MAC 2166 – Introdução à Computação

POLI - PRIMEIRO SEMESTRE DE 2007

Material Didático

Prof. Ronaldo Fumio Hashimoto

PONTEIROS

Objetivo

Nesta aula vamos falar de:

(1) Memória

(2) Ponteiro como um tipo de variável.

(3) Utilidade de Ponteiros

(4) Ponteiros e Funções

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 Fig. 1 um idéia do que pode ser uma memória de um computador).

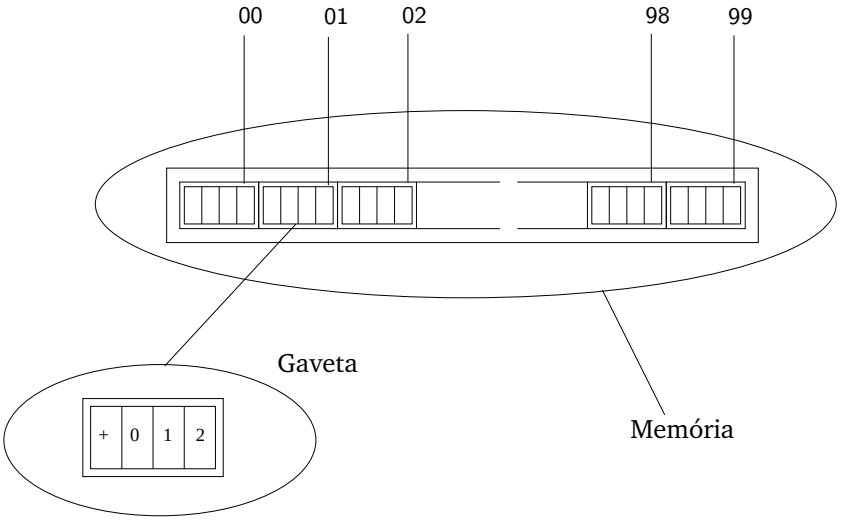


Figura 1: Memória de um Computador

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 seqüência 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 o lugar da onde ela está na memória.

Declaração de Variáveis

Quando se declara alguma variável são reservadas algumas gavetas para essa variável. Para variáveis inteiras são reservadas 4 (quatro) gavetas. Para **float** e **double** são reservadas 4 (quatro) e 8 (oito) gavetas, respectivamente.

Dessa forma, as seguintes declarações

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

reservam 4 (quatro), 4 (quatro) e 8 (oito) gavetas para a variáveis `n`, `y` e `x`, respectivamente.

Agora, vamos considerar a variável inteira `n`. Esta variável tem então 4 (quatro) 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`.

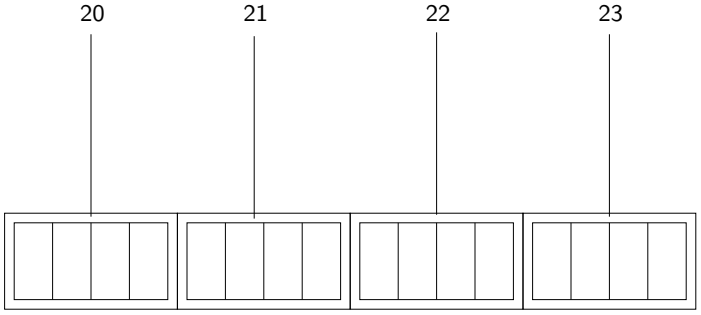


Figura 2: Quatro gavetas para a variável `n`

Agora, outra coisa importantíssima: para variável `n` é também associada um endereço. Apesar de ter 4 gavetas, para a variável `n` é associada somente 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** é um tipo de variável que consegue guardar endereço de variáveis. 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.

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.

Uso de Variáveis Tipo Ponteiros

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

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

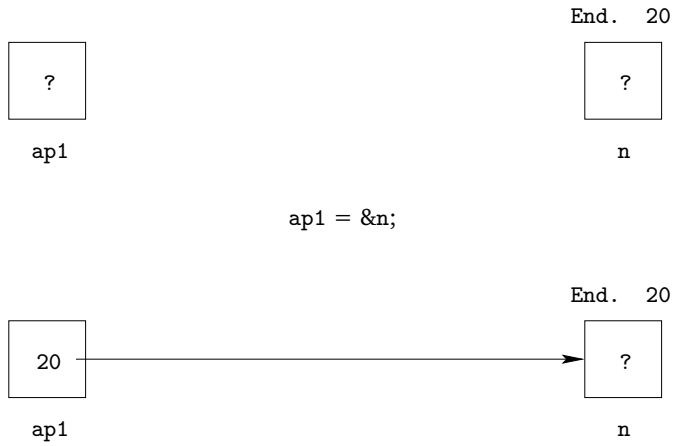


Figura 3: Operador &(“endereço de”).

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.

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`.

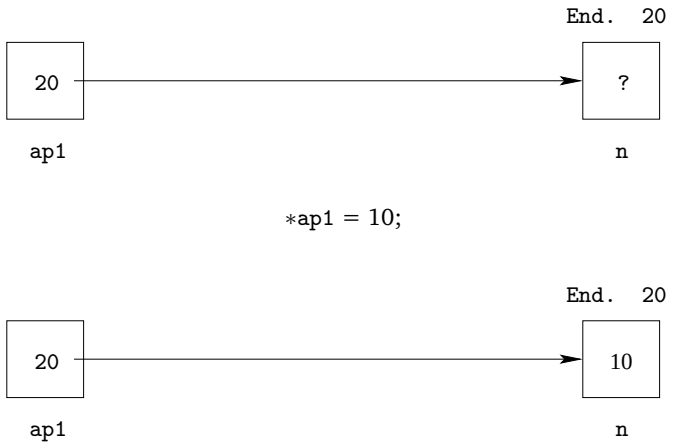


Figura 4: Operador * (“vai para”).

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;
}
```

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

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

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.

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.

Início →
Linha 12 →

main		
a	b	c
?	?	?
5	3	?

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`.

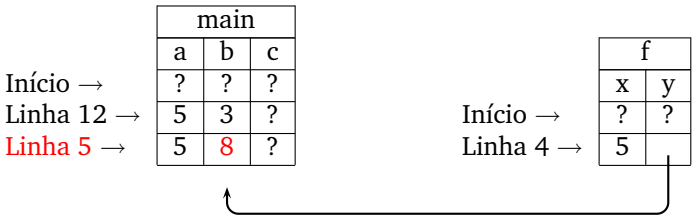
Início →
Linha 12 →

main		
a	b	c
?	?	?
5	3	?

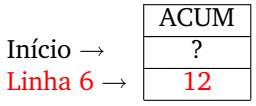
Início →
Linha 4 →

f	
x	y
?	?
5	

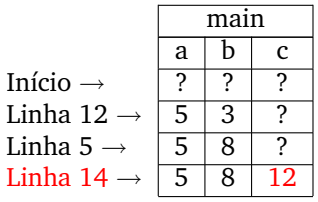
Na Linha 5, estamos usando o ponteiro `y` da função `f` com o operador "vai para". Ou seja, "vai para" a gaveta em que o `y` está apontando e guarde o resultado da expressão `x+3` neste lugar. Como `y` está apontando para a variável `b` da função `main`, então o resultado da expressão `x+3` será guardado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



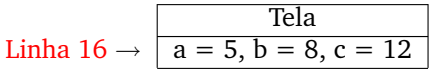
Na Linha 6 temos um **return**. Neste **return**, temos que calcular o valor da expressão `*y+4`. O `*y+4` significa: "vai para" a gaveta em que o `y` está apontando, pegue o valor inteiro que está lá e some com 4 (quatro). Como `y` está apontando para `b` da `main`, então a conta que deve ser feita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



e 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 9.



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



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  }
```

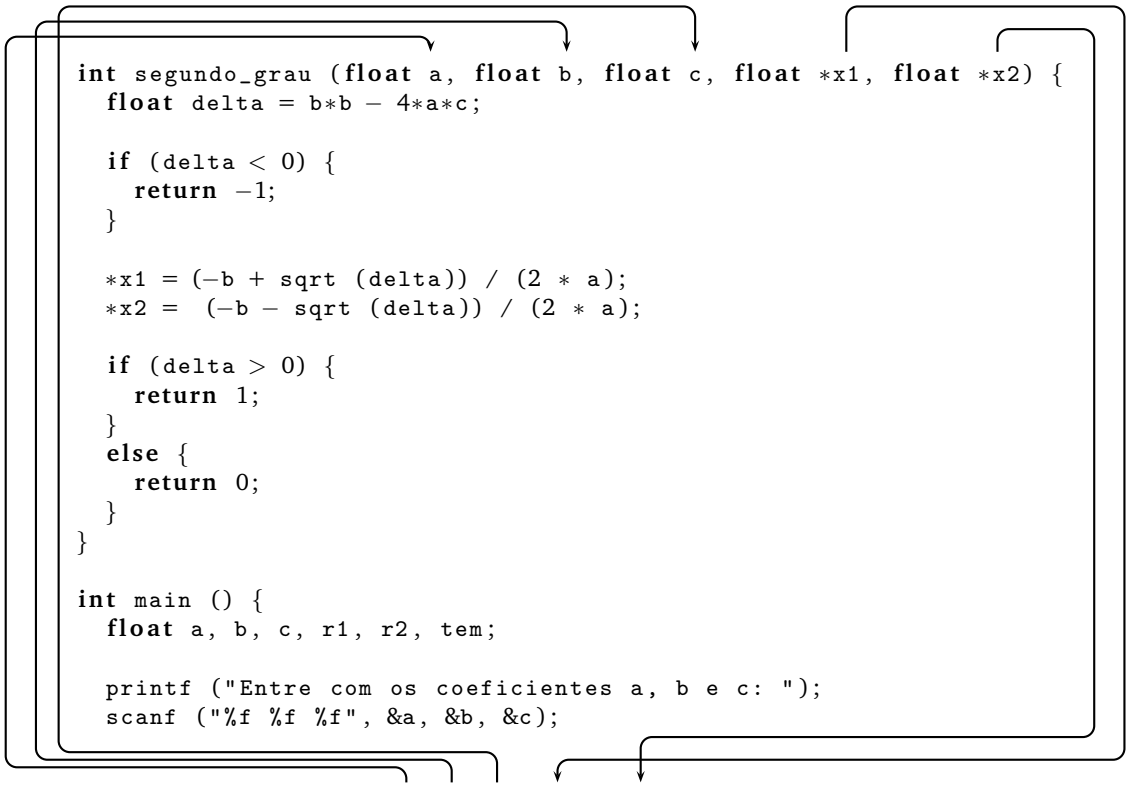
E todas as modificações que fizermos em `*y`, estamos na realidade fazendo na variável `b`.

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:

Problema

Faça um função que recebe como parâmetros de entrada três reais `a`, (`a` $\neq 0$), `b` e `c` e resolve a equação de 2o. grau $\boxed{\texttt{a}x^2+\texttt{b}x+\texttt{c}=0}$ 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 $\boxed{\texttt{a}x^2+\texttt{b}x+\texttt{c}=0}$. 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!



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

int segundo_grau (float a, float b, float c, float *x1, float *x2) {
  float delta = b*b - 4*a*c;

  if (delta < 0) {
    return -1;
  }

  *x1 = (-b + sqrt (delta)) / (2 * a);
  *x2 =  (-b - sqrt (delta)) / (2 * a);

  if (delta > 0) {
    return 1;
  }
  else {
    return 0;
  }
}

int main () {
  float a, b, c, r1, r2, tem;

  printf ("Entre com os coeficientes a, b e c: ");
  scanf ("%f %f %f", &a, &b, &c);


  tem = segundo_grau (a, b, c, &r1, &r2);


  if (tem < 0) {
    printf ("Eq. sem raizes reais\n");
  }

  if (tem == 0) {
    printf ("Eq. tem somente uma raiz\n");
    printf ("raiz = %f\n", r1);
  }

  if (tem > 0) {
    printf ("Eq. tem duas raizes distintas\n");
    printf ("Raízes %f e %f\n", r1, r2);
  }

  return 0;
}
```

Note que neste problema, os parâmetros `x1` e `x2` são ponteiros que modificam as variáveis `r1` e `r2` da funcão `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 funcão `main`. Faça uma simulação da solução do problema e tente enxergar que as soluções de fato ficam armazenadas nas variáveis `r1` e `r2` da funcão `main`.

Função do Tipo void

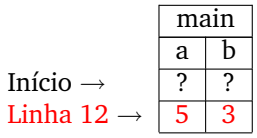
Funções do tipo **void** correspondem a funções que não retornam um valor via **return**. Na realidade, funções do tipo **void** devolvem valores via parâmetros que são ponteiros. Veja o seguinte exemplo didático:

Você ainda deve se estar perguntando: para que tudo isso?!? Observe atentamente o seguinte código para fins didáticos. A função `g` é uma função do tipo **void**:

```
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.

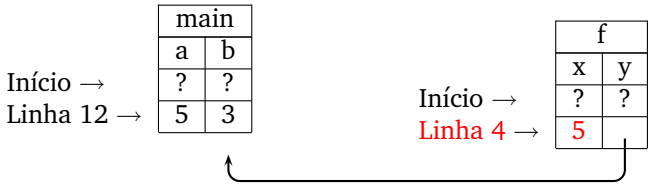
O programa sempre começa a ser executado na função principal.



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

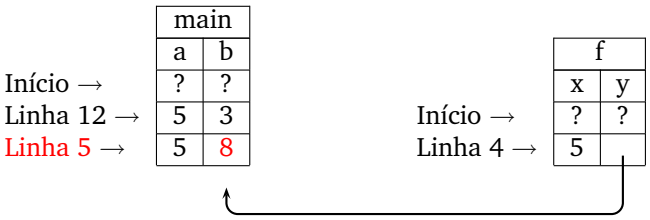
- `a` da função `main` → (para) `x` da função `g` e
- `&b` da função `main` → (para) `y` da função `g`

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.

	main	
	a	b
Início →	?	?
Linha 12 →	5	3
Linha 5 →	5	8
Linha 14 →	5	8

No final, na Linha 16, o programa imprime na tela o contéudo das variáveis `a` e `b` da função `main`:

	Tela
Linha 16 →	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? Veja o problema seguinte que usa uma função do tipo **void**.

Mais um Problema

(a) Faça uma função que converte uma coordenada polar (r,s) em coordenadas cartesianas (x,y). Esta função tem duas entradas (via parâmetros) que são os valores r e s e deve devolver dois valores: as coordenadas cartesianas x e y. Note que, como a função deve colocar os valores das coordenadas cartesianas em x e y, então estas devem ser parâmetros que são ponteiros:

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

void converte (float r, float s, float *ap_x, float *ap_y) {

  *ap_x = r * cos (s);
  *ap_y = r * sin (s);

}
```

Note que a função converte não precisa de um **return**, uma vez que ela muda os valores da função main via dois ponteiros *ap_x e *ap_y.

(b) Faça um programa que leia um inteiro n>0 e n coordenadas polares (r,s) e imprima as respectivas coordenadas cartesianas.

```
int main () {
  float x, y, rr, ss;
  int i, n;

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

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

    printf ("Entre com o angulo s: ");
    scanf ("%f", &ss);

    converte_polar (rr, ss, &x, &y);

    printf ("Cartesiana Cartesiana (%f, %f)\n", x, y);
  }

  return 0;
}
```

Erros Comuns com Funções

A seguir listamos alguns erros comuns cometidos quando se começa a aprender funções. Procure evitá-los:

- **usar** scanf **dentro de funções para ler os parâmetros.**

  Lembre-se que os parâmetros são passados para a função, na hora da sua chamada. Esses valores não devem ser lidos novamente através da scanf.

- **esquecer de colocar '&' na variável na chamada da função quando o parâmetro da função é um ponteiro.**

  Nesse caso, o parâmetro que é um ponteiro deve guardar o endereço da variável, e por isso deve-ser colocar o '&' (operador "endereço de") antes do nome da variável quando se está chamando a função.

- **esquecer de colocar '$*$' na declaração de um parâmetro que deve ser um ponteiro.**

  No caso de se desejar que um parâmetro de uma função seja um ponteiro é necessário colocar um '$*$ antes do nome do parâmetro.

- **esquecer de colocar '$*$' no uso de um parâmetro que é um ponteiro.**

  No uso de um parâmetro que é ponteiro é necessário utilizar o operador $*$ ("vai para") antes do nome do parâmetro para que a variável fora da função seja modificada.

Exercício

(a) Sabemos que um número imaginário (ou complexo) possui uma parte real r e outra imaginária t, e podemos escrever esse número como $c = r + i \cdot t$, onde $i \cdot i = -1$. O quadrado de c pode ser calculado com $c^2 = (r^2 - t^2) + i \cdot (2rt)$. Assim, a variável complexa c^2 tem parte real $(r^2 - t^2)$ e parte imaginária $2rt$.

Faça uma função com cabeçalho

```
void quadrado (float r, float t, float *r2, float *t2);
```

que recebe como parâmetros a parte real e imaginária de um número complexo $c = r + i \cdot t$ e devolve a parte real de c^2 em `*r2` e a parte imaginária de c^2 em `*t2`.

(b) Faça um programa que leia um número complexo $c = a + i \cdot b$, com a e b reais, um real eps, com $0 < eps < 1$ e um inteiro $n > 0$ e verifica se existe um inteiro $i > 0$ com $i <= n$ tal que o módulo da parte imaginária de $(c)^{2^i}$ seja menor que eps. Em caso afirmativo, seu programa deve imprimir o valor de i. Em caso negativo, seu programa deve imprimir como resposta "NÃO". Seu programa deve usar obrigatoriamente a função do item (a).

Note que $(c)^{2^i} = \underbrace{(((c^2)^2)^2 \cdots)^2)}_{i \text{ vezes}}$. Assim,

para $i = 3$, $(c)^{2^3} = (((c^2)^2)^2) = c^8$

para $i = 4$, $(c)^{2^4} = ((((c^2)^2)^2)^2) = c^{16}$