

17 Funções e Ponteiros

Ronaldo F. Hashimoto e Carlos H. Morimoto

Nessa aula introduzimos o mecanismo utilizado pelo C para permitir que funções devolvam mais de um valor, que é realizado através de variáveis do tipo ponteiros (ou apontadores). Para isso, vamos primeiramente descrever como a informação é armazenada na memória do computador, para definir o que é um ponteiro e sua utilidade.

Ao final dessa aula você deverá ser capaz de:

- Descrever como a informação é armazenada na memória do computador, e a diferença entre conteúdo e endereço de uma posição de memória.
- Definir o que é um ponteiro (ou apontador), e para que ele é utilizado.
- Declarar ponteiros.
- Utilizar ponteiros como parâmetros de funções e em programas.

17.1 Memória

Podemos dizer que a memória (RAM) de um computador pode ser constituída por **gavetas**. Para fins didáticos, considere um computador imaginário que contenha na memória 100 (cem) gavetas (veja na Figura 1 uma idéia do que pode ser uma memória de um computador).

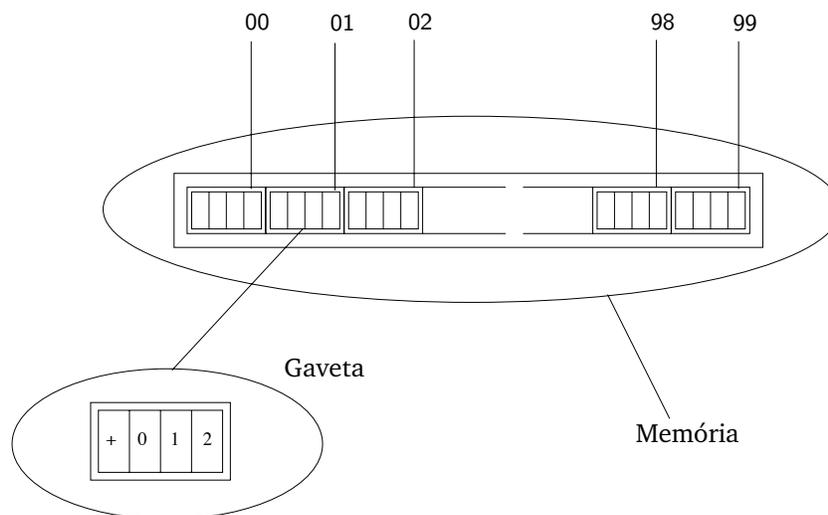


Figura 1: Memória de um Computador

Cada gaveta é dividida em regiões. Por questões didáticas, vamos considerar que cada gaveta tem 4 (quatro) regiões. Na região mais à esquerda pode-se colocar um sinal '+' ou '-'. Nas demais quatro regiões, dígitos de 0 a 9. Assim, em uma gaveta, podemos guardar números de -999 a +999.

Em computadores reais, cada gaveta tem 8 (oito) regiões (estas regiões são chamadas de *bits*). O bit mais à esquerda é conhecido como **bit de sinal**: 0 (zero) para números positivos e 1 (um) para números negativos. Estes 8 bits compõem o que chamamos de *byte*. Assim uma gaveta tem tamanho de 1 (um) byte. Dessa forma, uma gaveta guarda um número (que em computadores reais é uma sequência de 8 zeros e uns).

Agora, o mais importante: para cada gaveta na memória é associada uma identificação numérica (no exemplo da Fig. 1, esta identificação corresponde à numeração das gavetas de 00 a 99, mas em computadores reais, o último

número pode ser muito maior que 99). Estes números identificam as gavetas e são chamados de **endereços**. Assim, cada gaveta tem um endereço associado indicando a posição onde ela se encontra na memória.

17.2 Declaração de Variáveis

Quando se declara uma variável, é necessário reservar espaço na memória que seja suficiente para armazená-la, o que pode corresponder a várias gavetas dependendo do tipo da variável. Para variáveis inteiras são reservadas 4 gavetas (bytes). Para **float** e **double** são reservadas 4 e 8 bytes, respectivamente.

Dessa forma, as seguintes declarações

```
int n;  
float y;  
double x;
```

reservam 4, 4 e 8 bytes para as variáveis *n*, *y* e *x*, respectivamente.

Agora, vamos considerar a variável inteira *n*. Esta variável tem então 4 gavetas com endereços, digamos, 20, 21, 22 e 23 (veja a Fig. 2).

Considerando a região mais à esquerda como um sinal '+' ou '-' e nas demais 15 (quinze) regiões, dígitos de 0 a 9, essa variável inteira *n* no computador imaginário pode armazenar qualquer número inteiro que está no intervalo de -999.999.999.999.999 a +999.999.999.999.999. Em um computador real, a variável *n* pode armazenar qualquer número inteiro de -2.147.483.648 a +2.147.483.647.

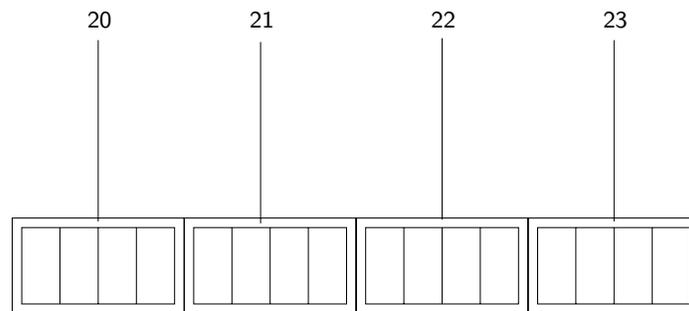


Figura 2: Quatro gavetas para a variável *n*

Agora, outra coisa importantíssima: toda variável é associada a um endereço. Para a variável *n*, apesar de ter 4 gavetas, ela é associada somente a um endereço. Vamos supor que seja o menor deles, ou seja, neste caso, o endereço 20.

Bem, por que estamos falando dessas coisas de **endereço de variáveis** se a matéria desta aula é **ponteiros**? É porque um **ponteiro** (ou **apontador**) é um tipo de variável que consegue guardar **endereço de variável**. Assim, é possível guardar o endereço da variável *n* em uma variável do tipo **ponteiro**.

Vamos aqui fazer uma suposição de que temos uma variável **ponteiro** *apt*. Legal, esta variável pode guardar endereços de variáveis. Então, como fazer com que a variável *apt* guarde o endereço de *n*? Em C, o seguinte comando

```
apt = &n;
```

faz com que a variável *apt* guarde o endereço de *n*. O operador & antes do nome da variável significa "endereço de". Assim, o comando de atribuição acima coloca em *apt* o "endereço de" de *n*.

Se uma variável do tipo **ponteiro** `apt` guarda o endereço da variável `n`, então podemos dizer que a variável `apt` **aponta** para a variável `n`, uma vez que `apt` guarda o local onde `n` está na memória. Você pode estar se perguntando para que serve tudo isso. Observe que ter uma variável que guarda o local de outra variável na memória pode dar um grande poder de programação. Isso nós vamos ver mais adiante ainda nesta aula.

17.3 Declaração de Variável Tipo Ponteiro

Uma outra coisa importante aqui é saber que ponteiros de variáveis inteiras são diferentes para ponteiros para variáveis do tipo **float** e **char**.

Para declarar uma variável ponteiro para **int**:

```
int * ap1;
```

Para declarar uma variável ponteiro para **float**:

```
float * ap2;
```

Para declarar uma variável ponteiro para **char**:

```
char * ap3;
```

As variáveis `ap1`, `ap2` e `ap3` são **ponteiros** (guardam endereços de memória) para **int**, **float** e **char**, respectivamente.

17.4 Uso de Variáveis Tipo Ponteiros

Para usar e manipular variáveis do tipo ponteiros, é necessário usar dois operadores:

- **&** : endereço de
- ***** : vai para

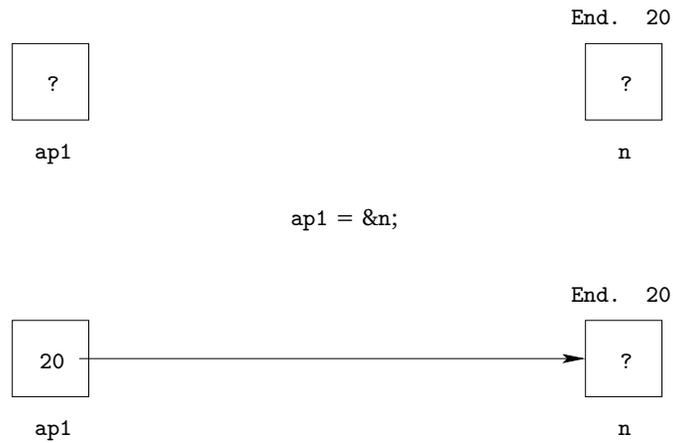


Figura 3: Operador &("endereço de").

17.4.1 Uso do Operador "endereço de"

Para uma variável ponteiro receber o endereço de uma outra variável é necessário usar o operador & ("endereço de"). Observe os seguintes exemplos:

```
ap1 = &n;    /* ap1 recebe o endereço da variável n */
ap2 = &y;    /* ap2 recebe o endereço da variável y */
ap3 = &x;    /* ap3 recebe o endereço da variável x */
```

Observe a Fig. 3. A execução do comando `ap1 = &n;` faz com que a variável `ap1` receba o endereço da variável `n`. Uma vez que a a variável `ap1` guarda o endereço da variável `n` é como se `ap1` estivesse apontando para a variável `n`. Por isso que variáveis que guardam endereços são chamadas de variáveis do tipo ponteiro.

17.4.2 Uso do Operador "vai para"

Uma vez que o ponteiro está apontando para uma variável (ou seja, guardando seu endereço) é possível ter acesso a essa variável usando a variável ponteiro usando o operador * ("vai para"):

```
* ap1 = 10;    /* vai para a gaveta que o ap1 está apontando e guarde 10 nesta gaveta */
* ap2 = -3.0; /* vai para a gaveta que o ap2 está apontando e guarde -3.0 nesta gaveta */
* ap3 = -2.0; /* vai para a gaveta que o ap3 está apontando e guarde -2.0 nesta gaveta */
```

Observe a Fig. 4. A execução do comando `*ap1 = 10;` diz o seguinte: vai para a gaveta que o `ap1` está apontando (ou seja, a variável `n`) e guarde 10 nesta gaveta, ou seja, guarde 10 na variável `n`.

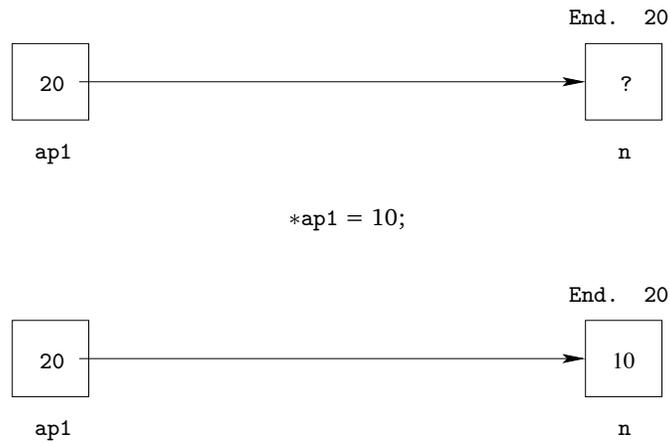


Figura 4: Operador * (“vai para”).

Uma observação importantíssima: o uso do * é diferente quando a gente declara de variável do tipo ponteiro e quando a gente usa como operador “vai para”. Observe com cuidado o seguinte código:

```

int main () {
    int n, m;
    float y;
    char x;

    int * ap1;
    float * ap2;
    char * ap3;

    n = 2; m = 3;
    y = 5.0; x = 's';

    ap1 = &n;
    ap2 = &y;
    ap3 = &x;

    * ap1 = m;
    * ap2 = -5.0;
    * ap3 = 'd';

    printf ("n = %d y = %f x = %c\n", n, y, x);
    return 0;
}

```

int * ap1;
float * ap2;
char * ap3;

Declaração das variáveis ponteiros. Note o uso do asterisco para declarar ponteiros.

ap1 = &n;
ap2 = &y;
ap3 = &x;

Uso de Ponteiros: & significa "endereço de". Note que aqui não tem o asterisco antes da variável ponteiro.

* ap1 = m;
* ap2 = -5.0;
* ap3 = 'd';

Uso de Ponteiros: Note que aqui usa-se o asterisco antes do ponteiro. Este asterisco significa "vai para" a gaveta que o ponteiro está indicando (apontando).

O que o último printf vai imprimir na tela? Se você não entendeu o código acima, digite, compile e execute e veja sua saída.

17.5 Para que serve Variáveis Ponteiros?

Você ainda deve se estar perguntando: para que tudo isso?!? Observe atentamente o seguinte código para fins didáticos:

```

1      # include <stdio.h>
2
3
4      int f (int x, int *y) {
5          *y = x + 3;
6          return *y + 4;
7      }
8
9      int main () {
10         int a, b, c;
11
12         a = 5; b = 3;
13
14         c = f (a, &b);
15
16         printf ("a = %d, b = %d, c = %d\n", a, b, c);
17
18         return 0;
19     }

```

O programa sempre começa a ser executado na função principal.

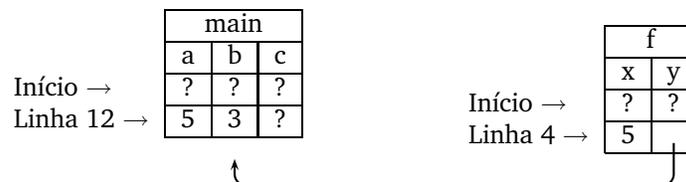
main		
a	b	c
?	?	?
5	3	?

Início →
Linha 12 →

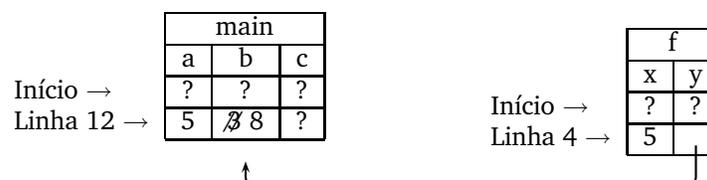
Na Linha 14 é chamada a função *f*. Note a passagem de parâmetros:

- *a* da função *main* → (para) *x* da função *f* e
- *&b* da função *main* → (para) *y* da função *f*

Note que no segundo parâmetro, a variável *y* da função *f* é um ponteiro para *int*. Note que a variável *y* recebe o endereço da variável *b* da função *main*. Dessa forma, a variável *y* da função *f* aponta para a variável *b* da função *main*.



Na Linha 5, estamos usando o ponteiro *y* da função *f* com o operador “vai para”. Ou seja, “vai para” a gaveta em que o *y* está apontando e guarde o resultado da expressão *x+3* neste lugar. Como *y* está apontando para a variável *b* da função *main*, então o resultado da expressão *x+3* será guardado na variável *b* da função *main*. Depois da execução da Linha 5, temos então:



Na Linha 6 temos um **return**. Neste **return**, temos que calcular o valor da expressão `*y+4`. O `*y+4` significa: “vai para” a gaveta em que o `y` está apontando, pegue o valor inteiro que está lá e some com 4 (quatro). Como `y` está apontando para `b` da `main`, então a conta que deve ser feita é: pegue o valor de `b` e some com 4 (quatro). Como `b` vale 8, então o resultado da expressão `*y+4` é 12 (doze). Este resultado é colocado no ACUM.

	ACUM
Início →	?
Linha 6 →	12

e o fluxo do programa volta para onde a função `f` foi chamada, ou seja, na Linha 14. Neste linha, como temos um comando de atribuição para a variável `c`, esta variável recebe o valor que está em ACUM, ou seja, recebe o valor 12.

	main		
	a	b	c
Início →	?	?	?
Linha 12 →	5	8	?
Linha 14 →	5	8	12

No final, o programa imprime na tela o conteúdo das variáveis `a`, `b` e `c` da função `main`:

Tela
a = 5, b = 8, c = 12

Note que na verdade então, o parâmetro `y` da função `f` (que é um ponteiro pois está declarado com um asterisco) aponta para a variável `b` da função `main`, uma vez que na chamada da função `f`, a variável `b` é o segundo parâmetro da função `f`. Dessa forma, podemos dizer que `y` da função `f` aponta para `b` da função `main`:

```

1      # include <stdio.h>
2
3
4      int f (int x, int *y) {
5          *y = x + 3;
6          return *y + 4;
7      }
8
9      int main () {
10         int a, b, c;
11
12         a = 5; b = 3;
13
14         c = f (a, &b);
15
16         printf ("a = %d, b = %d, c = %d\n", a, b, c);
17
18         return 0;
19     }

```

E todas as modificações que fizemos em `*y`, estamos na realidade fazendo na variável `b`.

Legal! E para que serve isso? Os ponteiros como **parâmetros de função** (como o parâmetro `y` da função `f`) servem para **modificar** as variáveis que estão **fora** da função (como a variável `b` da função `main`). E para que serve isso? Considere o seguinte problema:

17.6 Problema

Faça um função que receba como parâmetros de entrada três reais a , ($a \neq 0$), b e c e resolva a equação de 2o. grau $ax^2+bx+c=0$ devolvendo as raízes em dois ponteiros $*x1$ e $*x2$. Esta função ainda deve devolver via **return** o valor -1 se a equação não tem raízes reais, 0 se tem somente uma raiz real e 1 se a equação tem duas raízes reais distintas.

Note que a função pedida deve devolver **dois** valores que são as raízes da equação de 2o. grau $ax^2+bx+c=0$. Mas nós aprendemos que uma função só consegue devolver **um único** valor via **return**. Como então a função vai devolver dois valores? Resposta: usando ponteiros!

```
1      # include <stdio.h>
2      # include <math.h>
3
4
5
6      int segundo_grau (float a, float b, float c, float *x1, float *x2) {
7          float delta = b*b - 4*a*c;
8
9          if (delta < 0) {
10             return -1;
11         }
12
13         *x1 = (-b + sqrt (delta)) / (2 * a);
14         *x2 = (-b - sqrt (delta)) / (2 * a);
15
16         if (delta > 0) {
17             return 1;
18         }
19         else {
20             return 0;
21         }
22     }
23
24     int main () {
25         float a, b, c, r1, r2, tem;
26
27         printf ("Entre com os coeficientes a, b e c: ");
28         scanf ("%f %f %f", &a, &b, &c);
29
30         tem = segundo_grau (a, b, c, &r1, &r2);
31
32
33
34         if (tem < 0) {
35             printf ("Eq. sem raizes reais\n");
36         }
37
38         if (tem == 0) {
39             printf ("Eq. tem somente uma raiz\n");
40             printf ("raiz = %f\n", r1);
41         }
42
43         if (tem > 0) {
44             printf ("Eq. tem duas raizes distintas\n");
45             printf ("Raízes %f e %f\n", r1, r2);
46         }
47
48         return 0;
49     }
```

Note que neste problema, os parâmetros `x1` e `x2` são ponteiros que modificam as variáveis `r1` e `r2` da função `main`. Neste caso então, podemos colocar as soluções da equação de 2o. grau em `*x1` e `*x2` (da forma como foi feita na solução do problema) que estamos na realidade colocando as soluções nas variáveis `r1` e `r2` da função `main`. Faça uma simulação da solução do problema e verifique que as soluções de fato ficam armazenadas nas variáveis `r1` e `r2` da função `main`.

17.7 Função do Tipo void

Funções do tipo `void` correspondem a funções que não retornam um valor via `return`. Veja o seguinte exemplo didático:

Você ainda deve se estar perguntando: para que tudo isso?!? Observe atentamente o seguinte código para fins didáticos:

```
1      # include <stdio.h>
2
3
4      void g (int x, int *y) {
5          *y = x + 3;
6      }
7
8      int main () {
9          int a, b;
10
11
12         a = 5; b = 3;
13
14         g (a, &b);
15
16         printf ("a = %d, b = %d\n", a, b);
17
18         return 0;
19     }
```

Note que a função `g` é do tipo `void`. Ela de fato não tem a palavra `return` no final dela. Diferentemente da função `f` do exemplo anterior.

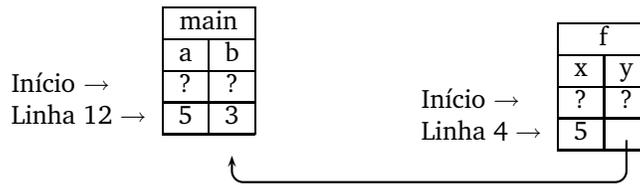
O programa sempre começa a ser executado na função principal.

	main	
Início →	a	b
Linha 12 →	?	?
	5	3

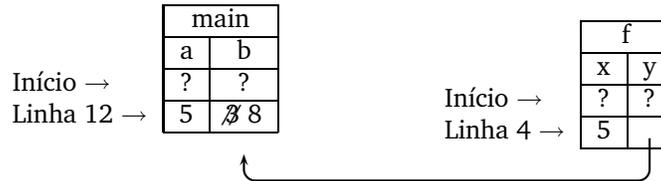
Na Linha 14 é chamada a função `g`. Note a passagem de parâmetros:

- `a` da função `main` → (para) `x` da função `g` e
- `&b` da função `main` → (para) `y` da função `g`

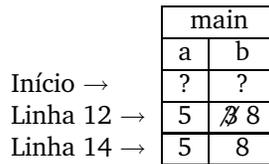
Note que no segundo parâmetro, o parâmetro `y` da função `g` é um ponteiro para `int`. Note que o parâmetro `y` recebe o endereço da variável `b` da função `main`. Dessa forma, o parâmetro `y` da função `f` aponta para a variável `b` da função `main`.



Na Linha 5, estamos usando o ponteiro `y` da função `g` com o operador “vai para”. Ou seja, “vai para” a gaveta em que o `y` está apontando e guarde o resultado da expressão `x+3` neste lugar. Como `y` está apontando para a variável `b` da função `main`, então o resultado da expressão `x+3` será guardado na variável `b` da função `main`. Depois da execução da Linha 5, temos então:



Na Linha 6, acabou a função `g` e fluxo do programa volta para onde a função `g` foi chamada, ou seja, na Linha 14.



No final, o programa imprime na tela o conteúdo das variáveis `a` e `b` da função `main`:

Tela	
a = 5, b = 8	

Então, para que serve uma função `void`? Funções do tipo `void` são úteis quando não necessitamos devolver um valor via `return`. E quando isso acontece?

17.8 Exercícios sugeridos

- (a) Faça uma função que converte uma coordenada cartesiana (x,y) em coordenadas polares (r,s) . Esta função tem duas entradas (via parâmetros) que são os valores `x` e `y` e deve devolver dois valores: as coordenadas polares `r` e `s`.

```

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

void converte_polar (float x, float y, float *apt_r, float * apt_s) {
    float s;

    *apt_r = sqrt (x*x + y*y);

    /* if (x == 0 && y >0) */
    s = 3.1415 / 2;
    if (x == 0 && y < 0)
    s = 3 * 3.1415 / 2;
    if (x == 0 && y == 0)
    s = 0;
    if (x != 0)
    s = atan (y/x);
    if (x < 0)
    s += 3.1415;
    *apt_s = s;
}

```

- (b) Faça um programa que leia um inteiro $n > 0$ e n coordenadas cartesianas reais (x,y) e imprima as respectivas coordenadas polares.

```

int main () {
    float x, y, r, s;
    int i, n;

    printf ("Entre com um inteiro n > 0: ");
    scanf ("%d", &n);

    for (i=0; i<n; i++) {
        printf ("Entre com a coordenada x: ");
        scanf ("%f", &x);

        printf ("Entre com a coordenada y: ");
        scanf ("%f", &y);

        converte_polar (x, y, &r, &s);

        printf ("Coordenada (%f, %f)\n", r, s);
    }

    return 0;
}

```

- (c) Dizemos que um número natural n é palíndromo se lido da direita para a esquerda ou da esquerda para a direita é o mesmo número. Exemplos:

- 567765 e 32423 são palíndromos.
- 567675 não é palíndromo.
- Socorram-me subi no onibus em Marrocos.
- Oto come doce seco de mocotó.
- A diva em Argel alegre-me a vida.

1) Escreva uma função que recebe um inteiro $n > 0$ e devolve o seu primeiro dígito, seu último dígito e altera o valor de n removendo seu primeiro e último dígitos. Exemplos:

valor inicial de n	primeiro dígito	último dígito	valor final de n
732	7	2	3
14738	1	8	473
1010	1	0	1
78	7	8	0
7	7	7	0

2) Escreva um programa que recebe um inteiro positivo n e verifica se n é palíndromo. Suponha que n não contém o dígito 0.

- (d) Escreva um programa que lê dois inteiros positivos m e n e calcula o mínimo múltiplo comum entre m e n . Para isso, escreva primeiro uma função com protótipo:

```
int divide (int *m, int *n, int d);
```

que recebe três inteiros positivos como parâmetros e retorna 1 se d divide pelo menos um entre $*m$ e $*n$, 0 caso contrário. Fora isso, se d divide $*m$, divide $*m$ por d , e o mesmo para $*n$.

- (e) Soma de números binários.

1) Faça uma função com protótipo

```
void somabit (int b1, int b2, int *vaium, int *soma);
```

que recebe três bits (inteiros 0 ou 1) $b1$, $b2$ e $*vaium$ e retorna um bit soma representando a soma dos três e o novo um bit "vai-um" em $*vaium$.

2) Escreva um programa que lê dois números em binário e calcula um número em binário que é a soma dos dois números dados. Utilize a função do item a.