Programação Procedimental

Funções e Recursão

Aula 05

Prof. Felipe A. Louza

Roteiro

Funções

Podemos agrupar um conjunto de comandos desenvolvidos para uma tarefa específica em uma função.

```
1   {
2   //cmd A
3   //cmd B
4   //cmd C
5   }
6
7   ...
8
9   {
10  //cmd A
11  //cmd B
12  //cmd C
13  }
```

```
1   funcao();
2
3
4
5
6
7   ...
8
9   funcao();
10
11
12
13
```

```
1  funcao(){
2     //cmd A
3     //cmd B
4     //cmd C
5  }
```

Funções

Funções são úteis para modularizar um código:

- Permite o reaproveitamento de código (minimiza erros e facilita alterações)
- Separa o programa em partes que possam ser logicamente compreendidas de forma isolada

Funções

Já usamos algumas funções no C como scanf e printf.

```c
#include <stdio.h>

int main(){
  int nome[100];

  scanf("%s", nome);
  printf("Hello! %s\n", nome);

return 0;
}
```

Declarando uma função

Uma função é declarada da seguinte forma:

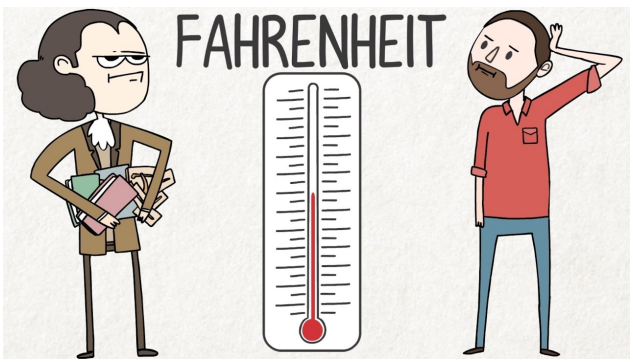
```
1 tipo nome(tipo parâmetro1, tipo parâmetro2, ..., tipo parâmetroN){
2   //comandos
3   return valor;
4 }
```

- O tipo de uma função determina qual será o tipo de seu valor de retorno (comando `return`).
- A lista de parâmetros pode ser vazia.

Definindo uma função

Vamos criar uma função que converte uma temperatura dada em Fahrenheit para Celsius

```
1 double fahrenheitToCelsius(double F){
2   ...
3 }
```



Definindo uma função

A função abaixo recebe como parâmetro a temperatura em Fahrenheit e retorna o valor em Celsius.

```
1 double fahrenheitToCelsius(double F){
2   double C = (F-32.0)*1.8;
3   return C;
4 }
```

$$°C = (°F - 32) \times 5 / 9$$

- Note que o valor de retorno (variável C) é do mesmo tipo da função.

Definindo uma função

Qualquer função pode chamar esta função:

```
1 double fahrenheitToCelsius(double F){
2   double C = (F-32.0)*1.8;
3   return C;
4 }
```

- É preciso passar como parâmetro um valor `double`, que será atribuído à variável `F`.

```
1 #include <stdio.h>
2
3 int main(){
4   double c;
5   c = fahrenheitToCelsius(90.0);
6   printf("Valor em Celsius: %lf\n", c);
7 }
```

Definindo uma função

- Podemos retornar diretamente o valor de uma expressão:

```
1 double fahrenheitToCelsius(double F){
2   double C = (F-32.0)*1.8;
3   return C;
4 }
```

```
1 double fahrenheitToCelsius(double F){
2   return (F-32.0)*1.8;
3 }
```

Definindo uma função

- Podemos utilizar o valor de retorno diretamente em uma expressão.

```
1 #include <stdio.h>
2
3 int main(){
4   printf("Valor em Celsius: %lf\n", fahrenheitToCelsius(90.0));
5 }
```

Fluxo de execução

Todo programa começa executando a função `main()`.

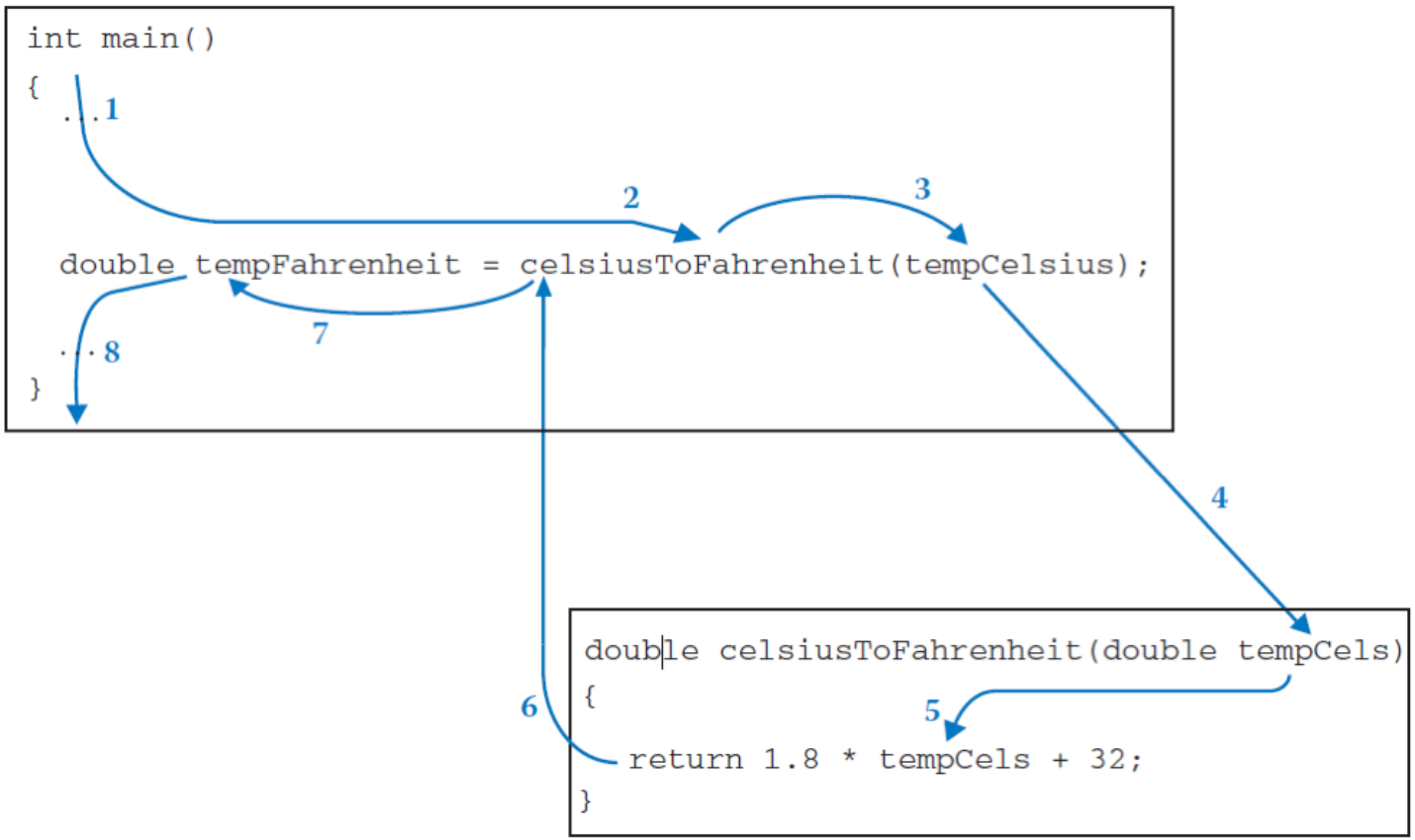
```c
#include <stdio.h>
double fahrenheitToCelsius(double F){
  return (F-32.0)*1.8;
}
double CelsiusToFahrenheit(double C){
  return 1.8*C+32.0;
}
int main(){
  double f;
  scanf("%lf", &f);
  double c = fahrenheitToCelsius(f) ;
  printf("Valor em Celsius: %lf\n", c);
  printf("Valor em Fahrenheit: %d\n", CelsiusToFahrenheit(c) );
}
```

Fluxo de execução

Fluxo de execução:

- Quando se encontra a chamada para uma função, o fluxo de execução passa para a função até o comando `return` (ou o fim da função).
- Depois, o fluxo de execução volta para o ponto onde a chamada da função ocorreu.

Fluxo de execução



Definindo uma função

Nada após o comando `return` será executado:

```c
#include <stdio.h>
double fahrenheitToCelsius(double F){
  return (F-32.0)*1.8;
  printf("Bla bla bla!\n");
}
double CelsiusToFahrenheit(double C){
  return 1.8*C+32.0;
  printf("1 2 3!\n");
}
int main(){
  double f;
  scanf("%lf", &f);
  double c = fahrenheitToCelsius(f) ;
  printf("Valor em Celsius: %lf\n", c);
  printf("Valor em Fahrenheit: %d\n", CelsiusToFahrenheit(c) );
}
```

Chamando uma função

Sempre devemos fornecer valores de mesmo tipo dos parâmetros.

```c
#include <stdio.h>
int somaComMensagem(int a, int b, char texto[100]){
  printf("%s = %d\n", st, a+b);
  return a+b;
}
int main(){
  somaComMensagem(4, 5, "Resultado da soma:");
}
```

- Saída:

```
Resultado da soma: = 9
```

- Já a chamada abaixo gerará um erro de compilação.

```c
int main(){
  somaComMensagem(4, "Resultado da soma:",  5);
}
```

Chamando uma função

Ao chamar uma função passando variáveis como parâmetros, estamos passando apenas os seus valores, que serão copiados para as variáveis da função.

```c
#include <stdio.h>

int incr(int x){
  x = x + 1;
  return x;
}

int main(){
  int a = 2, b;
  b = incr(a);
  printf("a = %d, b = %d\n", a, b);
}
```

- Os valores das variáveis na chamada da função não são afetados por alterações dentro da função.

Roteiro

O tipo `void`

O tipo `void` é um tipo especial.

- Ele representa "nada": uma variável desse tipo armazena conteúdo indeterminado.
- Em geral, usamos o `void` para informar quando uma função não retorna nenhum valor.

O tipo `void`

Por exemplo, a função abaixo imprime o número que for passado para ela como parâmetro e não devolve nada.

```c
#include <stdio.h>

void imprime(int numero){
  printf("Número %d\n", numero);
}

int main(){

  imprime(10);
  imprime(20);

return 0;
}
```

- Neste caso não utilizamos o comando `return`.

O tipo `void`

- A lista de parâmetros de uma função pode ser vazia.

```
1 int leNumero() {
2   int c;
3   printf("Digite um número:");
4   scanf("%d", &c);
5   return c;
6 }
```

Roteiro

Protótipos de funções

Até o momento, sempre definimos as nossas funções antes da função `main()`

- O que acontece se declararmos uma função depois do `main()`?

```
1  #include <stdio.h>
2
3  int main(){
4    int a = 0, b = 5;
5    printf ("%d\n", maior(a, b));
6  return 0;
7  }
8
9  int maior(int a, int b) {
10   if(a>b) return a;
11   else return b;
12 }
```

- Dependendo do compilador, ocorre um erro de compilação!

Protótipos de funções

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código:

- Protótipo/assinatura: a função sem o bloco, com a linha terminando com ;

```
1 int maior(int a, int b);
```

  - é uma "promessa" de que a função existirá no programa
  - permite chamar uma função que será definida depois

Protótipos de funções

```
1  #include <stdio.h>
2  #include <outras bibliotecas>
3
4  //Protótipos de funções
5  int fun1(Parâmetros);
6  int fun2(Parâmetros);
7  ...
8
9  int main(){
10   //comandos
11 }
12
13 int fun1(Parâmetros){
14 //comandos
15 }
16
17 int fun2(Parâmetros){
18 //comandos
19 }
20 ...
```

Exemplo

Exemplo:

```c
#include <stdio.h>

int soma(int op1, int op2);
int subt(int op1, int op2);

int main () {
  int a = 0, b = 5;
  printf (" soma = %d\n subtracao = %d\n", soma (a, b), subt(a, b));
  return 0;
}

int soma (int op1, int op2) {
  return (op1 + op2);
}

int subt (int op1, int op2) {
  return (op1 - op2);
}
```

Roteiro

Chamada de funções

Chamada de funções.

```
1   funcao1(){
2     //cmd A
3     funcao2();
4     //cmd I
5   }
6
7   ...
8
9   funcao2(){
10    //cmd B
11    funcao3();
12    //cmd H
13  }
```

```
1   funcao3(){
2     //cmd C
3     funcao4()
4     //cmd G
5   }
6
7   ...
8
9   funcao4(){
10    //cmd D
11    //cmd E
12    //cmd F
13  }
```

```
1   #include <...>
2   ...
3
4   funcao1();
5   funcao2();
6   funcao3();
7   funcao4();
8
9   int main(){
10    ...
11    funcao1();
12    ...
13  }
```

- Qualquer função pode chamar outra função.
  - Exceto a `main()` que é chamada apenas pelo sistema operacional.

Exemplo 1

```c
#include <stdio.h>

int fun1(int a);
int fun2(int b);

int main(){
  int c = 5;
  printf("%d => %d\n", c, fun1(c));
}
int fun1(int a){
  a = fun2(a+1);
  return a;
}
int fun2(int b){
  b = 2*b;
  return b;
}
```

- O que será impresso?

Exemplo 2

```c
#include <stdio.h>

int fun1(int a);
int fun2(int b);
int fun3(int b);

int main(){
  int c = 5;
  printf("%d => %d\n", c, fun1(c));
}
int fun1(int a){
  a = fun2(a+1);
  return a;
}
int fun2(int b){
  b = 2*b;
  return fun3(b);
}
int fun3(int c){
  return c*c;
}
```

- E agora?

Exemplo 3

Funções podem chamar elas mesmas (recursão)

```c
#include <stdio.h>

int fun1(int a);

int main(){
  int c = 5;
  printf("%d => %d\n", c, fun1(c));
}

int fun1(int a){
  a = fun1(a+1);
  return a;
}
```

- O que será impresso?

Exemplo 3

Cuidado: é preciso definir um critério de parada:

```
1  #include <stdio.h>
2
3  int fun1(int a);
4
5  int main(){
6    int c = 5;
7    printf("%d => %d\n", c, fun1(c));
8  }
9
10 int fun1(int a){
11   if(a<=10) a = fun1(a+1);
12   return a;
13 }
```

- O que será impresso?

Exemplo 3

Recursão também pode envolver diferentes funções:

```c
#include <stdio.h>

int fun1(int a);
int fun2(int b);

int main(){
  int c = 5;
  printf("%d => %d\n", c, fun1(c));
}

int fun1(int a){
  a = fun2(a+1);
  return a;
}
int fun2(int b){
  b = fun1(b+1);
  return b;
}
```

Exemplo 3

Atenção ao critério de parada:

```c
#include <stdio.h>

int fun1(int a);
int fun2(int b);

int main(){
  int c = 5;
  printf("%d => %d\n", c, fun1(c));
}

int fun1(int a){
  if(a<=10) a = fun2(a+1);
  return a;
}
int fun2(int b){
  b = fun1(b+1);
  return b;
}
```

Roteiro

Escopo de variáveis

O escopo de uma variável determina quais partes do código ela pode ser acessada.

```
1   funcao1(){
2     int a = 2;
3     int b = 3;
4     ...
5     ...
6   }
7   ...
8   funcao2(){
9     int b = 4;
10    int c = 5;
11    funcao3(c);
12    ...
13  }
```

```
1   funcao3(int c){
2     c = 6;
3     int d = 7;
4     funcao4(d);
5     ...
6   }
7   ...
8   funcao4(int d){
9     d = 8;
10    int e = 9;
11    ...
12    ...
13  }
```

```
1  #include <...>
2  ...
3
4  int a = 1;
5
6  funcao1();
7  ...
8
9  int main(){
10   ...
11   funcao1();
12   ...
13 }
```

Variáveis globais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9  void funcao2(int parametro) {
10   int local1, local2;
11   ...
12 }
13 int main() {
14   int local;
15 }
```

`global` é uma variável global:

- pode ser acessada em qualquer função
- variáveis globais só são usadas em casos específicos
- podem levar a erros difíceis de encontrar no programa

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9  void funcao2(int parametro) {
10   int local1, local2;
11   ...
12 }
13 int main() {
14   int local;
15 }
```

`local`, `local1`, `local2` e `parametro` são variáveis locais:

- existem apenas dentro da função onde foram definidas
- `local1` de `funcao1` é diferente de `local1` de `funcao2`
- Importante: quando a função acaba, o valor é perdido

Sobreposição de escopo

```c
#include <stdio.h>

int x;

void funcao1(int parametro) {
  x = 10;
  ...
}

void funcao2(int parametro) {
  int x;
  x = 10;
}
```

Variáveis locais têm precedência sobre variáveis globais

- Em `funcao1`, a variável global `x` tem seu valor alterado
- Em `funcao2`, a variável local `x` tem seu valor alterado

Atenção!

Um dos motivos que evitamos o uso de variáveis globais!

Variáveis locais e variáveis globais

Variáveis locais

- Ela existe somente dentro da função em que foi declarada, e após o término da execução desta, a variável deixa de existir.
- **Variáveis parâmetros também são locais**

Variáveis globais

- Ela é declarada fora de qualquer função.
- Essa variável é visível em todas as funções.
- Qualquer função pode alterá-la e ela existe durante toda a execução do programa.

Organização de um Programa

```c
#include <stdio.h>
#include <outras bibliotecas>

//Declaração de Variáveis Globais

//Protótipos de funções

int main(){
  //Declaração de variáveis locais
  ...
}

int fun1(Parâmetros){ //Parâmetros também são variáveis locais
  //Declaração de variáveis locais
  ...
}
int fun2(Parâmetros){ //Parâmetros também são variáveis locais
  //Declaração de variáveis locais
  ...
}
...
```

Roteiro

Exemplo 1

```c
#include <stdio.h>

void fun1();
void fun2();

int x;
int main(){
  x = 1;
  fun1();
  fun2();
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(){
  int x = 3;
  printf("fun2: %d\n",x);
}
```

Exemplo 2

```c
#include <stdio.h>

void fun1();
void fun2();

int x = 1;
int main(){
  int x=1;
  fun1();
  fun2();
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(){
  int x = 4;
  printf("fun2: %d\n",x);
}
```

Exemplo 3

```c
#include <stdio.h>

void fun1();
void fun2(int x);

int x = 1;
int main(){
  x=2;
  fun1();
  fun2(x);
  printf("main: %d\n", x);
}

void fun1(){
  x = x + 1;
  printf("fun1: %d\n",x);
}

void fun2(int x){
  x = x + 1 ;
  printf("fun2: %d\n",x);
}
```

Roteiro

Chamada de funções

Chamada de funções.

```
1   funcao1(){
2     //cmd A
3     funcao2();
4     //cmd I
5   }
6
7   ...
8
9   funcao2(){
10    //cmd B
11    funcao3();
12    //cmd H
13  }
```

```
1   funcao3(){
2     //cmd C
3     funcao4()
4     //cmd G
5   }
6
7   ...
8
9   funcao4(){
10    //cmd D
11    funcao4()
12    //cmd F
13  }
```

```
1   #include <...>
2   ...
3
4   funcao1();
5   funcao2();
6   funcao3();
7   funcao4();
8
9   int main(){
10    ...
11    funcao1();
12    ...
13  }
```

- Qualquer função pode chamar outra função (inclusive ela mesma).
  - Recursão!

Recursão

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, etc...
- ou podem ser vistos do ponto de vista da recursão
  - soma, exponenciação, etc...

Recursão

A ideia é que um problema pode ser resolvido da seguinte maneira:

- Primeiro, definimos as soluções para casos básicos
- Em seguida, tentamos reduzir o problema para instâncias menores do problema
- Finalmente, combinamos o resultado das instâncias menores para obter um resultado do problema original

Exemplo 1

Fatorial: $n!$

$$fat(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot fat(n-1) & \text{se } n > 0 \end{cases}$$

```
1 int fat(int n ) {
2   if ( n == 0) /* caso base */
3     return 1;
4   else /* caso geral */
5     return n * fat(n-1) ; /* instância menor */
6 }
```

- Calcule `fat(5)`:

Exemplo 2

Exponenciação: a^b

Seja a é um número inteiro e b é um número inteiro não-negativo

$$a^b = \begin{cases} 1 & \text{se } b = 0 \\ a \cdot a^{b-1} & \text{se } b > 0 \end{cases}$$

```
1 int potencia(int a, int b) {
2   if ( b == 0)
3     return 1;
4   else
5     return a*potencia(a, b-1) ;
6 }
```

- Calcule `potencia(2,5)`:

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais "elegantes"

Mas algumas vezes podem ser

- muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Exemplo 3

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ f(n-1) + f(n-2) & \text{se } n > 1 \end{cases}$$

```
1 int fib(int n) {
2   if(n == 0) return 0;
3   if(n == 1) return 1;
4   else
5     return fib(n-1)+fib(n-2);
6 }
```

- Calcule `fib(6)`:

Exemplo 3

Solução iterativa: Sequência de Fibonacci:

```
1  int fib_iterativo(int n) {
2    int ant, atual, prox, i;
3    ant = atual = 1;
4    for(i = 3; i<=n ; i++){
5      prox = ant + atual;
6      ant = atual;
7      atual = prox;
8    }
9  return atual ;
10 }
```

- Calcule `fib(6)`:

Exemplo 3

Número de operações:

- iterativo: $\approx n$
- recursivo: $\approx \texttt{fib}(n)$ (aproximadamente 1.6^n)

Roteiro

Vetores em funções

Vetores também podem ser passados como parâmetros em funções.

- Ao contrário dos tipos simples, vetores têm um comportamento diferente quando usados como parâmetros de funções.
- Protótipo:

```
1 void soma_um(int vetor[], int n);
```

  - Em geral, precisamos passar o tamanho do vetor como parâmetro.

Vetores em funções

```c
#include <stdio.h>

void soma_um(int vetor[], int n) {
  int i;
  for (i = 0; i < n; i++)
    vetor[i]++;
}

int main() {
  int i, vetor[5] = {1, 2, 3, 4, 5};
  soma_um(vetor, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", vetor[i]);

return 0;
}
```

No código acima é impresso 2 3 4 5 6

- Ou seja, a função altera o conteúdo do vetor.

Vetores em funções

Vetores são passados como referências para as funções.

- Ou seja, **não é criado** um novo vetor!
- Os valores de um vetor **podem ser alterados** dentro de uma função!

Vetores em funções

```c
#include <stdio.h>

void fun1(int x[], int tam){
  int i;
  for(i=0;i<tam;i++)
     x[i]=1;
}

int main(){
  int vetor[10];
  int i;

  for(i=0;i<10;i++)
    vetor[i]=0;

  fun1(vetor,5);
  for(i=0;i<10;i++)
    printf("%d\n",vetor[i]);
}
```

O que será impresso?

Vetores em funções

```c
#include <stdio.h>

void fun1(int x[], int tam){
  int i;
  for(i=0;i<tam;i++)
      x[i]=1;
}

int main(){
  int vetor[10];
  int i;

  for(i=0;i<10;i++)
    vetor[i]=0;

  fun1(vetor,100);
  for(i=0;i<10;i++)
    printf("%d\n",vetor[i]);
}
```

Cuidado com o parâmetro de tamanho.

Vetores em funções

Podemos informar o tamanho do vetor na assinatura: `int vet[10]`

```c
#include <stdio.h>

void fun1(int x[10], int tam){
  int i;
  for(i=0;i<tam;i++)
     x[i]=5;
}

int main(){
  int vetor[10];
  int i;

  for(i=0;i<10;i++)
    vetor[i]=0;

  fun1(vetor,10);
  for(i=0;i<10;i++)
    printf("%d\n",vetor[i]);
}
```

Roteiro

Vetores multidimensionais em funções

Para vetores multidimensionais (matrizes), apenas a primeira dimensão pode ser vazia, ex: `int matriz[][3][3]`

```
1  #include <stdio.h>
2
3  void fun1(int matriz[][3][3], int N, int M, int K){
4    int i,j,k, count=0;
5    for(k=0; k<K; k++)
6      for(i=0; i<N; i++)
7        for(j=0; j<M; j++)
8          matriz[i][j][k] = count++;
9  }
10
11 int main(){
12   int matriz[3][3][3];
13
14   fun1(matriz,10);
15   ...
16   ...
17 }
```

Vetores em funções

```c
#include <stdio.h>

void fun1(int matriz[][3][3], int N, int M, int K){
  int i,j,k, count=0;
  for(k=0; k<K; k++)
    for(i=0; i<N; i++)
      for(j=0; j<M; j++)
        matriz[i][j][k] = count++;
}

int main(){
  int matriz[3][3][3];

  int i,j,k;
  fun1(matriz, 3, 3, 3);
  for(k=0; k<3; k++){
    for(i=0; i<3; i++){
      for(j=0; j<3; j++)
        printf("%d ", matriz[i][j][k]);
      printf("\n");
    }
    printf("\n##\n");
  }
return 0;
}
```

Vetores em funções

Não podemos retornar um vetor por uma função.

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

  int[] leVet() {
    int i, vet[100];
    for (i = 0; i < 100; i++) {
      printf("Digite um numero:");
      scanf("%d", &vet[i]);
    }
    return vet;
  }
  ```

- O código acima não compila por causa do `int[]` .
  ```

Vetores em funções

Mas como um vetor é alterado dentro de uma função, podemos criar a seguinte função para ler vetores.

```c
#include <stdio.h>

void leVet(int vet[], int tam){
  int i;
  for(i = 0; i < tam; i++){
    printf("Digite numero:");
    scanf("%d",&vet[i]);
  }
}
```

- A função abaixo faz a impressão de um vetor.

```c
void escreveVet(int vet[], int tam){
  int i;
  for(i=0; i< tam; i++)
    printf("vet[%d] = %d\n",i,vet[i]);
}
```

Vetores em funções

Exemplo de uso das funções anteriores:

```
1  int main(){
2    int vet1[10], vet2[20];
3
4    printf("Lendo Vetor 1\n");
5    leVet(vet1,10);
6    printf("Lendo Vetor 2\n");
7    leVet(vet2,20);
8
9    printf("Imprimindo Vetor 1\n");
10   escreveVet(vet1,10);
11   printf("Imprimindo Vetor 2\n");
12   escreveVet(vet2,20);
13 }
```

Roteiro

Macros

Macros são trechos de cógidos que são substituídos durante o pré-processamento (antes da compilação).

- Devem ser extremamente pequenas e simples:

```
1 #define nome_macro valor
```

Macros

- Vimos como declarar constantes com macros.

```
1 #define PI 3.141592
```

- Macros também podem receber argumentos:

```
1 #define SOMA(a,b) a+b
```

Macros

```c
#include <stdio.h>

#define SOMA(x,y) x+y

int main() {

  int a, b, resultado;
  scanf("%d %d", &a, &b);

  resultado = SOMA(a,b); // resultado = a+b
  printf("a + b = %d\n", resultado);

return 0;
}
```

- No código acima `SOMA(x,y)` é substituído por `x+y`

Macros

```c
#include <stdio.h>

#define SOMA(x,y) x+y

int main() {

  int a, b, resultado;
  scanf("%d %d", &a, &b);

  //resultado = SOMA(a,b); // resultado = a+b
  printf("a + b = %d\n", SOMA(a,b));

return 0;
}
```

- Podemos usar `SOMA(x,y)` diretamente no `printf`

Macros

```c
#include <stdio.h>

#define SOMA(x,y) x+y ;

int main() {

  int a, b, resultado;
  scanf("%d %d", &a, &b);

  //resultado = SOMA(a,b); // resultado = a+b
  printf("a + b = %d\n", SOMA(a,b));

return 0;
}
```

- Cuidado: erro de compilação.

Macros

```c
#include <stdio.h>

#define SOMA(x,y) x+y

int main() {

  int a, b, resultado;
  scanf("%d %d", &a, &b);

  resultado = 5*SOMA(a,b); // resultado = 5*a+b
  printf("5 * a + b = %d\n", resultado);

return 0;
}
```

- Cuidado: erro de lógica, `SOMA(x,y)` é substituído por `x+y`

Macros

```c
#include <stdio.h>

#define SOMA(x,y) (x+y)

int main() {

  int a, b, resultado;
  scanf("%d %d", &a, &b);

  resultado = 5*SOMA(a,b); // resultado = 5*(a+b)
  printf("5 * (a + b) = %d\n", resultado);

return 0;
}
```

- Nesse caso, resolvemos com `(x+y)` na macro.

Macros

Precisamos ser cuidadosos com macros:

```c
#include <stdio.h>

#define SOMA(x,y) (x+y)

int main() {

  int a;
  scanf("%d", &a);

  printf("resultado  = %d\n", SOMA(a, "ABC"));

return 0;
}
```

- Não há checagem de tipos nos argumentos.

Macros

Vantagens: podem tornar o código mais legível.

- Evita a leitura de dados da memória.
- Evita procedimentos que ocorrem ao chamarmos uma função.

```
1  #include <stdio.h>
2
3  #define TAXA 0.3
4
5  int main() {
6
7    int taxa = 0.3;
8
9    ...
10   imposto = preco*taxa;
11   ...
12   imposto = preco*TAXA;
13   ...
14
15 return 0;
16 }
```

Macros

- Podemos definir uma macro em várias linhas (separadas por "\"):

```
1 #define nome_macro v\
2                    a\
3                    l\
4                    o\
5                    r
```

Roteiro

A função `main()`

A função `main()` retorna um tipo `int`:

```c
#include <stdio.h>

int main(){

  printf ("Ola turma!\n");

return 0;
}
```

- O comando `return 0;` informa ao sistema operacional se o programa funcionou corretamente.

A função `main()`

A função `main()` pode receber parâmetros direto do terminal:

```c
#include <stdio.h>

int main(int argc, char **argv){

  printf ("%d\t%s\n", argc, argv[0]);

return 0;
}
```

- `int argc` informa a quantidade de parâmetros (sempre ≥ 1).
- `char **argv` armazena em um vetor de strings os parâmetros do terminal.

A função `main()`

Valor armazenado em `argv[0]`:

```c
#include <stdio.h>

int main(int argc, char **argv){

  printf ("%s\n", argv[0]);

return 0;
}
```

- Executando:

```
$ gcc teste.c -o teste
$ ./teste
./teste
```

A função `main()`

Imprimindo todos os parâmetros:

```c
#include <stdio.h>

int main(int argc, char **argv){

  int i;
  for(i=1; i<argc; i++)
    printf ("%s\n", argv[i]);

return 0;
}
```

- Executando:

```
$ gcc teste.c -o teste
$ ./teste abc de f 0 1
abc de f 0 1
```

A função `main()`

Exemplo: soma de inteiros:

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

int soma(int a, int b){
  return a+b;
}
int main(int argc, char **argv){
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf("Soma: %d\n", soma(a, b));

return 0;
}
```

- Executando:

```
$ gcc teste.c -o teste
$ ./teste 1 5
Soma: 6
```

A função `main()`

Exemplo: soma de inteiros:

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

int soma(int a, int b){
  return a+b;
}
int main(int argc, char **argv){

  if(argc!=3) return 1;

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf("Soma: %d\n", soma(a, b));

return 0;
}
```

- Precisamos verificar se `argc==3`:

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da UNICAMP.
2. Materiais adaptados dos slides do Prof. Túlio Toffolo, da UFOP.