Estrutura de Dados

Prof. Maurício Neto

UNIFAVIP
wyden

O que vamos ver?

01

Return

02

Parâmetros por Valor e por Referência

01

Return

Return

O **return 0** no final da função main() em C indica que o programa foi executado com sucesso. A função main() é especial porque é o ponto de entrada do programa, e seu valor de retorno é usado pelo sistema operacional para saber se o programa terminou corretamente ou se houve algum erro

Return - Por que usamos return 0 na main ()

Indicar sucesso: O valor 0 tradicionalmente significa que o programa terminou sem erros. Quando o main() retorna 0, ele está comunicando ao sistema operacional que tudo correu como esperado.

```
int main()
{
    printf("Olá Mundo");

    return 0;
}
```

UNIFAVIP
wyden

Return - Por que usamos return 0 na main ()

Indicar falha: Caso o programa encontre algum erro ou problema, podemos usar um valor diferente de 0 (como 1 ou outro número) para indicar ao sistema que houve uma falha durante a execução.

```
int main()
{
    printf("Olá Mundo");

    return 1;
}
```

UNIFAVIP
wyden

Return - Por que usamos return 0 na main ()

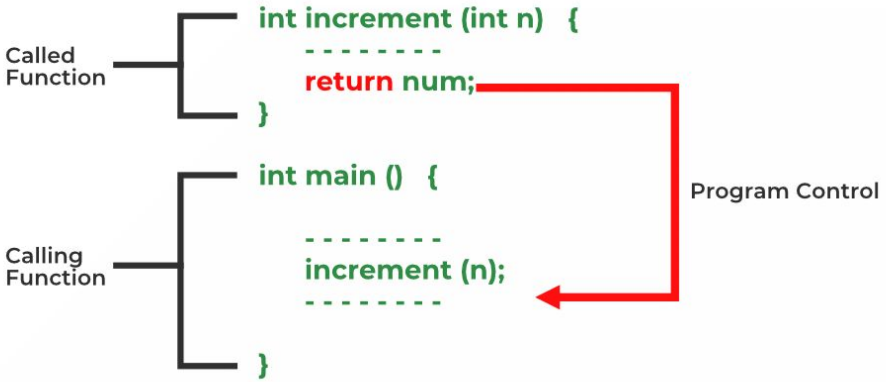
return 0 indica que o programa terminou corretamente.

Um valor **diferente de 0** indica que houve um **erro**.

É uma convenção importante para o sistema operacional e para que outros programas saibam se o seu programa foi executado corretamente.

Return

No C, return é usado para **encerrar** a **execução** de uma função e, **opcionalmente**, retornar um valor para o ponto de onde a função foi chamada.



UNIFAVIP wyden

Return

Funções com retorno de valor (não-void): Em funções que retornam um valor (como int, float, etc.), o **return** não apenas finaliza a função, mas também envia um valor de volta para o local onde a função foi chamada.

```c
int soma(int a, int b) {
    return a + b;
}

int main() {
    int resultado = soma(5, 3);
    printf("%d", resultado);
    return 0;
}
```

UNIFAVIP wyden

Return

Funções sem retorno de valor (void): Em funções do tipo void (que não retornam valores), o return pode ser usado apenas para terminar a execução da função antecipadamente, sem retornar nada.

```c
void saudacao(int hora) {
    if (hora < 12) {
        printf("Bom dia!\n");
        return;
    }
    printf("Boa tarde!\n");
}

int main() {
    saudacao(10);
    return 0;
}
```

Return - Regras Importantes

O return pode aparecer em qualquer lugar da função e pode haver múltiplos return dentro da mesma função.

O uso de return é obrigatório em funções que retornam um valor.

Em função do tipo void, o return é opcional e, se não for usado, a função retorna automaticamente quando chega ao final.

```c
#include <stdio.h>

int verificar_numero(int num) {
    if (num > 0) {
        return 1;
    } else if (num < 0) {
        return -1;
    }
    return 0;
}

int main() {
    int resultado = verificar_numero(10);
    printf("Resultado: %d\n", resultado);
    return 0;
}
```

Return - Porque usar?

Encapsulamento e Reutilização de Código

O return permite que você crie funções reutilizáveis que encapsulam um comportamento específico, retornando o resultado diretamente para o ponto de chamada. Isso evita a repetição de código e torna o programa mais modular.

```c
int soma(int a, int b) {
    return a + b;
}

int main() {
    int resultado = soma(5, 3);
    printf("%d", resultado);
    return 0;
}
```

UNIFAVIP wyden

Return - Porque usar?

Sem return **(repetição de código)**: Se você não usar return, teria que replicar a lógica em diversos lugares no código, o que resulta em uma manutenção mais difícil:

```c
#include <stdio.h>

int main() {
    int a = 5, b = 3;
    int resultado = a + b;
    printf("Resultado: %d\n", resultado);


    int c = 7, d = 2;
    int resultado2 = c + d;
    printf("Resultado: %d\n", resultado2);
    return 0;
}
```

UNIFAVIP wyden

Return - Porque usar?

Controle do Fluxo de Execução

O return permite controlar o momento exato em que uma função deve ser interrompida. Isso é útil em várias situações onde o código pode precisar sair de uma função antes de processar todas as instruções (por exemplo, quando uma condição é satisfeita).

```c
#include <stdio.h>

int encontrar_valor(int valor) {
    if (valor == 10) {
        return 1;
    }
    return 0;
}

int main() {
    int resultado = encontrar_valor(10);
    printf("Encontrou: %d\n", resultado);
    return 0;
}
```

UNIFAVIP wyden

Return - Porque usar?

Retornar Valores Calculados

Em muitas situações, as funções precisam retornar valores calculados para que outras partes do programa possam utilizá-los. O return faz isso de maneira clara e eficiente.

```c
#include <stdio.h>

int calcular_area(int largura, int altura) {
    return largura * altura;
}

int main() {
    int area = calcular_area(5, 10);
    printf("Área: %d\n", area);
    return 0;
}
```

UNIFAVIP wyden

Return - Porque usar?

Manutenção e Legibilidade

Funções que utilizam return para encapsular comportamentos e cálculos tornam o código mais legível e fácil de manter. Em vez de espalhar a lógica por todo o programa, você concentra o processamento em funções específicas.

```c
#include <stdio.h>

int verificar_numero(int num) {
    if (num > 0) {
        return 1;
    }
    return -1;
}
```

UNIFAVIP wyden

Return - Porque usar?

Evitar Efeitos Colaterais

Usar return permite que a função retorne um valor sem modificar variáveis globais ou outros estados fora da função. Isso torna o código mais previsível e seguro, pois limita os efeitos colaterais.

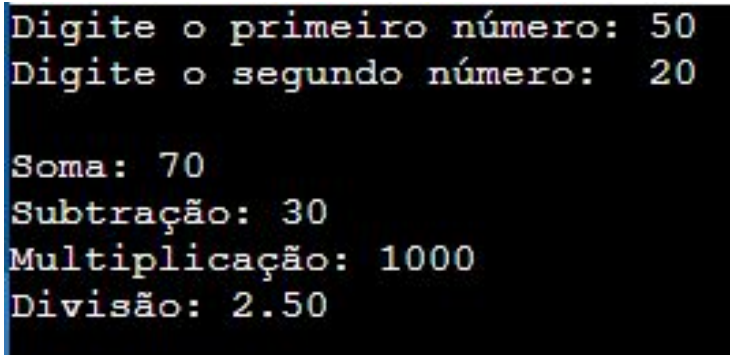
Return

Conclusão

Embora existam maneiras alternativas de controlar o fluxo de execução e compartilhar resultados entre funções, o return é uma ferramenta poderosa que promove a modularidade, reutilização de código, e facilita a manutenção e legibilidade. Ele oferece uma maneira clara de retornar resultados de uma função e interromper sua execução quando necessário, tornando o código mais eficiente e organizado.

Atividade

Faça um programa em C, que o usuário digitar dois números inteiros. E o programa deve utilizar funções para exibir na tela: Somar, subtrair, multiplicar e dividir. A main vai “chamar” todos esses números



UNIFAVIP wyden

02

Parâmetros por Valor e por Referência

Passagem de parâmetros

Definição: Quando uma função é chamada com parâmetros por valor, o valor das variáveis é copiado para os parâmetros da função. **Este processo é chamado de passagem por valor.**

Comportamento: A função trabalha com uma cópia dos valores originais, portanto, qualquer modificação feita na função não afeta a variável original.

```c
#include <stdio.h>

void soma(int x) {
    x = x + 10;  // Modifica a cópia do valor de 'x', não o original
}

int main() {
    int a = 5;
    soma(a);
    printf("%d", a);  // Exibe 5, pois a não foi alterado
}
```

Passagem de parâmetros por Valor

A função soma recebe um parâmetro inteiro x.
Dentro da função, x é incrementado em 10, mas isso não altera o valor da variável original que foi passada como argumento. Isso acontece porque, em C, os parâmetros são passados por valor, ou seja, uma cópia do valor é feita

```
void soma(int x) {
    x = x + 10;  //
}
```

Passagem de parâmetros por Valor

Aqui, a variável a é inicializada com o valor 5.
A função **soma** é chamada com a como **argumento**. No entanto, como mencionado anteriormente, a função não altera o valor de a.
Por fim, printf exibe o valor de a, que continua sendo 5.

```
int main() {
    int a = 5;
    soma(a);
    printf("%d", a);
}
```

Passagem de parâmetros por Valor

Vantagens:

Protege os valores originais de serem modificados.

Simples de entender e usar.

Desvantagens:

Ineficiente para passar grandes estruturas, pois faz cópias dos valores.

Passagem de parâmetros Referência

Definição: Quando uma função é chamada com parâmetros por referência, a função recebe o endereço de memória das variáveis em vez dos valores delas.

Comportamento: A função pode modificar os valores das variáveis originais, pois trabalha diretamente com o endereço de memória das variáveis.

Uso de Ponteiros: Para passar por referência em C, são utilizados ponteiros.

Passagem de parâmetros Referência

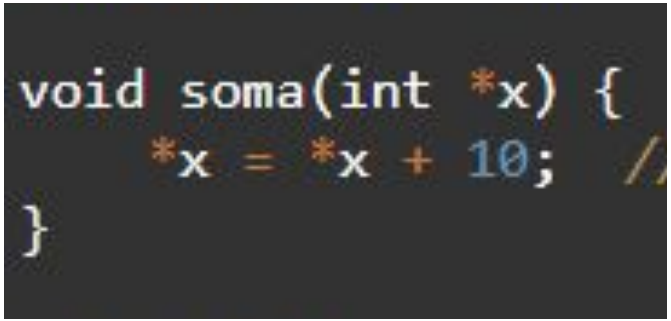
Outro exemplo semelhante, mas desta vez utilizando ponteiros para modificar o valor original da variável.

```c
#include <stdio.h>

void soma(int *x) {
    *x = *x + 10;  // Modifica o valor original usando ponteiro
}

int main() {
    int a = 5;
    soma(&a);  // Passa o endereço de 'a'
    printf("%d", a);  // Exibe 15, pois a foi alterado
}
```

Passagem de parâmetros Referência

Agora, a função soma recebe um ponteiro para um inteiro **(int *x)**.

O operador * é usado para acessar o valor que o ponteiro aponta, permitindo que a função modifique o valor original.



Passagem de parâmetros Referência

Aqui, **&a** é usado para passar o endereço da variável a, permitindo que a função soma altere seu valor diretamente.

```
int main() {
    int a = 5;
    soma(&a);  // Pa
    printf("%d", a);
}
```

UNIFAVIP wyden

Passagem de parâmetros Referência

Vantagens:
Evita cópia de dados, sendo mais eficiente para grandes estruturas.
Permite que a função altere os valores originais.

Desvantagens:
Pode ser mais difícil de entender.
Risco de modificar variáveis não intencionalmente.

Diferenças Entre Passagem por Valor e por Referência

Passagem por Valor	Passagem por Referência
A função recebe uma cópia do valor	A função recebe o endereço da variável
As alterações feitas na função não afetam a variável original	As alterações feitas na função afetam a variável original
Mais simples de entender	Requer o uso de ponteiros.
Ineficiente para grandes estruturas	Mais eficiente, pois evita a cópia de grandes valores

Passagem de parâmetros Referência

Quando Usar Cada Uma?

Passagem por Valor:

Quando os valores originais não devem ser modificados.
Para variáveis simples como int, char, etc.

Passagem por Referência:

Quando é necessário modificar os valores originais.
Para passar grandes estruturas como arrays ou structs.


Ponteiro?
Curso C
func( )
Aula 187
Programe seu futuro

Atividade

Implemente um programa em C que realiza operações com funções. O programa deverá solicitar ao usuário a entrada de dois números inteiros e realizar as seguintes operações utilizando funções com passagem de parâmetros por valor e por referência.

Input:

```
Digite o primeiro número: 10
Digite o segundo número: 30
```

Output:

```
Após troca por valor (nenhuma alteração):
a = 10, b = 30

Após troca por referência (valores trocados):
a = 30, b = 10
```