic >= 0)  
            tabela[posic].n_ocorr++;  
        else  
            inserir (palavra, - (10 + posic));  
    }  
    exhibir_tabela ();  
}
```

6.2.3 – Outras utilidades

- Modularização: isolar trabalhos específicos em módulos próprios
 - cálculo de fatorial
 - ordenação de um vetor
 - procura de um elemento em um vetor
 - inserção de um elemento em um vetor
 - etc..
- Reaproveitamento de código: uma mesma função pode ser utilizada em vários programas
- Por exemplo, uma função para resolver um sistema de equações lineares pode ter diversas aplicações:
 - resolver vários sistemas
 - resolver sistemas de equações não lineares
 - fazer ajuste de curvas
 - auxiliar a resolução de equações diferenciais parciais, etc.

6.3 – Localização de Variáveis

6.3.1 – Variáveis locais e globais

- **Variável local:** qualquer variável declarada no corpo de uma função é *local* a essa função.
- **Variável global:** é uma variável declarada fora do escopo de qualquer função.
- **Exemplo 6.4:** seja o programa.

```
#include <stdio.h>

int a = 33;

void main ( ){
    int b = 77;
    printf (“a = %d;\nb = %d;”, a, b);
}
```

– a é global e b é local à função *main*.

6.3.2 - Blocos

- Bloco: é um comando composto contendo declarações de variáveis no início.
- Uma declaração só é válida dentro do bloco em que ela é feita.
- **Exemplo 6.5:** aninhamento de blocos.

```
#include <stdio.h>
void main ( ) {
    int a = 2;
    printf ("a = %d;\n", a);
    {
        int a = 5;
        printf ("a = %d;\n", a);
    }
    printf("a = %d;\n", ++a);
}
```

Resultado:

```
a=2;
a=5;
a=3;
```

- Principal razão para o uso de blocos: reserva local na memória para variáveis, só quando necessário.
- **Blocos paralelos:** 2 blocos que residem num terceiro bloco, um seguido do outro, nos quais as variáveis declaradas em qualquer um deles não é reconhecida dentro do outro.
- Funções são declaradas em paralelo.

6.3.3 – Variáveis automáticas

- Até agora, todas as variáveis locais dos programas vistos neste curso são *automáticas*.
- Variáveis não automáticas são estudadas ainda neste capítulo.
- **Variável automática:** é aquela para a qual o sistema reserva memória no momento em que seu bloco de definição (ou declaração) começa a ser executado.
- Tal variável perde a reserva quando a execução de tal bloco se encerra.
- Se o bloco voltar a ser executado, nova reserva é feita, mas o valor da variável na execução anterior é perdido.
- Pode-se usar a palavra **auto** para explicitar esta classe; por ‘default’ toda variável local é automática. Então:

int a,b; equivale a
auto int a,b;

- **Exemplo 6.6:** quais as variáveis que estão no ar.

```

#include <stdio.h>
int a = 1;
void ff ( ) {long c, x;
4 printf ("444: a = %5d; c = %5ld; x = %5ld;\n", a, c, x);
  c = 11; x = 22; a = 33;
5 printf ("555: a = %5d; c = %5ld; x = %5ld;\n", a, c, x);
}
void main ( ) {int b; long c;    /* Bloco 1 */ 0
  printf ("000: a = %5d; b = %5d; c = %5ld;\n", a, b, c);
  b = 7; c = 9;
1 printf ("111: a = %5d; b = %5d; c = %5ld;\n", a, b, c);
  {
    /* Bloco 2 */
    int a;
2 printf ("222: a = %5d; b = %5d; c = %5ld;\n", a, b, c);
    a = 5;
3 printf ("333: a = %5d; b = %5d; c = %5ld;\n", a, b, c);
    ff ( );
  }
6 printf ("666: a = %5d; b = %5d; c = %5ld;\n", a, b, c);
}

```

| | | glob. | bloco 1 | | bloco 2 | ff () | |
|--------|--------------------|-------|---------|---|---------|--------|-----|
| Pontos | Blocos no ar | a | b | c | a | c | x |
| 0 | b 1 | 1 | ? | ? | ### | ### | ### |
| 1 | b 1 | 1 | 7 | 9 | ### | ### | ### |
| 2 | b 1 – b 2 | 1 | 7 | 9 | ? | ### | ### |
| 3 | b 1 – b 2 | 1 | 7 | 9 | 5 | ### | ### |
| 4 | b 1 – b 2 – ff () | 1 | 7 | 9 | 5 | ? | ? |
| 5 | b 1 – b 2 – ff () | 33 | 7 | 9 | 5 | 11 | 22 |
| 6 | b 1 | 33 | 7 | 9 | ### | ### | ### |

6.4 – Passagem de Parâmetros

6.4.1 – Importância do uso de parâmetros

- É comum subprogramas atuarem sobre um determinado valor ou uma determinada variável para produzir um resultado ou realizar uma tarefa; exemplos:
 - Calcular o fatorial do valor guardado numa variável;
 - Ordenar os elementos de um vetor;
 - Trocar os valores de duas variáveis entre si.
- Poder-se-ia usar variáveis globais para esse fim;
 - O subprograma calcularia o fatorial de uma variável global
 - Ou ordenaria os elementos de um vetor global
 - Ou trocava entre si os valores de duas variáveis globais.
- Mas, e se forem muitas as variáveis ou as expressões das quais se quer calcular o fatorial, ou muitos os vetores a serem ordenados, ou muitos os pares de variáveis a trocarem entre si seus valores?
 - Antes de cada chamada do subprograma, a(s) variável(eis) global(is) sobre a(s) qual(ais) ele atua deveria(m) ser carregada(s) com o(s) valor(es) ou variável(eis) com a(s) qual(is) se deseja trabalhar.

- Para evitar isso, usa-se parâmetros e argumentos.
- **Argumento:** variável ou expressão alvo de uma chamada do subprograma.
- **Parâmetro:** variável local do subprograma destinada a receber informações sobre o **argumento** correspondente.
- **Exemplo 6.7:** argumentos e parâmetros do programa do Exemplo 6.2:

São três chamadas da função *fat*: *fat* (*m*), *fat* (*n*) e *fat* (*m-n*)

Os argumentos são respectivamente **m**, **n** e **m-n**.
O parâmetro da função *fat* é **n**.

A cada chamada da função *fat*, o parâmetro **n** recebe o valor do argumento dessa chamada e a função calcula e devolve à função *main* o valor do fatorial de **n**.

6.4.2 – Modos de passagem de parâmetros

- As linguagens de programação mais conhecidas apresentam dois importantes modos de passagens de parâmetros:
 - Passagem por valor e
 - Passagem por referência.

- **Passagem por valor:** o valor do argumento é carregado no parâmetro; em geral o argumento é uma expressão.
- **Passagem por referência:** o parâmetro é alocado coincidindo com o endereço do argumento; nesse modo, o argumento deve ser uma variável.
- **Exemplo 6.8:** Passagem de parâmetros em **Pascal**:

```
program ppppp ;
var a, b: integer;
```

```
procedure xxxxx (x: integer; var y: integer);
begin x := x + 3; y := y + x; end;
```

```
begin
  a := 10; b := 20; xxxxx (a, b);
  write ("a = ", a, "; b = ", b);
end
```

Resultado: a = 10; b = 33

- **x** é parâmetro por valor
- **y** é parâmetro por referência
- **x** recebe o valor de **a**
- **y** é alocada coincidindo com **b**

- A Linguagem C só trabalha com passagem de parâmetros por **valor**.
- A passagem de parâmetros por **referência** é simulada por argumentos que são **endereços** de outras variáveis.

6.4.3 – Passagem por valor em C

- Chamada de uma função: seu nome e uma lista de argumentos entre parênteses.
- Os argumentos devem casar em número e compatibilidade de tipos com os parâmetros na lista de definição da função.
- Cada argumento é calculado e seu valor é atribuído ao parâmetro correspondente na função.
- Se um argumento é apenas o nome de uma variável, apenas o seu valor é transmitido à função.
- Depois da execução da função, o valor dessa variável não muda.
- **Exemplo 6.9:** seja o programa.

```
#include <stdio.h>
void ff (int a) {a += 1; printf (“Durante, a = %d\n”, a);}

void main ( ) {
    int a = 5;
    printf (“Antes, a = %d\n”, a); ff(a);
    printf (“Depois, a = %d\n”, a);
}
```

Só o valor de a foi enviado.

Resultado:

Antes, a = 5
Durante, a =
6

6.4.4 – Passagem por referência em C

- Às vezes, é desejável que a variável argumento conserve a mudança sofrida dentro da função.
- Ao invés de passar o seu **valor**, passa-se o seu **endereço**.
- **Exemplo 6.10:** Seja o programa:

```
#include <stdio.h>
void trocar (int *p, int *q) {
    int aux;
    aux = *p; *p = *q; *q = aux;
}
void main ( ) {
    int i = 3, j = 8;
    printf ("Antes, i = %d; j = %d\n", i, j);
    trocar (&i, &j);
    printf ("Depois, i = %d; j = %d\n", i, j);
}
```

Resultado:

Antes, i = 3; j = 8

Depois, i = 8; j = 3

- trocar (&i, &j) envia os endereços de i, j para p, q.
- *p = *q significa: palavra da memória cujo endereço está em p recebe conteúdo da palavra da memória cujo endereço está em q.
- **int *p** significa: a palavra da memória cujo endereço está em p é do tipo **int**.

6.5 – Funções como Parâmetros e Argumentos

- No exemplo da pg. 22, no Capítulo I, o programa calcula a integral somente para a função $f(x) = \log_{10}x + 5$.
- Pode-se fazer uma função que calcule integrais, na qual a função a ser integrada é um parâmetro.
- **Exemplo 6.11:** integrais de várias funções:

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

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

```
/* Declarações das 3 funções a serem integradas:
```

```
    log10(x)+5,    x**3 - x,    sen(x) + 2                                */
```

```
double f1 (double x) {  
    return (log10(x) + 5);  
}
```

```
double f2 (double x) {  
    return (pow(x,3) - x);  
}
```

```
double f3 (double x) {  
    return (sin(x) + 2);  
}
```

```

/*          Declaracao da funcao integradora          */

double integral (double f (double),float a,
                float b, float p) {

    double s1, s2, si, dx;
    long n;
    int i;

    s2 = 0; n = 5;
    do {
        s1 = s2; n = 2*n;
        dx = (b-a)/n;
        s2 = 0;
        for (i = 1 ; i <= n; i = i+1) {
            si = dx * (f(a+(i-1)*dx)+f(a+i*dx)) / 2;
            s2 = s2 + si;
        }
    } while (fabs(s1 - s2) >= p);
    return s2;
}

```


/* Programa principal

*/

```
void main () {  
    float a, b, p;  
    printf ("Integral de log10(x):\n\n\tlimite inferior: ");  
    scanf ("%f", &a);  
    printf ("\tlimite superior: "); scanf ("%f", &b);  
    printf ("\tprecisao: ");scanf ("%f", &p);  
    printf ("\n\tValor da integral: %lf\n\n",  
            integral (f1, a, b, p));  
    printf ("Integral de x**3 - x:\n\n\tlimite inferior: ");  
    scanf ("%f", &a);  
    printf ("\tlimite superior: "); scanf ("%f", &b);  
    printf ("\tprecisao: ");scanf ("%f", &p);  
    printf ("\n\tValor da integral: %lf\n\n",  
            integral (f2, a, b, p));  
    printf ("Integral de sen(x)+2:\n\n\tlimite inferior: ");  
    scanf ("%f", &a);  
    printf ("\tlimite superior: "); scanf ("%f", &b);  
    printf ("\tprecisao: ");scanf ("%f", &p);  
    printf ("\n\tValor da integral: %lf\n\n",  
            integral (f3, a, b, p));  
}
```

6.6 – Recursividade

- Uma função é recursiva se ela chama a si própria direta ou indiretamente.
- **Exemplo 6.10:** Cálculo de fatoriais

– Definição recursiva:

$$\text{Fat}(n) = \begin{cases} 1, & \text{se } n = 0 \text{ ou } 1 \\ n * \text{Fat}(n - 1), & \text{se } n > 1 \end{cases}$$

– Programa:

```
#include <stdio.h>
long fat (int n) {
    long f;
    if (n < 0) f = -1;
    else if (n <= 1) f = 1;
    else f = n * fat(n - 1);
    return f;
}
void main ( ) {
    char c; int n;
    do {
        printf ("Calculo de fatorial? (s/n): ");
        do scanf ("%c", &c); while (c != 's' && c != 'n');
        if (c == 's') {
            printf ("\n\tNumero: "); scanf ("%d", &n);
            printf ("\tFatorial: %ld\n\n", fat(n));
        }
    } while (c == 's');
}
```

- **Exemplo 6.13:** MDC entre dois números.

- Definição recursiva:

$$\text{mdc}(m,n) = \begin{cases} m, & \text{se } n = 0 \\ \text{mdc}(n, m \% n) & \text{se } n > 0 \end{cases}$$

- Programa:

```
#include <stdio.h>
int mdc (int m, int n) {
    int r;
    if (m < 0) m = -m;
    if (n < 0) n = -n;
    if (n == 0) r = m;
    else r = mdc (n, m % n);
    return r;
}
void main ( ) {
    char c; int m, n;
    do {
        printf ("MDC? (s/n): ");
        do scanf ("%c", &c); while (c != 's' && c != 'n');
        if (c == 's') {
            printf ("\n\tm: "); scanf ("%d", &m);
            printf ("\n\tn: "); scanf ("%d", &n);
            printf ("\tMDC(%d,%d): %d\n\n", m, n, mdc (m,
n));
        }
    } while (c == 's');
}
```

- **Exemplo 6.14:** Procura binária de números

Procura (Num, Vet, inf, sup) =

$\left\{ \begin{array}{l} \text{– Falso, se } \text{Num} < \text{Vet}[\text{inf}] \text{ ou} \\ \quad \text{se } \text{Num} > \text{Vet}[\text{sup}] \\ \\ \text{– Verdadeiro, se } \text{Num} == \text{Vet}[\text{medio}] \\ \\ \text{– Procura (Num, Vet, inf, medio - 1),} \\ \quad \text{se } \text{Num} < \text{Vet}[\text{medio}] \\ \\ \text{– Procura (Num, Vet, medio + 1, sup),} \\ \quad \text{se } \text{Num} > \text{Vet}[\text{medio}] \end{array} \right.$

/* Diretivas de pre-processamento */

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define N 50
```

```
/*      Funcao procura (num, vet, inf, sup)

        Procura 'num' no vetor 'vet', no intervalo
        [inf, sup] de seus indices.
        Retorna TRUE ou FALSE.                                */
```

```
char procura (int num, int vet[N], int inf, int sup) {
    int med, p;
    if (num < vet[inf] || num > vet[sup])
        p = FALSE;
    else {
        med = (inf + sup) / 2;
        if (num == vet[med]) p = TRUE;
        else if (num < vet[med])
            p = procura (num, vet, inf, med-1);
        else
            p = procura (num, vet, med+1, sup);
    }
    return p;
}
```

```

/*          Programa principal          */

void main () {
    int n, i, num; char c, okrel; int vetor[N];

/*  Leitura e validacao da relacao de numeros  */

    printf ("Numero de elementos: "); scanf ("%d",&n);
    printf ("\nNumeros:\n");
    for (i = 0; i < n; scanf("%d", &vetor[i]), i++);
    printf ("\n\nRelacao de numeros:\n\n");
    for (i = 0; i < n; printf("%4d", vetor[i]), i++);
    for (i = 0, okrel = TRUE; i < n - 1 && okrel; i++)
        if (vetor[i] > vetor[i+1]) okrel = FALSE;
    if (!okrel) printf ("\n\n\tRelacao desordenada\n");

/*  Procura de numeros na relacao  */

    else {
        printf ("\n\n");
        do {
            printf ("Procura? (s/n): ");
            do scanf("%c", &c); while (c != 's' && c != 'n');
            if (c == 's') {
                printf ("\n\tNumero: "); scanf ("%d", &num);
                if (procura (num, vetor, 0, n-1))
                    printf ("\t%d estah na relacao\n\n", num);
                else
                    printf ("\t%d nao estah na relacao\n\n", num);
            }
        } while (c == 's');
    }
}

```

6.7 – Protótipos de Funções

- Funções devem ser declaradas antes de serem usadas.
- Até agora, funções têm sido declaradas e definidas num só trecho de programa.
- Protótipos são declarações de funções feitas antes de sua definição.
- Eles permitem que funções sejam usadas antes de sua definição.
- O protótipo de uma função especifica o tipo da função e o tipo de seus parâmetros.
- Nos exemplos vistos neste capítulo, poderiam ter sido usados os seguintes protótipos:

- **long** fat (**int**);
- **int** procura (cadeia);
- **void** inserir (cadeia, **int**);
- **void** exibir_tabela (**void**);
- **void** ff (**void**);
- **void** ff (**int**);
- **void** trocar (**int***, **int***);
- **double** integral (**double**(*)(**double**), **float**, **float**, **float**);
- **int** mdc (**int**, **int**);
- **char** procura (**int**, **int**[], **int**, **int**)

Escrevendo os protótipos no início do programa a função *main* pode ser a primeira a ser definida. Facilita a metodologia *top-*

- **Exemplo 6.15:** análise de textos do exemplo 6.3 da pg 129

```
/*          Diretivas de processamento          */
-----

/*          Declaracoes de tipos e variaveis globais          */
-----

/*          Protótipos das funções auxiliares          */

int procura (cadeia);
void inserir (cadeia, int);
void exhibir_tabela (void);

/*          Programa principal          */

void main ( ) { ----- }

/*          Definicoes das funcoes auxiliares          */

int procura (cadeia palavra) { ----- }

void inserir (cadeia palavra, int posic) { ----- }

void exhibir_tabela ( ) { ----- }
```


6.8 – Classes de Armazenamento

- Toda variável e função em C tem dois atributos:
 - ‘tipo’ e ‘classe de armazenamento’
- Há quatro classes de armazenamento:
 - automática, externa, registrador e estática.

correspondentes às declarações:

- **auto**, **extern**, **register** e **static**.
- As variáveis automáticas só ocupam espaço na memória quando seu bloco de declaração está no ar. Elas já foram vistas na seção 6.3.3.

6.8.1 – Variáveis externas

- Toda variável declarada fora de qualquer função (global) é externa.
- Elas ocupam espaço na memória durante toda a execução do programa.
- A palavra **extern** pode ser usada em sua declaração, mas em alguns casos é dispensável: no exemplo 6.3,

int ntab; poderia ser escrito assim: **extern int** ntab;

- Principal uso da palavra ***extern***: em programas divididos em arquivos.
- **Exemplo 6.16:** seja um programa escrito em dois arquivos, file1.c e file2.c

file1.c:

```
#include <stdio.h>
#include "file2.c"
int a = 1, b = 2, c = 3;
void main ( ) {
    printf ("%3d\n", f( ));
    printf ("%3d%3d%3d\n", a, b, c);
}
```

Resultado:

```
12
 4 2 3
```

file2.c:

```
int f ( ) {
    extern int a;
    int b, c;
    a = b = c = 4;
    return (a + b + c);
}
```

Nesse exemplo, os dois arquivos podem ser compilados separadamente.

- A habilidade em compilar arquivos separadamente é importante ao se escrever grandes programas:
 - Armazena-se um grande programa em diversos arquivos;
 - Cada arquivo pode conter uma ou mais funções, ou somente declarações globais, ou somente protótipos de funções, etc.;
 - Ocorrendo erros de compilação, só os arquivos que os contém precisam ser compilados;
 - Os arquivos corretos são dispensados de nova compilação; se forem numerosos, ganha-se muito tempo com essa dispensa.
- Dois meios de comunicação entre funções:
 - variáveis externas (globais)
 - parâmetros.
- O uso de parâmetros, salvo exceções, é o meio preferido.
 - coopera para a modularidade e portabilidade das funções
 - reduz a possibilidade de efeitos colaterais indesejáveis.

- **Portabilidade:**

- Um rádio de carro teria má portabilidade se, ao retirá-lo de um carro para colocá-lo em outro, fosse necessário desconectar muitos cabos elétricos e conectá-los no outro carro.
- Uma função com variáveis globais disponível para muitos usuários, exige de quem vai utilizá-la, conhecimento dessas variáveis e adição das mesmas ao conjunto de variáveis globais de seu programa.

6.8.2 – Variáveis em registradores

- Registradores são memórias bem mais rápidas, sofisticadas e caras do que a memória principal.
- A declaração `register int i;`
 - diz que a variável inteira **i** deve, se possível, ser alocada num registrador;
 - a manipulação dessa variável fica muito mais rápida e eficiente
- O número de registradores é bem menor que o número de palavras da memória principal, logo essa alocação pode não ser possível.
- Tal variável é “desalocada” do registrador quando o bloco de declaração sair do ar.

- Variáveis intensamente acessadas são fortes candidatas à classe *register*; exemplo: índices de comandos **for**.

6.8.3 – Variáveis estáticas

- **Variáveis estáticas:** variáveis locais a funções que não são desalocadas quando suas funções saem do ar.
- Conservam seu valor entre uma chamada da função e outra.
- **Exemplo 6.17:** seja o programa

```
#include <stdio.h>
int i;
void f ( ) {
    static int a = 0; int b = 5;
    printf ("a = %3d; b = %3d;", a, b);
    b = i + 10;
    printf (" b = %3d;\n", b);
    a += 3;
}
void main ( ) {
    for (i = 1; i <= 10; i++) f ( );
}
```

Resultado:

```
a =  0; b =  5; b = 11;
a =  3; b =  5; b = 12;
a =  6; b =  5; b = 13;
a =  9; b =  5; b = 14;
a = 12; b =  5; b = 15;
a = 15; b =  5; b = 16;
a = 18; b =  5; b = 17;
a = 21; b =  5; b = 18;
a = 24; b =  5; b = 19;
a = 27; b =  5; b = 20;
```

- A declaração de uma variável estática dentro de uma função torna seu uso privativo àquela função.

