



CEFET – RJ / Campus Maria da Graça
Centro Federal de Educação Tecnológica
Celso Suckow da Fonseca – Rio de Janeiro



Prof. Cristiano Fuschilo
fuschilo@gmail.com

Estrutura de Dados
3º Período



Bacharelado em
Sistemas de
Informação

Funções



Funções



- Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.
- Uma função no C tem a seguinte forma geral:

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
    corpo_da_função
}
```
- O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo int, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:
`tipo nome1, tipo nome2, ... , tipo nomeN`



Return



- O comando `return` tem a seguinte forma geral:
- `return valor_de_retorno;` ou `return;`
- Digamos que uma função está sendo executada. Quando se chega a uma declaração `return` a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.
- Uma função pode ter mais de uma declaração `return`. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração `return`.





Exemplos de uso do return

```
#include <stdio.h>
int Square (int a) {
    return (a*a);
}
int main (){
    int num;
    printf ("Entre com um numero: ");
    scanf ("%d",&num);
    num=Square(num);
    printf ("\n\nO seu quadrado vale: %d\n",num);
    return 0;
}
```



Exemplos de uso do return



```
#include <stdio.h>
int EPar (int a) {
    if (a%2)      /* Verifica se a e divisivel por dois */
        return 0;    /* Retorna 0 se nao for divisivel */
    else
        return 1;    /* Retorna 1 se for divisivel */
}
int main () {
    int num;
    printf ("Entre com numero: ");
    scanf ("%d",&num);
    if (EPar(num))
        printf ("\n\nO numero e par.\n");
    else
        printf ("\n\nO numero e impar.\n");
    return 0;
}
```



Retorno de Valores



- É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas não podemos fazer:

```
func(a,b)=x; /* Errado! */
```

- No segundo exemplo vemos o uso de mais de um return em uma função.
- Fato importante: se uma função retorna um valor você não precisa aproveitar este valor. Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função printf() retorna um inteiro que nós nunca usamos para nada. Ele é descartado.



Protótipos de Funções



- Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função `main()`. Isto é, as funções estão fisicamente antes da função `main()`. Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função `main()`, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto as funções foram colocadas antes da função `main()`: quando o compilador chegasse à função `main()` ele já teria compilado as funções e já saberia seus formatos.
- Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?





Protótipos de funções

- Em C++ uma função só pode ser usada se esta já foi declarada. Em C, o uso de uma função não declarada geralmente causava uma warning do compilador, mas não um erro. Em C++ isto é um erro.
- Para usar uma função que não tenha sido definida antes da chamada — tipicamente chamada de funções entre módulos —, é necessário usar protótipos. Os protótipos de C++ incluem não só o tipo de retorno da função, mas também os tipos dos parâmetros:
`void f (int a, float b); // protótipo da função f`
- Uma tentativa de utilizar uma função não declarada gera um erro de símbolo desconhecido.





Como manter a coerência?

- A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:
 - `tipo_de_retorno nome_da_função (declaração_de_parâmetros);`
 - onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:





Exemplo

```
#include <stdio.h>
float Square (float a);
int main ()
{
    float num;
    printf ("Entre com um numero: ");
    scanf ("%f",&num);
    num=Square(num);
    printf ("\n\nO seu quadrado vale: %f\n",num);
    return 0;
}
float Square (float a)
{
    return (a*a);
}
```



Explicando



- Observe que a função `Square()` está colocada depois de `main()`, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.
- Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!





O Tipo void

- Agora vamos ver o único tipo da linguagem C que não detalhamos ainda: o void. Em inglês, void quer dizer vazio e é isto mesmo que o void é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros! Podemos agora escrever o protótipo de uma função que não retorna nada:

```
void nome_da_função (declaração_de_parâmetros);
```

- Numa função, como a acima, não temos valor de retorno na declaração return. Aliás, neste caso, o comando return não é necessário na função.



void



- Podemos, também, fazer funções que não têm parâmetros:
- tipo_de_retorno nome_da_função (void);
- ou, ainda, que não tem parâmetros e não retornam nada:
- void nome_da_função (void);



Exemplo de funções que usam o tipo void

```
#include <stdio.h>
void Mensagem (void);
int main ()
{
    Mensagem();
    printf ("\tDiga de novo:\n");
    Mensagem();
    return 0;
}
void Mensagem (void)
{
    printf ("Olá! Eu estou vivo.\n");
}
```



Funções que não recebem parâmetros



- Em C puro, um protótipo pode especificar apenas o tipo de retorno de uma função, sem dizer nada sobre seus parâmetros. Por exemplo,
- `float f();` // em C, não diz nada sobre os parâmetros de `f` é um protótipo incompleto da função `f`.
- Na realidade, esta é uma das diferenças entre C e C++. Um compilador de C++ interpretará a linha acima como o protótipo de uma função que retorna um `float` e não recebe nenhum parâmetro. Ou seja, é exatamente equivalente a uma função (`void`):
- `float f();` // em C++ é o mesmo que `float f(void);`



Arquivos-Cabeçalhos



- São aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em .h. A extensão .h vem de header (cabeçalho em inglês).
- Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm protótipos de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto.
- O corpo das funções cujos protótipos estão no arquivo-Cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.
- Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-Cabeçalhos e incluí-los também.





Exemplo

- Suponha que a função 'int EPar(int a)', seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado por exemplo de 'funcao.h' teremos a seguinte declaração:

```
int EPar(int a);
```

- O código da função será escrito num arquivo a parte. Vamos chamá-lo de 'funcao.c'. Neste arquivo teremos a definição da função:

```
int EPar (int a)
{
    if (a%2)          /* Verifica se a é divisível por dois */
        return 0;
    else
        return 1;
}
```





Programa Principal

Vamos chamar este arquivo aqui de 'princip.c'.

```
#include <stdio.h>
#include "funcao.h"
void main ()
{
    int num;
    printf ("Entre com numero: ");
    scanf ("%d",&num);
    if (EPar(num))
        printf ("\n\nO numero e par.\n");
    else
        printf ("\n\nO numero e impar.\n");
}
```





Sobrecarga de Funções

- Em C++ é possível definir duas funções com o mesmo nome desde que a quantidade ou o tipo de parâmetros sejam diferentes. Isto é, podemos dar o mesmo nome a duas ou mais funções desde que estas possuam um número diferente de parâmetros ou parâmetros de tipos diferentes.
-
- Esta característica é designada por **sobrecarga de funções** (*function overloading*).



Sobrecarga de Funções - Ex



```
#include <stdio.h>                                //O resultado será:  
  
int opera(int a, int b){                          //10  
    return (a * b);  
}  
  
float opera(float a, float b){                     //2.5  
    return (a / b);  
}  
  
int main (){  
    int x = 5, y = 2;  
    float n = 5.0, m = 2.0;  
  
    printf("%d\n\n%d", opera(x, y), opera(n, m) );  
    return 0;  
}
```





Explicando

- No exemplo anterior definimos duas funções com o mesmo nome, opera, mas uma delas aceita dois parâmetros do tipo int e a outra dois parâmetros do tipo float. O compilador sabe qual a função que pretendemos invocar analisando o tipo dos argumentos utilizados quando chamamos a função. Se for chamada com dois inteiros, utiliza a função que possui dois inteiros na sua definição. Se for chamada com dois reais, utiliza a função que possui dois reais na sua definição.
- Note que as duas versões da função opera do exemplo anterior realizam operações diferentes. A primeira multiplica os valores dos dois parâmetros, enquanto a segunda divide-os. Isto é o comportamento da função opera depende do tipo dos argumentos.
- Note por fim que uma função não pode ser sobre carregada apenas à custa do tipo de retorno. Isto é, o compilador não permite que duas funções difiram apenas no tipo de retorno.



Funções Recursivas



- A recursão é uma técnica que define um problema em termos de uma ou mais versões menores deste mesmo problema.
- A recursão pode ser utilizada sempre que for possível expressar a solução de um problema em função do próprio problema.
- Uma função é dita recursiva quando dentro do seu código existe uma chamada para si mesma.





Exemplo Funções Recursivas

- Calcular o Fatorial de um número N inteiro qualquer. Se formos analisar a forma de cálculo temos:

$$\text{fat}(n) = \begin{cases} 1, & \text{se } n = 0 \text{ (solução trivial)} \\ n \times \text{fat}(n - 1), & \text{se } n > 0 \text{ (solução recursiva)} \end{cases}$$

- Logo, temos que:

$$\text{fat}(5) = 5 \times \text{fat}(4)$$

$$\text{fat}(4) = 4 \times \text{fat}(3)$$

$$\text{fat}(3) = 3 \times \text{fat}(2)$$

$$\text{fat}(2) = 2 \times \text{fat}(1)$$

$$\text{fat}(1) = 1 \times \text{fat}(0)$$

$$\text{fat}(0) = 1$$



Fatorial - Não Recursiva e Recursiva



```
1 #include<stdio.h>
2 int fatorialrec(int num)
3 {
4     if (num == 0) {
5         return 1;
6     }
7     else {
8         return num * fatorialrec(num-1);
9     }
10 }
11 int fatorialsemrec(int num)
12 {
13     int f, i;
14     if (num == 0) {
15         return 1;
16     }
17     else {
18         f = 1;
19         for(i= num; i > 1; i--){
20             f = f * i;
21         }
22         return f;
23     }
24 }
25 int main() {
26     int num;
27     num = 5;
28     printf("\nfatR(%d) = %d", num, fatorialrec(num));
29     printf("\n\nfatS(%d) = %d", num, fatorialsemrec(num));
30
31 }
```



Exercício



- Faça um programa em C para calcular a soma dos n primeiros números dados pelo usuário na entrada. Criar duas funções soma (uma recursiva e a outra não recursiva) que recebe como parâmetro de entrada o número lido.
- Lembre-se:

$$\text{somarec}(n) = \begin{cases} 1, & \text{se } n = 1 \text{ (solução trivial)} \\ n + \text{somarec}(n - 1), & \text{se } n > 1 \text{ (solução recursiva)} \end{cases}$$



Estruturas de Dados



Estrutura de Dados



- Muitas vezes precisamos compor os dados para formar estruturas de dados complexas
- Variáveis compostas homogêneas (Arrays)
 - Conjunto de variáveis de mesmo tipo
- Variáveis compostas heterogêneas
 - Conjunto de variáveis de tipos diferentes
- Chamadas de:
 - Estruturas (Struct)



Aplicação de Estruturas (1)



- Estruturas podem ser usadas para armazenar informações relacionadas
- Exemplo 1: Produto

Livro (char[11])	L	i	n	g	u	a	g	e	m		C
Preco (float)											59,9000
Autor (char[11])	D	.		R	i	t	c	h	i	e	



Exemplo 2: Ficha de Cliente (Cadastro)



Nome (char[10])	H	e	I	e	n	a				
Idade (int)			3	0						
Telefone (int)			5	5	5	5				
Cidade (char[10])	S	a	o		P	a	u	l	o	



Definição de uma estrutura (registro) em C

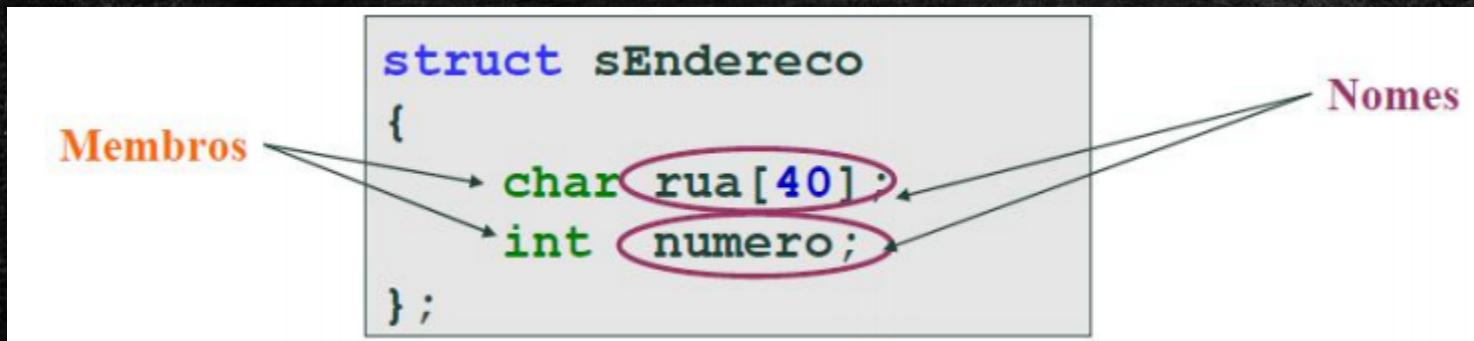
```
struct identificacao_da_estrutura {  
    tipo1 nome1;  
    tipo2 nome2; ...  
    tipoN nomeN;  
};
```

- Uma estrutura é um tipo de dado cujo formato é definido pelo programador



Estruturas

- Variáveis compostas heterogêneas (estruturas) são um conjunto de variáveis de tipos diferentes que são logicamente relacionadas.
- Essas variáveis compartilham o mesmo identificador e ocupam posições consecutivas de memória.
- Para as variáveis de uma estrutura:
 - Elas são denominadas membros;
 - São identificadas por nomes.





Exemplo - Declaração

- Vamos criar uma estrutura de endereço, que possa ser usada como se fosse um tipo de dado posteriormente
- Este código deve vir no início do programa, após os “includes”

```
struct sEndereco {  
    char rua[40];  
    int numero;  
    char cidade[30];  
    char estado[2];  
    long int CEP;  
};
```



Declaração de uma variável do tipo identificacao_da_estrutura



```
struct identificacao_da_estrutura nome_da_variavel;
```

```
struct identificacao_da_estrutura nome_da_variavel;
```





Exemplo - Programa

- Vamos criar uma programa que use a estrutura sEndereco e atribua Valores a todas as variáveis da estrutura

```
int main( )
{
    //cria variavel ender1 como struct sEndereco
    struct sEndereco ender1;

    strcpy(ender1.rua, "Rua 7 de Setembro");
    ender1.numero = 405;
    strcpy(ender1.cidade, "Goiania");
    strcpy(ender1.estado, "GO");
    ender1.CEP = 06599604;
}
```



Declarando, atribuindo, imprimindo



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

struct sRetangulo
{
    float altura, largura;
};

int main(void)
{
    struct sRetangulo ret1;

    ret1.altura = 10;
    printf("Digite o valor da largura: ");
    scanf("%f", &ret1.largura);

    printf("Altura: %.1f", ret1.altura);
    printf("Largura: %.1f", ret1.largura);

    return 0;
}
```





Estruturas Rotuladas

- Estruturas rotuladas criam um “rótulo” que pode ser referenciado posteriormente no código.
- Criação de rótulos.

```
struct rotulo_da_estrutura
{
    tipo1 nome1;
    tipo2 nome2;
    ...
    tipoN nomeN;
};
```



Estruturas Rotuladas

```
#include <stdio.h>

struct sHora {
    int hora;
    int minuto;
    int segundo;
};

int main(void) {
    struct sHora H;
    H.hora = 10;
    H.minuto = 15;
    H.segundo = 30;
    printf("%d:%d:%d", H.hora, H.minuto, H.segundo);
    return 0;
}
```





Estruturas Rotuladas e Nomeadas

- Uma estrutura rotulada e nomeada pode ser definida da seguinte forma:

```
#include <stdio.h>

typedef struct Hora {
    int hora;
    int minuto;
    int segundo;
} THora;

int main(void) {
    THora H;
    H.hora = 10;
    H.minuto = 15;
    H.segundo = 30;
    printf("%d:%d:%d", H.hora, H.minuto, H.segundo);
    return 0;
}
```





Estruturas Aninhadas

- Estruturas em que um ou mais de seus membros também sejam estruturas.

```
typedef struct rotulo_estrutura1 {  
    tipo1 nome1;  
    tipoN nomeN;  
} id_estrutura1;  
  
typedef struct rotulo_estrutura2 {  
    id_estrutura1 nome;  
    tipoN nomeN;  
} id_estrutura2;
```



Exemplo



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

typedef struct sHora
{
    int hora;
    int minuto;
    int segundo;
} Hora;

typedef struct sRelogio
{
    Hora H;
    char modelo[10];
} Relogio;
```



Estruturas Aninhadas



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

typedef struct sHora
{
    int hora;
    int minuto;
    int segundo;
} Hora;

typedef struct sRelogio
{
    Hora H;
    char modelo[10];
} Relogio;
```

```
int main(void)
{
    Relogio r1;

    r1.H.hora = 10;
    r1.H.minuto = 15;
    r1.H.segundo = 30;

    strcpy(r1.modelo, "Cassio");

    printf("Modelo: %s\n", r1.modelo);

    printf("%d:%d:%d", r1.H.hora,
           r1.H.minuto,
           r1.H.segundo);

    return 0;
}
```



Vetores e Estruturas



- É possível combinar vetores e estruturas para criação de diferentes estruturas de dados.
- Podemos ter uma estrutura contendo um membro do tipo vetor, ou;
- Criar um vetor cujo os elementos sejam estruturas





Declarando vetor de Estruturas

- Dada a estrutura listada abaixo:

```
struct lista {  
    char titulo[30];  
    char autor[30];  
    int regnum;  
    double preco;  
};
```

Membros do
tipo vetor

- Declare um vetor com 50 elementos do tipo lista





Declarando Vetores de Estruturas

```
struct lista livro[50];
```

- `livro` é um vetor de 50 elementos.
- Cada elemento do vetor é uma estrutura do tipo `struct lista`
- O que significa `livro[0]`, `livro[1]`, `livro[2]`, etc?

** Por meio dessa instrução o compilador providencia espaço de memória para 50 estruturas do tipo `struct lista`.





Exemplo

```
...
struct sEndereco
{
    char rua[40];
    int numero;
};

int main(void) {

    struct sEndereco listaend[5];

    listaend[0].numero = 100;

    strcpy(listaend[3].rua,"Av. Brasil");
    ...
}
```

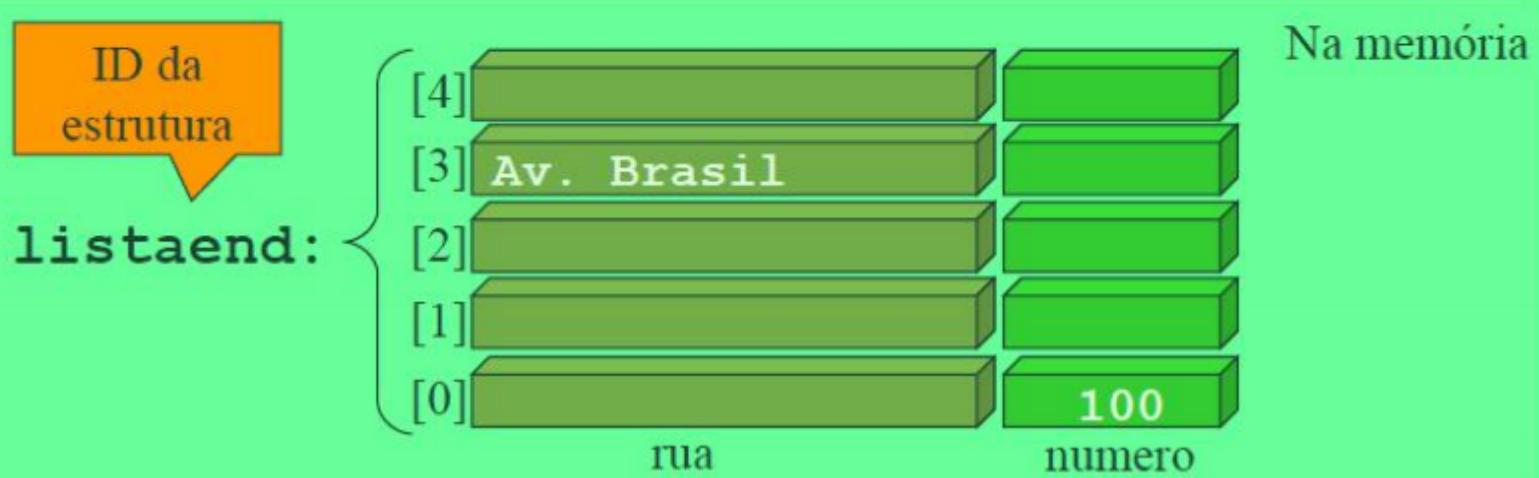


Trecho de exemplo



```
...
struct sEndereco
{
    char rua[40];
    int numero;
};

int main(void) {
    struct sEndereco listaend[5];
    listaend[0].numero = 100;
    strcpy(listaend[3].rua, "Av. Brasil");
    ...
}
```





Exemplo

```
#include <stdio.h>

struct sHora {
    int hor;
    int min;
    int seg;
};

int main(void) {

    struct sHora H[5];
    H[0].hor = 10;
    H[0].min = 15;
    H[0].seg = 30;
    printf("%d:%d:%d", H[0].hor, H[0].min, H[0].seg);
    return 0;
}
```



