Programar em C

Conteúdo

Capítulo 1

Capa

Programar em C

Índice>> Ir para o índice >>

Capítulo 2

Por que aprender a linguagem C

Em uma era onde o software está cada vez mais presente no nosso dia a dia é importante ter algumas bases de programação, e para tanto é importante ter um bom material com explicações claras e exemplos; e o livro Programar em C se presta bem ao exercício.

Mas por que *C* e não Java ou Basic, ou ainda Perl? Linguagens como o Java ou Perl são linguagens a base de bytecode interpretado por uma máquina virtual, sendo assim, não é um código interpretado diretamente pelo processador. Ao contrário de muitas linguagens de programação, o C permite ao programador endereçar a memória de maneira muito parecida como seria feito em Assembly. Linguagens como o Java ou o Perl fornecem mecanismos que permitem que o programador faça o seu trabalho sem ter que se preocupar com a atribuição de memória ou com apontadores. Geralmente isso é bom, uma vez que é bastante trabalhoso lidar com a alocação de memória quando escrevemos aplicações com algoritmos de alto nível. No entanto, quando lidamos com tarefas de baixo-nível como aquelas que um núcleo (kernel) tem obrigação de desempenhar, como a de copiar um conjunto de bytes para uma placa de rede, torna-se altamente necessário um acesso direto à memória — algo que não é possível fazer com Java. C pode ser diretamente compilado em código de máquina, e por isso é rápido e eficiente. Além disso, C permite personalizar como implementar cada coisa ao básico, como alocação de memória, permitindo adaptações para melhorar desempenho.

Vale lembrar que os softwares interpretadores de script ou bytecode, como Java e Python, são escritos em linguagens como C e C++.

Será uma surpresa que C seja uma linguagem tão popular?

Como num efeito dominó, a próxima geração de programas segue a tendência dos seus ancestrais. Sistemas operacionais desenvolvidos em C sempre têm bibliotecas de sistema desenvolvidas em C. Essas bibliotecas são usadas para criar bibliotecas de programa (como Xlib, OpenGL ou GTK), e seus desenvolvedores geralmente decidem usar a mesma linguagem das bibliotecas de sistema. Desenvolvedores de aplicação usam bibliotecas de programa para desenvolver processadores de texto, jogos, tocadores de mídia, etc. Muitos vão decidir trabalhar com a mesma linguagem que a biblioteca foi escrita, e assim o processo continua...

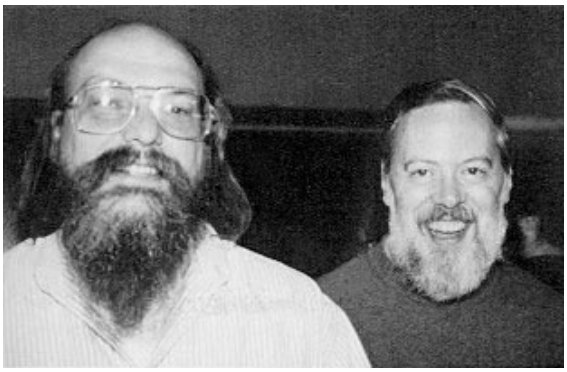
C é uma das linguagens de programação mais populares para se escrever sistemas operacionais, como o Microsoft Windows, o Mac OS X e o GNU/Linux. Sistemas operacionais comunicam-se diretamente com o hardware; não há nenhuma camada mais baixa para mediar seus pedidos. Originalmente, os sistemas operacionais eram escritos na linguagem Assembly, o que resultava em um código muito rápido e eficiente. Entretanto, escrever um sistema operacional em Assembly é um processo tedioso (lento), e produz um código que funcionará somente em uma arquitetura de CPU, tal como o x86 ou ARM. Escrever um sistema operacional em uma linguagem de alto nível, tal como C, possibilita que os programadores readaptem o sistema operacional a várias arquiteturas sem precisar reescrever todo o código. O núcleo (kernel) Linux é um exemplo de sistema operacional escrito em C, com apenas algumas seções do código escritas em Assembly, para poder executar instruções que só existem em uma ou outra arquitetura e para algumas otimizações.

Capítulo 3

História da linguagem C

3.1 História

3.1.1 Desenvolvimentos iniciais



Kenneth Thompson (à esquerda) e Dennis Ritchie (à direita), os criadores da linguagem C

O desenvolvimento inicial da linguagem C ocorreu nos laboratórios Bell da AT&T entre 1969 e 1973. Segundo Ritchie, o período mais criativo ocorreu em 1972. Deu-se o nome "C" à linguagem porque muitas das suas características derivaram de uma linguagem de programação anterior chamada "B". Há vários relatos que se referem à origem do nome "B": Ken Thompson dá crédito à linguagem de programação BCPL, mas ele também criou uma outra linguagem de programação chamada '*Bon*, em honra da sua mulher Bonnie.

Por volta de 1973, a linguagem C tinha se tornado suficientemente poderosa para que grande parte do núcleo de UNIX, originalmente escrito na linguagem de programação PDP-11/20 Assembly, fosse reescrito em C, tornando-se um dos primeiros núcleos de sistema operacional implementado em uma linguagem sem ser o Assembly. Como exemplos anteriores pode-se citar o sistema Multics (escrito em PL/I) e TRIPOS (escrito em BCPL).

3.1.2 C de K&R

Em 1978, Ritchie e Kernighan publicaram a primeira edição do livro *The C Programming Language*. Esse livro, conhecido pelos programadores de C como "K&R", serviu durante muitos anos como uma especificação informal da linguagem. A versão da linguagem C que ele descreve é usualmente referida como "C de K&R". (A segunda edição do livro cobre o posterior padrão ANSI C, descrito abaixo.) K&R introduziram as seguintes características na linguagem:

- Tipos de dados struct
- Tipos de dados long int
- Tipos de dados unsigned int
- O operador =+ foi alterado para +=, e assim sucessivamente (a análise léxica do compilador confundia o operador =+. Por exemplo, i =+ 10 e i = +10).

C de K&R é frequentemente considerado a parte mais básica da linguagem cujo suporte deve ser assegurado por um compilador C. Durante muitos anos, mesmo após a introdução do padrão C ANSI, ele era considerado o "menor denominador comum" em que programadores de C se apoiavam quando uma portabilidade máxima era desejada, já que nem todos os compiladores eram atualizados para suportar na íntegra o padrão C ANSI, e o código C de K&R razoavelmente bem escrito é também válido em relação ao C ANSI.

Nos anos que se seguiram à publicação do C K&R, algumas características "não-oficiais" foram adicionadas à linguagem, suportadas por compiladores da AT&T e de outros vendedores. Estas incluíam:

- Funções void e tipos de dados void *
- Funções que retornam tipos struct ou union
- Campos de nome struct num espaço de nome separado para cada tipo struct
- Atribuição a tipos de dados struct
- Qualificadores const para criar um objeto só de leitura
- Uma biblioteca-padrão que incorpora grande parte da funcionalidade implementada por vários vendedores

- Enumerações
- O tipo de ponto-flutuante de precisão simples

3.1.3 Os Padrões C ANSI e C ISO

Durante os finais da década de 1970, a linguagem C começou a substituir a linguagem BASIC como a linguagem de programação de microcomputadores mais usada. Durante a década de 1980, foi adotada para uso no PC IBM, e a sua popularidade começou a aumentar significativamente. Ao mesmo tempo, Bjarne Stroustrup, juntamente com outros nos laboratórios Bell, começou a trabalhar num projeto onde se adicionavam construções de linguagens de programação orientada por objetos à linguagem C. A linguagem que eles produziram, chamada C++, é nos dias de hoje a linguagem de programação de aplicações mais comum no sistema operativo Windows da companhia Microsoft; C permanece mais popular no mundo UNIX.

Em 1983, o Instituto Norte-Americano de Padrões (ANSI) formou um comité, X3j11, para estabelecer uma especificação do padrão da linguagem C. Após um processo longo e árduo, o padrão foi completo em 1989 e ratificado como ANSI X3.159-1989 "Programming Language C". Esta versão da linguagem é frequentemente referida como C ANSI. Em 1990, o padrão C ANSI, após sofrer umas modificações menores, foi adotado pela Organização Internacional de Padrões (ISO) como ISO/IEC 9899:1990. Um dos objetivos do processo de padronização C ANSI foi o de produzir um sobreconjunto do C K&R, incorporando muitas das características não-oficiais subsequentemente introduzidas. Entretanto, muitos programas tinham sido escritos e que não compilavam em certas plataformas, ou com um certo compilador, devido ao uso de bibliotecas de funções não-padrão e ao fato de alguns compiladores não aderirem ao C ANSI.

3.1.4 C99

Após o processo ANSI de padronização, as especificações da linguagem C permaneceram relativamente estáticas por algum tempo, enquanto que a linguagem C++ continuou a evoluir. (Em 1995, a Normative Ammendment 1 criou uma versão nova da linguagem C mas esta versão raramente é tida em conta.) Contudo, o padrão foi submetido a uma revisão nos finais da década de 1990, levando à publicação da norma ISO 9899:1999 em 1999. Este padrão é geralmente referido como "C99". O padrão foi adotado como um padrão ANSI em Março de 2000.

As novas características do C99 incluem:

- Funções em linha
- Levantamento de restrições sobre a localização da declaração de variáveis (como em C++)
- Adição de vários tipos de dados novos, incluindo o long long int (para minimizar a dor da transição de **32-bits** para **64-bits**), um tipo de dados boolean explicito e um tipo complex que representa números complexos
- Disposições de dados de comprimento variável
- Suporte oficial para comentários de uma linha iniciados por //, emprestados da linguagem C++
- Várias funções de biblioteca novas, tais como snprintf()
- Vários arquivos-cabeçalho novos, tais como stdint.h

O interesse em suportar as características novas de C99 parece depender muito das entidades. Apesar do GCC e vários outros compiladores suportarem grande parte das novas características do C99, os compiladores mantidos pela Microsoft e pela Borland não, e estas duas companhias não parecem estar muito interessadas adicionar tais funcionalidades, ignorando por completo as normas internacionais.

3.1.5 Resumo em inglês

Em 1947, três cientistas do Laboratório Telefonia Bell, William Shockley, Walter Brattain, e John Bardeen criaram o transistor. A computação moderna teve início. Em 1956 no MIT o primeiro computador completamente baseado em transistores foi concluído, the TX-0. Em 1958 na Texas Instruments, Jack Kilby construiu o primeiro circuito integrado. Mas mesmo antes do primeiro circuito integrado existir, a primeira linguagem de alto nível já tinha sido escrita.

Em 1954 Fortran, a Formula Translator, foi escrito. Começou como Fortran I em 1956. Fortran veio a ser Algol 58, o Algorithmic Language, em 1958. Algol 58 tornou-se Algol 60 em 1960. Algol 60 gerou CPL, o Combined Programming Language, em 1963. CPL passou a ser BCPL, Basic CPL, em 1967. BCPL engendrou B em 1969. E de B surgiu C em 1971.

B foi a primeira língua da linhagem C diretamente, tendo sido criado no Bell Labs por Ken Thompson. B era uma linguagem interpretada, utilizada no início, em versões internas do sistema operacional UNIX. Thompson e Dennis Ritchie, também da Bell Labs, melhorou B, chamando-NB; novas prorrogações para NB criaram C, uma linguagem compilada. A maioria dos UNIX foi reescrito em NB e C, o que levou a um sistema operacional mais portátil.

B foi, naturalmente, o nome de BCPL e C foi o seu sucessor lógico.

A portabilidade do UNIX foi a razão principal para a popularidade inicial de ambos, UNIX e C; pois ao invés de

criar um novo sistema operacional para cada nova máquina, system programmers could simply write the few system dependent parts required for the machine, and write a C compiler for the new system; and since most of the system utilities were written in C, it simply made sense to also write new utilities in the language.

Capítulo 4

Pré-requisitos

É pré-requisito para um bom aprendizado de qualquer linguagem de programação conceitos sobre lógica de programação.

Além disso, para programar em C, você precisa de um editor de textos e um compilador, discutidos a seguir.

4.1 Editor

Para editar o código de um programa, é apenas necessário um editor de textos, qualquer um, até mesmo o Bloco de Notas do Windows.

No entanto, há diversos editores que apresentam recursos que facilitam a edição de programas, como: destaque/coloração de sintaxe, complementação de código, formatação (indentação) automática, ajuda integrada, comandos integrados para compilar etc. Entre todos eles podemos destacar o Vim e o Emacs, ambos com versões para Windows, Linux e Mac OS.

Em sistemas GNU/Linux, a maioria dos editores de texto já possui recursos para facilitar a edição de programas em C. Principalmente, devido ao fato da maioria destes e boa parte do sistema terem sido programadas utilizando C ou C++.

Entretanto, o editor apenas edita o código. Para transforma-lo em linguagem de máquina e o executar, precisaremos de um compilador.

4.2 Compilador

O código em linguagem C consiste em instruções que o computador deverá seguir. O compilador realiza o trabalho de traduzir essas instruções para linguagem de máquina, de forma a poderem ser executadas pelo computador.

4.3 Ligador ou linker

A ligação de arquivos consiste na construção de uma imagem memória que contém partes de código compilados separadamente. Em outras palavras ele une os arquivos objetos e as bibliotecas (estáticas, dinâmicas) para formar uma nova biblioteca ou um executável.

4.4 Obtendo um compilador

Existem diversos compiladores disponíveis:

Para Windows ou DOS

- **MinGW** (antigo **mingw32**): uma espécie de *gcc* para Windows. É o compilador incluído com o **Dev-C++**, da Bloodshed. O Dev-C++ é um IDE (sigla em inglês para Ambiente Integrado de Desenvolvimento) que facilita a edição e compilação de programas. Tem tradução para Português do Brasil.

- **Borland C++**: a Borland disponibilizou um compilador gratuito que funciona em linha de comando, como alternativa ao IDE comercial.

- **DJGPP**: porte do *gcc* para DOS. Também funciona no Windows, mas se o objetivo for rodar no Windows, recomenda-se o uso do *mingw*, que pode usufruir de todos os recursos do Windows.

- **Microsoft Visual C++**: compilador comercial da Microsoft, que também tem um IDE. O Framework .NET, gratuito, também inclui o compilador (em linha de comando) do Visual C++.

- **Bloodshed DEV-C++**: ambiente de desenvolvimento integrado livre que utiliza os compiladores do projeto GNU para compilar programas para o sistema operacional Microsoft Windows.

Para Linux/Unix-like

- **gcc**: é um conjunto de compiladores oficiais do projeto GNU, de código aberto. Costumam vir instalados na maioria das distribuições GNU/Linux e está disponível para diversas plataformas, principalmente para as baseadas em sistemas do tipo unix.

- **GNU linker**: é o ligador do projeto GNU o nome do programa é “ld” e faz parte do pacote GNU Binary Utilities.

4.5 Links externos

- CodeBlocks: página para download do CodeBlocks, uma IDE para C ao estilo do Dev-C++, porém, mais nova.
- Dev-C++: página para download do Dev-C++.
- DJGPP: página oficial, com informações e links para download.
- GCC: página oficial do compilador para diversas plataformas.

Capítulo 5

Utilizando um compilador

5.1 Compiladores: visão geral

Um compilador é, geralmente, um programa de modo texto, que deve ser operado diretamente da linha de comando, sem nenhuma interface gráfica. Essa é uma das razões pelas quais muitas pessoas preferem usar IDEs. No entanto, saber um pouco sobre como usar o compilador pela linha de comando pode vir a ser útil, por exemplo quando você não tiver um IDE à disposição. Não é nenhum bicho-de-sete-cabeças, e a sintaxe da maioria dos compiladores é semelhante.

Para executar o compilador, você precisa abrir um terminal (ou "prompt de comando", como costuma ser chamado no Windows, ou ainda console). É lógico que se você estiver em um sistema sem ambiente gráfico (como o DOS), você não precisa fazer isso.

O Windows só tem um terminal nativo, que é o interpretador de comandos dele (cmd.exe ou command.com). Pacotes como o Cygwin e o MSys (do mesmo projeto que o MinGW) incluem terminais alternativos que funcionam basicamente à maneira do Linux.

No Linux, além dos terminais de modo texto, há vários emuladores de terminal, entre os quais estão o XTerm, o Konsole (KDE) e o Terminal do Gnome. O uso de todos eles é idêntico.

5.2 gcc

Com o gcc, compilador da GNU utilizado principalmente no sistema operacional linux ou de tipo unix (mas também com versão para a arquitetura/sistema operacional MSWindows®), você pode executar a compilação e a montagem separadamente ou com um único comando. Se você tem vários arquivos-fonte, é mais recomendável executar as duas etapas separadamente: se você atualizar apenas um arquivo, só precisará recompilar o que atualizou e depois remontar. No entanto, se você está desenvolvendo um projeto grande, é recomendável usar ferramentas de automação do processo de compilação, como o make.

Resumo:

gcc [OPÇÕES] nome_do_arquivo

Aqui são listadas algumas das opções do gcc:

- -c: Compila o código fonte mas não faz as ligações. A saída é um arquivo objeto.
- -o: serve para dar um nome ao arquivo de saída.
- -O2: ativa otimização no nível 2
- -g: salva os símbolos de depuração (o que permite usar um depurador)
- -Wall: ativa todos os avisos do compilador
- -pedantic: ativa os avisos necessários para que o código esteja estritamente de acordo com os padrões

Para compilar o arquivo "programa.c", gerando o código-objeto "programa.o":

gcc [OPÇÕES] -c programa.c

Para gerar o executável "programa binario" bin ou "programa.exe" no Windows/DOS a partir do código-objeto:

gcc [OPÇÕES] -o programa[.bin] programa.o

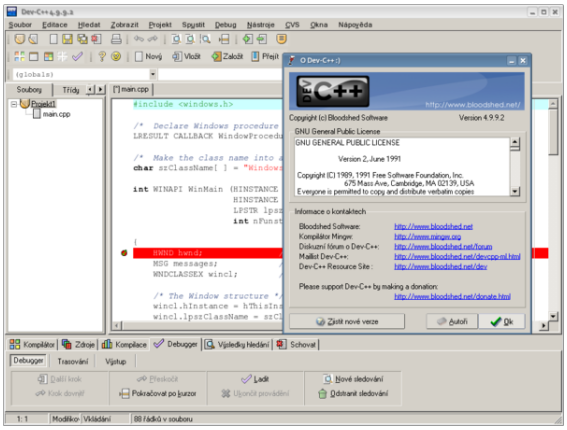
Para gerar o executável diretamente a partir do arquivo-fonte:

gcc [OPÇÕES] -o programa[.bin] programa.c

5.3 Dev-C++

Dev-C++ é uma IDE ("Integrated Development Environment") desenvolvido para ambiente MSWindows® (funcionando em ambientes *nix através do Wine). Embora ele possua uma interface gráfica, sua instalação inclui o compilador gcc do projeto mingw, completamente funcional via linha de comando.

Via interface gráfica a compilação é feita através do atalho de teclado <Ctrl>+F9. Via linha de comando, o compilador gcc utilizado por ele possui a maioria das mesmas opções básicas das versões *nix explicadas acima.



IDE Dev-C++

5.4 Visual C++

Em alguma versão não especificada do Visual C++, para compilar o arquivo “programa.c”, gerando o código-objeto “programa.obj":

cl /c programa.c

Para gerar o executável “programa.exe” a partir do código-objeto:

link /out:programa.exe programa.obj

Para gerar o executável a partir do arquivo-fonte:

cl programa.c

Capítulo 6

Noções de compilação

6.1 Compilação

Todo o código em linguagem C que escrevermos deve ser salvo em um arquivo, em formato texto, com a extensão ".c". Esse código não tem significado nenhum para a unidade de processamento; para que o processador possa executar nosso programa, este deve ser traduzido para a linguagem de máquina. Essa tradução se chama **compilação** e é feita pelo programa denominado compilador.

O compilador lê todo o código e cria um arquivo executável, em linguagem de máquina, específica para uma arquitetura de processadores e para um tipo de sistema operacional, o que significa que um programa compilado no Windows, por exemplo, não rodará nativamente no Linux se simplesmente copiarmos o executável. Devemos, para isso, recompilar o código-fonte do programa.

No Windows, os arquivos executáveis são aqueles com extensão ".exe". No Linux, os executáveis são simplesmente arquivos com o atributo "executável".

6.2 Etapas da compilação

O processo que chamamos corriqueiramente de compilação na verdade é um conjunto de etapas:

- o **preprocessamento**, etapa em que o pré-processador (programa às vezes acoplado ao compilador) lê o código-fonte e faz algumas substituições para que o programa possa ser compilado. Em C, o preprocessador tem diversos usos: compilação condicional (por exemplo, usar trechos diferentes do código para sistemas operacionais diferentes), macros, substituição de símbolos e inclusão de arquivos externos que declaram funções e variáveis.

- a **verificação sintática**, que procura por eventuais erros nos códigos dos programas: parênteses não fechados, falta de ponto-e-vírgula no final da instrução, etc. Todos esses problemas são alertados e causam a interrupção da compilação.

- a **compilação** propriamente dita, que transforma o código preprocessado em um *programa-objeto*, que está em linguagem de máquina porém não pronto para ser executado.

- a **linkedição** (*linking*, em inglês) dos programas-objeto e bibliotecas necessárias em um único executável, feita pelo *linkeditor* (*linker*). Em C, pode-se distribuir um programa em vários arquivos-fonte, o que ajuda na organização e permite compilar apenas a parte do programa correspondente quando é necessário realizar alguma mudança. Na montagem, todas as partes constituintes do programa são deslocadas e/ou cortadas conforme necessário para que tenhamos um programa executável.

Capítulo 7

Um programa em C

7.1 Um programa em C

É comum que o primeiro programa escrito em uma linguagem de programação seja um programa que escreve "Hello world!" ("Olá mundo!"). Apresentamos o código e, a seguir, analisaremos cada uma de suas linhas. Não se preocupe se não entender ainda alguns aspectos, tudo será abordado detalhadamente mais adiante.

Note que o número das linhas é dado apenas para facilitar a referência; se for copiar o código, lembre-se de tirar os números de linha.

1. /* o meu primeiro programa */ 2. #include <stdio.h> 3. int main() 4. { 5. printf ("Olá, mundo!"); 6. return (0); 7. }

O **texto** do programa também é conhecido como código do programa ou simplesmente **código fonte**. O código fonte é o programa escrito na linguagem de programação. Em nosso caso acima, chamamos código C ou simplesmente código.

Você deve copiar o código acima em um editor de texto como notepad e salvá-lo como **ola.c** (sem acento). Lembre-se de remover os números das linhas. Caso contrário o código não irá compilar. Esse arquivo agora representa o código fonte do programa escrito em C.

Salvando o código acima em um arquivo com a extensão ".c" e seguindo as instruções de compilação do do capítulo de utilização de compilador, você deverá ver como resultado um "Olá, mundo!" na tela. A seguir vamos a análise do código.

A primeira linha é um *comentário*, que para o compilador não tem nenhum significado. Qualquer texto que esteja entre as marcações /* e */, podendo inclusive ocupar várias linhas, será considerado como comentário e será completamente ignorado pelo compilador. É muito útil como documentação, explicando o que as próximas linhas de código fazem.

A linha 2 pede que seja inserido o conteúdo do arquivo *stdio.h* (que está num lugar já conhecido pelo compilador). Esse arquivo contém referências a diversas funções de entrada e saída de dados (*stdio* é abreviação de *Standard Input/Output*, ou Entrada e Saída Padronizadas), de modo que você precisará dele em praticamente todos os programas — ele é o meio de quase toda comunicação com o teclado, com a tela e com arquivos.[1]

Os programas em C são organizados em funções — todo código em C deve fazer parte de uma função. Em particular, todo programa deve ter uma função chamada **main**, pela qual será iniciada a execução do programa. A função é definida, no nosso exemplo, na linha 3, e delimitada pelas chaves { }.

A palavra-chave **int** significa que a função devolve um valor inteiro (você pode pensar nesse valor exatamente como o valor de uma função em matemática).

Na linha 5, executamos a função *printf*, que imprime na tela os parâmetros que lhe foram passados — no nosso exemplo, passamos a seqüência de caracteres "Olá, mundo!" como parâmetro. Essa é uma das funções definidas em um cabeçalho da biblioteca C, o arquivo *stdio.h*.

Note o ponto-e-vírgula no final da linha: todas as instruções em C devem terminar com um ponto-e-vírgula. (Essa é uma causa muito comum de erros de compilação).

Na linha 6, dizemos que a função *main* deve devolver (ou retornar) o valor 0 e terminar sua execução. (Esse é o valor inteiro que dissemos que íamos retornar na linha 3.)

O padrão da linguagem C diz que a função *main* deve devolver um valor inteiro, e esse valor diz se o programa foi executado com sucesso ou não. O valor zero indica que o programa foi finalizado sem nenhum erro, e valores diferentes de zero podem indicar diferentes erros. Você não precisará se preocupar com isso no início do seu estudo em C — o valor devolvido por um programa é geralmente usado em scripts, quando (por exemplo) um comando só pode ser executado se o anterior tiver ocorrido com sucesso.

7.2 Compilando o programa

7.2.1 Linux

A maioria das distribuições linux já possuem compilador C na instalação padrão. Para compilar o programa

acima(ola.c) abra um terminal, entre na pasta onde o arquivo se localiza e digite o seguinte comando:

gcc -o ola ola.c

O compilador irá gerar o arquivo executável chamado **ola** que pode ser executado da seguinte forma:

./ola

[1] Esse comando é uma *diretiva do pré-processador*; você aprenderá mais sobre esses comandos na seção Pré-processador.

Capítulo 8

Conceitos básicos

Você já viu um programa básico em C. Antes de começar a se dedicar ao estudo de C, é bom que você compreenda alguns termos e alguns aspectos da linguagem, o que facilitará sua compreensão dos capítulos seguintes. A seguir, formalizaremos alguns aspectos da estrutura básica da linguagem.

8.1 Estrutura básica

- Um programa em C é basicamente estruturado em **blocos** de código. Blocos nada mais são que conjuntos de instruções, e devem ser delimitados com chaves ({ ... }). Um bloco também pode conter outros blocos.

- Uma **instrução** geralmente corresponde a uma ação executada, e deve sempre terminar com ponto-e-vírgula (;).

- O compilador ignora espaços, tabulações e quebras de linha no meio do código; esses caracteres são chamados genericamente de **espaço em branco** (*whitespace*). Ou seja, os três trechos a seguir são equivalentes:

```
printf(“Olá mundo”);return 0;
printf (“Olá mundo”); return 0;
printf( “Olá mundo” ); return 0 ;
```

No entanto, você achará muito mais fácil de ler um estilo de código mais parecido com o segundo exemplo. Costuma-se usar (mas não abusar de) espaços e tabulações para organizar o código. Tal prática é chamada de **indentação do código**. Trata-se de uma convenção de escrita de códigos fonte que visa modificar a estética do programa para auxiliar a sua leitura e interpretação. Ela tem como objetivo indicar a hierarquia dos elementos.

- A linguagem é sensível à utilização de maiúsculas e minúsculas. Por exemplo, se você escrevesse Printf no lugar de printf, ocorreria um erro, pois o nome da função é totalmente em minúsculas.

8.1.1 Escopo

Geralmente, em programação, não queremos que outras funções usem as variáveis que estamos manipulando no momento. O conceito de **escopo** está justamente relacionado a isso. **Escopo** é o nível em que um dado pode ser acessado; em C há dois níveis: **local** e **global**. Uma variável *global* pode ser acessada por qualquer parte do programa; variáveis *locais* podem ser acessadas apenas dentro do bloco onde foram declaradas (ou nos seus subblocos), mas não fora dele (ou nos blocos que o contêm). Isso possibilita que você declare várias variáveis com o mesmo nome mas em blocos diferentes. Veja um exemplo:

```
int a; { int a; int b; } { int b; }
```

As duas variáveis chamadas b são diferentes e só podem ser acessadas dentro do próprio bloco. A primeira variável a é global, mas só pode ser acessada no segundo bloco, pois a variável local a no primeiro bloco oculta a variável global de mesmo nome. Note que isso é possível em C, e tome cuidado para não cometer erros por causa disso.

8.1.2 Bibliotecas

Uma biblioteca é um arquivo contendo um conjunto de funções (pedaços de código) já implementados e que podem ser utilizados pelo programador em seu programa. O comando **#include** é utilizado para declarar as bibliotecas que serão utilizadas pelo programa. Esse comando diz ao pré-processador para tratar o conteúdo de um arquivo especificado como se o seu conteúdo houvesse sido digitado no programa no ponto em que o comando **#include** aparece.

O comando **#include** permite duas sintaxes:

- **#include <nome_da_biblioteca>**: o pré-processador procurará pela biblioteca nos caminhos de procura pré-especificados do compilador. Usa-se essa sintaxe quando estamos incluindo uma biblioteca que é própria do sistema, como as bibliotecas **stdio.h** e **stdlib.h**;

- **#include “nome_da_biblioteca”**: o pré-

processador procurará pela biblioteca no mesmo diretório onde se encontra o nosso programa. Podemos ainda optar por informar o nome do arquivo com o caminho completo, ou seja, em qual diretório ele se encontra e como chegar até lá.

De modo geral, os arquivos de bibliotecas na linguagem C são terminados com a extensão **.h**. Veja dois exemplos do uso do comando **#include**:

#include <stdio.h> #include “D:\Programas\soma.h”

Na primeira linha, o comando **#include** é utilizado para adicionar uma biblioteca do sistema: **stdio.h** (que contém as funções de leitura do teclado e escrita em tela). Já na segunda linha, o comando é utilizado para adicionar uma biblioteca de nome **soma.h**, localizada no diretório “D:\Programas\".

8.2 Introdução às funções

Funções são muito usadas, não só em C, mas em linguagens de programação em geral. Uma função é basicamente um bloco de código que realiza uma certa tarefa. Quando queremos realizar aquela tarefa, simplesmente fazemos uma *chamada de função* para a função correspondente.

Uma função pode precisar que o programador dê certos dados para realizar a tarefa; esses dados são chamados **argumentos**. A função também pode retornar um valor, que pode indicar se a tarefa foi realizada com sucesso, por exemplo; esse valor é o **valor de retorno**. Podemos fazer uma analogia com as funções matemáticas: as variáveis independentes são os argumentos e o valor numérico da função é o valor de retorno.

Em C, para chamar uma função, devemos escrever o seu nome, seguido da lista de argumentos (separados por vírgula) entre parênteses, mesmo que não haja nenhum argumento. Lembre que a chamada de função também é uma instrução, portanto devemos escrever o ponto-e-vírgula no final. Alguns exemplos de chamadas de funções:

funcao(arg1, arg2, arg3); funcao();

Se quisermos saber o valor de retorno de uma função, podemos armazená-lo numa variável. Variáveis serão introduzidas logo adiante, mas a sintaxe é muito fácil de aprender:

valor_de_retorno = funcao(arg1, arg2);

Vejamos um exemplo completo:

//quadrado.c //calcula o quadrado de um número #include<stdio.h> int square(int num1) { return num1 * num1; } int main(){ int number; int result; printf("\nDigite um numero: "); scanf("%d”, &number); result = square(number); printf(“O Quadrado de %d eh: %d”, number, result); return 0; }

Em C, todo o código (exceto as declarações de variáveis e funções) deve estar dentro de funções. Todo programa deve ter pelo menos uma função, a função **main**, que é por onde começa a execução do programa.

8.3 Expressões

Um conceito muito importante em programação é o de *expressão*. Expressões são conjuntos de valores, variáveis, operadores e chamadas de funções que são *avaliados* ou *interpretados* para resultar num certo valor, que é chamado o valor da expressão. Por exemplo:

- 3 * 4 + 9 é uma expressão de valor 21;
- a + 3 * b é uma expressão equivalente à expressão matemática $a + 3b$;
- foo() é uma expressão cujo valor é o valor de retorno da função *foo*.

8.4 Comentários

Muitas vezes é bastante útil colocar comentários no código, por exemplo para esclarecer o que uma função faz, ou qual a utilidade de um argumento, etc. A maioria das linguagens de programação permite comentários; em C, eles podem aparecer de duas maneiras:

/* Comentários que podem ocupar várias linhas. */

e

// Comentários de uma linha só, que englobam // tudo desde as duas barras até o final da linha.

Tudo que estiver entre as marcas /* e */ ou entre // será ignorado pelo compilador. Note que os comentários de uma linha só (iniciados por //) foram incorporados ao padrão da linguagem apenas em 1999, e portanto alguns compiladores podem não os suportar. As versões mais recentes do GCC não terão problema em suportar esse tipo de comentário.

Capítulo 9

Variáveis

9.1 Variáveis

Em um programa, existe a necessidade de se guardar valores na memória, e isso é feito através de **variáveis**, que podem ser definidas simplificadamente como nomes que se referem a lugares na memória onde são guardados valores. Ao declararmos uma variável, não apenas estamos reservando um espaço de memória, como também estamos associando um nome a ele, o **identificador**. Ao invés de utilizarmos o endereço da variável na memória, que seria geralmente notado na forma hexadecimal, como por exemplo 0x0012FED4, referimo-nos ao endereço apenas pelo seu nome. Apenas para deixar claro, a própria notação em hexadecimal já é uma simplificação, pois computadores na verdade trabalham com binário.

Em C, para utilizar uma variável, ela deve ser primeiramente **declarada**, ou seja, devemos requisitar o espaço necessário para essa variável. Após reservar um espaço na memória, o computador irá associar a ele o nome da variável. Se você não declarar uma variável e tentar utilizá-la, o compilador irá avisá-lo disso e não continuará a compilação.

9.2 Declarando variáveis

Genericamente, para declarar uma variável, usamos a seguinte instrução:

tipo_da_variável nome_da_variável;

Por exemplo, para declarar uma variável do tipo int com o nome a, podemos escrever

int a;

É sempre necessário indicar o tipo da variável, pois cada tipo tem um tamanho diferente, ou seja, ocupa mais ou menos espaço na memória do computador. Mais adiante introduziremos os tipos de variável.

9.3 Atribuindo valores

Se quisermos associar um valor a uma variável, usamos o operador = (igual):

a = 5;

Nesse caso, estamos pedindo que o computador guarde o valor 5 no espaço alocado à variável a.

Observação: Apesar de este operador se assemelhar ao igual da matemática, sua função é diferente. Para verificar a igualdade de dois valores, usamos o operador de comparação "==" (dois iguais).

É possível também atribuir um valor a uma variável ao mesmo tempo que ela é declarada, o que é chamado de *inicializar* a variável. Por exemplo:

int a = 5;

É possível também declarar mais de uma variável de um mesmo tipo em uma única instrução, separando os nomes por vírgulas. Também é possível inicializar as variáveis dessa maneira:

int a, b, c, d; int e = 5, f = 6; int g, h = 2, i = 7, j;

Como o próprio nome já diz, o valor existente numa variável pode ser mudado, da mesma maneira que ele é normalmente atribuído. Se tivermos:

int a; a = 2; a = 3;

no final o valor da variável *a* será 3.

9.4 Exemplo de erro

a = 25;

Mesmo sabendo que é um exemplo de erro, escreva o código acima em um arquivo .c e tente compilar para se familiarizar com as mensagens de erro do compilador, assim você saberá o que fazer quando elas ocorrerem.

No exemplo acima não foi declarada a variável *a*, ao tentar compilar o compilador informa que o símbolo *a* não

foi definido.

9.5 Nomes de variáveis

Existem algumas restrições quanto ao nome que podemos dar a variáveis. Essas regras se aplicam também para nomear funções e estruturas.

- Os nomes de variáveis devem ser únicos no mesmo escopo: não podemos ter duas variáveis com o mesmo nome.
- O nome pode ser igual ao de outra variável já existente em escopo superior, porém é recomendado fortemente que não se use variáveis iguais sob pena de tornar o código do programa incompreensível ou de difícil análise;
- O C, assim como muitas outras linguagens de programação, é sensível à utilização de maiúsculas e minúsculas(*case sensitive*). Portanto, o código a seguir seria válido e geraria três variáveis diferentes:

int nome; int NOME; int Nome;

- Em nomes de variáveis, podemos usar letras maiúsculas ou minúsculas (de A a Z, sem acentos), algarismos arábicos (0-9) e o caractere sublinhado (_), mas o primeiro caractere deve ser uma letra ou o sublinhado.
- Algumas palavras não podem ser usadas para nomes de variáveis por serem palavras reservadas (palavras que têm significado especial na linguagem).
- O padrão C atual especifica que nomes de até 31 caracteres devem ser aceitos. Alguns compiladores podem até aceitar nomes maiores que isso, mas não considere isso uma regra e não use nomes tão longos.

Capítulo 10

Tipos de dados

'

Até agora você só viu as variáveis do tipo int, que servem para guardar números inteiros. A linguagem C tem outros tipos fundamentais. São eles:

- int, para números inteiros entre -2147483648 e 2147483647, utiliza 4 bytes;
- char, para caracteres individuais do padrão ASCII, utiliza 1 byte;
- float, para reais entre (aproximadamente) 10^{-38} e 10^{38}, utiliza 4 bytes, precisão de 7 dígitos;
- double, para reais entre (aproximadamente) 10^{-4932} e 10^{4932}, utiliza 8 bytes, precisão de 15 dígitos;
- bool, para indicar true (verdadeiro) ou false (falso), utiliza 1 byte; Presente apenas no padrão C99 em diante.

10.0.1 Explicando bits e bytes

Podemos pensar na memória do computador como uma fita, uma grande fita feita de frames sequenciais.

Em cada um desses frames, podemos colocar uma certa voltagem: tem voltagem ou não tem voltagem: se tem voltagem associamos o valor 1, se não tem voltagem associamos o valor 0. Daí termos a linguagem binária de zeros e uns.

Agora podemos fazer combinações se tivermos posição de zeros e uns, da direita para a esquerda.

- 00000000 1ª Combinação
- 00000001 2ª Combinação
- 00000010 3ª Combinação
- 00000011 4ª Combinação
- 00000100 5ª Combinação
- 00000101 6ª Combinação
- 00000110 7ª Combinação
- 00000111 8ª Combinação
- …

E na verdade podemos estender este conceito para um número infinito de combinações.

Ora o que aconteceu é que nos bastavam pouco menos de 256 combinações (8 bits ordenados) para termos uma combinação para cada letra, maiúscula e minúscula, número, pontos de exclamação, interrogação, etc. …e isso era o suficiente para a nossa comunicação. Mas para haver um certo consenso para que uma dada combinação desse um dado símbolo surgiu a tabela ASCII (surgiram outras tabelas quando se quis colocar os símbolos de outras línguas, como o japonês ou o chinês – ver tabela ISO) Portanto com 8 bits ou 8 casas conseguíamos ter qualquer símbolo que utilizamos. A esse conjunto de 8 bits chamamos de byte, mais convenientemente. Portanto, um byte tem 8 casas de zeros /uns , ou seja 2 elevado a 8 dá as 256 combinações. E o byte é a unidade básica que o C++ consegue operar e é representado pelo tipo char.

Pergunta: Quando tivermos mais do que 256 bytes acrescenta-se um outro byte?

- Sim. Com dois bytes o número de combinações é 256*256.

Pergunta: Qual a razão do computador usar apenas bytes como medida mínima? Será que não seria possível utilizar 7 bits ou 5 bits?

- Não é possível pelo fato do computador só entender 0 e 1 então no caso é impossível se ter um número ímpar de bits porque tudo tem que ter o 0 e o 1 por isso que tudo na informática evolui multiplicando-se por 2 (32, 64, 256, 512)

10.0.2 Números inteiros

Se dissermos que 2 bytes representam inteiros, poderemos utilizar as 65 536 combinações, pois 2 bytes -16bits- temos 2 elevado a 16 = 65 536 e isso dar-nos-ia esses números todos. Assim se quisermos apenas os positivos com o zero temos de [0, 65535].

Se quisermos ter números negativos e positivos podemos dividir esse valor a meio e dá 32768 para cada lado positivo e negativo, mas como temos de ter o zero vamos roubar um valor ao lado positivo e então ficamos com o intervalo $[-32768, 32767]$. E ficamos com as mesmas 65 536 combinações.

Apresentamos inteiro com 2 bytes, mas eles podem ter 4 bytes, isso vai depender do processador do computador, ie, com quantos bytes consegue ele lidar ao mesmo tempo.

Também existem outros tipos, como short (ou short int), que serve para inteiros menores, long (ou long int) para inteiros maiores. Qualquer tipo inteiro pode ser precedido por unsigned (o signed para COM negativos), para cortar os números negativos, permitindo maior capacidade de armazenamento de números positivos. Alguns compiladores aceitam o long long, para aumentar ainda mais o tamanho da variável, alguns desses só aceitam para o tipo int, outros também para o tipo double.

Podemos alterar a maneira como os dados são guardados com os **modificadores de tipo**. Você pode modificar os tipos de duas maneiras.

Tamanho: short e long

Você pode modificar o tamanho de uma variável usando os **modificadores de tipo**, que são dois: short e long. Note que float e char não podem ser modificados em tamanho.

- **short** diminui o espaço necessário para guardar a variável (diminuindo também a gama de valores que esta pode assumir). Só pode ser usado com int.
- **long** aumenta o espaço tomado pela variável, e portanto aumenta seu valor máximo e/ou sua precisão. Pode ser usado com int e double.
- O padrão C de 1999 adicionou um terceiro modificador, suportado pelos compiladores mais recentes, inclusive o gcc: long long, que aumentaria ainda mais a capacidade da variável. Alguns deles suportam esse modificador apenas para o tipo int, e outros suportam também para double.

Uma observação é necessária: segundo o padrão, não existe nenhuma garantia de que uma variável short int é menor que uma variável int, nem que long int é maior que int. Apenas é garantido que int não é maior que long nem menor que short. De fato, nos sistemas x86 de 32 bits (ou seja, a maioria dos computadores pessoais atualmente), o tamanho de int é igual ao de long. Geralmente, int será o tamanho nativo do processador — ou seja, 32 bits num processador de 32 bits, 16 bits num processador de 16 bits etc.

Sinal: signed e unsigned

Existe outro tipo de modificador, que define se o número vai ser guardado com **sinal** ou não. São os modificadores signed e unsigned, suportados pelos tipos inteiros apenas.

- **signed** diz que o número deve ser guardado com sinal, ou seja, serão permitidos valores positivos e negativos. Esse é o padrão, portanto esse modificador não é muito usado.
- **unsigned** diz que o número deve ser guardado **sem sinal**. Com isso, o valor máximo da variável aumenta, já que não teremos mais valores negativos. Por exemplo, com uma variável char podemos guardar valores de -128 a 127, mas com uma variável unsigned char pode guardar valores de 0 a 255.

Para usar esses modificadores, devemos colocá-los **antes** do nome do tipo da variável, sendo que o modificador de sinal deve vir antes do modificador de tamanho caso ambos sejam usados. Por exemplo:

unsigned char c; **short int** valor; **unsigned long int** resultado;
Nota: Você pode abreviar short int e long int para simplesmente short e long, respectivamente.

Tabela de tipos inteiros

Convém ver a tabela de tipos inteiros.

Tipo Num de bits Formato para leitura com scanf Intervalo Inicio Fim char 8 %c -128 127 unsigned char 8 %c 0 255 signed char 8 %c -128 127 int 16 %i -32.768 32.767 unsigned int 16 %u 0 65.535 signed int 16 %i -32.768 32.767 short int 16 %hi -32.768 32.767 unsigned short int 16 %hu 0 65.535 signed short int 16 %hi -32.768 32.767 long int 32 %li $-2.147.483.648$ 2.147.483.647 signed long int 32 %li $-2.147.483.648$ 2.147.483.647 unsigned long int 32 %lu 0 4.294.967.295

Nota: O tipo **long** é 32 bits como int em computadores de arquitetura 32 bits e 64 bits em computadores de arquitetura 64 bits no padrão LP64 (Mac OS X e Unix).

10.0.3 Números de ponto flutuante

Os números de ponto flutuante são uma tentativa para guardar números reais, como 3,1415 (pi), $-2,3333$, 0,00015, $6{,}02 \times 10^{23}$. Ao contrário dos números reais, os números representáveis pelo hardware são finitos. A maneira como os tipos de ponto flutuante são armazenados é abstrata para o programador, entretanto, o hardware segue o padrão IEEE 754 (Standard for Floating-Point Arithmetic).

O armazenamento é feito usando notação científica binária.

Os tipos float e double servem para guardar números de ponto flutuante. A diferença entre os dois é, além do intervalo de dados, a precisão. Geralmente, o tipo float guarda dados (com sinal positivo ou negativo) de 3,4E-38 a 3,4E+38 (além do zero). Já double suporta números tão pequenos quanto 1,7E-308 e no máximo 1,7E+308.

float 32 %f 3,4E-38 3.4E+38 double 64 %lf 1,7E-308 1,7E+308 long double 80/128 %Lf 3,4E-4932 3,4E+4932

Nota: O tipo **long double** trabalha em máquinas x64 no padrão LP64 (Mac OS X e Unix)

10.0.4 Bool

Este tipo surgiu porque muitas vezes apenas se quer ter 2 valores: sim/não ; verdadeiro/falso. Tem o tamanho de um byte e tem apenas dois valores 0 e 1 que corresponde a true e false.

Por que guardar um bool num byte quando se pode utilizar apenas um bit? A razão é que o computador usa no mínimo o byte, não o bit.

10.0.5 Endereços

Os vários locais na memória são identificados por um address, que tem uma lógica sequencial numerada. São necessários 16 bits para guardar o endereço de um byte. dito de outra forma são necessários 2 bytes para guardar a morada de um byte. será isto verdade?!! isso quer dizer que se guardarmos os endereços de todos os bytes, só temos 1/3 da memória disponível para guardar valores. Bem isto é um pouco estranho, mas repare-se que apenas vamos guardar os addresses das variáveis reservadas. Depois as variáveis nem sempre são de 1 byte, por isso apenas iremos guardar o endereço do primeiro byte e não de todos. por fim faz sentido guardar o endereço de outro endereço? Os endereços de memória (addresses) são normalmente expressos em linguagem hexadecimal (base 16, utilizam os 10 algarismos mais as 6 primeiras letras – de a a f - do alfabeto para fazerem as 16).

10.0.6 Compatibilidade de dados na atribuição de valor

Se tentarmos colocar um valor diferente do tipo esperado da variável? Temos um problema de compatibilidade de dados:

- **Caso 1**: Declaramos um int e colocamos uma letra

  Aqui não teremos problemas. Os literais de caracteres são, nativamente, do tipo *int*. O resultado será um inteiro que contém o valor ASCII do caractere dado.

- **Caso 2**: Declaramos um int e colocamos uma string (sequência de caracteres)

  Aqui teremos um erro de compilação, em que nos diz que não conseguimos converter “const char [5]" em “int”. Perceba com isso que o compilador tem alguns sistemas de conversão — note o caso 3.

- **Caso 3**: Declaramos um int e colocamos um float

  Neste caso, se colocarmos 77.33, irá ser apenas guardado o valor 77, perdendo-se a parte decimal.

- **Caso 4**: *overflow* — declaramos um *short* e colocamos um valor maior que o máximo

  Lembre-se que o tipo *short* guarda valores de –32767 a 32767. Se colocarmos 32768 (e o compilador não estender esses limites), *não* vai acontecer nenhum erro de compilação; o que resulta é que vai ser impresso um número negativo, –32767 (ou, como é comum em vários compiladores, –32768). A lógica disto tem a ver com a maneira como o computador guarda números negativos. Mas também podemos fazer uma analogia com as horas. Imaginemos que vamos somar 6 horas com 7 horas. O resultado seria 13, mas como não existe 13 no relógio, iríamos dar a volta nas horas e chegar ao 1. Assim o resultado será 1.

- **Caso 5**: *underflow* — declaramos um *short* e colocamos um valor inferior ao mínimo possível.

  Aqui temos exatamente a mesma lógica do caso de overflow, mas desta vez é excedido o limite inferior e não o superior.

- **Caso 6:** declaramos um unsigned int e colocamos um número negativo

  O que acontece aqui é semelhante a um underflow. Mas o que ocorre é que o número é guardado como seria se fosse um int comum, negativo. O que muda na prática é a interpretação desse número, de acordo com o tipo de dado que lhe está atribuído. Se tentarmos lê-lo como um unsigned int, obteremos um valor positivo obtido pela mesma lógica do overflow/underflow; se o lermos como um (signed) int, obteremos o mesmo valor negativo que lhe atribuímos.

10.0.7 Converter um tipo de variável

A conversão de uma variável consiste em converter o tipo de uma variável em um outro. Imagine que você esteja trabalhando com uma variável do tipo float e por alguma razão queira eliminar os números que estão depois da vírgula.

Esta operação pode ser realizada de duas maneiras.
Conversões do tipo implícita: Consiste em uma modificação do tipo de variável que é feita automaticamente pelo compilador.

Ex:

int x; x = 7.123;

Conversões do tipo explícita: Também chamada de operação **cast**, consiste em forçar a modificação do tipo de variável usando o operador cast "( )".

Ex:

int y; y = (int)7.123;

Veja um exemplo da conversão de tipo inteiro em caracteres. Aqui convertemos um numero decimal em um caractere ASCII.

#include <stdio.h> int main() { int y = 65; char x; x = (char) y; printf("O numero inteiro: %d \n O caractere: %c \n\n", y, x); }

10.0.8 Literais

Em programação, um **literal** é uma notação que representa um valor constante. Exemplos de literais em C são 415, 19.52, 'C', "João". Esses exemplos representam os quatro tipos de literais em C: literais de inteiros, literais de reais, literais de caracteres e literais de strings. Só com esses exemplos já é possível deduzir como se usam os literais; mas é importante fazer algumas observações:

- Literais de inteiros podem ser especificados nas bases decimal, octal ou hexadecimal. Se o literal for prefixado com "0x" ou "0X", ele será interpretado como hexadecimal; se o prefixo for apenas "0", será interpretado como octal; ou se não houver prefixo, será interpretado como decimal.
- Literais de reais podem ser especificados na forma decimal (144.57) ou em notação científica (1.4457e+2). Lembre-se que o separador decimal é o ponto e não a vírgula, como seria usual.
- Literais de caracteres devem vir entre aspas simples (') e conter a representação de apenas um caractere[1]. Usos válidos seriam: 'c', '\n', '\x1b', '\033'. Se você quiser usar a aspa simples como caractere, precede-a com uma barra invertida: '\''.
- Literais de strings devem vir entre aspas duplas ("). Para usar aspas duplas dentro de strings, preceda-as com barra invertida: "Ele disse \"Olá\".". Note que um literal de string adiciona o caractere nulo (\0) ao final da string, pois ele é, em C, a maneira de delimitar o final de uma string.

Na verdade, segundo o padrão C, literais de caracteres podem conter a representação de mais um caractere, mas o uso deles seria para representar números e não sequências de caracteres; é um aspecto pouco utilizado da linguagem C.

Capítulo 11

Constantes

11.1 Constantes

Em um capítulo anterior abordamos as variáveis e agora vamos abordar constantes. A razão é que as coisas estão mais maduras e uma pessoa sabe muito bem o que é uma constante. Que é simplesmente um valor que não se altera durante a execução do programa. A questão de não se alterar durante a escrita do programa é realmente a razão deste capítulo. Devemos separar as águas entre uma constante e um literal. O literal é o próprio valor.

Existem 3 maneiras para criar constantes:

1. #define
2. [const] [tipo da variável][nome da variável]
3. enumerations

Esta última vamos deixar para mais tarde, pois são uma boa introdução para as classes.

11.2 DEFINED CONSTANTS (#DEFINE)

#define PI 3.14159265 #define NEWLINE "\n”

Se colocarmos estas linhas no header, o que vai acontecer é o seguinte: O pré-processador irá verificar o nosso código fonte e sempre que encontrar a diretiva #define irá, literalmente, substituir cada ocorrência do identificador no código fonte pelo valor definido.

A vantagem disto é que:

- Podemos ter um identificador ao nosso gosto, e sempre que necessitarmos do valor escrevemos o identificador, em vez do valor, até porque se o valor fosse complicado poderíamos enganar-nos a escrever. Claro que nos poderíamos enganar também a escrever o identificador, daí a escolhermos um nome familiar.
- E se necessitarmos de alterar o valor, alteramos apenas 1 vez, em vez de todas as vezes onde apareceria o valor.

O formato geral é:

#define identificador valor

Repare que a diretiva de preprocessador não tem o “;”- “ponto e vírgula” no fim! O que é normal para diretivas de Preprocessador.

O que é que acontece se tivermos o “;” no fim? Será que encontrei um bug? se eu colocar o ; no #define NEWLINE '\n'; não acontece nada.

Vale lembrar que cada #define é preprocessador, ou seja, não pode ser alterado dentro do programa durante sua execução.

// defined constants: calculate circumference #include <stdio.h> #define PI 3.14159 #define NEWLINE "\n” int main (){ double r=5.0; // radius double circle; circle = 2 * PI * r; // utilizamos o Pi e não 3. printf("%f”, circle); printf("%s”, NEWLINE); return 0; }

11.3 Declared constants (const)

Nós podemos transformar uma variável numa constante do gênero:

- const int tamanho = 100;
- const char tabul = '\t';
- const int codigo = 12440;

Com o prefixo “const”, dizemos que a variável não poderá alterar o seu valor.

Repare que se fosse uma variável eu poderia ter:

int a=5;

e logo a seguir dizer

a=6;

e o valor do a ficava com o valor de 6;

Agora com o prefixo *const* eu não poderei alterar o valor, porque constante é constante, não pode mudar o valor.

Esta maneira é bem melhor do que a anterior para declarar constantes, primeiro porque aqui temos a informação do tipo de variável, em segundo porque podemos fazer este const int a; como uma função local e não global.

Capítulo 12

Entrada e saída simples

12.1 Entrada e saída simples

Se você pensar bem, perceberá que um computador é praticamente inútil se não tiver nenhuma maneira de interagir com o usuário. Por exemplo, se você abrir um processador de texto, nada irá acontecer até que você abra um arquivo ou digite algum texto no teclado. Mas, da mesma maneira, é necessário que o computador forneça informação também: como você poderia saber se uma tarefa foi concluída?

As trocas de informação entre o computador e o usuário são chamadas **entrada** e **saída** (*input* e *output*, em inglês). *Entrada* é a informação fornecida a um programa; *saída* é a informação fornecida pelo programa. É comum referir-se aos dois termos simultaneamente: *entrada/saída* ou E/S (I/O, em inglês).

Frequentemente são usados os termos "saída padrão" (standard output, stdout) e "entrada padrão" (standard input, stdin). Eles se referem, na maioria das vezes, ao monitor e ao teclado, que são os meios básicos de interação com o usuário. No entanto, os sistemas operacionais permitem redirecionar a saída e a entrada de programas para outros dispositivos ou arquivos.

As funções de entrada e saída na linguagem C trabalham com **fluxos** (*streams*, em inglês) de dados, que são uma forma de abstração de dados de maneira sequencial. Assim, toda entrada e saída é feita da mesma maneira, com as mesmas funções, não importando o dispositivo com o qual estamos nos comunicando (teclado, terminal, arquivo, etc.). As **mesmas funções que descrevem o acesso aos arquivos podem ser utilizadas para se acessar um terminal de vídeo**.

Em C, as funções da biblioteca padrão para entrada e saída estão declaradas no cabeçalho **stdio.h**. Uma delas já foi introduzida em seções anteriores: **printf()**. A seguir daremos mais detalhes sobre essa função e introduziremos outras.

12.2 puts() e putchar()

puts significa "put string" (colocar string), utilizado para "colocar" uma string na saída de dados. **putchar** significa "put char" (colocar caractere), utilizado para "colocar" um caractere na saída de dados.

São as funções mais simples do cabeçalho *stdio.h*. Ambas enviam (ou "imprimem") à saída padrão os caracteres fornecidos a elas; putchar() manda apenas um caractere, e puts() manda uma sequência de caracteres (ou *string*). Exemplo:

puts ("Esta é uma demonstração da função puts."); putchar ('Z');

Note que junto com a função puts devemos usar literais de string (com aspas duplas), e com putchar devemos usar literais de caractere (com aspas simples). Se você tentasse compilar algo como putchar ("T"), o compilador daria uma mensagem de erro. Lembre-se que "T" é diferente de 'T'.

Podemos também colocar caracteres especiais, como a tabulação (\t) e a quebra de linha (\n):

puts ("Primeira linha\nSegunda linha\te um grande espaço"); putchar ('\n'); *// apenas envia uma quebra de linha*

Este código resultaria em algo parecido com:

Primeira linha Segunda linha e um grande espaço

Observe que a função puts() sempre coloca uma quebra de linha após imprimir a string. Já com as funções putchar() e printf() (vista a seguir), isso não ocorre. O código abaixo, por exemplo:

putchar('c'); putchar('h'); putchar('\n'); puts("String."); puts("Outra string.");

imprimiria algo parecido com isto:

ch String. Outra string.

Observe que os caracteres 'c' e 'h' são exibidos na mesma linha, pois não foi inserida uma quebra de linha entre eles. Já as strings "String." e "Outra string." são exibidas em linhas diferentes, pois a função puts() insere uma quebra de linha após cada string, mesmo que não haja um caractere

'\n' nas literais de string do código.

Os outros caracteres especiais são introduzidos adiante.

Note que o argumento **deve** ser uma sequência de caracteres. Se você tentar, por exemplo, imprimir o número 42 desta maneira:

puts(42);

Na verdade o que o compilador tentará fazer é imprimir a sequência de caracteres que começa na posição 42 da memória (provavelmente ele irá alertá-lo sobre isso se você tentar compilar esse código). Se você tentar executar esse código, provavelmente ocorrerá uma falha de segmentação (erro que ocorre quando um programa tenta acessar memória que não lhe pertence). A maneira correta de imprimir o número 42 seria colocá-lo entre aspas duplas:

puts(“42”);

12.3 printf()

printf vem de “print formatted” (imprimir formatado).

À primeira vista, a função printf() pode parecer idêntica à puts(). No entanto, ela é muito mais poderosa. Ela permite facilmente imprimir valores que não são sequências de caracteres, além de poder formatar os dados e juntar várias sequências de caracteres. Por isso, a função printf() é muito mais usada que a puts().

Ela pode ser usada exatamente como a função puts(), se fornecermos a ela apenas uma sequência de caracteres:

printf(“Este é um programa em C”);

Ela também pode ser escrita da seguinte forma:

printf(“Ola” " mundo” "!!!");

Mas e se precisarmos imprimir o conteúdo de uma variável? A função printf também pode fazer isso! Você deve, obviamente, especificar **onde** o valor da variável deve ser impresso. Isso é feito através da especificação de formato %d, caso a variável seja do tipo *int* (sequências para outros tipos serão dadas adiante). Você também precisará, logicamente, especificar **qual** variável imprimir. Isso é feito dando-se mais um argumento à função printf(). O código deverá ficar assim:

int teste; teste = 42; printf (“A variável 'teste' contém o número %d.”, teste);

Tendo colocado isso no seu programa, você deverá ver na tela:

A variável 'teste' contém o número 42.

Vamos supor que você queira imprimir um número não inteiro. Você teria que trocar "%d” por %f. Exemplo:

float pi; pi = 3.1415; printf (“O valor de pi é %f.”, pi);

O código acima irá retornar:

O valor de pi é 3.1415.

Você pode imprimir quantos valores quiser, bastando para isso colocar mais argumentos e mais especificações de formato, lembrando de colocar na ordem certa. Alguns compiladores, como o gcc, mostram um aviso caso o número de argumentos seja diferente do número de especificações de formato, o que provavelmente causaria resultados indesejados. A sintaxe geral da função printf() é:

printf (“string de formatação”, arg1, arg2, ...);

Suponha que você tem um programa que soma dois valores. Para mostrar o resultado da conta, você poderia fazer isso:

int a, b, c; ... *//leitura dos dados* c = a + b; ***//c** é o resultado da soma* printf ("%d + %d = %d”, a, b, c);

O que resultaria em, para a = 5 e b = 9:

5 + 9 = 14

A seguir mostramos os especificadores de formato para vários tipos de dados.

12.3.1 Especificações de formato

A documentação mais técnica os chama de “especificadores de conversão”, pois o que ocorre na maioria das vezes é, de fato, a conversão de um valor numérico em uma sequência de caracteres que representa aquele valor. Mas o nome “formato” não deixa de estar correto, pois eles especificam em que *formato* (inteiro, real etc.) está o argumento correspondente.

Observação Se você quiser imprimir um sinal de porcentagem, use %%. Exemplo: printf(“O lucro para o último mês foi de 20%%.”);

Numa sequência de controle, é possível também indicar a largura do campo, o número de casas decimais, o tamanho da variável e algumas opções adicionais. O formato geral é:

%[opções][largura do campo][.precisão][tamanho da variável]tipo de dado

A única parte obrigatória é o tipo de dado. Todas as outras podem ser omitidas.

Opções

As opções são parâmetros opcionais que alteram a formatação. Você pode especificar zero ou mais delas, colocando-as logo após o sinal de porcentagem:

- 0: o tamanho do campo deve ser preenchido com zeros à esquerda quando necessário, se o parâmetro correspondente for numérico.

- - (hífen): o valor resultante deve ser alinhado à esquerda dentro do campo (o padrão é alinhar à direita).
- (espaço): no caso de formatos que admitem sinal negativo e positivo, deixa um espaço em branco à esquerda de números positivos.
- +: o sinal do número será sempre mostrado, mesmo que seja positivo.
- ' (aspa simples/apóstrofe): números decimais devem ser exibidos com separador de milhares caso as configurações regionais o especifiquem. Essa opção normalmente só funciona nos sistemas Unix.

Largura do campo

Como o próprio nome já diz, especifica qual a largura mínima do campo. Se o valor não ocupar toda a largura do campo, este será preenchido com espaços ou zeros. Por exemplo, podemos imprimir um código de até 5 dígitos preenchido com zeros, de maneira que os valores 1, 27, 409 e 55192 apareçam como 00001, 00027, 00409 e 55192.

A largura deve ser especificada logo após as opções, se presentes, e pode ser um número — que especifica a largura — ou um asterisco, que diz que a largura será especificada pelo próximo argumento (ou seja, o argumento anterior ao valor a ser impresso). Neste exemplo, o campo terá largura igual ao valor de num e o valor impresso será 300:

printf ("%*d", num, 300);

O campo é impresso de acordo com as seguintes regras:

- Se o valor for mais largo que o campo, este será expandido para poder conter o valor. O valor nunca será cortado.
- Se o valor for menor que o campo, a largura do campo será preenchida com espaços ou zeros. Os zeros são especificados pela opção 0, que precede a largura.
- O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se a opção - (hífen ou sinal de menos) antes da largura do campo.

Por exemplo, compare as três maneiras de exibir o número 15:

printf ("%5d", 15); *// exibe " 15"* printf ("%05d", 15); *// exibe "00015"* printf ("%−5d", 15); *// exibe "15 "*

E alguns outros exemplos:

printf ("%−10s", "José"); *// exibe "José "* printf ("%10s", "José"); *// exibe " José"* printf ("%4s", "José"); *// exibe "José"*

Precisão

A precisão pode ter quatro significados diferentes:

- Se a conversão solicitada for **inteira** (d, i, o, u, x, X): o número mínimo de dígitos a exibir (será preenchido com zeros se necessário).
- Se a conversão for **real** (a, A, e, E, f, F): o número de casas decimais a exibir. O valor será arredondado se a precisão especificada no formato for menor que a do argumento.
- Se a conversão for em **notação científica** (g, G): o número de algarismos significativos. O valor será arredondado se o número de algarismos significativos pedido for maior que o do argumento.
- Se a conversão for de uma **sequência de caracteres** (s): o número máximo de caracteres a exibir.

Assim como a largura do campo, a precisão pode ser especificada diretamente por um número ou com um asterisco, mas deve ser precedida por um ponto.

Alguns exemplos:

printf ("%.5d", 314); *// exibe "00314"* printf ("%.5f", 2.4); *// exibe "2.40000"* printf ("%.5g", 23456789012345); *// exibe "2.3457e+13"* printf ("%.5s", "Bom dia"); *// exibe "Bom d"*

É claro que podemos combinar a largura com a precisão. Por exemplo, %10.4f indica um campo de número real de comprimento total dez e com 4 casas decimais. Note que, na largura do campo, o valor inteiro é levado em conta, inclusive o ponto decimal, e não apenas a parte inteira. Por exemplo, essa formatação aplicada ao número 3.45 irá resultar nisto:

" 3.4500"

Tamanho da variável

É importante ressaltar que quando são usados modificadores de tamanho de tipos, a maneira como os dados são armazenados pode tornar-se diferente. Assim, devemos informar à função printf() precisamente qual o tipo da variável cujo valor desejamos exibir. A função printf() admite cinco principais modificadores de tamanho de variável:

- **hh**: indica que a conversão inteira corresponde a uma variável char. Por exemplo, poderíamos usar o formato %hhd para exibir uma variável do tipo char na base decimal.
- **h**: indica que a conversão inteira corresponde a uma variável short.
- **l**: indica que a conversão inteira corresponde a uma variável long.

- **ll**: indica que a conversão inteira corresponde a uma variável long long.
- **L**: indica que a conversão de número real corresponde a uma variável long double.

Quando o tipo da variável não tem modificadores de tamanho (long ou short), não se usa nenhum modificador de tamanho da variável na função printf().

12.3.2 Sequências de escape

Sequências de escape são combinações de caracteres que têm significado especial, e são sempre iniciadas por uma barra invertida (\). Você pode usá-las em qualquer literal de caractere ou string. Por exemplo, a string "linha 1\nlinha 2" equivale a:

linha 1 linha 2

pois a sequência \n indica uma quebra de linha. Como foi citado anteriormente, a função printf(), diferentemente de puts(), não imprime automaticamente uma quebra de linha no final da string. O código abaixo, por exemplo:

printf("string 1"); printf("string 2");

Imprimiria isto:

string 1string 2

Isso pode ser útil, pois às vezes é desejável permanecer na mesma linha.

A seguir apresentamos a tabela com as sequências de escape suportadas pela linguagem C:

Representação octal e hexadecimal

Também é possível trocar uma sequência de escape pelo seu valor em octal ou hexadecimal. Você pode, por exemplo, trocar o caractere "\n" pelo valor octal "\12" ou hexadecimal "\x0A". Vejamos mais alguns exemplos.

Hexadecimal Octal Caracter \x00 \00 \0 \x0A \12 \n \x0D \15 \r \x07 \07 \a \x08 \10 \b \x0B \13 \v

12.4 scanf()

A função **scanf()** lê dados da entrada padrão (teclado) e os guarda em variáveis do programa. Assim como para printf(), usamos uma string de formatação para especificar como serão lidos os dados. A sintaxe de scanf() é esta:

scanf ("string de formatação", &arg1, &arg2, ...);

Como você pode ver, a sintaxe é quase igual à de printf(), com exceção do E comercial (&). Você entenderá melhor o seu uso nas seções seguintes, mas adiantamos que ele é um operador que retorna o endereço de uma variável. Isso é necessário pois a função scanf() deve modificar as variáveis, e quando não usamos o operador de endereço, passamos apenas o valor de uma variável para a função. Isso será explicado melhor no capítulo sobre ponteiros. O fato de scanf receber endereços de variáveis (em vez de seus valores) também explica por que ele precisa ser informado da diferença entre %f (float) e %lf (double) enquanto que o printf não precisa.

Um exemplo básico da utilização de scanf() é este:

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

O que este exemplo faz é declarar uma variável e aguardar o usuário digitar algo. Os dados só serão processados quando o usuário apertar Enter. Depois disso, os caracteres digitados pelo usuário serão convertidos para um valor inteiro e esse inteiro será guardado no endereço que corresponde à variável a. Se o valor digitado não puder ser convertido (porque o usuário não digitou nenhum algarismo válido), a variável não será modificada.

Assim como na função printf(), podemos receber quantos valores quisermos, bastando usar vários especificadores de conversão:

int a; char b; float c; scanf ("%d %c %f", &a,&b,&c);

Dessa maneira, se o usuário digitar 120 z 17.63, teremos a igual a 120, b igual ao caractere 'z' e c igual ao número 17,63. Se o usuário tentar digitar mais de um espaço entre os dados ou simplesmente nenhum espaço, ainda assim o programa obterá os dados certos. Por exemplo, 120z17.63 também dará o mesmo resultado.

Agora uma questão um pouco mais difícil: vamos supor que especificamos um formato inteiro e o usuário digitou um número real, como por exemplo 12.5. O que deverá acontecer?

#include <stdio.h> int main () { int a; printf ("Digite um número: "); scanf ("%d", &a); printf ("\nO número digitado foi %d", a); return (0); }

Se você testar com o valor 12.5, vai ver que o programa retornará o número 12, pois a função scanf() apenas interpreta os caracteres válidos para aquele formato.

Os especificadores de conversão são praticamente os mesmos que os da função printf(), com algumas mudanças. A maioria deles pula espaços em branco, exceto dois.

- **%i** não é mais sinônimo de %d. O que %i faz é interpretar o valor digitado como hexadecimal, se iniciar-se por 0x ou 0X; como octal, se iniciar-se por 0; ou como decimal, caso nenhuma dessas condições seja verificada.
- **%a**, **%e/%E** e **%g** são sinônimos de %f.
- **%lf** deve ser usado para variáveis do tipo **double**.
- **%s** lê uma sequência de caracteres não-brancos (qualquer caractere exceto espaço, tabulação, quebra de linha etc.), ou seja, uma palavra.

- **%c** lê uma sequência de caracteres, **sem ignorar espaços**. O padrão é ler um caractere, se não for especificada a largura do campo.

- **%[...]** lê uma sequência de caracteres, **sem ignorar espaços**, especificando entre colchetes quais caracteres devem ser aceitos, ou, se o primeiro caractere dentro dos colchetes for um acento circunflexo (^), quais **não** devem ser aceitos. Além disso, se colocarmos um traço entre dois caracteres, todos os caracteres entre os dois serão incluídos no padrão. Por exemplo, se quisermos incluir qualquer letra minúscula, poderiámos escrever %[a-z]; se quiséssemos também incluir as maiúsculas, colocaríamos %[a-zA-Z]. A leitura pára quando for encontrado um caractere que não coincide com o padrão especificado.

Já os modificadores funcionam de maneira bastante diferente:

- O modificador * (asterisco) especifica que o valor atual deve ser lido da maneira especificada, mas não será guardado em nenhuma variável, e portanto não deve haver um ponteiro correspondente a esse valor. Por exemplo, poderiámos ter um programa que espera ler uma palavra e depois um número, mas não importa qual palavra é. Nesse caso usaríamos o modificador *: scanf ("%*s %d", &numero). O programa leria a palavra e guardaria o número na variável numero.

- Como na função printf(), existe o especificador de largura do campo, que deve aparecer antes do especificador de conversão, mas em scanf() ele especifica a **largura máxima**. Se a largura máxima foi definida como n, scanf() pulará para o próximo campo se já tiver lido n caracteres. Por exemplo, scanf ("%4d", &num) lerá um número de até quatro algarismos. Se o usuário digitar mais, o excedente será não será lido por essa chamada, *mas poderá ser lido por uma próxima chamada a scanf*.

Mais detalhes sobre os especificadores de conversão e os modificadores podem ser encontrados na documentação da biblioteca padrão.

12.4.1 Valor de retorno

A funcão scanf() retorna o número de conversões realizadas com sucesso. Isso é útil pois, se o valor contido numa variável após a chamada de scanf() for igual ao valor anterior, não é possível saber se o valor digitado foi o mesmo que já havia ou se não foi feita a conversão. Para obter esse número de conversões realizadas, você deve guardar o resultado numa variável do tipo int. Veja como proceder:

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

Este exemplo lê dois números inteiros e os guarda nas variáveis a e b. O número de conversões realizadas é guardado na variável num. Se após o scanf, num for diferente de 2, é sinal de que o usuário digitou algo incompatível com o formato desejado.

Note que aqui introduzimos um conceito novo: o **valor de retorno** de uma função. Ele pode ser obtido simplesmente associando o valor de uma variável à chamada da função. Ele será detalhado na seção Funções, mas já é possível compreender um pouco sua utilização.

12.5 gets() e getchar()

gets() e **getchar()**, assim como scanf(), lêem da entrada padrão. Assim como puts() e putchar(), não suportam formatação. Como o nome sugere, getchar() lê apenas um caractere, e gets() lê uma string até o final da linha ou até que não haja mais dados para ler, e adiciona o terminador de string "\0".

A sintaxe das funções é:

gets(ponteiro_para_string); char c; c = getchar();

No entanto, existe um problema com a função gets(). Veja o exemplo a seguir:

#include <stdio.h> int main() { char buffer[10]; printf("Entre com o seu nome: "); gets(buffer); printf("O nome é: %s", buffer); return 0; }

A notação char buffer[10], que ainda não foi introduzida (e será detalhada na seção Vetores (arrays)), pede que seja reservado um espaço para 10 caracteres para a string buffer. Portanto, se usuário digitar mais de 9 caracteres (pois o terminador de string é adicionado ao que o usuário digitou), os caracteres excedentes adicionais serão colocados na área de memória subsequente à ocupada pela variável, escrevendo uma região de memória que não está reservada à string. Este efeito é conhecido como "estouro de buffer" e pode causar problemas imprevisíveis. Por isso, **não se deve usar a função gets()**; mais tarde introduziremos a função fgets(), que não apresenta esse problema e que deve ser usada no lugar de gets().

12.6 sprintf() e sscanf()

sprintf e sscanf são semelhantes a printf e scanf. Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou lêem em uma string. A única mudança nos argumentos é a necessidade de especificar a string que deve ser lida ou atribuída no início. Veja os exemplos para entender melhor.

#include <stdio.h> int main() { int i; char string1[30]; printf("Entre um valor inteiro: "); scanf("%d", &i);

sprintf(string1, “Valor de i = %d”, i); puts(string1); return 0; }

Nesse exemplo, a mensagem que queríamos exibir na tela foi primeiramente salva em uma string, e depois essa string foi enviada para a tela. Se você olhar bem, se você tivesse alocado um valor menor para string1, também ocorreria um estouro de buffer. Para evitar esse problema, existe a função snprintf, que tem mais um argumento: o tamanho da string (deve ser colocado depois da string onde a mensagem será gravada).

#include <stdio.h> int main() { int i, j, k; char string1[] = “10 20 30"; sscanf(string1, "%d %d %d”, &i, &j, &k); printf(“Valores lidos: %d, %d, %d”, i, j, k); return 0; }

Nesse exemplo, usamos a função sscanf para interpretar os valores contidos na string e guardá-los nas variáveis numéricas.

Capítulo 13

Operações matemáticas (Básico)

13.1 Operações matemáticas

Em C, fazer operações matemáticas simples é bastante fácil e intuitivo. Por exemplo, se quisermos que uma variável contenha o resultado da conta 123 + 912, fazemos assim:

var = 123 + 912;

Os operadores aritméticos básicos são 5: + (adição), - (subtração), * (multiplicação), / (divisão) e % (resto de divisão inteira).

Outro exemplo:

int a = 15; int b = 72; int c = a * b; */* **c** valerá 15×72 */*

Podemos usar mais de um operador na mesma expressão. A precedência é igual à usada na matemática comum:

a = 2 + 4 * 10; */* retornará 42, o mesmo que (2 + (4 * 10)) */* a = 2 + 40 / 2 + 5; */* retornará 27, o mesmo que (2 + (40 / 2) + 5) */*

Você pode usar parênteses, como em expressões matemáticas normais:

a = (2 + 4) * 10; */* retornará 60 */* a = (2 + 40) / (2 + 5); */* retornará 6 */*

Note que uma operação entre números inteiros sempre retornará um número inteiro. Isso é evidente para a adição, subtração e multiplicação. Mas em uma divisão de inteiros, por exemplo 3/2, a expressão retornará apenas a parte inteira do resultado, ou seja, 1.

Se quisermos um resultado não-inteiro, um dos operandos deve ser não-inteiro. Nesse exemplo, poderíamos usar 3.0/2 ou 3/2.0, ou mesmo 3./2 ou (1.0 * 3)/2, pois, em C, uma operação envolvendo um número não-inteiro sempre terá como resultado um número real.

Note que em C o separador decimal é o ponto e não a vírgula.

O seguinte exemplo poderia surpreender, pois o programa dirá que o valor de f continua sendo 3.

#include <stdio.h> int main() { int i = 5; int j = 2; float f = 3.0; f = f + j / i; printf("O valor de f é %f", f); return 0; }

Mas, segundo a precedência dos operadores, j / i deveria ser calculado primeiro, e como ambos os valores são do tipo inteiro, o valor dessa expressão é zero.

É importante que você grave um arquivo .c com o código acima e execute usando o compilador para ver o funcionamento com os próprios olhos.

13.1.1 Abreviações

Alguns tipos de atribuições são bastante comuns, e por isso foram criadas abreviações. Por exemplo, é muito comum incrementar em uma unidade o valor de uma variável (em loops, por exemplo). Em vez de escrever var = var + 1, podemos escrever simplesmente var++. Da mesma maneira, existe o operador de decremento, que *decrementa* em uma unidade o valor da variável: var-- (equivalente a var = var - 1).

Os operadores de decremento e incremento também podem ser utilizados antes do nome da variável. Isso significa que estas duas instruções são equivalentes:

var++; ++var;

Agora vamos supor que você use em seu programa uma variável que aumenta de 10 em 10 unidades. É claro que usar var++ dez vezes não abreviaria nada. Em vez disso, existe a abreviação var += 10.

Genericamente, para qualquer dos cinco operadores aritméticos *op*, vale a abreviação:

var = var *op* num; var *op*= num;

Ou seja, os seguintes pares são equivalentes:

x *= 12; x = x * 12; x /= 10; x = x / 10; x -= 2; x = x - 2; x %= 11; x = x % 11;

Este exemplo clarifica o uso dos operadores de incremento:

#include <stdio.h> int main() { int a, b; a = b = 5; printf("%d\n", ++a + 5); printf("%d\n", a); printf("%d\n", b++ + 5); printf("%d\n", b); return 0; }

O resultado que você deve obter ao executar o exemplo é:

11 6 10 6

Esse resultado mostra que ++var e var++ **não são a mesma coisa** se usados como uma expressão. Quando

usamos os operadores na forma prefixal (antes do nome da variável), o valor é retornado depois de ser incrementado; na forma sufixal, o valor é retornado e depois incrementado. O mesmo vale para o operador de decremento.

E o que aconteceria se você escrevesse algo como o seguinte?

printf("%d\n", a / ++a);

A resposta é: não sabemos. Segundo o padrão C, o resultado disso é indefinido (o que significa que pode variar de um compilador para outro). Não existe uma regra sobre avaliar primeiro o numerador ou o denominador de uma fração. Ou seja, **não use uma variável mais de uma vez numa expressão se usar operadores que a modificam**.

Talvez você tenha achado estranha a linha:

a = b = 5;

Isso é possível porque atribuições são feitas da direita para a esquerda e uma instrução de atribuição é também uma expressão que retorna o valor atribuído. Ou seja, a expressão b = 5 retornou o valor 5, que foi usado pela atribuição a = (b = 5), equivalente a a = 5.

Capítulo 14

Operações matemáticas (Avançado)

O cabeçalho <math.h> contém protótipos de algumas funções na área de matemática. Na versão de 1990 do padrão ISO, somente a versão double das funções foram específicadas; na versão de 1999 foram adicionadas as versões float e long double.

As funções podem ser agrupadas nas seguintes categorias:

1. Funções Trigonométricas
2. Funções Hiperbólicas
3. Funções Exponencial e Logaritmo
4. Funções pow e sqrt
5. Funções de Arredondamento para Números Inteiros, Valores Absolutos e Resto da Divisão

14.1 Funções Trigonométricas

14.1.1 As funções acos e asin

A função acos retorna o arco-cosseno dos seus argumentos em radianos, e a função asin retorna o arco-seno dos seus argumentos em radianos. Todas as funções esperam por argumentos que estejam no intervalo $[-1,+1]$. O arco-cosseno retorna valores no intervalo $[0,\pi]$; o arco-seno retorna valores no intervalo $[-\pi/2,+\pi/2]$.

#include <math.h> float asinf(float x); /* C99 */ float acosf(float x); /* C99 */ double asin(double x); double acos(double x); long double asinl(long double x); /* C99 */ long double acosl(long double x); /* C99 */

14.1.2 As funções atan e atan2

As funções atan retornam o arco-tangente dos arguments em radianos, e a função atan2 retorna o arco-tangente de y/x em radianos. As funções atan retornam o valor no intervalo $[-\pi/2,+\pi/2]$ (a razão pelo que $\pm\pi/2$ está incluido no intervalo é porque os valores decimais pode representar o infinito, e $\text{atan}(\pm\infty) = \pm\pi/2$); as funções atan2 retornam o valor no intervalo $[-\pi,+\pi]$. Para a função atan2, um "domain error" pode ocorrer se os dois argumentos forem zero.

#include <math.h> float atanf(float x); /* C99 */ float atan2f(float y, float x); /* C99 */ double atan(double x); double atan2(double y, double x); long double atanl(long double x); /* C99 */ long double atan2l(long double y, long double x); /* C99 */

14.1.3 As funções cos, sin e tan

As funções cos, sin, e tan retornam o coseno, seno, e tangente do argumento, expresso em radianos.

#include <math.h> float cosf(float x); /* C99 */ float sinf(float x); /* C99 */ float tanf(float x); /* C99 */ double cos(double x); double sin(double x); double tan(double x); long double cosl(long double x); /* C99 */ long double sinl(long double x); /* C99 */ long double tanl(long double x); /* C99 */

14.2 Funções Hiperbólicas

As funções cosh, sinh and tanh computam o coseno hiperbólico, o seno hiperbólico e a tangente hiperbólica respectivamente. Para as funções de seno e coseno hiperbólico, um erro de ...

#include <math.h> float coshf(float x); /* C99 */ float sinhf(float x); /* C99 */ float tanhf(float x); /* C99 */ double cosh(double x); double sinh(double x); double tanh(double x); long double coshl(long double x); /* C99 */ long double sinhl(long double x); /* C99 */ long double tanhl(long double x); /* C99 */

14.3 Funções Exponencial e Logaritmo

14.3.1 A função exp

As funções exp computam a função exponencial de x (e^x). Um "range error" ocorre se o valor de x é muito grande.

#include <math.h> float expf(float x); /* C99 */ double exp(double x); long double expl(long double x); /* C99 */

14.3.2 As funções frexp, ldexp e modf

As funções frexp dividem um número real numa fração normalizada e um número inteiro múltiplo de 2. As funções guardam o número inteiro no objeto apontado por ex.

As funções frexp retornam o valor x de forma que x tem o valor [1/2, 1) ou zero, e value é igual a x vezes 2 elevado a *ex. Se value for zero, as duas partes do resultado seram zero.

As funções ldexp multiplicam um número real por um número inteiro múltiplo de 2 e retornam o resultado. Um "range error" pode ocorrer.

As funções modf divide o argumento value entre um parte inteira e uma fração, cada uma tem o mesmo sinal do argumento. As funções guardam o parte inteira no objeto apontado por *iptr e retornam o fração.

#include <math.h> float frexpf(float value, int *ex); /* C99 */ double frexp(double value, int *ex); long double frexpl(long double value, int *ex); /* C99 */ float ldexpf(float x, int ex); /* C99 */ double ldexp(double x, int ex); long double ldexpl(long double x, int ex); /* C99 */ float modff(float value, float *iptr); /* C99 */ double modf(double value, double *iptr); long double modfl(long double value, long double *iptr); /* C99 */

14.3.3 As funções log e log10

As funções log computam o logaritmo natural do argumento e retornam o resultado. Um "domain error" ocorre se o argumento for negativo. Um "range error" pode ocorrer se o argumento for zero.

As funçõs log10 computam o logaritmo comum (base-10) do argumento e retornam o resultado. Um "domain error" ocorre se o argumento for negativo. Um "range error" ocorre se o argumento for zero.

#include <math.h> float logf(float x); /* C99 */ double log(double x); long double logl(long double x); /* C99 */ float log10f(float x); /* C99 */ double log10(double x); long double log10l(long double x); /* C99 */

14.4 Funções pow e sqrt

14.4.1 As funções pow

As funções pow computam x elevado a y e retornam o resultado. Um "domain error" ocorre se x for negativo e y não for um número inteiro. Um "domain error" ocorre se o resultado não puder ser representado quando x é zero e y é menor ou igual a zero. Um "range error" pode ocorrer.

#include <math.h> float powf(float x, float y); /* C99 */ double pow(double x, double y); long double powl(long double x, long double y); /* C99 */

14.4.2 As funções sqrt

As funções sqrt computam a raiz positiva de x e retornam o resultado. Um "domain error" ocorre se o argumento for negativo.

#include <math.h> float sqrtf(float x); /* C99 */ double sqrt(double x); long double sqrtl(long double x); /* C99 */

14.5 Funções de Arredondamento para Números Inteiros, Valores Absolutos e Resto da Divisão

14.5.1 As funções ceil e floor

As funções ceil computam o menor número inteiro que não seja menor que x e retornam o resultado; as funções floor computam o maior número inteiro que não seja maior que x e retornam o resultado.

#include <math.h> float ceilf(float x); /* C99 */ double ceil(double x); long double ceill(long double x); /* C99 */ float floorf(float x); /* C99 */ double floor(double x); long double floorl(long double x); /* C99 */

14.5.2 As funções fabs

As funções fabs computam o valor absoluto do número real x e retornam o resultado.

#include <math.h> float fabsf(float x); /* C99 */ double fabs(double x); long double fabsl(long double x); /* C99 */

14.5.3 As funções fmod

As funções fmod computam o resto de x/y e retornam o valor x - *i* * y, pra algum número inteiro *i* onde, se y for um número diferente de zero, o resultado tem o mesmo sinal de x e magnitude menor que a magnitude de y. Se y for zero, dependendo da implementação da função, ocorrerá um "domain error" ou a função fmod retornará zero.

#include <math.h> float fmodf(float x, float y); /* C99 */ double fmod(double x, double y); long double fmodl(long double x, long double y); /* C99 */

14.6 Ligações externas

Biblioteca de referência C++ (C++ Reference Library) - cmath (math.h)

Capítulo 15

Operadores

15.1 Operadores Aritméticos

Tabela: Operadores aritmeticos

Notar o último operador. Notar que são operadores que operam apenas com 2 operandos (operadores binários). Na divisão euclidiana temos 30 dividido 7 tem por quociente 4 e como resto 2.

30 / 7 = 4 30 = 7 x 4 + 2 30 % 7 = 2

Existe uma maneira de fazer abreviaturas:

Isto é mais uma abreviatura para os programadores escreverem menos. Há quem ache isto muito estúpido pois é um esforço de assimilação desnecessário em troca a escrever uma letra.

Iremos ver que ter a++ ou ++a é diferente! Mas isso vai ser na história dos loops. (iremos ter situações tipo"++a+5" que seria a+5 mas antes fazer a+1+5.)

Mais uma nota. Em relação ao operador adição ele para além dos números também permite adicionar strings, isto é, junta a segunda string no fim da primeira string. No entanto se juntarmos um dígito com uma string isso já não é permitido.

15.2 Precedência de Operadores aritméticos

Precedência de operadores aritméticos (o operador aritmético tem maior precedência do que o operador de asignment)

Table 4-3: Prioridade dos operadores aritméticos

No caso de termos na mesma instrução operadores com o mesmo nível de precedência (prioridade) fazer a regra da esquerda para a direita. eg. a=8/2*4 seria 16 e não 1, porque temos a divisão está no lado esquerdo.

Mais um ponto em relação ao operador "%" módulo (modulus). Podemos fazer o módulo para números inteiros mas se tentarmos para números do tipo float (ou um deles) fica indefinido. Geralmente resulta num erro de compilação (mas isso vai depender do compilador)

Notar igualmente overflow de que já falamos (antes e depois de compilar). ou seja pego no valor de uma variável adiciono o valor de uma segunda variável e dou esse resultado a uma terceira variável. Isto pode resultar em overflow. Será trabalho do programador em controlar isto.

O que é que resulta se adicionarmos um int por um float e esse float com casas decimais e colocarmos esse resultado num int? o que resulta é que o resultado fica truncado. é a mesma situação de declarar um int e colocar um float. como foi visto no capitulo das variáveis.

15.3 type casting

É fazer com que o resultado saia com a tipologia desejada.

Neste exemplo estamos a fazer com que o firstOp seja convertido para tipo float, quando antes tínhamos declarado como um int, ou seja o valor 10 passa a ser 10.0. e agora como temos um float a dividir por um int, o que acontece é que há uma conversão automática, ou seja o 2ª int é convertido em float, fazendo com que o resultado seja um float Podemos utilizar qualquer uma das expressões seguintes para exprimir o tycasting.

float result = (float) firstOp / secondOp; float result = float (firstOp) / secondOp; float result = firstOp / (float) secondOp; float result = firstOp / float (secondOp);

15.4 Expoentes

O C e o C++ não têm o operador expoente, no entanto, tem a função pow (de power) que está no cabeçalho da biblioteca padrão <math.h>. a função pow() tem 2 argumentos, o primeiro para a base e o 2º para o expoente. o 1º argumento tem de ser float ou double.

15.5 Operadores relacionais

Permite fazer comparações lógicas de ordenação de números, e ainda de letras (mas não strings) Table: Relational Operators

Voçê poderia se perguntar: Como é que o computador faz essa comparação ? de onde é que ele sabe que um número A é maior que outro B?

Resposta: Considere que você quisesse comparar dois dados tipo char, lembrando que um char na verdade é um número inteiro na tabela ASCII. Sendo assim suponha que gostarias de comparar o caractere 'a' que é igual a 97 na tabela ascii com o caractere 't' que é 116 na tabela; assim, ao comparar 97 com 116 o que aconteçe na memória é a comparação de 01100001 (97) com 01110100 (116) em um registrador específico, vão sendo somadas as potências de 2 da esquerda para a direita de forma que fica evidente para ele (o registrador) quem é maior. Isso é o que aconteçe quando comparamos duas strings com a função strcmp e ela retorna um número para a diferença entre elas. Esse número é justamente a diferença entre os valores da tabela ASCII entre o primeiro caractere das duas.

notar o operador == que é a comparação de igualdade. o operador = é de atribuição.

Estes operadores também são binários, ie, comparam dois operandos. o resultado de uma expressão relacional dá um valor bool (verdadeiro=1 ou falso=0)

- 4 != 4 false
- 4 == 5 false

Eu ainda posso comparar um int com um float que isso não dá problema. ou seja com dados númericos não há problema. Comparações entre dois caracteres também não há problema pois os caracteres são números na tabela ASCII. Mas não usem para strings (pois aí estaríamos a comparar o quê, se as strings são conjunto de caracteres?) Não esquecer o ponto que o digito pode ser um char ou estar em forma númerica. e esse char vai ter o valor na tabela. nós num capitulo posterior iremos ver que poderemos fazer a conversão de char para int e vice versa.

15.6 Precedência dos operadores relacionais

Table: Precedence of Relational Operators

Novamente existe a regra da esquerda para a direita caso haja igualdade de precedência

15.7 Operadores lógicos

Estes operadores comparam já condições de precedência Table: Logical Operators

Estes operadores também são binários mas desta vez os operandos são resultados boolean, que podem advir dos operadores relacionais (comparação) pois davam valores boolean.

- Para o operador and (&&) – basta uma das comparações ser falsa paro resultado ser falso
- Para o operador or (||) – basta uma das comparações dos operandos ser verdadeira para se tornar verdadeira
- Por fim o operador not— é um operador unário – é apenas para um valor boolean que pode ser resultado de comparação

Exemplo:

if (age <= 12 || age >= 65) printf(“Admission is free”); else printf(“You have to pay”);

15.8 Precedência Operadores lógicos e Relacionais

Tabela: A preçedençia dos operadores logicos e relacionais

Operador (da mais alta para a mais baixa) ! Relacionais (>, >=, <, <=, ==. !=) && ||

cuidado!

- if (!age > 12 && age < 65)

Note o ! no exemplo. é sempre bom recorrer aos parênteses

15.9 Operadores Lógicos Bit a Bit

15.9.1 Deslocamento de bits

x = a << b é igual a x = a*2^b; x = a >> b é igual a x = a/2^b;

15.10 Todos os Operadores

Comparações de precedência entre Operadores aritméticos, relacionais e lógicos

15.11 Exercícios

- (7 == 5) // Avalia como falso.
- (5 > 4) //Avalia como verdadeiro
- (3 != 2) //Avalia como verdadeiro

- (6 >= 6) //Avalia como verdadeiro
- (5 < 5) // Avalia como falso
- (a == 5) // Avalia como falso , porque a não é igual à 5.
- (2*3 >= 6) // Avalia como verdadeiro porque (2*3 >= 6) é verdadeiro.
- (3+4 > 2*6) // Avalia como falso porque (3+4 > 2*6) é falso.
- !(5 == 5) // Avalia como falso,porque a expressão a direita (5 == 5) é verdadeira.
- !true // Avalia como falso
- !false //Avalia como verdadeiro
- ((5 == 5) && (3 > 6)) // Avalia como falso (true && false).
- ((5 == 5) || (3 > 6)) // Avalia como verdadeiro (true || false).

Capítulo 16

Controle de fluxo

16.1 Controle de fluxo

Dificilmente um programa em C irá executar sempre as mesmas instruções, na mesma ordem, independentemente do que tenha acontecido anteriormente ou do valor que foi fornecido. É muito comum que alguém queira que um pedaço de código só seja executado se uma certa condição for verdadeira; também é comum querer que um pedaço de código seja repetido várias vezes, de tal maneira que simplesmente copiar o código não resolveria o problema ou seria trabalhoso demais. Para casos como esses, existem as **estruturas de controle de fluxo**.

Em C, existem várias instruções relacionadas ao controle de fluxo:

- **if**, que executa um bloco apenas se uma condição for verdadeira;
- **switch**, que executa um bloco de acordo com o valor de uma expressão ou variável;
- **for**, que executa um bloco repetidas vezes enquanto uma condição for verdadeira, executando uma instrução (geralmente de incremento ou decremento de uma variável) após cada execução;
- **while**, que executa um bloco enquanto uma condição for verdadeira;
- **do**, semelhante ao while, mas a condição é avaliada após a execução (e não antes);
- **goto**, que simplesmente pula para um lugar predefinido.

Porém, antes de entrar no estudo dessas estruturas, você deve saber como escrever uma condição. É o que explicamos a seguir.

16.2 Expressões de condição

Uma expressão de condição é uma expressão normal em C que, quando avaliada, será interpretada como verdadeira ou falsa. Em C, na verdade, esse valor é um valor inteiro que sendo 0 (zero) significa falso, sendo qualquer outro número significa verdadeiro.

Geralmente em expressões condicionais usamos os operadores *relacionais*, ou seja, que avaliam a relação entre seus dois operandos. Existem seis deles:

Todos esses operadores são binários, ou seja, trabalham com dois valores ou operandos. Esses operadores sempre comparam o valor da esquerda com o da direita, ou seja, a expressão a > b significa "*a* é maior que *b*".

Note que para saber se dois números são iguais devemos usar **dois sinais de igual**. Um erro muito comum é esquecer de um deles, transformando a comparação numa atribuição — por exemplo:

```
if (x = 1) ...
```

O que acontece aqui é que a variável *x* recebe o valor 1, de modo que a expressão entre parênteses também terá o valor 1 — tornando a "condição" sempre verdadeira. Similarmente, se usássemos o número zero, a expressão sempre seria falsa. Portanto, sempre tome cuidado com esse tipo de comparação. A maneira certa de comparar com um número é:

```
if (x == 1) ...
```

Também é comum que combinemos condições. Por exemplo, podemos querer que um número seja menor que 10 **ou** maior que 50. Como o operador "ou" é "||", escreveríamos: n < 10 || n > 50. A seguir você vê os operadores lógicos:

Algumas explicações sobre os operadores lógicos:

- O operador **"não"** é unário, ou seja, é uma operação que envolve apenas um valor. O que ele faz é inverter o valor de seu operando: retorna falso se a expressão for verdadeira e vice-versa. Deve-se usar parênteses ao negar uma expressão: !(x > 6), por exemplo.
- O operador **"ou"** retorna "verdadeiro" se pelo menos um dos operandos for verdadeiro; retorna "falso" apenas se ambos forem falsos.

- O operador **"e"** retorna "verdadeiro" apenas se ambos os seus operandos forem verdadeiros.

Observação Se você quer saber se um número está **entre** outros dois, a sintaxe matemática (10 < n < 50) não funcionará. Se você usar esse código, na verdade primeiramente será avaliada a expressão 10 < n, que poderá resultar em 0 ou 1. Portanto, a expressão equivale a (0 ou 1) < 50, o que é sempre verdadeiro.

A comparação correta envolveria o operador "e" (&&): 10 < n && n < 50.

Pelo fato de todo valor diferente de zero ser avaliado como verdadeiro e zero como falso, existem as seguintes equivalências (apenas quando estas expressões são usadas como condições):

(x == 0) equivale a (!x) (x != 0) equivale a (x)

16.3 Testes

Testes são estruturas de controle que executam certos blocos de código apenas se uma certa condição for verdadeira. Existem três estruturas desse tipo em C:

16.3.1 if

O teste **if** avalia uma condição e, se ela for verdadeira, executa um bloco de código. A sintaxe correspondente a isso é:

if (condição) { ... /* bloco a ser executado se a condição for verdadeira */ }

Mas também podemos especificar um bloco a ser executado caso a condição for falsa. Nesse caso, escrevemos:

if (condição) { ... /* bloco a ser executado se a condição for verdadeira */ } else { ... /* bloco a ser executado se a condição for falsa */ }

As chaves podem ser omitidas caso haja **apenas uma instrução** no bloco. Por exemplo:

if (x == 5) printf ("x é igual a 5.\n");

Perceba que, se esquecermos as chaves, o compilador não deverá dar nenhum erro; no entanto, tudo que exceder a primeira instrução será executado incondicionalmente, **mesmo que esteja na mesma linha**! No exemplo a seguir, a frase "x é igual a 5" seria exibida mesmo que o número não fosse 5!

if (x == 5) j++; printf ("x é igual a 5.\n");

Podemos avaliar diversas condições com os testes if, bastando para isso colocar um novo teste no bloco else. Também é possível aninhar blocos if, ou seja, colocar um dentro de outro:

if (x > 9) { printf ("x é maior que 9.\n"); } else if (x >= 5) { printf ("x é maior ou igual a 5, mas não maior que 9.\n"); } else { if (x == 0) { printf ("x é igual a zero.\n"); } else { printf ("x é não-nulo e menor que 5.\n"); } }

16.3.2 switch

O teste **switch** compara uma expressão com diversos valores que podem estar associados a blocos de códigos diferentes, e executa o bloco de código correspondente ao valor encontrado. Você também pode especificar um bloco que deve ser executado caso nenhum dos outros valores seja encontrado: é o bloco default ("padrão" em inglês).

switch (expressão) { case valor1: instruções; break; case valor2: instruções; break; ... default: instruções; }

Note que no teste **switch** não precisamos usar chaves em volta dos blocos, a menos que declaremos variáveis neles. Um exemplo da utilização de **switch** seria a criação de um menu:

int opcao; printf ("[1] Cadastrar cliente\n" "[2] Procurar cliente\n" "[3] Inserir pedido\n" "[0] Sair\n\n" "Digite sua escolha: "); scanf ("%d", &opcao); switch (opcao) { case 1: cadastra_cliente(); break; case 2: procura_cliente(); break; case 3: insere_pedido(); break; case 0: return 0; default: printf ("Opção inválida!\n"); }

A instrução **break** indica que deve-se continuar a execução após o final do bloco switch (pulando o que estiver no meio). Se ela não fosse usada, para um certo valor encontrado, seriam executadas também as instruções de todos os valores abaixo dele. Em alguns casos, podemos omitir intencionalmente a instrução break. Por exemplo, no exemplo acima, não colocamos uma instrução break para o valor zero, pois quando retornamos de uma função (return 0) o bloco switch já é abandonado.

Também podemos querer que uma instrução seja executada para mais de um valor. Vamos supor que no nosso menu as duas primeiras opções fossem "Cadastrar pessoa física" e "Cadastrar pessoa jurídica", e tívessemos uma função que faz o cadastro diferentemente dependendo do valor da variável pessoa_fisica. Poderíamos fazer um código assim:

switch (opcao) { case 1: /* pessoa física */ pessoa_fisica = 1; case 2: cadastra(); break; ... }

Nesse caso, para qualquer uma das duas opções seria executada a função cadastra, mas se selecionarmos "pessoa física" a variável será atribuída antes.

16.3.3 Operador ternário *"?:"*

O operador ternário **?:** é uma alternativa abreviada da estrutura **if/else**. Ele avalia uma expressão e retorna um certo valor se ela for verdadeira, ou outro valor se ela for falsa. Sua sintaxe é:

condição ? valorSeVerdadeira : valorSeFalsa

Note que, ao contrário de **if**, ao usarmos o operador condicional ?: precisamos sempre prover tanto o valor para o caso de a condição ser falsa quanto o valor para o caso de ela ser verdadeira.

O operador condicional pode ser usado em situações como essa:

int horaAbertura = (diaSemana == DOMINGO) ? 11 : 9; printf ("Abrimos às %d horas", horaAbertura);

Ou seja, se o dia da semana for domingo, a variável horaAbertura será definida para 11; caso contrário, será definida para 9.

Outro exemplo:

if (numMensagens > 0) { printf ("Você tem %d mensage%s", numMensagens, (numMensagens > 1) ? "ns" : "m"); }

Neste caso, o programa utilizaria "mensagens" caso houvesse mais de uma mensagem, e "mensagem" caso houvesse apenas uma mensagem.

16.4 Loops

Loops são conjuntos de instruções que devem ser executadas repetidas vezes, enquanto uma condição for verdadeira. Em C há 3 tipos de loops: while, do ... while e for.

16.4.1 while

O loop **while** testa uma condição; se ela for verdadeira, o bloco correspondente é executado e o teste é repetido. Se for falsa, a execução continua logo após o bloco. A sintaxe de while é:

while (condição) { ... }

Por exemplo:

while (a < b) { printf ("%d é menor que %d", a, b); a++; }

Este código seria executado até que a fosse igual a b; se a fosse igual ou maior que b, nada seria executado. Por exemplo, para b = 10 e a < 10, a última mensagem que o usuário veria é "9 é menor que 10".

Repare que o loop **while** é como fosse um **if**, ou seja, o bloco é executado se a condição for verdadeira. A diferença é que ao final da execução, o while é executado novamente, mas o if não. No loop **while** (assim como nos loops **do** e **for**) também podemos usar a sintaxe abreviada para apenas uma instrução:

while (a < b) a++;

Loops infinitos

Você pode fazer loops infinitos com **while**, usando uma condição que é sempre verdadeira, como "1 == 1" ou simplesmente "1" (que, como qualquer valor não-nulo, é considerado "verdadeiro"):

while (1) { ... }

Você pode sair de um loop — infinito ou não — com a instrução break, que você já viu no teste **switch** e será explicada mais abaixo.

16.4.2 do ... while

O loop **"do ... while"** é exatamente igual ao **"while"** exceto por um aspecto: a condição é testada depois do bloco, o que significa que o bloco é executado pelo menos uma vez. A estrutura do ... while executa o bloco, testa a condição e, se esta for verdadeira, volta para o bloco de código. Sua sintaxe é:

do { ... } while (condição);

Note que, ao contrário das outras estruturas de controle, **é necessário colocar um ponto-e-vírgula** após a condição.

do { printf ("%d\n", a); a++; } while (a < b);

Um exemplo de utilização de do ... while é em um menu. Pediríamos que o usuário escolhesse uma opção até que ele escolhesse uma opção válida:

#include <stdio.h> int main () { int i; do { printf ("Escolha a fruta pelo número:\n\n"); printf ("\t(1) Mamão\n"); printf ("\t(2) Abacaxi\n"); printf ("\t(3) Laranja\n\n"); scanf("%d", &i); } while (i < 1 || i > 3); switch (i) { case 1: printf ("Você escolheu mamão.\n"); break; case 2: printf ("Você escolheu abacaxi.\n"); break; case 3: printf ("Você escolheu laranja.\n"); break; } return 0; }

16.4.3 for

O loop **for** é nada mais que uma abreviação do loop **while**, que permite que alguma inicialização seja feita antes do loop e que um incremento (ou alguma outra ação) seja

feita após cada execução sem incluir o código dentro do bloco. A sua forma geral é

for (inicialização; condição; incremento) { instruções; }

E equivale a

inicialização; while (condição) { instruções; incremento; }

Um exemplo do uso de **for**:

for (a = 1; a < 10; a++) { if(a == 1) puts (“Numero de voltas previstas 9.”); printf(“Numero de loop ou volta : %i ", a); printf(“Valor de a : %i ", a); }

Nesse exemplo, primeiro definimos o valor de a como 1; depois, o código (...) é repetido enquanto a for menor que dez, incrementando em uma unidade o valor de a após cada execução do código. Analisando essas condições, você podera perceber que o código será executado nove vezes: na primeira execução, temos a = 1; após a nona execução, a é igual a 10, e portanto o bloco não será mais repetido.

Também podemos dar mais de uma instrução de inicialização ou de incremento (separadas por vírgula), além de poder usar naturalmente condições compostas com o uso dos operadores lógicos:

for (a = 1, b = 1; a < 10 && (b / a) < 20; a++, b *= 2) { ... }

Nesse exemplo, “a” e “b” são inicializados com o valor 1. A cada loop, o valor de “a” é incrementado em uma unidade e o de “b” é dobrado. Isso ocorre enquanto “a” for menor que 10 e a razão entre “b” e “a” for menor que 20. Se você construir uma tabela com os valores de cada variável a cada loop (ou colocar algum contador dentro do loop), verá que ocorrem oito execuções.

Assim como **while**, o loop **for** testa a condição; se a condição for verdadeira ele executa o bloco, faz o incremento e volta a testar a condição. Ele repete essas operações até que a condição seja falsa.

Podemos omitir qualquer um dos elementos do **for** se desejarmos. Se omitirmos a inicialização e o incremento, o comportamento será exatamente igual ao de **while**. Se omitirmos a condição, ficaremos com um loop infinito:

for (inicialização; ; incremento) { ... }

Podemos também omitir o bloco de código, se nos interessar apenas fazer incrementos ou se quisermos esperar por alguma situação que é estabelecida por uma função externa; nesse caso, usamos o ponto-e-vírgula após os parênteses de **for**. Isso também é valido para o loop **while**:

for (inicialização; condição; incremento) ; while (condição) ;

Por exemplo, suponha que temos uma biblioteca gráfica que tem uma função chamada graphicsReady(), que indica se podemos executar operações gráficas. Este código executaria a função repetidas vezes até que ela retornasse “verdadeiro” e então pudéssemos continuar com o programa:

while (!graphicsReady()) ;

16.4.4 break e continue

Você já viu **break** sendo usado para sair do teste **switch**; no entanto, ele funciona também nos loops — while, do e for. Nos três casos, ele sai do último loop iniciado (mesmo que haja mais de um). Por exemplo:

while (1) { if (a > b) break; a++; }

break sempre faz com que a execução do programa continue na primeira instrução seguinte ao loop ou bloco.

A instrução **continue** é parecida com **break**, porém ao executá-la saltamos para a próxima iteração loop ao invés de terminá-lo. Usar **continue** equivale a chegar ao final do bloco; os incrementos são realizados (se estivermos em um loop **for**) e a condição é reavaliada (qualquer que seja o loop atual).

#include <stdio.h> int main() { int opcao = 0; while (opcao != 5) { printf(“Escolha uma opção entre 1 e 5: "); scanf("%d”, &opcao); /* se a opção for inválida, volta ao início do loop */ if (opcao > 5 || opcao < 1) continue; switch (opcao) { case 1: printf("\n --> Primeira opcao..”); break; case 2: printf("\n --> Segunda opcao..”); break; case 3: printf("\n --> Terceira opcao..”); break; case 4: printf("\n --> Quarta opcao..”); break; case 5: printf("\n --> Abandonando..”); break; } } return 0; }

Esse exemplo recebe uma opção do usuário. Se ele digitar uma opção inválida (ou seja, não for um número de 1 a 5), a instrução **continue** voltará ao começo do loop e o programa pedirá novamente a entrada do usuário. Se ele digitar uma opção válida, o programa seguirá normalmente.

16.5 Saltos incondicionais: goto

O **goto** é uma instrução que salta incondicionalmente para um local específico no programa. Esse local é identificado por um rótulo. A sintaxe da instrução **goto** é:

goto nome_do_rótulo;

Os nomes de rótulo são identificadores sufixados por dois-pontos (:), no começo de uma linha (podendo ser prece-

didos por espaços). Por exemplo:

nome_do_rótulo: ... goto nome_do_rótulo;

Muitos programadores evitam usar o goto pois a maioria dos saltos pode ser feita de maneira mais clara com outras estruturas da linguagem C. Na maioria das aplicações usuais, pode-se substituir o goto por testes, loops e chamadas de funções.

16.6 Terminando o programa

O programa pode ser terminado imediatamente usando a função *exit*:

void exit (int codigo_de_retorno);

Para utilizá-la deve-se colocar um include para o arquivo de cabeçalho stdlib.h. Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o código_de_retorno. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema.

#include <stdio.h> #include <stdlib.h> /* Para a função exit() */ int main (void) { FILE *fp; ... fp=fopen (“exemplo.bin”,"wb”); if (!fp) { printf (“Erro na abertura do arquivo. Fim de programa.”); exit (1); } ... return 0; }

Capítulo 17

Funções

17.1 O que é função

Uma **função** é um pedaço de código que faz alguma tarefa específica e pode ser chamado de qualquer parte do programa quantas vezes desejarmos.

Podemos também dizer que funções agrupam operações em um só nome que pode ser chamado em qualquer parte do programa. Essas operações são então executadas todas as vezes que chamamos o nome da função.

Utilizamos funções para obter:

- **Clareza do código**: separando pedaços de código da função main(), podemos entender mais facilmente o que cada parte do código faz. Além disso, para procurarmos por uma certa ação feita pelo programa, basta buscar a função correspondente. Isso torna muito mais fácil o ato de procurar por erros.

- **Reutilização**: muitas vezes queremos executar uma certa tarefa várias vezes ao longo do programa. Repetir todo o código para essa operação é muito trabalhoso, e torna mais difícil a manutenção do código: se acharmos um erro nesse código, teremos que corrigi-lo em todas as repetições do código. Chamar uma função diversas vezes contorna esses dois problemas.

- **Independência**: uma função é relativamente independente do código que a chamou. Uma função pode modificar variáveis globais ou ponteiros, mas limitando-se aos dados fornecidos pela chamada de função.

A ideia funções é permitir você encapsular várias operações em um só escopo que pode ser invocado ou chamado através de um nome. Assim é possível então chamar a função de várias partes do seu programa simplesmente usando o seu nome.

Exemplo:

#include <stdio.h> int main(void) { imprime_par(3,4); imprime_par(−2,8); return 0; }

No exemplo acima, a função **imprime_par** foi usada para executar o pedaço de programa que imprime um par de números. A saída do programa acima será:

{3,4} {−2,8}

A função **imprime_par** é definida da seguinte forma:

void imprime_par(int a, int b) { printf("{ %d, %d }\n",a,b); }

O programa completo em C é mostrado abaixo:

#include <stdio.h> /** * Declaração da função imprime_par * Essa função recebe dois inteiros como argumento e os imprime * da seguinte forma {a,b} */ void imprime_par(int a, int b); int main(int argc, char **argv) { imprime_par(3,4); //chamando a função imprime_par(−2,8); //chamando novamente return 0; } //Implementação da função //A implementação da função pode conter várias linhas de código void imprime_par(int a, int b) { printf("{ %d, %d }\n",a,b); }

A definição de funções em C devem ser feitas antes do uso das mesmas. Por isso em nosso exemplo definimos a função **imprime_par** antes de usá-la dentro do main.

A linha que define ou declara a função também é conhecida como **assinatura** da função. Normalmente as assinaturas das funções são definidas dentro de arquivos de cabeçalho **.h**

17.2 Definindo uma função

Uma função pode necessitar de alguns dados para que possa realizar alguma ação baseada neles. Esses dados são chamados **parâmetros** da função. Além disso, a função pode retornar um certo valor, que é chamado **valor de retorno**. Os parâmetros (e seus tipos) devem ser especificados explicitamente, assim como o tipo do valor de retorno.

A forma geral da definição de uma função é:

[tipo de retorno da função] [nome da função] (1º parâmetro, 2º parâmetro, ...) { //código }

- Para o nome da função e dos parâmetros valem as mesmas regras que foram dadas para os nomes de variáveis. Não podemos usar o mesmo nome para funções diferentes em um programa.

- Todas as funções devem ser definidas antes da função **main**, ou deve ser feito o **protótipo** da função, que veremos mais adiante.

- O código deve estar obrigatoriamente dentro das chaves e funciona como qualquer outro bloco.

17.2.1 Valor de retorno

Freqüentemente, uma função faz algum tipo de processamento ou cálculo e precisa retornar o resultado desse procedimento. Em C, isso se chama **valor de retorno** e pode ser feito com a instrução **return**. Para poder retornar um valor, precisamos especificar seu tipo (char, int, float, double e variações). Para efetivamente retornar um valor, usamos a instrução return seguida do valor de retorno, que pode ou não vir entre parênteses. Um exemplo bem simples de função que retorna um valor inteiro:

int tres() { return 3; // poderia também ser return (3); }

O tipo de retorno, além dos tipos normais de variáveis (char, int, float, double e suas variações), pode ser o tipo especial **void**, que na verdade significa que não há valor de retorno.

Nota Muitos livros dizem que a função main tem tipo de retorno **void**, o que não está correto. Segundo o padrão da linguagem C, a função main deve ter retorno do tipo **int**. Compiladores como o gcc darão mensagens de erro caso a função main() não seja definida corretamente.

17.2.2 Parâmetros

Como já foi dito, um parâmetro é um valor que é fornecido à função quando ela é chamada. É comum também chamar os parâmetros de **argumentos**, embora argumento esteja associado ao valor de um parâmetro.

Os parâmetros de uma função podem ser acessados da mesma maneira que variáveis locais. Eles na verdade funcionam exatamente como variáveis locais, e modificar um argumento não modifica o valor original no contexto da chamada de função, pois, ao dar um argumento numa chamada de função, ele é copiado como uma variável local da função. A única maneira de modificar o valor de um parâmetro é usar ponteiros, que serão introduzidos mais adiante.

Para declarar a presença de parâmetros, usamos uma *lista de parâmetros* entre parênteses, com os parâmetros separados por vírgulas. Cada declaração de parâmetro é feita de maneira semelhante à declaração de variáveis: a forma geral é tipo nome. Por exemplo:

int funcao (int a, int b) float funcao (float preco, int quantidade) double funcao (double angulo)

Para especificar que a função não usa nenhum parâmetro, a lista de parâmetros deve conter apenas a palavra-chave **void**. No entanto, ela é freqüentemente omitida nesses casos. Portanto, você poderia escrever qualquer uma destas duas linhas:

void funcao (void) void funcao ()

Note que os nomes dos parâmetros são usados apenas na própria função (para distinguir os argumentos); eles não têm nenhuma relação com as variáveis usadas para chamar a função.

17.2.3 Chamadas de funções

Para executar uma função, fazemos uma **chamada de função**, que é uma instrução composta pelo nome da função, seguido pela lista de argumentos entre parênteses:

nome_da_função (arg1, arg2, ...);

Os argumentos podem ser qualquer tipo de expressão: podem ser variáveis, valores constantes, expressões matemáticas ou até mesmo outras chamadas de função.

Lembre que você deve sempre dar o mesmo número de argumentos que a função pede. Além disso, embora algumas conversões de tipo sejam feitas automaticamente pelo compilador, você deve atender aos tipos de argumentos.

Note que o valor dos argumentos é **copiado** para a função, de maneira que as variáveis originais ficam inalteradas mesmo que na função tentemos alterá-las. A isso chamamos passagem de argumentos *por valor* (ao contrário de *por referência*). Veremos como modificar as variáveis originais na seção Ponteiros.

A própria chamada de função também é uma expressão cujo valor é o valor de retorno da função, bastando colocá-la no lado direito de um sinal de igual para guardar o valor numa variável. Por exemplo, se a função "quadrado" retorna o quadrado de um número inteiro, podemos fazer assim para calcular o quadrado de 11 na variável *x*:

int x = quadrado (11);

17.3 Dois exemplos

#include <stdio.h> int quadrado (int x) { return (x * x); } void saudacao (void) { printf ("Olá!\n"); } void despedida (void) { printf ("Fim do programa.\n"); } int main () { int numero, resultado; saudacao (); printf

(“Digite um número inteiro: "); scanf ("%d”, &numero); resultado = quadrado (numero); printf (“O quadrado de %d é %d.\n”, numero, resultado); despedida (); return 0; }

Você veria na tela, ao executar o programa:

Olá! Digite um número inteiro: 42 O quadrado de 42 é 1764. Fim do programa.

Repare que, ao chegar na chamada de uma função, o programa passa o controle para essa função e, após seu término, devolve o controle para a instrução seguinte na função original.

Mais um exemplo, com uma função de 3 argumentos:

#include <stdio.h> /* Multiplica 3 numeros */ void mult (float a, float b, float c) { printf ("%f”,a*b*c); } int main () { float x, y; x = 23.5; y = 12.9; mult (x, y, 3.87); return 0; }

17.4 Protótipo ou Declaração de função

Quando um programa C está sendo compilado e uma chamada de função é encontrada, o compilador precisa saber o tipo de retorno e os parâmetros da função, para que ele possa manipulá-los corretamente. O compilador só tem como saber isso se a função já tiver sido definida. Portanto, se tentarmos chamar uma função que está definida abaixo da linha onde estamos fazendo a chamada, ou mesmo em outro arquivo, o compilador dará uma mensagem de erro, pois não conseguiu reconhecer a função.

//Exemplo de erro de chamada de função int main() { int a = 1; int b = 2; soma(a,b); // erro: a função está definida abaixo desta linha! } void soma(int a, int b) { printf("%d”, a+b); }

Nesses casos, podemos **declarar** uma função antes de defini-la. Isso facilita o trabalho de usar diversas funções: você não precisará se importar com a ordem em que elas aparecem nos arquivos.

A declaração de função (também chamada de protótipo de função) nada mais é que a definição da função sem o bloco de código. Como uma instrução, ela deve ser seguida de um ponto-e-vírgula. Portanto, para declarar a função:

int quadrado (int x) { return (x * x); }

escreveríamos:

int quadrado (int x);

Numa declaração, também podemos omitir os nomes dos parâmetros, já que estes são ignorados por quem chama a função:

int quadrado (int);

Poderíamos, por exemplo, reorganizar o início do programa-exemplo dado um pouco acima, o que permitiria colocar as funções em qualquer ordem mesmo que houvesse interdependência entre elas:

#include <stdio.h> int quadrado (int x); void saudacao (void); void despedida (void); // seguem as funções do programa

Note que a definição da função não deve contradizer a declaração da mesma função. Se isso ocorrer, uma mensagem de erro será dada pelo compilador.

17.5 Variáveis locais versus globais

Quando declaramos as variáveis, nós podemos fazê-lo

- Dentro de uma função
- Fora de todas as funções inclusive a main().

As primeiras são as designadas como locais: só têm validade dentro do bloco no qual são declaradas. As últimas são as globais, elas estão vigentes em qualquer uma das funções.

Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local. Daqui conclui-se e bem que, podemos ter variáveis com o mesmo nome, o que contradiz o que nós dissemos no capitulo das variáveis.

Então reformulamos:

> Apenas na situação em que temos 2 variáveis locais é que é colocada a restrição de termos nomes diferentes caso contrário não conseguiríamos distinguir uma da outra.

“largo” e “alto” são variáveis internas fazem parte de “minhaFuncion()".

/*espanhol para incultos :)*/ <== Comentários da função void minhaFuncion() { double largo = 5; double alto = 6; }

As variáveis largo e alto não estão definidas aqui abaixo, isto quer dizer que elas nao tem nem um valor.

E não podemos usar os valores definido dentro da “minhaFuncion”, pois não há nenhuma instrução que defina que valor usar. Lembre-se: O computador não vai adivinhar qual valor usar. Deve-se definir cada instrução.

void calcular() /*Não houve definição de valor entre parenteses*/ { long superficie = largo * alto; /*Error bip bip valor nao definido*/ return(superficie); }

Nesse exemplo abaixo, poderemos usar o valor das variáveis externas dentro de todas as funções. Exemplo:

#include <stdio.h> /* Variaveis externas */ long largo = 10; long alto = 20; void F_soma () { /*soma é uma variavel interna e largo e alto sao variaveis externas */ long soma = largo + alto ; printf("largo + alto = %i \n", soma); } long calcular() { long superficie = largo * alto; return superficie; } int main(void) { F_somma (); printf("Superficie : %ld \n", calcular()); return 0 ; }

Curiosidade A palavra reservada "auto" serve para dizer que uma variável é local, mas a utilização de **auto** não é mais necessária pois as variáveis declaradas dentro de um bloco já são consideradas locais.

17.6 Passagem de parâmetros por valor e por referência

O que nós temos feito quando chamamos uma função é a dita **chamada por valor**. Quer dizer, quando chamamos uma função e passamos parâmetros para a *função protótipo* e depois para a *função definição*, o valor dos argumentos passados são copiados para os parâmetros da função. Estes existem independentemente das variáveis que foram passadas. Eles tomam apenas uma cópia do valor passado, e se esse valor for alterado o valor dos argumentos passados não são alterados. Ou seja, não são alterados os valores dos parâmetros fora da função. Este tipo de chamada de função é denominado chamada (ou passagem de parâmetros) **por valor**.

Dito de outra maneira. Passamos a variável "a", ela entra na definição da função como copia de "a" e entra como variável "b". Se a variável "b" for alterada no decorrer da função, o valor de "a" não é alterado.

#include <stdio.h> float quadrado(float num); //protótipo da função quadrado() int main () { float num, res; //declaro 2 variáveis: num , res printf("Entre com um numero: "); scanf("%f", &num); //associo o valor inserido á variável num res = quadrado(num); //chamo a função quadrado e passo o parâmetro num printf("\n\nO numero original e: %f\n", num); printf("e seu quadrado vale: %f\n", res); getchar(); return 0; } float quadrado (float num) //descrição da função quadrado { return num * num; //retorna num ao quadrado }

Quando a função main() é executada, ela chega a meio e vê uma chamada para a função quadrado() e onde é passado o parâmetro "num". Ela já estava a espera, pois "viu" o protótipo. Ela então vai executar a função que está depois da função do **main**(). E o que acontece é que o "num", vai ficar com o dobro do valor. Esse valor do **main**() vai entrar novamente no **main**(). E é associado á variável "res". Depois temos a impressão da variável "num" e "res". Ora o que acontece é que o valor do "num" fica igual ao valor antes de entrar na função. Fazemos a mesma coisa agora com a variável "a" e "b", e vemos que agora a função a é alterada. Resumindo, o valor variável quando entra numa outra função não é alterado (na passagem por valor).

Quando o valor do parâmetro é alterado denominamos chamada (ou passagem) **por referência**. O C não faz chamadas por referência. Mas podemos simular isto com outra arma do C que são os **ponteiros**, que serão melhor explicados mais adiante.

17.7 void

Como dissemos, uma função retorna um valor. E pode receber parâmetros. O void é utilizado da seguinte forma:

void função(void) { //codigo }

No exemplo acima, a palavra **void** define que:

- não vai receber parâmetros; e
- não vai retornar qualquer valor.

Ou melhor, **void** é uma explicitação do programador que aquela função não vai receber ou retornar nenhum valor.

O valor da função é ignorado, mas a função realmente retorna um valor, por isso para que o resultado não seja interpretado como um erro e bom declarar void.

Nota

> Não se pode utilizar **void** na função principal **main**, apesar de existirem exemplos com void em algumas bibliografias. Infelizmente, alguns compiladores aceitam void main(). O main() é especial e tem de retornar um int. Uma execução bem sucedida do programa costuma retornar 0 (zero) e, em caso de erro, retorna 1 (um).

17.8 Recursividade

Uma função pode chamar a si própria. Uma função assim é chamada **função recursiva**. Há várias operações matemáticas recursivas, das quais exemplos bem conhecidos são a seqüência de Fibonacci e o fatorial.

Daremos o exemplo do cálculo do fatorial de um número, definido como o produto de todos os números naturais (não nulos) menores ou iguais a ele — por exemplo, 5!

(lê-se “cinco fatorial”) é igual a $5 \times 4 \times 3 \times 2 \times 1$. Atenção à convenção $0! = 1$.

Uma maneira de definir o algoritmo de fatorial é:

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n(n-1)!, & \text{se } n \geq 2 \end{cases}$$

E a implementação correspondente seria esta:

#include <stdio.h> #include <stdlib.h> int fat(int n) { if (n) return n*fat(n-1); else return 1; } int main() { int n; printf("\n\nDigite um valor para n: "); scanf("%d”, &n); printf("\nO fatorial de %d e' %d”, n, fat(n)); return 0; }

Exemplo 2 :

#include <stdio.h> #include <stdlib.h> unsigned long fib(unsigned int n){ if (n == 0 || n == 1) return n; else return fib(n - 1) + fib(n - 2); } int main(){ int n; printf("\n\nDigite um valor para n: "); scanf("%d”, &n); printf("\n F(%d) = %d \n ",n, fib(n)); return 0; }

Vamos introduzir o valor 5 para este programa.

São feitas as seguintes chamadas recursivas. Observe a estrutura upside-down (árvore de cabeça para baixo) criada pelas chamadas recursivas.

Fibonacci(5) / \ / \ / \ / \ / \ F(4) + F(3) / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ F(3) + F(2) F(2) + F(1) /\ /\ | \ \ / \ / \ | \ \ / \ / \ | \ \ / \ / \ | \ \ F(2) + F(1) F(1) + F(0) F(1) + F(0) 1 /\ | | | | | / \ | | | | | / \ | | | | | / \ | | | | | F(1) + F(0) 1 1 0 1 0 | | | | | | | | | 1 0

Cada vez que a sub-rotina chama a si mesmo, ela deve armazenar o estado atual da sub-rotina (linha atual que está sendo executada, os valores de todas as variáveis, etc) em uma estrutura de dados chamada de “pilha”.

Se você usar a recursividade durante um longo período de tempo, a pilha vai ficar muito grande e o programa dará uma mensagem de aviso.

17.9 inline

Uma função **inline**, em vez de ser chamada, será movida para o local de chamada no momento da compilação.

Se fizermos um paralelismo com as diretivas de compilação, como #define, ela vai substituir cada chamada da função pela própria função, é como fosse uma macro.

Mas isto só tem vantagens para códigos pequenos e para quem necessite muito da velocidade no processamento. Alguns compiladores já fazem isto automaticamente.

Para tornar uma função **inline** basta preceder a declaração da função com o nome **inline**.

inline [tipo_de_retorno] [nome_da_função] (argumentos) { //código }

Capítulo 18

Pré-processador

18.1 O pré-processador

O pré-processador C é um programa que examina o programa fonte escrito em C e executa certas modificações nele, baseado nas **diretivas de compilação** (ou *diretivas do pré-processador*). As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, executado pelo compilador antes da execução do processo de compilação propriamente dito.

Portanto, o pré-processador modifica o programa fonte, que ainda não estaria pronto para ser entregue ao compilador. Todas as diretivas de compilação são iniciadas pelo caractere # (sharp). As diretivas podem ser colocadas em qualquer parte do programa, mas não podem ser colocadas na mesma linha que outra diretiva ou instrução.

As principais diretivas de compilação são:

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif

18.2 Diretivas de compilação

18.2.1 #include

A diretiva **#include** diz ao pré-processador para incluir naquele ponto um arquivo especificado. Sua sintaxe é:

#include "*nome_do_arquivo*"

ou

#include <*nome_do_arquivo*>

A diferença entre se usar "" e <> é somente a ordem de procura nos diretórios pelo arquivo especificado. Se você quiser informar o nome do arquivo com o caminho completo, ou se o arquivo estiver no diretório de trabalho, use “arquivo”. Se o arquivo estiver nos caminhos de procura pré-especificados do compilador, isto é, se ele for um arquivo do próprio sistema (como é o caso de arquivos como *stdio.h*, *string.h*, etc...), use <arquivo>.

18.2.2 #define

A diretiva **#define** tem duas utilidades. Uma delas é apenas definir um símbolo que pode ser testado mais tarde. Outra é definir uma constante ou ainda uma macro com parâmetros. As três maneiras de usar a diretiva são:

#define **nome_do_símbolo** #define **nome_da_constante** *valor_da_constante* #define **nome_da_macro**(parâmetros) *expressão_de_substituição*

- Toda vez que o pré-processador encontrar *nome_da_constante* no código a ser compilado, ele deve substituí-lo por *valor_da_constante*.
- Toda vez que o pré-processador encontrar *nome_da_macro*(*parâmetros*), ele deve substituir por *expressão_de_substituição*, também substituindo os parâmetros encontrados na expressão de substituição; funciona mais ou menos como uma função. Veja o exemplo para entender melhor.

Exemplo 1:

#include <stdio.h> #define PI 3.1416 #define VERSAO “2.02” int main () { printf (“Programa versão %s\n”, VERSAO); printf (“O numero pi vale: %f\n”, PI); return 0; }

Exemplo 2:

#define max(A, B) ((A > B) ? (A) : (B)) #define min(A, B) ((A < B) ? (A) : (B)) ... x = max(i, j); y = min(t, r);

Aqui, a linha de código: x = max(i, j); será substituída pela linha: x = ((i) > (j) ? (i) : (j));. Ou seja, atribuiremos a *x* o maior valor entre *i* ou *j*.

Quando você utiliza a diretiva #define, nunca deve haver espaços em branco no identificador (o nome da macro). Por exemplo, a macro #define PRINT (i) printf(" %d \n", i) não funcionará corretamente porque existe um espaço em branco entre PRINT e (i).

18.2.3 #undef

A diretiva #undef tem a seguinte forma geral:

#undef **nome_da_macro**

Ela faz com que a macro que a segue seja apagada da tabela interna que guarda as macros. O compilador passa a partir deste ponto a não conhecer mais esta macro.

18.2.4 #ifdef e #ifndef

O pré-processador também tem estruturas condicionais. No entanto, como as diretivas são processadas antes de tudo, só podemos usar como condições expressões que envolvam constantes e símbolos do pré-processador. A estrutura **ifdef** é a mais simples delas:

#ifdef **nome_do_símbolo** *código ...* #endif

O código entre as duas diretivas só será *compilado* se o símbolo (ou constante) *nome_do_símbolo* já tiver sido definido. Há também a estrutura **ifndef**, que executa o código se o símbolo **não** tiver sido definido.

Lembre que o símbolo deve ter sido definido através da diretiva #define.

18.2.5 #if

A diretiva **#if** tem a seguinte forma geral:

#if **expressão** *código ...* #endif

A sequência de declarações será compilada apenas se a expressão fornecida for verdadeira. É muito importante ressaltar que a expressão fornecida não pode conter nenhuma variável, apenas valores constantes e símbolos do pré-processador.

18.2.6 #else

A diretiva **#else** funciona como na estrutura de bloco if (condição) {...} else {...}:

#if **expressão** /* ou #ifndef **expressão** */ *código* /* será executado se a expressão for verdadeira */ #else *código* /* será executado se a expressão for falsa */ #endif

Um exemplo:

#define WINDOWS ... /* código */ ... #ifdef WINDOWS #define CABECALHO “windows_io.h” #else #define CABECALHO “unix_io.h” #endif #include CABECALHO

18.2.7 #elif

A diretiva **#elif** serve para implementar uma estrutura do tipo if (condição) {...} else if (condição) {...}. Sua forma geral é:

#if expressão_1 *código* #elif expressão_2 *código* #elif expressão_3 *código* . . . #elif expressão_n *código* #endif

Podemos também misturar diretivas #elif com #else; obviamente, só devemos usar uma diretiva #else e ela deve ser a última (antes de #endif).

18.3 Usos comuns das diretivas

Um uso muito comum das diretivas de compilação é em arquivos-cabeçalho, que só precisam/devem ser incluídos uma vez. Muitas vezes incluímos indiretamente um arquivo várias vezes, pois muitos cabeçalhos dependem de outros cabeçalhos. Para evitar problemas, costuma-se envolver o arquivo inteiro com um bloco condicional que só será compilado se o arquivo já não tiver incluído. Para isso usamos um símbolo baseado no nome do arquivo. Por exemplo, se nosso arquivo se chama “cabecalho.h”, é comum usar um símbolo com o nome CABECALHO_H:

#ifndef CABECALHO_H #define CABECALHO_H . . . #endif

Se o arquivo ainda não tiver sido incluído, ao chegar na primeira linha do arquivo, o pré-processador não encontrará o símbolo CABECALHO_H, e continuará a ler o arquivo, o que lhe fará definir o símbolo. Se tentarmos incluir novamente o arquivo, o pré-processador pulará todo o conteúdo pois o símbolo já foi definido.

18.4 Concatenação

O preprocessador C oferece duas possibilidades para manipular uma cadeia de caracteres .
A primeira é usando o operador # que permite substituir a grafia de um parâmetro .

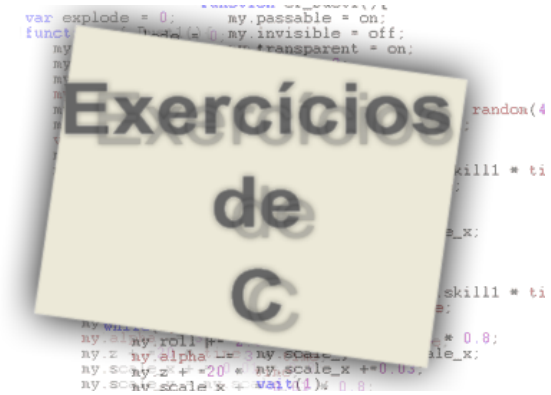
#include<stdio.h> int main (void) { /* mad equivale a “mad” */ #define String(mad) #mad printf (String(Estou aqui) "\n”); }

A segunda é usando o operador ## que serve para concatenar vários parâmetros .
Ex: ban##ana é igual a banana .

#include<stdio.h> int main (void) { int teste = 1000 ; #define CONCAT(x, y) x##y /* igual a “tes” + “te” */ printf (" %i \n”, CONCAT (tes, te)); }

Capítulo 19

Exercícios



Exercícios

19.1 Questões

O que faz o seguinte programa?

#include <stdio.h> int main() { int x = 4; scanf("%d", &x); printf("%d", 2*x); getchar(); }

19.2 Escrevendo programas

19.2.1 Exercício 1

Escreva uma função que peça dois números inteiros ao usuário e exibe o valor soma seguido pelo maior deles.

19.2.2 Exercício 2

Faça um programa que converta Celsius para Fahrenheit.

#include <stdio.h> #include <stdlib.h> /* By : forshaw */ int main() { int GrausCel, GrausFah; printf("Digite a temperatura em gruas Celsius => "); scanf("%d", &GrausCel); GrausFah = 1.8 * GrausCel + 32; printf("\nA temperatura em graus Fahrenheit eh <%d>\n\n", GrausFah); return 0; }

19.2.3 Exercício 3

Faça um programa que vai lendo cada caractere que o usuário digitar. Quando o usuário digitar o caractere 'x', o programa deve exibir todos os caracteres que foram digitados antes do 'x'.

#include <stdio.h> #include <stdlib.h> int main() { int ch; ch = getchar(); while(ch != 'x') { if(ch != '\n') { printf("O caractere era %c, e o valor %d\n", ch, ch); ch = getchar(); } } return 0; }
#include <stdio.h> #include <stdlib.h> /* By : forshaw Versão sem getchar(), apenas com scanf(), por um mundo mais didático. (: */ int main() { int x, j; char palavra[100]; printf("Digite seus caracteres\n\n"); for (x=0; x<100; x++) { scanf("%c", &palavra[x]); if (palavra[x] == 'x') { printf("\n==========
printf("Voce teclou: \n\n"); for (j=0; j<x; j++) { printf("%c", palavra[j]); } printf("\n\n"); break; } } return 0; }

19.2.4 Exercício 4

Escreva um programa que começa pedindo um número N ao usuário e depois pede N números. O programa deverá mostrar na tela todos esses números ordenados do menor para o maior. Escreva sua própria função de ordenação.

#include <stdio.h> #include <stdlib.h> /* By : forshaw */ int main() { int n, *numeros; int x, y, temp; printf("Digite um numero: \n"); scanf("%d", &n); numeros = (int *) malloc(n * sizeof(int)); /* Isso simplesmente cria um array de tamanho n (int numeros[n]) */ printf("Agora digite %d numeros: \n", n); for (x=0; x<n; x++) scanf("%d", &numeros[x]); if (x==n) /* Isso e, quando acabar o for de cima */ for (x=0; x<n; x++) for (y=0; y<n; y++) /* Varre todo o array[y] comparando-o com um array[x] constante (por vez) */ if (numeros[x] < numeros[y]) { temp = numeros[x]; numeros[x] = numeros[y]; /* Aqui e a velha troca de dados :) */ numeros[y] = temp; } if (x==n) /* Isso e, quando acabar o for de cima */ printf("Agora em ordem crescente: \n"); for (x=0; x<n; x++) printf("%d\n", numeros[x]); return 0; }

19.2.5 Exercício 5

Faça um programa que encontra a raiz quadrada aproximada de um número. Para isso, use um dos métodos existentes.

#include <stdio.h> #include <stdlib.h> /* By : forshaw */ int main() { int num, i, result, j; printf(“Digite o numero que queres a raiz => "); scanf("%d”, &num); j = num; result = 0; i = 1; while (num >= i){ num -= i; i += 2; result++; } if (num == 0) printf("\nA raiz exata de %d eh >%d<\n\n”, j, result); else printf("\nA raiz aproximada de %d eh >%d<\n\n”, j, result); }

19.2.6 Exercício 6

O código abaixo gera números primos.

#include <stdio.h> #include <stdlib.h> /* By : forshaw */ int main() { int limite; int x, y; printf(“Digite o limite do gerador : "); scanf("%d”, &limite); if (limite >= 2) /* Para sempre incluir o 2, ja que e o unico par primo */ printf(“2\n”); for (x=1; x<=limite; x++) for (y=2; y<x; y++) if (x%y == 0) /* Testa se e divisivel por algum anterior, e caso ser verdadeiro, sair da condição (continue;) */ y = x+1; else if (y == x-1) /* No final da condição (ja que um numero n > 2 não é divisivel por n-1) */ { printf("%d\n”, x); y = x+1; /* Termina para garantir um unico printf() */ } return 0; }

Código simples e fácil de entender:

// Por Warley V. Barbosa #include <stdio.h> int main() { int num, primo, i, res; do { printf(“Digite um número (0 para encerrar): \n”); scanf("%d”, &num); primo = 1; for (i = 2; i < num; i++) /* repete a partir do número dois e vai até o número anterior de 'num', incrementando à variável i a cada repetição */ { if (num % i == 0) { // se o resto do 'num' por 'i' for zero o número não é primo primo = 0; break; // pula para a instrução seguinte } } if ((primo) && (num > 1)) // 1 não é primo! nem zero... printf(“O número %d é primo! \n”, num); else printf(“O número %d não é primo... \n”, num); } while (num != 0); // encerra quando usuário digitar 0 }

Partindo deste código, tente criar um que some o último primo resultante com o anterior.

19.2.7 Exercício 7

Faça uma calculadora:

#include <stdio.h> #include <stdlib.h> int expr(void); int mul_exp(void); int unary_exp(void); int primary(void); int main() { int val; for(;;) { printf(“expression: "); val = expr(); if(getchar() != '\n') { printf(“error\n”); while(getchar() != '\n'); /* nada */ } else { printf(“result is %d\n”, val); } } return 0; } int expr(void) { int val, ch_in; val = mul_exp(); for(;;) { switch(ch_in = getchar()) { default: ungetc(ch_in,stdin); return(val); case '+': val = val + mul_exp(); break; case '-': val = val - mul_exp(); break; } } } int mul_exp(void) { int val, ch_in; val = unary_exp(); for(;;) { switch(ch_in = getchar()) { default: ungetc(ch_in, stdin); return(val); case '*': val = val * unary_exp(); break; case '/': val = val / unary_exp(); break; case '%': val = val % unary_exp(); break; } } } int unary_exp(void) { int val, ch_in; switch(ch_in = getchar()) { default: ungetc(ch_in, stdin); val = primary(); break; case '+': val = unary_exp(); break; case '-': val = -unary_exp(); break; } return(val); } int primary(void) { int val, ch_in; ch_in = getchar(); if(ch_in >= '0' && ch_in <= '9') { val = ch_in - '0'; goto out; } if(ch_in == '(') { val = expr(); getchar(); /* skip closing ')' */ goto out; } printf(“error: primary read %d\n”, ch_in); exit(EXIT_FAILURE); out: return(val); }

19.2.8 Exercício 8

Esse programa converte um inteiro em bytes e depois realiza a operação inversa.
Faça um programa que converte um short em bytes.

#include <stdio.h> int main (void) { unsigned char bytes[4]; /* Aqui o índice indica o número de elementos */ unsigned long n = 123000; bytes[0] = (n >> 24) & 0xFF; bytes[1] = (n >> 16) & 0xFF; bytes[2] = (n >> 8) & 0xFF; bytes[3] = n & 0xFF; int i; char b[500]; sprintf(b,"0 = %d, 1 = %d, 2 = %d, 3 = %d”,bytes[0],bytes[1],bytes[2],bytes[3]); int inteiro = (bytes[0]<<24)+(bytes[1] << 16)+(bytes[2] << 8)+bytes[3]; printf(“bytes %s\n”,b); printf(“int = %i\n”,inteiro); getchar(); }

Capítulo 20

Vetores

20.1 Vetores

Vetores, também chamados ***arrays*** (do inglês) ou arranjo ou ainda **matrizes**, são uma maneira de armazenar vários dados num mesmo nome de variável através do uso de índices numéricos. Em C, vetores devem sempre conter dados do mesmo tipo de variável.

Declaramos vetores de maneira muito semelhante à declaração de variáveis normais. A única diferença é que depois do nome da variável deve ser informada **a quantidade de elementos** do vetor. Para declarar um vetor chamado vetor, com cinco elementos inteiros, escrevemos:

int vetor[5];

Note que a quantidade de elementos de um vetor não pode ser alterada depois que o vetor for declarado. Para criar vetores de tamanho dinâmico, podemos usar ponteiros, que serão abordados mais adiante.

Da mesma maneira que podemos inicializar uma variável junto com sua declaração, podemos usar as chaves ({}) para inicializar um array.

int vetor[5] = {17, 42, 9, 33, 12};

Para fazer referência a um valor de um elemento contido em um vetor, usamos a notação vetor[índice], que serve tanto para obter quanto para definir o valor de um elemento específico, dada sua posição. Note que os elementos são numerados a começar do zero, e, portanto, se o número de elementos é N , o índice ou posição do último elemento será $N - 1$.

vetor[0] = 3; int x = vetor[2]; int y = vetor[5]; // ERRO!

Repare em que a última linha contém um erro: ela referencia um elemento do vetor que não existe. No entanto, o compilador não se recusará a compilar esse código; dará apenas um aviso. Se essa linha for executada, a variável y receberá um valor que não tem nada a ver com o vetor.

20.1.1 Abreviando as declarações

Ao inicializar um vetor com vários valores, pode ser trabalhoso contar todos os valores para colocar o tamanho do vetor na declaração. Por isso, em C podemos omitir o número de elementos quando os valores são inicializados; o tamanho do vetor será o número de valores inicializados. Por exemplo, as duas notações abaixo são equivalentes:

int valores[5] = {1, 2, 3, 4, 5}; int valores[] = {1, 2, 3, 4, 5};

20.1.2 Exemplo de Aplicação de Vetores

O código abaixo é de um programa que recebe 5 números inteiros e informa qual destes é maior.

#include <stdio.h> #include <stdlib.h> #include <conio.h> int main(void) { int vetor[5]; int x, i; printf (“digite 5 numeros\n”); for (i = 0; i < 5; i++) /*Este laco faz o scan de cada elemento do vetor*/ { scanf("%d”, &vetor[i]); } i = 1; x = vetor[0]; while (i < 5) /*Este laco compara cada elemento do vetor*/ { if (vetor[i] > x) { x = vetor[i]; } i++; } printf("\n O maior numero que voce digitou foi %d .\n”,x); getch (); return 0; }

20.2 Vetores multidimensionais (matrizes)

Podemos declarar ainda mais variáveis

tipo_da_variável nome_da_variável [altura][largura];

Atenção que:

- Índice mais à direita varia mais rapidamente que o índice à esquerda.
- Não esquecer os índices variam de zero ao valor declarado menos um.

Podemos ter ainda conjunto de variáveis multidimensionais.

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

onde a iniciação é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN] = {lista_de_valores}; float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 }; int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }; char str [10] = { 'J', 'o', 'a', 'o', '\0' }; char str [10] = “Joao"; char str_vect [3][10] = { “Joao”, “Maria”, “Jose” }; int matrx[2][4]= { { 1,2,3,4 }, { 5,6,7,8 } };

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho a priori. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa

Uma tabela de inteiros positivos de duas dimensão (3 linhas, 4 colunas) se definiria da seguinte forma:

int Tabela [3][4]

Suponha que o primeiro índice é o índice da linha e o segundo da coluna .
Então teríamos:

Exemplo da inicialização de um vetor multidimensional usando loops.

```
int i,j;
for (i=0; i<=2; i++){
for (j=0; j<=3; j++){
Tabela[i][j] = 0;
}
}
```

20.3 Argumentos na função main

Na seção Funções, dissemos que a função main() aceita dois argumentos, mas não falamos mais pois um deles envolve o conceito de vetores. Agora iremos falar mais sobre eles.

Os dois argumentos de main() indicam os argumentos dados para o programa na linha de comando, na forma de um vetor de strings.

20.3.1 Exemplo de uso de parâmetros na função main

// somaComArgcArgv.c #include<stdio.h> #include<stdlib.h> int main(int argc, char *argv[]) { int result; if (argc != 3) { printf(“Digite soma <valor1> <valor2>\n”); return 0; } // fim if (argc != 3) result = atoi(argv[1]) + atoi(argv[2]); printf("\nO resultado da soma de %s e %s eh: %d\n”, argv[1], argv[2], result); }

Capítulo 21

Strings

21.1 Strings

Strings (Inglês) são cadeias ou seqüências ordenadas de caracteres. Na verdade já trabalhamos com strings neste livro, mas preferimos deixar maiores explicações para um momento em que já tivesse sido introduzido o conceito de vetor.

A linguagem C, ao contrário de outras linguagens de programação, não possui um tipo de dados correspondente às strings; no lugar, usam-se vetores (e ponteiros, como veremos mais adiante). Em C, strings são vetores de caracteres terminados pelo caractere nulo ('\0'). Por exemplo:

char nome[] = {'P', 'e', 'd', 'r', 'o', '\0'};

No entanto, escrever strings dessa maneira é muito trabalhoso; por isso, foi criada uma notação abreviada que equivale à notação acima e elimina a necessidade de colocar o caractere terminador:

char nome[] = “Pedro";

Assim como nos vetores, podemos acessar e modificar elementos individuais de uma string. Podemos também diminuir o tamanho de uma string: uma vez que a única marcação do tamanho é o terminador \0, colocar um terminador em outro local determinará o novo final da string. No entanto, aumentar o tamanho da string é mais difícil; isso ficará para outra seção.

Atenção ao usar-se **acentos** numa string. Como existem diferentes formas de codificar caracteres acentuados, o tratamento de uma string do tipo:

char nome[] = “João";

pode ser diferente de uma máquina para outra. Neste capítulo não serão tratados acentos, este assunto será abordado mais adiante.

21.2 Funções da biblioteca padrão

A biblioteca padrão fornece várias funções úteis para manipular strings. A seguir mostraremos algumas delas. Para usá-las, você deve incluir o cabeçalho string.h no início dos seus arquivos.

21.2.1 strlen

strlen retorna o tamanho, em caracteres, de uma string dada. Na verdade o strlen() procura o terminador de string e calcula a distância dele ao início da string. Por exemplo:

char nome[15] = “Maria da Silva"; int s = strlen (nome); // s conterá o valor 14

21.2.2 strcpy

strcpy copia o conteúdo de uma string para outra e coloca um terminador de string. Sua sintaxe é strcpy (*destino*, *origem*).

char nome[] = “Clarice Lispector"; char nome2[] = “Oswald de Andrade"; strcpy (nome, nome2); // agora nome conterá “Oswald de Andrade”

Tome cuidado com strcpy(), pois se a string a ser copiada for maior que a string de destino, provavelmente você gravará dados em lugares indesejados — um problema conhecido como **estouro de buffer**. Para evitar esse problema, use a função **strncpy**, que recebe um terceiro argumento que corresponde ao número máximo de caracteres a serem copiados:

char msg[] = “Bom dia!"; char nome[] = “Maria da Silva"; strncpy (msg, nome, strlen(msg)); // agora msg conterá “Maria da”

21.2.3 strcat

strcat concatena duas strings, adicionando o conteúdo da segunda ao final da primeira, além do terminador (\0). Note que a primeira string deve ter espaço suficiente para conter a segunda, para que não ocorra um “estouro de buffer”. Por exemplo:

char nome[50] = “Maria"; char sobrenome[] = " da Silva"; strcat (nome, sobrenome); // agora nome contém “Maria da Silva”

Analogamente à função strncpy, existe também a função **strncat**, onde o número máximo de caracteres a serem copiados é o terceiro argumento.

21.2.4 strcmp

Se você tentar criar duas strings com o mesmo conteúdo e compará-las como faria como números, verá que elas "não são iguais". Isso ocorre porque, na verdade, o que está sendo comparado são *os endereços de memória* onde estão guardadas as strings. Para comparar *o conteúdo* de duas strings, você deve usar a função **strcmp** (ou suas variantes):

int **strcmp** (char **s1*, char **s2*);

O valor de retorno é:

- menor que zero se *s1* for menor que *s2*;
- igual a zero se *s1* e *s2* são iguais;
- maior que zero se *s1* for maior que *s2*.

Costuma parecer estranho dizer que uma string é *menor* ou *maior* que outra; na verdade essa comparação é entre a primeira letra que difere nas duas strings. Assim, se tivermos s1 = "abc" e s2 = "abd", diremos que *s2* é **maior** que *s1*, pois na primeira posição em que as duas strings diferem, a letra em *s2* é "maior".

É importante notar que a comparação feita por strcmp distingue maiúsculas de minúsculas. Isto é, as strings "ABC" e "abc" **não** são iguais para essa função.

As variantes mais usadas de strcmp são:

- **strncmp** - compara apenas os *n* primeiros caracteres das duas strings, sendo *n* um terceiro argumento.
- **stricmp** - compara duas strings sem distinção entre maiúsculas e minúsculas. A sintaxe é igual à de strcmp. Essa função não faz parte da biblioteca padrão, mas é comumente encontrada como extensão particular de várias delas.

21.2.5 strrchr

strrchr Retorna um ponteiro sobre a ultima ocorrência de c de uma string apontada por s se não retorna NULL . Sua sintaxe é strrchr(const char *s, int c);.
Exemplo:

char path[50] = "/teste/string"; char *p = strrchr(path, '/'); *p++; printf("Resultado: %s \n", p);

21.2.6 memcpy

Sintaxe:

#include <string.h> void *memcpy (void *dest, const void *srce, size_t n);

Descrição: Copiar um bloco de n octetos de srce para dest.

Atenção:Se as regiões de srce e dest se sobreporem o comportamento da função é imprevisível.

Valor de retorno : memcpy retorna o valor de dest .

Ex:

#include <stdio.h> #include <string.h> int main() { int tab[2][5] = { { 1, 2, 3, 4, 5}, {11, 12, 13, 14, 15} }; int temp[2][5]; memcpy(temp, tab, sizeof(tab)); puts("Resultado:\n"); printf("temp[1][4] = %d\n", temp[1][4]); return 0; }

21.2.7 memset

Sintaxe:

#include <string.h> void *memset (void *buffer, int c, size_t n);

Descrição: memset inicializa n octetos do buffer com o inteiro c.

Valor de retorno : O valor do buffer.

Ex:

#include <stdio.h> #include <string.h> int main() { char buf[] = "W.I.K.I."; printf("Buf antes 'memset': %s\n", buf); memset(buf, '*', strlen(buf)); printf("Buf depois 'memset': %s\n", buf); return 0; }

21.2.8 sprintf

Descrição: A diferença entre printf e sprintf e que printf retorna o resultado para a saída padrão (tela), enquanto sprintf retorna o resultado em uma variável. Isto é muito conveniente, porque você pode simplesmente digitar a frase que você quer ter e sprintf lida com a própria conversão e coloca o resultado na string que você deseja.

Sintaxe:

#include <stdio.h> int sprintf(char *s, const char *formato, ...);

Ex:

#include <stdio.h> #include <string.h> int main() { char var[256]; char sobrenome[] = "Simpson"; char nome[] = "Homer"; int idade = 30; sprintf(var, "%s %s tem %d anos",sobrenome, nome, idade); printf ("Resultado : %s\n", var); return 0; }

Capítulo 22

Passagem de parâmetros

22.1 Passagem de Parâmetros

Esta explicação é para quem compila com o GNU gcc.
O que são parâmetros?
Com os programas em interface gráfica usa-se botões ou ícones.
Quando utiliza-se os parâmetros com o console ou prompt os parâmetros são reconhecidos como opções.
Para quem usa sistemas do tipo Unix como o Linux, onde o console não é banalizado como em outros SO's, é mais fácil de se entender.
Imagine que exista um programa cujo nome é "Calcular" e que ele serve para executar operações aritméticas.
Pense agora na sua execução no shell.

$./Calcular restar

"Calcular" é o nome, a "chamada" ao seu programa, enquanto que "restar" é um parâmetro, uma opção.
Esse programa pode comportar vários parâmetros como somar, subtrair e multiplicar, por exemplo.

Exemplo:

/*----------Parâmetros.c----------*/ #include <stdio.h> /* É igual a int main(int argc, char *argv[]) */ int main(int argument_count, char *argument_value[]) { int i; printf("Nome do Programa :%s\n", argument_value[0]); for (i = 1 ; i < argument_count; i++) printf("Parâmetros passados %d : %s\n", i, argument_value[i]); /* De um enter no fim*/ }

Para compilar:

user@SO:/meu_diretotio\protect\char"0024\relax gcc Parametros.c -o Argumentos

Como diríamos é só passar alguns argumentos para o compilador ;)

Examinando o código

Vamos dar uma olhada na função main(int argc, char *argv[]) vocês podem remarcar os nomes:

- argc "argument count" : Conta o numero de argumentos incluindo o nome do programa.
- E argv "arguments values" : Vamos dizer que cada argv[] é um nome de parâmetro.
- Então temos um que nos da a quantidade de parâmetros e outro que nos da os nomes de cada parâmetro ou opção.

São nomes tradicionais eles podem ser modificado para outros nomes desde que os tipos continuem sendo os mesmos.

Exe:

NOME opçao1 opçao2 opçao3 : argc = 4 $./Calcular somar depois restar : argv[] vai de argv[0] a argv[3]

Aqui argc é igual a 4.

argv[] é na realidade uma tabela de ponteiros exe:

argv[0] e igual a '.' '/' 'C' 'a' 'l' 'c' 'u' 'l ' 'a' 'r' '/0' Calcular

argv[1] e igual a 's' 'o' 'm' 'a' 'r' '/0' somar

Capítulo 23

Tipos de dados definidos pelo usuário

23.1 Tipos de dados definidos pelo usuário

Muitas vezes é necessário manipular dados complexos que seriam difíceis de representar usando apenas os tipos primitivos (char, int, double, float). Para isso, há, em C, três tipos de dados que podem ser definidos pelo usuário:

- estruturas (*struct*);
- uniões (*union*);
- enumerações (*enum*).

As estruturas e uniões são compostas por várias variáveis (escolhidas pelo programador), por isso são ditos *definidos pelo usuário*. Já as enumerações são, resumidamente, tipos cujos valores devem pertencer a um conjunto definido pelo programador.

23.2 Estruturas

Uma **estrutura** (ou *struct*) é um tipo de dados resultante do agrupamento de várias variáveis nomeadas, não necessariamente similares, numa só; essas variáveis são chamadas *membros* da estrutura. Para declarar uma estrutura, usamos a palavra-chave **struct**, seguida do nome que se deseja dar à estrutura (ao tipo de dados) e de um bloco contendo as declarações dos membros. Veja um exemplo:

struct mystruct { int a, b, c; double d, e, f; char string[25]; };

Este exemplo cria um tipo de dados denominado *mystruct*, contendo sete membros (*a*, *b*, *c*, *d*, *e*, *f*, *string*). Note que o nome *mystruct* é o nome do tipo de dados, não de uma variável desse tipo.

Um exemplo simples de aplicação de estruturas seria uma ficha pessoal que tenha nome, telefone e endereço; a ficha seria uma estrutura.

Ou, mais amplamente, uma estrutura seria uma representação de qualquer tipo de dado definido por mais de uma variável. Por exemplo, o tipo FILE* é na verdade um ponteiro para uma estrutura que contém alguns dados que o sistema usa para controlar o acesso ao fluxo/arquivo. Não é necessário, para a maioria dos programadores, conhecer a estrutura do tipo FILE.

23.2.1 Definindo o tipo

A definição de um tipo de estrutura é feita com a palavra-chave **struct**, seguida do nome a ser dado ao tipo e de um bloco contendo as declarações dos elementos da estrutura:

struct nome_do_tipo { tipo_elem a; tipo_elem b, c; ... };

É muito importante incluir o ponto-e-vírgula ao final do bloco!

23.2.2 Declarando

Para declarar uma variável de um tipo já definido, fornecemos o nome do tipo, **incluindo a palavra-chave struct**:

struct nome_do_tipo *variavel*;

Também é possível condensar a definição do tipo e a declaração em um passo, substituindo o nome do tipo pela definição, sem o ponto-e-vírgula:

struct mystruct { int a, b, c; double d, e, f; char string[25]; } variavel;

Também é possível inicializar uma estrutura usando as chaves {} para envolver os elementos da estrutura, separados por vírgulas. Os elementos devem estar na ordem em que foram declarados, mas não é obrigatório inicializar todos; no entanto, para inicializar um elemento, todos os anteriores devem ser inicializados também. Por exemplo, poderíamos declarar valores iniciais para a variável acima da seguinte maneira:

struct mystruct variavel = {4, 6, 5, 3.14, 2.718, 0.95, “Teste"}; struct mystruct v2 = {9, 5, 7};

23.2.3 Inicializador designado

Para quem usa o C99 com o compilador GNU. Durante a inicialização de um estrutura é possível especificar o nome do campo com '.nome_do_campo =' antes do valor. Exemplo:

struct mystruct v2 = {.a=9,.b=5,.c=7};

23.2.4 Acessando

Para acessar e modificar os membros de uma estrutura, usamos o operador de seleção. (ponto). À esquerda do ponto deve estar o nome da variável (estrutura) e à direita, o nome do membro. Podemos usar os membros como variáveis normais, inclusive passando-os para funções como argumentos:

variavel.a = 5; variavel.f = 6.17; strcpy (variavel.string, “Bom dia”); printf ("%d %f %s\n”, variavel.a, variavel.f, variavel.string);

23.2.5 Vetores de estruturas

Sendo as estruturas como qualquer outro tipo de dados, podemos criar vetores de estruturas. Por exemplo, suponha algum programa que funcione como um servidor e permita até 10 usuários conectados simultaneamente. Poderíamos guardar as informações desses usuários num vetor de 10 estruturas:

struct info_usuario { int id; char nome[20]; long endereco_ip; time_t hora_conexao; }; struct info_usuario usuarios[10];

E, por exemplo, para obter o horário em que o 2º usuário usuário se conectou, poderíamos escrever usuarios[1].hora_conexao.

23.2.6 Atribuição e cópia

Podemos facilmente copiar todos os campos de uma estrutura para outra, fazendo uma atribuição simples como a de inteiros:

struct ponto { int x; int y; }; ... struct ponto a = {2, 3}; struct ponto b = {5, 8}; b = a; // agora o ponto *b* também tem coordenadas (2, 3)

No entanto, devemos ter cuidado se a estrutura contiver campos ponteiros, pois, nesses casos, o que será copiado é o endereço de memória (e não o conteúdo daquele endereço). Por exemplo, se tivermos uma estrutura que comporta um inteiro e uma string, uma cópia sua conterá o mesmo inteiro e **um ponteiro para a mesma string**, o que significa que alterações na string da cópia serão refletidas também no original!

23.2.7 Passando para funções

Já vimos acima que podemos normalmente passar membros de uma estrutura como argumentos de funções. Também é possível passar estruturas inteiras como argumentos:

#include <stdio.h> struct ponto { int x; int y; }; void imprime_ponto (struct ponto p) { printf ("(%d, %d)\n”, p.x, p.y); } int main () { struct ponto a = {3, 7}; imprime_ponto (a); return 0; }

No entanto, há dois possíveis problemas nisso:

- Alterações nos membros da estrutura só terão efeito dentro da função chamada, mas não na função que a chamou. Isso ocorre pois a estrutura é passada por valor (e não por referência).
- Quando a estrutura contiver muitos elementos, a passagem por valor tornar-se-á um processo de cópia de muitos dados. Por isso, é de costume passar estruturas por referência (como ponteiros), mesmo que a estrutura em questão seja pequena.

23.3 Uniões

Uniões são parecidas com estruturas, mas há uma diferença fundamental: nas uniões, todos os elementos ocupam o mesmo espaço de memória. Por isso, só é possível acessar um elemento por vez, já que uma mudança em um elemento causará mudança em todos os outros. A definição e a declaração de uniões é igual à das estruturas, trocando a palavra struct por **union**.

Há principalmente dois usos para as uniões:

- **economia de espaço**, já que guardam-se várias variáveis no mesmo espaço;
- **representação de uma informação de mais de uma maneira**. Um exemplo disso são os endereços IP, que na biblioteca de sockets podem ser representados como um grupo de 4 octetos (char) ou como um único valor inteiro (int). Isso é feito com uma união parecida com esta: union ip_address { int s_long; char s_byte[4]; }; Dessa maneira, o endereço pode ser facilmente representado de maneira humanamente legível (com 4 octetos), sem dificultar o processamento interno (com o valor inteiro).

23.4 Enumerações

Enumeração (*enum*) ou **tipo enumerado** é um tipo de dados que tem como conjunto de valores possíveis um conjunto finito de identificadores (nomes) determinados pelo programador. Em C, cada identificador em uma enumeração corresponde a um inteiro.

Enumerações são definidas de maneira similar às estruturas e uniões, com algumas diferenças. A palavra chave usada é **enum**.

enum nome_enumeração { IDENTIFICADOR_1, IDENTIFICADOR_2, ... IDENTIFICADOR_n };

Note as diferenças: não há ponto-e-vírgula no final ou no meio das declarações (mas ainda há no final do bloco), e não há declaração de tipos.

Com essa declaração, ao IDENTIFICADOR_1 será atribuido o valor 0, ao IDENTIFICADOR_2 será atribuído o valor 1, e assim por diante. Podemos também explicitar os valores que quisermos colocando um sinal de igual e o valor desejado após o identificador.

- Caso não haja valor determinado para o primeiro identificador, ele será zero. Para os demais identificadores, o padrão é seguir a ordem dos números, a partir do valor do identificador anterior.
- Podemos misturar identificadores de valor determinado com identificadores de valor implícito, bastando seguir a regra acima.

Por exemplo:

enum cores { VERMELHO, /* *0* */ AZUL = 5, /* *5* */ VERDE, /* *6* */ AMARELO, /* *7* */ MARROM = 10 /* *10* */ };

23.4.1 Uso

Da mesma maneira que criamos uma variável de um tipo struct ou union, podemos criar variáveis de um tipo enumerado (enum):

enum cores cor_fundo;

Para atribuir valores a uma variável enumerada, podemos usar como valor tanto o identificador quanto o valor correspondente. Seriam equivalentes, portanto:

cor_fundo = VERDE; cor_fundo = 6;

Na verdade, variáveis enumeradas agem de maneira quase igual aos inteiros; é possível, assim, atribuir valores que não correspondem a nenhum dos identificadores.

23.5 Campo de bits

Na linguagem c o campo de bits (bitfields) é uma estrutura um pouco estranha , em vez de usar variáveis com tipos diferentes os campos são formados com as partes de um inteiro. O tamanho de um campo de bits não pode ser maior que o tipo usado , aqui um short .

typedef struct { unsigned short campo_1: 6, /* Tamanho 6 bit */ campo_2: 6, campo_3: 1, campo_4: 1, campo_5: 2; }BIT_FIELD_1;

Essa estrutura esta formada por um tipo que tem o tamanho de um short esse mesmo tipo será divido em porções menores. No exemplo acima os campos tem os tamanhos 6,6,1,1,2 igual a 16 bits que é o tamanho de um unsigned short . Para acessar os campos usamos o mesmo método que usamos com estruturas normais .

BIT_FIELD_1 meu_campo; meu_campo.campo_1 = 16; meu_campo.campo_4 = 0;

Capítulo 24

Enumeração

24.1 Enumerations (enum)

Aqui vamos retornar a um tópico antigo.

Enumerations são um outro método de definir constantes. Recordam-se? Tínhamos o:

1. #define
2. const int a = 1;
3. enumerations.

24.2 Criando um novo tipo de dados

As enumerations definem um nova tipo de variável e limita desde logo os valores.

enum colors {black, blue, green, cyan, red, purple, yellow, white};

A maneira mais simples de interpretar uma enumeration é imagina-la como uma matriz de apenas uma linha. Temos o nome da linha de temos as várias células na linha. Cada constante enumerada (muitas vezes chamado de enumerator) tem um valor inteiro (**caso não seja especificado ele começa em zero**)

Exemplo:

Mas podemos definir o valor tipo

enum forma {quadrado=5, rectangulo,triangulo=27, circulo, elipse}

ficaríamos com a nossa linha do tipo:

reparem nos valores dos números.

A vantagem em termos enumerações é que se uma variável é declarada tipo enumeração, tem um tipo único e os seus valores estão limitados e poderão ser verificados durante a compilação.

É tal como as estruturas criar tipos de variáveis.

#include <stdio.h> /*Definindo o cabeçalho*/ enum cores { AZUL = 1, VERDE, BRANCO, }; /*Aqui um ponto virgula*/ /*typedef transformamos 2 palavras em uma -> tipo_cores*/ typedef enum cores tipo_cores ; /*A função default da lib ou glibc*/ int main(void) { /*Agora usando o nosso novo tipo * Aqui sem typedef teríamos que colocar enum cores */ tipo_cores cor = VERDE ; if(cor == 1) { printf("Cor azul \n"); } if(cor == 2) { printf("Cor verde \n"); } /* printf não será executado */ if(cor == 3) { printf("Cor branco \n"); } return 0 ; /*De um enter depois de } para evitar warning */ }

Aqui podemos ver um exemplo com uma função "mostrarRes()" e um switch:
Em este exemplo uma constante e definida e o valor das outra será definido automaticamente.

#include <stdio.h> #include <stdlib.h> void mostrarRes(int quem); /*Aqui os valores Italia = 4 e Brasil = 5 são incrementados automaticamente*/ enum { ARGENTINA = 3, ITALIA, BRASIL }; int main(void) { /*Colocamos 5 se você for Argentino coloque 3 */ int n = BRASIL ; mostrarRes(n); } void mostrarRes(int quem) { switch(quem) { case BRASIL : printf("Brasil invencível como de costume\n"); break; case ARGENTINA : printf("Argentina um dia quem sabe\n") ; break; case ITALIA : printf("Foi sorte\n") ; break; default : printf("Se estou vivo teve erro do sistema xx \n") ; } printf("The end , hasta la vista\n \n"); /*De um enter depois de } para evitar warning */ }

Capítulo 25

União

25.1 Unions

As unions são muito parecidas com as estruturas, estas guardam variáveis de vários tipos, e portanto guardam cada variável de acordo com a seu tipo, ie, se tivermos uma variável membro que é um int e outro float, ela guarda exatamente de acordo com esse tipo.

O que se passa aqui é que vai guardar as variáveis todas com um único tipo, que é aquele que ocupa mais espaço dentro dos tipos das variáveis membro, ou seja, se tivermos uma variável membro int e outra float , a union vai guardar estas variáveis como fossem as duas float.

25.2 Declaração

union mytypes_t { int i; float f; } mytypes;

25.3 Unions com estruturas

Neste exemplo temos unions e estruturas misturados.

union mix_t { long l; struct { short hi; short lo; } s; char c[4]; } mix;

Repare que a estrutura não tem nome

25.4 Anonymous unions – estruturas com unions

// estrutura usando “regular union” struct { char title[50]; char author[50]; union { float dollars; int yens; } price; } book; // estrutura usando “anonymous union” struct { char title[50]; char author[50]; union { float dollars; int yens; }; } book;

Se declararmos uma união sem nome, ela vai ficar anônima e poderemos acessar seus membros diretamente através dos nomes dos membros.

// Uso regular book.price.dollars book.price.yens // Uso anonimo book.dollars book.yens

Capítulo 26

Estruturas

26.1 Estruturas

As estruturas (structs) permitem com que possamos ter variáveis de vários tipos aglomerados sob o mesmo nome. E esse mesmo nome vai passar a ser um novo tipo de dados tal como o int ou float.

Mas o uso disto é que podemos ter valores que tenham alguma relação lógica, por exemplo guardar um int de idade e um string de nome. Isto pode ser atributos de uma pessoa. Ou seja podemos empacotar várias variáveis de vários tipos com o objetivo de representar o mundo real e dar um nome a essas variáveis todas.

Ao fazer isto criamos um tipo de dados da mesma forma como fazemos em relação ao int ou ao float.

26.2 Declarar uma estrutura

A sintaxe é:

struct <identificador> { <tipo> campo_um ; <tipo> campo_dois ; };

Aqui o tipo struct indica que vamos criar uma estrutura. O nome ou identificador pode ser alunos, família, etc . (têm de ser válidos identifiers) Não esquecer o ponto e vírgula ";" no fim da declaração. Campo_um e Campo_dois são variáveis membro – member variables – ou campo da estrutura.

Assim criamos novos tipos de dados.

Primeiro método:

struct minha_estrutura { int variavel_um; int campo_dois; char fruta[40]; } ;

Aqui o identificador do tipo "struct" é "minha_estrutura" dentro dessa estrutura temos três campos o ultimo é "fruta"

Agora podemos usar esse tipo "struct" para definir variáveis.

struct minha_estrutura nova_estructura;

Para ter acesso aos membros definidos dentro da estrutura utilizamos um operador de seleçao de membro "."(um ponto).

nova_estrutura.fruta[0];

Nos dá o primeiro caracter da palavra contida dentro do membro "fruta".

Para inicializar um campo da estrutura o processo é o mesmo que usamos com as variáveis.

nova_estrutura.campo_dois = 100;

26.3 Matrizes de estruturas

Uma estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, criar matrizes de estruturas. Vamos ver como ficaria a declaração de um vetor de 100 fichas pessoais:

struct minha_estrutura fichas [100];

Poderíamos então acessar um campo dando um índice do vetor fichas:

fichas[12].variavel_um;

26.4 Declarar instâncias (objetos) da estrutura

Podemos declarar os objetos de duas formas:

- Ao mesmo tempo que declaramos a estrutura

struct product { int weight; float price; } apple, banana, melon;

- Ou como uma variável normal

struct product { .. } int main() { struct product apple, banana, melon; }

E até podemos declarar um array delas

Person p[20];

Pergunta: como é que é feito exatamente os objetos?

Para cada objeto vão ser feito uma cópia dos elementos da estrutura.

Agora isso significa que os objetos são distintos entre si em termos de reserva de memória? ie, à medida que enumero os objetos vão ser reservado para cada objeto o tamanho x de bytes? ou somam-se todos os objetos e reserva-se para todos os objetos de uma forma seguida? Penso que deve ser a 1ª opção.

Se tivermos apenas um objeto (ou variável da estrutura) não é necessário darmos o nome da estrutura

struct { char item[40]; // name of item double cost; // cost double retail; // retail price int on_hand; // amount on hand int lead_time; // number of days before resupply } temp;

26.5 Acessar as variáveis membro das estruturas

Agora queremos dar valores a cada uma das pessoas, queremos dar o nome e a altura, para isso faríamos;

strcpy(p1.name, “Tiago”); p1.altura =1.9;

A forma genérica é:

structure-varname.member-name

ou seja

[objecto_estrutura][member_estrutura]

Exemplo

#include <stdio.h> const int MAX = 3; struct Person { char name[100]; int height; }; int main () { Person p[MAX]; for (int x = 0; x < MAX; x++) { printf(“Enter person’s name: "); getline(cin, p[x].name); printf(“Enter height in meters: "); scanf("%d\n”, &p[x].height); } printf(“Outputting person data\n”); printf("======================\n”); for (int x = 0; x < MAX; x++){ printf(“Person #%d’s name is %s and height is %d.\n”, x + 1, p[x].name, p[x].height); } return 0; }

26.6 Iniciar uma estrutura

Podemos iniciar uma estrutura usando uma **lista de iniciação**, que seria algo como:

Person p1 = {"Jeff Kent”, 72};

isto basicamente é igual a arrays, apenas com a diferença de termos tipos diferentes. Logo a ordem vai interessar, por exemplo se escrevêssemos

Person p1 = {72, “Jeff Kent"}; //não iria funcionar- erro de compilação

26.7 Ponteiros para estruturas

struct movies_t { string title; int year; }; movies_t amovie; movies_t * pmovie;

Nós criámos algo

movies_t title year amovie * pmovie

Vejamos que temos um ponteiro como instância.

// pointers to structures #include <stdio.h> struct movies_t { char title[100]; int year; }; int main () { string mystr; movies_t amovie; movies_t *pmovie; pmovie = &amovie; //atribuímos valor ao ponteiro printf(“Enter title: "); fgets(pmovie->title, 100, stdin); //operador -> printf(“Enter year: "; scanf("%d”, &pmovie->year); printf("\nYou have entered:\n”); printf("%s (%d)\n”, pmovie->title, pmovie->year); //operador -> return 0; }

Como já devem ter deduzido o operador -> será muito similar a pmovie->title é equivalente a (*pmovie).title

Mas olhem que é diferente a:

*pmovie.title que equivalente a *(pmovie.title)

26.8 Passando estruturas como argumento de funções

A estrutura é passada como ponteiro.

#include <stdio.h> #include <string.h> struct Person { string name; int height; }; void setValues(Person*); void getValues(const Person*); int main () { Person p1; setValues(&p1); printf(“Outputting person data\n”); printf("======================\n”); getValues(&p1); return 0; } void setValues(Person* pers) { printf(“Enter person’s name: "); fgets(pers.name, 100, stdin); printf(“Enter height in inches: "); scanf("%d”, &pers.height); } void getValues(const Person* pers) { printf(“Person’s name is %s and height is %d.”, pers.name, pers.height); }

26.9 Estruturas aninhadas

A ideia é ter uma estrutura dentro de outra estrutura.

#include <stdio.h> struct Date //estrutura chamada de date { int day; int month; int year; }; struct Person { char name[100]; int height; Date bDay; //temos uma nova variável, mas notem o tipo }; void setValues(Person*); void getValues(const Person*); int main () { Person p1; setValues(&p1); printf(“Outputting person data\n”); printf("======================\n”); getValues(&p1); return 0; } void setValues(Person* pers) { printf(“Enter person’s name: "); fgets(pers.name, 100, stdin); printf(“Enter height in inches: "); scanf("%d”, &pers.height); printf(“Enter day, month and year of birthday separated by spaces: "); scanf("%d %d %d\n”, &pers.bDay.day, &pers.bDay.month, &pers.bDay.year); } void getValues(const Person* pers) { printf(“Person’s name: %s\n”, pers.name); printf(“Person’s height in inches is: %d\n”, pers.height); printf(“Person’s birthday in dd/mm/yyyy format is: %d/%d/%d\n”, pers.bDay.day, pers.bDay.month, pers.bDay.year); }

Reparem que a estrutura Date tem de ser declarada antes da estrutura Person, pois caso contrário o compilador não entenderia o tipo declarado na estrutura Person.

Pergunta: Por que não podemos acrescentar mais membros (campos) nas estruturas?

Porque elas são compiladas estaticamente com posição de memória já alocada e tipo já conhecido em tempo de compilação

Pergunta: Ao invés de termos apenas variáveis nas estruturas, poderíamos ter também funções?

Sim, como ponteiros para funções.

Capítulo 27

Ponteiros

Poderíamos escrever um livro inteiro sobre ponteiros, pois o conteúdo é demasiadamente extenso. Por esse motivo este assunto foi dividido em **básico**, **intermediário** e **avançado**, assim o leitor poderá fazer seus estudos conforme suas necessidades.

É recomendável para quem está vendo programação pela primeira vez aqui que não se preocupe com o avançado sobre ponteiros por enquanto.

27.1 Básico

27.1.1 O que é um ponteiro?

Um ponteiro é simplesmente uma variável que armazena o endereço de outra variável.

Um exemplo : O que é o ponteiro de um relógio? É o que aponta para as horas, minutos ou segundos. Um ponteiro aponta para algo. Em programação, temos as variáveis armazenadas na memória, e um ponteiro aponta para um endereço de memória.

Imagine as variáveis como documentos, a memória do computador como pastas para guardar os documentos, e o ponteiro como atalhos para as pastas.

Não se desespere caso não consiga entender num primeiro momento, o conceito fica mais claro com a prática.

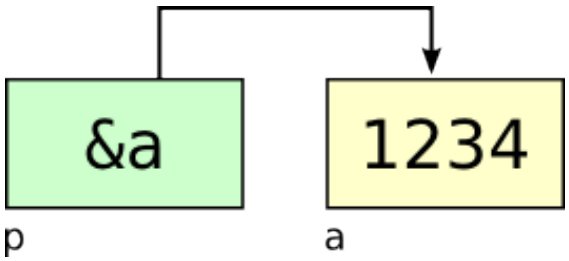
27.1.2 Declarando e acessando ponteiros

Um ponteiro, como qualquer variável, deve ter um tipo, que é o tipo da variável para a qual ele aponta. Para declarar um ponteiro, especificamos *o tipo da variável para a qual ele aponta* e seu nome **precedido por asterisco**:

int ponteiro ; /* declara uma variável comum do tipo inteiro */ int *ponteiro ; /* declara um ponteiro para um inteiro */

Tome cuidado ao declarar vários ponteiros em uma linha, pois o asterisco deve vir **antes de cada nome de variável**. Note os três exemplos:

int p, q, r; // estamos a declarar três variáveis comuns int *p, q, r; // cuidado! apenas **p** será um ponteiro! int *p, *q, *r; // agora sim temos três ponteiros Para acessar o



Esquema de um ponteiro

endereço de uma variável, utilizamos o operador **&** (E comercial), chamado “operador de referência” ou “operador de endereço”. Como o nome sugere, ele retorna o endereço na memória de seu operando. Ele é *unário* e deve ser escrito antes do seu operando. Por exemplo, se uma variável nome foi guardada no endereço de memória 1000, a expressão &nome valerá 1000.

Com isso, fica claro o esquema ao lado: a variável a contém o valor 1234 e o ponteiro p contem o endereço de a (&a).

Para atribuir um valor **ao ponteiro**, usamos apenas seu nome de variável. Esse valor deve ser um endereço de memória, portanto obtido com o operador &:

int a; int *p; p = &a;

Claro que também podemos inicializar um ponteiro:

int *p = &a;

Nos dois casos, o ponteiro p irá apontar para a variável a.

Mas, como o ponteiro contém um endereço, podemos também atribuir um valor à variável guardada nesse endereço, ou seja, **à variável apontada pelo ponteiro**. Para isso, usamos o operador ***** (asterisco), que basicamente significa “o valor apontado por”.
Ex:

int a ; int *p = &a ; *p = 20 ;

Para ver o resultado :

printf (" a :%i\n”, a); printf ("*p :%i\n”, *p);

Cuidado! Você **nunca** deve usar um ponteiro sem antes

inicializá-lo; esse é um erro comum. Inicialmente, um ponteiro pode apontar para qualquer lugar da memória do computador. Ou seja, ao tentar ler ou gravar o valor apontado por ele, você estará manipulando um lugar desconhecido na memória!

int *p; *p = 9;

Nesse exemplo, estamos a manipular um lugar desconhecido da memória! Se você tentar compilar esse código, o compilador deverá dar uma mensagem de aviso; durante a execução, provavelmente ocorrerá uma falha de segmentação (erro que ocorre quando um programa tenta acessar a memória alheia).

Um exemplo mais elaborado:

#include <stdio.h> int main() { int i = 10 ; int *p ; p = &i ; *p = 5 ; printf ("%d\t%d\t%p\n", i, *p, p); return 0; }

Primeiramente declaramos a variável i, com valor 10, e o ponteiro p, que apontará para o endereço de i. Depois, guardamos o valor 5 no endereço apontado por p. Se você executar esse exemplo, verá algo parecido com:

5 5 0022FF74

É claro que os valores de i e de *p serão iguais, já que p aponta para i. O terceiro valor é o endereço de memória onde está i (e, consequentemente, é o próprio valor de p), e será diferente em cada sistema.

Cuidado! Os operadores unários & e * não podem ser confundidos com os operadores binários **AND** bit a bit e **multiplicação**, respectivamente.

27.1.3 Ponteiro e NULL

Uma falha de segmentação ou em inglês (segmentation fault) ocorre quando um programa tenta acessar um endereço na memória que está reservado ou que não existe.Nos sistemas Unix quando acontece este tipo de erro o sinal SIGSEGV é enviado ao programa indicando uma falha de segmentação.

Aqui o ponteiro contem null, definido com o endereço (0x00000000) que causa uma falha de segmentação .

/*Endereço invalido*/ #define null ((char*) 0) int main(void){ int a = 5; int *p = null; *p = a; }

Esse programa termina anormalmente. Você esta tentando colocar o valor 5 em um endereço inválido.

Para que isso não aconteça o ponteiro deve ser inicializado com um endereço valido. Exemplo :

#include <stdio.h> #include <errno.h> #include <stddef.h> int main(void){ int a = 5; int *p = NULL; p = &a; /* A operação não é permitida */ if(p == NULL) return -EPERM ; else{ printf("Endereço a disposição:%p\n", p); *p = a; /* Pode colocar 5 */ } }

NULL está definido dentro do cabeçalho stddef.h . Aqui você não espera que o programa acabe com algum tipo de mágica, se NULL é igual ao valor do ponteiro isso significa que não foi encontrado nem um endereço acessível, então você para. Caso contrario você estará executando uma operação que não é permitida. Ou colocar 5 em (0x00000000) .

27.1.4 Mais operações com ponteiros

Suponhamos dois ponteiros inicializados p1 e p2. Podemos fazer dois tipos de atribuição entre eles:

p1 = p2;

Esse primeiro exemplo fará com que p1 aponte para o mesmo lugar que p2. Ou seja, usar p1 será equivalente a usar p2 após essa atribuição.

*p1 = *p2;

Nesse segundo caso, estamos a igualar **os valores apontados** pelos dois ponteiros: alteraremos o valor apontado por p1 para o valor apontado por p2.

Agora vamos dar mais alguns exemplos com o ponteiro p:

p++;

Aqui estamos a incrementar *o ponteiro*. Quando incrementamos um ponteiro ele passa a apontar para o **próximo valor do mesmo tipo** em relação ao valor para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos, ele passa a apontar para o próximo inteiro. **Note que o incremento não ocorre byte-a-byte!**

(*p)++;

Aqui, colocamos *p entre parênteses para especificar que queremos alterar o valor apontado por p. Ou seja, aqui iremos incrementar o conteúdo da variável apontada pelo ponteiro p.

*p++

Neste caso, o efeito não é tão claro quanto nos outros exemplos. A *precedência* do operador ++ sobre o operador * faz com que a expressão seja equivalente a (*p)++. O valor atual de *p* é retornado ao operador *, e o valor de p é incrementado. Ou seja, obtemos o valor atual do ponteiro e já o fazemos apontar para o próximo valor.

x = *(p + 15);

Esta linha atribui a uma variável x o conteúdo do décimoquinto inteiro adiante daquele apontado por p. Por exemplo, suponhamos que tivéssemos uma série de variáveis i0, i1, i2, … i15 e que p apontasse para i0. Nossa variável x receberia o valor de i15.

Tente acompanhar este exemplo dos dois tipos de atribuição de ponteiros:

int *a, *b, c = 4, d = 2; a = &c; // **a** apontará para **c** b =

&d; // **b** apontará para **d** *b = 8; // altero o valor existente na variavel **d** *a = *b; // copio o valor de **d** (apontado por **b**) // para **c** (apontado por **a**) *a = 1; // altero o valor da variável **c** b = a; // **b** aponta para o mesmo lugar que **a**, // ou seja, para **c** *b = 0; // altero o valor de **c**

27.2 Intermediário

27.2.1 Ponteiro de estrutura

Para começar e deixar mais claro definimos uma estrutura simples com dois campos.

struct { int i; double f; } minha_estrutura;

O passo seguinte é definir um ponteiro para essa estrutura.

struct minha_estrutura *p_minha_estrutura;

A partir do ponteiro podemos ter acesso a um campo da estrutura usando um seletor "->" (uma flecha).

p_minha_estrutura-> i = 1; p_minha_estrutura-> f = 1.2;

O mesmo resultado pode ser obtido da seguinte forma.

(*p_minha_estrutura).i = 1; (*p_minha_estrutura).f = 1.2;

O operador cast também e bastante utilizado para estruturar áreas de estoque temporários (buffer). Os tipos dentro da estrutura devem ser o mesmo do arranjo para evitar problemas de alinhamento.

A seguir um pequeno exemplo:

#include <stdio.h> typedef struct estruturar{ char a ; char b ; } estruturar; int main() { char buffer[2] = {17, 4}; estruturar *p; p = (struct estruturar*) &buffer; printf("a: %i b: %i", p->a,p->b); // getchar(); /* Se o ambiente for windows, descomente o começo da linha. */ return 0; }

27.2.2 Ponteiros como parâmetros de funções

Comecemos por uma situação-problema: eu tenho 2 variáveis e quero trocar o valor delas. Vamos começar com um algoritmo simples, dentro da função main():

#include <stdio.h> int main() { int a = 5, b = 10, temp; printf ("%d %d\n", a, b); temp = a; a = b; b = temp; printf ("%d %d\n", a, b); return 0; }

Esse exemplo funcionará exatamente como esperado: primeiramente ele imprimirá "5 10" e depois ele imprimirá "10 5". Mas e se quisermos trocar várias vezes o valor de duas variáveis? É muito mais conveniente criar uma função que faça isso. Vamos fazer uma tentativa de implementação da função swap (troca, em inglês):

#include <stdio.h> void swap(int i, int j) { int temp; temp = i; i = j; j = temp; } int main() { int a, b; a = 5; b = 10; printf ("%d %d\n", a, b); swap (a, b); printf ("%d %d\n", a, b); return 0; }

No entanto, o que queremos não irá acontecer. Você verá que o programa imprime duas vezes "5 10". Por que isso acontece? Lembre-se do escopo das variáveis: as variáveis a e b são locais à função main(), e quando as passamos como argumentos para swap(), seus valores são copiados e passam a ser chamados de i e j; a troca ocorre entre i e j, de modo que quando voltamos à função main() nada mudou.

Então como poderíamos fazer isso? Como são retornados dois valores, não podemos usar o valor de retorno de uma função. Mas existe uma alternativa: os **ponteiros**!

#include <stdio.h> void swap (int *i, int *j) { int temp; temp = *i; *i = *j; *j = temp; } int main () { int a, b; a = 5; b = 10; printf ("\n\nEles valem %d, %d\n", a, b); swap (&a, &b); printf ("\n\nEles agora valem %d, %d\n", a, b); return 0; }

Neste exemplo, definimos a função swap() como uma função que toma como argumentos dois ponteiros para inteiros; a função faz a troca *entre os valores apontados pelos ponteiros*. Já na função main(), passamos *os endereços das variáveis* para a função swap(), de modo que a função swap() possa modificar variáveis locais de outra função. O único possível inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um & na frente das variáveis que estivermos passando para a função.

Se você pensar bem, já vimos uma função em que passamos os argumentos precedidos de &: é a função *scanf()*! Por que fazemos isso? É simples: chamamos a função scanf() para que ela ponha nas nossas variáveis valores digitados pelo usuário. Ora, essas variáveis são locais, e portanto só podem ser alteradas por outras funções através de ponteiros!

Quando uma função recebe como parâmetros os endereços e não os valores das variáveis, dizemos que estamos a fazer uma **chamada por referência**; é o caso desse último exemplo. Quando passamos diretamente os valores das variáveis para uma função, dizemos que é uma **chamada por valor**; foi o caso do segundo exemplo. Veja mais um exemplo abaixo:

// passagem_valor_referencia.c #include<stdio.h> int cubo_valor(int); int cubo_referencia(int *); int main(){ int number = 5; printf("\nO valor original eh: %d", number); number = cubo_valor(number); printf("\nO novo valor de number eh: %d", number); printf("\n----------"); number = 5; printf("\nO valor original eh: %d", number); cubo_referencia(&number); printf("\nO novo valor de number eh: %d", number); return 0; } int cubo_valor(int a){ return a * a * a; } int

cubo_referencia(int *aPtr){ *aPtr = *aPtr * *aPtr * *aPtr; }

27.2.3 Ponteiros e vetores

Em C, os elementos de um vetor são sempre guardados sequencialmente, a uma distância fixa um do outro. Com isso, é possível facilmente passar de um elemento a outro, percorrendo sempre uma mesma distância para frente ou para trás na memória. Dessa maneira, podemos usar ponteiros e a aritmética de ponteiros para percorrer vetores. Na verdade, vetores *são* ponteiros — um uso particular dos ponteiros. Acompanhe o exemplo a seguir.

#include <stdio.h> int main () { int i; int vetorTeste[3] = {4, 7, 1}; int *ptr = vetorTeste; printf("%p\n", vetorTeste); printf("%p\n", ptr); printf("%p\n", &ptr); for (i = 0; i < 3; i++) { printf("O endereço do índice %d do vetor é %p\n", i, &ptr[i]); printf("O valor do índice %d do vetor é %d\n", i, ptr[i]); } return 0; }

Começamos declarando um vetor com três elementos; depois, criamos um ponteiro para esse vetor. Mas repare que **não colocamos o operador de endereço** em vetorTeste; fazemos isso porque um vetor já representa um endereço, como você pode verificar pelo resultado da primeira chamada a printf().

Como você já viu anteriormente neste capítulo, podemos usar a sintaxe *(ptr + 1) para acessar o inteiro seguinte ao apontado pelo ponteiro *ptr*. Mas, se o ponteiro aponta para o vetor, o próximo inteiro na memória será o próximo elemento do vetor! De fato, em C as duas formas *(ptr + *n*) e ptr[*n*] são equivalentes.

Não é necessário criar um ponteiro para usar essa sintaxe; como já vimos, o vetor em si já é um ponteiro, de modo que qualquer operação com *ptr* será feita igualmente com *vetorTeste*. Todas as formas abaixo de acessar o segundo elemento do vetor são equivalentes:

vetorTeste[1]; *(vetorTeste + 1); ptr[1]; *(ptr + 1)

Veja mais este exemplo:

#include <stdio.h> int main() { int numbers[5]; int *p; int n; p = numbers; *p = 10; p++; *p = 20; p = &numbers[2]; *p = 30; p = numbers + 3; *p = 40; p = numbers; *(p + 4) = 50; for (n = 0; n < 5; n++) cout << numbers[n] << ", "; return 0; }

Ele resume as várias formas de acessar elementos de um vetor usando ponteiros.

27.2.4 Indexação estranha de ponteiros

o C permite fazer um tipo indexação de um vetor quando uma variável controla seu índice. O seguinte código é válido e funciona: Observe a indexação **vetor[i]**.

#include <stdio.h> int main () { int i; int vetor[10]; for (i = 0; i < 10; i++) { printf ("Digite um valor para a posicao %d do vetor: ", i + 1); scanf ("%d", &vetor[i]); //isso é equivalente a fazer *(x + i) } for (i = 0; i < 10; i++) printf ("%d\n", vetor[i]); return (0); }

Essa indexação, apesar de estranha, funciona corretamente e sem aviso na compilação. Ela é prática, mas, para os iniciantes, pode parecer complicada. É só treinar para entender.

27.2.5 Comparando endereços

Como os endereços são números, eles também podem ser comparados entre si. Veja o exemplo a seguir, com efeito equivalente ao primeiro exemplo da seção anterior:

#include <stdio.h> int main() { int vetorTeste[3] = {4, 7, 1}; int *ptr = vetorTeste; int i = 0; while (ptr <= &vetorTeste[2]) { printf("O endereço do índice %d do vetor é %p\n", i, ptr); printf("O valor do índice %d do vetor é %d\n", i, *ptr); ptr++; i++; } return 0; }

Esse programa incrementa o ponteiro enquanto esse endereço for igual (ou menor) ao endereço do último elemento do vetor (lembre-se que os índices do vetor são 0, 1 e 2).

27.3 Avançado

27.3.1 Ponteiros para ponteiros

Note que um ponteiro é uma variável como outra qualquer, e por isso também ocupa espaço em memória. Para obtermos o endereço que um ponteiro ocupa em e memória, usamos o operador &, assim como fazemos nas variáveis comuns.

Mas e se estivéssemos interessados em guardar o endereço de um ponteiro, que tipo de váriavel deveria recebelo? A resposta é: um ponteiro, isto é, um ponteiro para outro ponteiro.

Considere a seguinte declaração:

int x = 1;

Declaramos uma variável chamada x com o valor 1.

Como já sabemos, para declarar um ponteiro, deve-se verificar o tipo da variável que ele irá apontar (neste caso int) e colocar um asterisco entre o tipo da variável e o nome do ponteiro:

int * p_x = &x;

Declaramos um ponteiro apontado para x.

Agora, para se guardar o endereço de um ponteiro, os

mesmos passos devem ser seguidos. Primeiramente verificamos os tipo da variável que será apontada (int *) e colocamos um asterisco entre o tipo e nome do ponteiro:

int ** p_p_x = &p_x;

Declaramos um ponteiro que irá apontar para o ponteiro p_x, ou seja, um ponteiro para ponteiro. Note que C não impõe limites para o número de asteriscos em uma variável.

No exemplo a seguir, todos os printf irão escrever a mesma coisa na tela.

#include <stdio.h> int main(void) { int x = 1; int *p_x = &x; // p_x aponta para x int **p_p_x = &p_x; // p_p_x aponta para o ponteiro p_x printf("%d\n", x); // Valor da variável printf("%d\n", *p_x); // Valor da variável apontada por p_x printf("%d\n", **p_p_x); // Valor da variável apontada pelo endereço apontado por p_p_x return 0; }

Percebe que **p_p_x consiste no valor da variável apontada pelo endereço apontado por p_p_x.

Uma aplicação de ponteiros para ponteiros está nas strings, já que strings são vetores, que por sua vez são ponteiros. Um vetor de strings seria justamente um ponteiro para um ponteiro.

27.3.2 Passando vetores como argumentos de funções

Os ponteiros podem ser passados como argumentos de funções.

Parâmetro ponteiro passando um array.

#include <stdio.h> void atribuiValores(int[], int); void mostraValores(int[], int); int main() { int vetorTeste[3]; // crio um vetor sem atribuir valores atribuiValores(vetorTeste, 3); mostraValores(vetorTeste, 3); return 0; } void atribuiValores(int valores[], int num) { for (int i = 0; i < num; i++) { printf(“Insira valor #%d: ", i + 1); scanf("%d", &valores[i]); } } void mostraValores(int valores[], int num) { for (int i = 0; i < num; i++) { printf(“Valor #%d: %d\n", i + 1, valores[i]); } }

Repare que passamos dois parâmetros para as funções:

1. O “nome” do vetor, que representa o seu endereço na memória. (Temos 3 maneiras para passar o endereço do vetor: diretamente pelo seu “nome”, via um ponteiro ou pelo endereço do primeiro elemento.)

2. Uma constante, que representa o número de elementos do vetor. Isso é importante pois o C não guarda informações sobre o tamanho dos vetores; você não deve tentar alterar ou acessar valores que não pertencem ao vetor.

É claro que devemos passar o endereço do vetor (por “referência”), pois os seus valores são alterados pela função atribuiValores. De nada adiantaria passar o vetor por valor, pois o valor só seria alterado localmente na função (como já vimos no caso de troca do valor de duas variáveis).

Por causa dessa equivalência entre vetores e ponteiros, podemos fazer uma pequena alteração no protótipo (tanto na declaração quanto na definição) das funções atribuiValores e mostraValores, sem precisar alterar o código interno dessas funções ou a chamada a elas dentro da função main ? trocando

void atribuiValores(int[], int); void mostraValores(int[], int);

por

void atribuiValores(int*, int); void mostraValores(int*, int);

Para o compilador, você não fez mudança alguma, justamente por conta dessa equivalência. Em ambos os casos, foi passado o endereço do vetor para as funções.

27.3.3 Ponteiros para funções

Os ponteiros para funções servem, geralmente, para passar uma função como argumento de uma outra função. Neste exemplo

#include <stdio.h> int soma(int a, int b) { return (a + b); } int operacao(int x, int y, int (*func)(int,int)) { int g; g = (*func)(x, y); return (g); } int main () { int m; m = operacao(7, 5, soma); printf("%d\n", m); return 0; }

Veja que criamos uma função que retorna a soma dos dois inteiros a ela fornecidos; no entanto, ela não é chamada diretamente. Ela é chamada pela função *operacao*, através de um ponteiro. A função *main* passa a função *soma* como argumento para *operacao*, e a função *operacao* chama essa função que lhe foi dada como argumento.

Note bem o terceiro argumento da função *operacao*: ele é um **ponteiro para uma função**. Nesse caso, ele foi declarado como um ponteiro para uma função que toma dois inteiros como argumentos e retorna outro inteiro. O * indica que estamos declarando um ponteiro, e não uma função. **Os parênteses em torno de *func são essenciais**, pois sem eles o compilador entenderia o argumento como *uma função que retorna um ponteiro para um inteiro*.

A forma geral para declarar um ponteiro para uma função é:

tipo_retorno (**nome_do_ponteiro*)(**lista de argumentos**)

Para chamar a função apontada pelo ponteiro, há duas sintaxes. A sintaxe original é

(**nome_do_ponteiro*)(**argumentos**);

Se *ptr* é um ponteiro para uma função, faz bastante sentido que a função em si seja chamada por *ptr. No entanto, a sintaxe mais moderna permite que ponteiros para funções sejam chamados exatamente da mesma maneira que funções:

nome_do_ponteiro(**argumentos**);

Por fim, para inicializar um ponteiro para função, não precisamos usar o operador de endereço (ele já está implícito). Por isso, quando chamamos a função *operacao*, não precisamos escrever &soma.

Veja mais um exemplo — na verdade, uma extensão do exemplo anterior:

#include <stdio.h> int soma(int a, int b) { return (a+b); } int subtracao(int a, int b) { return (a-b); } int (*menos)(int, int) = subtracao; int operacao(int x, int y, int (*func)(int,int)) { int g; g = func(x, y); return (g); } int main() { int m, n; m = operacao(7, 5, soma); n = operacao(20, m, menos); printf("%d\n", n); return 0; }

Aqui, criamos mais uma função, *subtracao*, além de criar um outro ponteiro para ela (uma espécie de "atalho"), *menos*. Na função *main*, referimo-nos à função de subtração através desse atalho.

Veja também que aqui usamos a sintaxe moderna para a chamada de ponteiros de funções, ao contrário do exemplo anterior.

Capítulo 28

Mais sobre variáveis

28.1 typedef

A instrução **typedef** serve para definir um novo nome para um certo tipo de dados — intrínseco da linguagem ou definido pelo usuário. Por exemplo, se fizéssemos a seguinte declaração:

typedef **unsigned int** *uint*;

poderíamos declarar variáveis inteiras sem sinal (*unsigned int*) da seguinte maneira:

uint numero; *// equivalente a "unsigned int numero;"*

Como exemplo vamos dar o nome de inteiro para o tipo int:

typedef int inteiro;

Como se vê, *typedef* cria uma espécie de "apelido" para um tipo de dados, permitindo que esse tipo seja referenciado através desse apelido em vez de seu identificador normal.

Um dos usos mais comuns de *typedef* é abreviar a declaração de tipos complexos, como *structs* ou estruturas. Veja este exemplo:

struct *pessoa* { char nome[40]; int idade; }; **struct** *pessoa* joao;

Observe que, para declarar a variável *joao*, precisamos escrever a palavra struct. Podemos usar *typedef* para abreviar essa escrita:

typedef **struct** *_pessoa* { char nome[40]; int idade; } *Pessoa*; **Pessoa** joao;

Um "apelido" de tipo é utilizado com bastante frequência, embora não costumemos dar por isso: é o tipo FILE, usado nas funções de entrada/saída de arquivos.

typedef **struct** *_iobuf* { char* _ptr; int _cnt; char* _base; int _flag; int _file; int _charbuf; int _bufsiz; char* _tmpfname; } *FILE*;

Então, quando declaramos algo como

FILE *fp;

na verdade estamos a declarar um ponteiro para uma estrutura, que será preenchida mais tarde pela função *fopen*.

Atenção! Você não deve tentar manipular uma estrutura do tipo FILE; sua composição foi apresentada apenas como exemplo ou ilustração.

28.2 sizeof

O operador sizeof é usado para se saber o tamanho de variáveis ou de tipos. Ele retorna o tamanho do tipo ou variável em bytes como uma constante. Devemos usá-lo para garantir portabilidade. Por exemplo, o tamanho de um inteiro pode depender do sistema para o qual se está compilando. O sizeof é um operador porque ele é substituído pelo tamanho do tipo ou variável no momento da compilação. Ele não é uma função. O sizeof admite duas formas:

sizeof nome_da_variável sizeof (nome_do_tipo)

Se quisermos então saber o tamanho de um float fazemos sizeof(float). Se declararmos a variável f como float e quisermos saber o seu tamanho faremos sizeof f. O operador sizeof também funciona com estruturas, uniões e enumerações.

Outra aplicação importante do operador sizeof é para se saber o tamanho de tipos definidos pelo usuário. Seria, por exemplo, uma tarefa um tanto complicada a de alocar a memória para um ponteiro para a estrutura ficha_pessoal, criada na primeira página desta aula, se não fosse o uso de sizeof. Veja o exemplo:

typedef struct { const char *nome; const char *sobrenome; int idade; } Pessoa; int main(void) { Pessoa *joaquim; joaquim = malloc(sizeof(Pessoa)); joaquim->nome = "Joaquim"; joaquim->sobrenome = "Silva"; joaquim->idade = 15; }

Outro exemplo:

#include <string.h> #include <stdio.h> int main(void) { char *nome; nome = malloc(sizeof(char) * 10); sprintf(nome, "wikibooks"); printf("Site: http://pt.%s.org/", nome); /* Imprime: Site: http://pt.wikibooks.org/ */ }

A sentença abaixo NÃO funciona, pois sizeof é substi-

tuído pelo tamanho de um ***tipo*** em tempo de compilação.

const char *FRASE; FRASE = “Wikibooks eh legal"; printf(“Eu acho que o tamanho da string FRASE é %d”, sizeof(FRASE));

28.3 Conversão de tipos

As atribuições no C tem o seguinte formato:

destino=origem;

Se o destino e a origem são de tipos diferentes o compilador faz uma conversão entre os tipos. Mas nem todas as conversões são possíveis. O primeiro ponto a ser ressaltado é que o valor de origem é convertido para o valor de destino antes de ser atribuído e não o contrário.

Em C, cada tipo básico ocupa uma determinada porção de bits na memória, logo, a conversão entre tipos nem sempre é algo nativo da linguagem, por assim dizer. Há funções como atol e atof que convertem string em inteiro longo (long int) e string em double, respectivamente. Mas em muitos casos é possível usar o casting.

É importante lembrar que quando convertemos um tipo numérico para outro, nós nunca ganhamos precisão. Nós podemos perder precisão ou no máximo manter a precisão anterior. Isto pode ser entendido de uma outra forma. Quando convertemos um número não estamos introduzindo no sistema nenhuma informação adicional. Isto implica que nunca vamos ganhar precisão.

Abaixo vemos uma tabela de conversões numéricas com perda de precisão, para um compilador com palavra de 16 bits:

De Para Informação Perdida unsigned char char Valores maiores que 127 são alterados short int char Os 8 bits de mais alta ordem int char Os 8 bits de mais alta ordem long int char Os 24 bits de mais alta ordem long int short int Os 16 bits de mais alta ordem long int int Os 16 bits de mais alta ordem float int Precisão - resultado arredondado double float Precisão - resultado arredondado long double double Precisão - resultado arredondado

28.3.1 Casting: conversão manual

Se declararmos **a = 10/3**, sabemos que o resultado é **3,333**, ou seja a divisão de dois números inteiros dá um número real. Porém o resultado em C será o inteiro 3. Isso acontece, porque as constantes são do tipo inteiro e operações com inteiros tem resultado inteiro. O mesmo ocorreria em **a = b/c** se b e c forem inteiros.

Se declararmos:

int a;

O resultado será **3**.

Mesmo que declarássemos:

float a;

o resultado continua a ser 3 mas desta vez, **3,0000**.

Para fazer divisão que resulte número real, é necessário fazer cast para um tipo de ponto flutuante:

a = (float)10/3 a = 10/(float)3

Nesse caso, o 10 ou o 3 é convertido para float. O outro número continua como inteiro, mas ao entrar na divisão com um float, ele é convertido automaticamente para float. A divisão é feita e depois atribuída à variável a.

Em poucas palavras, casting é colocar um tipo entre parênteses antes da atribuição de uma variável. A forma geral para cast é:

(tipo)variável (tipo)(expressão) variavel_destino = (tipo)variavel_origem;

Mas existem umas conversões automáticas:

int f(void) { float f_var; double d_var; long double l_d_var; f_var = 1; d_var = 1; l_d_var = 1; d_var = d_var + f_var; /*o float é convertido em double*/ l_d_var = d_var + f_var; /*o float e o double convertidos em long double*/ return l_d_var; }

Repare que a conversão é feita de menor para o maior.

É possível fazer a conversão ao contrário de um tipo com mais bits para um com menos bits e isso é truncar. Nesse caso, o cast explícito é necessário. Assim, um número **float**: 43.023 ao ser convertido para **int** deverá ser “cortado”, ficando inteiro: **43**. Se converter long para short, os bits mais significativos são perdidos na conversão.

O operador cast também e bastante utilizado para estruturar áreas de estoque temporários (buffer). A seguir um pequeno exemplo:

#include <stdio.h> typedef struct estruturar{ char a ; char b ; }; int main() { char buffer[2] = {17, 4}; estruturar *p; p = (struct estruturar*) &buffer; char* x = (char*)malloc(10); printf(“a: %i b: %i”, p->a,p->b); getchar(); return 0; }

28.4 Atributos das variáveis

Estes modificadores, como o próprio nome indica, mudam a maneira com a qual a variável é acessada e modificada. Alguns dos exemplos usam conceitos que só serão abordados nas seções seguintes, então você pode deixar esta seção para depois se assim o desejar.

28.4.1 const

O modificador **const** faz com que a variável não possa ser modificada no programa. Como o nome já sugere é útil para se declarar constantes. Poderíamos ter, por exemplo:

const float PI = 3.1415;

Podemos ver pelo exemplo que as variáveis com o modificador const podem ser inicializadas. Mas PI não poderia ser alterado em qualquer outra parte do programa. Se o programador tentar modificar PI o compilador gerará um erro de compilação.

Outro uso de **const**, aliás muito comum que o outro, é evitar que um parâmetro de uma função seja alterado pela função. Isto é muito útil no caso de um ponteiro, pois o conteúdo de um ponteiro pode ser alterado por uma função. Para proteger o ponteiro contra alterações, basta declarar o parâmetro como const.

<

nclude <stdio.h>

int sqr (const int *num); int main(void) { int a = 10; int b; b = sqr(&a); } int sqr (const int *num) { return ((*num)*(*num)); }

No exemplo, num está protegido contra alterações. Isto quer dizer que, se tentássemos fazer

*num = 10;

dentro da função sqr(), o compilador daria uma mensagem de erro.

28.4.2 volatile

O modificador **volatile** diz ao compilador que a variável em questão pode ser alterada sem que este seja avisado. Isto evita "bugs" que poderiam ocorrer se o compilador tentasse fazer uma otimização no código que não é segura quando a memória é modificada externamente.

Digamos que, por exemplo, tenhamos uma variável que o BIOS do computador altera de minuto em minuto (um relógio, por exemplo). Seria importante que declarássemos esta variável como volatile.

Um uso importante de variáveis volatile é em aplicações com várias *threads* (linhas de execução), onde a memória é compartilhada por vários pedaços de código que são executados simultaneamente.

28.4.3 extern

O modificador **extern** diz ao compilador que a variável indicada foi declarada em outro arquivo que não podemos incluir diretamente, por exemplo o código de uma biblioteca padrão. Isso é importante pois, se não colocarmos o modificador **extern**, o compilador irá declarar uma nova variável com o nome especificado, "ocultando" a variável que realmente desejamos usar. E se simplesmente não declarássemos a variável, já sabemos que o compilador não saberia o tamanho da variável.

Quando o compilador encontra o modificador **extern**, ele marca a variável como não resolvida, e o montador se encarregará de substituir o endereço correto da variável.

extern float sum; extern int count; float returnSum (void) { count++; return sum; }

Neste exemplo, o compilador irá saber que count e sum estão sendo usados no arquivo mas que foram declarados em outro.

Uma variável externa frequentemente usada é a variável errno (declarada no arquivo-cabeçalho errno.h), que indica o último código de erro encontrado na execução de uma função da biblioteca padrão ou do sistema.

28.4.4 static

O funcionamento das variáveis declaradas como **static** depende de se estas são globais ou locais.

- Variáveis globais static funcionam como variáveis globais dentro de um módulo, ou seja, são variáveis globais que não são (e nem podem ser) conhecidas em outros módulos (arquivos). Isto é util se quisermos isolar pedaços de um programa para evitar mudanças acidentais em variáveis globais. Isso é um tipo de *encapsulamento* — que é, simplificadamente, o ato de não permitir que uma variável seja modificada diretamente, mas apenas por meio de uma função.
- Variáveis locais estáticas são variáveis cujo valor é mantido de uma chamada da função para a outra. Veja o exemplo:

int count (void) { static int num = 0; num++; return num; }

A função count() retorna o número de vezes que ela já foi chamada. Veja que a variável local int é inicializada. Esta inicialização só vale para a primeira vez que a função é chamada pois num deve manter o seu valor de uma chamada para a outra. O que a função faz é incrementar num a cada chamada e retornar o seu valor. A melhor maneira de se entender esta variável local static é implementando. Veja por si mesmo, executando seu próprio programa que use este conceito.

28.4.5 register

O computador pode guardar dados na memória (RAM) e nos registradores internos do processador. As variáveis (assim como o programa como um todo) costumam ser armazenadas na memória. O modificador register diz ao

compilador que a variável em questão deve ser, se possível, guardada em um registrador da CPU.

Vamos agora ressaltar vários pontos importantes:

- **Porque usar register?** Variáveis nos registradores da CPU vão ser acessadas em um tempo muito menor pois os registradores são muito mais rápidos que a memória. No entanto, a maioria dos compiladores otimizantes atuais usa registradores da CPU para variáveis, então o uso de register é freqüentemente desnecessário.
- **Em que tipo de variável podemos usar o register?** Antes da criação do padrão ANSI C, register aplicava-se apenas aos tipos int e char, mas o padrão atual permite o uso de register para qualquer um dos quatro tipos fundamentais. É claro que seqüências de caracteres, arrays e estruturas também não podem ser guardadas nos registradores da CPU por serem grandes demais.
- **register é um *pedido* que o programador faz ao compilador.** Este não precisa ser atendido necessariamente, e alguns compiladores até ignoram o modificador *register*, o que é permitido pelo padrão C.
- **register não pode ser usado em variáveis globais**, pois isto implicaria em um registrador da CPU ficar o tempo todo ocupado por essa variável.

Um exemplo do uso do register é dado a seguir:

int main (void) { register int count; for (count = 0; count < 10; count++) { ... } return 0; }

O loop acima, em compiladores que não guardam variáveis em registradores por padrão, deve ser executado mais rapidamente do que seria se não usássemos o *register*. Este é o uso mais recomendável para o *register*: uma variável que será usada muitas vezes em seguida.

Capítulo 29

Mais sobre funções

29.1 Os argumentos argc e argv

A função main(), como dissemos antes, é uma função especial. Introduzimo-la como uma função sem parâmetros; no entanto, ela também pode receber parâmetros formais. No entanto, o programador não pode escolher quais serão. Eles devem ser os seguintes:

int main (**int** *argc*, **char** **argv*[])

- **argc** (*argument count*) é um inteiro e possui o número de argumentos com os quais o programa foi chamado na linha de comando. Ele é no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

- **argv** (*argument values*) é um ponteiro para uma matriz de strings (conceitos que serão abordados mais à frente). Cada string desta matriz é um dos parâmetros da linha de comando. argv[0] sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em argv que temos argc.

Como pode se imaginar, os nomes dos parâmetros “argc” e “argv” podem ser mudados, mas por questão de padronização não se costuma modificá-los.

Exemplo: Escreva um programa que faça uso dos parâmetros argv e argc. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame data:

data 19 04 99

O programa deverá imprimir: 19 de abril de 1999

#include <stdio.h> #include <stdlib.h> int main(int argc, char *argv[]) { int mes; char *nome_mes [] = { “Janeiro”, “Fevereiro”, “Março”, “Abril”, “Maio”, “Junho”, “Julho”, “Agosto”, “Setembro”, “Outubro”, “Novembro”, “Dezembro” }; if(argc == 4) /* Testa se o número de parâmetros fornecidos está correto, o primeiro parâmetro é o nome do programa, o segundo o dia, o terceiro o mes e o quarto os dois últimos algarismos do ano */ { mes = atoi(argv[2]); /* argv contem strings. A string referente ao mes deve ser transformada em um numero inteiro. A funcao atoi esta sendo usada para isto: recebe a string e transforma no inteiro equivalente */ if (mes<1 || mes>12) /* Testa se o mes e' valido */ printf(“Erro!\nUso: data dia mes ano, todos inteiros”); else printf("\n%s de %s de 19%s”, argv[1], nome_mes[mes-1], argv[3]); } else printf(“Erro!\nUso: data dia mes ano, todos inteiros”); }

29.2 Lista de argumentos

Na linguagem C é possível funções como “printf” onde o número de argumentos podem variar. As reticências (...) indicam um numero variável de argumentos ou argumentos com tipos variável. Ex:

void f_erro(int n, char *fmt, ...);

Essa declaração indica que se deve fornecer pelo menos dois argumentos, um do tipo int e um do tipo char mais pode se fornecer argumentos suplementares. Ou seja, “não há limites para sua criatividade"! Ex:

f_erro(3, “Erro: missão impossível "); f_erro(valor, "%s %d\n”, mensagem, errno);

E necessário ter pelo menos um argumento antes dos pontos. Veja um exemplo incorreto.

void erreur(...);

O arquivo de cabeçalho stdarg.h declara um tipo va_list e define três macros para manipular uma lista de argumentos cuja quantidade e tipos são desconhecidos pela função.

va_start, va_arg et va_end (va como variable argument)

Sintaxe:

#include <stdarg.h> void va_start(va_list ap, last); type va_arg(va_list ap, type); void va_end(va_list ap); void va_copy(va_list dest, va_list src);

Descrição:

va_start:

A macro va_start inicializa ap para uso poste-

rior por va_arg e va_end e deve ser chamada primeiro.

O parâmetro last é o nome do último parâmetro antes da lista de argumentos variáveis, isto é, o último parâmetro o qual a função conheçe o tipo.

Porque o endereço deste parâmetro pode ser usado na macro va_start, ele não deve ser declarado como uma variável register, ou como uma função ou como um array.

va_arg:

A macro va_arg retorna o primeiro argumento variável e faz ap apontar o próximo argumento. O parâmetro ap é aquele inicializado por va_start. O parâmetro type é um nome de tipo. Pode-se apontar para um objeto de um tipo específico simplesmente adicionando um * ao tipo.

O primeiro uso da macro va_arg após a macro va_start retorna o argumento após last. Chamadas sucessivas retornam os valores dos outros argumentos.

Se não existe próximo argumento, ou se type não é compatível com o tipo do próximo argumento, erros aleatórios ocorrerão.

Se ap é passado para uma função que usa va_arg(ap,type) então o valor de ap é destruído após o retorno da função.

va_end:

Cada chamada de va_start deve ter uma chamada correspondente a va_end na mesma função. Após a chamada de va_end a variável ap é destruída. Várias chamadas com va_start e va_end aninhadas são possíveis. va_end pode ser uma macro ou uma função.

Exemplo 1

/* Calcula a soma de n inteiros */ /* o ultimo argumento deve ser 0 */ #include <stdio.h> #include <stdarg.h> int soma(int n1, ...) { va_list pa; int som, n; som = n1; va_start(pa, n1); while((n = va_arg(pa, int)) != 0) som = som + n; va_end(pa); return som; } main() { printf(“1 + 3 + 5 + 7 + 9 = %d\n”, soma(1,3,5,7,9,0)); printf(“1 + 1 = %d\n”, soma(1,1,0)); return 0; } /*-- resultado ---------- 1 + 3 + 5 + 7 + 9 = 25 1 + 1 = 2 ----------*/

Exemplo 2

#include <stdio.h> #include <stdarg.h> void meu_printf(char *fmt, ...) { va_list pa; int n; char *s, c; float f; va_start(pa, fmt); while (*fmt != '\0') { if (*fmt == '%') { /* (*++fmt) equivale a (*fmt = *fmt + 1)*/ switch (*++fmt) { case '%' : putchar('%'); break; case 'c' : /* char*/ c = va_arg(pa, int); putchar(c); break; case 'd' : /* int */ n = va_arg(pa, int); printf("%d”, n); break; case 'f' : /* float */ f = va_arg(pa, double); /* !!!!! */ printf("%f”, f); break; case 's’ : /* string */ s = va_arg(pa, char *); for (; *s != '\0'; s++) putchar(*s); break; } /* end switch */ } else putchar(*fmt); /*incrementa o ponteiro*/ fmt++; } va_end(pa); } int main() { meu_printf(“float = %f\n”, (float) 1.2345); meu_printf(“int = %d char = %c String = %s\n”, 123, 'A', “C is beautiful !"); return 0; }

Capítulo 30

Bibliotecas

30.1 Bibliotecas

Bibliotecas são conjuntos de funções que foram feitas por alguém e que podem ser usadas por outros programas sem que nos preocupemos com o código dessas funções.

Além da vantagem de organizar o código, bibliotecas também têm a vantagem de poderem ser utilizadas em vários programas sem necessidade de copiar grandes trechos de código; basta dizer ao compilador que queremos adicionar aquela biblioteca ao executável.

Por exemplo, vamos tentar criar a nossa própria biblioteca, com duas funções: uma para gerar números (pseudo-)aleatórios e uma para calcular o valor de pagamento de uma amortização com juros compostos. Também incluiremos uma função para gerar um número inicial a partir da hora atual, o que fará com que as seqüências de números não sejam sempre as mesmas.

Chamaremos a biblioteca de **teste1**.

#include <math.h> #include <time.h> int rand_seed = 10; /* Gerador de números pseudo-aleatórios */ int rand () { rand_seed = rand_seed * 1103515245 + 12345; return (unsigned int) (rand_seed / 65536) % 32768; } void init_seed () { rand_seed = time (NULL); } /* Cálculo do valor de cada pagamento de uma amortização * Dados: vp = valor presente; * n = número de pagamentos; * i = taxa de juros (em formato decimal) */ double vf (double vp, int n, double i) { return (vp * i * pow (1 + i, n - 1) / (pow (1 + i, n) - 1)); }

As linhas acima são o arquivo do código da nossa biblioteca. Abaixo está o código de um programa que testará essa biblioteca. Lembre-se de que os dois trechos devem estar em arquivos separados.

#include <stdio.h> int main() { int r1, r2, n_pgtos; double a_vista, juros, v_pgto; r1 = rand (); r2 = rand (); printf (“Números aleatórios: %d, %d\n\n”, r1, r2); printf (" Valor à vista: "); scanf ("%lf”, &a_vista); printf (“Número de pagamentos: "); scanf ("%d”, &n_pgtos); printf (" Taxa de juros: "); scanf ("%lf”, &juros); juros /= 100; /* converte a porcentagem em número */ v_pgto = vf (a_vista, n_pgtos, juros); printf (“Valor de cada pagamento: %lf\n”, v_pgto); return 0; }

Algo que você deve ter notado é que nesse arquivo não demos nenhuma informação sobre as funções *vf* e *rand* nele usadas. Realmente, se você tentar compilar o código como está, o compilador dará um aviso; mas ao tentar criar o executável, o montador não poderá continuar pois não recebeu nenhuma informação sobre onde as funções estão.

Para isso, precisamos realizar três passos adicionais antes de compilar o programa teste:

1. Fazer um arquivo-cabeçalho com informações sobre as funções. Esse arquivo será incluido com a diretiva *#include*, da mesma maneira que cabeçalhos padrão como “stdio.h” ou “math.h”.

2. Compilar a biblioteca separadamente.

3. Instruir o compilador/montador a procurar pela biblioteca ao compilar o programa teste.

30.2 O arquivo-cabeçalho

Arquivos-cabeçalho são arquivos que contém informações que servem para o compilador reconhecer funções (“VER: convenções para chamadas a funções ou calling convention”), macros, tipos de dados e variáveis que não estão no arquivo sendo compilado. Esses arquivos costumam ter a extensão ".h” — é o caso, por exemplo, dos cabeçalhos padrão *stdio.h* e *math.h*. A letra H é usada pois é a inicial de *header* (cabeçalho em inglês).

Em uma biblioteca, os cabeçalhos contêm, os protótipos das funções disponibilizadas pela biblioteca e, quando necessário, sobre os tipos de estruturas usados. Bibliotecas mais complexas costumam dividir essas funções entre vários arquivos.

Para fazer nosso próprio cabeçalho, precisamos colocar as declarações das funções disponíveis na biblioteca:

int rand (); void init_seed (); double vf (double, int, double);

Se você se lembra da última lição, poderá sugerir que coloquemos algumas linhas a mais:

#ifndef _TESTE1_H #define _TESTE1_H int rand (); void init_seed (); double vf (double, int, double); #endif

Agora, sempre que precisarmos usar a biblioteca *teste1*, basta incluir o arquivo *teste1.h* no início do nosso programa:

#include “teste1.h”

Note que se o cabeçalho estiver instalado nos diretórios padrão do compilador ou do sistema, você deve trocar as aspas pelos sinais de menor/maior (< ... >).

30.3 Compilação da biblioteca

Tendo salvo o código da biblioteca no arquivo *teste1.c*, você deve compilar a biblioteca.

30.3.1 No GCC

- Compile o arquivo-fonte normalmente, mas sem gerar o executável:

gcc -c teste1.c -o libteste1.o

- Crie o arquivo da biblioteca com o comando **ar**. Você ainda não o conhece, mas a sintaxe é simples: basta digitar ar rv, seguido do nome do arquivo da biblioteca e depois dos nomes dos arquivos-objeto a serem incluídos (separados por espaços). No GCC, as bibliotecas estáticas costumam ter o nome “lib*nome*.a”.

ar rv libteste1.a libteste1.o

30.3.2 No MS Visual C++

No Visual C++, o nome padrão das bibliotecas é "*nome*.lib”, assim como em vários outros compiladores para Windows. Nele, os comandos correspondentes aos dois passos acima são:

cl /c teste1.c lib /out:teste1.lib teste1.obj

30.4 Compilação do programa

Após criar o arquivo objeto libteste1.o com o comando (gcc -c teste1.c -o libteste1.o) e a biblioteca estática com o comando “ar” , você deve instruir o compilador com as opções de edição de links para poder incluí-la no seu programa:

- No GCC:

gcc main.c -L. -l libteste1.a -o main.bin -lm

Note as opções que você não conhecia: -L e -l . A primeira indica em que diretório deve ser procurada a biblioteca; o ponto indica o diretório atual. Se essa opção for omitida, o compilador procurará apenas nos diretórios padrão. A segunda é uma opção do editor de links indicando uma biblioteca a ser incluída; o compilador procurará pelo arquivo adicionando o prefixo *lib* e a extensão *.a*, daí a necessidade de dar o nome “libteste1.a” à biblioteca. Mais bibliotecas podem ser incluídas como a -lm que neste caso serve para chamar a biblioteca math do math.h, sem este comando ele poderá apresentar um erro na hora da compilação.

- No Visual C++:

link /out:main.exe main.obj teste1.lib

Note que nesse caso simplesmente especificamos os arquivos que devem ser montados. O diretório de procura pode ser especificado pela opção /libpath:*diretório*.

Capítulo 31

Entrada e saída em arquivos

31.1 Trabalhando com arquivos

Já vimos como podemos receber e enviar dados para usuário através do teclado e da tela; agora veremos também como ler e gravar dados em arquivos, o que é aliás muito importante ou até essencial em muitas aplicações.

Assim como as funções de entrada/saída padrão (teclado e tela), as funções de entrada/saída em arquivos estão declaradas no cabeçalho *stdio.h* que significa "STanDard Input-Output". Aliás, as funções para manipulação de arquivos são muito semelhantes às usadas para entrada/saída padrão. Como já dissemos na seção sobre a entrada e saída padrões, a manipulação de arquivos também se dá por meio de **fluxos** (*streams*).

Na manipulação de um arquivo, há basicamente três etapas que precisam ser realizadas:

1. abrir o arquivo;
2. ler e/ou gravar os dados desejados;
3. fechar o arquivo.

Em C, todas as operações realizadas com arquivos envolvem seu *identificador de fluxo*, que é uma variável do tipo FILE * (sobre o qual não cabe agora falar). Para declarar um identificador de fluxo, faça como se fosse uma variável normal:

FILE *fp; // não se esqueça do asterisco!

31.2 Abrindo e fechando um arquivo

Não surpreendentemente, a primeira coisa que se deve fazer para manipular um arquivo é abri-lo. Para isso, usamos a função fopen(). Sua sintaxe é:

FILE ***fopen** (char **nome_do_arquivo*, char **modo_de_acesso*);

- O nome do arquivo deve ser uma string ou com o caminho completo (por exemplo, /usr/share/appname/app.conf ou C:\Documentos\nomes.txt) ou o caminho em relação ao diretório atual (nomes.txt, ../app.conf) do arquivo que se deseja abrir ou criar.

- O modo de acesso é uma string que contém uma seqüência de caracteres que dizem se o arquivo será aberto para gravação ou leitura. Depois de aberto o arquivo, você só poderá executar os tipos de ação previstos pelo modo de acesso: não poderá ler de um arquivo que foi aberto somente para escrita, por exemplo. Os modos de acesso estão descritos na tabela a seguir.

Em ambientes DOS/Windows, ao ler arquivos binários (por exemplo, programas executáveis ou certos tipos de arquivos de dados), deve-se adicionar o caractere "b" ao final da string de modo (por exemplo, "wb" ou "r+b") para que o arquivo seja lido/gravado corretamente.

Isso é necessário porque no modo texto (o padrão quando não é adicionado o *b*) ocorrem algumas traduções de caracteres (por exemplo, a terminação de linha "\r\n" é substituída apenas por "\n" na leitura) que poderiam afetar a leitura/gravação dos arquivos binários (indevidamente inserindo ou suprimindo caracteres).

- O valor de retorno da função fopen() é muito importante! Ele é o identificador do fluxo que você abriu e é só com ele que você conseguirá ler e escrever no arquivo aberto.

- Se houver um erro na abertura/criação do arquivo, a função retornará o valor NULL. O erro geralmente acontece por duas razões:
  - O arquivo não existe, caso tenha sido requisitado para leitura.
  - O usuário atual não tem permissão para abrir o arquivo com o modo de acesso pedido. Por exemplo, o arquivo é somente-leitura, ou está bloqueado para gravação por outro programa, ou pertence a outro usuário e não tem permissão para ser lido por outros.

Ao terminar de usar um arquivo, você deve fechá-lo. Isso é feito pela função fclose():

int **fclose** (FILE **fluxo*);

- O único argumento é o identificador do fluxo (retornado por fopen). O valor de retorno indica o sucesso da operação com o valor zero.
- Fechar um arquivo faz com que qualquer caractere que tenha permanecido no "buffer" associado ao fluxo de saída seja gravado. Mas, o que é este "buffer"? Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o "buffer") em vez de serem escritos em disco imediatamente. Quando o "buffer" estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caractere que fôssemos gravar, tivéssemos que posicionar a cabeça de gravação em um ponto específico do disco, apenas para gravar aquele caractere, as gravações seriam muito lentas. Assim estas gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.
- A função exit() fecha todos os arquivos que um programa tiver aberto.
- A função fflush() força a gravação de todos os caracteres que estão no buffer para o arquivo.

31.2.1 Exemplo

Um pequeno exemplo apenas para ilustrar a abertura e fechamento de arquivos:

#include <stdio.h> int main() { FILE *fp; fp = fopen ("README", "w"); if (fp == NULL) { printf ("Houve um erro ao abrir o arquivo.\n"); return 1; } printf ("Arquivo README criado com sucesso.\n"); fclose (fp); return 0; }

31.2.2 Arquivos pré-definidos

Na biblioteca padrão do C, existem alguns fluxos pré-definidos que não precisam (nem devem) ser abertos nem fechados:

- **stdin**: dispositivo de entrada padrão (geralmente o teclado)
- **stdout**: dispositivo de saída padrão (geralmente o vídeo)
- **stderr**: dispositivo de saída de erro padrão (geralmente o vídeo)
- **stdaux**: dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
- **stdprn**: dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

31.3 Escrevendo em arquivos

Para escrever em arquivos, há quatro funções, das quais três são análogas às usadas para saída padrão:

A seguir apresentamos os protótipos dessas funções:

void **fputc** (int *caractere*, FILE **fluxo*);
void **fputs** (char **string*, FILE **fluxo*);
void **fprintf** (FILE **fluxo*, char **formatação*, ...);
int **fwrite** (void **dados*, int *tamanho_do_elemento*, int *num_elementos*, FILE **fluxo*);

- Sintaxe quase igual à de printf(); só é necessário adicionar o identificador de fluxo no início.

31.3.1 fwrite

- Esta função envolve os conceitos de ponteiro e vetor, que só serão abordados mais tarde.

A função fwrite() funciona como a sua companheira fread(), porém escreve no arquivo. Seu protótipo é:

unsigned fwrite(void *buffer,int numero_de_bytes,int count,FILE *fp);

A função retorna o número de itens escritos. Este valor será igual a count a menos que ocorra algum erro. O exemplo abaixo ilustra o uso de fwrite e fread para gravar e posteriormente ler uma variável float em um arquivo binário.

#include <stdio.h> #include <stdlib.h> int main() { FILE *pf; float pi = 3.1415; float pilido; if((pf = fopen("arquivo.bin", "wb")) == NULL) /* Abre arquivo binário para escrita */ { printf("Erro na abertura do arquivo"); exit(1); } if(fwrite(&pi, sizeof(float), 1,pf) != 1) /* Escreve a variável pi */ printf("Erro na escrita do arquivo"); fclose(pf); /* Fecha o arquivo */ if((pf = fopen("arquivo.bin", "rb")) == NULL) /* Abre o arquivo novamente para leitura */ { printf("Erro na abertura do arquivo"); exit(1); } if(fread(&pilido, sizeof(float), 1,pf) != 1) /* Le em pilido o valor da variável armazenada anteriormente */ printf("Erro na leitura do arquivo"); printf("\nO valor de PI, lido do arquivo e': %f", pilido); fclose(pf); return 0; }

Nota-se o uso do operador sizeof, que retorna o tamanho em bytes da variável ou do tipo de dados.

31.3.2 fputc

A função fputc é a primeira função de escrita de arquivo que veremos. Seu protótipo é:

int fputc (int ch, FILE *fp);

Escreve um caractere no arquivo.O programa a seguir lê uma string do teclado e escreve-a, caractere por caractere em um arquivo em disco (o arquivo arquivo.txt, que será aberto no diretório corrente).

#include <stdio.h> #include <stdlib.h> int main() { FILE *fp; char string[100]; int i; fp = fopen(“arquivo.txt”,"w"); /* Arquivo ASCII, para escrita */ if(!fp) { printf(“Erro na abertura do arquivo”); exit(0); } printf(“Entre com a string a ser gravada no arquivo:"); gets(string); for(i=0; string[i]; i++) putc(string[i], fp); /* Grava a string, caractere a caractere */ fclose(fp); return 0; }

Depois de executar este programa, verifique o conteúdo do arquivo arquivo.txt (você pode usar qualquer editor de textos). Você verá que a string que você digitou está armazenada nele.

31.4 Lendo de arquivos

Novamente, há quatro funções, das quais três se assemelham às usadas para a saída padrão:

int **fgetc** (FILE **fluxo*);
void **fgets** (char **string*, int *tamanho*, FILE **fluxo*);
void **fscanf** (FILE **fluxo*, char **formatação*, ...);
int **fread** (void **dados*, int *tamanho_do_elemento*, int *num_elementos*, FILE **fluxo*);

31.4.1 fgetc

- Está função requer como parâmetro o indicador de fluxo do arquivo, retorna um caractere do arquivo ou EOF, caso ocorra um erro ou o final do arquivo seja atingido, podendo ser verificado respectivamente por *ferror* e *feof*.

Exemplo:

#include <stdio.h> #include <stdlib.h> int main() { FILE *fl; int c; if((fl = fopen(“caminho/do/arquivo”, “r”)) == NULL) { perror(“Erro: fopen”); exit(EXIT_FAILURE); } while((c = fgetc(fl)) != EOF) printf(“Caractere lido: %c\n”, c); if((c == EOF) && (feof(fl) == 0) && (ferror(fl) != 0)) perror(“Erro: fgetc”); fclose(fl); return EXIT_SUCCESS; }

31.4.2 fgets

- Ao chamar a função fgets(), você deve fornecer o ponteiro para a string onde os dados lidos devem ser guardados, além do tamanho máximo dos dados a serem lidos (para que a memória reservada à string não seja ultrapassada).

Para se ler uma string num arquivo podemos usar fgets() cujo protótipo é:

char *fgets (char *str, int tamanho,FILE *fp);

A função recebe 3 argumentos: a string a ser lida, o limite máximo de caracteres a serem lidos e o ponteiro para FILE, que está associado ao arquivo de onde a string será lida. A função lê a string até que um caracter de nova linha seja lido ou tamanho-1 caracteres tenham sido lidos. Se o caracter de nova linha ('\n') for lido, ele fará parte da string, o que não acontecia com gets.

A função fgets é semelhante à função gets(), porém, além dela poder fazer a leitura a partir de um arquivo de dados e incluir o caracter de nova linha na string, ela ainda especifica o tamanho máximo da string de entrada. Como vimos, a função gets não tinha este controle, o que poderia acarretar erros de “estouro de buffer”. Portanto, levando em conta que o ponteiro fp pode ser substituído por stdin, como vimos acima, uma alternativa ao uso de gets é usar a seguinte construção:

fgets (str, tamanho, stdin);

31.4.3 fscanf

- Sintaxe quase igual à de scanf(); só é necessário adicionar o identificador de fluxo no início.

31.4.4 fscanf

A função fscanf() funciona como a função scanf(). A diferença é que fscanf() lê de um arquivo e não do teclado do computador. Protótipo:

int fscanf (FILE *fp,char *str,...);
#include <stdio.h> #include <stdlib.h> int main() { FILE *p; char str[80],c; printf("\n\n Entre com um nome para o arquivo:\n”); /* Le um nome para o arquivo a ser aberto: */ gets(str); if (!(p = fopen(str,"w"))) /* Caso ocorra algum erro na abertura do arquivo..*/ { /* o programa aborta automaticamente */ printf(“Erro! Impossivel abrir o arquivo!\n”); exit(1); } fprintf(p,"Este e um arquivo chamado:\n%s\n”, str); fclose(p); /* Se nao houve erro, imprime no arquivo, fecha ...*/ p = fopen(str,"r”); /* abre novamente para a leitura */ while (!feof(p)) { fscanf(p,"%c”,&c); printf("%c”,c); } fclose(p); return 0; }

31.4.5 fread

- Essa função envolve os conceitos de ponteiro e vetor, que só serão abordados mais tarde.

Podemos escrever e ler blocos de dados. Para tanto, temos as funções fread() e fwrite(). O protótipo de fread() é:

unsigned fread (void *buffer, int numero_de_bytes, int count, FILE *fp);

O buffer é a região de memória na qual serão armazenados os dados lidos. O número de bytes é o tamanho da unidade a ser lida. count indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

numero_de_bytes*count

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que count quando o fim do arquivo for encontrado ou ocorrer algum erro.

Quando o arquivo for aberto para dados binários, fread pode ler qualquer tipo de dados.

31.5 Movendo pelo arquivo

31.5.1 fseek

Para se fazer procuras e acessos randômicos em arquivos usa-se a função fseek(). Esta move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado. Seu protótipo é:

int fseek (FILE *fp, long numbytes, int origem);

O parâmetro origem determina a partir de onde os numbytes de movimentação serão contados. Os valores possíveis são definidos por macros em stdio.h e são:

Nome Valor Significado SEEK_SET 0 Início do arquivo SEEK_CUR 1 Ponto corrente no arquivo SEEK_END 2 Fim do arquivo

Tendo-se definido a partir de onde irá se contar, numbytes determina quantos bytes de deslocamento serão dados na posição atual.

31.5.2 rewind

Volta para o começo do arquivo de um fluxo

31.5.3 feof

EOF (“End of file”) indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função feof(). Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

int feof (FILE *fp);

Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por getc com EOF. O programa a seguir abre um arquivo já existente e o lê, caracter por caracter, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela:

#include <stdio.h> #include <stdlib.h> int main() { FILE *fp; char c; fp = fopen(“arquivo.txt”,"r”); /* Arquivo ASCII, para leitura */ if(!fp) { printf(“Erro na abertura do arquivo”); exit(0); } while((c = getc(fp)) != EOF) /* Enquanto não chegar ao final do arquivo */ printf("%c”, c); /* imprime o caracter lido */ fclose(fp); return 0; }

Verifique o exemplo.

#include <stdio.h> #include <stdlib.h> #include <string.h> int main() { FILE *p; char c, str[30], frase[80] = “Este e um arquivo chamado: "; int i; printf("\n\n Entre com um nome para o arquivo:\n”); gets(str); /* Le um nome para o arquivo a ser aberto: */ if (!(p = fopen(str,"w”))) /* Caso ocorra algum erro na abertura do arquivo..*/ { printf(“Erro! Impossivel abrir o arquivo!\n”); exit(1); /* o programa aborta automaticamente */ } strcat(frase, str); for (i=0; frase[i]; i++) putc(frase[i],p); fclose(p); /* Se nao houve erro,imprime no arquivo e o fecha ...*/ p = fopen(str,"r”); /* Abre novamente para leitura */ c = getc(p); /* Le o primeiro caracter */ while (!feof(p)) /* Enquanto não se chegar no final do arquivo */ { printf("%c”,c); /* Imprime o caracter na tela */ c = getc(p); /* Le um novo caracter no arquivo */ } fclose(p); /* Fecha o arquivo */ }

31.6 Outras funções

31.6.1 ferror e perror

Protótipo de ferror:

int ferror (FILE *fp);

A função retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo. se torna muito útil quando queremos verificar se cada acesso a um arquivo teve sucesso, de modo que consigamos garantir a integridade dos nossos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado.

Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc. Uma função que pode ser usada em conjunto com ferror() é a função perror() (print error), cujo argumento é uma string que normalmente indica em

que parte do programa o problema ocorreu.

#include <stdio.h> #include <stdlib.h> int main() { FILE *pf; char string[100]; if((pf = fopen(“arquivo.txt”,"w")) ==NULL) { printf("\nNao consigo abrir o arquivo ! "); exit(1); } do { printf("\nDigite uma nova string. Para terminar, digite <enter>: "); gets(string); fputs(string, pf); putc('\n', pf); if(ferror(pf)) { perror(“Erro na gravacao”); fclose(pf); exit(1); } }while (strlen(string) > 0); fclose(pf); }

Capítulo 32

Gerenciamento de memória

32.1 Alocação dinâmica

Todos os dados de um programa são armazenados na memória do computador; é muito comum necessitar reservar um certo espaço na memória para poder guardar dados mais tarde. Por exemplo, poderíamos reservar um espaço de 1000 bytes para guardar uma string que o usuário viesse a digitar, declarando um vetor de 1000 caracteres. E se quiséssemos reservar um espaço que só é conhecido no tempo de execução do programa? E se o espaço fosse muito grande, de modo que declarar vetores de tal tamanho seria inconveniente (pois, entre outras coisas, aumenta sem necessidade o tamanho do executável)?

Para solucionar esse problema, existe a **alocação dinâmica de memória**, que como o nome sugere, é uma maneira de alocar memória à medida que o programa vai sendo executado. As quatro funções relacionadas com a alocação dinâmica serão descritas a seguir.

32.1.1 malloc e free

Essas duas funções são as mais básicas para o gerenciamento de memória. **malloc** é responsável pela alocação de um pedaço de memória, e **free** é responsável por liberar esse pedaço de memória.

A função malloc() serve para alocar memória e tem o seguinte protótipo:

void *malloc (unsigned int num); void free (void * ptr);

Para alocar um espaço na memória, precisamos fornecer à função malloc o número de bytes desejados. Ela aloca na memória e retorna um ponteiro void * para o primeiro byte alocado. O ponteiro void* pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função malloc() retorna um ponteiro nulo.

Para saber o tamanho do bloco a alocar, precisaremos usar o operador sizeof. Ele permite também saber automaticamente o tamanho de *structs* criadas pelo usuário.

Veja um exemplo de alocação dinâmica:

#include <stdio.h> #include <stdlib.h> int main(int argc, char *argv[]) { /* ponteiro para memória que será alocada */ int *p; int i; /* alocar 10 elementos inteiros, ou seja, (sizeof (int) * 10) */ p = (int *) malloc (sizeof (int) * 10); if (p == NULL) { printf (“Erro: Não foi possivel alocar memória\n”); exit(1); } for(i = 0; i < 10; i++) { p[i] = i * 2; printf ("%d\n”, p[i]); } /* libera a memória alocada por malloc */ free (p); return 0; }

Outros exemplos:

int main() { int *p, *q; p = malloc(sizeof(int)); q = p; *p = 10; printf("%d\n”, *q); *q = 20; printf("%d\n”, *q); } int main() { int *p, *q; p = malloc(sizeof(int)); q = malloc(sizeof(int)); *p = 10; *q = 20; *p = *q; printf("%d\n”, *p); }

- O compilador aceita *p=*q porque são ambos int.
- O compilador aceita também p=q porque ambos são ponteiros e apontam para o mesmo tipo.
- Podemos simplificar p = malloc(sizeof(int)); por p = malloc(4); mas como temos sistemas operacionais de 16,32, 64 bits a primeira declaração torna as coisas mais portáveis.

32.1.2 calloc

A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

void *calloc(size_t nelem, size_t elsize);

A função calloc reserva um bloco com o tamanho (nelem x elsize) octetos consecutivos, isto é, aloca memória suficiente para um vetor de num objetos de tamanho size. Diferente de malloc(), o bloco reservado é inicializado a 0. Essa função retorna um ponteiro void* para o primeiro byte alocado. O ponteiro void* pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função calloc() retorna um ponteiro nulo.

Exemplo:

#include <stdio.h> #include <stdlib.h> /* Para usar calloc() */ int main (){ int *p; int n; int i; ... /* Determina o

valor de n em algum lugar */ p = calloc(n, sizeof(int)); /* Aloca n números inteiros p pode agora ser tratado como um vetor com n posicoes */ //p = malloc(n*sizeof(int)); /* Maneira equivalente usando malloc. */ if (!p) { printf ("** Erro: Memoria Insuficiente **"); exit(0); } for (i=0; i<n; i++) /* p pode ser tratado como um vetor com n posicoes */ p[i] = i*i; ... return 0; }

No exemplo acima, é alocada memória suficiente para se colocar n números inteiros. O operador sizeof() retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro void * que calloc() retorna é convertido para um int* pelo cast e é atribuído a p. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, p terá um valor nulo, o que fará com que !p retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(a-1)].

32.1.3 realloc

A função realloc() serve para realocar memória e tem o seguinte protótipo:

void *realloc(void *ptr, size_t size);

A função realloc ajusta o tamanho de um bloco a size octetos consecutivos. A função modifica o tamanho da memória previamente alocada com malloc, calloc ou realloc e apontada por *ptr* para o tamanho especificado por *size*. O valor de *size* pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, o bloco antigo é liberado e nenhuma informação é perdida. Se não precisar mover, o valor retornado é igual a ptr. Se *ptr* for nulo, a função aloca size bytes e devolve um ponteiro, funcionando como malloc(); se *size* é zero, a memória apontada por ptr é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

Exemplo:

#include <stdio.h> #include <string.h> #include <stdlib.h> int main() { char *str1=NULL, *str2=NULL; str1 = (char *) malloc(11); strcpy(str1, “ABCDEFGHIJ”); str2 = (char *) realloc(str2, 20); printf(“Endereço de str1 : %p\n”, str1); printf(“Endereço de str2 : %p\n”, str2); str1 = (char *) realloc(str1, 100); printf(“Novo endereço de str1 : %p\n”, str1); printf(“Conteudo de str1 : %s\n”, str1); free(str1); free(str2); return 0; }

32.1.4 Alocação Dinâmica de Vetores

A alocação dinâmica de vetores utiliza os conceitos aprendidos na aula sobre ponteiros e as funções de alocação dinâmica apresentados. Um exemplo de implementação para vetor real é fornecido a seguir:

#include <stdio.h> #include <stdlib.h> float *Alocar_vetor_real (int n) { float *v; /* ponteiro para o vetor */ if (n < 1) { /* verifica parametros recebidos */ printf ("** Erro: Parametro invalido **\n”); return (NULL); } v = calloc (n, sizeof(float)); /* aloca o vetor */ if (v == NULL) { printf ("** Erro: Memoria Insuficiente **"); return (NULL); } return (v); /* retorna o ponteiro para o vetor */ } float *Liberar_vetor_real (float *v) { if (v == NULL) return (NULL); free(v); /* libera o vetor */ return (NULL); /* retorna o ponteiro */ } int main (void) { float *p; int a; ... /* outros comandos, inclusive a inicializacao de a */ p = Alocar_vetor_real (a); ... /* outros comandos, utilizando p[] normalmente */ p = Liberar_vetor_real (p); }

32.1.5 Alocação Dinâmica de Matrizes

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, ou seja é um ponteiro para ponteiro, o que é denominado indireção múltipla. A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. Um exemplo de implementação para matriz real bidimensional é fornecido a seguir. A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

#include <stdio.h> #include <stdlib.h> float **Alocar_matriz_real (int m, int n) { float **v; /* ponteiro para a matriz */ int i; /* variavel auxiliar */ if (m < 1 || n < 1) { /* verifica parametros recebidos */ printf ("** Erro: Parametro invalido **\n”); return (NULL); } /* aloca as linhas da matriz */ v = calloc (m, sizeof(float *)); /*Um vetor de m ponteiros para float */ if (v == NULL) { printf ("** Erro: Memoria Insuficiente **"); return (NULL); } for (i = 0; i < m; i++) /* aloca as colunas da matriz */ { v[i] = calloc (n, sizeof(float)); /* m vetores de n floats */ if (v[i] == NULL) { printf ("** Erro: Memoria Insuficiente **"); return (NULL); } } return (v); /* retorna o ponteiro para a matriz */ } float **Liberar_matriz_real (int m, int n, float **v) { int i; /* variavel auxiliar */ if (v == NULL) return (NULL); if (m < 1 || n < 1) { /* verifica parametros recebidos */ printf ("** Erro: Parametro invalido **\n”); return (v); } for (i=0; i<m; i++) free (v[i]); /* libera as linhas da

```
matriz */ free (v); /* libera a matriz (vetor de ponteiros) */ return (NULL); /* retorna um ponteiro nulo */ } int main (void) { float **mat; /* matriz a ser alocada */ int l, c; /* numero de linhas e colunas da matriz */ int i, j; ... /* outros comandos, inclusive inicializacao para l e c */ mat = Alocar_matriz_real (l, c); for (i = 0; i < l; i++) for ( j = 0; j < c; j++) mat[i][j] = i+j; ... /* outros comandos utilizando mat[][] normalmente */ mat = Liberar_matriz_real (l, c, mat); ... }
```

Capítulo 33

Sockets

33.1 Abstrações

A versão Unix BSD 4.1c de 1982 para VAX foi a primeira a incluir TCP/IP no kernel do sistema operacional, oferecendo ao mesmo tempo uma interface de programação como abstração para esses protocolos. Os soquetes ou sockets são uma API (Application Program Interface) isso quer dizer uma interface entre os programas e a camada de transporte. Exemplo: TCP, UDP. Os soquetes podem usar outros protocolos como AppleTalk, Xérox XNS, etc. A API de sockets foi desenvolvida para a linguagem C e são uma das principais API para sistemas do tipo UNIX. O Windows possui uma interface similar conhecida com o nome de Winsock.

33.2 Funções da biblioteca padrão

int accept(int, struct sockaddr *restrict, socklen_t *restrict); int bind(int, const struct sockaddr *, socklen_t); int connect(int, const struct sockaddr *, socklen_t); int getpeername(int, struct sockaddr *restrict, socklen_t *restrict); int getsockname(int, struct sockaddr *restrict, socklen_t *restrict); int getsockopt(int, int, int, void *restrict, socklen_t *restrict); int listen(int, int); ssize_t recv(int, void *, size_t, int); ssize_t recvfrom(int, void *restrict, size_t, int, struct sockaddr *restrict, socklen_t *restrict); ssize_t recvmsg(int, struct msghdr *, int); ssize_t send(int, const void *, size_t, int); ssize_t sendmsg(int, const struct msghdr *, int); ssize_t sendto(int, const void *, size_t, int, const struct sockaddr *, socklen_t); int setsockopt(int, int, int, const void *, socklen_t); int shutdown(int, int); int socket(int, int, int); int sockatmark(int); int socketpair(int, int, int, int[2]);

33.3 Famílias de endereço

Existem varias famílias de endereço e cada uma corresponde a um protocolo em particular.
As famílias mais usadas são :

AF_UNIX: Protocolo interno do UNIX

AF_INET: Protocolo Internet
AF_NS : Protocolo de Xerox NS

33.4 Estruturas de endereço

Varias chamada ao sistema de redes do unix precisam apontar para uma estrutura de endereço de socket.
A definição dessas estruturas esta definida dentro do cabeçalho <sys/socket.h>.

struct sockaddr { u_short sa_family ; char sa_data[14] ; } ;

sa_family: Família de endereço leva o valor AF_xxx .
sa_data: endereço especifico de protocolo .

Para a família internet as estrutura estão definidas dentro do cabeçalho <netinet/in.h>.

struct in_addr { u_long s_addr ; } ;

struct sockaddr_in { short sin_family ; u_short sin_port ; struct in_addr sin_addr ; char sin_zero[8] ; } ;

Capítulo 34

Makefiles

34.1 Makefile

O objetivo de Makefile é definir regras de compilação para projetos de software. Tais regras são definidas em arquivo chamado **Makefile**. O programa **make** interpreta o conteúdo do Makefile e executa as regras lá definidas. Alguns Sistemas Operacionais trazem programas similares ao make, tais como gmake, nmake, tmake, etc. O programa make pode variar de um sistema a outro pois não faz parte de nenhuma normalização .

O texto contido em um Makefile é usado para a compilação, ligação(linking), montagem de arquivos de projeto entre outras tarefas como limpeza de arquivos temporários, execução de comandos, etc.

Vantagens do uso do Makefile:

- Evita a compilação de arquivos desnecessários. Por exemplo, se seu programa utiliza 120 bibliotecas e você altera apenas uma, o make descobre (comparando as datas de alteração dos arquivos fontes com as dos arquivos anteriormente compilados) qual arquivo foi alterado e compila apenas a biblioteca necessária.

- Automatiza tarefas rotineiras como limpeza de vários arquivos criados temporariamente na compilação

- Pode ser usado como linguagem geral de script embora seja mais usado para compilação

As explicações a seguir são para o utilitário GNU make (gmake) que é similar ao make.

Então vamos para a apresentação do Makefile através da compilação de um pequeno projeto em linguagem C.

- Criar uma pasta com esses 4 arquivos :

teste.c ,teste.h , main.c, Makefile.

- De um nome para a pasta Projeto.

/*================ teste.c ======================*/ #include <stdio.h> #include <stdlib.h> /*Uma funçao makeTeste()*/ void makeTeste(void){ printf(“O Makefile é super Legal\n”); }

Aqui escrevemos o header :

/*====================== teste.h ===============*/ /*================== Cabeçalho ou header ========*/ #ifndef _H_TESTE #define _H_TESTE /* A nossa função */ void makeTeste(void); /* De um enter depois de endif*/ /*Para evitar warning*/ #endif

Agora a função main :

/*====================== main.c ================*/ #include <stdio.h> #include <stdlib.h> #include “teste.h” /* Aqui main ;(*/ int main(void){ makeTeste(); return (0); }

Para compilar fazemos um arquivo Makefile minimal.

#Para escrever comentários ## ############################ Makefile ######################### all: teste teste: teste.o main.o # O compilador faz a ligação entre os dois objetos gcc -o teste teste.o main.o #-----> Distancia com o botão TAB ### e não com espaços teste.o: teste.c gcc -o teste.o -c teste.c -W -Wall -ansi -pedantic main.o: main.c teste.h gcc -o main.o -c main.c -W -Wall -ansi -pedantic clean: rm -rf *.o mrproper: clean rm -rf teste

Para não ter erros os espaços devem ser feito com a tecla TAB.
E compilar é só ir dentro da pasta “Projeto” apertar F4 escrever make e apertar enter.

Uma vez compilado podemos modificar teste.c . Se teste.c foi modificado então make modifica teste.o e se não deixa teste.o como esta.

- all : É o nome das regras a serem executadas.

- teste: teste.c .Pode ser interpretado com arquivo_de_destino: arquivo_de_origem.

- clean: Apaga os arquivos intermediários.Se você escrever no console make clean

ele apaga os arquivos objeto da pasta.

- mrproper: Apaga tudo o que deve ser modificado.No console escreva make mrproper

34.1.1 Sintaxe de criação do arquivo

O makefile funciona de acordo com regras, a sintaxe de uma regra é:

regra: dependências Apertar o botão TAB comando comando ...

Regras complementares

- all : É o nome das regras a serem executadas.
- clean: Apaga os arquivos intermediários.
- mrproper: Apaga tudo o que deve ser modificado.

Definir Variáveis

As variáveis servem para facilitar o trabalho.
Em vez de mudar varias linhas mudamos só o conteúdo da variável.
Deve ser por isso que se chama variável, não?
Definimos da forma seguinte.

NOME=CONTEÚDO E para utilizar esta variável colocamos entre $() .
Então ela vai ficar assim $(NOME)

Vamos para o exemplo com o nosso Makefile.
Colocamos em vez de :

- NOME SRC
- E em vez de CONTEÚDO main.c .
- E para poder usar $(SRC)

Será que na pratica funciona?. Vamos ver..

#Para escrever comentários ## ############################ Makefile ######################### #Definimos a variável SRC=main.c all: teste teste: teste.o main.o gcc -o teste teste.o main.o #-----> Distancia com o botao TAB ### e nao com espaços teste.o: teste.c gcc -o teste.o -c teste.c -W -Wall -ansi -pedantic # #Coloquei $(SRC) em todos os lugares aonde estava main.c main.o: $(SRC) teste.h gcc -o main.o -c $(SRC) -W -Wall -ansi -pedantic clean: rm -rf *.o mrproper: clean rm -rf teste

Todos os lugares do código que contem o CONTEÚDO da variável são modificados colocando no lugar respectivo o NOME da
variável.

Variáveis Personalizadas

- CC=gcc .Definimos CC para nomes de compiladores de C ou C++ .Aqui o gcc.
- CFLAGS=-W -Wall -ansi -pedantic .Serve para definir opções passadas ao compilador.

Para o c++ o NOME e CXXFLAGS .

- LDFLAGS e utilizado para editar as opções de links.
- EXEC=teste .EXEC define o NOME do futuro programa executável.
- OBJ=teste.o main.o . Para cada arquivo.c um arquivo OBJETO e criado com a extensão ".o" arquivo.o .

Então e só olhar na sua pasta todos os arquivos com a extensão ".c" e colocar na variável OBJ com a extensão".o" .

- Outra maneira e mesma coisa. OBJ agora e igual a main.o teste.o

SRC = main.c teste.c
OBJ= $(SRC:.c=.o)

- E super manero a tua idéia camarada.
- Mais tenho 200 arquivos.c e não quero olhar o nome de todos um por um.
  - Tem outra idéia??
  - Poderíamos utilizar *c mais não podemos utilizar este caracter joker na definição de uma variável.
  - Então vamos utilizar o comando " wildcard " ele permite a utilização de caracteres joker na definição de variáveis.
    Fica assim.

SRC= $(wildcard *.c) OBJ= $(SRC:.c=.o)

- Observação se quiser fazer aparecer uma mensagem durante a compilação escreva @echo "Minha mensagem" .
- E mais tem um monte de mensagens e fica muito feio

- Tem outra idéia??.. O pessoal vamos parando ;) não sou uma maquina de idéias.
- Para deixar as mensagens em modo silencioso coloque "@" no começo do comando.
- Fica assim

@$(CC) -o $@ $^

Variáveis internas

$@ Nome da regra. $< Nome da primeira dependência $^ Lista de dependências $? Lista de dependências mais recentes que a regra. $* Nome do arquivo sem sufixo

As regras de interferência

Não disse nada antes porque estávamos no estado principiantes “noob”.
São regras genéricas chamadas por default.

- .c.o : .Ela significa fazer um arquivo.o a partir de um arquivo.c .
- %.o: %.c .A mesma coisa. A linha teste.o: teste.c pode ser modificada com essa regra.
- .PHONY: .Preste bem atenção. Esta regra permite de evitar conflitos.
    - Por exemplo “clean:" e uma regra sem nem uma dependência não temos nada na pasta que se chame clean.
    - Agora vamos colocar na pasta um arquivo chamado clean. Se você tentar apagar os “arquivos.o” escrevendo “make clean” não vai acontecer nada porque make diz que clean não foi modificado.
    - Para evitar esse problema usamos a regra .PHONY : . Fica assim.
    - .PHONY: clean mrproper
    - .PHONY: diz que clean e mrproper devem ser executados mesmo se arquivos com esses nomes existem.

Agora vamos modificar mais uma vez o nosso Makefile com tudo o que sabemos sobre variáveis.

#Para escrever comentários ## ############################## Makefile ########################## #Definimos a variável CC=gcc CFLAGS=-W -Wall -ansi -pedantic EXEC=teste OBJ=teste.o main.o all: $(EXEC) @echo “Vou começar a compilação” #Não coloquei a variável OBJ para que possam entender as variáveis internas. #Se entenderam podem colocar $(OBJ) no lugar de teste.o main.o teste: teste.o main.o # $@ = teste: # $^ = teste.o main.o $(CC) -o $@ $^ # teste.o:teste.c %.o: %.c $(CC) -o $@ -c $< $(CFLAGS) main.o: main.c teste.h $(CC) -o $@ -c $< $(CFLAGS) .PHONY: clean mrproper clean: rm -rf *.o @echo “Compilaçao prontinha” mrproper: clean rm -rf $(EXEC)

- Po legal ;) parece até trabalho de gente grande.

Sub Makefiles

Ler tudo isso só para compilar um programa??
O sub-makefile e lançado por meio de um “Makefile principal” vamos simplificar para o Patrão Makefile.
Aonde estávamos??...Ah sim, para que serve??
O Makefile Principal executa os sub-makesfiles de outras pastas.
Como ele faz??

Usamos uma variável pre-definida $(MAKE).

Bom, ao trabalho. Crie dentro da pasta “Projetos” outra pasta com o nome “sub-make”.Dentro da pasta sub-make crie um arquivo
Makefile e um arquivo submake.c

Dentro da pasta sub-make coloque este Makefile.

##################Pasta:sub-make ## Makefile ################## CC=gcc CFLAGS=-W -Wall -ansi -pedantic EXEC=teste2 SRC= $(wildcard *.c) OBJ= $(SRC:.c=.o) all: $(EXEC) @echo “compilando sub-makefile” @echo “sub-makefile compilado” teste2: $(OBJ) @$(CC) -o $@ $^ .PHONY: clean mrproper clean: @rm -rf *.o mrproper: clean @rm -rf $(EXEC)

Agora vamos escrever o arquivo submake.c .

#include <stdio.h> #include <stdlib.h> /* Informação * Nao utilizem este código para fazer um kernel */ int main(void) { printf(“Sou o binário que está em sub-make”); printf(“Finalmente em fim vivo graças ao Patrão Makefiles ;)"); return (0); }

Agora retorne na pasta “Projeto” vamos modificar o Makefile .
Vamos colocar a seguinte linha:

@cd sub-make && $(MAKE)

- Explicando: "@" silencioso “cd” para abrir a pasta sub-make "&&" e executar make "$(MAKE)"
- Vamos fazer a mesma coisa para “clean:" e “mrproper:" então ao executar “make clean” no console ele vai executar o mesmo comando no sub-makefile.

########################## O Makefile principal ##########################" CC=gcc CFLAGS=-W -Wall -ansi -pedantic EXEC=teste SRC= $(wildcard

*.c) OBJ= $(SRC:.c=.o) all: $(EXEC) @echo “Compilando Projeto” @echo “O patrão foi compilado” #A linha que vai compilar sub-make @cd sub-make && $(MAKE) teste: $(OBJ) @$(CC) -o $@ $^ %.o: %.c @$(CC) -o $@ -c $< $(CFLAGS) main.o: main.c teste.h @$(CC) -o $@ -c $< $(CFLAGS) .PHONY: clean mrproper clean: @rm -rf *.o *~ # E a mesma coisa que dar um F4 dentro da pasta sub-make # e escrever make clean @cd sub-make && $(MAKE) $@ mrproper: clean @rm -rf $(EXEC) #modificamos aqui também @cd sub-make && $(MAKE) $@

Não esqueça de dar TAB em todas as linhas que estão em baixo dos ":" dois pontinhos. OK agora é só dar um F4 dentro da pasta projetos e você tem três comandos a disposição.

- make
- make clean
- make mrproper

Make install

Automatizando a instalação do programa com a regra install: .

- install: .Coloca o binário ou executável em uma determinada pasta, como por exemplo /bin ou /usr/bin no Linux. Pode ser em qualquer outra, utilizando o comando “mv” ou “cp” para mover ou copiar.
- Crie uma pasta bin dentro de “Projetos”. Devem saber que não devem colocar nada inútil que venha da internet na pasta raiz do linux.
- Vamos fazer duas variáveis:
  - prefix=/caminho/ate onde/esta/Projetos
  - bindir=$(prefix)/bin .Igual a /caminho ate/Projetos/dentro de Projetos a pasta bin .
  - E adicionarmos a regra install:all com seus comandos.

Modificando o make principal.

########################## O Makefile principal ##########################" #Coloque o caminho até Projeto aqui prefix=/home/USER/Projeto bindir=$(prefix)/bin CC=gcc CFLAGS=-W -Wall -ansi -pedantic EXEC=teste SRC= $(wildcard *.c) OBJ= $(SRC:.c=.o) all: $(EXEC) @echo “Compilando Projeto” @echo “O patrao foi compilado” #A linha que vai compilar sub-make @cd sub-make && $(MAKE) teste: $(OBJ) @$(CC) -o $@ $^ %.o: %.c @$(CC) -o $@ -c $< $(CFLAGS) main.o: main.c teste.h @$(CC) -o $@ -c $< $(CFLAGS) #Entao depois e so executar make e depois make install install:all @mv $(EXEC) $(bindir)/ .PHONY: clean mrproper clean: @rm -rf *.o *~ # E a mesma coisa que dar um F4 dentro da pasta sub-make # e escrever make clean @cd sub-make && $(MAKE) $@ mrproper: clean @cd bin && rm -rf $(EXEC) #modificamos aqui tambem @cd sub-make && $(MAKE) $@

Então quando você digitar no console “make” depois “make install” ele vai colocar o binario que esta em “Projetos” dentro de “bin”.
Se você quiser colocar o binario que esta na pasta “submake” na pasta “bin”

- Copiar e colar no makefile da “sub-make” as variaveis “prefix” e “bindir"e a regra install:com seu comando.
- E no “Makefile principal” em baixo de “install:" coloque esta linha @cd sub-make && $(MAKE) $@
- Aqui eu modifiquei o “mrproper” porque agora os binarios que devem ser apagados com “make mrproper” estão em “bin”.
- Vou deixar voces modificarem o “mrproper” do “sub-makefile” como pessoas adultas e responsaveis ;) Valeu galera.

Os comandos no console são:

- make
- make install
- make clean
- make mrproper .Para apagar os binarios.

Capítulo 35

Lista de palavras reservadas

A linguagem C possui um total de 32 palavras conforme definido pelo padrão ANSI, que são elas:

É importante lembrar que todas as palavras reservadas são escritas em minúsculo e não podem ser utilizadas para outro propósito. Alguns compiladores incluem outras palavras reservadas como, **asm**, **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, **typeof**.

Capítulo 36

Seqüências de escape

O C tem várias seqüências de escape. Elas servem geralmente para inserir um caractere especial numa String.

Algumas dessas seqüências são:

- \a - Alarm, Alarme = Toca o alarme sonoro do sistema
- \b - Back space, Retrocesso = Apaga o caractere à esquerda do cursor
- \n - NewLine, Nova linha = Pula uma linha
- \t - Tabulação horizontal = Equivale à dar um TAB na string
- \r - Carriage Return, Retorno do Carro = Volta para o início da linha.
- \t - Horz. Tab, Tabulação Harizontal = Salta à frente conforme seus ajustes de tabulação
- \0 - Null, Nulo = Caractere nulo ou zero geralmente estabelecido como fim de string

Capítulo 37

Lista de funções

Aqui estão as várias funções presentes em C separadas por cabeçalho:

- stdio.h
  - printf
  - scanf
  - vsnprintf
  - sprintf
  - vprintf
  - fprintf
  - fscanf
  - feof
  - fflush
  - calloc
  - malloc
  - system
  - gets
  - fgets
  - puts
  - fputs
- stdlib.h
  - atoi
  - atof
  - atol
  - itoa
- string.h
  - strcmp
  - stricmp
  - strlen
  - strstr
  - strcat
  - strcpy
  - strncpy
  - strncat
  - strchr
  - strrev
- signal.h
- iso10646.h
- time.h
- math.h
  - tan
  - sin
  - cos
  - atan
  - asin
  - acos
  - pow
  - sqrt
  - abs

Capítulo 38

Lista de bibliotecas

Cabeçalhos de bibliotecas padrão ANSI C (C89)/ISO C (C90):

Cabeçalhos adicionados no ISO C (C94/C95):

Cabeçalhos adicionados no ISO C (C99) (suportados somente em compiladores mais novos):

38.1 Ligações externas

- The Open Group Base Specifications Issue 7 (english)
- Biblioteca C (english)

Capítulo 39

Dicas de programação em C

39.1 Convenções tipográficas

Uma das melhores maneiras de obter um código claro e usando identificadores coerentes.
Por exemplo é bom poder identificar rapidamente as variáveis em função de suas propriedades .
Veja abaixo algumas delas.

prefixos identificadores - ponteiro p_ - tabela estática(static array) a_ ou sa_ - tabela dinâmica (dynamic array) da_ - cadeia de caracteres(string) s_

Em um código com a variável "p_exemplo" podemos deduzir rapidamente que estamos usando um ponteiro .

39.2 A função printf é a melhor amiga de um programador

Um programador novato tende a ver apenas duas aplicações para o printf:

1. Solicitar entrada para o usuário do programa.
2. Imprimir o resultado do programa.

O fato é que um programador pode aplicar o printf a fim de saber o que ocorre durante a execução de programa. Isto permite, dentre outras coisas, detectar erros.

Por exemplo, suponha um programa no qual várias funções e rotinas são executadas. Algo como:

int main(int argc, char *argv[]) { ... funcao1(...); funcao2(...); funcao3(...); funcao4(...); ... return 0; }

Digamos que o programa tenha sido compilado com sucesso, mas ocorra algum erro durante sua execução. Podemos usar o printf para detectar o erro da seguinte maneira:

int main(int argc, char *argv[]) { ... printf("iniciando funcao1"); funcao1(...); printf("completa função1, iniciando funcao2"); funcao2(...); printf("completa função2, iniciando funcao3"); funcao3(...); printf("completa função3, iniciando funcao4"); funcao4(...); printf("completa função4"); ... return 0; }

Isto permite o programador determinar até que ponto o programa roda antes de dar erro, facilitando muito a detecção deste.

Outro exemplo de como o printf é útil na detecção de problemas. Suponha um programa cheio de laços aninhados. Tal como:

for(...) { while(...) { ... for(...) { ... } } }

Caso durante a execução o programa entre em um loop infinito, uma forma de detectar em qual dos laços está o problema é:

for(...) { printf("Teste 1"); while(...) { printf("Teste 2"); ... for(...) { printf("Teste 3"); ... } } }

A impressão que se repetir eternamente é aquela dentro do laço problemático.

Um último exemplo de detecção de problemas por meio do printf. Suponha que a resposta dada por um programa não é a esperada, que a resposta consiste na impressão de uma variável x, a qual recebe diversas atribuições ao longo do programa. Podemos identificar o erro dando um printf em x após cada uma de suas atribuições:

x=... printf("primeira atribuicao de x eh %tipo", x); ... x=... printf("segunda atribuicao de x eh %tipo", x); ... x=... printf("terceira atribuicao de x eh %tipo", x); ... printf("A resposta eh %tipo", x);

Caso o valor de x dependa do valor de outras variáveis que não são impressas, imprimi-las pode ajudar na detecção do problema.

Para uso como debug, a linguagem C apresenta duas macros que quando utilizadas junto com o printf são ótimos recursos.

- __FILE__ = nome do arquivo.
- __LINE__ = numero da linha de execuçãão.

O Compilador gcc ainda dispõe de uma outra macro bastante util:

- __PRETTY_FUNCTION__ = nome da função atual.

... printf("%d:%s:%s\n", __LINE__, __FILE__, __PRETTY_FUNCTION__); ...

O trecho acima vai te dar uma saida para debug muito util com o seguinte conteudo:

Exemplo: 3:hello.c:main

39.3 Tecle 1 para rodar

Existem duas formas de manter um programa rodando enquanto o usuário desejar:

1. Conter a maior parte do programa dentro de um laço.
2. Usar o comando goto(lembre-se que o comando goto não é de uso aconselhado para a programação estruturada).

Alguns exemplos:

Com while:

int main(int argc, char *argv[]) { int rodando=1; while(rodando==1)/*Este laco mantem o programa rodando enquanto o usuario desejar*/ { ... printf("\nDigite 1 para continuar rodando o programa."); printf("\nDigite qualquer outro numero para encerrar o programa. "); scanf("%d", &rodando); } return 0; }

Com do...while

int main(int argc, char *argv[]) { short int rodando; do /*Este laco mantem o programa rodando enquanto o usuario desejar*/ { ... printf("\nDigite 1 para manter o programa rodando. "); scanf("%d", &rodando); }while(rodando==1); return 0; }

Com o goto

int main(int argc, char *argv[]) { MARCA: ... FIM: int y; printf("Tecle 1 para continuar rodando o programa. Tecle 0 para encerrar o programa\n"); scanf("%d",&y); if(y==1) { goto MARCA; } if(y!=1 && y!=0) { goto FIM; } return 0; }

Capítulo 40

Listas encadeadas

Listas encadeadas são estruturas de dados lineares e dinâmicas, a grande vantagem que elas possuem em relação ao uso de vetor é o fato de terem tamanho máximo relativamente infinito (o tamanho máximo é o da memória do computador), ao mesmo tempo que podem ter o tamanho mínimo de 1 elemento evitando o desperdício de memória.

40.1 Primitivas

Não existe nenhuma normalização quanto as primitivas usadas para a manipulação de uma lista.
Abaixo você pode ver uma lista com algumas delas .

- Colocar o índice sobre o primeiro elemento da lista.
- Colocar o índice sobre o último elemento da lista .
- Colocar o índice sobre o elemento que segue o elemento atual .
- Colocar o índice sobre o elemento que precede o elemento atual .
- Verificar se a lista está vazia : Se a lista estiver vazia retorna verdadeiro, se não, falso.
- Verificar se é o primeiro elemento : Retorna verdadeiro se o elemento atual é o primeiro, se não, falso.
- Verificar se é o último elemento : Retorna verdadeiro se o elemento atual é o último, se não, falso.
- Verificar o número de elementos da lista : Retorna o número de elementos da lista.
- Adicionar um elemento no início : Adicionar um elemento antes do primeiro elemento da lista .
- Adicionar um elemento no fim : Adicionar um elemento depois do último elemento da lista .
- Inserção : Inserir um elemento antes do elemento atual .
- Troca : Trocar o elemento atual .
- Remoção : Remover o elemento atual .
- Listar todos os elementos da lista .

40.2 Lista encadeada linear

Cada nó ou elemento de uma lista encadeada irá possuir guardar o valor do nó e o endereço do próximo nó. Em uma lista encadeada linear o ultimo elemento aponta para NULL .

struct No{ char *p_dados; struct No *p_prox; };

40.3 Iniciar uma lista

A função abaixo demonstra como iniciar uma lista criando o espaço da raiz na memória.

void criar_Lista(struct No **p_Raiz){ *p_Raiz = NULL; }

40.4 Inserção

Existem 3 tipos de inserção em uma lista, pode-se inserir no começo, no final ou entre dois elementos da lista.

40.4.1 Inserção no início

int inserir_No_Inicio(struct No **p_Raiz, char *p_String){ struct No *p_Novo; /** Alocação dinâmica da memoria */ if((p_Novo = (struct No *) malloc(sizeof(struct No))) == NULL){ puts(“Falta Memoria\n”); return -1 ; } p_Novo->p_dados = p_String; p_Novo->p_prox = *p_Raiz; *p_Raiz = p_Novo; }

40.4.2 Inserção no fim

int inserir_No_Fim(struct No **p_Raiz, char *p_String){ struct No *p_Novo; if((p_Novo = (struct No *) malloc(sizeof(struct No))) == NULL){ puts(“Falta Memoria\n”); return -1 ; } p_Novo->p_dados = p_String; p_Novo->p_prox = NULL; if(*p_Raiz == NULL) *p_Raiz = p_Novo; else{ struct No *e_atual;

/*@ Elemento atual*/ e_atual = *p_Raiz; /*@ Primeiro elemento*/ while(e_atual->p_prox != NULL){ e_atual = e_atual->p_prox; } e_atual->p_prox = p_Novo; } }

40.5 Remoção

Assim como na inserção também existem 3 tipos de remoção, no início, no fim ou entre dois elementos da lista.

40.5.1 Remoção no início

void remover_No_Inicio(struct No **p_Raiz){ if(*p_Raiz == NULL) printf("\nA lista ja esta vazia\n"); else{ struct No *p_atual; p_atual = *p_Raiz; *p_Raiz = (*p_Raiz)->p_prox; free(p_atual); } }

40.5.2 Remoção no fim

void remover_No_Fim(struct No **p_Raiz){ if(*p_Raiz == NULL) printf("\nA lista ja esta vazia"); else{ struct No *p_atual, *p_anterior ; p_atual = *p_Raiz; while(p_atual->p_prox != NULL){ p_anterior = p_atual ; p_atual = p_atual->p_prox; } p_anterior->p_prox = NULL; free(p_atual); } }

40.6 Exibição

40.6.1 Do fim para a raiz

void mostrar_Do_Fim_Para_Raiz(struct No *p_Raiz){ if(p_Raiz == NULL) printf("\nLista vazia"); else{ struct No *p_Atual_Corredor, *p_Atual_Fim; p_Atual_Corredor = p_Raiz; p_Atual_Fim = p_Raiz; while(p_Atual_Fim->p_prox != NULL){ //ir para o ultimo elemento p_Atual_Fim = p_Atual_Fim->p_prox; } while(p_Atual_Corredor != p_Atual_Fim){ if(p_Atual_Corredor->p_prox == p_Atual_Fim){ printf(" <- %s", p_Atual_Fim->p_dados); p_Atual_Fim = p_Atual_Corredor; p_Atual_Corredor = p_Raiz; } else p_Atual_Corredor = p_Atual_Corredor->p_prox; } printf(" <- %s", p_Atual_Fim->p_dados); } }

40.6.2 Da raiz para o fim

void mostrar_Da_Raiz_Para_Fim(struct No *p_Raiz){ if(p_Raiz == NULL) printf("\nLista vazia"); else{ struct No *p_atual; p_atual = *p_Raiz; while(p_atual != NULL){ printf("%s", p_atual->p_dados); p_atual = p_atual->p_prox; } } }

Capítulo 41

Pilha

41.1 Pilha

Pilha ou stack é uma lista linear em que todas as inserções e remoções de elemento só podem ser feitos em uma extremidade chamada topo.As pilhas também são chamadas de estruturas LIFO (Last In First Out) ou seja o último elemento inserido é o primeiro removido.

41.2 Construção do protótipo de um elemento da lista.

typedef struct Elemento_da_lista{ char *dados; struct Elemento_da_lista *proximo; }Elemento; struct Localizar{ Elemento *inicio; int tamanho; } Pilha;

41.3 Inicialização

void iniciar (Localizar *monte){ monte->inicio = NULL; monte->tamanho = 0; }

41.4 Inserir um elemento na pilha(push)

Algoritmo:
Declaração do elemento(s) a ser inserido.
Alocação da memória para o novo elemento
Inicializar o campo de dados.
Preencher o ponteiro inicio com o primeiro elemento
Colocar em dia o tamanho da pilha.

int empilhar(Localizar * monte, char *dados){ Elemento *novo_elemento; if ((novo_elemento = (Elemento *) malloc (sizeof (Elemento))) == NULL) return −1; if ((novo_elemento->dados = (char *) malloc (50 * sizeof (char))) == NULL) return −1; strcpy (novo_elemento->dados, dados); novo_elemento->proximo = monte->inicio; monte->inicio = novo_elemento; monte->tamanho++; }

41.5 Retirar um elemento da pilha (pop)

int desempilhar (Localizar *monte){ Elemento *p_elemento; if (monte->tamanho == 0) return −1; p_elemento = monte->inicio; monte->inicio = monte->inicio->proximo; free (p_elemento->dados); free (p_elemento); monte->tamanho--; return 0; }

41.6 Imprimir os elementos da pilha

void mostrar(Localizar * monte){ Elemento *atual; int i; atual = monte->inicio; for(i=0;i<monte->tamanho;++i){ printf("\t\t%s\n”, atual->dados); atual = atual->proximo; } }

Capítulo 42

Fila ou Queue

Capítulo 43

Fila

Uma fila ou queue em inglês é uma estrutura de dados que usa o método FIFO(acrônimo para First In, First Out, que em português significa primeiro a entrar, primeiro a sair).
A idéia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início.
Exemplo de fila em C:

#include <stdio.h> #include <string.h> #include <stdlib.h> void q_enter(void); void q_list(void); int q_store(char *ptr); int q_delete(void); int count = 0; //contador int store = 0; // proxima posição na fila int retrieve = 0; // recupera a posição da fila char *queue[100]; // vetor da fila int main() { int i = 0; for (i = 0; i < 100; i++) { queue[i] = NULL; } q_enter(); // entra os dados na fila printf("\n\nTodos os dados da fila (FIFO):\n"); q_list(); q_delete(); // Apaga a primeira entrada da fila printf("\n\nA fila depois delete(FIFO):\n"); q_list(); printf("\n\nNumero de elementos restantes na fila: %i \n", count); getchar(); // espera return 0; } void q_enter(void) { static char str[100], *ptr; puts("Pressione a tecla ENTER sem nome pra sair\n"); do { printf("Entre o nome:"); gets(str); ptr = (char *) malloc(strlen(str)); //alocar um espaço na memória strcpy(ptr,str); if (*str) { count++; q_store(ptr); // Guarda o endereço da seqüência de caracteres } } while (*str); //Sair se não houver uma entrada } // listar a fila void q_list(void) { int k; for (k = retrieve; k < store; k++) { printf("Elemento %d : %s \n",k+1,queue[k]); } } // Guarda os itens na fila int q_store(char *ptr) { if (store == 100) { printf("\nA lista esta cheia!\n"); return 0 ; } queue[store] = ptr; store++; // próximo índice da fila } // Apaga um item da fila int q_delete(void) { if (store == retrieve) { printf("\nA fila esta vazia!"); return 0 ; } count--; retrieve++; }

Capítulo 44

Árvores binárias

44.1 Arvore binária

Uma arvore binária é uma estrutura de dados que pode ser representada como uma hierarquia onde cada elemento é chamado de nó. O nó inicial ou o primeiro elemento é chamado de raiz. Em uma árvore binária um elemento pode ter um máximo de dois filhos no nível inferior denominados como sub-árvore esquerda e sub-árvore direita.Um nó sem filhos é chamado de folha. A profundidade de um nó é a distância deste nó até a raiz e a distancia entre a folha mais distante e a raiz é a altura da arvore.Um conjunto de nós com a mesma profundidade é denominado, nível da árvore.

44.2 Struct

struct No{ int numero; struct No *esquerda; struct No *direita; }; typedef struct No No;

44.3 Iniciar

void criarArvore(No **pRaiz){ *pRaiz = NULL; }

44.4 Inserção

void inserir(No **pRaiz, int numero){ if(*pRaiz == NULL){ *pRaiz = (No *) malloc(sizeof(No)); (*pRaiz)->esquerda = NULL; (*pRaiz)->direita = NULL; (*pRaiz)->numero = numero; }else{ if(numero < (*pRaiz)->numero) inserir(&(*pRaiz)->esquerda, numero); if(numero > (*pRaiz)->numero) inserir(&(*pRaiz)->direita, numero); } }

44.5 Remoção

No *MaiorDireita(No **no){ if((*no)->direita != NULL) return MaiorDireita(&(*no)->direita); else{ No *aux = *no; if((*no)->esquerda != NULL) // se nao houver essa verificacao, esse nó vai perder todos os seus filhos da esquerda! *no = (*no)->esquerda; else *no = NULL; return aux; } } No *MenorEsquerda(No **no){ if((*no)->esquerda != NULL) return MenorEsquerda(&(*no)->esquerda); else{ No *aux = *no; if((*no)->direita != NULL) // se nao houver essa verificacao, esse nó vai perder todos os seus filhos da direita! *no = (*no)->direita; else *no = NULL; return aux; } } void remover(No **pRaiz, int numero){ if(*pRaiz == NULL){ // esta verificacao serve para caso o numero nao exista na arvore. printf(“Numero nao existe na arvore!"); getch(); return; } if(numero < (*pRaiz)->numero) remover(&(*pRaiz)->esquerda, numero); else if(numero > (*pRaiz)->numero) remover(&(*pRaiz)->direita, numero); else{ // se nao eh menor nem maior, logo, eh o numero que estou procurando! :) No *pAux = *pRaiz; // quem programar no Embarcadero vai ter que declarar o pAux no inicio do void! :[if (((*pRaiz)->esquerda == NULL) && ((*pRaiz)->direita == NULL)){ // se nao houver filhos... free(pAux); (*pRaiz) = NULL; } else{ // so tem o filho da direita if ((*pRaiz)->esquerda == NULL){ (*pRaiz) = (*pRaiz)->direita; pAux->direita = NULL; free(pAux); pAux = NULL; } else{ //so tem filho da esquerda if ((*pRaiz)->direita == NULL){ (*pRaiz) = (*pRaiz)->esquerda; pAux->esquerda = NULL free(pAux); pAux = NULL; } else{ //Escolhi fazer o maior filho direito da subarvore esquerda. pAux = MaiorDireita(&(*pRaiz)->esquerda); //se vc quiser usar o Menor da esquerda, so o que mudaria seria isso: pAux->esquerda = (*pRaiz)->esquerda; // pAux = MenorEsquerda(&(*pRaiz)->direita); pAux->direita = (*pRaiz)->direita; (*pRaiz)->esquerda = (*pRaiz)->direita = NULL; free((*pRaiz)); *pRaiz = pAux; pAux = NULL; } } } } }

44.5.1 Em ordem

void exibirEmOrdem(No *pRaiz){ if(pRaiz != NULL){ exibirEmOrdem(pRaiz->esquerda); printf("\n%i", pRaiz->numero); exibirEmOrdem(pRaiz->direita); } }

44.5.2 Pré-ordem

void exibirPreOrdem(No *pRaiz){ if(pRaiz != NULL){ printf("\n%i", pRaiz->numero); exibirPreOrdem(pRaiz->esquerda); exibirPreOrdem(pRaiz->direita); } }

44.5.3 Pós-ordem

void exibirPosOrdem(No *pRaiz){ if(pRaiz != NULL){ exibirPosOrdem(pRaiz->esquerda); exibirPosOrdem(pRaiz->direita); printf("\n%i", pRaiz->numero); } }

44.6 Contar nós

int contarNos(No *pRaiz){ if(pRaiz == NULL) return 0; else return 1 + contarNos(pRaiz->esquerda) + contarNos(pRaiz->direita); }

44.7 Contar folhas

int contarFolhas(No *pRaiz){ if(pRaiz == NULL) return 0; if(pRaiz->esquerda == NULL && pRaiz->direita == NULL) return 1; return contarFolhas(pRaiz->esquerda) + contarFolhas(pRaiz->direita); }

44.8 Altura da árvore

int maior(int a, int b){ if(a > b) return a; else return b; } int altura(No *pRaiz){ if((pRaiz == NULL) || (pRaiz->esquerda == NULL && pRaiz->direita == NULL)) return 0; else return 1 + maior(altura(pRaiz->esquerda), altura(pRaiz->direita)); }

44.9 Estrutura Completa

Capítulo 45

Algoritmos de ordenação

Capítulo 46

Insertion sort

```
void insertion_sort(int tabela[], int largura) { int i, memoria, contador; bool marcador; for(i=1; i<largura; i++) { memoria = tabela[i]; contador = i-1; do { marcador = false; if(tabela[contador] > memoria) { tabela[contador+1] = tabela[contador]; contador--; marcador = true; } if(contador < 0) marcador = false; } while(marcador); } tabela[contador+1] = memoria;
```

Capítulo 47

Selection sort

```
void selectionSort( int vetorDesordenado[], int tamanhoVetor ) //Funçao selection recebendo vetor e tamanho { int i, j, posicaoValorMinimo; for (i = 0; i < ( tamanhoVetor - 1 ); i++) //Loop para percorrer o vetor { posicaoValorMinimo = i; //O valor minimo de posiçao do vetor a ser percorrido e 0 for (j = ( i + 1 ); j < tamanhoVetor; j++)//Percorreremos o vetor da posiçao 1 ate o tamanho estimado { if( vetorDesordenado[j] < vetorDesordenado[posicaoValorMinimo] ) //Se a posiçao que vamos verificar for menos que a posiçao que temos em maos { posicaoValorMinimo = j;//A variavel 'j' recebe esse valor } } if ( i != posicaoValorMinimo ) { trocarPosicaoValores( &vetorDesordenado[i], &vetorDesordenado[posicaoValorMinimo] );//vamos chamar uma outra funçao para trocar as posiçoes de lugares } } }
void trocarPosicaoValores( int *posicaoA, int *posicaoB )//Funçao para trocar as posiçoes que estamos olhando { int temporario; temporario = *posicaoA; *posicaoA = *posicaoB; *posicaoB = temporario; }
```

Capítulo 48

Bubble sort

O ***bubble sort***, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa n^2 operações relevantes, onde *n* representa o número de elementos do vetor. No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de Ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

48.0.1 Código da Função

void BubbleSort(int vetor[], int tamanho) { int aux, i, j; for(j=tamanho-1; j>=1; j--) { for(i=0; i<j; i++) { if(vetor[i]>vetor[i+1]) { aux=vetor[i]; vetor[i]=vetor[i+1]; vetor[i+1]=aux; } } } }

48.0.2 Código da Função Melhorado

Termina a execução quando nenhuma troca é realizada após uma passada pelo vetor.

void BubbleSort(int vetor[], int tamanho) { int memoria, troca, i, j; troca=1; /*A variável "troca" será a verificação da troca em cada passada*/ for(j=tamanho-1; (j>=1) && (troca==1); j--) { troca=0; /*Se o valor continuar 0 na próxima passada quer dizer que não houve troca e a função é encerrada.*/ for(i=0; i<j; i++) { if(vetor[i]>vetor[i+1]) { memoria=vetor[i]; vetor[i]=vetor[i+1]; vetor[i+1]=memoria; troca=1; /*Se houve troca, "troca" recebe 1 para continuar rodando.*/ } } } }

Capítulo 49

Algoritmo de alocação

49.1 first fist

49.2 best fit

Varre toda a memória e escolhe a página mais ajustada ao tamanho do processo.

#include <stdio.h> #include <windows.h> int main(){ int p,m; printf(“Entre o numero de processos:"); scanf("%d”,&p); printf(“Entre o numero de blocos de memoria:"); scanf("%d”,&m); int parr[p]; struct memoria{ int id; // identificador int tamanho; }marr[m]; int i; for(i=0;i<p;i++) { printf(“Entre o tamanho do processo %d:",i+1); scanf("%d”,&parr[i]); } for(i=0;i<m;i++) { printf(“Entre o tamanho do bloco de memoria %d:",i+1); scanf("%d”,&marr[i].tamanho); marr[i].id=i+1; } int j; int tamanho = 0; for(i; tamanho <= marr[i].tamanho; i++) tamanho = marr[i].tamanho; int tamanho_velho = tamanho ; int im ; bool marcador = false ; for(i=0;i<p;i++){ for(j=0;j<m;j++){ if((marr[j].tamanho>=parr[i]) && (marr[j].tamanho < tamanho)){ im = j; tamanho = marr[j].tamanho; marcador = true ; } } if(marcador){ marcador = false ; marr[im].tamanho-=parr[i]; tamanho = tamanho_velho ; printf("\n\nAloca o processo %d no bloco memoria %d\n Tamanho restante apos alocar %d\n”,i+1,marr[im].id,marr[im].tamanho); }else {printf(“Memoria insuficiente para o processo %d”,i);break;} } system (“pause”); return 0; }

49.3 worst fit

O algoritmo worst fit aloca o bloco de memória na região que tem o maior espaço livre.

Está técnica por procurar ocupar primeiro as partições maiores termina por deixar espaços livres que poderiam ser utilizados para que outros blocos de outros programas as utilizassem, diminuindo e/ou retardando a fragmentação.

49.4 Next Fit

49.5 Buddy System

Capítulo 50

Lista de autores

50.1 Lista de autores

- Edudobay - Eduardo Sangiorgio Dobay
- EvertonS - Everton.S.Baron
- fabiobasso - Fábio Basso
- Lightningspirit
- ThiagoL
- Uder
- Wbrito
- RenatoResende
- Maxtremus
- Noturno99 - Bruno Sampaio Pinho da Silva

50.2 Fontes, contribuidores e licenças de texto e imagem

50.2.1 Texto

- **Programar em C/Capa** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Capa?oldid=275857 *Contribuidores:* Joaodaveiro, Lightningspirit, Jorge Morais, SallesNeto BR, Edudobay, Wbrito, Master, Voz da Verdade, Delemon, David Stress~ptwikibooks, ???? robot, EvertonS, Elvire, He7d3r.bot e Anónimo: 9
- **Programar em C/Por que aprender a linguagem C** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Por_que_aprender_a_linguagem_C?oldid=271298 *Contribuidores:* João Jerónimo, Lightningspirit, Jorge Morais, SallesNeto BR, Edudobay, Sourf, Wbrito, Thiagol, He7d3r.bot, Fabiobasso, Abacaxi e Anónimo: 10
- **Programar em C/História da linguagem C** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Hist%C3%B3ria_da_linguagem_C?oldid=271327 *Contribuidores:* Lightningspirit, Jorge Morais, Scorpion~ptwikibooks, Edudobay, He7d3r, He7d3r.bot, JackPotte, Abacaxi e Anónimo: 14
- **Programar em C/Pré-requisitos** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Pr%C3%A9-requisitos?oldid=271305 *Contribuidores:* Marcos Antônio Nunes de Moura, Lightningspirit, Jorge Morais, Edudobay, Wbrito, Albmont, He7d3r, He7d3r.bot, Abacaxi e Anónimo: 13
- **Programar em C/Utilizando um compilador** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Utilizando_um_compilador?oldid=289238 *Contribuidores:* Marcos Antônio Nunes de Moura, Edudobay, Master, Albmont, Thiagol, EvertonS, He7d3r.bot, JackBot, Fabiobasso, Abacaxi, Wesley Ferdinando R. Carvalho, Rodrigo Leite Valentin e Anónimo: 4
- **Programar em C/Noções de compilação** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/No%C3%A7%C3%B5es_de_compila%C3%A7%C3%A3o?oldid=212773 *Contribuidores:* SallesNeto BR, Edudobay, Wbrito, Thiagol, PatiBot, He7d3r.bot e Aprendiz de feiticeiro
- **Programar em C/Um programa em C** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Um_programa_em_C?oldid=263594 *Contribuidores:* Marcos Antônio Nunes de Moura, Jorge Morais, Edudobay, Wbrito, Thiagol, EvertonS, Awillian~ptwikibooks, He7d3r.bot, Fabiobasso e Anónimo: 11
- **Programar em C/Conceitos básicos** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Conceitos_b%C3%A1sicos?oldid=288078 *Contribuidores:* Edudobay, Wbrito, Thiagol, He7d3r, He7d3r.bot, Alguém, Fabiobasso e Anónimo: 4
- **Programar em C/Variáveis** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Vari%C3%A1veis?oldid=271328 *Contribuidores:* Marcos Antônio Nunes de Moura, Jorge Morais, Edudobay, Wbrito, Thiagol, He7d3r, EvertonS, Mr.Yahoo!, He7d3r.bot, JackPotte, Defender, Abacaxi e Anónimo: 13
- **Programar em C/Tipos de dados** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Tipos_de_dados?oldid=272760 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, Marcelo-Silva, Wbrito, Master, Raylton P. Sousa, He7d3r.bot, Fabiobasso, Abacaxi, PODEROS ARAN e Anónimo: 12
- **Programar em C/Constantes** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Constantes?oldid=256737 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, SallesNeto BR, Marcelo-Silva, Wbrito, Master, He7d3r, Abacaxi, Iraziel e Anónimo: 6
- **Programar em C/Entrada e saída simples** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Entrada_e_sa%C3%ADda_simples?oldid=281484 *Contribuidores:* Marcos Antônio Nunes de Moura, Edudobay, Wbrito, Albmont, Thiagol, EvertonS, He7d3r.bot, Yuu eo, Abacaxi e Anónimo: 13
- **Programar em C/Operações matemáticas (Básico)** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Opera%C3%A7%C3%B5es_matem%C3%A1ticas_(B%C3%A1sico)?oldid=248640 *Contribuidores:* Edudobay, Wbrito, He7d3r.bot, Abacaxi e Anónimo: 4
- **Programar em C/Operações matemáticas (Avançado)** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Opera%C3%A7%C3%B5es_matem%C3%A1ticas_(Avan%C3%A7ado)?oldid=270610 *Contribuidores:* Marcos Antônio Nunes de Moura, SallesNeto BR, Wbrito, Thiagol, Rogerboff, He7d3r.bot e Anónimo: 5
- **Programar em C/Operadores** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Operadores?oldid=291798 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, Marcelo-Silva, Wbrito, Master, Petrusz1, Raylton P. Sousa, He7d3r.bot, Abacaxi e Anónimo: 6
- **Programar em C/Controle de fluxo** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Controle_de_fluxo?oldid=267376 *Contribuidores:* Edudobay, Wbrito, Albmont, Thiagol, He7d3r, Rogerboff, He7d3r.bot, Hycesar, Abacaxi e Anónimo: 8
- **Programar em C/Funções** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Fun%C3%A7%C3%B5es?oldid=291823 *Contribuidores:* Marcos Antônio Nunes de Moura, Edudobay, Wbrito, Albmont, Rogerboff, EvertonS, Awillian~ptwikibooks, He7d3r.bot, Fabiobasso, Hycesar, Victor Aurélio, Abacaxi, Cleiton wi e Anónimo: 26
- **Programar em C/Pré-processador** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Pr%C3%A9-processador?oldid=266815 *Contribuidores:* Lgrave, SallesNeto BR, Edudobay, Rogerboff, EvertonS, He7d3r.bot, Hycesar, Abacaxi e Anónimo: 2
- **Programar em C/Exercícios** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Exerc%C3%ADcios?oldid=290510 *Contribuidores:* Marcos Antônio Nunes de Moura, Lightningspirit, Jorge Morais, Wbrito, Albmont, Delemon, Raylton P. Sousa, He7d3r.bot, Alguém, Abacaxi, Artur Filipe Kalopa Daniel, Forshaw2 e Anónimo: 8
- **Programar em C/Vetores** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Vetores?oldid=272709 *Contribuidores:* Marcos Antônio Nunes de Moura, Dante Cardoso Pinto de Almeida, Edudobay, Wbrito, He7d3r, EvertonS, He7d3r.bot, Jonas AGX, Ajraddatz, Fabiobasso, Hycesar, Abacaxi e Anónimo: 20
- **Programar em C/Strings** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Strings?oldid=288271 *Contribuidores:* Marcos Antônio Nunes de Moura, Jorge Morais, Edudobay, Wbrito, Albmont, PatiBot, He7d3r.bot, Defender, Stryn, Abacaxi e Anónimo: 14
- **Programar em C/Passagem de parâmetros** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Passagem_de_par%C3%A2metros?oldid=245335 *Contribuidores:* Wbrito, David Stress~ptwikibooks, EvertonS, He7d3r.bot, Abacaxi e Anónimo: 3
- **Programar em C/Tipos de dados definidos pelo usuário** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Tipos_de_dados_definidos_pelo_usu%C3%A1rio?oldid=234520 *Contribuidores:* Edudobay, Wbrito, He7d3r.bot e Anónimo: 5

- **Programar em C/Enumeração** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Enumera%C3%A7%C3%A3o?oldid=255616 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, Marcelo-Silva, Wbrito, Master, He7d3r, EvertonS, Abacaxi e Anónimo: 10
- **Programar em C/União** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Uni%C3%A3o?oldid=255617 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, Marcelo-Silva, Wbrito, Master, He7d3r.bot, Abacaxi e Anónimo: 4
- **Programar em C/Estruturas** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Estruturas?oldid=281604 *Contribuidores:* Marcos Antônio Nunes de Moura, Daveiro, Jorge Morais, Marcelo-Silva, Wbrito, Master, Albmont, EvertonS, He7d3r.bot, Abacaxi e Anónimo: 19
- **Programar em C/Ponteiros** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Ponteiros?oldid=291505 *Contribuidores:* Marcos Antônio Nunes de Moura, Jorge Morais, Edudobay, Wbrito, Albmont, EvertonS, Jesielt, He7d3r.bot, Noturno99, Fabiobasso, Hycesar, Abacaxi, Júnior D. Eskelsen,, C++NERD, Su~ptwikibooks e Anónimo: 21
- **Programar em C/Mais sobre variáveis** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Mais_sobre_vari%C3%A1veis?oldid=273621 *Contribuidores:* Edudobay, Wbrito, He7d3r, GabrielFalcao, PatiBot, He7d3r.bot, Abacaxi e Anónimo: 4
- **Programar em C/Mais sobre funções** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Mais_sobre_fun%C3%A7%C3%B5es?oldid=250214 *Contribuidores:* EvertonS e Abacaxi
- **Programar em C/Bibliotecas** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Bibliotecas?oldid=265808 *Contribuidores:* Edudobay, Wbrito, He7d3r, Rogerboff, EvertonS, He7d3r.bot, Torneira e Anónimo: 4
- **Programar em C/Entrada e saída em arquivos** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Entrada_e_sa%C3%ADda_em_arquivos?oldid=291179 *Contribuidores:* Marcos Antônio Nunes de Moura, Edudobay, Wbrito, EvertonS, PatiBot, He7d3r.bot, MateusGPe, Abacaxi e Anónimo: 6
- **Programar em C/Gerenciamento de memória** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Gerenciamento_de_mem%C3%B3ria?oldid=266464 *Contribuidores:* Marcos Antônio Nunes de Moura, Edudobay, PatiBot, He7d3r.bot, Frigotoni, Abacaxi, Gabrielhtec e Anónimo: 7
- **Programar em C/Sockets** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Sockets?oldid=253321 *Contribuidores:* Jorge Morais, Albmont, EvertonS, He7d3r.bot, Abacaxi e Anónimo: 6
- **Programar em C/Makefiles** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Makefiles?oldid=272765 *Contribuidores:* Jorge Morais, David Stress~ptwikibooks, He7d3r, EvertonS, He7d3r.bot e Anónimo: 31
- **Programar em C/Lista de palavras reservadas** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Lista_de_palavras_reservadas?oldid=186025 *Contribuidores:* Jorge Morais, He7d3r.bot e Anónimo: 1
- **Programar em C/Seqüências de escape** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Seq%C3%BC%C3%AAncias_de_escape?oldid=186035 *Contribuidores:* SallesNeto BR, Master, Devarde, He7d3r.bot e Anónimo: 1
- **Programar em C/Lista de funções** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Lista_de_fun%C3%A7%C3%B5es?oldid=186024 *Contribuidores:* SallesNeto BR, Master, Devarde e He7d3r.bot
- **Programar em C/Lista de bibliotecas** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Lista_de_bibliotecas?oldid=254253 *Contribuidores:* Marcos Antônio Nunes de Moura, Jorge Morais, EvertonS, He7d3r.bot, Abacaxi e Anónimo: 2
- **Programar em C/Dicas de programação em C** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Dicas_de_programa%C3%A7%C3%A3o_em_C?oldid=258291 *Contribuidores:* Dante Cardoso Pinto de Almeida, He7d3r, He7d3r.bot, Abacaxi e Anónimo: 6
- **Programar em C/Listas encadeadas** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Listas_encadeadas?oldid=291714 *Contribuidores:* Jorge Morais, Maxtremus, EvertonS, Ruy Pugliesi, He7d3r.bot, Abacaxi, Gabrielhtec e Anónimo: 15
- **Programar em C/Pilha** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Pilha?oldid=276545 *Contribuidores:* Marcos Antônio Nunes de Moura, He7d3r.bot, Lukas[23], Augustowebd.perito e Anónimo: 4
- **Programar em C/Fila ou Queue** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Fila_ou_Queue?oldid=275858 *Contribuidores:* EvertonS, Defender, Abacaxi e Anónimo: 3
- **Programar em C/Árvores binárias** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/%C3%81rvores_bin%C3%A1rias?oldid=390030 *Contribuidores:* Marcos Antônio Nunes de Moura, Maxtremus, EvertonS, Ruy Pugliesi, He7d3r.bot, Wiki13, Abacaxi, Aldnonymous, JefersonM123 e Anónimo: 30
- **Programar em C/Algoritmos de ordenação** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Algoritmos_de_ordena%C3%A7%C3%A3o?oldid=290285 *Contribuidores:* Marcos Antônio Nunes de Moura, EvertonS, He7d3r.bot, Abacaxi e Anónimo: 7
- **Programar em C/Algoritmo de alocação** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Algoritmo_de_aloca%C3%A7%C3%A3o?oldid=270419 *Contribuidores:* Jorge Morais, EvertonS, He7d3r.bot, Torneira e Anónimo: 3
- **Programar em C/Lista de autores** *Fonte:* https://pt.wikibooks.org/wiki/Programar_em_C/Lista_de_autores?oldid=290231 *Contribuidores:* Marcos Antônio Nunes de Moura, EvertonS, He7d3r.bot, RenatoResende, Fabiobasso, Cardinhotk e Anónimo: 4

50.2.2 Imagens

- **Ficheiro:Crystal_Clear_app_kaddressbook.png** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/7/74/Crystal_Clear_app_kaddressbook.png *Licença:* LGPL *Contribuidores:* All Crystal Clear icons were posted by the author as LGPL on kde-look; *Artista original:* Everaldo Coelho and YellowIcon;
- **Ficheiro:Dev-C++_(cs).png** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/c/c1/Dev-C%2B%2B_%28cs%29.png *Licença:* GPL *Contribuidores:* Obra do próprio *Artista original:* cs:User:DaBler
- **Ficheiro:EsquemaPonteiro.png** *Fonte:* https://upload.wikimedia.org/wikibooks/pt/1/12/EsquemaPonteiro.png *Licença:* ? *Contribuidores:* ? *Artista original:* ?

- **Ficheiro:Exercicios_c_cover.png** *Fonte:* https://upload.wikimedia.org/wikibooks/pt/0/01/Exercicios_c_cover.png *Licença:* ? *Contribuidores:* ? *Artista original:* ?
- **Ficheiro:Ken_n_dennis.jpg** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/3/36/Ken_n_dennis.jpg *Licença:* Public domain *Contribuidores:* http://www.catb.org/~{}esr/jargon/html/U/Unix.html *Artista original:* Desconhecido
- **Ficheiro:Merge-arrows.svg** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/5/52/Merge-arrows.svg *Licença:* Public domain *Contribuidores:* ? *Artista original:* ?
- **Ficheiro:Nuvola_apps_konsole.png** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/2/24/Nuvola_apps_konsole.png *Licença:* LGPL *Contribuidores:* http://icon-king.com *Artista original:* David Vignoni / ICON KING
- **Ficheiro:Programar_c_cover.png** *Fonte:* https://upload.wikimedia.org/wikibooks/pt/6/6d/Programar_c_cover.png *Licença:* ? *Contribuidores:* ? *Artista original:* ?
- **Ficheiro:Recycle001.svg** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/4/44/Recycle001.svg *Licença:* Public domain *Contribuidores:* Originally from en.wikipedia; description page is (was) here *Artista original:* Users Cbuckley, Jpowell on en.wikipedia
- **Ficheiro:Searchtool.svg** *Fonte:* https://upload.wikimedia.org/wikipedia/commons/6/61/Searchtool.svg *Licença:* LGPL *Contribuidores:* http://ftp.gnome.org/pub/GNOME/sources/gnome-themes-extras/0.9/gnome-themes-extras-0.9.0.tar.gz *Artista original:* David Vignoni, Ysangkok

50.2.3 Licença

- Creative Commons Attribution-Share Alike 3.0