Modularização

Neste conteúdo, serão estudados os principais aspectos da modularização na programação em C, incluindo a divisão do código-fonte em módulos, a implementação de funções e procedimentos, a passagem de parâmetros por valor e por referência, e o escopo das variáveis locais e globais. Esses tópicos permitem o desenvolvimento de software de forma organizada, eficiente e escalável.

Profa. Daisy Albuquerque

Preparação

Antes de iniciar o conteúdo deste tema, será necessário ter um compilador C instalado em sua máquina, como o GCC. Caso queira, instale o Dev-C++, que é um ambiente de desenvolvimento integrado livre que utiliza os compiladores do projeto GNU para compilar programas para o sistema operacional Microsoft Windows. Também será necessário possuir o arquivo com os códigos que serão utilizados neste tema. Clique aqui para fazer o download.

Objetivos

- Descrever conceitos gerais de procedimentos e funções.
- Identificar os tipos de funções predefinidas na linguagem C.
- Distinguir passagem de parâmetros por valor e por referência.
- Localizar o escopo das variáveis locais e globais.

Introdução

Neste vídeo, demonstraremos como modularizar um programa na linguagem C usando funções e procedimentos, identificando os tipos de funções já predefinidas na biblioteca padrão do C. Relacionaremos, ainda, os tipos de passagem de parâmetros – por valor e por referência – identificando o escopo das variáveis como local e global.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Padronizando a forma de programar

A modularização é uma técnica fundamental na programação em C, que envolve a divisão do código-fonte em módulos autônomos e interconectados, facilitando a construção, a manutenção e a escalabilidade dos programas. Inspirada pela Revolução Industrial e a evolução da engenharia de software, a modularização permite a reutilização de código, a clareza na leitura e a eficiência no desenvolvimento. Vamos explorar a importância da modularização, suas vantagens, e como ela se integra com conceitos de programação estruturada, como funções e procedimentos, proporcionando aos alunos uma base sólida para a criação de software de alta qualidade.

Neste vídeo, vamos ver que a modularização em C divide o código-fonte em módulos autônomos, facilitando construção, manutenção e escalabilidade, permitindo a reutilização de código, clareza na leitura e eficiência no desenvolvimento.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

História

Modularização é o ato ou efeito de modularizar ou tornar modular, isto é, o desenvolvimento, a construção ou a fabricação de algo em unidades ou peças autônomas que são passíveis de serem combinadas com outras para formarem um todo. No campo computacional, a modularização é empregada para dividir o programa em partes funcionais, partes essas que conversam umas com as outras.

Para entender melhor a definição de modularização, é necessário traçar um paralelo com a Revolução Industrial, que aconteceu na Inglaterra, no século XIX.

Sapateiro consertando um sapato.

Antes da Revolução Industrial, todo produto era manufaturado, ou seja, construído inteiro por um artesão e seu aprendiz.

O sapateiro construía todo o sapato, desde a sola até o acabamento em couro, por exemplo.

Para construir um conjunto de cadeiras, o marceneiro recebia a madeira e “esculpia” uma a uma.

Com o advento da Revolução Industrial, o trabalho passou a ser segregado. Uma pessoa construía uma parte da cadeira, – o assento, por exemplo – outra o encosto e outra montava. Tempos depois, a Ford chamou esse processo de linha de montagem, termo utilizado até hoje.

Dessa forma, um produto que era inteiriço passou a ser divido em módulos, que se interligavam e construíam o produto.

Na computação não foi muito diferente. O primeiro computador digital registrado, o ENIAC, não tinha o conceito de software, era tudo hardware. Porém, com a evolução da computação, em meados dos anos 1950, o cientista **Von Neumann** criou a arquitetura de computadores, cuja proposta inicial dividia o computador em três partes:

- Entrada

- Processamento
- Saída

Começava, assim, a **Revolução da Computação**. As máquinas de Von Neumann, como são conhecidas as que adotam esse modelo, dividem o hardware do computador em três partes:

- Unidade central de processamento
- Dispositivos de entrada/saída
- Memória

John Von Neumann introduziu o projeto lógico de computadores com programa armazenado na memória, modularizando o computador em partes como hardware e software.

Após essa fase, começaram a surgir os programas e o interesse comercial por computadores. Inicialmente, os programas eram escritos todos como um produto inteiriço, à semelhança de como era antes da Revolução Industrial. Sendo assim, algumas partes do código eram utilizadas mais de uma vez, o que deixava os programas com muita redundância no código, manutenção complicada e difícil, além da prática de utilizar os desvios com os "**Go To**".

No final da década de 1960, então, ocorreu a crise do software, devido às dificuldades enfrentadas no seu desenvolvimento frente ao rápido crescimento da demanda, à complexidade dos problemas a serem resolvidos e à ausência de técnicas bem estabelecidas para o desenvolvimento de sistemas.

Foi nessa época, mais precisamente em 1968, que ocorreu a Conferência da OTAN sobre Engenharia de Software (NATO — Software Engineering Conference) em Garmisch, Alemanha. O principal objetivo dessa reunião era estabelecer práticas mais maduras para o processo de desenvolvimento e, por essa razão, o encontro é considerado como o nascimento da disciplina de engenharia de software.

> A criação da engenharia de software surgiu em uma tentativa de contornar a crise e dar um tratamento de engenharia (mais sistemático e controlado) ao desenvolvimento de sistemas de software complexos.

Ao lado da engenharia de software surgiu a programação estruturada, que possibilitou dividir o código em pedaços especializados e acoplá-los mais tarde, produzindo, assim, o software que atendia às necessidades comerciais com mais facilidade.

Vale destacar que o processo é semelhante à Revolução Industrial do século XIX, só que no final das décadas de 1960 e 1970.

Módulos

Como já visto anteriormente, a modularização consiste em decompor um programa em uma série de subprogramas individuais. Trata-se de um método utilizado para facilitar a construção de grandes programas, através de sua divisão em pequenas etapas.

> A modularização segue o lema "dividir para conquistar", ou seja, divide o problema em subproblemas menores, sucessivamente, conforme a necessidade.

A primeira etapa, por onde começa a execução do programa, é chamada de programa principal, e as demais são os subprogramas propriamente ditos, que são executados sempre que ocorre uma chamada, o que é feito através da especificação de seus nomes:

Programa principal

O programa principal é o ponto de entrada do software, onde a execução do código começa. Ele inicia as instruções e chama subprogramas ou funções auxiliares para cumprir a lógica e os objetivos do software.

Subprograma

É um programa que, geralmente, resolve um pequeno problema, e que está subordinado a um outro que solicitará seu acionamento. É possível que um subprograma chame outro subprograma.

Mas por que devemos modularizar?

A divisão do programa em módulos possui várias vantagens, entre elas:

- Evitar que os programas fiquem grandes demais e difíceis de serem lidos e compreendidos.
- Facilitar a leitura do código-fonte do programa.
- Separar o programa em partes (blocos) que possam ser compreendidas de forma isolada (criação de módulos).
- Evitar que um trecho seja repetido várias vezes dentro de um mesmo programa.
- Permitir a alteração de um trecho do programa de forma mais rápida.
- Utilizar um código em diferentes partes do programa, sem que ele precise ser escrito em cada local em que se deseje utilizá-lo.
- Permitir o reaproveitamento de código já construído em outros programas (bibliotecas).

Ao dividir o programa em peças menores, será possível conectá-las para formar uma solução sempre que precisarmos resolver um problema mais complexo. Dessa forma, é necessário que essas peças sejam reutilizáveis, assim como blocos de montar.

Essas peças reutilizáveis podem ser chamadas de procedimentos e funções, que são formados por um bloco de código que executa uma determinada ação. De modo geral, cada função e cada procedimento deve realizar apenas uma ação, pois assim se tornam mais reutilizáveis e úteis para os programas.

Código estruturado

Inicialmente, os programas eram blocos lógicos, com início e fim, e compostos de estruturas de controle, como a de condição, seleção e repetição. Esses blocos consistiam em um conjunto de declarações e comandos delimitados pelas palavras início e fim, visando a aumentar a funcionalidade do programa.

Ao passo, porém, que os programas iam se tornando mais complexos, a divisão em partes menores os tornava mais claros e de fácil entendimento. Com o advento da programação estruturada, os programadores começaram a criar estruturas simples, usando as sub-rotinas.

Uma sub-rotina é uma ferramenta de programação que serve basicamente a dois objetivos:

1° Objetivo

Evitar que uma sequência de comandos repetida em vários locais de um algoritmo tenha que ser escrita várias vezes.

2° Objetivo

Dividir a estrutura de um algoritmo em partes fechadas e logicamente coerentes.

As sub-rotinas devem codificar a solução para um problema pequeno e específico. Elas podem ser funções ou procedimentos (quando não retornam valores):

- Os procedimentos são estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado.

- As funções são um tipo especial de procedimento, no qual, depois de executada a chamada, o valor calculado é retornado no nome da função, que passa a ser uma variável da expressão. Ou seja, elas sempre retornam algum valor para quem chamou, diferentemente dos procedimentos, que não retornam valor algum.

Portanto, em um arquivo de programa, funções/procedimentos conversam com o programa principal, e vice-versa.

Atividade 1

Considere um programa em linguagem C que precisa calcular a média de notas de alunos em diferentes disciplinas. Você decide aplicar a técnica de modularização. Quais são as vantagens de modularizar esse programa e como isso pode ser feito de maneira eficaz?

A

A modularização permite escrever todo o código em um único bloco, facilitando a leitura e a manutenção do programa.

B

Ao modularizar, você cria várias funções específicas, como uma para entrada de dados, outra para o cálculo da média, e outra para exibir resultados, tornando o código mais organizado e reutilizável.

C

Modularizar o programa aumenta a complexidade do código, tornando-o mais difícil de entender e manter.

D

A modularização impede a reutilização de código em outros programas, pois as funções são exclusivas ao programa original.

E

Modularizar um programa elimina a necessidade de usar variáveis locais, pois todo o código compartilha as mesmas variáveis globais.

A alternativa B está correta.

A modularização divide o programa em funções específicas, cada uma responsável por uma tarefa distinta, como entrada de dados, cálculo e exibição de resultados. Isso melhora a organização e a legibilidade do código, facilita a manutenção e permite a reutilização de funções em outros programas. As opções A, C, D e E estão incorretas, pois a modularização não envolve escrever todo o código em um único bloco, não aumenta a complexidade do código, permite a reutilização de funções, e promove o uso de variáveis locais dentro de cada função para melhor gerenciamento de escopo.

Funções e procedimentos

Funções são blocos de código que retornam um valor após sua execução, enquanto procedimentos, representados por funções do tipo void, executam ações sem retornar valores. Compreender esses conceitos é essencial para escrever programas modulares, eficientes e de fácil manutenção, permitindo a reutilização de código e melhor organização do projeto. A separação clara entre declaração e implementação de funções também será abordada, facilitando o desenvolvimento colaborativo e a leitura do código.

Este vídeo vai mostrar a importância de compreender os conceitos de funções e procedimentos para programas modulares, eficientes e de fácil manutenção, permitindo a reutilização de código e melhor organização. Vai demonstrar, ainda, que a separação entre declaração e implementação facilita o desenvolvimento colaborativo e a leitura do código.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Funções

As funções são procedimentos que retornam um único valor ao final de sua execução.

No exemplo a seguir, apresentamos alguns casos, como as funções **sqrt()**, **scanf()** e **printf()**, que calculam a raiz quadrada, leem um número da entrada padrão e imprimem um valor na saída padrão. Veja:

```c
x = sqrt(4);
if (scanf ("%d", &x) == EOF)
printf("Fim de arquivo.\n");
```

Declarando uma função

Uma função em pseudolinguagem possui o seguinte formato:

```c
1   funcao < nome-de-função >
2       [( < sequência-declarações-de-parâmetros >)]: < tipo-de-dado >
3       var
4       // Seção de Declarações Internas
5       inicio
6       // Seção de Comandos
7
fimfuncao
```

Toda função deve ter um tipo que determina qual será o tipo de seu valor de retorno. Já os parâmetros de uma função determinam qual será o seu comportamento (como se fosse uma função matemática, na qual o parâmetro determina o valor da função). É possível, porém, que uma função não tenha parâmetros, basta não os informar. Na declaração a seguir, estão listados os parâmetros que serão utilizados na função:

```c
tipo nome_da_funcao ( tipo < parametro1 >,
tipo < parametro2 >, . . . , tipo < parametron >) {
comandos;
return( valor de retorno );
}
```

É aconselhado declarar as funções antes do programa principal, mas o que ocorreria se declarássemos depois?

Embora aparentemente funcione, isso não pode ser feito na prática, pois alguns compiladores simplesmente não aceitam essa sintaxe, obrigando que a declaração ocorra antes da **função main** (ou melhor, antes de qualquer outra função que utilize uma determinada função).

Função main

A função main é o chamado programa principal. Ela apenas é uma função especial, que é invocada automaticamente pelo programa quando esse inicia sua execução.

A saída para a questão citada anteriormente seria fazer uma declaração da função antes da função **main**(), mas colocar a sua implementação depois dela. Declarar uma função sem a sua implementação é muito semelhante a declará-la com a implementação. Substituímos as chaves e seu conteúdo por ponto e vírgula, ou seja:

```c
tipo nome_da_funcao ( tipo < parametro1 >,
tipo < parametro2 >, . . . , tipo < parametron >);
```

No exemplo a seguir, em linguagem C, somente declaramos a função **imprime()**, sem implementá-la:

```c
void imprime (int numero);
```

Com isso, separamos a declaração da implementação, permitindo que ela possa vir em qualquer lugar do código (antes ou depois de qualquer outra função). Além disso, um programa que declara todas as funções antes de usá-las tende a ser mais claro, pois o programador já sabe qual o conjunto de funções que ele pode usar, sem se preocupar com a forma como elas foram implementadas (ou sequer como elas foram desenvolvidas, caso o código esteja sendo desenvolvido por uma equipe).

Exemplo de função

No exemplo a seguir, em linguagem C, identificamos a função soma() que realiza a soma de dois valores, que são passados por parâmetro:

```c
1   int soma (int a, int b) {
2       return (a + b);
3   }
```

A expressão contida dentro do comando **return** é chamada de valor de retorno, e corresponde à resposta de uma determinada função. Esse comando é sempre o último a ser executado por uma função, e nada após ele será executado.

Quando utilizado, o comando return informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente, ou qualquer outro valor caso contrário.

Invocando uma função

Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável.

```
c

variavel = funcao (parametros);
```

Na verdade, podemos invocar uma função em qualquer lugar onde faríamos a leitura de uma variável, mas nunca a escrita. Veja o exemplo a seguir.

```
c

printf ("Soma de x e y: %d\n", soma(x, y));
```

Nesse exemplo, ao executar a função **printf** será invocada a função soma, que irá calcular a soma das variáveis x e Y. Ao retornar a função, o resultado será apresentado na tela.

As variáveis passadas como parâmetros indicam os valores com os quais a função irá trabalhar. Esses valores são copiados para os parâmetros da função, que pode manipulá-los.

Os parâmetros passados pela função não necessariamente possuem os mesmos nomes que os que a função espera. Esses parâmetros serão mantidos intocados durante a execução da função.

O tipo **void** é um tipo especial, utilizado principalmente em funções. Ele representa o “nada”, ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.

Procedimentos

São estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado, ou seja, procedimentos em linguagem C são funções do tipo void.

Declarando um procedimento

Um procedimento possui o seguinte formato:

```
c

1   procedimento < nome-de-procedimento >
2       [(< sequência-de-declarações-de-parâmetros >)]
3   var
4       // Seção de Declarações Internas
5   inicio
6       // Seção de Comandos
7
fimprocedimento
```

Por exemplo, o procedimento a seguir imprime o número que for passado para ele como parâmetro:

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

Ao se ignorar o valor de retorno de uma função e, para esta chamada, ela será equivalente a um procedimento.

Invocando um procedimento

Para invocarmos um procedimento, devemos utilizá-lo como utilizaríamos qualquer outro comando, ou seja:

```c
procedimento (parametros);
```

Atividade 2

Você está desenvolvendo um programa em C que deve calcular a soma de dois números e imprimir o resultado. Para isso, decide utilizar uma função para a soma e um procedimento para a impressão do resultado. Como você deve declarar e invocar essas funções e procedimentos de maneira adequada?

A

Declarar a função e o procedimento antes da função main(), mas implementar ambos depois, garantindo que o compilador reconheça suas assinaturas.

B

Declarar a função e o procedimento após a função main(), pois o compilador irá automaticamente reconhecer suas assinaturas.

C

Declarar a função antes da função main() e o procedimento depois, pois o procedimento não precisa ser conhecido pelo compilador antecipadamente.

D

Implementar a função e o procedimento diretamente dentro da função main() para garantir que sejam executados corretamente.

E

Declarar e implementar tanto a função quanto o procedimento antes da função main(), pois o compilador requer a implementação completa antes de qualquer uso.

A alternativa A está correta.

Para garantir que o compilador reconheça as assinaturas de funções e procedimentos, é recomendado declará-los antes da função main(). Isso permite que suas implementações possam ser localizadas em qualquer parte do código, facilitando a manutenção e a organização. A prática de declarar antes e implementar depois ajuda a manter um código mais limpo e legível, especialmente em projetos colaborativos. As opções B, C, D e E são incorretas porque não seguem as boas práticas de modularização e podem causar problemas de reconhecimento de funções pelo compilador.

Principais bibliotecas

As funções predefinidas na linguagem C são essenciais para facilitar diversas operações comuns, como manipulação de strings, entrada e saída de dados, e cálculos matemáticos. Essas funções estão organizadas em bibliotecas específicas, acessíveis por meio de arquivos header com extensão “.h”. Para utilizar essas funções, é necessário incluir as bibliotecas apropriadas no início do programa usando a diretiva #include. Conhecer as bibliotecas padrão, como stdio.h, stdlib.h, ctype.h, time.h, dos.h, string.h e math.h, permite ao programador aproveitar ao máximo as funcionalidades oferecidas pela linguagem C.

O vídeo vai abordar as funções predefinidas em C, que facilitam operações como manipulação de strings, entrada e saída de dados, e cálculos matemáticos, mostrar que são organizadas em bibliotecas padronizadas e apresentar as bibliotecas padrão como stdio.h e string.h

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conceitos

As funções predefinidas na linguagem C estão estipuladas nas bibliotecas da linguagem, nos arquivos header com extensão “.h”. Para usá-las nos programas, é necessário incluir no início do mesmo, através da diretiva #include, a biblioteca que possui a função que será usada.

Dessa maneira, existem diversos header padrão na linguagem C, entre eles:

stdio.h
Funções de E/S padrão nos dispositivos stdin, stdout e files.

stdlib.h
Funções de E/S padrão nos dispositivos stdin, stdout e files. Funções para conversão de números em cadeias de caracteres.

ctype.h
Funções para tratamento de caractere.

time.h
Funções para manipulação de horários e datas.

dos.h
Funções de acesso as INT's da BIOS e DOS do MS-DOS.

string.h
Funções de manipulação de strings (TC).

math.h
Funções matemáticas em geral.

Atividade 1

Você está desenvolvendo um programa em C que precisa calcular a raiz quadrada de um número, converter um número para uma string, e manipular datas e horários. Quais bibliotecas você deve incluir no seu programa para acessar essas funcionalidades predefinidas?

A

stdio.h, stdlib.h, time.h

B

math.h, string.h, time.h

C

stdlib.h, dos.h, ctype.h

D

stdio.h, math.h, dos.h

E

string.h, stdio.h, ctype.h

A alternativa B está correta.

Para calcular a raiz quadrada de um número, é necessário incluir a biblioteca math.h, que contém funções matemáticas. Para converter um número para uma string, a biblioteca string.h deve ser incluída, pois ela oferece funções para manipulação de strings. Para manipular datas e horários, a biblioteca time.h é necessária. As outras opções não cobrem todas as funcionalidades exigidas pelo enunciado.

Funções de E/S padrão

As funções de entrada e saída (E/S) padrão em C são fundamentais para a interação entre o programa e o usuário, permitindo a leitura de dados do teclado e a exibição de informações na tela. Utilizando a biblioteca stdio.h, funções como printf, scanf, putchar, puts, getchar e gets desempenham papéis cruciais na formatação e manipulação de dados. Essas funções são essenciais para qualquer programador, facilitando a

implementação de operações de E/S de maneira eficiente e controlada, tornando o desenvolvimento de aplicações mais intuitivo e organizado.

O vídeo vai abordar as funções de entrada e saída (E/S) em C, usando a biblioteca stdio.h, essenciais para a interação com o usuário. Vai demonstrar ainda que as funções disponibilizadas nessa biblioteca viabilizam a leitura de dados e exibição de informações, tornando o desenvolvimento mais intuitivo e organizado.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As funções de E/S padrão são responsáveis pelas operações de entrada e saída de dados. Quando um programa na linguagem C é executado, o sistema operacional é responsável por abrir três arquivos:

stdin
Arquivo de entrada padrão.

stdout
Arquivo de saída padrão.

stderr
Erro padrão.

Os arquivos stdout e stderr são direcionados para a saída do vídeo, e o arquivo stdin é direcionado para o teclado.

Declaração

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

```
c

#include
```

Invocando funções de E/S padrão

printf

A sua função é ser responsável pela impressão formatada dos dados. Ela imprime os dados no dispositivo de saída padrão, tela do computador. Os mesmos são formatados de acordo com os códigos de formatação, conforme a tabela a seguir:

Caractere	Impresso Como	Tipo de Argumento
d, i	Número decimal	int
o	Número octal sem sinal (sem zero à esquerda)	int

x,X	Número hexadecimal sem sinal (sem um 0x ou 0X à esquerda), usando abcdef ou ABCDEF para 10,..., 15	int
U	Número decimal sem sinal	int
C	Único caractere	int
S	Imprime caracteres da string até um '\ o' ou o número de caracteres dado pela precisão	char
F	[-] m.dddddd, onde o número de d's é dado pela precisão (padrão 6)	double
e, g	[-] m.dddddde+/-xx or [-] m.ddddddE+/-xx, onde o número de d's é dado pela precisão (padrão 6)	double
g, G	Use % e ou % E se o expoente for menor que = 4 ou maior ou igual à precisão; caso contrário, use %f.	double
P	Ponteiro	void *
%	Nenhum argumento impresso, imprima um %	

Tabela: Tabela de conversões para a função printf.
Daisy Albuquerque.

A sintaxe da função é:

```
c

printf("format_string"), arg1, arg2, ...);
```

Onde **format_string** é formado por caracteres ordinários e especificações de formato. Os caracteres ordinários são copiados diretamente na saída padrão. A especificação de formato é composta pelos seguintes elementos:

- %
- [flags]
- [width]
- [.precision]
- [size]
- type

Vamos conhecer mais sobre alguns elementos?

1 [flags]
Composto pelos seguintes caracteres:

- "-" $\rightarrow$ Posicione o valor à esquerda.
- " + " $\rightarrow$ O valor deve ser precedido do sinal (" $+^{*}$ ou " - ").
- "" $\rightarrow$ Implica na impressão com sinal negativo apenas.
- "0" $\rightarrow$ Preenche o valor com zeros.

2
[width]
Composto pelos seguintes valores:

- $\langle n \rangle \rightarrow$ Ao menos <n> caracteres são impressos. Caso o comprimento seja menor que <n>, será completado com brancos.
- O $\langle n \rangle \rightarrow$ Caso o comprimento seja menor que <n>, completa com zeros.

3
[precision]
Composto pelos seguintes valores:

- $.0 \rightarrow$ Impressão de float sem ponto decimal.
- . <n> $\rightarrow$ Limita o número de casas decimais a <n>.

4
[size]
Composto pelos seguintes caracteres:

- l $\rightarrow$ Implica na definição de um modificador LONG.
- $\mathrm{h} \rightarrow$ Implica na definição de um modificador SHORT.

5 [type]
Composto pelos seguintes caracteres:

- $\mathrm{d} \rightarrow$ O argumento é dado na forma inteiro decimal.
- $0 \rightarrow$ O argumento é apresentado em octal (inteiro).
- $x \rightarrow$ O argumento é dado na forma inteiro hexadecimal.
- $u \rightarrow$ O argumento é considerado decimal inteiro sem sinal.
- $\mathrm{c} \rightarrow$ O argumento é um caractere único.
- $s \rightarrow$ O argumento é uma cadeia de caracteres (string).
  e $\rightarrow$ O argumento é dado como float ou double em notação cientifica (com expoente).
- $\mathrm{f} \rightarrow$ O argumento é dado como float ou double em notação decimal comum (sem expoente).
- $g \rightarrow$ Usar "E" ou "F", conforme a necessidade.
- $\mathrm{p} \rightarrow$ O argumento é um pointer (TC).
- $\% \rightarrow$ Reproduz o próprio símbolo $\%$ ".

Onde **arg1, arg2, arg3, ... (argumentos)** são apontadores para a área de armazenamento, ou seja, ponteiros para variáveis (fornecem o endereço da variável, exceto para arrays). O primeiro argumento é obrigatório. Se o programador invocar a função printf sem argumentos – printf() – uma mensagem de erro é emitida: *too few arguments to function 'printf'*.

Por exemplo, no comando a seguir, a nota do aluno, que é um valor em ponto flutuante que será exibido com 3 dígitos e 2 casas decimais:

```
c

printf ("\nNota do aluno:%3.2f", nota);
```

No comando a seguir, será impressa a variável string como uma cadeia de caracteres, e a variável contador será impressa em hexadecimal:

```
c

printf ("Lista de itens: %s\nContador = %x", string, contador);
```

No comando a seguir, as variáveis a, b e c serão impressas como números decimais, alinhadas à esquerda e com 5 dígitos:

```
c

printf ("a: %-5D b: %-5D c: %-5D", a, b, c);
```

Analisemos o programa a seguir (Programa1.c):

```c
//Programa 1
#include < stdio.h >
#include < stdlib.h >

int main(int argc, char *argv[]){
    int i;
    i=123;
    printf("%d\n", i);
    return 0;
}
```

O primeiro argumento da função **printf()** corresponde a um arranjo de caracteres (string), que são tratados da seguinte forma:

- Caracteres ordinários que são copiados para o fluxo de saída.
- Caracteres de especificação de conversões.

A distinção entre esses dois tipos relaciona-se ao caractere '%' (por cento). Os caracteres que seguem o '%' devem obedecer a várias regras e, se estas não forem obedecidas, o comportamento não é definido na linguagem C.

No programa mencionado anteriormente, os caracteres '%' e 'd' correspondem a uma especificação de conversão.

> Umas das regras da função printf() é que, além do primeiro argumento, deve haver tantos argumentos quanto o número de especificações de conversão.

O caractere 'd' após o '%', nesse programa, especifica que deve haver um argumento que será convertido como um valor do tipo int na base 10, será impresso 123 na saída padrão e pulada uma linha por causa do /n.

Os caracteres '\' e 'n' correspondem a caracteres ordinários e especificam o envio de um caractere "funcional" correspondente a uma "nova linha".

Substituindo a chamada da função **printf("%d\n", i);** por **printf("%x\n", i);** ocorre a especificação de conversão do argumento como um valor do tipo int, mas na base 16, e seria impresso o valor 7B, que corresponde a 123 em decimal.

Analisemos outro programa (Programa2.c):

```c
//Programa 2
#include < stdio.h >
#include < stdlib.h >

int main(int argc, char *argv[]){
    double pi;
    pi=3.14;
    printf("%f\n", pi);
    return 0;
}
```

Esse programa é similar ao anterior, mas agora é utilizada uma variável do tipo double e é especificada uma conversão de um tipo double para o formato [-] m.dddddd, através da especificação de formato %f.

Os dois programas anteriores podem ser combinados, e teremos o Programa3.c:

```c
//Programa 3
#include < stdio.h >
#include < stdlib.h >

int main(int argc, char *argv[]){
    int i; double pi;
    i=123; pi=3.14;
    printf("%d %f\n", i, pi);
    return 0;
}
```

As duas especificações de conversão (%d e %f), exigem dois argumentos (i e pi), que serão impressos respectivamente pelos formatos especificados, decimal e ponto flutuante.

scanf

Sua função é ser responsável pela entrada formatada dos dados. Ela lê os dados do dispositivo de entrada padrão (teclado do computador), que são formatados de acordo com os códigos de formatação conforme a tabela a seguir. Pertence ao grupo de funções de entrada formatada, e atua de forma análoga à saída formatada (printf), mas no sentido inverso.

Caractere	Dados de entrada	Tipo de Argumento
d	Número inteiro decimal	int *
i	Inteiro	int *
o	Octal inteiro	int *
u	Inteiro decimal sem sinal	unsigned int*
x	Número inteiro hexadecimal	int *
c	Caracteres	char *
s	Cadeia de caracteres (string)	char *
e, f, g	Número de ponto flutuante com o ponto decimal opcional e expoente opcional	float *
%	% Literal	

Tabela: Tabela de conversões para a função scanf.
Daisy Albuquerque.

A sintaxe da função é:

```c
scanf ("format_string", arg1, arg2, arg3, ...);
```

Onde:

- **format_string**: Opera de forma semelhante ao printf, entretanto, agora serão apresentadas as formatações para a entrada de dados que serão lidos pelo teclado.

- **arg1, arg2, arg3, ... (argumentos)**: São apontadores para a área de armazenamento, ou seja, ponteiros para variáveis (fornecem o endereço da variável, exceto para arrays), sendo obrigatório o primeiro argumento. Se o programador invocar a função scanf sem argumentos (scanf()), uma mensagem de erro é emitida: too few arguments to function 'scanf'.

É importante não se esquecer de colocar o "**&;**" antes do nome da variável. A falta deste caractere é um erro muito comum dos programadores iniciantes.

Por exemplo, a função scanf a seguir irá ler os dados digitados no teclado e armazenará na variável salario como tipo float:

```
c

scanf(“%f”, &salario);
```

Agora, a função **scanf** irá armazenar os dados digitados no teclado como inteiro na variável número:

```
c

scanf("%i", &numero);
```

Analisemos o programa a seguir (Programa4.c):

```
c

//Programa 4
#include < stdio.h >
#include < stdlib.h >

int main(int argc, char *argv[]){
    int i;
    scanf("%i", &i);
    printf("%i\n", i);
    return 0;
}
```

Esse programa usa a função **scanf()** para obter um valor para a variável i a partir dos caracteres que foram digitados no teclado. O dado será armazenado na variável i como um inteiro. A função **printf** irá exibir na tela o conteúdo da variável com no formato inteiro.

A função scanf() lê uma ou mais linhas para obter caracteres a serem usados nas conversões. Um possível problema que pode ocorrer na função scanf() é um certo tratamento desigual para as diferentes conversões.

Comentário

Quando é especificado no primeiro parâmetro uma conversão do tipo %d, a função scanf() irá ignorar diversos caracteres para poder encontrar uma sequência de caracteres entre ‘0’ e ‘9’. Mas, com relação à conversão %c, a função scanf() não ignora nenhum caractere (nem mesmo o correspondente ao momento em que o usuário teclou “enter”).

A função scanf retorna ao ponto de chamada somente após ter processado toda a especificação de conversão dada pelo primeiro argumento. O caractere de mudança de linha é tratado como caractere em branco exceto na conversão %c.

Outras funções para entrada e saída de dados

Vamos conhecer algumas outras funções para entrada e saída de dados:

putchar

Essa função envia um único caractere pela saída padrão (escreve apenas um caractere). Equivale ao comando: printf("%c",carac). **Sintaxe:** putchar();

puts

Essa função envia uma string na saída padrão. A string será adicionada com um caractere '\n'. Equivale a: printf("%s\n", string). **Sintaxe:** puts(string);

getchar

Essa função é responsável por ler um caractere da entrada padrão. Equivale à função putchar, mas em sentido inverso. **Sintaxe:** [variável] = getchar();

gets

Essa função é responsável por ler uma string da entrada padrão. Análoga à função puts, mas em sentido inverso. **Sintaxe:** [variável] = gets();

Para que nenhum lixo do buffer de teclado atrapalhe o uso da função getchar, coloque fflush(stdin), no Windows, ou _fpurrge(stdin), no Linux, antes da leitura do caractere.

Atividade 2

Você está desenvolvendo um programa em C que precisa ler a idade de um usuário e imprimir uma mensagem formatada contendo a idade. Quais funções de E/S padrão você deve utilizar para realizar essas operações e como deve ser a sintaxe correta?

A

Utilizar a função gets para ler a idade e a função putchar para imprimir a mensagem formatada.

B

Utilizar a função getchar para ler a idade e a função printf para imprimir a mensagem formatada.

C

Utilizar a função scanf para ler a idade e a função printf para imprimir a mensagem formatada.

D

Utilizar a função puts para ler a idade e a função printf para imprimir a mensagem formatada.

E

Utilizar a função scanf para ler a idade e a função puts para imprimir a mensagem formatada.

A alternativa C está correta.

A função scanf é ideal para ler dados formatados, como inteiros, diretamente do teclado, enquanto a função printf é utilizada para imprimir mensagens formatadas na tela. A combinação dessas duas funções permite a leitura eficiente da idade do usuário e a exibição de uma mensagem formatada. As outras opções não são adequadas: gets e getchar são usadas para strings e caracteres respectivamente, enquanto puts é usada para imprimir strings completas e não é adequada para formatação detalhada como printf.

Funções de manipulação de arquivo

As funções de manipulação de arquivos em C são essenciais para trabalhar com arquivos de forma eficiente, permitindo operações como leitura, escrita e atualização. Disponíveis na biblioteca stdio.h, essas funções incluem fopen para abrir arquivos, fclose para fechá-los, fgetc e fputc para leitura e escrita de caracteres, e fprintf e fscanf para operações formatadas. Compreender essas funções e seu uso adequado é fundamental para qualquer programador que precise manipular dados persistentes, garantindo a integridade e a acessibilidade das informações armazenadas em arquivos.

O vídeo vai mostrar que funções de manipulação de arquivos em C, disponíveis na biblioteca stdio.h, permitem leitura, escrita e atualização eficiente de dados gravados em memória secundária, viabilizando a persistência da informação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As funções de manipulação de arquivos são responsáveis pelas operações com estes, e se encontram na biblioteca stdio.h.

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

```
c

#include
```

Invocando funções de manipulação de arquivos

fopen

A função fopen abre um arquivo, assim como os dispositivos console, impressora e saída serial, que podem ser abertos como arquivos.

As funções de manipulação de arquivos, assim como os ponteiros, devem ser declaradas juntamente com a declaração de variáveis. Sendo assim, vamos entender melhor a definição de ponteiros para arquivos conforme a sintaxe a seguir:

```
c
FILE *fp;
```

Onde:

- – FILE: Tipo de dado definido no stdio.h.
- – *fp: Ponteiro para o arquivo que será manipulado no programa.

Sintaxe:

```
c
FILE *fopen(const char *nome_arquivo, const char *modo);
```

Onde **nome_arquivo** é o nome do arquivo aberto, e o modo pode conter um dos seguintes valores:

- "r" – Leitura (open).
- "w" – Escrita (rewrite).
- "a" – Adição (append).
- "r+" – Para atualizar um arquivo (read e write).
- "w+" – Para criar um arquivo para atualização.
- "a+" – Adição, em arquivo para leitura e escrita.

A função fopen() retorna o ponteiro para o arquivo ou NULL, em caso de erro na abertura do mesmo. Todas as outras operações sobre arquivos serão referenciadas através deste "apontador de arquivo".

Após declarar a variável do tipo arquivo, pode ser aberto o arquivo utilizando a função fopen(), conforme o exemplo a seguir:

```
c
fp = fopen(“arquivo.txt”,”r”);
```

Outras funções

Vamos conhecer algumas outras funções a seguir:

fgetc
Responsável pela leitura de um caractere de um arquivo indicado pelo ponteiro file_pointer. No caso de chegar ao fim do arquivo, será retornado EOF. Sintaxe: [caractere] = fgetc (file_pointer).

fputc
Responsável pela inserção de um caractere no arquivo. A função escreve o caractere na próxima posição do arquivo apontado pelo file_pointer. É a operação inversa de getc. Sintaxe: fputc (caractere, file_pointer).

ungetc
Responsável em devolver o último caractere lido do arquivo para o file_pointer, aceitando apenas um caractere devolvido por vez (coloca em um buffer temporário). Sintaxe: ungetc (caractere, file_pointer).

fprintf
Responsável pela saída formatada em arquivos, semelhante à função printf, mas atua sobre arquivos, definidos pelo primeiro parâmetro, que é um ponteiro para o arquivo. Sintaxe: fprintf (file_pointer,"controle", arg1, arg2,...).

fscanf
Responsável pela entrada formatada a partir de arquivos. Atua como a função inversa de fprintf e análoga à função scanf, só que atuando sobre arquivos. Sintaxe: fscanf (file_pointer,"controle",arg1,arg2,...).

fclose
Responsável pelo fechamento do arquivo em uso. É utilizado para encerrar o acesso a um arquivo previamente aberto, liberando os recursos associados a ele. Sintaxe: fclose (file_pointer).

Outras funções
fflush(): Descarrega o buffer; fgets(): Obtém uma string do arquivo; fputs(): Insere uma string no arquivo; fread(): Lê um bloco de dados do arquivo; fwrite(): Escreve um bloco de dados no arquivo; fseek(): Reposiciona o ponteiro para o arquivo; rewind(): Reposiciona o ponteiro para o início do arquivo; ftell(): Retorna a posição do ponteiro.

Atividade 3

Você está desenvolvendo um programa em C que precisa abrir um arquivo para leitura, ler seu conteúdo caractere por caractere, e depois fechar o arquivo. Quais funções você deve utilizar para realizar essas operações e qual seria a sintaxe correta para abrir, ler e fechar o arquivo?

A

Utilizar fopen para abrir o arquivo em modo leitura, fgetc para ler os caracteres, e fclose para fechar o arquivo.

B

Utilizar fopen para abrir o arquivo em modo escrita, fputc para ler os caracteres, e fclose para fechar o arquivo.

C

Utilizar fopen para abrir o arquivo em modo leitura, fscanf para ler os caracteres, e fclose para fechar o arquivo.

D

Utilizar fopen para abrir o arquivo em modo atualização, fgetc para ler os caracteres, e fclose para fechar o arquivo.

E

Utilizar fopen para abrir o arquivo em modo leitura, fgets para ler os caracteres, e fclose para fechar o arquivo.

A alternativa A está correta.

A função fopen é usada para abrir arquivos, e no modo "r" (leitura), permite que o arquivo seja lido. A função fgetc é adequada para ler caracteres individuais de um arquivo, retornando EOF ao final do arquivo. Finalmente, fclose é usada para fechar o arquivo, garantindo que todos os buffers sejam descarregados corretamente. As outras opções são incorretas porque usam modos ou funções inadequadas para a leitura caractere por caractere em arquivos abertos para leitura.

Funções de manipulação de tipos de dados

As funções de manipulação de tipos de dados em C, encontradas na biblioteca ctype.h, são utilizadas para verificar e transformar caracteres. Essas funções permitem determinar se um caractere é maiúsculo, minúsculo, numérico, alfabético, um espaço, ou um caractere ASCII. Além disso, funções como tolower e toupper são usadas para converter caracteres entre maiúsculas e minúsculas. Utilizar essas funções é essencial para validar e processar entradas de usuários, garantindo que os dados sejam tratados de forma correta e consistente.

O vídeo vai mostrar como funções de manipulação de dados em C, disponíveis na biblioteca ctype.h, verificam e transformam caracteres, sendo essenciais para validação e processamento correto de dados.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As funções de manipulação de tipos de dados podem ser usadas para tratamento de caractere, permitindo verificar se um determinado caractere é ASCII, se é numérico, se é uma letra maiúscula, minúscula etc.

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

```
c

#include
```

Invocando funções de manipulação de tipos de dados

Vamos agora invocar as funções de manipulação de tipos de dados.

isupper

Verifica se o caractere é maiúsculo. Retorna diferente de zero se for maiúsculo e zero no caso contrário (minúscula). Possui função análoga, mas de funcionamento inverso a esta: islower (c), que verifica se o caractere é minúsculo.

Sintaxe: valor_log = isupper (caractere).

isalpha

Verifica se é caractere alfabético. Segue o mesmo princípio de isupper. **Sintaxe**: valor_log = isalpha (caractere);

Funções análogas:

- isdigit (c) – Verifica se é um dígito.
- isspace (c) – Verifica se é caractere de espaço.
- isascii (c) – Verifica se é caractere ASCII.
- isprint (c) – Verifica se é caractere de impressão.

tolower

Converte de maiúscula para minúscula. Possui função análoga, mas de funcionamento inverso a esta: toupper (c).

Sintaxe: caractere = tolower (caractere).

Atividade 4

Você está desenvolvendo um programa em C que precisa verificar se um caractere fornecido pelo usuário é uma letra maiúscula e, se for, convertê-la para minúscula. Quais funções você deve utilizar para realizar essas operações e qual seria a sintaxe correta?

A

Utilizar isalpha para verificar se é uma letra e toupper para converter para minúscula.

B

Utilizar isupper para verificar se é maiúscula e tolower para converter para minúscula.

C

Utilizar isascii para verificar se é uma letra e tolower para converter para minúscula.

D

Utilizar isdigit para verificar se é uma letra e toupper para converter para minúscula.

E

Utilizar islower para verificar se é maiúscula e tolower para converter para minúscula.

A alternativa B está correta.

A função isupper é usada para verificar se um caractere é maiúsculo, retornando um valor diferente de zero se for verdade. A função tolower converte um caractere maiúsculo para minúscula. Assim, a combinação dessas duas funções permite verificar se o caractere é maiúsculo e convertê-lo corretamente para minúscula. As outras opções são incorretas porque utilizam funções inadequadas para a verificação e conversão especificadas no enunciado.

Funções de manipulação de string

As funções de manipulação de strings na linguagem C, disponíveis na biblioteca string.h, são essenciais para operações comuns como copiar, comparar, concatenar e medir o comprimento de strings. Funções como strlen, strcpy, strcmp, strcat, sprintf e sscanf facilitam o trabalho com cadeias de caracteres, permitindo manipulações eficientes e seguras. Essas funções são fundamentais para o desenvolvimento de programas que lidam com processamento de texto, entrada e saída de dados, e formatação de informações, garantindo que as strings sejam tratadas de forma adequada e eficaz.

O vídeo vai mostrar que funções de manipulação de strings em C, na biblioteca string.h, permitem copiar, comparar, concatenar e medir comprimento de strings. Mostrará ainda que funções como strlen, strcpy, strcmp, strcat, sprintf e sscanf facilitam manipulações eficientes e seguras, essenciais para processamento de texto, entrada e saída de dados, e formatação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As funções de manipulação de strings podem ser usadas para copiar, comparar, concatenar, dentre outras funcionalidades. Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

```c
#include
```

Invocando funções de manipulação de string

Vamos agora invocar as funções de manipulação de string.

strlen

Retorna o tamanho de uma string. **Sintaxe**: strlen(string); onde "string" é um tipo char *c.

strcpy

Copia uma string em outra. **Sintaxe**: strcpy(s,t); onde "s" e "t" são do tipo char *c.

strcmp

Compara duas strings. **Sintaxe**: strcmp(s,t); onde "s" e "t" são do tipo char *c.

strcat

Concatena duas strings. **Sintaxe**: strcat(s,t); onde "s" e "t" são do tipo char *c.

Atividade 5

Você está desenvolvendo um programa em C que precisa ler duas strings do usuário, concatená-las e armazenar o resultado em uma terceira string. Quais funções você deve utilizar para realizar essas operações e qual seria a sintaxe correta para copiá-las e concatená-las?

A

Utilizar strlen para copiar as strings e strcat para concatená-las.

B

Utilizar strcpy para copiar as strings e strcmp para concatená-las.

C

Utilizar strcpy para copiar as strings e strcat para concatená-las.

D

Utilizar strcat para copiar as strings e sprintf para concatená-las.

E

Utilizar strcpy para copiar as strings e sprintf para concatená-las.

A alternativa C está correta.

A função strcpy é utilizada para copiar o conteúdo de uma string para outra, enquanto strcat é usada para concatenar duas strings. Portanto, a combinação dessas duas funções permite copiar as strings e depois concatená-las corretamente. As outras opções são incorretas porque utilizam funções inadequadas para as operações especificadas no enunciado.

Elementos sintáticos e semânticos das funções

Parâmetros são variáveis que recebem valores passados para sub-rotinas, enquanto argumentos são os valores reais fornecidos. A passagem de parâmetros pode ser feita por valor, no qual uma cópia do valor é passada, ou por referência, na qual o endereço de memória da variável é utilizado. Entender essas técnicas é fundamental para manipular dados e modificar variáveis eficientemente dentro de sub-rotinas.

Neste vídeo, você vai ver os principais elementos sintáticos e semânticos da declaração de funções, destacando-se a passagem de parâmetros que pode ser feita por valor ou referência.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Definições

No módulo anterior, estudamos as funções predefinidas, tais como **printf**, **scanf**, **getchar**, **putchar**, dentre outras que fazem parte da biblioteca padrão C, ou seja, são obtidas ao se adquirir qualquer compilador de C tal como GCC.

Essas funções, de acordo com a sua funcionalidade, possuem parâmetros, e são invocadas pelo programa principal passando argumentos de acordo com a sua sintaxe.

Exemplo

A função scanf é responsável por efetuar a leitura dos dados de uma fonte externa. E, ao analisar sua sintaxe – scanf(“expressão de controle”, lista de argumentos) –, observamos que possui argumentos que declaram o formato dos valores a serem lidos e os endereços das variáveis que irão receber esses valores da fonte externa.

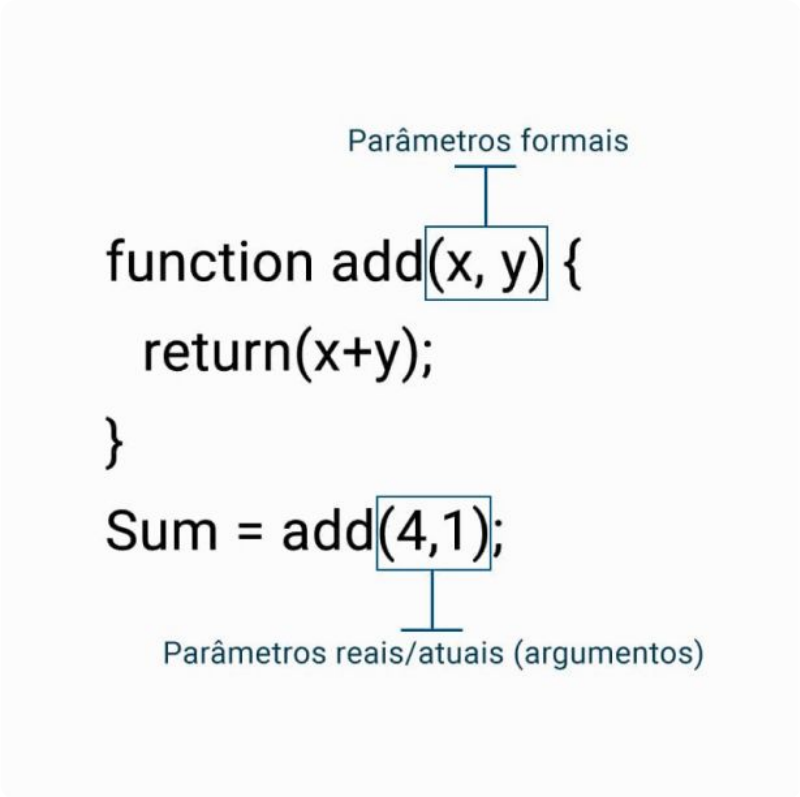
Sendo assim, vamos entender a passagem desses valores entre o programa principal e suas sub-rotinas.

Na computação, um parâmetro é uma variável que recebe valores passados para sua sub-rotina. Esta utiliza esses valores atribuídos aos parâmetros para alterar o seu comportamento em tempo de execução.

Parametriza-se, portanto, a sub-rotina com o objetivo de receber informações que virão posteriormente, durante a execução do programa.

> Os parâmetros, também chamados de parâmetros formais, são definidos na declaração da sub-rotina. O nome de cada um deles serve como uma variável local dentro da função. Os argumentos, também conhecidos por parâmetros reais/atuais, representam os valores, ou seja, as variáveis atuais passadas para os parâmetros da sub-rotina, quando é invocada.

O exemplo a seguir demonstra essas definições: considere uma função que realiza a operação de adição de dois números inteiros. Ela (**function add**) possui dois parâmetros: x e y, que são os parâmetros formais. Ao ser invocada a **função add** através da linha de comando **sum = add(4,1);** são passados dois argumentos, que são os números 4 e 1. Veja:



Exemplo de definição.

Passagem de parâmetros

Quando invocamos uma sub-rotina, devemos fornecer para cada um dos seus parâmetros um valor de mesmo tipo, respeitando a ordem e a quantidade de parâmetros declarados. Essa passagem pode ser feita de duas formas:

Passagem por valor

A primeira é através de uma cópia do valor do argumento na chamada da sub-rotina para o parâmetro declarado na sub-rotina, que se denomina passagem por valor.

Passagem por referência

A segunda maneira é através da passagem do endereço onde se encontra a variável usada como argumento na chamada da sub-rotina, conhecida por passagem por referência.

Para ilustrar os dois tipos de passagem, vamos considerar a troca de valores entre variáveis, que é um problema frequente de programação e pode ser usado na ordenação de um vetor, por exemplo. Uma troca é realizada em três passos e sempre com o auxílio de uma variável auxiliar.

O Programa 5, a seguir, cria uma função troca() que realiza a troca de valores entre duas variáveis:

```c
//Programa 5
#include < stdio.h >
void troca(int x, int y) {
    int aux;
    aux = x;
    x = y;
    y = aux;
}

int main(){
    int v1=5, v2=10;
    printf("Antes da troca: v1=%d e v2=%d\n", v1, v2);
    troca(v1,v2);
    printf("Depois da troca: v1=%d e v2=%d\n", v1, v2);
    exit(0);
}
```

O programa principal invoca a sub-rotina troca() com dois argumentos, a variável v1 e a variável v2, na linha 13. Acompanhe:

```c
troca(v1,v2);
```

Os parâmetros da sub-rotina troca(), x e y, recebem os valores 5 e 10, respectivamente. Esses parâmetros são as variáveis da sub-rotina.

A sub-rotina troca() possui também uma variável “aux”, que recebe o valor do parâmetro x. Em seguida, a variável x recebe o valor da variável y, e a variável y recebe o valor de aux.

Sendo assim, ao executar o programa:

- O programa principal declara duas variáveis inteiras, v1 e v2, que recebem os valores 5 e 10, respectivamente.
- O programa principal invoca a sub-rotina troca() com dois argumentos, o valor da variável v1 e o valor da variável v2 - troca(5, 10).
- A sub-rotina troca() recebe os valores nos seus parâmetros, x e y - troca(5,10).
- A sub-rotina executa os comandos: Variável local aux recebe o valor do parâmetro x, x recebe o valor do parâmetro y, e y recebe o valor da variável aux, ficando as variáveis com os seguintes valores: aux = 5, x = 10 e y = 5.
- A troca de valores, porém, não foi realizada na função main, apenas na função troca.

Ao invocarmos uma sub-rotina passando variáveis no lugar dos parâmetros, os valores das mesmas são copiados para os parâmetros da sub-rotina. Sendo assim, as alterações dentro da sub-rotina no valor dos parâmetros não afetam as variáveis usadas na chamada desta.

Essa forma de relacionamento do programa principal com a sub-rotina chamamos de **passagem por valor**. Nela, é passada realmente uma cópia do valor indicado na chamada do procedimento ou função. Se alterarmos o conteúdo da variável recebida como parâmetro, alteraremos a cópia do valor original e, portanto, o original não é modificado.

Comentário

Podemos dizer que a passagem por valor é feita entregando a “xerox” das informações, jamais o original. Dessa forma, a sub-rotina pode “riscar e rabiscar” os dados que foram passados, sem, no entanto, danificar o “documento original”.

Voltando ao exemplo anterior, quando o programa principal é executado, são criadas as variáveis v1 e v2. Por terem sido declaradas na função principal do programa, sua visibilidade (ou seja, o escopo para se manipulá-las) é restrita a esta função.

Quando a função troca() é invocada, novas variáveis x e y, visíveis apenas dentro da função troca(), são criadas, suspendendo temporariamente a visibilidade de v1 e v2 criadas em main(). A chamada de troca() em main(), porém, resulta na passagem da cópia dos valores de v1 e v2 de main() para as novas variáveis x e y que são criadas em troca().

Assim, dentro da função troca(), quaisquer alterações de x e y não resultarão em modificações de v1 e v2 em main(), pois o que está sendo alterado é apenas uma cópia. Ou seja, caracterizou a passagem por valor. Mas quando desejamos passar uma variável para uma sub-rotina de modo que seu valor possa ser alterado pela sub-rotina, devemos utilizar a **passagem por referência**.

Nesse caso, o parâmetro da sub-rotina deve ser declarado como ponteiro para um tipo. E, na chamada da sub-rotina, deve-se passar o endereço de memória da variável, de mesmo tipo do ponteiro, que terá seu valor alterado.

O artifício corresponde em passar como argumento para a sub-rotina o endereço de memória da variável, não o seu valor. Portanto, é enviada para a função não uma cópia do valor da variável, e sim uma referência a esta. Assim, qualquer alteração dos parâmetros realizada na função corresponde a uma alteração nas variáveis originais.

Então, para realizar a passagem por referência, ponteiros devem ser utilizados. A estratégia a ser adotada é:

Passo 1

Passar o endereço de uma variável, obtido através do operador & e armazenado em uma variável do tipo ponteiro.

Passo 2

Dentro da função, usar ponteiros para alterar os locais para onde eles apontam e que correspondem aos valores das variáveis originais.

Dessa forma, podemos alterar o conteúdo da variável pela passagem por referência.

Vamos voltar ao Programa 5, para reescrevê-lo e alterar o valor das variáveis. Chamaremos de Programa 6:

```c
//Programa 6
#include < stdio.h >
void troca(int *x, int *y) {
    int aux;
    if(x != NULL && y != NULL){ //se endereços forem válidos
    aux = *x; //faz a troca
    *x = *y;
    *y = aux;
    }
}

int main(){
    int v1=5, v2=10;
    troca(&v1, &v2);
    printf("v1 = %d e v2 = %d\n", v1, v2);
}
```

Note que a variável aux continua a ser declarada do tipo int, pois seu propósito é armazenar um dos valores e não endereço.

A sub-rotina troca() é invocada, e nos seus argumentos, no lugar dos valores das variáveis v1 e v2, encontramos o endereço de memória das variáveis ao se usar o operador “&;”. Observe:

```c
troca(&v1, &v2);
```

A sub-rotina troca() possui nos seus parâmetros os ponteiros *x e *y, responsáveis em receber o endereço de memória das variáveis v1 e v2. Veja:

```c
void troca(int *x, int *y)
```

Sendo assim, ao executar o programa:

- O programa principal declara as variáveis x e y, que recebem os valores 5 e 10, respectivamente (x=5 e y=10).
- O programa principal invoca a sub-rotina troca(), que passa por referência o endereço de memória das variáveis v1 e v2, por exemplo: troca(6487580, 6487576).
- A sub-rotina troca() recebe os endereços de memória das variáveis v1 e v2 nos seus parâmetros *x e *y (void troca(6487580, 6487576)).
- Os ponteiros *x e *y, parâmetros da sub-rotina, agora estão com os endereços das variáveis v1 e v2.
- A variável aux, declarada localmente na sub-rotina, recebe o valor da variável v1, apontada pelo ponteiro *x (aux = 5).
- O valor armazenado no endereço de memória apontado por *x é trocado pelo valor armazenado no endereço de memória apontado pelo *y.

- O valor armazenado no endereço de memória apontado por *y é trocado pelo valor da variável aux.

Na passagem por referência, se alterarmos o conteúdo da variável recebida como parâmetro, alteraremos o valor original, pois temos à nossa disposição o endereço exato onde está localizada na memória e, portanto, podemos alterar diretamente o valor original.

Quando usamos a passagem por referência, não podemos passar valores numéricos ou expressões na chamada da função. Deve ser uma variável única onde possamos definir exatamente o seu endereço.

Conforme mencionado, todo parâmetro das sub-rotinas é destruído ao final da execução da mesma. Isso também é válido para a passagem por referência, mas na realidade o que é destruído é a referência à variável usada no programa principal, e não a própria variável.

Atividade 1

Você está desenvolvendo um programa em C que deve trocar os valores de duas variáveis inteiras. Para garantir que a troca afete as variáveis originais no programa principal, qual método de passagem de parâmetros você deve utilizar e como deve ser a declaração da função de troca?

A

Utilizar passagem por valor, com a função declarada como void troca(int x, int y).

B

Utilizar passagem por valor, com a função declarada como void troca(int *x, int *y).

C

Utilizar passagem por referência, com a função declarada como void troca(int x, int y).

D

Utilizar passagem por referência, com a função declarada como void troca(int *x, int *y).

E

Utilizar passagem por referência, com a função declarada como int troca(int *x, int *y).

A alternativa D está correta.

Para trocar os valores de duas variáveis inteiras de forma que a alteração reflita no programa principal, deve-se usar a passagem por referência. Isso é feito passando os endereços das variáveis para a função. A declaração correta é void troca(int *x, int *y), na qual x e y são ponteiros que apontam para os endereços das variáveis a serem trocadas. As outras opções são incorretas porque utilizam passagem por valor ou declarações inadequadas, que não modificam as variáveis originais.

Protótipos de funções

Os protótipos de sub-rotinas são declarações essenciais em C que permitem a implementação de funções e procedimentos em diferentes partes do arquivo-fonte. Ao definir protótipos ou assinaturas de sub-rotinas no início do programa, você garante que o compilador reconheça suas referências antes mesmo de suas

definições completas. Isso facilita a organização do código e permite chamadas a sub-rotinas antes de suas implementações. Exploraremos a importância dos protótipos de sub-rotinas, sua sintaxe e como usá-los para criar programas mais estruturados e eficientes.

Neste vídeo, você vai ver que protótipos de sub-rotinas em C permitem implementar funções e procedimentos em diferentes partes do arquivo-fonte, garantindo que o compilador reconheça referências antes das definições completas. Verá ainda que eles facilitam a organização do código e permitem chamadas a sub-rotinas antes de suas implementações.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Na passagem por valor ou por referência, é necessária a implementação de sub-rotinas no programa principal e, para implementá-la sem partes distintas do arquivo-fonte e depois de utilizá-las, usamos protótipos ou assinaturas de sub-rotina.

Os protótipos ou assinaturas de sub-rotinas correspondem à primeira linha da definição de uma função ou procedimento. Ela deve aparecer antes do uso dessa sub-rotina.

Em geral, colocam-se os protótipos no início do arquivo-fonte. Observe a sintaxe logo abaixo de duas sub-rotinas:

```
c

tipo nome da função (tipo parâmetro1, ..., tipo parâmetro) {
    comandos;
    return;
}

void nome do procedimento (tipo parâmetro1, …, tipo parametron) {
      commandos;
}
```

A primeira linha das sub-rotinas tipo nome da função (tipo parâmetro1, ..., tipo parâmetron) e void nome do procedimento (tipo parâmetro1, ..., tipo parametron) é a assinatura da função e do procedimento, respectivamente. E deve ser inserida no início do programa ou antes do uso da sub-rotina, como apresentado no Programa 7:

```c
//Programa 7
#include< stdio.h >;

int soma(int x, int y);//assinatura da função

int main(void) {
    int num1, num2, num3;
    printf("Digite primeiro numero:");
    scanf("%d", &num1);
    printf("Digite segundo numero:");
    scanf("%d", &num2);
    num3 = soma(num1, num2); // Chamada da função
    printf("A soma de: %d + %d = %d\n", num1, num2, num3);
    return(0);
}

int soma(int x, int y){ // Definição da função
    return (x+y);
}
```

Na linha 4, temos a referência à assinatura da função soma e, na linha 18, a sua definição. Observe que o programa principal invoca a função soma na linha 12, antes da sua definição, e isso somente é possível porque foi inserida a assinatura no início do programa.

Atividade 2

Você está desenvolvendo um programa em C que precisa calcular a média de dois números. Você decide utilizar uma função para realizar esse cálculo, mas deseja definir a função após a função main. Como você deve proceder para que o programa compile corretamente?

A

Definir a função média antes da função main e não utilizar protótipos.

B

Utilizar um protótipo da função média antes da função main e definir a função após a função main.

C

Definir a função média dentro da função main.

D

Não é possível utilizar uma função definida após a função main.

E

Utilizar a função média sem protótipo, definindo-a após a função main.

A alternativa B está correta.

Para que o programa compile corretamente quando a função é definida após a função main, é necessário declarar um protótipo da função antes da função main. Isso informa ao compilador sobre a existência da função e sua assinatura, permitindo que ela seja chamada antes de sua definição. A opção B é a correta porque segue essa prática, enquanto as outras opções são inadequadas, pois não utilizam protótipos ou sugerem práticas incorretas.

Passagem de vetores

A passagem de vetores para funções em C é sempre realizada por referência, permitindo que as funções modifiquem diretamente os elementos do vetor original. A sintaxe para passar um vetor a uma função utiliza um ponteiro, indicando o endereço inicial do vetor. Isso possibilita a manipulação eficiente de grandes conjuntos de dados sem a necessidade de copiar todos os elementos. Vamos explorar como declarar, inicializar e manipular vetores dentro de funções, destacando a importância de entender a passagem de vetores por referência para criar programas eficientes e bem estruturados.

Neste vídeo, vamos ver a passagem de um vetor como parâmetro de uma função. A passagem de vetores é peculiar, e vamos ver o motivo.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A passagem de vetores para funções é sempre por referência. A sintaxe de passagem de um vetor pode ser feita com:

```
c

tipo nome[ ];
```

Onde: Tipo corresponde ao tipo dos elementos do vetor, nome é o nome atribuído ao vetor e [ ] indica que a variável é do tipo vetor.

O “[ ]” pode ser utilizado sem um valor, pois em C não interessa qual a dimensão do vetor que é passado a uma função, mas, sim, o tipo dos seus elementos.

Comentário

Para entender melhor a definição, observe o Programa 8. Ele declara uma função v_iniciacao(), que inicializa o vetor de inteiros com zero. Em seguida, o vetor de 10 posições é alimentado por números inteiros de 0 a 9 e, por último, a função v_imprime() imprime os valores do vetor.

Para passar um vetor para uma função usando linguagem C, utiliza-se um ponteiro como parâmetro da função. Ao usarmos o ponteiro como parâmetro, estamos passando o endereço inicial do vetor, não os seus elementos. Veja:

```c
//Programa 8
#include < stdio.h >

void v_iniciacao(int *vet, int n) {
    int i;
    for (i=0; i < n; i++)
    vet[i]=0;
}

void v_imprime(int *vet, int n){
    int I;
    for(i=0; i < n; i++)
      printf(" %d - ",vet[i]);
    printf("\n");
}

int main(void) {
    int vet[10], i;
    v_iniciacao(vet,10);
    // Impressao apos inicializacao.
    printf("Impressao do vetor antes da atribuicao.");
    v_imprime(vet,10);
    // Mudando valores de vet.
    for(i=0; i < 10; i++){
    vet[i]=i;
    }
    // Impressao apos atribuicao.
    printf("Impressao do vetor apos a atribuicao.");
    v_imprime(vet,10);
    return(0);
}
```

Atividade 3

Você está desenvolvendo um programa em C que precisa inicializar um vetor de inteiros com zeros e, em seguida, imprimir seus valores. Quais funções você deve criar e como deve passar o vetor para essas funções para garantir que seja corretamente manipulado?

A

Criar funções que utilizem passagem por valor e passar o vetor diretamente.

B

Criar funções que utilizem passagem por valor e passar cada elemento do vetor individualmente.

C

Criar funções que utilizem passagem por referência, declarando os parâmetros como ponteiros para inteiro.

D

Criar funções que utilizem passagem por referência, declarando os parâmetros como vetores de tamanho fixo.

E

Criar funções que utilizem passagem por valor, mas alocar dinamicamente o vetor dentro das funções.

A alternativa C está correta.

Para manipular corretamente um vetor dentro de funções, você deve usar a passagem por referência, o que é feito declarando os parâmetros da função como ponteiros para inteiro. Isso permite que a função acesse e modifique diretamente os elementos do vetor original. A opção C é a correta porque segue essa prática, enquanto as outras opções não permitem a manipulação direta dos elementos do vetor ou sugerem práticas inadequadas para o contexto.

Usando passagem de parâmetros na prática

Na linguagem C, a passagem de parâmetros em sub-rotinas, como funções e procedimentos, é um conceito fundamental que afeta diretamente a forma como os dados são manipulados e compartilhados dentro de um programa. Existem duas principais formas de passagem de parâmetros: por valor e por referência.

Na passagem por valor, uma cópia do valor do argumento é passada para a sub-rotina, o que significa que alterações feitas no parâmetro dentro da sub-rotina não afetam a variável original. Por outro lado, na passagem por referência, o endereço da variável é passado para a sub-rotina, permitindo que alterações feitas no parâmetro afetem diretamente a variável original.

Essa técnica é especialmente útil para manipular grandes estruturas de dados, como vetores e estruturas complexas, sem a sobrecarga de copiar os dados inteiros. Entender quando e como usar cada tipo de passagem de parâmetro é essencial para escrever programas eficientes e bem-organizados em C.

Neste vídeo, vamos acompanhar a elaboração de uma função e usar os mecanismos de passagem de parâmetros.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1:

Definição da função dobrar_valor utilizada na passagem de parâmetro por valor.

```c
#include

// Função que dobra o valor (passagem por valor)

void dobrar_valor(int num) {

    num = num * 2;  // Modifica apenas a cópia do valor

    printf("Dentro da função (por valor), o valor dobrado é: %d\n", num);

}
```

Passo 2:

Definição da função dobrar_valor_ref utilizada na passagem de parâmetro por referência.

```c
// Função que dobra o valor (passagem por referência)

void dobrar_valor_ref(int *num) {

    *num = *num * 2;  // Modifica o valor original usando o ponteiro

    printf("Dentro da função (por referência), o valor dobrado é: %d\n", *num);

}
```

Passo 3: Implementação do programa principal

```c
int main() {
    int numero = 10;

    // Passagem por valor
    printf("Antes da função (por valor), o número é: %d\n", numero);
    dobrar_valor(numero);
    printf("Depois da função (por valor), o número é: %d\n", numero);

    // Passagem por referência
    printf("\nAntes da função (por referência), o número é: %d\n", numero);
    dobrar_valor_ref(&numero);  // Passamos o endereço da variável
    printf("Depois da função (por referência), o número é: %d\n", numero);

    return 0;
}
```

Passo 4: Compilação e execução

Compile o programa usando um compilador C, como o GCC:

```c
gcc -o media_programa media_programa.c
```

Execute o programa:

```c
./media_programa
```

Passo 5: Validação dos resultados

Verifique a saída do programa para garantir que a média das notas seja calculada e impressa corretamente. A média das notas {80, 90, 70, 85, 95} deve ser 84.00. Se a saída for correta, o programa está funcionando conforme esperado.

Faça você mesmo!

Você está desenvolvendo um programa em C que deve calcular a média de um conjunto de notas armazenadas em um vetor. A função que calcula a média deve receber o vetor de notas e o número de elementos no vetor. Como você deve declarar e implementar essa função, e como deve chamar a função no programa principal?

A

Declarar a função com passagem por valor e passar o vetor diretamente: float calcular_media(int notas[], int n).

B

Declarar a função com passagem por valor e passar cada elemento do vetor individualmente.

C

Declarar a função com passagem por referência, utilizando ponteiros: float calcular_media(int *notas, int n).

D

Declarar a função com passagem por referência, mas alocar dinamicamente o vetor dentro da função.

E

Declarar a função sem parâmetros e acessar o vetor como variável global.

A alternativa C está correta.

Para calcular a média dos elementos de um vetor, a função deve ser capaz de acessar todos os elementos do vetor. Declarar a função utilizando ponteiros permite passar o vetor por referência, evitando a cópia desnecessária de dados e permitindo a manipulação direta dos elementos do vetor. A opção C é a correta, pois segue essa prática, enquanto as outras opções não são adequadas para manipular o vetor eficientemente ou não utilizam boas práticas de programação.

O que é escopo

Em linguagens de programação, o escopo das variáveis define onde e como essas variáveis podem ser acessadas. Na linguagem C, as variáveis podem ser classificadas em três tipos de escopo: variáveis locais, variáveis globais e parâmetros formais. Variáveis globais são declaradas fora de sub-rotinas e podem ser acessadas por qualquer parte do programa. Variáveis locais são declaradas dentro de sub-rotinas e só podem ser usadas dentro dessas sub-rotinas. Parâmetros formais são declarados na lista de parâmetros de uma sub-rotina e são utilizados exclusivamente dentro da sub-rotina onde são definidos. Compreender esses conceitos é fundamental para gerenciar dados e funções de forma eficiente em um programa.

Você vai ver que o escopo das variáveis em C define onde e como elas podem ser acessadas. Verá ainda que variáveis podem ser locais, globais ou parâmetros formais, determinando sua disponibilidade e uso dentro do programa.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Nas linguagens de programação, as variáveis são vinculadas aos seus valores em tempo de execução. E o escopo de uma vinculação é o conjunto de trechos de um programa que se consegue usar as variáveis.

No caso da linguagem c, as variáveis são divididas quanto ao escopo em três tipos:

- Variáveis locais
- Variáveis globais
- Parâmetros formais

Sendo assim, podemos declarar as variáveis em três locais distintos:

1

Primeiro local

Fica fora das sub-rotinas de um programa. Essas são chamadas de variáveis globais e podem ser usadas a partir de qualquer lugar do programa. Como elas estão fora das sub-rotinas, podemos dizer que todas as funções as veem.

2

Segundo local

Fica dentro da sub-rotina de um programa. Essas são chamadas de variáveis locais e só têm validade dentro da sub-rotina no qual são declaradas, isto é, só a sub-rotina em que a variável é declarada sabe da sua existência.

3Terceiro local

Está na lista de parâmetros de uma sub-rotina. Essas são chamadas de parâmetros formais e são declaradas na definição dos parâmetros de uma sub-rotina. Apesar de receberem valores externos, elas são conhecidas apenas pela sub-rotina onde são declaradas.

Atividade 1

Você está desenvolvendo um programa em C que calcula a soma de dois números e imprime o resultado. Para isso, você utiliza uma função soma que recebe dois parâmetros. Como você deve declarar as variáveis que armazenam os números e o resultado para que sejam acessíveis em toda a função main e na função soma?

A

Declarar todas as variáveis como globais para que possam ser acessadas de qualquer lugar no programa.

B

Declarar todas as variáveis como locais dentro da função soma.

C

Declarar as variáveis que armazenam os números como globais e o resultado como local dentro da função soma.

D

Declarar as variáveis que armazenam os números como locais dentro da função main e o resultado como global.

E

Declarar as variáveis que armazenam os números como locais dentro da função main e passar esses valores como parâmetros para a função soma.

A alternativa E está correta.

Declarar as variáveis dentro da função main e passar os valores como parâmetros para a função soma é a prática mais adequada, pois mantém o escopo das variáveis limitado ao contexto em que são necessárias, evitando efeitos colaterais indesejados e tornando o código mais modular e fácil de entender. A opção E é a correta porque promove boas práticas de programação, enquanto as outras opções utilizam escopos inadequados ou ineficientes para essa situação.

Variáveis locais e globais

O escopo das variáveis em C define onde e como as variáveis podem ser acessadas dentro de um programa. As variáveis podem ser classificadas como locais, globais ou parâmetros formais. Variáveis locais são declaradas dentro de sub-rotinas e só são acessíveis dentro dessas sub-rotinas. Variáveis globais são declaradas fora de qualquer sub-rotina e podem ser acessadas por todo o programa. Parâmetros formais são

variáveis locais que são declaradas na lista de parâmetros de uma sub-rotina e recebem valores quando a sub-rotina é chamada. Compreender o escopo das variáveis é crucial para escrever programas claros e evitar conflitos de nomes e erros.

Este vídeo vai demonstrar que o escopo das variáveis em C determina onde e como elas podem ser acessadas. Mostra também que podem ser locais, globais ou parâmetros formais, impactando a visibilidade e uso dentro do programa.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Variáveis locais

As variáveis locais são declaradas dentro de um bloco de código de uma sub-rotina. Só podem ser usadas ou modificadas pela sub-rotina onde estão declaradas, isto é, dentro desta ou do bloco de código do programa.

Curiosidade

A existência dessas variáveis depende da execução da sub-rotina onde estão declaradas, isso significa que somente existem enquanto a sub-rotina onde foi declarada estiver sendo executada.

Conforme visualizado no Programa 9, a seguir, na linha 5 foram declaradas na função principal **main()**três variáveis locais: **num1**, **num2** e **num3**, isto é, essas três variáveis pertencem à função principal main() e seu escopo está relacionado à execução da função.

```
c

1   //Programa 9 - soma dois numeros
2   #include < stdio.h >
3   int main ()
4   {
5       int num1, num2, num3;
6       num1 = 10;
7       num2 = 20;
8       num3 = num1 + num2;
9       printf("%d + %d = %d", num1, num2, num3);
10      return(0);
11  }
```

Qualquer valor que as variáveis num1, num2 e num3 assumam será válido apenas dentro da função main. Supondo que existisse uma sub-rotina nesse programa, essas variáveis não poderiam ser acessadas.

Variáveis globais

As variáveis globais são declaradas fora da sub-rotina. São acessíveis em todos os escopos, em qualquer ponto de um programa, mesmo em outros módulos.

Podem, portanto, ser usadas ou modificadas por qualquer sub-rotina do programa onde estão declaradas. Em outras palavras, as variáveis globais estão disponíveis durante toda a execução do programa.

O valor de uma variável global ao longo do ciclo de vida do programa é sempre válido. Conforme visualizado no Programa 10, na linha 3 foi declarada a variável num3 como uma variável global:

```c
//Programa 10 - soma dois numeros
#include < stdio.h >
int num3;
int main ()
{
    int num1, num2;
    num1 = 10;;
    num2 = 20;
    num3 = num1 + num2;
    printf("%d + %d = %d", num1, num2, num3);
    return(0);
}
```

Em um programa, as variáveis locais e as globais podem ter o mesmo identificador, isto é, o mesmo nome. Porém, dentro de uma sub-rotina, as variáveis locais sobrepõem o valor das globais.

No Programa 11, apresentado a seguir, encontramos uma declaração de variável global e variável local com o mesmo nome nas linhas 3 e 7, respectivamente, que é a variável num.

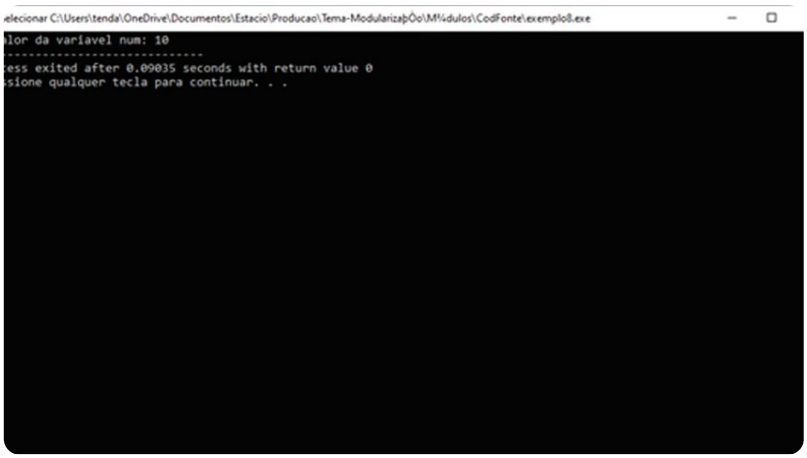
Comentário

Repare que na linha 4 a variável global num recebe o valor 20. Esse valor é válido apenas externamente à função main. Como nesta existe uma variável com o mesmo nome, quando, na linha 8, a variável num recebe o valor 10, esta é a mais interna ao programa, ou seja, a variável local que foi definida na linha 7.

Para comprovar, vamos executar o Programa 11. Podemos observar que, ao executar a função printf na linha 9, o valor impresso será o valor 10, que é o atribuído a variável local num:

```c
//Programa 11 - imprime o valor da variável num
#include < stdio.h >
int num;
num = 20;
int main ()
{
    int num;
    num = 10;
    printf("O valor da variavel num: %d", num);
    return 0;
}
```

Na imagem a seguir, podemos observar a saída da execução do programa 11:



Execução do Programa 11.

Parâmetros formais

Os parâmetros formais de uma sub-rotina também são variáveis locais. Eles são declarados como sendo as entradas de uma sub-rotina, sendo assim, são variáveis locais da função. É possível alterar o valor de um parâmetro formal, pois essa alteração não terá efeito na variável que foi passada à sub-rotina.

Na linguagem C, quando se passa parâmetros para uma sub-rotina, são passadas apenas cópias das variáveis. Isso significa que os parâmetros formais existem independentemente das variáveis que foram passadas para a sub-rotina, eles tomam apenas uma cópia dos valores passados.

Comentário

Observando o Programa 12, a seguir, na linha 7 foi declarada a função soma com dois parâmetros formais, x e y. Essa função recebe dois números inteiros e retorna a soma desses dois números.

Na linha 18, a variável local **num3** recebe o retorno da função soma.

Os parâmetros formais da função **soma()** são variáveis locais dessa função, seus valores são visíveis somente durante a execução da função.

```c
//Programa 12: soma dois numeros
#include < stdio.h >
//variável global
int num3;

//Função soma
int soma(int x, int y){
    int v_soma;
    v_soma = x + y;
    return(v_soma);
    }

int main ()
{
    int num1, num2;
    num1 = 10;
    num2 = 20;
    num3=soma(num1, num2);
    printf("%d + %d = %d", num1, num2, num3);
    return 0;
}
```

Nesse exemplo, podemos observar ainda sobre variáveis globais: Repare que na linha 4 foi definida a variável global num3. Agora considere que também tenha sido declarada uma variável global num1. Como visto anteriormente, como também há a num1 na função main, a que será manipulada dentro da função main é a variável local declarada na linha 15, portanto, assumindo o valor 10.

E o que aconteceria se na função soma fosse utilizada a variável num1? Qual das duas variáveis num1 seria acessada? Claro que seria a variável global num1, que tem escopo em qualquer parte do programa.

Atividade 2

Você está desenvolvendo um programa em C que deve calcular a soma de dois números e armazenar o resultado em uma variável global. Para isso, você utiliza uma função soma que recebe dois parâmetros. Onde você deve declarar a variável que armazenará o resultado da soma para que ela seja acessível tanto no programa principal quanto na função soma?

A

Declarar a variável como global e utilizar passagem por referência.

B

Declarar a variável como local dentro da função main.

C

Declarar a variável como local dentro da função soma.

D

Declarar a variável como global fora de qualquer sub-rotina.

E

Declarar a variável como parâmetro formal na função soma.

A alternativa D está correta.

Para que a variável que armazena o resultado da soma seja acessível tanto no programa principal quanto na função soma, ela deve ser declarada como global. Declarar a variável fora de qualquer sub-rotina garante que ela possa ser acessada e modificada por qualquer parte do programa. As outras opções são inadequadas porque declaram a variável em um escopo que não permite seu acesso em ambas as funções ou utilizam técnicas não necessárias para esse cenário específico.

Usando escopo na prática

O escopo das variáveis na linguagem C define onde e como as variáveis podem ser acessadas dentro de um programa. Existem três tipos principais de escopo: variáveis locais, variáveis globais e parâmetros formais. Variáveis locais são declaradas dentro de sub-rotinas e só podem ser usadas dentro dessas sub-rotinas, existindo apenas durante a execução delas. Variáveis globais são declaradas fora de qualquer sub-rotina e podem ser acessadas por qualquer parte do programa, mantendo seu valor ao longo de toda a execução do programa.

Parâmetros formais são variáveis locais que são declaradas na lista de parâmetros de uma sub-rotina e são usadas para receber valores quando a sub-rotina é chamada. A compreensão do escopo das variáveis é essencial para evitar conflitos de nomes e garantir que os dados sejam manipulados corretamente em diferentes partes do programa. O uso adequado do escopo das variáveis ajuda a criar programas mais organizados, eficientes e fáceis de manter.

Nesta atividade, vamos explorar as regras de escopo da linguagem C e observar o comportamento e as consequências dessas regras.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1: Definição da variável global

```
c

#include

// Variável global para o total de itens no estoque

int totalEstoque = 100;
```

Passo 2:

Definição da função adicionarEstoque utilizada na adição de itens ao estoque.

```c
// Procedimento para adicionar itens ao estoque
void adicionarEstoque(int quantidade) {  // 'quantidade' é um parâmetro formal
    totalEstoque += quantidade;
    printf("Itens adicionados: %d\n", quantidade);
    printf("Estoque atualizado: %d\n", totalEstoque);
}
```

Passo 3:

Definição da função removerEstoque utilizada na remoção de itens do estoque.

```c
// Procedimento para remover itens do estoque
void removerEstoque(int quantidade) {  // 'quantidade' é um parâmetro formal
    if (quantidade <= totalEstoque) {
        totalEstoque -= quantidade;
        printf("Itens removidos: %d\n", quantidade);
        printf("Estoque atualizado: %d\n", totalEstoque);
    } else {
        printf("Erro: Quantidade insuficiente no estoque!\n");
    }
}
```

Passo 4: Implementação do programa principal

```c
// Função principal
int main() {
    // Variável local para armazenar a quantidade de itens a ser processada
    int quantidade;

    // Adicionar itens ao estoque
    quantidade = 20;  // 'quantidade' é um argumento passado para a função
    adicionarEstoque(quantidade);

    // Remover itens do estoque
    quantidade = 15;  // 'quantidade' é um argumento passado para a função
    removerEstoque(quantidade);

    // Tentar remover uma quantidade maior do que o estoque
    quantidade = 200;  // 'quantidade' é um argumento passado para a função
    removerEstoque(quantidade);

    return 0;
}
```

Passo 5: Compilação e execução

Compile o programa usando um compilador C, como o GCC:

```c
gcc -o soma_programa soma_programa.c
```

Execute o programa:

```c
./soma_programa
```

Passo 6: Análise e validação dos resultados

Verifique a saída do programa. A saída esperada deve ser:

```
c

10 + 20 = 30
```

Confirme que o programa calcula corretamente a soma dos dois números e imprime o resultado.

Passo 7: Exploração adicional

1. Modifique o programa para adicionar novas variáveis locais e globais.

2. Experimente declarar uma variável global para armazenar o resultado e modifique a função soma para usar essa variável global.

3. Observe as mudanças no comportamento do programa e discuta as implicações do uso de variáveis globais versus locais.

Exemplo de modificação adicional

Adicionando uma variável global

Declare uma variável global fora de qualquer sub-rotina.

```
c

#include

int soma(int x, int y); // Protótipo da função
int resultado_global; // Variável global

int main() {
    int num1, num2;

    // Inicializa as variáveis
    num1 = 10;
    num2 = 20;

    // Chama a função soma e armazena o resultado na variável global
    resultado_global = soma(num1, num2);

    // Imprime o resultado
    printf("%d + %d = %d\n", num1, num2, resultado_global);

    return 0;
}

int soma(int x, int y) {
    return x + y;
}
```

Compile e execute novamente o programa para observar o comportamento com a variável global.

Faça você mesmo!

Você está desenvolvendo um programa em C que deve calcular a soma de dois números, armazenar o resultado em uma variável e imprimir o resultado. A função soma deve receber dois parâmetros. Onde você

deve declarar a variável que armazenará o resultado para garantir que ela seja acessível no programa principal e na função soma?

A

Declarar a variável como global e utilizar passagem por referência.

B

Declarar a variável como local dentro da função main e retornar o resultado da função soma.

C

Declarar a variável como local dentro da função soma.

D

Declarar a variável como global fora de qualquer sub-rotina e modificar seu valor dentro da função soma.

E

Declarar a variável como parâmetro formal na função soma e retornar o resultado.

A alternativa B está correta.

Declarar a variável que armazenará o resultado da soma como local dentro da função main e retornar o resultado da função soma é a prática mais adequada. Isso mantém o escopo das variáveis limitado ao contexto no qual são necessárias, evitando efeitos colaterais indesejados e tornando o código mais modular e fácil de entender. A opção B é correta porque promove boas práticas de programação, enquanto as outras opções utilizam escopos inadequados ou técnicas desnecessárias para esse cenário.

Considerações finais

- Conceito de escopo de variáveis em C.
- Diferença entre variáveis locais, variáveis globais e parâmetros formais.
- Utilização de variáveis locais dentro de sub-rotinas.
- Acesso e modificação de variáveis globais em todo o programa.
- Importância dos parâmetros formais como variáveis locais.
- Passagem de parâmetros por valor e por referência.
- Declaração e uso de protótipos de funções.
- Manipulação de vetores através de passagem por referência.
- Implementação prática de funções para cálculo e manipulação de dados.
- Práticas recomendadas para organização e modularidade do código em C.

Explore +

Para saber mais sobre os assuntos tratados neste tema, assista aos vídeos do canal **Linguagem C Programação Descomplicada**.

Referências

SCHILDT, H. **C, completo e total.** 3. ed. São Paulo: Makron Books, 1996, cap. 5, p. 113.