C
Funções

Adriano Cruz
adriano@nce.ufrj.br

Instituto de Matemática
Departamento de Ciência da Computação
UFRJ

17 de outubro de 2013

Section Summary

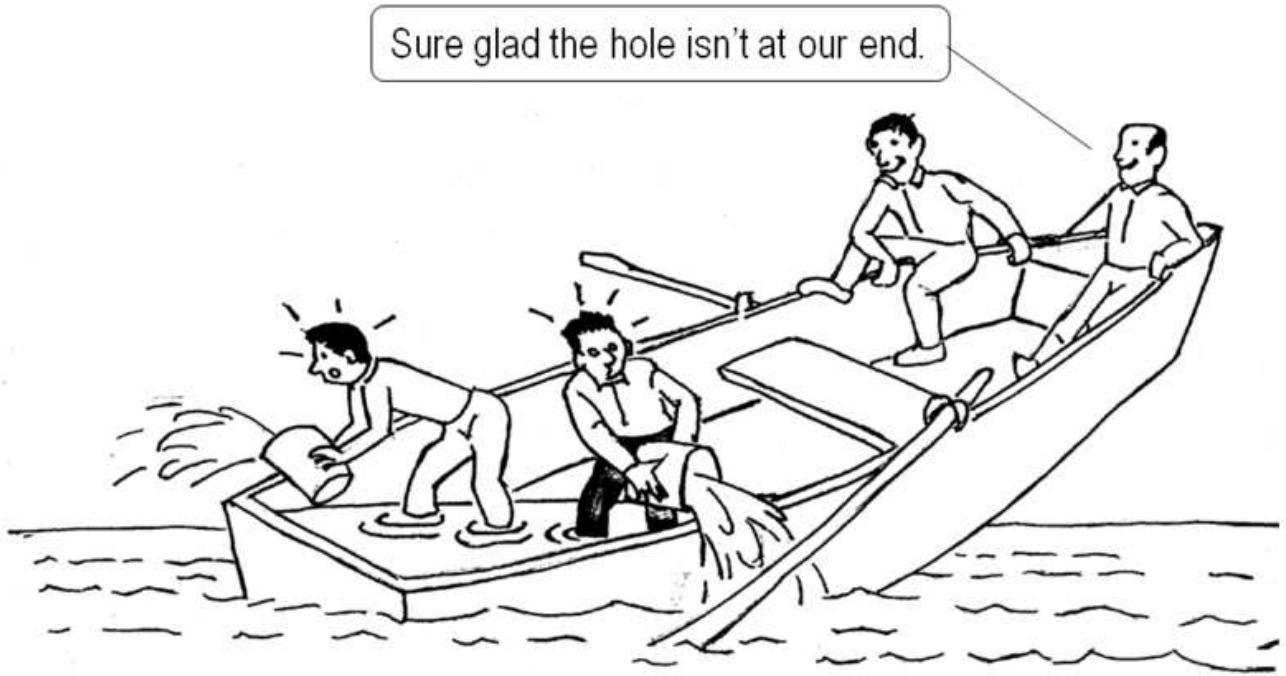
Bibliografia

1. Adriano Cruz. *Curso de Linguagem* C, Disponível em `http://equipe.nce.ufrj.br/adriano`

Bibliografia

1. Adriano Cruz. *Curso de Linguagem* C, Disponível em `http://equipe.nce.ufrj.br/adriano`
2. Ulysses de Oliveira. *Programando em* C, Editora Ciência Moderna.

Dividindo Tarefas?



Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

Exemplo: Funções de entrada e saída são o exemplo mais direto deste reuso.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

Exemplo: Funções de entrada e saída são o exemplo mais direto deste reuso.

Vantagens: Diminui o tempo de desenvolvimento do programas.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

Exemplo: Funções de entrada e saída são o exemplo mais direto deste reuso.

Vantagens: Diminui o tempo de desenvolvimento do programas.

Vantagens: Funções foram testadas por diversos usuários contribuindo para a redução dos custos de desenvolvimento dos projetos.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

- **Exemplo:** Funções de entrada e saída são o exemplo mais direto deste reuso.
- **Vantagens:** Diminui o tempo de desenvolvimento do programas.
- **Vantagens:** Funções foram testadas por diversos usuários contribuindo para a redução dos custos de desenvolvimento dos projetos.

Divisão: Divisão do trabalho necessário para construir um aplicativo.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

- **Exemplo:** Funções de entrada e saída são o exemplo mais direto deste reuso.
- **Vantagens:** Diminui o tempo de desenvolvimento do programas.
- **Vantagens:** Funções foram testadas por diversos usuários contribuindo para a redução dos custos de desenvolvimento dos projetos.

Divisão: Divisão do trabalho necessário para construir um aplicativo.

- **Exemplo:** Fábrica de Software.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

- **Exemplo:** Funções de entrada e saída são o exemplo mais direto deste reuso.
- **Vantagens:** Diminui o tempo de desenvolvimento do programas.
- **Vantagens:** Funções foram testadas por diversos usuários contribuindo para a redução dos custos de desenvolvimento dos projetos.

Divisão: Divisão do trabalho necessário para construir um aplicativo.

- **Exemplo:** Fábrica de Software.
- **Vantagens:** Programadores trabalham independentemente acelerando o desenvolvimento dos programas.

Para que?

Reuso: Reuso de código desenvolvido por outros programadores.

- **Exemplo:** Funções de entrada e saída são o exemplo mais direto deste reuso.
- **Vantagens:** Diminui o tempo de desenvolvimento do programas.
- **Vantagens:** Funções foram testadas por diversos usuários contribuindo para a redução dos custos de desenvolvimento dos projetos.

Divisão: Divisão do trabalho necessário para construir um aplicativo.

- **Exemplo:** Fábrica de Software.
- **Vantagens:** Programadores trabalham independentemente acelerando o desenvolvimento dos programas.
- **Vantagens:** Permite que os testes do sistema completo sejam feitos mais facilmente e com mais garantia de correção.

Para que?

Eu sozinho: Permite dividir um trabalho complexo em diversas fatias menores permitindo ao programador se concentrar a cada vez em problemas mais simples.

Section Summary

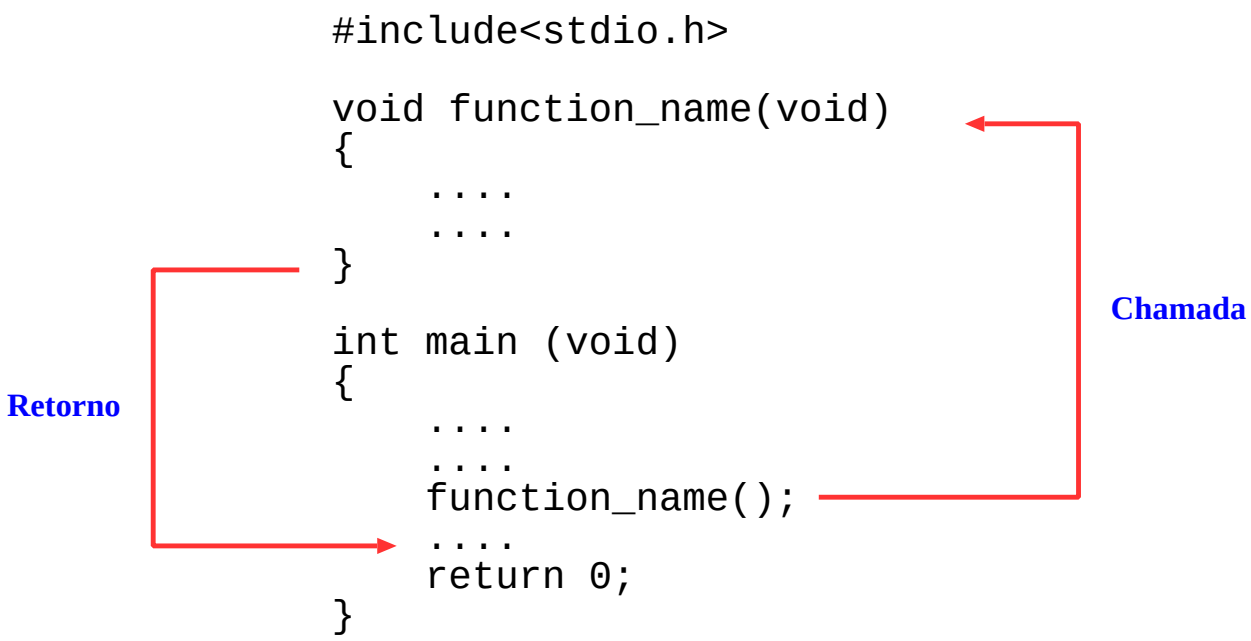
Forma Geral

```
tipo nome (tipo nome1, tipo nome2, ..., tipo nomeN )
{
    declaração das variáveis
    corpo da função
}
```

E daí?

Uma função recebe uma lista de argumentos (`nome1, nome2, ..., nomeN`), executa comandos com estes argumentos e pode retornar ou não um resultado para a função que chamou esta função.

Funções


#include<stdio.h>
void function_name(void)
{
....
....
}
int main (void)
{
....
....
function_name();
....
return 0;
}
Chamada
Retorno

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.
- Não é possível usar uma única definição de tipo para várias variáveis.

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.
- Não é possível usar uma única definição de tipo para várias variáveis.
- A lista de argumentos pode ser vazia, ou seja, a função não recebe nenhum argumento.

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.
- Não é possível usar uma única definição de tipo para várias variáveis.
- A lista de argumentos pode ser vazia, ou seja, a função não recebe nenhum argumento.
- O nome da função pode ser qualquer identificador válido.

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.
- Não é possível usar uma única definição de tipo para várias variáveis.
- A lista de argumentos pode ser vazia, ou seja, a função não recebe nenhum argumento.
- O nome da função pode ser qualquer identificador válido.
- O `tipo` que aparece antes do nome da função especifica o tipo do resultado que será devolvido ao final da execução da função.

Observações

- A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados.
- Não é possível usar uma única definição de tipo para várias variáveis.
- A lista de argumentos pode ser vazia, ou seja, a função não recebe nenhum argumento.
- O nome da função pode ser qualquer identificador válido.
- O `tipo` que aparece antes do nome da função especifica o tipo do resultado que será devolvido ao final da execução da função.
- O tipo **void** pode ser usado para declarar funções que não retornam valor algum.

Retornando.

- Há basicamente duas maneiras de terminar a execução de uma função.

Retornando.

- Há basicamente duas maneiras de terminar a execução de uma função.
- Normalmente usa-se o comando **return** para retornar o resultado da função.

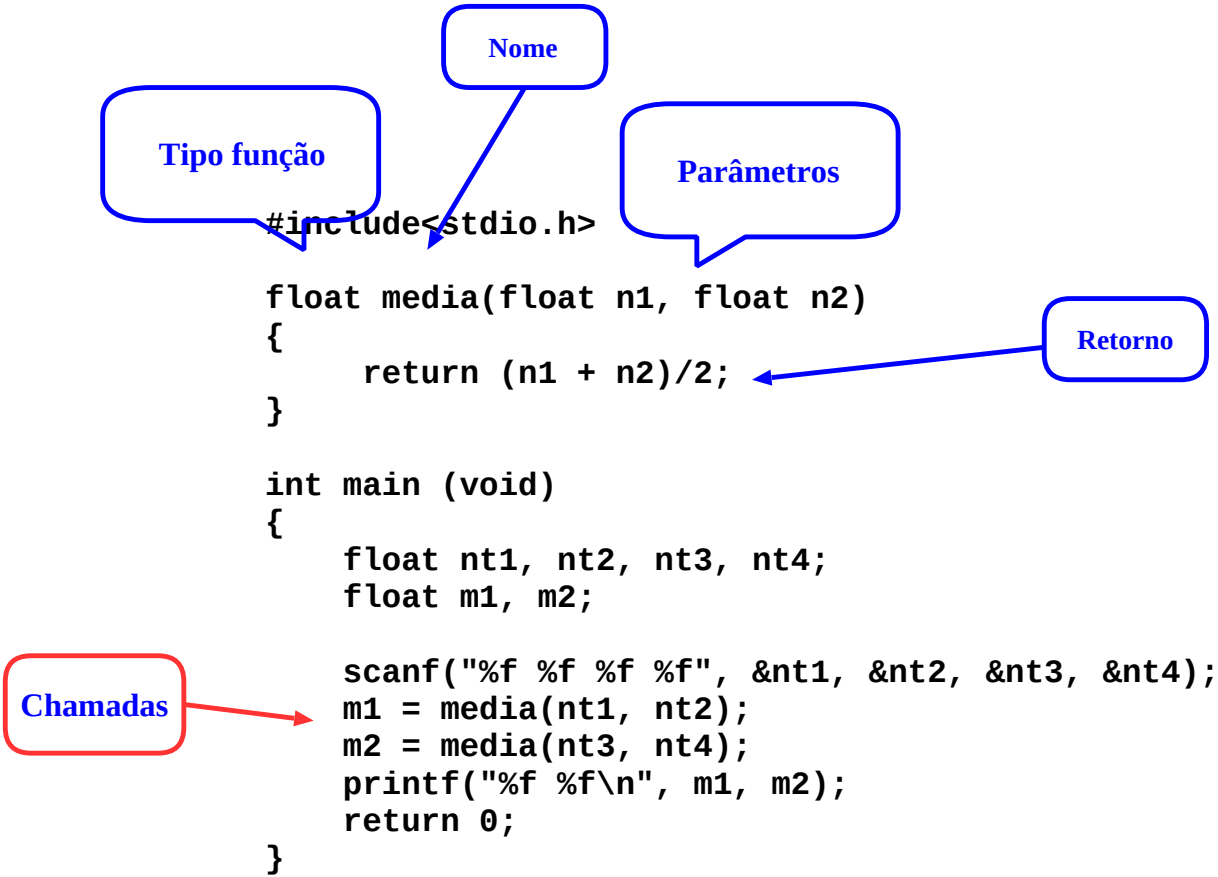
Retornando.

- Há basicamente duas maneiras de terminar a execução de uma função.
- Normalmente usa-se o comando **return** para retornar o resultado da função.
- Portanto, quando o comando
  return `expressão;`
  for executado, o valor da `expressão` é devolvido para a função que chamou.

Retornando.

- Há basicamente duas maneiras de terminar a execução de uma função.
- Normalmente usa-se o comando **return** para retornar o resultado da função.
- Portanto, quando o comando
  return `expressão;`
  for executado, o valor da `expressão` é devolvido para a função que chamou.
- Quando não há valor para retornar o comando **return** não precisa ser usado e a função termina quando a chave que indica o término do corpo da função é atingido.

Exemplo


Nome
Tipo função
Parâmetros
Retorno
Chamadas
#include<stdio.h>
float media(float n1, float n2)
{
return (n1 + n2)/2;
}
int main (void)
{
float nt1, nt2, nt3, nt4;
float m1, m2;
scanf("%f %f %f %f", &nt1, &nt2, &nt3, &nt4);
m1 = media(nt1, nt2);
m2 = media(nt3, nt4);
printf("%f %f\n", m1, m2);
return 0;
}

Observações

- É importante notar que o nome da função pode aparecer em qualquer lugar onde o nome de uma variável apareceria.

Observações

- É importante notar que o nome da função pode aparecer em qualquer lugar onde o nome de uma variável apareceria.
- Além disso os tipos e o número de parâmetros que aparecem na declaração da função e na sua chamada devem estar na mesma ordem e ser tipos equivalentes.

Observações

- É importante notar que o nome da função pode aparecer em qualquer lugar onde o nome de uma variável apareceria.
- Além disso os tipos e o número de parâmetros que aparecem na declaração da função e na sua chamada devem estar na mesma ordem e ser tipos equivalentes.
- Os nomes das variáveis nos programas que usam uma função podem ser diferentes dos nomes usados na definição da função.

Exemplo

```c
#include <stdio.h>
int fat(int n)
{
    int f = 1;

    for ( ; n>1; n--) {
        f *= n;
    }
    return f;
}
int main ( void )
{
    int n, p, c;
    n = 5; p = 3;
    c = fat(n) / (fat(p) * fat(n-p));
    printf("%d \n", c);
    return 0;
}
```

Section Summary

Protótipos

O padrão ANSI estendeu a declaração da função para permitir que o compilador faça uma verificação mais rígida da compatibilidade entre os tipos que a função espera receber e àqueles que são fornecidos. Protótipos de funções ajudam a detectar erros antes que eles ocorram, impedindo que funções sejam chamadas com argumentos inconsistentes.

Protótipos

O padrão ANSI estendeu a declaração da função para permitir que o compilador faça uma verificação mais rígida da compatibilidade entre os tipos que a função espera receber e àqueles que são fornecidos. Protótipos de funções ajudam a detectar erros antes que eles ocorram, impedindo que funções sejam chamadas com argumentos inconsistentes.

A forma geral de definição de um protótipo é a seguinte:

```
tipo nome (tipo nome1, tipo nome2, ..., tipo nomeN);
```

Exemplo

```c
#include <stdio.h>

/* Prototipo da funcao */
int soma (int a, int b);

/* Funcao Principal */
int main() {
     int a=5, b=9;
     printf("%d\n", soma(a,b));
 return 0;
}

/* Definicao da funcao */
int soma(int a, int b) {
    return a+b;
}
```

Section Summary

Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.

Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.
- Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis.

Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.
- Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis.
- As variáveis podem ser declaradas basicamente em três lugares:

Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.
- Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis.
- As variáveis podem ser declaradas basicamente em três lugares:
  - dentro de funções, variáveis locais

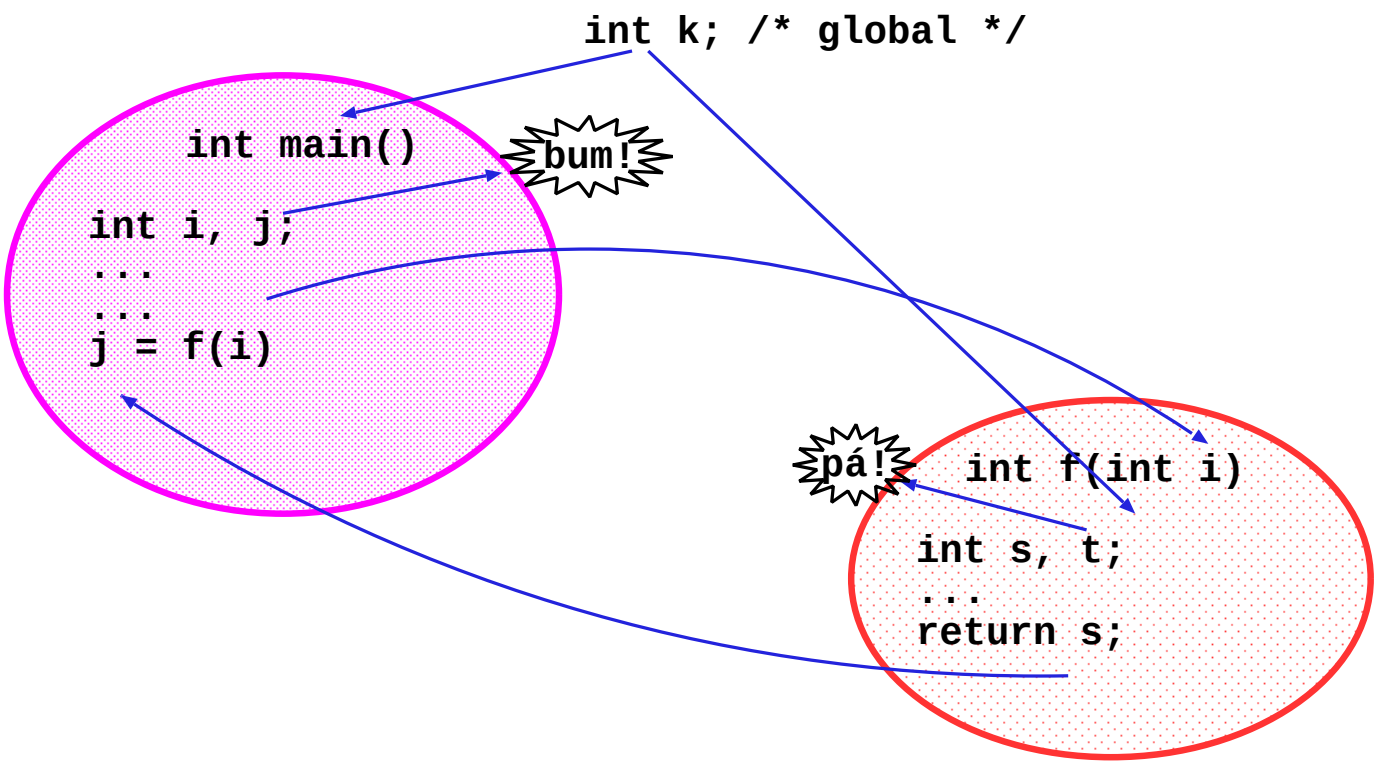
Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.
- Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis.
- As variáveis podem ser declaradas basicamente em três lugares:
  - dentro de funções, variáveis locais
  - fora de todas as funções, variáveis globais

Escopo de Variáveis

- Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes.
- Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis.
- As variáveis podem ser declaradas basicamente em três lugares:
  - dentro de funções, variáveis locais
  - fora de todas as funções, variáveis globais
  - na lista de parâmetros das funções parâmetros formais.

Escopo de Variáveis



Variáveis Locais

- As variáveis locais são aquelas declaradas dentro de uma função ou um bloco de comandos.

Variáveis Locais

- As variáveis locais são aquelas declaradas dentro de uma função ou um bloco de comandos.
- Elas passam a existir quando do início da execução do bloco de comandos ou função onde foram definidas e são destruídas ao final da execução do bloco.

Variáveis Locais

- As variáveis locais são aquelas declaradas dentro de uma função ou um bloco de comandos.
- Elas passam a existir quando do início da execução do bloco de comandos ou função onde foram definidas e são destruídas ao final da execução do bloco.
- Uma variável local só pode ser referenciada, ou seja usada, dentro da função (ou bloco) onde foi declarada.

Variáveis Locais

- As variáveis locais são aquelas declaradas dentro de uma função ou um bloco de comandos.
- Elas passam a existir quando do início da execução do bloco de comandos ou função onde foram definidas e são destruídas ao final da execução do bloco.
- Uma variável local só pode ser referenciada, ou seja usada, dentro da função (ou bloco) onde foi declarada.
- Variáveis locais são invisíveis para outras funções do mesmo programa.

Exemplo

```c
#include <stdio.h>
void pares(void) {
    int i;
    for (i = 2; i <= 10; i += 2) {
        printf("%d: ", i);
    }
}
void impares(void) {
    int i;
    for (i = 3; i <= 11; i += 2) {
        printf("%d: ", i);
    }
}
int main(int argc, char *argv[]) {
    pares();
    printf("\n");
    impares();
    return 0;
}
```

Variáveis Globais

- As variáveis globais são definidas fora de qualquer função e são portanto disponíveis para qualquer função.

Variáveis Globais

- As variáveis globais são definidas fora de qualquer função e são portanto disponíveis para qualquer função.
- Este tipo de variável pode servir como uma canal de comunicação entre funções, uma maneira de transferir valores entre elas.

Variáveis Globais

- As variáveis globais são definidas fora de qualquer função e são portanto disponíveis para qualquer função.
- Este tipo de variável pode servir como uma canal de comunicação entre funções, uma maneira de transferir valores entre elas.
- Por exemplo, se duas funções tem de partilhar dados, mais uma não chama a outra, uma variável global tem de ser usada.

Variáveis Globais

```c
#include <stdio.h>
int i;  /* variavel global */

void soma1(void) {
    i += 1;
    printf("Funcao soma1: i = %d\n", i);
}
void sub1(void) {
    int i = 10;
    i -= 1;
    printf("Funcao sub1: i = %d\n", i);
}
int main(int argc, char *argv[]) {
    i = 0;
    soma1();
    sub1();
    printf("Funcao main: i = %d\n", i);
    return 0;
}
```

Resultados

- O resultado da execução deste programa é o seguinte:

Resultados

- O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

Resultados

- O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

- Observe que a variável global `i` recebe o valor `0` no início da função `main`.

Resultados

- O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

- Observe que a variável global `i` recebe o valor `0` no início da função `main`.
- A função `soma1` ao executar um comando que aumenta o valor de `i` em uma unidade está aumentando a variável global.

Resultados

- O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

- Observe que a variável global `i` recebe o valor `0` no início da função `main`.
- A função `soma1` ao executar um comando que aumenta o valor de `i` em uma unidade está aumentando a variável global.
- Em seguida vemos que a função `sub1` define uma variável local também chamada `i` e, portanto, a alteração feita por esta função somente modifica esta variável.

Resultados

- O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

- Observe que a variável global `i` recebe o valor `0` no início da função `main`.
- A função `soma1` ao executar um comando que aumenta o valor de `i` em uma unidade está aumentando a variável global.
- Em seguida vemos que a função `sub1` define uma variável local também chamada `i` e, portanto, a alteração feita por esta função somente modifica esta variável.
- Finalmente, a função `main` imprime o valor final da variável global.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.
- Eles são criados no início da execução da função e destruídos no final.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.
- Eles são criados no início da execução da função e destruídos no final.
- Parâmetros são valores que as funções recebem da função que a chamou.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.
- Eles são criados no início da execução da função e destruídos no final.
- Parâmetros são valores que as funções recebem da função que a chamou.
- Normalmente os parâmetros são inicializados durante a chamada da função, pois para isto foram criados.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.
- Eles são criados no início da execução da função e destruídos no final.
- Parâmetros são valores que as funções recebem da função que a chamou.
- Normalmente os parâmetros são inicializados durante a chamada da função, pois para isto foram criados.
- No entanto, as variáveis que atuam como parâmetros são iguais a todas as outras e podem ser modificadas, operadas, etc, sem nenhuma restrição.

Parâmetros Formais

- As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais.
- Eles são criados no início da execução da função e destruídos no final.
- Parâmetros são valores que as funções recebem da função que a chamou.
- Normalmente os parâmetros são inicializados durante a chamada da função, pois para isto foram criados.
- No entanto, as variáveis que atuam como parâmetros são iguais a todas as outras e podem ser modificadas, operadas, etc, sem nenhuma restrição.
- Parâmetros podem ser passados para funções de duas maneiras: passagem por valor ou passagem por referência.

Section Summary

Passagem por valor

Na passagem por valor uma cópia do valor do argumento é passado para a função. Neste caso a função que recebe este valor, ao fazer modificações no parâmetro, não estará alterando o valor original que somente existe na função que chamou.

Exemplo

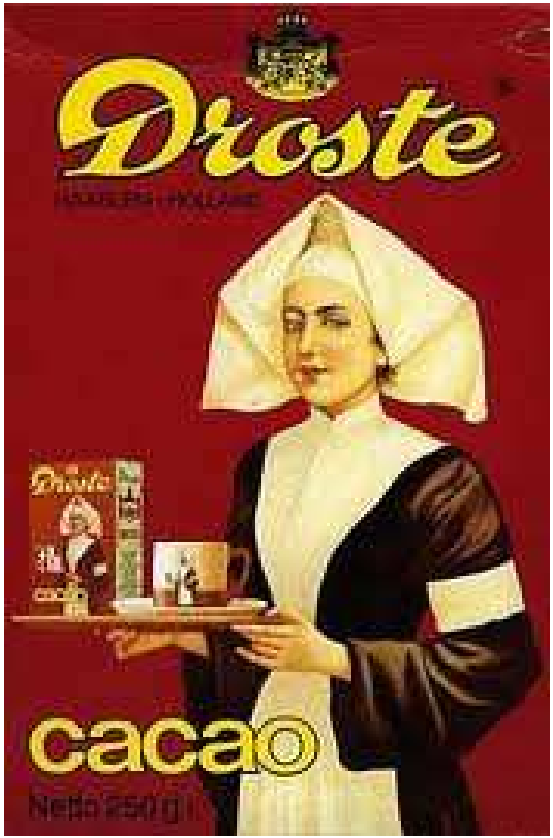
```c
#include <stdio.h>

float Eleva(float a, int b) {
    float res = 1.0;
    for ( ; b>0; b--) res *= a;
    return res;
}
int main() {
    float numero;
    int potencia;

    puts("Entre com um numero");
    scanf("%f", &numero);
    puts("Entre com a potencia");
    scanf("%d", &potencia);
    printf("%f Elevado a %d e igual a %f\n",
        numero, potencia, Eleva(numero, potencia));
    return 0;
}
```

Outro exemplo

```c
#include <stdio.h>
void trocar(int a, int b)
{
    int temp;
    temp = a; a = b; b = temp;
}

int main(int argc, char *argv[])
{
    int a = 10, b = 20;
    trocar(a, b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Section Summary

Recursão



Recursão

- Funções em C podem ser usadas recursivamente, isto é uma função pode chamar a si mesmo.

Recursão

- Funções em C podem ser usadas recursivamente, isto é uma função pode chamar a si mesmo.
- É como se procurássemos no dicionário a definição da palavra recursão e encontrássemos o seguinte texto:
  recursão: s.f. Veja a definição em recursão

Recursão

- Funções em C podem ser usadas recursivamente, isto é uma função pode chamar a si mesmo.
- É como se procurássemos no dicionário a definição da palavra recursão e encontrássemos o seguinte texto:
  recursão: s.f. Veja a definição em recursão
- Um exemplo simples de função que pode ser escrita com chamadas recursivas é o fatorial de um número inteiro. O fatorial de um número, sem recursão, é definido como

$$n! = n * (n-1) * (n-2) * \cdots * 2 * 1$$

Fatorial sem recursão

```c
unsigned long int fat (unsigned long int num) {
    unsigned long int fato=1, i;

    for (i=num; i>1; i--) fato = fato * i;

    return fato;
}
```

Fatorial com recursão

Alternativamente, o fatorial pode ser definido como o produto deste número pelo fatorial de seu predecessor, ou seja

$$n! = n * (n - 1)!$$

Fatorial com recursão

```
unsigned long int fat (unsigned long int num) {
if (num == 0)
        return 1;
    else
        return num * fat (num-1);
}
```

Comentários

- Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido.

Comentários

- Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido.
- No caso da função `fat` o processo é interrompido quando o valor do número passado como parâmetro vale 0.

Comentários

- Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido.
- No caso da função `fat` o processo é interrompido quando o valor do número passado como parâmetro vale 0.
- É importante notar que recursão não traz obrigatoriamente economia de memória porque os valores sendo processados tem de ser mantidos em pilhas.

Comentários

- Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido.
- No caso da função `fat` o processo é interrompido quando o valor do número passado como parâmetro vale 0.
- É importante notar que recursão não traz obrigatoriamente economia de memória porque os valores sendo processados tem de ser mantidos em pilhas.
- Nem será mais rápido, e as vezes pode ser até mais lento porque temos o custo de chamada as funções.

Comentários

- Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido.
- No caso da função `fat` o processo é interrompido quando o valor do número passado como parâmetro vale 0.
- É importante notar que recursão não traz obrigatoriamente economia de memória porque os valores sendo processados tem de ser mantidos em pilhas.
- Nem será mais rápido, e as vezes pode ser até mais lento porque temos o custo de chamada as funções.
- As principais vantagens da recursão são códigos mais compactos e provavelmente mais fáceis de serem lidos.

Potenciação x^n

```
float Elevar(float x, int n) {
    if (n <= 1) {
        return x;
    }
    else {
        return x * Elevar(x, b-1);
    }
}
```

Antes ou Depois?

```c
#include <stdio.h>
void imprime1(int n) {
    if (n < 0) return;
    printf("%2d, ", n);
    imprime1(n-1);
}
void imprime2(int n) {
    if (n < 0) return;
    imprime2(n-1);
    printf("%2d, ", n);
}
int main (void) {
    imprime1(10);
    printf("\n");
    imprime2(10);
    printf("\n");
    return 0;
}
```

Recursão o que é?

Recursão

É um processo que quebra um problema em problemas menores, que serão quebrados em problemas menores ainda, até que chegamos no menor problema possível e na sua solução (caso básico), neste ponto começamos a retornar começando pelo caso básico.

Passos para escrever uma função recursiva

- Escreva um protótipo da função recursiva.

Passos para escrever uma função recursiva

- Escreva um protótipo da função recursiva.
- Escreva um comentário que descreve o que a função deve fazer.

Passos para escrever uma função recursiva

- Escreva um protótipo da função recursiva.
- Escreva um comentário que descreve o que a função deve fazer.
- Determine o caso base (pode haver mais de um) e a solução deste caso.

Passos para escrever uma função recursiva

- Escreva um protótipo da função recursiva.
- Escreva um comentário que descreve o que a função deve fazer.
- Determine o caso base (pode haver mais de um) e a solução deste caso.
- Determine quel é o problema menor do que o atual a ser resolvido.

Passos para escrever uma função recursiva

- Escreva um protótipo da função recursiva.
- Escreva um comentário que descreve o que a função deve fazer.
- Determine o caso base (pode haver mais de um) e a solução deste caso.
- Determine quel é o problema menor do que o atual a ser resolvido.
- Use a solução do problema menor para resolver o problema maior.

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:

```
int soma(int vetor[], int n);
```

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:
  `int soma(int vetor[], int n);`
- Escreva um comentário que descreve o que a função deve fazer:
  "A função deve somar n elementos de um vetor inteiro".

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:
  `int soma(int vetor[], int n);`
- Escreva um comentário que descreve o que a função deve fazer:
  "A função deve somar n elementos de um vetor inteiro".
- Determine o caso base (pode haver mais de um) e a solução deste caso:
  O caso base seria a soma de um vetor do elemento que restou que é ele próprio.

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:
  `int soma(int vetor[], int n);`
- Escreva um comentário que descreve o que a função deve fazer:
  "A função deve somar n elementos de um vetor inteiro".
- Determine o caso base (pode haver mais de um) e a solução deste caso:
  O caso base seria a soma de um vetor do elemento que restou que é ele próprio.
- Determine quel é o problema menor a ser resolvido:
  O problema menor a ser resolvido é `soma(vetor, n-1)`.

Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:
  `int soma(int vetor[], int n);`
- Escreva um comentário que descreve o que a função deve fazer:
  "A função deve somar n elementos de um vetor inteiro".
- Determine o caso base (pode haver mais de um) e a solução deste caso:
  O caso base seria a soma de um vetor do elemento que restou que é ele próprio.
- Determine quel é o problema menor a ser resolvido:
  O problema menor a ser resolvido é `soma(vetor, n-1)`.
- Use a solução do problema menor para resolver o problema maior.
  vetor[n-1] + soma(vetor, n-1)

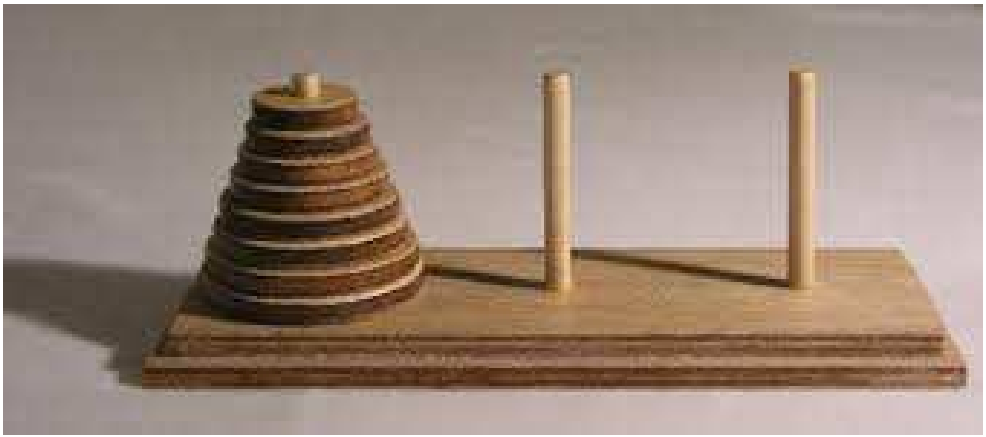
Aplicando os passos

- Considere que vamos resolver a soma de um vetor de n elementos.
- Escreva um protótipo da função recursiva:
  `int soma(int vetor[], int n);`
- Escreva um comentário que descreve o que a função deve fazer:
  "A função deve somar n elementos de um vetor inteiro".
- Determine o caso base (pode haver mais de um) e a solução deste caso:
  O caso base seria a soma de um vetor do elemento que restou que é ele próprio.
- Determine quel é o problema menor a ser resolvido:
  O problema menor a ser resolvido é `soma(vetor, n-1)`.
- Use a solução do problema menor para resolver o problema maior.
  vetor[n-1] + soma(vetor, n-1)
- Lembre-se que em C o elemento 1 está na posição 0, portanto o elemento n está na posição $n - 1$.

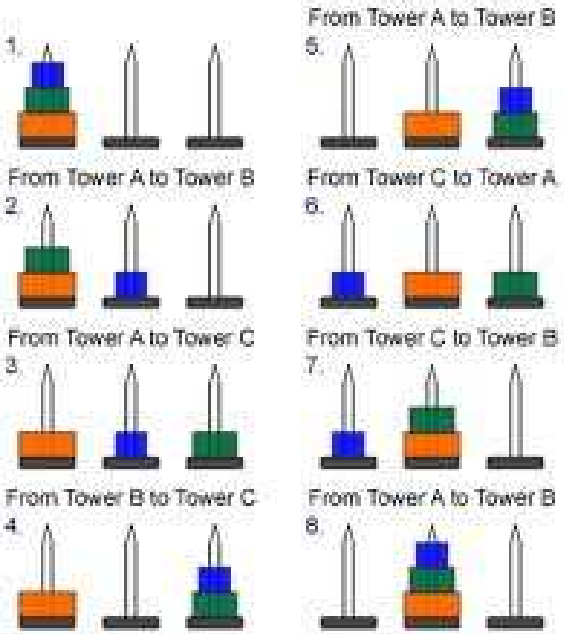
Solução

```c
#include <stdio.h>

int soma(int v[], int n) {
    if (n == 1) {
        return v[0];
    }
    else {
        return v[n-1] + soma(v, n-1);
    }
}
int main (void) {
    int v[5] = {7, 2, -2, 4, 15};

    printf("%d\n", soma(v, 5));
    return 0;
}
```

Torres de Hanói



Torres de Hanói



Algoritmo recursivo

- if n==1 mova este disco de A para B e pare.

Algoritmo recursivo

- if n==1 mova este disco de A para B e pare.
- Mova os n-1 discos superiores de A para C usando B como auxiliar.

Algoritmo recursivo

- if n==1 mova este disco de A para B e pare.
- Mova os n-1 discos superiores de A para C usando B como auxiliar.
- Mova o disco restante de A para B.

Algoritmo recursivo

- if n==1 mova este disco de A para B e pare.
- Mova os n-1 discos superiores de A para C usando B como auxiliar.
- Mova o disco restante de A para B.
- Mova os n-1 discos de C to B, usando A como auxiliar.

Algoritmo recursivo

- if n==1 mova este disco de A para B e pare.
- Mova os n-1 discos superiores de A para C usando B como auxiliar.
- Mova o disco restante de A para B.
- Mova os n-1 discos de C to B, usando A como auxiliar.
- MÁGICA! NÃO PRECISA SABER COMO MOVER.....

Algoritmo em C

```
#include <stdio.h>
void towers(int nDiscos, char dePino, char paraPino,
                         char auxPino) {
/* Se um disco somente, mova-o e retorne */
    if(nDiscos==1) {
        printf("Move disco 1 do pino %c para pino %c↩
           \n",
               dePino, paraPino);
    return;
    }
/*  Move n-1 discos superiores de A para C, usando B↩
    */
    towers(nDiscos-1, dePino, auxPino, paraPino);

/* Move discos restantes de A para B */
    printf("Move disco %d do pino %c para pino %c\n"↩
       ,
            nDiscos, dePino, paraPino);
```

Algoritmo em C

```
int main() {
    int n;
    printf("Entre o numero de discos: ");
    scanf("%d",&n);
    printf("O algoritmo faz os seguintes movimentos:\↩
        n");
    towers(n, 'A', 'B', 'C');
    printf("\n");
    return 0;
}
```

Passagem por Referência

Na passagem por referência o que é passado para a função é o endereço do parâmetro e, portanto, a função que recebe pode, através do endereço, modificar o valor do argumento diretamente na função que chamou. Para a passagem de parâmetros por referência é necessário o uso de ponteiros.

Isto será visto na próxima apresentação.

Passagem de Vetores e Matrizes

Matrizes são um caso especial e exceção a regra que nomes de variáveis como parâmetros indicam passagem por valor. Como veremos mais adiante, o nome de um vetor corresponde ao endereço do primeiro elemento do array. Quando um nome de vetor é passado como parâmetro o endereço do primeiro elemento é passado.

Passagem de Vetores e Matrizes

- Existem basicamente três maneiras de declarar um vetor como um parâmetro de uma função.

Passagem de Vetores e Matrizes

- Existem basicamente três maneiras de declarar um vetor como um parâmetro de uma função.
- Na primeira ele é declarado segundo as regras de declaração de uma variável do tipo vetor.

Exemplo primeiro caso

```
#include <stdio.h>
#define DIM 80
int conta (char v[DIM], char c);
int main() {
    char c, linha[DIM];
    int maiusculas[26], minusculas[26];
    gets (linha);
    for (c = 'a'; c <= 'z'; c++)
        minusculas[c-'a'] = conta(linha, c);
    for (c = 'A'; c <= 'Z'; c++)
        maiusculas[c-'A'] = conta(linha, c);
    for (c = 'a'; c <= 'z'; c++)
        if (minusculas[c-'a'])
           printf("%c apareceu %d vezes\n", c, ↩
              minusculas[c-'a']);
     for (c = 'A'; c <= 'Z'; c++)
        if (maiusculas[c-'A'])
           printf("%c apareceu %d vezes\n", c, ↩
              maiusculas[c-'A']);
```

Exemplo primeiro caso

```c
#include<stdio.h>
#define DIM 80
int conta (char v[DIM], char c) {
    int i=0, vezes=0;
    while (v[i] != '\0')
        if (v[i++] == c) vezes++;
    return vezes;
}
```

Segundo caso

- Uma outra maneira, leva em conta que apenas o endereço do vetor é passado.

Segundo caso

- Uma outra maneira, leva em conta que apenas o endereço do vetor é passado.
- Neste modo o parâmetro é declarado como um vetor sem dimensão.

Segundo caso

- Uma outra maneira, leva em conta que apenas o endereço do vetor é passado.
- Neste modo o parâmetro é declarado como um vetor sem dimensão.
- Isto é perfeitamente possível porque a função somente precisa receber o endereço onde se encontra o vetor.

Segundo caso

- Uma outra maneira, leva em conta que apenas o endereço do vetor é passado.
- Neste modo o parâmetro é declarado como um vetor sem dimensão.
- Isto é perfeitamente possível porque a função somente precisa receber o endereço onde se encontra o vetor.
- C não confere onde estão os limites de vetores e portanto a função precisa do endereço inicial do vetor e uma maneira de descobrir o final do vetor.

Segundo caso

- Uma outra maneira, leva em conta que apenas o endereço do vetor é passado.
- Neste modo o parâmetro é declarado como um vetor sem dimensão.
- Isto é perfeitamente possível porque a função somente precisa receber o endereço onde se encontra o vetor.
- C não confere onde estão os limites de vetores e portanto a função precisa do endereço inicial do vetor e uma maneira de descobrir o final do vetor.
- Esta maneira pode ser, por exemplo, uma constante, ou o caractere '\0' em um vetor de caracteres.

Exemplo segundo caso

```c
#include <stdio.h>
#define DIM 6
void Le_vetor (int v[], int tam);
void Imprime_vetor (int v[], int tam);
void Inverte_vetor (int v[], int tam);

int main() {
    int v[DIM];

    Le_vetor(v, DIM);
    Imprime_vetor (v, DIM);
    Inverte_vetor (v, DIM);
    Imprime_vetor (v, DIM);
    return 0;
}
```

Exemplo segundo caso

```
void Le_vetor (int v[], int tam) {
    int i;
    for ( i = 0; i < tam; i++) {
        printf("%d = ? ", i); scanf("%d", &v[i]);
    }
}
void Imprime_vetor (int v[], int tam) {
    int i;
    for (i = 0; i < tam; i++)
        printf("%d = %d\n", i, v[i]);
}
void Inverte_vetor (int v[], int tam) {
    int i, temp;
    for (i = 0; i < tam/2; i++){
        temp = v[i];
        v[i] = v[tam-i-1];
        v[tam-i-1] = temp;
    }
}
```

Passagem de Vetores e Matrizes

A terceira maneira de passagem de parâmetros implica no uso de ponteiros, o que somente iremos ver no próximo capítulo.

Section Summary

return

O comando **return** é usado para retornar o valor calculado para a função que chamou. Qualquer expressão pode aparecer no comando, que tem a a seguinte forma geral:

```
return expressão;
```

Comentários

- A função que chamou é livre para ignorar o valor retornado.

Comentários

- A função que chamou é livre para ignorar o valor retornado.
- Além disso a função pode não conter o comando e portanto nenhum valor é retornado.

Comentários

- A função que chamou é livre para ignorar o valor retornado.
- Além disso a função pode não conter o comando e portanto nenhum valor é retornado.
- Neste caso a função termina quando o último comando da função é executado.

Comentários

- A função que chamou é livre para ignorar o valor retornado.
- Além disso a função pode não conter o comando e portanto nenhum valor é retornado.
- Neste caso a função termina quando o último comando da função é executado.
- Quando o comando **return** não existe, o valor de retorno é considerado indefinido.

Comentários

- A função que chamou é livre para ignorar o valor retornado.
- Além disso a função pode não conter o comando e portanto nenhum valor é retornado.
- Neste caso a função termina quando o último comando da função é executado.
- Quando o comando **return** não existe, o valor de retorno é considerado indefinido.
- As funções que não retornam valores devem ser declaradas como do tipo **void**.

Section Summary

Comentários

- A função `main` como todas as funções também pode ter parâmetros.

Comentários

- A função `main` como todas as funções também pode ter parâmetros.
- Como a função `main` é sempre a primeira a ser executada, os parâmetros que ela recebe são fornecidos pela linha de comando ou pelo programa que iniciou a sua execução.

Comentários

- A função `main` como todas as funções também pode ter parâmetros.
- Como a função `main` é sempre a primeira a ser executada, os parâmetros que ela recebe são fornecidos pela linha de comando ou pelo programa que iniciou a sua execução.
- No caso da função `main` são usados dois argumentos especiais **int** `argc` e **char** `**argv`.

int argc

- O primeiro argumento, `argc`, é uma variável inteira que indica quantos argumentos foram fornecidos para a função.

int argc

- O primeiro argumento, `argc`, é uma variável inteira que indica quantos argumentos foram fornecidos para a função.
- Observar que `argc` vale sempre pelo menos 1, porque o nome do programa é sempre o primeiro argumento fornecido ao programa.

int argc

- O primeiro argumento, `argc`, é uma variável inteira que indica quantos argumentos foram fornecidos para a função.
- Observar que `argc` vale sempre pelo menos 1, porque o nome do programa é sempre o primeiro argumento fornecido ao programa.
- A partir do segundo argumento em diante é que aparecem os argumentos fornecidos.

char **argc

- O outro parâmetro é um vetor de cadeias de caracteres, e portanto, caso sejam fornecidos números, estes devem ser convertidos para o formato requerido.

char **argc

- O outro parâmetro é um vetor de cadeias de caracteres, e portanto, caso sejam fornecidos números, estes devem ser convertidos para o formato requerido.
- Cada um dos argumentos do programa é um elemento deste vetor.

char **argc

- O outro parâmetro é um vetor de cadeias de caracteres, e portanto, caso sejam fornecidos números, estes devem ser convertidos para o formato requerido.
- Cada um dos argumentos do programa é um elemento deste vetor.
- A primeira linha da função `main` pode ter a seguinte forma
  void `main (`**int** `argc, `**char** `**argv)`

char **argc

- O outro parâmetro é um vetor de cadeias de caracteres, e portanto, caso sejam fornecidos números, estes devem ser convertidos para o formato requerido.
- Cada um dos argumentos do programa é um elemento deste vetor.
- A primeira linha da função `main` pode ter a seguinte forma
  void `main (int argc, char **argv)`
- ou **void** `main (int argc, char *argv[])`

Exemplo I argc, argc

```c
#include <stdio.h>
int main (int argc, char **argv)
{
    printf("Ola, eu sou %s\n", argv[0]);
    return 0;
}
```

Exemplo II argc, argv

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

int fat (int num);

int main(int argc, char *argv[]) {
    int numero, fatorial, i;

    if (argc < 2) {
        printf("Uso: %s num1 num2 ... .\n", argv[0])↩
            ;
        return 1;
    }
    for (i=1; i<argc; i++) {
        numero = atoi(argv[i]); /* Ascii TO Int */
        fatorial = fat(numero);
        printf("Fat de %d = %d.\n", numero, fatorial↩
            );
    }
```

Exemplo II argc, argv

```
int fat (int num) {
    if (num == 0)
        return 1;
    else
        return num * fat (num-1);
}
```

The End