MAKRON
Books

Parte 1

A Linguagem C

A primeira parte deste livro apresenta uma discussão completa da linguagem de programação C. O Capítulo 1 fornece uma rápida exposição da linguagem C — o programador mais experiente talvez queira passar diretamente para o Capítulo 2. O Capítulo 2 examina os tipos de dados internos, variáveis, operadores e expressões. O Capítulo 3 apresenta os comandos de controle do programa. O Capítulo 4 discute matrizes e strings. O Capítulo 5 trabalha com ponteiros. O Capítulo 6 discute funções. O Capítulo 7 aborda estruturas, uniões e os tipos definidos pelo usuário. O Capítulo 8 examina as E/S pelo console. O Capítulo 9 aborda as E/S de arquivo e, finalmente, o Capítulo 10 discute o pré-processador e faz comentários.

O assunto desta parte (e a maior parte do material deste livro) reflete o padrão ANSI para C. No entanto, o padrão original de C, oriundo do UNIX versão 5, também é focalizado, e as diferenças mais importantes são salientadas. O livro aborda tanto o C ANSI quanto o original como garantia de que você encontrará informações pertinentes ao seu ambiemte de programação em C.

1

Uma Visão Geral de C

A finalidade deste capítulo é apresentar uma visão geral da linguagem de programação C, suas origens, seus usos e sua filosofia. Este capítulo destina-se principalmente aos novatos em C.

As Origens de C

A linguagem C foi inventada e implementada primeiramente por Dennis Ritchie em um DEC PDP-11 que utilizava o sistema operacional UNIX. C é o resultado de um processo de desenvolvimento que começou com uma linguagem mais antiga, chamada BCPL, que ainda está em uso, em sua forma original, na Europa. BCPL foi desenvolvida por Martin Richards e influenciou uma linguagem chamada B, inventada por Ken Thompson. Na década de 1970, B levou ao desenvolvimento de C.

Por muitos anos, de fato, o padrão para C foi a versão fornecida com o sistema operacional UNIX versão 5. Ele é descrito em *The C Programming Language*, de Brian Kernighan e Dennis Ritchie (Englewood Cliffs, N.J.: Prentice Hall, 1978). Com a popularidade dos microcomputadores, um grande número de implementações de C foi criado. Quase que por milagre, os códigos-fontes aceitos por essas implementações eram altamente compatíveis. (Isto é, um programa escrito com um deles podia normalmente ser compilado com sucesso usando-se um outro.) Porém, por não existir nenhum padrão, havia discrepâncias. Para remediar essa situação, o ANSI (American National Standards Institute) estabeleceu, no verão de 1983, um comitê para criar um padrão que definiria de uma vez por todas a linguagem C. No momento em que esta obra foi escrita, o comitê

do padrão ANSI estava concluindo o processo formal de adoção. Todos os principais compiladores C já implementaram o padrão C ANSI. Este livro aborda totalmente o padrão ANSI e enfatiza-o. Ao mesmo tempo, ele contém informações sobre a antiga versão UNIX de C. Em outras palavras, independentemente do compilador que esteja usando, você encontrará assuntos aplicáveis aqui.

C É uma Linguagem de Médio Nível

C é freqüentemente chamada de linguagem de médio nível para computadores. Isso não significa que C seja menos poderosa, difícil de usar ou menos desenvolvida que uma linguagem de alto nível como BASIC e Pascal, tampouco implica que C seja similar à linguagem assembly e seus problemas correlatos aos usuários. C é tratada como uma linguagem de médio nível porque combina elementos de linguagens de alto nível com a funcionalidade da linguagem assembly. A Tabela 1.1 mostra como C se enquadra no espectro das linguagens de computador.

Tabela 1.1 A posição de C no mundo das linguagens.

Nível mais alto	Ada
	Modula-2
	Pascal
	COBOL
	FORTRAN
	BASIC
Médio nível	C++
	C
	FORTH
	Macro-assembler
Nível mais baixo	Assembler

Como uma linguagem de médio nível, C permite a manipulação de bits, bytes e endereços — os elementos básicos com os quais o computador funciona. Um código escrito em C é muito portável. *Portabilidade* significa que é possível adaptar um software escrito para um tipo de computador a outro. Por exemplo, se você pode facilmente converter um programa escrito para DOS de tal forma a executar sob Windows, então esse programa é portável.

Todas as linguagens de programação de alto nível suportam o conceito de tipos de dados. Um *tipo de dado* define um conjunto de valores que uma variável pode armazenar e o conjunto de operações que pode ser executado com

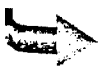
essa variável. Tipos de dados comuns são inteiro, caractere e real. Embora C tenha cinco tipos de dados internos, ela não é uma linguagem rica em tipos de dados como Pascal e Ada. C permite quase todas conversões de tipos. Por exemplo, os tipos caractere e inteiro podem ser livremente misturados na maioria das expressões. C não efetua nenhuma verificação no tempo de execução, como a validação dos limites das matrizes. Esses tipos de verificações são de responsabilidade do programador.

As versões originais de C não realizavam muitos (se é que realizavam algum) testes de compatibilidade entre um parâmetro de uma função e o argumento usado para chamar a função. Por exemplo, na versão original de C, você poderia chamar uma função, usando um ponteiro, sem gerar uma mensagem de erro, mesmo que essa função tivesse sido definida, na realidade, como recebendo um argumento em ponto flutuante. No entanto, o padrão ANSI introduziu o conceito de *protótipos de funções*, que permite que alguns desses erros em potencial sejam mostrados, conforme a intenção do programador. (Protótipos serão discutidos mais tarde no Capítulo 6.)

Outro aspecto importante de C é que ele tem apenas 32 palavras-chaves (27 do padrão de fato estabelecido por Kernighan e Ritchie, mais 5 adicionadas pelo comitê ANSI de padronização), que são os comandos que compõem a linguagem C. As linguagens de alto nível tipicamente têm várias vezes esse número de palavras reservadas. Como comparação, considere que a maioria das versões de BASIC possuem bem mais de 100 palavras reservadas!

C É uma Linguagem Estruturada

Embora o termo *linguagem estruturada em blocos* não seja rigorosamente aplicável a C, ela é normalmente referida simplesmente como linguagem estruturada. C tem muitas semelhanças com outras linguagens estruturadas, como ALGOL, Pascal e Modula-2.



NOTA: A razão pela qual C não é, tecnicamente, uma linguagem estruturada em blocos, é que as linguagens estruturadas em blocos permitem que procedimentos e funções sejam declarados dentro de procedimentos e funções. No entanto, como C não permite a criação de funções dentro de funções, não pode ser chamada formalmente de uma linguagem estruturada em blocos.

A característica especial de uma linguagem estruturada é a *compartimentalização* do código e dos dados. Trata-se da habilidade de uma linguagem seccionar e esconder do resto do programa todas as informações necessárias para se realizar uma tarefa específica. Uma das maneiras de conseguir

essa compartimentalização é pelo uso de sub-rotinas que empregam variáveis locais (temporárias). Com o uso de variáveis locais é possível escrever sub-rotinas de forma que os eventos que ocorrem dentro delas não causem nenhum efeito inesperado nas outras partes do programa. Essa capacidade permite que seus programas em C compartilhem facilmente seções de código. Se você desenvolve funções compartimentalizadas, só precisa saber o que uma função faz, não como ela faz. Lembre-se de que o uso excessivo de variáveis globais (variáveis conhecidas por todo o programa) pode trazer muitos erros, por permitir efeitos colaterais indesejados. (Qualquer um que já tenha programado em BASIC está bem ciente deste problema.)

Uma linguagem estruturada permite muitas possibilidades na programação. Ela suporta, diretamente, diversas construções de laços (loops), como **while**, **do-while** e **for**. Em uma linguagem estruturada, o uso de **goto** é proibido ou desencorajado e também a forma comum de controle do programa, (que ocorre em BASIC e FORTRAN, por exemplo). Uma linguagem estruturada permite que você insira sentenças em qualquer lugar de uma linha e não exige um conceito rigoroso de campo (como em FORTRAN).

A seguir estão alguns exemplos de linguagens estruturadas e não estruturadas.

Não estruturadas	**Estruturadas**
FORTRAN	Pascal
BASIC	Ada
COBOL	C++
	C
	Modula-2

Linguagens estruturadas tendem a ser modernas. De fato, a marca de uma linguagem antiga de computador é não ser estruturada. Hoje, a maioria dos programadores considera as linguagens estruturadas mais fáceis de programar e fazer manutenção.

O principal componente estrutural de C é a função — a sub-rotina isolada de C. Em C, funções são os blocos de construção em que toda a atividade do programa ocorre. Elas admitem que você defina e codifique separadamente as diferentes tarefas de um programa, permitindo, então, que seu programa seja modular. Após uma função ter sido criada, você pode esperar que ela trabalhe adequadamente em várias situações, sem criar efeitos inesperados em outras partes do programa. O fato de você poder criar funções isoladas é extremamente importante em projetos maiores nos quais um código de um programador não deve afetar acidentalmente o de outro.

Uma outra maneira de estruturar e compartimentalizar o código em C é pelo uso de blocos de código. Um *bloco de código* é um grupo de comandos de programa conectado logicamente que é tratado como uma unidade. Em C, um bloco de código é criado colocando-se uma seqüência de comandos entre chaves. Neste exemplo,

```
if (x < 10)  {
   printf("muito baixo, tente novamente\n");
   scanf("%d", &x);
}
```

os dois comandos após o **if** e entre chaves são executados se **x** for menor que 10. Esses dois comandos, junto com as chaves, representam um bloco de código. Eles são uma unidade lógica: um dos comandos não pode ser executado sem que o outro também seja. Atente para o fato de que todo comando em C pode ser um comando simples ou um bloco de comandos. Blocos de código permitem que muitos algoritmos sejam implementados com clareza, elegância e eficiência. Além disso, eles ajudam o programador a conceituar a verdadeira natureza da rotina.

C É uma Linguagem para Programadores

Surpreendentemente, nem todas as linguagens de computador são para programadores. Considere os exemplos clássicos de linguagens para não-programadores: COBOL e BASIC. COBOL não foi destinada para facilitar a vida do programador, aumentar a segurança do código produzido ou a velocidade em que o código pode ser escrito. Ao contrário, COBOL foi concebida, em parte, para permitir que não-programadores leiam e presumivelmente (embora isso seja improvável) entendam o programa. BASIC foi criada essencialmente para permitir que não-programadores programem um computador para resolver problemas relativamente simples.

Em contraposição, C foi criada, influenciada e testada em campo por programadores profissionais. O resultado final é que C dá ao programador o que ele quer: poucas restrições, poucas reclamações, estruturas de bloco, funções isoladas e um conjunto compacto de palavras-chave. Usando C, um programador pode conseguir aproximadamente a eficiência de código assembly combinada com a estrutura de ALGOL ou Modula-2. Não é de admirar que C seja tranqüilamente a linguagem mais popular entre excelentes programadores profissionais.

O fato de C freqüentemente ser usada em lugar da linguagem assembly é o fator mais importante para a sua popularidade entre os programadores. A linguagem assembly usa uma representação simbólica do código binário real que o computador executa diretamente. Cada operação em linguagem assembly leva a uma tarefa simples a ser executada pelo computador. Embora a linguagem assembly dê aos programadores o potencial de realizar tarefas com máxima flexibilidade e eficiência, é notoriamente difícil de trabalhar quando se está desenvolvendo ou depurando um programa. Além disso, como assembly não é uma linguagem estruturada, o programa final tende a ser um código "espaguete" — um emaranhado de jumps, calls e índices. Essa falta de estrutura torna os programas em linguagem assembly difíceis de ler, aperfeiçoar e manter. Talvez mais importante: as rotinas em linguagem assembly não são portáveis entre máquinas com unidades centrais de processamento (CPUs) diferentes.

Inicialmente, C era usada na programação de sistema. Um *programa de sistema* forma uma porção do sistema operacional do computador ou de seus utilitários de suporte. Por exemplo, os programas que seguem são freqüentemente chamados de programas de sistema:

- Sistemas operacionais
- Interpretadores
- Editores
- Programas de planilhas eletrônicas
- Compiladores
- Gerenciadores de banco de dados

Em virtude da sua portabilidade e eficiência, à medida que C cresceu em popularidade, muitos programadores começaram a usá-la para programar todas as tarefas. Por haver compiladores C para quase todos os computadores, é possível tomar um código escrito para uma máquina, compilá-lo e rodá-lo em outra com pouca ou nenhuma modificação. Esta portabilidade economiza tempo e dinheiro. Os compiladores C também tendem a produzir um código-objeto muito compacto e rápido — menor e mais rápido que aquele da maioria dos compiladores BASIC, por exemplo.

Além disso, os programadores usam C em todos os tipos de trabalho de programação porque eles gostam de C! Ela oferece a velocidade da linguagem assembly e a extensibilidade de FORTH, mas poucas das restrições de Pascal ou Modula-2. Cada programador C pode, de acordo com sua própria personalidade, criar e manter uma biblioteca única de funções customizadas, para ser usada em muitos programas diferentes. Por admitir — na verdade encorajar — a compilação separada, C permite que os programadores gerenciem facilmente grandes projetos com mínima duplicação de esforço.

Compiladores *Versus* Interpretadores

Os termos *compiladores* e *interpretadores* referem-se à maneira como um programa é executado. Existem dois métodos gerais pelos quais um programa pode ser executado. Em teoria, qualquer linguagem de programação pode ser compilada ou interpretada, mas algumas linguagens geralmente são executadas de uma maneira ou de outra. Por exemplo, BASIC é normalmente interpretada e C, compilada (especialmente no auxílio à depuração ou em plataformas experimentais como a desenvolvida na Parte 5). A maneira pela qual um programa é executado não é definida pela linguagem em que ele é escrito. Interpretadores e compiladores são simplesmente programas sofisticados que operam sobre o código-fonte do seu programa. Como a diferença entre um compilador e um interpretador pode não ser clara para todos os leitores, a breve descrição seguinte esclarecerá o assunto.

Um interpretador lê o código-fonte do seu programa uma linha por vez, executando a instrução específica contida nessa linha. Um compilador lê o programa inteiro e converte-o em um *código-objeto*, que é uma tradução do código-fonte do programa em uma forma que o computador possa executar diretamente. O código-objeto é também conhecido como código binário ou código de máquina. Uma vez que o programa tenha sido compilado, uma linha do código-fonte, mesmo alterada, não é mais importante na execução do seu programa.

Quando um interpretador é usado, deve estar presente toda vez que você executar o seu programa. Por exemplo, em BASIC você precisa primeiro executar o interpretador, carregar seu programa e digitar **RUN** cada vez que quiser usá-lo. O interpretador BASIC examina seu programa uma linha por vez para correção e então executa-o. Esse processo lento ocorre cada vez que o programa for executado. Um compilador, ao contrário, converte seu programa em um código-objeto que pode ser executado diretamente por seu computador. Como o compilador traduz seu programa de uma só vez, tudo o que você precisa fazer é executar seu programa diretamente, geralmente apenas digitando seu nome. Assim, o tempo de compilação só é gasto uma vez, enquanto o código interpretado incorre neste trabalho adicional cada vez que o programa executa.

A Forma de um Programa em C

A Tabela 1.2 lista as 32 palavras-chave (ou palavras reservadas) que, combinadas com a sintaxe formal de C, formam a linguagem de programação C. Destas, 27 foram definidas pela versão original de C. As cinco restantes foram adicionadas pelo comitê ANSI: **enum, const, signed, void** e **volatile**.

Tabela 1.2 Uma lista das palavras-chave de C ANSI

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Além disso, muitos compiladores C acrescentaram diversas palavras-chave para explorar melhor a organização da memória da família de processadores 8088/8086, que suporta programação interlinguagens e interrupções. Aqui é mostrada uma lista das palavras-chave estendidas mais comuns:

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

Seu compilador pode também suportar outras extensões que ajudem a aproveitar melhor seu ambiente específico.

Todas as palavras-chave de C são minúsculas. Em C, maiúsculas e minúsculas são diferentes: **else** é uma palavra-chave, mas ELSE não. Uma palavra-chave não pode ser usada para nenhum outro propósito em um programa em C — ou seja, ela não pode servir como uma variável ou nome de uma função.

Todo programa em C consiste em uma ou mais funções. A única função que necessariamente precisa estar presente é a denominada **main()**, que é a primeira função a ser chamada quando a execução do programa começa. Em um código de C bem escrito, **main()** contém, em essência, um esboço do que o programa faz. O esboço é composto de chamadas de funções. Embora **main()** não seja tecnicamente parte da linguagem C, trate-a como se fosse. Não tente usar **main()** como nome de uma variável porque provavelmente confundirá o compilador.

A forma geral de um programa em C é ilustrada na Figura 1.1, onde **f1()** até **fN()** representam funções definidas pelo usuário.


```
declarações globais

tipo devolvido main(lista de parâmetros)
{
  seqüência de comandos
}

tipo devolvido f1(lista de parâmetros)
{
  seqüência de comandos
}

tipo devolvido f2(lista de parâmetros)
{
  seqüência de comandos
}
.
.
.
tipo devolvido fN(lista de parâmetros)
{
```

Figura 1.1 A forma geral de um programa em C

A Biblioteca e a Linkedição

Tecnicamente falando, é possível criar um programa útil e funcional que consista apenas nos comandos realmente criados pelo programador. Porém, isso é muito raro porque C, dentro da atual definição da linguagem, não oferece nenhum método de executar operações de entrada/saída (E/S). Como resultado, a maioria dos programas inclui chamadas a várias funções contidas na *biblioteca* C *padrão*.

Todo compilador C vem com uma biblioteca C padrão de funções que realizam as tarefas necessárias mais comuns. O padrão C ANSI especifica o conjunto mínimo de funções que estará contido na biblioteca. No entanto, seu compilador provavelmente conterá muitas outras funções. Por exemplo, o padrão C ANSI não define nenhuma função gráfica, mas seu compilador provavelmente inclui alguma.

Em algumas implementações de C, a biblioteca aparece em um grande arquivo; em outras, ela está contida em muitos arquivos menores, uma organização que aumenta a eficiência e a praticidade. Porém, para simplificar, este livro usa a forma singular em referênca à biblioteca.

Os implementadores do seu compilador C já escreveram a maioria das funções de propósito geral que você usará. Quando chama uma função que não faz parte do programa que você escreveu, o compilador C "memoriza" seu nome. Mais tarde, o *linkeditor* (linker) combina o código que você escreveu com o código-objeto já encontrado na biblioteca padrão. Esse processo é chamado de *linkedição*. Alguns compiladores C têm seu próprio linkeditor, enquanto outros usam o linkeditor padrão fornecido pelo seu sistema operacional.

As funções guardadas na biblioteca estão em formato *relocável*. Isso significa que os endereços de memória das várias instruções em código de máquina não estão absolutamente definidos — apenas informações relativas são guardadas. Quando seu programa é linkeditado com as funções da biblioteca padrão, esses endereços relativos são utilizados para criar os endereços realmente usados. Há diversos manuais e livros técnicos que explicam esse processo com mais detalhes. Contudo, você não precisa de nenhuma informação adicional sobre o processo real de relocação para programar em C.

Muitas das funções de que você precisará ao escrever seus programas estão na biblioteca padrão. Elas agem como blocos básicos que você combina. Se escreve uma função que usará muitas vezes, você também pode colocá-la em uma biblioteca. Alguns compiladores permitem que você coloque sua função na biblioteca padrão; outros exigem a criação de uma biblioteca adicional. De qualquer forma, o código estará lá para ser usado repetidamente.

Lembre-se de que o padrão ANSI apenas especifica uma biblioteca padrão *mínima*. A maioria dos compiladores fornece bibliotecas que contêm muito mais funções que aquelas definidas pelo ANSI. Além disso, algumas funções encontradas na versão original de C para UNIX não são definidas pelo padrão ANSI por serem redundantes. Este livro aborda todas as funções definidas pelo ANSI como também as mais importantes e largamente usadas pelo padrão C UNIX antigo. Ele também examina diversas funções muito usadas, mas que não são definidas pelo ANSI nem pelo antigo padrão UNIX. (Funções não-ANSI serão indicadas para evitar confusão.)

Compilação Separada

Muitos programas curtos de C estão completamente contidos em um arquivo-fonte. Contudo, quando o tamanho de um programa cresce, também aumenta seu tempo de compilação (e tempos de compilação longos contribuem para paciências curtas!). Logo, C permite que um programa seja contido em muitos arquivos e que cada arquivo seja compilado separadamente. Uma vez que todos os arquivos estejam compilados, eles são linkeditados com qualquer rotina de

biblioteca, para formar um código-objeto completo. A vantagem da compilação separada é que, se houver uma mudança no código de um arquivo, não será necessária a recompilação do programa todo. Em tudo, menos nos projetos mais simples, isso economiza um tempo considerável. (Estratégias de compilação separada são abordadas em detalhes na Parte 4.)

Compilando um Programa em C

Compilar um programa em C consiste nestes três passos:

1. Criar o programa
2. Compilar o programa
3. Linkeditar o programa com as funções necessárias da biblioteca

Alguns compiladores fornecem ambientes de programação integrados que incluem um editor. Com outros, é necessário usar um editor separado para criar seu programa. Os compiladores só aceitam a entrada de arquivos de texto padrão. Por exemplo, seu compilador não aceitará arquivos criados por certos processadores de textos porque eles têm códigos de controle e caracteres não-imprimíveis.

O método exato que você utiliza para compilar um programa depende do compilador que está em uso. Além disso, a linkedição varia muito entre os compiladores e os ambientes. Consulte seu manual do usuário para detalhes.

O Mapa de Memória de C

Um programa C compilado cria e usa quatro regiões, logicamente distintas na memória, que possuem funções específicas. A primeira região é a memória que contém o código do seu programa. A segunda é aquela onde as variáveis globais são armazenadas. As duas regiões restantes são a pilha e o "heap". A *pilha* tem diversos usos durante a execução de seu programa. Ela possui o endereço de retorno das chamadas de função, argumentos para funções e variáveis locais. Ela também guarda o estado atual da CPU. O *heap* é uma região de memória livre que seu programa pode usar, via funções de alocação dinâmica de C, em aplicações como listas encadeadas e árvores.

A disposição exata de seu programa pode variar de compilador para compilador e de ambiente para ambiente. Por exemplo, a maioria dos compiladores para a família de processadores 8086 tem seis maneiras diferentes de or-

ganizar a memória em razão da arquitetura segmentada de memória do 8086. Os modelos de memória da família de processadores 8086 são discutidos mais adiante neste livro.

Embora a disposição física exata de cada uma das quatro regiões possa diferir entre tipos de CPU e implementações de C, o diagrama da Figura 1.2 mostra conceitualmente como seu programa aparece na memória.

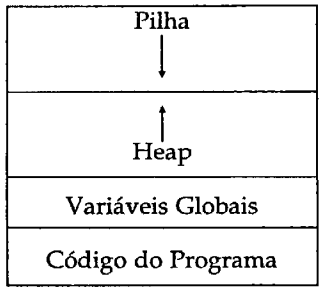


Figura 1.2 Um mapa conceitual de memória de um programa em C.

C *Versus* C++

Antes de concluir este capítulo, é necessário dizer algumas palavras sobre C++. Algumas vezes os novatos confundem o que é C++ e como difere de C. Para ser breve, C++ é uma versão estendida e melhorada de C que é projetada para suportar programação orientada a objetos (OOP, do inglês Object Oriented Programming). C++ contém e suporta toda a linguagem C e mais um conjunto de extensões orientadas a objetos. (Ou seja, C++ é um superconjunto de C.) Como C++ é construída sobre os fundamentos de C, você não pode programar em C++ se não entender C. Portanto, virtualmente todo o material apresentado neste livro aplica-se também a C++.

NOTA: Para uma descrição completa da linguagem C++ veja o livro, C++ — The Complete Reference, de Herbert Schildt.

Hoje em dia, e por muitos anos ainda, a maioria dos programadores ainda escreverá, manterá e utilizará programas C, e não C++. Como mencionado, C suporta programação estruturada. A programação estruturada tem-se mostrado eficaz ao longo dos 25 anos em que tem sido usada largamente. C++ é projetada principalmente para suportar OOP, que incorpora os princípios da programação estruturada, mas inclui objetos. Embora a OOP seja muito eficaz para uma certa classe de tarefas de programação, muitos programas não se beneficiam da sua aplicação. Por isso, "código direto em C" estará em uso por muito tempo ainda.

Um Compilador C++ Funcionará com Programas C?

Hoje em dia é difícil ver um compilador anunciado ou descrito simplesmente como um "compilador C". Em vez disso, é comum ver um compilador anunciado como "compilador C/C++", ou às vezes simplesmente como compilador C++. Esta situação faz surgir naturalmente a pergunta: "Um compilador C++ funcionará com programas C?". A resposta é: "Sim!". Qualquer um e todos os compiladores que podem compilar programas C++ também podem compilar programas C. Portanto, se seu compilador é denominado um "compilador C++", não se preocupe, também é um compilador C padrão ANSI completo.

Uma Revisão de Termos

Os termos a seguir serão usados freqüentemente durante toda essa referência. Você deve estar completamente familiarizado com eles.

- *Código-Fonte* O texto de um programa que um usuário pode ler, normalmente interpretado como o programa. O código-fonte é a entrada para o compilador C.
- *Código-Objeto* Tradução do código-fonte de um programa em código de máquina que o computador pode ler e executar diretamente. O código-objeto é a entrada para o linkeditor.
- *Linkeditor* Um programa que une funções compiladas separadamente em um programa. Ele combina as funções da biblioteca C padrões com o código que você escreveu. A saída do linkeditor é um programa executável.
- *Biblioteca* O arquivo contendo as funções padrão que seu programa pode usar. Essas funções incluem todas as operações de E/S como também outras rotinas úteis.
- *Tempo de compilação* Os eventos que ocorrem enquanto o seu programa está sendo compilado. Uma ocorrência comum em tempo de compilação é um erro de sintaxe.
- *Tempo de execução* Os eventos que ocorrem enquanto o seu programa é executado.

MAKRON Books

2

Expressões em C

Este capítulo examina o elemento mais fundamental da linguagem C: a expressão. Como você verá, as expressões em C são substancialmente mais gerais e poderosas que na maioria das outras linguagens de programação. As expressões são formadas pelos elementos mais básicos de C: dados e operadores. Os dados podem ser representados por variáveis ou constantes. C, como a maioria das outras linguagens, suporta uma certa quantidade de tipos diferentes de dados. Também provê uma ampla variedade de operadores.

Os Cinco Tipos Básicos de Dados

Há cinco tipos básicos de dados em C: caractere, inteiro, ponto flutuante, ponto flutuante de precisão dupla e sem valor (**char**, **int**, **float**, **double** e **void**, respectivamente). Como você verá, todos os outros tipos de dados em C são baseados em um desses tipos. O tamanho e a faixa desses tipos de dados variam de acordo com o tipo de processador e com a implementação do compilador C. Um caractere ocupa geralmente 1 byte e um inteiro tem normalmente 2 bytes, mas você não pode fazer esta suposição se quiser que seus programas sejam portáveis a uma gama mais ampla de computadores. O padrão ANSI estipula apenas a *faixa* mínima de cada tipo de dado, não o seu tamanho em bytes.

O formato exato de valores em ponto flutuante depende de como eles são implementados. Inteiros geralmente correspondem ao tamanho natural de uma palavra do computador host. Valores do tipo **char** são normalmente usados para conter valores definidos pelo conjunto de caracteres ASCII. Valores fora dessa faixa podem ser manipulados diferentemente entre as implementações de C.

A faixa dos tipos **float** e **double** é dada em dígitos de precisão. As grandezas dos tipos **float** e **double** dependem do método usado para representar os números em ponto flutuante. Qualquer que seja o método, o número é muito grande. O padrão ANSI especifica que a faixa mínima de um valor em ponto flutuante é de 1E-37 a 1E+37. O número mínimo de dígitos de precisão é exibido na Tabela 2.1 para cada tipo de ponto flutuante.

O tipo **void** declara explicitamente uma função que não retorna valor algum ou cria ponteiros genéricos. Ambas as utilizações são discutidas nos capítulos subseqüentes.

Tabela 2.1 Todos os tipos de dados definidos no padrão ANSI.

Tipo	Tamanho aproximado em bits	Faixa mínima
char	8	-127 a 127
unsigned char	8	0 a 255
signed char	8	-127 a 127
int	16	-32.767 a 32.767
unsigned int	16	0 a 65.535
signed int	16	O mesmo que **int**
short int	16	O mesmo que **int**
unsigned short int	16	0 a 65.535
signed short int	16	O mesmo que **short int**
long int	32	-2.147.483.647 a 2.147.483.647
signed long int	32	O mesmo que **long int.**
unsigned long int	32	0 a 4.294.967.295
float	32	Seis dígitos de precisão
double	64	Dez dígitos de precisão
long double	80	Dez dígitos de precisão

Modificando os Tipos Básicos

Exceto o **void**, os tipos de dados básicos podem ter vários modificadores precedendo-os. Um modificador é usado para alterar o significado de um tipo básico para adaptá-lo mais precisamente às necessidades de diversas situações. A lista de modificadores é mostrada aqui:

signed
unsigned
long
short

Os modificadores **signed, short, long** e **unsigned** podem ser aplicados aos tipos básicos caractere e inteiro. Contudo, **long** também pode ser aplicado a **double.** (Note que o padrão ANSI elimina o **long float** porque ele tem o mesmo significado de um **double**.)

A Tabela 2.1 mostra todas as combinações de tipos de dados que atendem ao padrão ANSI juntamente com suas faixas mínimas e larguras aproximadas em bits.

O uso de **signed** com inteiros é permitido, mas redundante porque a declaração padrão de inteiros assume um número com sinal. O uso mais importante de **signed** é modificar **char** em implementações em que esse tipo, por padrão, não tem sinal.

Algumas implementações podem permitir que **unsigned** seja aplicado aos tipos de ponto flutuante (como em **unsigned double**). Porém, isso reduz a portabilidade de seu código e geralmente não é recomendável. Qualquer tipo expandido ou adicional não definido pelo padrão proposto ANSI provavelmente não será suportado por todas as implementações de C.

A diferença entre inteiros com ou sem sinal é a maneira como o bit mais significativo do inteiro é interpretado. Se um inteiro com sinal é especificado, o compilador C gerará um código que assume que o bit de mais alta ordem de um inteiro deve ser interpretado como *indicador de sinal*. Se o indicador de sinal é 0, o número é positivo; se é 1, o número é negativo.

Em geral, os números negativos são representados usando-se o *complemento de dois*, que inverte todos os bits em um número (exceto o indicador de sinal), adiciona 1 a esse número e põe o indicador de sinal em 1.

Inteiros com sinal são importantes em muitos algoritmos, mas eles têm apenas metade da grandeza absoluta de seus irmãos sem sinal. Por exemplo, aqui está 32.767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Se o bit mais significativo fosse colocado em 1, o número seria interpretado como -1. Porém, se você o declarar como sendo um **unsigned int**, o número se tornará 65.535 quando o bit mais significativo for 1.

Nomes de Identificadores

Em C, os nomes de variáveis, funções, rótulos e vários outros objetos definidos pelo usuário são chamados de *identificadores*. Esses identificadores podem variar de um a diversos caracteres. O primeiro caractere deve ser uma letra ou um sublinhado e os caracteres subseqüentes devem ser letras, números ou sublinhados. Aqui estão alguns exemplos de nomes de identificadores corretos e incorretos:

Correto	**Incorreto**
count	1count
test23	hi!there
high_balance	high...balance

O padrão C ANSI determina que identificadores podem ter qualquer tamanho, mas pelo menos os primeiros 6 caracteres devem ser significativos se o identificador estiver envolvido em um processo externo de linkedição. Esses identificadores, chamados *nomes externos*, incluem nomes de funções e variáveis globais que são compartilhadas entre arquivos. Se o identificador não é usado em um processo externo de linkedição, os primeiros 31 caracteres serão significativos. Esse tipo de identificador é denominado *nome interno*. Consulte o manual do usuário para ver exatamente quantos caracteres significativos são permitidos pelo compilador que você está usando.

Um nome de identificador pode ser maior que o número de caracteres significativos reconhecidos pelo compilador. Porém, os caracteres que ultrapassarem o limite serão ignorados. Por exemplo, se seu compilador reconhece 31 caracteres significativos, os seguintes identificadores serão considerados por ele como sendo iguais:

Nomes_de_identificadores_excessivamente_longos_sao_incomodos
Nomes_de_identificadores_excessivamente_longos_sao_incomodos_para_usar

Em C, letras maiúsculas e minúsculas são tratadas diferentemente. Logo, **count, Count** e **COUNT** são três identificadores distintos. Em alguns ambientes, o tipo da letra (maiúscula ou minúscula) dos nomes de funções e de variáveis globais pode ser ignorado se o linkeditor for indiferente ao tipo (mas a maioria dos ambientes atuais suporta linkedição sensível à caixa alta ou baixa).

Um identificador não pode ser igual a uma palavra-chave de C e não deve ter o mesmo nome que as funções que você escreveu ou as que estão na biblioteca C.

Variáveis

Como você provavelmente sabe, uma *variável* é uma posição nomeada de memória, que é usada para guardar um valor que pode ser modificado pelo programa. Todas as variáveis em C devem ser declaradas antes de serem usadas. A forma geral de uma declaração é

tipo lista_de_variáveis;

Aqui, *tipo* deve ser um tipo de dado válido em C mais quaisquer modificadores; e *lista_de_variáveis* pode consistir em um ou mais nomes de identificadores separados por vírgulas. Aqui estão algumas declarações:

```
int i, j, l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Lembre-se de que, em C, o nome de uma variável não tem nenhuma relação com seu tipo.

Onde as Variáveis São Declaradas

As variáveis serão declaradas em três lugares básicos: dentro de funções, na definição dos parâmetros das funções e fora de todas as funções. Estas são variáveis locais, parâmetros formais e variáveis globais, respectivamente.

Variáveis Locais

Variáveis que são declaradas dentro de uma função são chamadas de *variáveis locais*. Em algumas literaturas de C, variáveis locais são referidas como variáveis *automáticas*, porque em C você pode usar a palavra-chave **auto** para declará-las. Este livro usa o termo variável local, que é mais comum. Variáveis locais só podem ser referenciadas por comandos que estão dentro do bloco no qual as variáveis foram declaradas. Em outras palavras, variáveis locais não são reconhecidas fora de seu próprio bloco de código. Lembre-se, um bloco de código inicia-se em abre-chaves ({) e termina em fecha-chaves (}).

Variáveis locais existem apenas enquanto o bloco de código em que foram declaradas está sendo executado. Ou seja, uma variável local é criada na entrada de seu bloco e destruída na saída.

O bloco de código mais comum em que as variáveis locais foram declaradas é a função. Por exemplo, considere as seguintes funções:


```
void func1(void)
{
   int x;

   x = 10;
}

void func2(void)
{
   int x;

   x = -199;
}
```

A variável inteira **x** é declarada duas vezes, uma vez em **func1()** e outra em **func2()**. O **x** em **func1()** não tem nenhuma relação ou correspondência com o x em **func2()**. A razão para isso é que cada **x** é reconhecido apenas pelo código que está dentro do mesmo bloco da declaração de variável.

A linguagem C contém a palavra-chave **auto**, que pode ser usada para declarar variáveis locais. Porém, já que todas as variáveis não globais são, por padrão, assumidas como sendo **auto**, esta palavra-chave quase nunca é usada. Logo, os exemplos deste livro não a usam. (Dizem que a palavra-chave **auto** foi incluída em C para fornecer compatibilidade em nível de fonte com sua predecessora B.)

A maioria dos programadores declara todas as variáveis usadas por uma função imediatamente após o abre-chaves da função e antes de qualquer outro comando. Porém, as variáveis locais podem ser declaradas dentro de qualquer bloco de código. O bloco definido por uma função é simplesmente um caso especial. Por exemplo,

```
void f(void)
{
   int t;

   scanf("%d", &t);

   if(t == 1) {
     char s[80];   /* isto é criado apenas
                      na entrada deste bloco */
     printf("entre com o nome:");
     gets(s);
     /* faz alguma coisa ...*/
   }
}
```

Aqui, a variável local **s** é criada na entrada do bloco de código **if** e destruída na saída. Além disso, **s** é reconhecida apenas dentro do bloco **if** e não pode ser referenciada em qualquer outro lugar — mesmo nas outras partes da função que a contém.

A principal vantagem em declarar uma variável local dentro de um bloco condicional é que a memória para ela só será alocada se necessário. Isso acontece porque variáveis locais não existirão até que o bloco em que elas são declaradas seja iniciado. Você deve preocupar-se com isso quando estiver produzindo código para controladores dedicados (como um controlador de porta de garagem, que responde a um código de segurança digital) em que a memória RAM é escassa, por exemplo.

Declarar variáveis dentro do bloco de código que as utiliza também ajuda a evitar efeitos colaterais indesejados. Como a variável não existe fora do bloco em que é declarada, ela não pode ser acidentalmente alterada. Porém, quando cada função realiza uma tarefa lógica bem definida, você não precisa "proteger" variáveis dentro de uma função do código que constitui a função. Isso explica por que as variáveis usadas por uma função são geralmente declaradas no início da função.

Lembre-se de que você deve declarar todas as variáveis locais no início do bloco em que elas são definidas, antes de qualquer comando do programa. Por exemplo, a função seguinte está tecnicamente incorreta e não será compilada na maioria dos compiladores.

```
/* Esta função está errada. */
void f(void)
{
  int i;

  i = 10;

  int j;  /* esta linha irá provocar um erro */

  j = 20;
}
```

Porém, se você tivesse declarado **j** dentro de seu próprio bloco de código ou antes do comando **i = 10**, a função teria sido aceita. Por exemplo, as duas versões mostradas aqui estão sintaticamente corretas:

```
/* Define j dentro de seu próprio bloco de código. */
void f(void)
{
  int i;

  i = 10;
  { /* define j em seu próprio bloco de código */
    int j;

    j = 20;
  }
}
```

```
/* Define j no início do bloco da função. */
void f(void)
{
  int i;
  int j;

  i = 10;
  j = 20;
}
```

NOTA: Como ponto interessante, a restrição que exige que todas as variáveis sejam declaradas no início do bloco foi removida em C++. Em C++, as variáveis podem ser declaradas em qualquer ponto dentro de um bloco.

Como todas as variáveis locais são criadas e destruídas a cada entrada e saída do bloco em que elas são declaradas, seu conteúdo é perdido quando o bloco deixa de ser executado. É especialmente importante lembrar disso ao chamar uma função. Quando uma função é chamada, suas variáveis locais são criadas e, ao retornar, elas são destruídas. Isso significa que as variáveis locais não podem reter seus valores entre chamadas. (No entanto, você pode ordenar ao compilador que retenha seus valores usando o modificador **static**.)

A menos que especificado de outra forma, variáveis locais são armazenadas na pilha. O fato de a pilha ser uma região de memória dinâmica e mutável explica por que variáveis locais não podem, em geral, reter seus valores entre chamadas de funções.

Você pode inicializar uma variável local com algum valor conhecido. Esse valor será atribuído à variável cada vez que o bloco de código em que ela é declarada for executado. Por exemplo, o programa seguinte imprime o número 10 dez vezes:

```
#include <stdio.h>

void f(void);

void main(void)
```

```
{
  int i;

  for(i=0; i<10; i++) f();
}

void f(void)
{
  int j = 10;

  printf("%d ", j);

  j++;  /* esta linha não tem nenhum efeito */
}
```

Parâmetros Formais

Se uma função usa argumentos, ela deve declarar variáveis que receberão os valores dos argumentos. Essas variáveis são denominadas *parâmetros formais* da função. Elas se comportam como qualquer outra variável local dentro da função. Como é mostrado no fragmento de programa seguinte, suas declarações ocorrem depois do nome da função e dentro dos parênteses:

```
/* Retorna 1 se c é parte da string s; 0 se não é o caso */
is_in(char *s, char c)
{
   while(*s)
     if(*s == c) return 1;
     else s++;

   return 0;
}
```

A função **is_in()** tem dois parâmetros: **s** e **c**. Essa função devolve 1 se o caractere especificado em **c** estiver contido na string **s**; 0 se não estiver.

Você deve informar à C que tipo de variáveis são os parâmetros formais, declarando-os como mostrado acima. Uma vez feito isso, elas podem ser usadas dentro da função como variáveis locais normais. Tenha sempre em mente que, como variáveis locais, elas também são dinâmicas e são destruídas na saída da função.

Você deve ter certeza de que os parâmetros formais que estão declarados são do mesmo tipo dos argumentos que você utiliza para chamar a função.

Se há uma discordância de tipos, resultados inesperados podem ocorrer. Ao contrário de muitas outras linguagens, C geralmente fará alguma coisa, inclusive em circunstâncias não usuais, mesmo que não seja o que você quer. Há poucos erros em tempo de execução e nenhuma verificação de limites. Como programador, você deve ter certeza de que erros de incongruência de tipo não ocorrerão.

Embora a linguagem C forneça os *protótipos de funções*, que podem ser usados para ajudar a verificar se os argumentos usados para chamar a função são compatíveis com os parâmetros, ainda podem ocorrer problemas. (Isto é, os protótipos de função não eliminam inteiramente incongruências de tipo de parâmetro). Além disso, você deve incluir explicitamente protótipos de funções em seu programa para receber esse benefício extra. (O uso de protótipos de funções é discutido em profundidade no Capítulo 6.)

Analogamente às variáveis locais, você pode fazer atribuições a parâmetros formais de uma função ou usá-los em qualquer expressão permitida em C. Embora essas variáveis recebam o valor dos argumentos passados para a função, elas podem ser usadas como qualquer outra variável local.

Variáveis Globais

Ao contrário das variáveis locais, as *variáveis globais* são reconhecidas pelo programa inteiro e podem ser usadas por qualquer pedaço de código. Além disso, elas guardam seus valores durante toda a execução do programa. Você cria variáveis globais declarando-as fora de qualquer função. Elas podem ser acessadas por qualquer expressão independentemente de qual bloco de código contém a expressão.

No programa seguinte, a variável **count** foi declarada fora de todas as funções. Embora sua declaração ocorra antes da função **main()**, ela poderia ter sido colocada em qualquer lugar anterior ao seu primeiro uso, desde que não estivesse em uma função. No entanto, é melhor declarar variáveis globais no início do programa.

```
#include <stdio.h>
int count;  /* count é global */

void func1(void);
void func2(void);

void main(void)
{
   count = 100;
   func1();
```

```
}

void func1(void)
{
   int temp;

   temp = count;
   func2();
   printf("count é %d", count); /* imprimirá 100 */
}

void func2(void)
{
   int count;
   for (count=1; count<10; count++)
     putchar('.');
}
```

Olhe atentamente para esse programa. Observe que, apesar de nem **main()** nem **func1()** terem declarado a variável **count**, ambas podem usá-la. A função **func2()**, porém, declarou uma variável local chamada **count**. Quando **func2()** referencia **count**, ela referencia apenas sua variável local, não a variável global. Se uma variável global e uma variável local possuem o mesmo nome, todas as referências ao nome da variável dentro do bloco onde a variável local foi declarada dizem respeito à variável local e não têm efeito algum sobre a variável global. Pode ser conveniente, mas esquecer-se disso poderá fazer com que seu programa seja executado estranhamente, embora pareça correto.

O armazenamento de variáveis globais encontra-se em uma região fixa da memória, separada para esse propósito pelo compilador C. Variáveis globais são úteis quando o mesmo dado é usado em muitas funções em seu programa. No entanto, você deve evitar usar variáveis globais desnecessárias. Elas ocupam memória durante todo o tempo em que seu programa está executando, não apenas quando são necessárias. Além disso, usar uma variável global onde uma variável local poderia ser usada torna uma função menos geral, porque ela conta com alguma coisa que deve ser definida fora dela. Finalmente, usar um grande número de variáveis globais pode levar a erros no programa por causa de desconhecidos — e indesejáveis — efeitos colaterais. Isso pode ser evidenciado no BASIC padrão, em que todas as variáveis são globais. Um problema maior no desenvolvimento de grandes projetos é a mudança acidental do valor de uma variável porque ela é usada em algum outro lugar do programa. Isso pode acontecer em C se você usar variáveis globais demais em seus programas.

Uma das principais razões para uma linguagem estruturada é a compartimentalização ou separação de código e dados. Em C, esse isolamento é conseguido pelo uso de variáveis locais e funções. Por exemplo, a Figura 2.1 mostra duas maneiras de escrever **mul()** — uma função simples que calcula o produto de dois inteiros.

Ambas as funções retornam o produto das variáveis **x** e **y**. Contudo, a versão generalizada, ou *parametrizada*, pode ser usada para retornar o produto de *quaisquer* dois inteiros, enquanto a versão específica só pode ser usada para encontrar o produto das variáveis globais **x** e **y**.

Geral

```
mul(int x, int y)
{
 return(x*y);
}
```

Específica

```
int x, y;
mul( void )
{
 return (x*y);
}
```

Figura 2.1 Duas formas de escrever **mul()**.

Modificadores de Tipo de Acesso

O C introduziu dois novos modificadores (também chamados *quantificadores*) que controlam a maneira como as variáveis podem ser acessadas ou modificadas. Esses modificadores são **const** e **volatile**. Devem preceder os modificadores de tipo e os nomes que eles modificam.

const

Variáveis do tipo **const** não podem ser modificadas por seu programa. (Uma variável **const** pode, entretanto, receber um valor inicial.) O compilador pode colocar variáveis desse tipo em memória de apenas leitura (ROM). Por exemplo:

```
const int a=10;
```

cria uma variável inteira chamada **a**, com um valor inicial 10, que seu programa não pode modificar. Você pode, porém, usar a variável **a** em outros tipos de expressões. Uma variável **const** recebe seu valor de uma inicialização explícita ou por algum recurso dependente do hardware.

O qualificador **const** pode ser usado para proteger os objetos apontados pelos argumentos de uma função de serem modificados por esta função. Isto é, quando um ponteiro é passado para uma função, esta função pode modificar a variável real apontada pelo ponteiro. Entretanto, se o ponteiro é especificado como **const** na declaração dos parâmetros, o código da função não será capaz de modificar o que ele aponta. Por exemplo, a função **sp_to_dash()**, no programa seguinte, imprime um traço para cada espaço do seu argumento string. Ou melhor, a string "isso é um teste" será impressa "isso-é-um-teste". O uso de **const** na declaração do parâmetro assegura que o código dentro da função não possa modificar o objeto apontado pelo parâmetro.

```
#include <stdio.h>

void sp_to_dash(const char *str);

void main(void)
{
  sp_to_dash("isso é um teste");
}

void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str == ' ') printf("%c", '-');
    else printf("%c", *str);
    str++;
  }
}
```

Se você escrevesse **sp_to_dash()** de forma que a string fosse modificada, ela não seria compilada. Por exemplo, se você tivesse codificado **sp_to_dash()** como segue, obteria um erro:

```
/* isso está errado */
void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str==' ' ) *str = '-'; /* não faça isto */
    printf("%c", *str);
    str++;
  }
}
```


Muitas funções da biblioteca C padrão usam **const** em suas declarações de parâmetros. Por exemplo, a função **strlen()** tem este protótipo:

```
size_t strlen(const char *str);
```

Especificar *str* como **const** assegura que **strlen()** não modificará a string apontada por *str*. Em geral, quando uma função da biblioteca padrão não tem necessidade de modificar um objeto apontado por um argumento, ele é declarado como **const**.

Você também pode usar **const** para verificar se seu programa não modifica uma variável. Lembre-se de que uma variável do tipo **const** pode ser modificada por algo externo ao seu programa. Por exemplo, um dispositivo de hardware pode ajustar seu valor. Porém, declarando uma variável como **const**, você pode provar que qualquer alteração nesta variável ocorre devido a eventos externos.

volatile

O modificador **volatile** é usado para informar ao compilador que o valor de uma variável pode ser alterado de maneira não explicitamente especificada pelo programa. Por exemplo, um endereço de uma variável global pode ser passado para a rotina de relógio do sistema operacional e usado para guardar o tempo real do sistema. Nessa situação, o conteúdo da variável é alterado sem nenhum comando de atribuição explícito no programa. Isso é importante porque muitos compiladores C automaticamente otimizam certas expressões, assumindo que o conteúdo de uma variável é imutável, se sua referência não aparecer no lado esquerdo da expressão; logo, ela pode não ser reexaminada toda vez que for referenciada. Além disso, alguns compiladores mudam a ordem de avaliação de uma expressão durante o processo de compilação. O modificador **volatile** previne a ocorrência dessas mudanças.

É possível usar **const** e **volatile** juntos. Por exemplo, se 0x30 é assumido como sendo o valor de uma porta que é mudado apenas por condições externas, a declaração seguinte é precisamente o que você quer para prevenir qualquer possibilidade de efeitos colaterais acidentais.

```
const volatile unsigned char *port = 0x30;
```

Especificadores de Tipo de Classe de Armazenamento

Há quatro especificadores de classe de armazenamento suportados por C.

extern
static
register
auto

Esses especificadores são usados para informar ao compilador como a variável deve ser armazenada. O especificador de armazenamento precede o resto da declaração da variável. Sua forma geral é:

especificador_de_armazenamento tipo nome_da_variável;

extern

Uma vez que C permite que módulos de um programa grande sejam compilados separadamente para então serem linkeditados juntos, uma forma de aumentar a velocidade de compilação e ajudar no gerenciamento de grandes projetos, deve haver alguma maneira de dizer a todos os arquivos sobre as variáveis globais solicitadas pelo programa. Lembre-se de que você pode declarar uma variável global apenas uma vez. Se você tentar declarar duas variáveis com o mesmo nome dentro do mesmo arquivo, seu compilador C poderá imprimir uma mensagem de erro como "nome de variável duplicado" ou poderá simplesmente escolher uma variável. O mesmo problema ocorre se você simplesmente declara todas as variáveis globais necessárias ao seu programa em cada arquivo. Embora o compilador não emita nenhuma mensagem de erro em tempo de compilação, você estaria realmente tentando criar duas (ou mais) cópias de cada variável. O transtorno começaria quando você tentasse linkeditar seus módulos. O linkeditor mostraria a mensagem de erro como "rótulo duplicado" porque ele não saberia que variável usar. A solução seria declarar todas as suas variáveis globais em um arquivo e usar declarações **extern** nos outros, como na Figura 2.2.

No arquivo 2, a lista de variáveis globais foi copiada do arquivo 1 e o especificador **extern** foi adicionado às declarações. O especificador **extern** diz ao compilador que os tipos e nomes de variável que o seguem foram declarados em outro lugar. Em outras palavras, **extern** deixa o compilador saber o que os tipos e nomes são para essas variáveis globais sem realmente criar armazenamento para elas novamente. Quando o linkeditor unir os dois módulos, todas as referências a variáveis externas serão resolvidas.

Quando utiliza uma variável global dentro de uma função que está no mesmo arquivo que a declaração da variável global, você pode usar **extern**, como mostrado aqui:

Arquivo 1

```
int x, y;
char ch;
main(void)
{
 .
 .
 .
}

func1()
{
  x = 123;
}
```

Arquivo 2

```
extern int x, y;
extern char ch;
func22(void)
{
  x = y/10;
}

func23()
{
  y = 10;
}
```

Figura 2.2 Uso de variáveis globais em módulos compilados separadamente.

```
int first, last;   /* declaração global de first e last */

void main(void)
{
  extern int first;  /* uso opcional da declaração extern */
  .
  .
  .
}
```

Embora as declarações de variáveis **extern** possam ocorrer dentro do mesmo arquivo da declaração global, elas não são necessárias. Se o compilador C encontra uma variável que não foi declarada, ele verifica se ela tem o mesmo nome de alguma variável global. Se tiver, o compilador assumirá que a variável global está sendo referenciada.

Variáveis static

Dentro de sua própria função ou arquivo, variáveis **static** são variáveis permanentes. Ao contrário das variáveis globais, elas não são reconhecidas fora de sua função ou arquivo, mas mantêm seus valores entre chamadas. Essa característica torna-as úteis quando você escreve funções generalizadas e funções de biblioteca que podem ser usadas por outros programadores. O especificador **static** tem efeitos diferentes em variáveis locais e em variáveis globais.

Variáveis Locais static

Quando o modificador **static** é aplicado a uma variável local, o compilador cria armazenamento permanente para ela quase da mesma forma como cria armazenamento para uma variável global. A diferença fundamental entre uma variável local **static** e uma variável global é que a variável local **static** é reconhecida apenas no bloco em que está declarada. Em termos simples, uma variável local **static** é uma variável local que retém seu valor entre chamadas de função.

Variáveis locais **static** são muito importantes na criação de funções isoladas, porque diversos tipos de rotinas devem preservar um valor entre as chamadas. Se variáveis **static** não fossem permitidas, variáveis globais teriam de ser usadas, abrindo brechas para possíveis efeitos colaterais. Um exemplo de função que requer uma variável local **static** é um gerador de série de números que produz um novo número baseado no anterior. Seria possível declarar uma variável global para reter esse valor. Porém, cada vez que a função é usada, você deve lembrar-se de declarar essa variável global e garantir que ela não conflite com nenhuma outra variável global já declarada. Além disso, usar uma variável global tornaria essa função difícil de ser colocada em uma biblioteca de funções. A melhor solução é declarar a variável que retém o número gerado como **static**, como neste fragmento de programa.

```
series(void)
{
   static int series_num;

   series_num = series_num+23;
   return (series_num);
}
```

Nesse exemplo, a variável **series_num** permanece existindo entre as chamadas da função em vez da criação e exclusão que as variáveis locais normais fariam. Isso significa que cada chamada a **series()** pode produzir um novo membro da série, baseado no número precedente, sem declarar essa variável globalmente.

Você pode dar à variável local **static** um valor de inicialização. Esse valor é atribuído apenas uma vez — e não toda vez que o bloco de código é inserido, de forma análoga às variáveis locais normais. Por exemplo, essa versão de **series()** inicializa **series_num** com 100:

```
series(void)
{
   static int series_num = 100;
   series_num = series_num+23;
   return series_num;
}
```


Da forma como a função se acha agora, a série sempre começa com o valor 123. Enquanto isso é aceitável para algumas aplicações, a maioria dos geradores de séries permite ao usuário especificar o ponto inicial. Uma maneira de dar a **series_num** um valor especificado pelo usuário é tornar **series_num** uma variável global e, em seguida, ajustar seu valor de acordo com o especificado. Porém, **series_num** foi feita **static** justamente para não ser definida como global. Isso leva ao segundo uso de **static**.

Variáveis Globais static

Aplicar o especificador **static** a uma variável global informa ao compilador para criar uma variável global que é reconhecida apenas no arquivo no qual a mesma foi declarada. Isso significa que, muito embora a variável seja global, rotinas em outros arquivos não podem reconhecê-la ou alterar seu conteúdo diretamente; assim, não está sujeita a efeitos colaterais. Entretanto, para as poucas situações onde uma variável local **static** não possa fazer o trabalho, você pode criar um pequeno arquivo que contenha apenas as funções que precisam da variável global **static** e compilar separadamente esse arquivo sem medo de efeitos colaterais.

Para ilustrar uma variável global **static**, o exemplo de gerador de série da seção anterior foi recodificado de forma que um valor somente inicialize a série por meio de uma chamada a uma segunda função denominada **series_ start()**. O arquivo inteiro, que contém **series()**, **series_start()** e **series_num** é mostrado aqui:

```
/* Isso deve estar em um único arquivo - preferencialmente
   isolado. */

static int series_num;
void series_start(int seed);
int series(void);
series(void)
{
   series_num = series_num+23;
   return series_num;
}

/* inicializa series_num */
void series_start(int seed)
{
   series_num = seed;
}
```

Para inicializar o gerador de série, deve-se chamar **series_start()** com algum valor inteiro conhecido. Depois disso, chamadas a **series()** geram os próximos elementos da série.

Revisando: Os nomes das variáveis locais **static** são reconhecidos apenas na função ou bloco de código em que elas são declaradas. Os nomes das variáveis globais **static** são reconhecidos apenas no arquivo em que elas residem. Isso significa que, se você colocar as funções **series()** e **series_start()** em uma biblioteca, poderá usar as funções, mas não poderá referenciar a variável **series_num**, que está escondida do resto do código do seu programa. De fato, você pode, inclusive, declarar e usar outra variável chamada **series_num** em seu programa (em outro arquivo, é claro). Em essência, o modificador **static** permite variáveis que são reconhecidas pelas funções que precisam delas, sem confundir outras funções.

As variáveis **static** admitem que você, o programador, esconda porções de seu programa das outras partes. Isso pode ser uma vantagem imensa quando se tenta gerenciar um programa muito grande e complexo. O especificador de classe de armazenamento **static** deixa você criar funções gerais que podem ir para bibliotecas que serão utilizadas posteriormente.

Variáveis register

O especificador de armazenamento **register** tradicionalmente era aplicado apenas a variáveis dos tipos **int** e **char**. Contudo, o padrão C ANSI ampliou sua definição de forma que ele pode ser aplicado a qualquer variável.

Originalmente, o especificador **register** solicitava ao compilador C que armazenasse o valor das variáveis declaradas com esse especificador num registrador da CPU em vez da memória, onde as variáveis normais são armazenadas. Isso significa que operações nas variáveis **register** poderiam ocorrer muito mais rapidamente que nas variáveis armazenadas na memória, pois o valor dessas variáveis era realmente conservado na CPU e não era necessário acesso à memória para determinar ou modificar seus valores.

Hoje, uma vez que agora o padrão C ANSI permite que você modifique qualquer tipo de variável com **register**, ele alterou a definição do que **register** faz. O padrão C ANSI simplesmente determina que "o acesso ao objeto é o mais rápido possível". Na prática, caracteres e inteiros são colocados nos registradores da CPU. Objetos maiores, como matrizes, obviamente não podem ser armazenados em um registrador, mas eles ainda podem receber um tratamento diferenciado. Dependendo da implementação do compilador C e de seu ambiente operacional, variáveis **register** podem ser manipuladas de quaisquer formas consideradas cabíveis pelo implementador do compilador. O padrão C ANSI também permite que o compilador ignore o especificador **register** e trate as variáveis modificadas por ele como se não fossem, mas isso raramente ocorre na prática.

Você só pode aplicar o especificador **register** a variáveis locais e a parâmetros formais em uma função. Assim, variáveis globais **register** não são permitidas. Aqui está um exemplo de como declarar uma variável **register** do tipo **int** e usá-la para controlar um laço. Essa função calcula o resultado de M^e para inteiros:

```
int_pwr(register int m, register int e)
{
   register int temp;

   temp = 1;

   for(; e; e--) temp = temp * m;
   return temp;
}
```

Neste exemplo, tanto **e** como **m** e **temp** são declaradas como variáveis **register** porque são usadas dentro do laço. O fato de variáveis **register** serem otimizadas para velocidade torna-as ideais ao controle de laço. Geralmente, variáveis **register** são usadas quando mais apropriadas, isto é, em lugares onde são feitas muitas referências a uma mesma variável. Isso é importante porque você pode declarar qualquer número de variáveis como sendo do tipo **register**, mas nem todas recebem a mesma otimização de velocidade.

O número de variáveis em registradores dentro de qualquer bloco de código é determinado pelo ambiente e pela implementação específica de C. Você não deve preocupar-se em declarar muitas variáveis **register** porque o compilador C automaticamente transforma variáveis **register** em variáveis comuns quando o limite for alcançado. (Isso é feito para assegurar a portabilidade do código em C por meio de uma ampla linha de processadores.)

Por todo este livro, muitas variáveis de controle de laço serão do tipo **register**. Normalmente, pelo menos duas variáveis **register** do tipo **char** ou **int** podem de fato ser colocadas em registradores da CPU. Como os ambientes variam enormemente, consulte o manual do usuário do seu compilador para determinar se você pode aplicar quaisquer outros tipos de opções de otimizações.

Como uma variável **register** pode ser armazenada em um registrador da CPU, variáveis **register** não podem ter endereços. Isto é, você não pode encontrar o endereço de uma variável **register** usando o operador **&** (discutido mais adiante neste capítulo).

Embora o padrão C ANSI tenha expandido a descrição de **register**, na prática ele geralmente só tem um efeito significativo com os tipos inteiro e caractere. Logo, você provavelmente não deve contar com aumentos substanciais da velocidade para os outros tipos de variáveis.

Inicialização de Variáveis

Você pode dar à maioria das variáveis em C um valor, no mesmo momento em que elas são declaradas, colocando um sinal de igual e uma constante após o nome da variável. A forma geral de uma inicialização é

tipo nome_da_variável = constante;

Alguns exemplos são

```
char ch = 'a';

int first = 0;

float balance = 123.23;
```

Variáveis globais e variáveis locais **static** são inicializadas apenas no começo do programa. Variáveis locais (incluindo variáveis locais **static**) são inicializadas cada vez que o bloco no qual estão declaradas for inserido. Variáveis locais e **register** que não são inicializadas possuem valores desconhecidos antes de ser efetuada a primeira atribuição a elas. Variáveis globais não inicializadas e variáveis locais estáticas são inicializadas com zero.

Constantes

Em C, *constantes* referem-se a valores fixos que o programa não pode alterar. Constantes em C podem ser de qualquer um dos cinco tipos de dados básicos. A maneira como cada constante é representada depende do seu tipo. Constantes de caractere são envolvidas por aspas simples ('). Por exemplo, 'a' e '%' são constantes tipo caractere. O padrão ANSI também define caracteres multi bytes (usados principalmente em ambientes de língua estrangeira).

Constantes inteiras são especificadas como números sem componentes fracionários. Por exemplo, 10 e -100 são constantes inteiras. Constantes em ponto flutuante requerem o ponto decimal seguido pela parte fracionária do número. Por exemplo, 11.123 é uma constante em ponto flutuante. C também permite que você use notação científica para números em ponto flutuante.

Existem dois tipos de ponto flutuante: **float** e **double**. Há também diversas variações dos tipos básicos que você pode gerar usando os modificadores de tipo. Por padrão, o compilador C encaixa uma constante numérica no menor tipo de dado compatível que pode contê-lo. Assim, 10 é um **int**, por padrão, mas 60.000 é **unsigned** e 100.000 é **long**. Muito embora o valor 10 possa caber em um tipo caractere, o compilador não atravessará os limites do tipo. A única exceção para a regra do menor tipo são constantes em ponto flutuante, assumidas como **doubles**.

Na maioria dos programas que você escreverá, os padrões do compilador são adequados. Porém, você pode especificar precisamente o tipo da constante numérica que deseja por meio da utilização de um sufixo. Para tipos em ponto flutuante, se você colocar um F após o número, ele será tratado como **float**. Se você colocar um L, ele se tornará um **long double**. Para tipos inteiros, o sufixo U representa **unsigned** e o L representa **long**. Aqui estão alguns exemplos:

Tipo de dado	Exemplos de constantes
int	1 123 21000 -234
long int	35000L -34L
short int	10 -12 90
unsigned int	10000U 987U 40000
float	123.23F 4.34e-3F
double	123.23 12312333 -0.9876324
long double	1001.2L

Constantes Hexadecimais e Octais

Às vezes é mais fácil usar um sistema numérico na base 8 ou 16 em lugar de 10 (nosso sistema decimal padrão). O sistema numérico na base 8 é chamado *octal* e utiliza os dígitos de 0 a 7. Em octal, o número 10 é o mesmo que 8 em decimal. O sistema numérico na base 16 é chamado *hexadecimal* e utiliza os dígitos de 0 a 9 mais as letras de A a F, que representam 10, 11, 12, 13, 14 e 15, respectivamente. Por exemplo, o número hexadecimal 10 é 16 em decimal. Em virtude de esses números serem usados freqüentemente, C permite especificar constantes inteiras em hexadecimal ou octal em lugar de decimal. Uma constante hexadecimal deve consistir em um 0x seguido por uma constante na forma hexadecimal. Uma constante octal começa com 0. Aqui estão alguns exemplos:

```
int hex = 0x80;    /* 128 em decimal */
int oct = 012;     /* 10 em decimal */
```

Constantes String

C suporta outro tipo de constante: a string. Uma *string* é um conjunto de caracteres colocado entre aspas duplas. Por exemplo, "isso é um teste" é uma string. Você viu exemplos de strings em alguns dos comandos **printf()** dos programas de exemplo. Embora C permita que você defina constantes string, ela não possui formalmente um tipo de dado string.

Você não deve confundir strings com caracteres. Uma constante de um único caractere é colocada entre aspas simples, como em "a". Contudo, "a" é uma string contendo apenas uma letra.

Constantes Caractere de Barra Invertida

Colocar entre aspas simples todas as constantes tipo caractere funciona para a maioria dos caracteres imprimíveis. Uns poucos, porém, como o retorno de carro (CR), são impossíveis de inserir pelo teclado. Por essa razão, C criou as constantes especiais de caractere de barra invertida.

C suporta diversos códigos de barra invertida (listados na Tabela 2.2) de forma que você pode facilmente entrar esses caracteres especiais como constantes. Você deve usar os códigos de barra invertida em lugar de seus ASCII equivalentes para aumentar a portabilidade.

Tabela 2.2 Códigos de barra invertida.

Código	Significado
\b	Retrocesso (BS)
\f	Alimentação de formulário (FF)
\n	Nova linha (LF)
\r	Retorno de carro (CR)
\t	Tabulação horizontal (HT)
\"	Aspas duplas
\'	Aspas simples
\0	Nulo
\\	Barra invertida
\v	Tabulação vertical
\a	Alerta (beep)
\N	Constante octal (onde N é uma constante octal)
\xN	Constante hexadecimal (onde N é uma constante hexadecimal)

Por exemplo, o programa seguinte envia à tela um caractere de nova linha e uma tabulação e, em seguida, escreve a string **isso é um teste**.


```
#include <stdio.h>
void main(void)
{
  printf("\n\tIsso é um teste");
}
```

Operadores

C é muito rica em operadores internos. (Na realidade, C dá mais ênfase aos operadores que a maioria das outras linguagens de computador.) C define quatro classes de operadores: aritméticos, relacionais, lógicos e bit a bit. Além disso, C tem alguns operadores especiais para tarefas particulares.

O Operador de Atribuição

Em C, você pode usar o operador de atribuição dentro de qualquer expressão válida de C. Isso não acontece na maioria das linguagens de computador (incluindo Pascal, BASIC e FORTRAN), que tratam os operadores de atribuição como um caso especial de comando. A forma geral do operador de atribuição é

nome_da_variável = expressão;

onde expressão pode ser tão simples como uma única constante ou tão complexa quanto você necessite. Como BASIC e FORTRAN, C usa um único sinal de igual para indicar atribuição (ao contrário de Pascal e Modula-2, que usam a construção :=). O *destino*, ou a parte esquerda, da atribuição deve ser uma variável ou um ponteiro, não uma função ou uma constante.

Freqüentemente, em literaturas de C e nas mensagens de erro dos compiladores, você verá esses dois termos: *lvalue* e *rvalue*. Exposto de forma simples, um *lvalue* é qualquer objeto que pode ocorrer no lado esquerdo de um comando de atribuição. Para todos os propósitos práticos, "lvalue" significa "variável". O termo *rvalue* refere-se às expressões do lado direito de uma atribuição e significa simplesmeste o valor da expressão.

Conversão de Tipos em Atribuições

Conversão de tipos refere-se à situação em que variáveis de um tipo são misturadas com variáveis de outro tipo. Em um comando de atribuição, a *regra de conversão de tipos* é muito simples: o valor do lado direito (o lado da expressão) de uma atribuição é convertido no tipo do lado esquerdo (a variável destino), como ilustrado por este exemplo:

```
int x;
char ch;
float f;

void func(void)
{
   ch = x;      /* linha 1 */
   x = f;       /* linha 2 */
   f = ch;      /* linha 3 */
   f = x;       /* linha 4 */
}
```

Na linha 1, os bits mais significativos da variável inteira **x** são ignorados, deixando **ch** com os 8 bits menos significativos. Se **x** está entre 256 e 0, então **ch** e **x** têm valores idênticos. De outra forma, o valor de **ch** reflete apenas os bits menos significativos de **x**. Na linha 2, **x** recebe a parte inteira de **f**. Na linha 3, **f** converte o valor inteiro de 8 bits armazenado em **ch** no mesmo valor em formato de ponto flutuante. Isso também acontece na linha 4, exceto por **f** converter um valor inteiro de 16 bits no formato de ponto flutuante.

Quando se converte de inteiros para caracteres, inteiros longos para inteiros e inteiros para inteiros curtos, a regra básica é que a quantidade apropriada de bits significativos será ignorada. Isso significa que 8 bits são perdidos quando se vai de inteiro para caractere ou inteiro curto, e 16 bits são perdidos quando se vai de um inteiro longo para um inteiro.

A Tabela 2.3 reúne essas conversões de tipos. Lembre-se de que a conversão de um **int** em um **float** ou **float** em **double** etc. não aumenta a precisão ou exatidão. Esses tipos de conversão apenas mudam a forma em que o valor é representado. Além disso, alguns compiladores C (e processadores) sempre tratam uma variável **char** como positiva, não importando que valor ela tenha quando é convertida para **int** ou **float**. Outros compiladores tratam valores de variáveis **char** maiores que 127 como números negativos. De forma geral, você deve usar variáveis **char** para caracteres e usar **ints**, **short ints** ou **signed chars** quando for necessário evitar um possível problema de portabilidade.

Para utilizar a Tabela 2.3 para fazer uma conversão não mostrada, simplesmente converta um tipo por vez até acabar. Por exemplo, para converter **double** em **int**, primeiro converta **double** em **float** e, então, **float** em **int**.

Linguagens como Pascal proíbem conversão automática de tipos. No entanto, C foi projetada para simplificar a vida do programador, permitindo que o trabalho seja feito em C em vez de assembler. Para substituir o assembler, C tem de permitir essas conversões de tipos.

Tabela 2.3 Conversões de tipos comuns (assumindo uma palavra de 16 bits).

Tipo do destino	Tipo da expressão	Possível informação perdida
signed char	char	Se valor > 127, o destino é negativo
char	short int	Os 8 bits mais significativos
char	int	Os 8 bits mais significativos
char	long int	Os 24 bits mais significativos
int	long int	Os 16 bits mais significativos
int	float	A parte fracionária e possivelmente mais
float	double	Precisão, o resultado é arredondado
double	long double	Precisão, o resultado é arredondado

Atribuições Múltiplas

C permite que você atribua o mesmo valor a muitas variáveis usando atribuições múltiplas em um único comando. Por exemplo, esse fragmento de programa atribui a **x**, **y** e **z** o valor 0:

```
x = y = z = 0;
```

Em programas profissionais, valores comuns são atribuídos a variáveis usando esse método.

Operadores Aritméticos

A Tabela 2.4 lista os operadores aritméticos de C. Os operadores -, +, * e / trabalham em C da mesma forma em que na maioria das outras linguagens. Eles podem ser aplicados em quase qualquer tipo de dado interno permitido em C. Quando / é aplicado a um inteiro ou caractere, qualquer resto é truncado. Por exemplo, 5/2 será igual a 2 em uma divisão inteira.

Tabela 2.4 Operadores aritméticos.

Operador	Ação
-	Subtração, também menos unário
+	Adição
*	Multiplicação
/	Divisão
%	Módulo da divisão (resto)
--	Decremento
++	Incremento

O operador módulo % também trabalha em C da mesma forma que em outras linguagens, devolvendo o resto de uma divisão inteira. Contudo, % não pode ser usado nos tipos em ponto flutuante. O seguinte fragmento de código ilustra %.

```
int x, y;

x = 5;
y = 2;

printf("%d", x/y);  /* mostrará 2 */
printf("%d",x%y);     /* mostrará 1, o resto da divisão inteira */

x = 1;
y = 2;

printf("%d %d", x/y, x%y);  /* mostrará 0 1 */
```

A última linha imprime 0 e 1 porque 1/2 em uma divisão inteira é 0 com resto 1.

O menos unário multiplica seu único operando por -1. Isto é, qualquer número precedido por um sinal de subtração troca de sinal.

Incremento e Decremento

C inclui dois operadores úteis geralmente não encontrados em outras linguagens. São os operadores de incremento e decremento, ++ e --. O operador ++ soma 1 ao seu operando, e -- subtrai 1. Em outras palavras:

```
x = x+1;
```

é o mesmo que

```
++x;
```

e

```
x = x-1;
```

é o mesmo que

```
x--;
```


Ambos os operadores de incremento e decremento podem ser utilizados como prefixo ou sufixo do operando. Por exemplo:

```
x = x+1;
```

pode ser escrito

```
++x;
```

ou

```
x++;
```

Há, porém, uma diferença quando esses operadores são usados em uma expressão. Quando um operador de incremento ou decremento precede seu operando, C executa a operação de incremento ou decremento antes de usar o valor do operando. Se o operador estiver após seu operando, C usará o valor do operando antes de incrementá-lo ou decrementá-lo. O exemplo a seguir:

```
x = 10;
y = ++x;
```

coloca 11 em y. Porém, se o código fosse escrito como

```
x = 10;
y = x++;
```

y receberia 10. Em ambos os casos, x recebe 11; a diferença está em quando isso acontece.

A maioria dos compiladores C produz código-objeto para as operações de incremento e decremento muito rápidas e eficientes — código esse que é melhor que aquele gerado pelo uso da sentença de atribuição equivalente. Por essa razão, você deve usar os operadores de incremento e decremento sempre que puder.

A precedência dos operadores aritméticos é a seguinte:

Mais alta	++ – –
	– (menos unário)
	* / %
Mais baixa	+ –

Operadores do mesmo nível de precedência são avaliados pelo compilador da esquerda para a direita. Obviamente, parênteses podem ser usados para alterar a ordem de avaliação. C trata parênteses da mesma forma que todas as outras

linguagens de programação. Parênteses forçam uma operação, ou um conjunto de operações, a ter um nível de precedência maior.

Operadores Relacionais e Lógicos

No termo *operador relacional*, relacional refere-se às relações que os valores podem ter uns com os outros. No termo *operador lógico*, lógico refere-se às maneiras como essas relações podem ser conectadas. Uma vez que os operadores lógicos e relacionais freqüentemente trabalham juntos, eles serão discutidos aqui em conjunto.

A idéia de verdadeiro e falso é a base dos conceitos dos operadores lógicos e relacionais. Em C, verdadeiro é qualquer valor diferente de zero. Falso é zero. As expressões que usam operadores relacionais ou lógicos devolvem zero para falso e 1 para verdadeiro.

A Tabela 2.5 mostra os operadores lógicos e relacionais. A tabela verdade dos operadores lógicos é mostrada a seguir, usando 1s e 0s.

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Ambos os operadores são menores em precedência do que os operadores aritméticos. Isto é, uma expressão como 10 > 1 + 12 é avaliada como se fosse escrita 10 > (1+12). O resultado é, obviamente, **falso**.

Tabela 2.5 Operadores lógicos e relacionais

Operadores relacionais

Operador	Ação
>	Maior que
>=	Maior que ou igual
<	Menor que
<=	Menor que ou igual
==	Igual
!=	Diferente

Operadores lógicos

Operador	Ação
&&	AND
||	OR
!	NOT

É permitido combinar diversas operações em uma expressão como mostrado aqui:

```
10>5 && !(10 < 9) || 3 <= 4
```

Neste caso, o resultado é **verdadeiro**.

Embora C não tenha um operador lógico OR exclusivo (XOR), você pode facilmente criar uma função que execute essa tarefa usando os outros operadores lógicos. O resultado de uma operação XOR é verdadeiro se, e somente se, um operando (mas não os dois) for verdadeiro. O programa seguinte contém a função **xor()**, que devolve o resultado de uma operação OR exclusivo realizada nos dois argumentos:

```
#include <stdio.h>

int xor(int a, int b);

void main(void)
{
  printf("%d", xor(1, 0));
  printf("%d", xor(1, 1));
  printf("%d", xor(0, 1));
  printf("%d", xor(0, 0));
}

/* Executa uma operação lógica XOR usando os dois argumentos. */
xor(int a, int b)
{
  return(a || b) && !(a && b);
}
```

A tabela seguinte mostra a precedência relativa dos operadores relacionais e lógicos.

maior	!
	> >= < <=
	== !=
	&&
menor	||

Como no caso das expressões aritméticas, é possível usar parênteses para alterar a ordem natural de avaliação de uma expressão relacional e/ou lógica. Por exemplo,

```
!0 && 0 || 0
```

é falso. Porém, quando parênteses são adicionados à mesma expressão, como mostrado aqui, o resultado é verdadeiro:

```
!(0 && 0) || 0
```

Lembre-se de que toda expressão relacional e lógica produz como resultado 0 ou 1. Então, o seguinte fragmento de programa não apenas está correto, como imprimirá o número 1 na tela:

```
int x;

x = 100;
printf("%d", x>10);
```

Operadores Bit a Bit

Ao contrário de muitas outras linguagens, C suporta um completo conjunto de operadores bit a bit. Uma vez que C foi projetada para substituir a linguagem assembly na maioria das tarefas de programação, era importante que ela tivesse a habilidade de suportar muitas das operações que podem ser feitas em linguagem assembly. *Operação bit a bit* refere-se a testar, atribuir ou deslocar os bits efetivos em um byte ou uma palavra, que correspondem aos tipos de dados **char** e **int** e variantes do padrão C. Operações bit não podem ser usadas em **float**, **double**, **long double**, **void** ou outros tipos mais complexos. A Tabela 2.6 lista os operadores que se aplicam às operações bit a bit. Essas operações são aplicadas aos bits individuais dos operandos.

Tabela 2.6 Operadores bit a bit.

Operador	Ação
&	AND
|	OR
^	OR exclusivo (XOR)
~	Complemento de um
>>	Deslocamento à esquerda
<<	Deslocamento à direita

As operações bit a bit AND, OR e NOT (complemento de um) são governadas pela mesma tabela verdade de seus equivalentes lógicos, exceto por trabalharem bit a bit. O OR exclusivo (^) tem a tabela verdade mostrada aqui:

p	q	p ^ q
0	0	0
1	0	1
1	1	0
0	1	1

Como a tabela indica, o resultado de um XOR é verdadeiro apenas se exatamente um dos operandos for verdadeiro; caso contrário, será falso.

Operações bit a bit encontram aplicações mais freqüentemente em "drivers" de dispositivos — como em programas de modems, rotinas de arquivos em disco e rotinas de impressoras — porque as operações bit a bit mascaram certos bits, como o bit de paridade. (O bit de paridade confirma se o restante dos bits em um byte não se modificaram. É geralmente o bit mais significativo em cada byte.)

Imagine o operador AND como uma maneira de desligar bits. Isto é, qualquer bit que é zero, em qualquer operando, faz com que o bit correspondente no resultado seja desligado. Por exemplo, a seguinte função lê um caractere da porta do modem usando a função **read_modem()** e, então, zera o bit de paridade.

```
char get_char_from_modem(void)
{
  char ch;

  ch = read_modem();  /* lê um caractere do modem */
  return (ch & 127);
}
```

A paridade é indicada pelo oitavo bit, que é colocado em 0, fazendo-se um AND com um byte em que os bits de 1 a 7 são 1 e o bit 8 é 0. A expressão **ch & 127** significa fazer um AND bit a bit de **ch** com os bits que compõem o número 127. O resultado é que o oitavo bit de **ch** está zerado. No seguinte exemplo, assuma que **ch** tenha recebido o caractere "A" e que o bit de paridade tenha sido ativado.

```
          Bit de paridade
              |
          1 1 0 0 0 0 0 1    ch contém "A" com a paridade ligada
          0 1 1 1 1 1 1 1    127 em binário
      &   ---------------    faz AND bit a bit
          0 1 0 0 0 0 0 1    "A" sem paridade
```

O operador OR, ao contrário de AND, pode ser usado para ligar um bit. Qualquer bit que é 1, em qualquer operando, faz com que o bit correspondente no resultado seja ligado. Por exemplo, o seguinte mostra a operação 128 ¦ 3:

```
  1 0 0 0 0 0 0 0   128 em binário
  0 0 0 0 0 0 1 1   3 em binário
¦ ---------------   OR bit a bit
  1 0 0 0 0 0 1 1   resultado
```

Um OR exclusivo, normalmente abreviado por XOR, ativa um bit se, e somente se, os bits comparados forem diferentes. Por exemplo, 127^120 é:

```
  0 1 1 1 1 1 1 1   127 em binário
  0 1 1 1 1 0 0 0   120 em binário
^ ---------------   XOR bit a bit
  0 0 0 0 0 1 1 1   resultado
```

Lembre-se de que os operadores lógicos e relacionais sempre produzem um resultado que é 1 ou 0, enquanto as operações similares bit a bit produzem quaisquer valores arbitrários de acordo com a operação específica. Em outras palavras, operações bit a bit podem possuir valores diferentes de 0 e 1, mas os operadores lógicos sempre conduzem a 0 ou 1.

Os operadores de deslocamento, >> e <<, movem todos os bits de uma variável para a direita ou para a esquerda, como especificado. A forma geral do comando de deslocamento à direita é

variável >> *número de posições de bits*

A forma geral do comando de deslocamento à esquerda é

variável << *número de posições de bits*

Conforme os bits são deslocados para uma extremidade, zeros são colocados na outra. Lembre-se de que um deslocamento *não* é uma rotação. Ou seja, os bits que saem por uma extremidade não voltam para a outra. Os bits deslocados são perdidos e zeros são colocados.

Operações de deslocamento de bits podem ser úteis quando se decodifica a entrada de um dispositivo externo, como um conversor D/A, e quando se lêem informações de estado. Os operadores de deslocamento em nível de bits também podem multiplicar e dividir inteiros rapidamente. Um deslocamento à direita efetivamente multiplica um número por 2 e um deslocamento à esquerda divide-o por 2, como mostrado na Tabela 2.7. O programa seguinte ilustra os operadores de deslocamento.

Tabela 2.7 Multiplicação e divisão com operadores de deslocamento.

unsigned char x;	x a cada execução da sentença	Valor de x
x=7;	0 0 0 0 0 1 1 1	7
x=x<<1;	0 0 0 0 1 1 1 0	14
x=x<<3;	0 1 1 1 0 0 0 0	112
x=x<<2;	1 1 0 0 0 0 0 0	192
x=x>>1;	0 1 1 0 0 0 0 0	96
x=x>>2;	0 0 0 1 1 0 0 0	24

Cada deslocamento à esquerda multiplica por 2. Note que se perdeu informação após o x<<2 porque um bit foi deslocado para fora.

Cada deslocamento à direita divide por 2. Note que divisões subseqüentes não trazem de volta bits anteriormente perdidos.

```
/* Um exemplo de deslocamento de bits. */
#include <stdio.h>

void main(void)
{
  unsigned int i;
  int j;

  i = 1;

  /* deslocamentos à esquerda */
  for(j=0; j<4; j++) {
    i = i << 1;  /*  desloca i de 1 à esquerda,
                     que é o mesmo que multiplicar por 2 */
    printf("deslocamento à esquerda %d: %d\n", j, i);
  }

  /* deslocamentos à direita */
  for(j=0; j<4; j++) {
    i = i >> 1;  /* desloca i de 1 à direita,
                    que é o mesmo que dividir por 2 */
    printf("deslocamento à direita %d:%d\n", j, i);
  }
}
```

O operador de complemento a um, ~, inverte o estado de cada bit da variável especificada. Ou seja, todos os 1s são colocados em 0 e todos os 0s são colocados em 1.

Os operadores bit a bit são usados freqüentemente em rotinas de criptografia. Se você deseja fazer um arquivo em disco parecer ilegível, realize algumas manipulações bit a bit nele. Um dos métodos mais simples é complementar cada byte usando o complemento de um para inverter cada bit no byte, como mostrado aqui:

Byte original	0 0 1 0 1 1 0 0	
Após o 1º complemento	1 1 0 1 0 0 1 1	Iguais
Após o 2º complemento	0 0 1 0 1 1 0 0	

Note que uma seqüência de dois complementos produz o número original. Logo, o primeiro complemento representa a versão codificada de cada byte. O segundo complemento decodifica-o ao seu valor original.

Você poderia usar a função **encode()**, mostrada aqui, para codificar um caractere.

```
/* Uma função simples de criptografia. */
char encode(char ch)
{
  return(~ch); /* complementa */
}
```

O Operador ?

C contém um operador muito poderoso e conveniente que substitui certas sentenças da forma if-then-else. O operador ternário **?** tem a forma geral

Exp1 ? Exp2 : Exp3;

onde *Exp1*, *Exp2* e *Exp3* são expressões. Note o uso e o posicionamento dos dois-pontos.

O operador **?** funciona desta forma: *Exp1* é avaliada. Se ela for verdadeira, então *Exp2* é avaliada e se torna o valor da expressão. Se *Exp1* é falsa, então *Exp3* é avaliada e se torna o valor da expressão. Por exemplo, em

```
x = 10;

y = x>9 ? 100 : 200;
```

a **y** é atribuído o valor 100. Se **x** fosse menor que 9, **y** teria recebido o valor 200. O mesmo código, usando o comando **if-else**, é


```
x = 10;

if (x>9) y = 100;
else y = 200;
```

O operador **?** será discutido mais completamente no Capítulo 3 com relação às outras sentenças condicionais de C.

Os Operadores de Ponteiros & e *

Um *ponteiro* é um endereço na memória de uma variável. Uma *variável de ponteiro* é uma variável especialmente declarada para guardar um ponteiro para seu tipo especificado. Saber o endereço de uma variável pode ser de grande ajuda em certos tipos de rotinas. Contudo, ponteiros têm três funções principais em C. Eles podem fornecer uma maneira rápida de referenciar elementos de uma matriz. Os ponteiros também permitem que as funções em C modifiquem seus parâmetros de chamada. Por último, eles suportam listas encadeadas e outras estruturas dinâmicas de dados. O Capítulo 5 é dirigido exclusivamente a ponteiros. Porém, esse capítulo aborda de forma breve os dois operadores que são usados para manipular ponteiros.

O primeiro operador de ponteiro é **&**. Ele é um operador unário que devolve o endereço na memória de seu operando. (Lembre-se de que um operador unário requer apenas um operando.) Por exemplo,

```
m = &count;
```

põe o endereço na memória da variável **count** em **m**. Esse endereço é a posição interna da variável no computador. Ele não tem nenhuma relação com o valor de **count**. Você pode imaginar **&** como significando "o endereço de". Desta forma, a sentença de atribuição anterior significa "**m** recebe o endereço de **count**".

Para entender melhor essa atribuição, assuma que a variável **count** usa a posição de memória 2000 para armazenar seu valor. Também assuma que **count** tem o valor 100. Então, após a atribuição anterior, **m** tem o valor 2000.

O segundo operador é *, que é o complemento de **&**. O * é um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se **m** contém o endereço da variável **count**,

```
q = *m;
```

coloca o valor de **count** em **q**. Agora **q** tem o valor 100 porque 100 está armazenado na posição 2000, o endereço na memória que está armazenado em **m**. Pense no * como significando "no endereço de". Neste caso, a sentença poderia ser lida como "**q** recebe o valor do endereço de **m**".

Infelizmente, o símbolo de multiplicação e o símbolo de "no endereço de" são iguais, e o símbolo para o AND bit a bit e o símbolo de "o endereço de" também são iguais. Esses operadores não têm nenhuma relação um com o outro. Ambos, & e *, têm uma precedência maior que todos os operadores aritméticos, exceto o menos unário, que tem a mesma precedência.

Variáveis que guardam ponteiros devem ser declaradas como tal. Variáveis que armazenam endereços da memória, ou ponteiros, como são chamados em C, devem ser declarados colocando-se * em frente ao nome da variável para indicar ao compilador que ela guardará um ponteiro para aquele tipo de variável. Por exemplo, para declarar uma variável ponteiro **ch** para **char**, escreva

```
char *ch;
```

Aqui, **ch** não é um caractere, mas um ponteiro para caractere — há uma grande diferença. O tipo de dado que o ponteiro aponta, neste caso **char**, é chamado o *tipo base* do ponteiro. De qualquer forma, a variável ponteiro é uma variável que mantém o endereço de um objeto do tipo base. Logo, um ponteiro para caractere (ou qualquer ponteiro) é de tamanho suficiente para guardar um endereço como definido pela arquitetura do computador em que está rodando. Lembre-se de que um ponteiro deve ser usado apenas para apontar para dados que são do tipo base do ponteiro.

Você pode misturar diretivas de ponteiro e de não-ponteiros na mesma declaração. Por exemplo:

```
int x, *y, count;
```

declara **x** e **count** como sendo do tipo inteiro e **y** como um ponteiro para o tipo inteiro.

Os seguintes operadores * e & põem o valor 10 na variável chamada **target**. Como esperado, esse programa mostra o valor 10 na tela.


```
#include <stdio.h>

void main(void)
{
  int target, source;
  int *m;

  source = 10;
  m = &source;
  target = *m;

  printf("%d", target);
}
```

O Operador em Tempo de Compilação sizeof

O operador **sizeof** é um operador em tempo de compilação unário que retorna o tamanho, em bytes, da variável ou especificador de tipo, em parênteses, que ele precede. Por exemplo, assumindo que inteiros são de 2 bytes e que **floats** são de 8 bytes,

```
float f;

printf("%f", sizeof f);
printf("%d", sizeof (int));
```

irá mostrar na tela **8 2**.

Lembre-se de que para calcular o tamanho de um tipo, o nome do tipo deve ser colocado entre parênteses. Isso não é necessário para nomes de variáveis, embora não haja qualquer mal em fazê-lo.

O padrão ANSI (usando **typedef**) define um tipo especial chamado **size_t**, que corresponde de forma imprecisa a um inteiro sem sinal. Tecnicamente, o valor devolvido por **sizeof** é do tipo **size_t**. Para todos os fins práticos, porém, você pode imaginá-lo (e usá-lo) como se fosse um valor sem sinal.

sizeof ajuda basicamente a gerar códigos portáveis que dependam do tamanho dos tipos de dados internos de C. Por exemplo, imagine um programa de banco de dados que precise armazenar seis valores inteiros por registro. Se você quer transportar o programa de banco de dados para vários computadores, não deve assumir o tamanho de um inteiro, mas deve determinar o tamanho real do inteiro usando **sizeof**. Sendo esse o caso, você poderia usar a seguinte rotina para escrever um registro em um arquivo em disco:

```
/* Escreve 6 inteiros em um arquivo em disco. */
void put_rec(int rec[6], FILE *fp)
{
  int len;

  len = fwrite(rec, sizeof rec, 1, fp);
  if(len != 1) printf("erro de escrita");
}
```

Codificada como mostrado, **put_rec()** compila e roda corretamente em qualquer computador, não importando quantos bytes tenha um inteiro.

O Operador Vírgula

O operador vírgula é usado para encadear diversas expressões. O lado esquerdo de um operador vírgula é sempre avaliado como **void**. Isso significa que a expressão do lado direito torna-se o valor de toda a expressão separada por vírgulas. Por exemplo,

```
x = (y=3, y+1);
```

primeiro atribui o valor 3 a **y** e, em seguida, atribui o valor 4 a **x**. Os parênteses são necessários porque o operador vírgula tem uma precedência menor que o operador de atribuição.

Essencialmente, a vírgula provoca uma seqüência de operações. Quando ela é usada do lado direito de uma sentença de atribuição, o valor atribuído é o valor da última expressão da lista separada por vírgulas.

O operador vírgula tem, de certa forma, o mesmo significado da palavra e em português normal, como na frase "faça isso e isso e isso".

Os Operadores Ponto (.) e Seta (->)

Os operadores . (ponto) e -> (seta) referenciam elementos individuais de estruturas e uniões. *Estruturas* e *uniões* são tipos de dados compostos que podem ser referenciados segundo um único nome (veja o Capítulo 7).

O operador ponto é usado quando se está referenciando a estrutura ou união real. O operador seta é usado quando um ponteiro para uma estrutura é usado. Por exemplo, dada a estrutura global

```
struct employee
{
  char name[80];
  int age;
  float wage;
} emp;

struct employee *p = &emp; /* endereço de emp em p */
```


você escreveria o seguinte código para atribuir o valor 123.23 ao elemento **wage** da estrutura **emp**:

```
emp.wage = 123.23;
```

No entanto, a mesma atribuição, usando um ponteiro para **emp**, seria

```
p->wage = 123.23;
```

Parênteses e Colchetes Como Operadores

Em C, parênteses são operadores que aumentam a precedência das operações dentro deles.

Colchetes realizam indexação de matrizes (eles serão discutidos no Capítulo 4). Dada uma matriz, a expressão dentro de colchetes provê um índice dentro dessa matriz. Por exemplo:

```
#include <stdio.h>
char s[80];

void main(void)
{
  s[3] = 'X';
  printf("%c", s[3]);
}
```

Esse código primeiro atribui o valor **'X'** ao quarto elemento (lembre-se de que todas as matrizes em C começam em 0) da matriz **s** e imprime esse elemento.

Resumo das Precedências

A Tabela 2.8 lista a precedência de todos os operadores de C. Note que todos os operadores, exceto os operadores unários e ?, associam da esquerda para a direita. Os operadores unários (*, & e -) e ? associam da direita para a esquerda.

Tabela 2.8 A precedência dos operadores em C.

Maior	() [] ->
	! ~ ++ -- - (tipo) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	!
	&&
	! !
	?:
	= += -= *= /= etc.
Menor	,

Expressões

Operadores, constantes e variáveis são os elementos que constituem as expressões. Uma *expressão* em C é qualquer combinação válida desses elementos. Uma vez que a maioria das expressões tende a seguir as regras gerais da álgebra, elas são freqüentemente tomadas como certas. Contudo, existem uns poucos aspectos de expressões que se referem especificamente a C.

Ordem de Avaliação

O padrão C ANSI não estipula que as subexpressões de uma expressão devam ser avaliadas em uma ordem especificada. Isso deixa o compilador livre para rearranjar uma expressão para produzir o melhor código. No entanto, isso também significa que seu código nunca deve contar com a ordem em que as subexpressões são avaliadas. Por exemplo, a expressão

```
x = f1() + f2();
```

não garante que **f1()** será chamada antes de **f2()**.

Conversão de Tipos em Expressões

Quando constantes e variáveis de tipos diferentes são misturadas em uma expressão, elas são convertidas a um mesmo tipo. O compilador C converte todos os operandos no tipo do maior operando, o que é denominado *promoção de tipo*. Isso é feito operação por operação, como descrito nas regras de conversão de tipos abaixo.

SE um operando é **long double**
ENTÃO o segundo é convertido para **long double**
SENÃO, SE um operando é **double**
ENTÃO o segundo é convertido para **double**
SENÃO, SE um operando é **float**
ENTÃO o segundo é convertido para **float**
SENÃO, SE um operando é **unsigned long**
ENTÃO o segundo é convertido para **unsigned long**
SENÃO, SE um operando é **long**
ENTÃO o segundo é convertido para **long**
SENÃO, SE um operando é **unsigned int**
ENTÃO o segundo é convertido para **unsigned int**

Há ainda um caso adicional especial: se um operando é **long** e o outro é **unsigned int**, e se o valor do **unsigned int** não pode ser representado por um **long**, os dois operandos são convertidos para **unsigned long**.

Uma vez que essas regras de conversão tenham sido aplicadas, cada par de operandos é do mesmo tipo e o resultado de cada operação é do mesmo tipo de ambos os operandos.

Por exemplo, considere as conversões de tipo que ocorrem na Figura 2.3. Primeiro, o caratere **ch** é convertido para um inteiro e **float f** é convertido para **double**. Em seguida, o resultado de **ch/i** é convertido para **double** porque **f*d** é **double**. O resultado final é **double** porque, nesse momento, os dois operandos são **double**.

Casts

Você pode forçar uma expressão a ser de um determinado tipo usando um *cast*. A forma genérica de um cast é

(tipo) expressão

onde *tipo* é qualquer tipo de dados válido em C. Por exemplo, para garantir que a expressão **x/2** resulte em um valor do tipo **float**, escreva

```
(float) x/2;
```

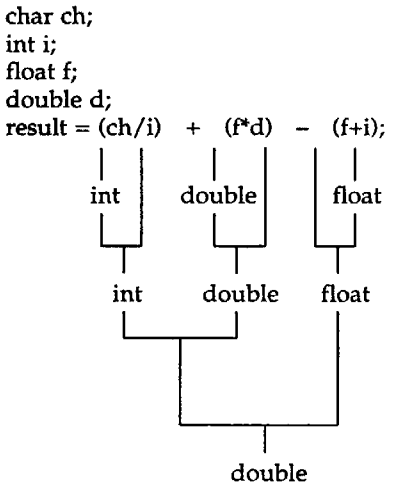


Figura 2.3 Um exemplo de conversão de tipo.

Casts são operadores tecnicamente. Como operador, um cast é unário e possui a mesma precedência que qualquer outro operador unário.

Embora os casts não sejam muito usados em programação, eles podem ser muito úteis quando necessários. Por exemplo, suponha que você deseje usar um inteiro para controlar uma repetição, no entanto as operações sobre o valor exigem uma parte fracionária, como no programa seguinte:

```
#include <stdio.h>

void main(void) /* imprime i e i/2 com frações */
{
  int i;

  for (i=1; i<=100; ++i)
    printf("%d / 2 é: %f\n", i, (float) i /2);
}
```

Sem o cast **(float)**, teria sido efetuada somente uma divisão inteira. O cast garante que a parte fracionária do resultado seja exibida.

Espaçamento e Parênteses

Você pode acrescentar tabulações e espaços a expressões em C para torná-las mais legíveis. Por exemplo, as duas próximas expressões são a mesma:

```
x=10/y ~ (127/x);

x = 10 / y ~ (127/x);
```

Parênteses redundantes ou adicionais não causam erros nem diminuem a velocidade de execução da expressão. Você deve usar parênteses para esclarecer a ordem de avaliação, tanto para você mesmo como para os demais. Por exemplo, quais das duas expressões a seguir é mais fácil de ler?

```
x=y/2-34*temp&127;

x = (y/3) - ((34*temp) &127);
```

C Reduzido

Existe uma variante do comando de atribuição, às vezes chamada de *C reduzido*, que simplifica a codificação de um certo tipo de operações de atribuição. Por exemplo,

```
x = x+10;
```

pode ser escrito

```
x += 10;
```

O par operador **+=** diz ao compilador para atribuir a **x** o valor de **x** mais 10.

Essas abreviações existem para todos os operadores binários em C (aqueles que requerem dois operandos). A forma geral de uma abreviação C como:

var = var operador expressão

é o mesmo que

var operador = expressão

Considere um outro exemplo:

```
x = x-100;
```

é o mesmo que

```
x -= 100;
```

Você verá a notação abreviada sendo utilizada largamente em programas escritos profissionalmente em C; você deve familiarizar-se com ela.

MAKRON Books

3

Comandos de Controle do Programa

Este capítulo discute os ricos e variados comandos de controle do programa em C. O padrão ANSI divide os comandos de C nestes grupos:

- Seleção
- Iteração
- Desvio
- Rótulo
- Expressão
- Bloco

Estão incluídos nos comando de seleção **if** e **switch**. (O termo "comando condicional" é freqüentemente usado em lugar de "comando de seleção". No entanto, o padrão ANSI usa "seleção", como também este livro.) Os comandos de iteração são **while, for** e **do-while**. São também normalmente chamados de comandos de laço. Os comandos de salto ou desvio são **break, continue, goto** e **return**. Os comandos de rótulo são **case** e **default** (discutidos juntamente com o comando **switch**) e o comando **label** (discutido com **goto**). Sentenças de expressão são aquelas compostas por uma expressão C válida. Sentenças de bloco são simplesmente blocos de código. (Lembre-se de que um bloco começa com um { e termina com um }.)

Como muitos comandos em C contam com a saída de alguns testes condicionais, começaremos revendo os conceitos de verdadeiro e falso em C.

Verdadeiro e Falso em C

Muitos comandos em C contam com um teste condicional que determina o curso da ação. Uma expressão condicional chega a um valor verdadeiro ou falso. Em C, ao contrário de muitas outras linguagens, um valor verdadeiro é qualquer valor diferente de zero, incluindo números negativos. Um valor falso é 0. Esse método para verdadeiro e falso permite que uma ampla gama de rotinas sejam codificadas de forma extremamente eficiente, como você verá em breve.

Comandos de Seleção

C suporta dois tipos de comandos de seleção: **if** e **switch**. Além disso, o operador **?** é uma alternativa ao **if** em certas circunstâncias.

if

A forma geral da sentença **if** é

if (*expressão*) *comando*;
else *comando*;

onde *comando* pode ser um único comando, um bloco de comandos ou nada (no caso de comandos vazios). A cláusula **else** é opcional.

Se a *expressão* é verdadeira (algo diferente de 0), o comando ou bloco que forma o corpo do **if** é executado; caso contrário, o comando ou bloco que é o corpo do **else** (se existir) é executado. Lembre-se de que apenas o código associado ao **if** ou o código associado ao **else** será executado, nunca ambos.

O comando condicional controlando o **if** deve produzir um resultado escalar. Um *escalar* é um inteiro, um caractere ou tipo de ponto flutuante. No entanto, é raro usar um número em ponto flutuante para controlar um comando condicional, porque isso diminui consideravelmente a velocidade de execução. (A CPU executa diversas instruções para efetuar uma operação em ponto flutuante. Ela usa relativamente poucas instruções para efetuar uma operação com caractere ou inteiro.)

Por exemplo, considere o programa a seguir, que é uma versão muito simples do jogo de adivinhar o "número mágico". Ele imprime a mensagem "**** Certo ****" quando o jogador acerta o número mágico. Ele produz o número mágico usando o gerador de números randômicos de C, que devolve um número arbitrário entre 0 e **RAND-MAX** (que define um valor inteiro que é 32.767 ou maior) exige o arquivo de cabeçalho **stdlib.h**.


```
/* Programa de números mágicos #1. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Certo **");
}
```

Levando o programa do número mágico adiante, a próxima versão ilustra o uso da sentença **else** para imprimir outra mensagem em resposta a um número errado.

```
/* Programa de números mágicos #2. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Certo **");
  else printf("Errado");
}
```

ifs Aninhados

Um **if** aninhado é um comando **if** que é o objeto de outro **if** ou **else**. ifs aninhados são muito comuns em programação. Em C, um comando **else** sempre se refere

ao comando **if** mais próximo, que está dentro do mesmo bloco do **else** e não está associado a outro **if**. Por exemplo

```
if(i)
{
  if(j) comando 1;
  if(k) comando 2; /* este if */
  else  comando 3; /* está associado a este else */
}
else comando 4;  /* associado a if(i) */
```

Como observado, o último **else** não está associado a **if(j)** porque não pertence ao mesmo bloco. Em vez disso, o último **else** está associado ao **if(i)**. O **else** interno está associado ao **if(k)**, que é o **if** mais próximo.

O padrão C ANSI especifica que pelo menos 15 níveis de aninhamento devem ser suportados. Na prática, a maioria dos compiladores permite substancialmente mais.

Você pode usar um **if** aninhado para melhorar o programa do número mágico dando ao jogador uma realimentação sobre um palpite errado.

```
/* Programa de números mágicos #3. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("*** Certo ***");
    printf(" %d é o número mágico\n", magic);
  }
  else {
    printf("Errado, ");
    if(guess > magic)  printf("muito alto\n");
    else printf("muito baixo\n");
  }
}
```


A Escada if-else-if

Uma construção comum em programação é a forma *if-else-if*, algumas vezes chamada de *escada if-else-if* devido a sua aparência. A sua forma geral é

```
if(expressão)comando;
else
  if(expressão)comando;
  else
    if(expressão)comando;
.
.
.
      else comando;
```

As condições são avaliadas de cima para baixo. Assim que uma condição verdadeira é encontrada, o comando associado a ela é executado e desvia do resto da escada. Se nenhuma das condições for verdadeira, então o último **else** é executado. Isto é, se todos os outros testes condicionais falham, o último comando **else** é efetuado. Se o último **else** não está presente, nenhuma ação ocorre se todas as condições são falsas.

Embora seja tecnicamente correta, o recuo da escada if-else-if anterior pode ser excessivamente profundo. Por essa razão, a escada if-else-if é geralmente recuada desta forma:

```
if(expressão)
  comando;
else if(expressão)
  comando;
else if(expressão)
  comando;
.
.
.
else
  comando;
```

Usando uma escada if-else-if, o programa do número mágico torna-se:

```
/* Programa de números mágicos #4. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
```

```
{
  int magic;  /* número mágico */
  int guess;  /* palpite do usuário */

  magic = rand();  /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("*** Certo ***");
    printf("%d é o número mágico", magic);
  }
  else if(guess > magic)
   printf("Errado, muito alto");
  else printf("Errado, muito baixo");
}
```

O ? Alternativo

Você pode usar o operador **?** para substituir comandos **if-else** na forma geral:

if(condição) expressão;
else *expressão;*

Contudo, os corpos de **if** e **else** devem ser uma expressão simples — nunca um outro comando de C.

O **?** é chamado de um *operador ternário* porque ele requer três operandos. Ele tem a seguinte forma geral

Exp1 ? *Exp2* : *Exp3*

onde *Exp1*, *Exp2* e *Exp3* são expressões. Note o uso e o posicionamento dos dois-pontos.

O valor de uma expressão **?** é determinada como segue: *Exp1* é avaliada. Se for verdadeira, *Exp2* será avaliada e se tornará o valor da expressão **?** inteira. Se *Exp1* é falsa, então *Exp3* é avaliada e se torna o valor da expressão. Por exemplo, considere

```
x = 10;
y = x>9 ? 100 : 200;
```


Nesse exemplo, o valor 100 é atribuido a **y**. Se **x** fosse menor que 9, **y** teria recebido o valor 200. O mesmo código escrito com o comando **if-else** seria

```
x = 10;
if (x>9) y = 100;
else y = 200;
```

O seguinte programa usa o operador **?** para elevar ao quadrado um valor inteiro digitado pelo usuário. Contudo, este programa preserva o sinal (10 ao quadrado é 100 e -10 ao quadrado é -100).

```
#include <stdio.h>

void main(void)
{
  int isqrd, i;

  printf("Digite um número: ");
  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);

  printf("%d ao quadrado é %d", i, isqrd);
}
```

O uso do operador **?** para substituir comandos **if-else** não é restrito a atribuições apenas. Lembre-se de que todas as funções (exceto aquelas declaradas como **void**) podem retornar um valor. Logo, você pode usar uma ou mais chamadas a funções em uma expressão em C. Quando o nome da função é encontrado, a função é executada e, então, seu valor de retorno pode ser determinado. Portanto, você pode executar uma ou mais chamadas a funções usando o operador **?**, colocando as chamadas nas expressões que formam os operandos, como em

```
#include <stdio.h>

int f1(int n);
int f2(void);

void main(void)
{
  int t;

  printf("Digite um número: ");
```

```
  scanf("%d", &t);

  /* imprime a mensagem apropriada */
  t ? f1(t) +  f2() : printf("foi digitado zero");
}

f1(int n)
{
  printf("%d ", n);
  return 0;
}

f2(void)
{
  printf("foi digitado");
  return 0;
}
```

Inserir um 0 nesse exemplo faz com que a função **printf()** seja chamada, mostrando a mensagem **foi digitado zero**. Se você inseriu qualquer outro número, **f1()** e **f2()** serão executadas. Note que o valor da expressão **?** é descartado nesse exemplo. Você não precisa atribuí-lo a nada.

Um aviso: alguns compiladores C rearranjam a ordem de avaliação de uma expressão numa tentativa de otimizar o código-objeto. Isso pode fazer com que as funções que formam os operandos do operador **?** sejam executadas em uma seqüência diferente da pretendida.

Usando o operador **?**, você pode reescrever o programa do número mágico mais uma vez.

```
/* Número mágico programa #5. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int magic;
  int guess;

  magic = rand(); /* gera o número mágico */

  printf("Adivinhe o número mágico: ");
  scanf("%d", &guess);

  if(Guess == magic) {
    printf("** Certo **");
    printf("%d é o número mágico", magic);
  }
  else
    guess > magic ? printf("Alto") : printf("Baixo");
}
```


Aqui, o operador **?** mostra a mensagem apropriada baseado no resultado do teste **guess > magic**.

A Expressão Condicional

Algumas vezes, os iniciantes em C se confundem pelo fato de que você pode usar qualquer expressão válida em C para controlar o **if** ou o operador **?**. Isto é, você não fica restrito a expressões envolvendo os operadores relacionais e lógicos (como é o caso nas linguagens como BASIC ou Pascal). O programa precisa simplesmente chegar a um valor zero ou não-zero. Por exemplo, o seguinte programa lê dois inteiros do teclado e mostra o quociente. Ele usa um comando **if**, controlado pelo segundo número, para evitar um erro de divisão por zero.

```
/* Divide o primeiro número pelo segundo. */

#include <stdio.h>

void main(void)
{
  int a, b;

  printf("Digite dois números: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("não pode dividir por zero\n");
}
```

Esse método funciona porque, se **b** é 0, a condição que controla **if** é falsa e o **else** é executado. De outra forma, a condição é verdadeira (não-zero) e a divisão é efetuada. Contudo, escrever o comando **if** desta forma:

```
if(b != 0) printf("%d\n", a/b);
```

é redundante, potencialmente ineficiente e considerado mau estilo.

switch

C tem um comando interno de seleção múltipla, **switch**, que testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere. Quando o valor coincide, os comandos associados àquela constante são executados. A forma geral do comando **switch** é

```
switch(expressão) {
case constante1:
        seqüência de comandos
        break;
case constante2:
        seqüência de comandos
        break;
case constante3:
        seqüência de comandos
        break;
.
.
.
default:
        seqüência de comandos
}
```

O valor da *expressão* é testado, na ordem, contra os valores das constantes especificadas nos comandos **case**. Quando uma coincidência for encontrada, a seqüência de comandos associada àquele **case** será executada até que o comando **break** ou o fim do comando **switch** seja alcançado. O comando **default** é executado se nenhuma coincidência for detectada. O **default** é opcional e, se não estiver presente, nenhuma ação será realizada se todos os testes falharem.

O padrão ANSI C especifica que um **switch** pode ter pelo menos 257 comandos **case**. Na prática, você desejará limitar o número de comandos **case** a uma quantidade menor, para obter mais eficiência. Embora **case** seja um rótulo, ele não pode existir sozinho, fora de um **switch**.

O comando **break** é um dos comandos de desvio em C. Você pode usá-lo em laços tal como no comando **switch** (veja a seção "Comandos de Iteração"). Quando um **break** é encontrado em um **switch**, a execução do programa "salta" para a linha de código seguinte ao comando **switch**.

Há três coisas importantes a saber sobre o comando **switch**:

- O comando **switch** difere do comando **if** porque **switch** só pode testar igualdade, enquanto **if** pode avaliar uma expressão lógica ou relacional.

- Duas constantes **case** no mesmo **switch** não podem ter valores idênticos. Obviamente, um comando **switch** incluído em outro switch mais externo pode ter as mesmas constantes **case**.
- Se constantes de caractere são usadas em um comando **switch**, elas são automaticamente convertidas para seus valores inteiros.

O comando **switch** é freqüentemente usado para processar uma entrada, via teclado, como em uma seleção por menu. Como mostrado aqui, a função **menu()** mostra o menu de um programa de validação da ortografia e chama os procedimentos apropriados:

```
void menu (void)
{
  char ch;

  printf("1. Checar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("Pressione Qualquer Outra Tecla para Abandonar\n");
  printf("     Entre com sua escolha:   ");

  ch=getchar();  /* Lê do teclado a seleção */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :
      printf("Nenhuma opção selecionada");
    }
}
```

Tecnicamente, os comandos **break**, dentro do **switch**, são opcionais. Eles terminam a seqüência de comandos associados com cada constante. Se o comando **break** é omitido, a execução continua pelos próximos comandos **case** até que um **break**, ou o fim do **switch**, seja encontrado. Por exemplo, a seguinte função usa a natureza de "passar caindo" pelos **cases** para simplificar o código de um device-driver que manipula a entrada:

```
/* Processa um valor */
void inp_handler(int i)
{
  int flag;

  flag = -1;

  switch(i) {
    case 1:  /* Estes cases têm uma seqüência */
    case 2:  /* de comandos em comum */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

Essa rotina ilustra dois aspectos do comando **switch**. Primeiro, você pode ter comandos case sem comandos associados. Quando isto ocorre, a execução simplesmente cai no **case** seguinte. Nesse caso, todos os três primeiros **cases** executam os mesmos comandos.

```
flag = 0;
break;
```

Segundo, a execução continua no próximo **case** se nenhum comando **break** estiver presente. Se **i** for igual a 4, **flag** receberá o valor 1 e, como não há nenhum comando **break** no fim desse **case**, a execução continuará e a função **error(flag)** será chamada. Se **i** fosse igual a 5, **error(flag)** teria sido chamada com o valor -1 em **flag**.

O fato de os **cases** poderem ser executados em conjunto quando nenhum **break** estiver presente evita a duplicação indesejável de código, resultando em um código muito eficiente.

Note que os comandos associados a cada **case** não são blocos de código mas, sim, *seqüências de comandos*. (Obviamente, o comando **switch** inteiro define um bloco.) Essa distinção técnica é importante apenas em certas situações. Por exemplo, o seguinte fragmento de código está errado e não será compilado porque você não pode declarar uma variável local em uma seqüência de comandos (você só pode declarar variáveis locais no início de um bloco.)


```
/* Isto está incorreto. */
switch(c) {
   case 1:
     int t;
     .
     .
     .
```

Embora um pouco estranho, você poderia adicionar uma variável local, como mostrado aqui. Neste caso, a declaração ocorre no início do bloco do **switch**.

```
/* Embora estranho, isto está correto. */
switch(c)
{
   int t;
   case 1:
     .
     .
     .
```

Obviamente, você pode criar um bloco de código como um dos comandos da seqüência e declarar uma variável local dentro dele, como exibido aqui:

```
/* Isto é correto. */
switch (c) {
  case 1:
    { /* Cria bloco */
       int t;
      .
      .
      .
    }
    .
    .
    .
```

Comandos switch Aninhados

Você pode ter um **switch** como parte de uma seqüência de comandos de um outro **switch**. Mesmo se as constantes dos **cases** dos **switchs** interno e externo possuírem valores comuns, não ocorrerão conflitos. Por exemplo, o seguinte fragmento de código é perfeitamente aceitável:

```
switch(x) {
   case 1:
     switch(y) {
       case 0: printf ("erro de divisão por zero");
               break;
       case 1: process(x,y);
     }
     break;
   case 2:
     .
     .
     .
```

Comandos de Iteração

Em C, e em todas as outras linguagens modernas de programação, comandos de iteração (também chamados laços) permitem que um conjunto de instruções seja executado até que ocorra uma certa condição. Essa condição pode ser predefinida (como no laço **for**) ou com o final em aberto (como nos laços **while** e **do-while**).

O Laço for

O formato geral do laço **for** de C é encontrado, de uma forma ou de outra, em todas as linguagens de programação baseadas em procedimentos. Contudo, em C, ele fornece flexibilidade e capacidade surpreendentes.

A forma geral do comando **for** é

for(*inicialização*; *condição*; *incremento*) *comando*;

O laço **for** permite muitas variações. Entretanto, a inicialização é, geralmente, um comando de atribuição que é usado para colocar um valor na variável de controle do laço. A *condição* é uma expressão relacional que determina quando o laço acaba. O *incremento* define como a variável de controle do laço varia cada

vez que o laço é repetido. Você deve separar essas três seções principais por pontos-e-vírgulas. Uma vez que a condição se torne falsa, a execução do programa continua no comando seguinte ao **for**.

Por exemplo, o seguinte programa imprime os números de 1 a 100 na tela:

```
#include <stdio.h>

void main(void)
{
   int x;

   for(x=1; x <= 100; x++) printf("%d ",x);
}
```

No programa, **x** é inicialmente ajustado para 1. Uma vez que **x** é menor que 100, **printf()** é executado e **x** é incrementado em 1 e testado para ver se ainda é menor ou igual a 100. Esse processo se repete até que **x** fique maior que 100; nesse ponto, o laço termina. Nesse exemplo, x é a variável de controle do laço, que é alterada e testada toda vez que o laço se repete.

O seguinte exemplo é um laço **for** que contém múltiplos comandos:

```
for(x=100; x != 65; x-=5) {
   z = x*x;
   printf("O quadrado de %d, %f",x, z);
}
```

Tanto a multiplicação de **x** por si mesmo como a função **printf()** são executadas até que **x** seja igual a 65. Note que o laço *é executado de forma inversa*: **x** é inicializado com 100 e será subtraído de 5 cada vez que o laço se repetir.

Nos laços **for**, o teste condicional sempre é executado no topo do laço. Isso significa que o código dentro do laço pode não ser executado se todas as condições forem falsas logo no início. Por exemplo, em

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y); /* este é o único comando
                    printf() que executará */
```

o laço nunca executará, pois **x** e **y** já são iguais desde a sua entrada. Já que a expressão condicional é avaliada como falsa, nem o corpo do laço nem a porção de incremento do laço são executados. Logo, y ainda tem o valor 10 e a saída é o número 10 escrito apenas uma vez na tela.

Variações do Laço for

A discussão anterior descreveu a forma mais comum do laço **for**. Porém, C oferece diversas variações que aumentam a flexibilidade e a aplicação do laço **for**.

Uma das variações mais comuns usa o operador vírgula para permitir que duas ou mais variáveis controlem o laço. (Lembre-se de que você usa o operador vírgula para encadear expressões numa configuração "faça isto e isto". Veja o Capítulo 2.) Por exemplo, as variáveis **x** e **y** controlam o seguinte laço e ambas são inicializadas dentro do comando **for**:

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y-'0'; /* subtrai o código ASCII do caractere 0 de y */
    .
    .
    .
}
```

Vírgulas separam os dois comandos de inicialização. Cada vez que **x** é incrementado, o laço repete e o valor de **y** é ajustado pela entrada do teclado. Tanto **x** como **y** devem ter o valor correto para que o laço termine. Embora os valores de **y** sejam definidos pela entrada do teclado, **y** deve ser inicializado com 0 de forma que seu valor seja definido antes da avaliação da expressão condicional. (Se **y** não fosse definido, ele poderia conter um 10, tornando o teste condicional falso e impedindo a execução do laço.)

A função **converge()**, mostrada a seguir, ilustra múltiplas variáveis de controle de laço. A função **converge()** expõe uma string escrevendo os caracteres vindos de ambas as extremidades, convergindo no meio da linha especificada. Isso requer um posicionamento do cursor em vários e desconexos pontos da tela. Uma vez que C roda sob uma ampla variedade de ambientes, ele não define uma função de posicionamento do cursor. Porém, virtualmente, todo compilador C fornece uma, embora seu nome possa variar. O programa a seguir usa a função **gotoxy()**, do Borland, para posicionar o cursor. (Ela exige o cabeçalho **conio.h**.)

```
/* Versão Borland. */
#include <stdio.h>
#include <conio.h> /* arquivo de cabeçalho não-padrão */
#include <string.h>

void converge(int line, char *message);
```



```
void main (void)
{
  converge(10, "Isto é um teste de converge().");
}

/* Essa função mostra uma string iniciando do lado esquerdo da
   linha especificada. Ela escreve caracteres de ambas as
   extremidades, convergindo para o centro. */
void converge(int line, char *message)
{
  int i, j;

  for(i=1, j=strlen(message); i<j; i++, j--) {
    gotoxy(i, line); printf("%c", message[i-1]);
    gotoxy(j, line); printf("%c", message[j-1]);
  }
}
```

No Microsoft C, o equivalente a **gotoxy()** é **_settextposition()**, que usa o arquivo de cabeçalho **graph.h**. O programa anterior, recodificado para o Microsoft C, é mostrado aqui:

```
/* Versão para Microsoft. C */
#include <stdio.h>
#include <graph.h> /* arquivo de cabeçalho não-padrão */
#include <string.h>

void converge(int line, char *message);

void main(void)
{
  converge(10, "Isto é um teste de converge().");
}

/* Essa função mostra uma string iniciando do lado esquerdo
   da linha especificada. Ela escreve caracteres de ambas as
   extremidades, convergindo para o centro.  */
void converge(int line, char *message)
{
  int i, j;

  for(i=1, j=strlen(message); i<j; i++, j--) {
    _settextposition(line, i);
    printf("%c", message[i-1]);
```

```
      _settextposition(line, j);
      printf("%c", message[j-1]);
    }
  }
```

Se você usa um compilador C diferente, será necessária uma verificação nos seus manuais do usuário para saber o nome da função de posicionamento do cursor.

Nas duas versões de **converge()**, o laço **for** usa duas variáveis de controle do laço, **i** e **j**, para indexar a string pelas extremidades. Quando o laço repete, **i** aumenta e **j** diminui. O laço parará quando **i** for igual a **j**, garantindo, assim, que todos os caracteres sejam escritos.

A expressão condicional não precisa envolver um teste com a variável que controla o laço e algum valor final. Na realidade, a condição pode ser qualquer sentença relacional ou lógica. Isso significa que você pode testar diversas possíveis formas de término.

Por exemplo, você poderia usar a seguinte função para conectar um usuário a um sistema remoto. O usuário tem três tentativas para entrar com a senha. O laço termina quando as três tentativas forem utilizadas ou o usuário entrar com a senha correta.

```
void sign_on(void)
{
  char str[20] ;

  int x;

  for(x=0; x<3 && strcmp(str, "senha"); ++x) {
    printf("Digite a senha por favor:");
    gets(str);
  }

  if (x==3) return;
  /* else conecte o usuário ... */
}
```

Essa função usa **strcmp()**, a função da biblioteca padrão que compara duas strings e retorna zero se forem iguais.

Lembre-se de que cada uma das três seções pode consistir em qualquer expressão válida em C. As expressões não precisam ter relação alguma com os usos mais comuns das seções. Com isso em mente, considere o seguinte exemplo:

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



```
int sqrnum(int num);
int readnum(void);
int prompt(void);

void main(void)
{
  int t;

  for(prompt(); t=readnum(); prompt())
    sqrnum(t);
}

prompt(void)
{
  printf("Digite um número: ");
  return 0;
}

readnum(void)
{
  int t;
  scanf("%d", &t);
  return t;
}

sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```

Olhe atentamente o laço **for** em **main()**. Note que cada parte do laço **for** é composta de chamadas a funções que interagem com o usuário e lêem um número inserido pelo teclado. Se o número inserido for 0, o laço terminará, porque a expressão condicional será falsa. Caso contrário, o número é elevado ao quadrado. Assim, esse laço **for** usa as porções de inicialização e incremento de uma maneira não tradicional, mas completamente válida.

Uma outra característica interessante do laço **for** é que partes da definição do laço não precisam existir. De fato, não há necessidade de que uma expressão esteja presente em nenhuma das seções — as expressões são opcionais. Por exemplo, esse laço será executado até que o usuário insira o número **123**:

```
for(x=0; x!=123; ) scanf("%d", &x);
```

Note que a porção de incremento da definição do **for** está vazia. Isso significa que cada vez que o laço se repetir, x será testado para verificar se é igual a 123, mas nenhuma atitude adicional será tomada. Se você digitar **123** no teclado, porém, a condição do laço se tornará falsa e o laço terminará.

Freqüentemente a inicialização é feita fora do comando **for**. Isso geralmente acontece quando a condição inicial da variável de controle do laço for calculada por algum método complexo, como neste exemplo:

```
gets(s);  /* lê uma string para s */
if(*s) x = strlen(s); /* obtém o comprimento da string */
else x = 10;

for( ;x<10; ) {
  printf("%d",x);
  ++x;
}
```

A seção de inicialização foi deixada em branco e x é inicializado antes que o laço comece a ser executado.

O Laço Infinito

Embora você possa usar qualquer comando de laço para criar um laço infinito, o **for** é tradicionalmente usado para esse fim. Já que nenhuma das três expressões que formam o laço **for** é obrigatória, você pode fazer um laço sem fim deixando a expressão condicional vazia, como aqui:

```
for (;;) printf("Esse laço será executado para sempre.\n");
```

Você pode ter uma inicialização e uma expressão de incremento, mas programadores C usam mais usualmente a construção **for(;;)** para significar um laço infinito.

De fato, a construção **for(;;)** não garante um laço infinito porque o comando **break** de C, encontrado em qualquer lugar dentro do corpo de um laço, provoca um término imediato (**break** será discutido mais adiante neste capítulo). O controle do programa, então, continua no código seguinte ao laço, como mostrado aqui:

```
ch='\0';

for( ; ; ) {
  ch = getchar();  /* obtém um caractere */
  if(ch=='A') break; /* sai do laço */
}

printf("você digitou um A");
```


Esse laço será executado até que o usuário digite um A.

Laços for sem Corpos

Como definido pela sintaxe de C, um comando pode ser vazio. Isso significa que o corpo do laço **for** (ou qualquer outro laço) também pode ser vazio. Você pode usar esse fato para aumentar a eficiência de certos algoritmos e para criar laços para atraso de tempo.

Remover espaços de uma "stream" de entrada é uma tarefa comum de programação. Por exemplo, um programa de banco de dados pode permitir uma requisição do tipo "mostre todos os saldos menores que 400". O banco de dados precisa ser alimentado com cada palavra separadamente, sem espaços. Isto é, o processador de entrada do banco de dados reconhece "**show**" mas não "**show**". O seguinte laço remove os primeiros espaços da stream apontada por **str**.

```
for( ; *str == ' '; str++) ;
```

Como você pode ver, esse laço não tem corpo — e nem precisa de um.

Os laços para *atraso de tempo* são muito usados em programas. O seguinte código mostra como criar um usando **for**:

```
for(t=0; t<ALGUM_VALOR; t++) ;
```

O Laço while

O segundo laço disponível em C é o laço **while**. A sua forma geral é

while(*condição*) *comando*;

onde *comando* é um comando vazio, um comando simples ou um bloco de comandos. A *condição* pode ser qualquer expressão, e verdadeiro é qualquer valor não-zero. O laço se repete quando a condição for verdadeira. Quando a condição for falsa, o controle do programa passa para a linha após o código do laço.

O seguinte exemplo mostra uma rotina de entrada pelo teclado, que simplesmente se repete até que o usuário digite A:

```
wait_for_char(void)
```

```
{
  char ch;

  ch = '\0';  /* inicializa ch */
  while(ch != 'A')  ch = getchar();
  return ch;
}
```

Primeiro, **ch** é inicializado com nulo. Como uma variável local, seu valor não é conhecido quando **wait_for_char()** é executado. O laço **while** verifica se **ch** não é igual a **A**. Como **ch** foi inicializado com nulo, o teste é verdadeiro e o laço começa. Cada vez que o usuário pressiona uma tecla, o teste é executado novamente. Uma vez digitado **A**, a condição se torna falsa, porque **ch** fica igual a **A**, e o laço termina.

Como os laços **for**, os laços **while** verificam a condição de teste no início do laço, o que significa que o código do laço pode não ser executado. Isso elimina a necessidade de se efetuar um teste condicional antes do laço. A função **pad()** fornece uma boa ilustração disso. Ela adiciona espaços ao final de uma string até um comprimento predefinido. Se a string já é do tamanho desejado, nenhum espaço é adicionado.

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

void pad(char *s, int length);
void main(void)
{
  char str[80];

  strcpy(str, "isto é um teste");
  pad(str, 40);
  printf("%d", strlen(str));
}

/* Acrescenta espaços ao final da string. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s);  /* encontra o comprimento */

  while(l<length) {
    s[l] = ' ';  /* insere um espaço */
```



```
    l++;
  }
  s[l] = '\0';  /* strings precisam terminar com um nulo */
}
```

Os dois argumentos de **pad()** são **s**, um ponteiro para a string a ser ajustada, e **length**, o número de caracteres que **s** deve ter. Se a string **s** já é igual ou maior que **length**, o código dentro do laço **while** não será executado. Se **s** é menor que **length**, **pad()** adiciona o número necessário de espaços. A função **strlen()**, parte integrante da biblioteca padrão, devolve o tamanho de uma string.

Se diversas condições forem necessárias para terminar um laço **while**, normalmente uma única variável forma a expressão condicional. O valor dessa variável será alterado em vários pontos internos ao laço. Neste exemplo

```
void func1(void)
{
  int working;

  working = 1;  /* i.e., verdadeiro */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

qualquer uma das três rotinas pode retornar falso e fazer com que o laço termine.

Não é necessário haver nenhum comando no corpo do laço **while**. Por exemplo,

```
while((ch=getchar()) != 'A') ;
```

será simplesmente executado até que o usuário digite **A**. Se você se sente desconfortável colocando a atribuição dentro da expressão condicional do **while**, lembre-se de que o sinal de igual é apenas um operador que calcula o valor do operando do lado direito.

O Laço do-while

Ao contrário dos laços **for** e **while**, que testam a condição do laço no começo, o laço **do-while** verifica a condição ao final do laço. Isso significa que um laço **do-while** sempre será executado ao menos uma vez. A forma geral do laço **do-while** é

```
do{
    comando;
} while(condição);
```

Embora as chaves não sejam necessárias quando apenas um comando está presente, elas são geralmente usadas para evitar confusão (para você, não para o compilador) com o **while**. O laço **do-while** repete até que a *condição* se torne falsa.

O seguinte laço **do-while** lerá números do teclado até que encontre um número menor ou igual a 100.

```
do {
  scanf("%d", &num);
} while(num > 100);
```

Talvez o uso mais comum do laço **do-while** seja em uma rotina de seleção por menu. Quando o usuário entra com uma resposta válida, ela é retornada como o valor da função. Respostas inválidas provocam uma repetição do laço. O seguinte código mostra uma versão melhorada do menu do verificador de ortografia que foi desenvolvido anteriormente neste capítulo:

```
void menu(void)
{
  char ch;

  printf("1. Verificar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("     Digite sua escolha:   ");

  do {
    ch = getchar();  /* lê do teclado a seleção */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```


Aqui, o laço **do-while** é uma boa escolha, porque você sempre deseja que uma função do menu execute ao menos uma vez. Depois que as opções forem mostradas, o programa será executado até que uma opção válida seja selecionada.

Comandos de Desvio

C tem quatro comandos que realizam um desvio incondicional: **return**, **goto**, **break** e **continue**. Destes, você pode usar **return** e **goto** em qualquer lugar em seu programa. Você pode usar os comandos **break** e **continue** em conjunto com qualquer dos comandos de laço. Como discutido anteriormente neste capítulo, você também pode usar o **break** com **switch**.

O Comando return

O comando **return** é usado para retornar de uma função. Ele é um comando de desvio porque faz com que a execução retorne (salte de volta) ao ponto em que a chamada à função foi feita. Se **return** tem um valor associado a ele, esse valor é o valor de retorno da função. Se nenhum valor de retorno é especificado, assume-se que apenas lixo é retornado. (Alguns compiladores C irão automaticamente retornar 0 se nenhum valor for especificado, mas não conte com isso.)

A forma geral do comando **return** é

return *expressão*;

Lembre-se de que a *expressão* é opcional. Entretanto, se estiver presente, ela se tornará o valor da função.

Você pode usar quantos comandos **return** quiser dentro de uma função. Contudo, a função parará de executar tão logo ela encontre o primeiro **return**. A } que finaliza uma função também faz com que a função retorne. É o mesmo que um **return** sem nenhum valor especificado.

Uma função declarada como **void** não pode ter um comando **return** que especifique um valor. (Como uma função void não retorna valor, não tem sentido que o comando **return** especifique um valor nas funções **void**).

Veja o Capítulo 6 para mais informações sobre **return**.

O Comando goto

Uma vez que C tem um rico conjunto de estruturas de controle e permite um controle adicional usando **break** e **continue**, há pouca necessidade do **goto**. A grande preocupação da maioria dos programadores sobre o **goto** é a sua tendência de tornar os programas ilegíveis. Entretanto, embora o **goto** tenha sido desencorajado alguns anos atrás, ele tem recentemente polido um pouco a sua imagem manchada. Não há nenhuma situação na programação que necessite do **goto**. Apesar disso, contudo, **goto** é uma conveniência que, se usada prudentemente, pode ser uma vantagem em certas situações na programação. Sendo assim, **goto** não é usado fora desta seção.

O comando **goto** requer um rótulo para sua operação. (Um rótulo é um identificador válido em C seguido por dois-pontos.) O padrão ANSI refere-se a esse tipo de construção como uma sentença de rótulo. Além disso, o rótulo tem de estar na mesma função do **goto** que o usa — você não pode efetuar desvios entre funções. A forma geral do comando **goto** é

goto *rótulo*;
.
.
.
rótulo:

onde *rótulo* é qualquer rótulo válido existente antes ou depois do **goto**. Por exemplo, você poderia criar um laço de 1 até 100 usando **goto** e um rótulo, como mostrado aqui:

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

O Comando break

O comando **break** tem dois usos. Você pode usá-lo para terminar um **case** em um comando **switch** (abordado anteriormente na seção sobre o **switch**, neste capítulo). Você também pode usá-lo para forçar uma terminação imediata de um laço, evitando o teste condicional normal do laço.

Quando o comando **break** é encontrado dentro de um laço, o laço é imediatamente terminado e o controle do programa retorna no comando seguinte ao laço. Por exemplo,

```
#include <stdio.h>

void main(void)
{
  int t;

  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }
}
```

escreve os números de 1 até 10 na tela. Então, o laço termina porque o **break** provoca uma saída imediata do laço, desrespeitando o teste condicional **t<100**.

Os programadores geralmente usam o comando **break** em laços em que uma condição especial pode provocar uma terminação imediata. Por exemplo, aqui, o pressionamento de uma tecla pode parar a execução da função **look_up()**:

```
look_up(char *name)
{
  do {
    /* procura nomes ... */
    if(kbhit()) break;
  } while(!found);
  /* processa a concordância */
}
```

A função **kbhit()** retorna 0 se você não pressionar uma tecla. Caso contrário, ela retorna um valor não-zero. Devido às grandes diferenças entre ambientes computacionais, o padrão ANSI não define **kbhit()**, mas é quase certo que você a tem (ou alguma com um nome um pouco diferente) fornecida com seu compilador.

Um comando **break** provoca uma saída apenas do laço mais interno. Por exemplo,

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

escreve os números de 1 a 10 na tela 100 vezes. Cada vez que o compilador encontra **break**, transfere o controle de volta para a repetição **for** mais externa.

Um **break** usado em um comando **switch** somente afetará esse **switch**. Ele não afeta qualquer repetição que contenha o **switch**.

A Função exit()

Da mesma forma que você pode sair de um laço, pode sair de um programa usando a função **exit()** da biblioteca padrão. Essa função provoca uma terminação imediata do programa inteiro, forçando um retorno ao sistema operacional. Com efeito, a função **exit()** age como se ela estivesse finalizando o programa inteiro.

A forma geral da função **exit()** é

void exit(int *código_de_retorno*);

O valor de *código_de_retorno* é retornado ao processo chamador, que é normalmente o sistema operacional. O zero é geralmente usado como um código de retorno que indica uma terminação normal do programa. Outros argumentos são usados para indicar algum tipo de erro.

Programadores geralmente usam **exit()** quando uma condição mandatória para a execução do programa não é satisfeita. Por exemplo, imagine um jogo para computador que queira uma placa gráfica colorida. A função **main()** desse jogo poderia se parecer com isto:

```
void main(void)
{
  if(!virtual_graphics()) exit(1);
  play();
}
```

onde **virtual_graphics()** é uma função definida pelo usuário que retorna verdadeiro se a placa colorida está presente. Se a placa não está no sistema, **virtual_graphics()** retorna falso e o programa termina.

Como um outro exemplo, essa versão de **menu()** usa **exit()** para abandonar o programa e retornar ao sistema operacional:

```
void menu(void)
{
  char ch;

  printf("1. Verificar Ortografia\n");
  printf("2. Corrigir Erros de Ortografia\n");
  printf("3. Mostrar Erros de Ortografia\n");
  printf("4. Abandonar\n");
  printf("     Digite sua escolha:   ");

  do {
    ch=getchar();  /* lê do teclado a seleção */
      switch(ch) {
         case '1':
           check_spelling();
           break;
         case '2':
           correct_errors();
           break;
         case '3':
           display_errors();
           break;
         case '4':
           exit(0);  /* retorna ao OS */
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

O Comando continue

O comando **continue** trabalha de uma forma um pouco parecida com a do comando **break**. Em vez de forçar a terminação, porém, **continue** força que ocorra a próxima iteração do laço, pulando qualquer código intermediário. Para o laço **for**, **continue** faz com que o teste condicional e a porção de incremento do laço sejam executados. Para os laços **while** e **do-while**, o controle do programa passa para o teste condicional. Por exemplo, o seguinte programa conta o número de espaços contidos em uma string inserida pelo usuário:

```
/* Conta espaços */
#include <stdio.h>
```

```
void main(void)
{
  char s[80], *str;
  int space;

  printf("Digite uma string: ");
  gets(s);
  str = s;

  for(space=0; *str; str++) {
    if(*str !=' ') continue;
    space++;
  }
  printf("%d espaços\n", space);
}
```

Cada caractere é testado para ver se é um espaço. Se não é, o comando **continue** força o **for** a iterar novamente. Se o caractere é um espaço, **space** é incrementada.

O seguinte exemplo mostra que você pode usar **continue** para apressar a saída de um laço, forçando o teste condicional a ser realizado mais cedo.

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1);  /* desloca o alfabeto uma posição */
  }
}
```

Esta função codifica uma mensagem deslocando todos os caracteres uma letra acima. Por exemplo, um **A** se tornaria um **B**. A função terminará quando um **$** for digitado. Depois que um **$** for inserido, nenhuma saída adicional ocorrerá porque o teste condicional, trazido a efeito pelo **continue**, encontrará **done** como sendo verdadeiro e provocará a saída do laço.

Comandos de Expressões

O Capítulo 2 abrange completamente as expressões de C. Porém, alguns pontos especiais são mencionados aqui. Lembre-se de que um comando de expressão é simplesmente uma expressão válida em C seguida por um ponto-e-vírgula, como em

```
func();  /* uma chamada a uma função */
a = b+c; /* um comando de atribuição */
b+f();   /* um comando válido, que não faz nada */
;        /* um comando vazio */
```

O primeiro comando de expressão executa uma chamada a uma função. O segundo é uma atribuição. O terceiro elemento é estranho, mas é avaliado pelo compilador C, porque a função **f()** pode realizar alguma tarefa necessária. O último exemplo mostra que C permite que um comando seja vazio (algumas vezes chamado de *comando nulo*).

Blocos de Comandos

Blocos de comandos são simplesmente grupos de comandos relacionados que são tratados como uma unidade. Os comandos que constituem um bloco estão logicamente conectados. Um bloco começa com um { e termina com um } correspondente. Programadores normalmente usam blocos de comandos para criar um objeto multicomandos para algum outro comando, como o **if**. No entanto, você pode pôr um bloco de comandos em qualquer lugar onde seja possível a colocação de um outro comando qualquer. Por exemplo, este é um código em C perfeitamente válido (embora não usual):

```
#include <stdio.h>

void main(void)
{
  int i;

  {  /* um bloco de comandos */
     i = 120;
     printf("%d", i);
  }
}
```

MAKRON Books

4

Matrizes e Strings

Uma *matriz* é uma coleção de variáveis do mesmo tipo que é referenciada por um nome comum. Um elemento específico em uma matriz é acessado por meio de um índice. Em C, todas as matrizes consistem em posições contíguas na memória. O endereço mais baixo corresponde ao primeiro elemento e o mais alto, ao último elemento. Matrizes podem ter de uma a várias dimensões. A matriz mais comum em C é a de string, que é simplesmente uma matriz de caracteres terminada por um nulo. Essa abordagem a strings dá a C maior poder e eficiência que às outras linguagens.

Em C, matrizes e ponteiros estão intimamente relacionados; uma discussão sobre um deles normalmente refere-se ao outro. Este capítulo focaliza matrizes, enquanto o Capítulo 5 examina mais profundamente os ponteiros. Você deve ler ambos para entender completamente essas construções importantes de C.

Matrizes Unidimensionais

A forma geral para declarar uma matriz unidimensional é

tipo nome_var[tamanho];

Como outras variáveis, as matrizes devem ser explicitamente declaradas para que o compilador possa alocar espaço para elas na memória. Aqui, *tipo* declara o tipo de base da matriz, que é o tipo de cada elemento da matriz; *tamanho* define quantos elementos a matriz irá guardar. Por exemplo, para declarar um matriz de 100 elementos, chamada **balance**, e do tipo **double**, use este comando:


```
double balance[100];
```

Em C, toda matriz tem 0 como o índice do seu primeiro elemento. Portanto, quando você escreve

```
char p[10];
```

você está declarando uma matriz de caracteres que tem dez elementos, **p[0]** até **p[9]**. Por exemplo, o seguinte programa carrega uma matriz inteira com os números de 0 a 99:

```
void main(void)
{
  int x[100];  /* isto reserva 100 elementos inteiros */
  int t;

  for(t=0; t<100; ++t) x[t] = t;
}
```

A quantidade de armazenamento necessário para guardar uma matriz está diretamente relacionada com seu tamanho e seu tipo. Para uma matriz unidimensional, o tamanho total em bytes é calculado como mostrado aqui:

total em bytes = sizeof(tipo) * tamanho da matriz

C não tem verificação de limites em matrizes. Você poderia ultrapassar o fim de uma matriz e escrever nos dados de alguma outra variável ou mesmo no código do programa. Como programador, é seu trabalho prover verificação dos limites onde for necessário. Por exemplo, este código compilará sem erros, mas é incorreto, porque o laço **for** fará com que a matriz **count** ultrapasse seus limites.

```
int count[10], i;

/* isto faz com que count seja ultrapassada */
for(i=0; i<100; i++) count[i]= i;
```

Matrizes unidimensionais são, essencialmente, listas de informações do mesmo tipo, que são armazenadas em posições contíguas da memória em uma ordem de índice. Por exemplo, a Figura 4.1 mostra como a matriz **a** apareceria na memória se ela começasse na posição de memória 1000 e fosse declarada como mostrado aqui:

```
char a[7];
```

Elemento	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Endereço	1000	1001	1002	1003	1004	1005	1006

Figura 4.1 Uma matriz de sete elementos começando na posição 1000.

Gerando um Ponteiro para uma Matriz

Você pode gerar um ponteiro para o primeiro elemento de uma matriz simplesmente especificando o nome da matriz, sem nenhum índice. Por exemplo, dado

```
int sample[10];
```

você pode gerar um ponteiro para o primeiro elemento simplesmente usando o nome **sample**. Por exemplo, o seguinte fragmento atribui a **p** o endereço do primeiro elemento de **sample**:

```
int *p;
int sample[10];

p = sample;
```

Você também pode especificar o endereço do primeiro elemento de uma matriz usando o operador **&**. Por exemplo, **sample** e **&sample[0]** produzem os mesmos resultados. Porém, em códigos C escritos profissionalmente, você quase nunca verá algo como **&sample[0]**.

Passando Matrizes Unidimensionais para Funções

Em C, você não pode passar uma matriz inteira como um argumento para uma função. Você pode, porém, passar um ponteiro para uma matriz para uma função, especificando o nome da matriz sem um índice. Por exemplo, o seguinte fragmento de programa passa o endereço de **i** para **func1()**:

```
void main(void)
{
  int i[10];

  func1(i);
  .
  .
  .
}
```


Se uma função recebe uma matriz unidimensional, você pode declarar o parâmetro formal em uma entre três formas: como um ponteiro, como uma matriz dimensionada ou como uma matriz não-dimensionada. Por exemplo, para receber **i**, uma função chamada **func1()** pode ser declarada como

```
void func1(int *x) /* ponteiro */

{
  .
  .
  .
}
```

ou

```
void func1(int x[10]) /* matriz dimensionada */

{
  .
  .
  .
}
```

ou finalmente como

```
void func1(int x[]) /* matriz não-dimensionada */

{
  .
  .
  .
}
```

Todos os três métodos de declaração produzem resultados idênticos, porque cada um diz ao compilador que um ponteiro inteiro vai ser recebido. A primeira declaração usa, de fato, um ponteiro. A segunda emprega a declaração de matriz padrão. Na última versão, uma versão modificada de uma declaração de matriz simplesmente especifica que uma matriz do tipo **int**, de algum tamanho, será recebida. Como você pode ver, o comprimento da matriz não importa à função, porque C não realiza verificação de limites. De fato, até onde diz respeito ao compilador,

```
void func1(int x[32])
{
     .
     .
     .
}
```

também funciona, porque o compilador C gera um código que instrui **func1()** a receber um ponteiro — ele não cria realmente uma matriz de 32 elementos.

Strings

O uso mais comum de matrizes unidimensionais é como string de caracteres. Lembre-se de que, em C, uma string é definida como uma matriz de caracteres que é terminada por um nulo. Um nulo é especificado como '\0' e geralmente é zero. Por essa razão, você precisa declarar matrizes de caracteres como sendo um caractere mais longo que a maior string que elas devem guardar. Por exemplo, para declarar uma matriz **str** que guarda uma string de 10 caracteres, você escreveria

```
char str[11];
```

Isso reserva espaço para o nulo no final da string.

Embora C não tenha o tipo de dado string, ela permite constantes string. Uma *constante string* é uma lista de caracteres entre aspas. Por exemplo,

"alo aqui"

Você não precisa adicionar o nulo no final das constantes string manualmente — o compilador C faz isso por você automaticamente.

C suporta uma ampla gama de funções de manipulação de strings. As mais comuns são:

Nome	Função
strcpy(s1, s2)	Copia s2 em s1.
strcat(s1, s2)	Concatena s2 ao final de s1.
strlen(s1)	Retorna o tamanho de s1.
strcmp(s1, s2)	Retorna 0 se s1 e s2 são iguais; menor que 0 se s1<s2; maior que 0 se s1>s2.
strchr(s1, ch)	Retorna um ponteiro para a primeira ocorrência de ch em s1.
strstr(s1, s2)	Retorna um ponteiro para a primeira ocorrência de s2 em s1.

Essas funções usam o cabeçalho padrão **STRING.H**. (Essas e outras funções de string são discutidas em detalhes na Parte 2.) O seguinte programa ilustra o uso dessas funções de string:

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

void main(void)
{
  char s1[80], s2[80];

  gets(s1);

  gets(s2);

  printf("comprimentos: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf ("As strings são iguais\n");
  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "Isso é um teste.\n");
  printf(s1);
  if(strchr("alo", 'o')) printf("o está em alo\n");
  if(strstr("ola aqui", "ola")) printf("ola encontrado");
}
```

Se você rodar esse programa e digitar as strings "**alo**" e "**alo**", a saída será

```
comprimentos: 33
As string são iguais
aloalo
Isso é um teste.
o está em alo
ola encontrado
```

Lembre-se de que **strcmp()** retorna falso se as strings são iguais. Assegure-se de usar o operador ! para reverter a condição, como mostrado, se você estiver testando igualdade.

Matrizes Bidimensionais

C suporta matrizes multidimensionais. A forma mais simples de matriz multidimensional é a matriz bidimensional — uma matriz de matrizes unidimensionais. Para declarar uma matriz bidimensional de inteiros **d** de tamanho 10,20, você escreveria

```
int d[10][20];
```

Preste bastante atenção à declaração. Muitas linguagens de computador usam vírgulas para separar as dimensões da matriz; C, em contraste, coloca cada dimensão no seu próprio conjunto de colchetes.

Similarmente, para acessar o ponto 1,2 da matriz **d**, você usaria

```
d[1][2];
```

O seguinte exemplo carrega uma matriz bidimensional com os números de 1 a 12 e escreve-os linha por linha.

```
#include <stdio.h>

void main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

/* agora escreva-os */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }
}
```


Neste exemplo, **num[0][0]** tem o valor 1, **num[0][1]**, o valor 2, **num[0][2]**, o valor 3 e assim por diante. O valor de **num[2][3]** será 12. Você pode visualizar a matriz **num** como mostrada aqui:

num [t] [i]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Matrizes bidimensionais são armazenadas em uma matriz linha-coluna, onde o primeiro índice indica a linha e o segundo, a coluna. Isso significa que o índice mais à direita varia mais rapidamente do que o mais à esquerda quando acessamos os elementos da matriz na ordem em que eles estão realmente armazenados na memória. Veja a Figura 4.2 para uma representação gráfica de uma matriz bidimensional na memória. Você pode pensar no primeiro índice como um "ponteiro" para a linha correta.

No caso de uma matriz bidimensional, a seguinte fórmula fornece o número de bytes de memória necessários para armazená-la:

bytes = tamanho do 1º índice * tamanho do 2º índice * sizeof(tipo base)

Portanto, assumindo inteiros de dois bytes, uma matriz de inteiros com dimensões 10,5 teria

10 * 5 * 2

ou 100 bytes alocados.

Quando uma matriz bidimensional é usada como um argumento para uma função, apenas um ponteiro para o primeiro elemento é realmente passado. Porém, uma função que recebe uma matriz bidimensional como um parâmetro deve definir pelo menos o comprimento da segunda dimensão. Isso ocorre porque o compilador C precisa saber o comprimento de cada linha para indexar a matriz corretamente. Por exemplo, uma função que recebe uma matriz bidimensional de inteiros com dimensões 10,10 é declarada desta forma:

```
void func1(int x[][10])
{
     .
     .
     .
}
```

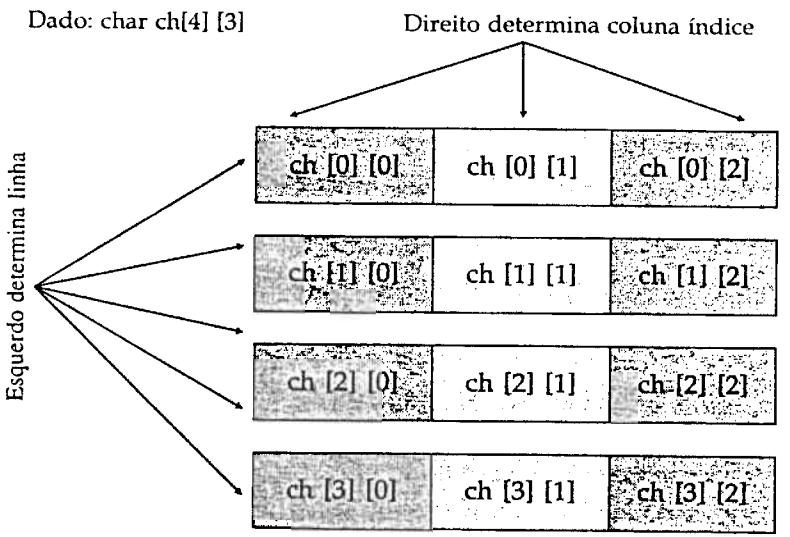


Figura 4.2 Uma matriz bidimensional na memória.

Você pode especificar a primeira dimensão, se quiser, mas não é necessário. O compilador C precisa saber a segunda dimensão para trabalhar em sentenças como

```
x[2][4]
```

dentro da função. Se o comprimento das linhas não é conhecido, o compilador não pode determinar onde a terceira linha começa.

O seguinte programa usa uma matriz bidimensional para armazenar as notas numéricas de cada aluno de uma sala de aula. O programa assume que o professor tem três turmas e um máximo de 30 alunos por turma. Note a maneira como a matriz **grade** é acessada em cada uma das funções.

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

/* Um banco de dados simples para notas de alunos. */

#define CLASSES  3
```



```
#define GRADES  30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

void main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(D)igitar notas\n");
      printf("(M)ostrar notas\n");
      printf("(S)air\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='D' && ch!='M' && ch!='S');

    switch(ch) {
      case 'D':
        enter_grades();
        break;
      case 'M':
        disp_grades(grade);
        break;
      case 'S':
        exit(0);
    }
  }
}

/* Digita a nota dos alunos. */
void enter_grades(void)
{
  int t, i;

  for(t=0; t<CLASSES; t++) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade [t][i]= get_grade(i);
  }
}
```

```
/* Lê uma nota. */
get_grade(int num)
{
  char s[80];

  printf("Digite a nota do aluno # %d:\n",num+1);
  gets(s);
  return(atoi(s));
}

/* Mostra as notas. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Turma # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Aluno #%d é %d\n", i+1, g[t][i]);
  }
}
```

Matrizes de Strings

Não é incomum, em programação, usar uma matriz de strings. Por exemplo, o processador de entrada de um banco de dados pode verificar os comandos do usuário com base em uma matriz de comandos válidos. Para criar uma matriz de strings, use uma matriz bidimensional de caracteres. O tamanho do índice esquerdo indica o número de strings e o tamanho do índice do lado direito especifica o comprimento máximo de cada string. O código seguinte declara uma matriz de 30 strings, cada qual com um comprimento máximo de 79 caracteres:

```
char str_array[30][80];
```

É fácil acessar uma string individual: você simplesmente especifica apenas o índice esquerdo. Por exemplo, o seguinte comando chama **gets()** com a terceira string em **str_array**.

```
gets(str_array[2]);
```

O comando anterior é funcionalmente equivalente a


```
gets(&str_array[2][0]);
```

mas a primeira das duas formas é muito mais comum em códigos C escritos profissionalmente.

Para entender melhor como matrizes de string funcionam, estude o programa seguinte, que usa uma matriz de string como base para um editor de texto muito simples:

```
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

/* Um editor de texto muito simples. */
void main(void)


{
  register int t, i, j;

  printf("Digite uma linha vazia para sair.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* sai com linha em branco */


}


  for(i=0; i<t; i++) {
   for(j=0; text[i][j]; j++) putchar(text[i][j]);
   putchar('\n');
  }
}
```

Este programa recebe linhas de texto até que uma linha em branco seja inserida. Então, ele mostra novamente cada linha, um caractere por vez.

Matrizes Multidimensionais

C permite matrizes com mais de duas dimensões. O limite exato, se existe, é determinado por seu compilador. A forma geral da declaração de uma matriz multidimensional é

tipo nome[*Tamanho1*][*Tamanho2*][*Tamanho3*]...[*TamanhoN*];

Matrizes de três ou mais dimensões não são freqüentemente usadas devido à quantidade de memória de que elas necessitam. Por exemplo, uma matriz de quatro dimensões do tipo caractere e com tamanhos 10,6,9,4 requer

10 * 6 * 9 * 4

ou 2.160 bytes. Se a matriz guardasse inteiros de 2 bytes, 4.320 bytes seriam necessários. Se a matriz guardasse **double** (assumindo 8 bytes por **double**), 17.280 bytes seriam necessários. O armazenamento necessário cresce exponencialmente com o número de dimensões. Grandes matrizes multidimensionais são geralmente alocadas dinamicamente, uma parte por vez, com as funções de alocação dinâmica de C e ponteiros. Essa abordagem é chamada de *matriz esparsa* e é discutida no Capítulo 21.

Em matrizes multidimensionais, toma-se tempo do computador para calcular cada índice. Isso significa que acessar um elemento em uma matriz multidimensional é mais lento do que acessar um elemento em uma matriz unidimensional.

Quando passar matrizes multidimensionais para funções, você deve declarar todas menos a primeira dimensão. Por exemplo, se você declarar a matriz **m** como

```
int m[4][3][6][5];
```

uma função, **func1()**, que recebe **m**, se pareceria com isto:

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

Obviamente, você pode incluir a primeira dimensão, se quiser.

Indexando Ponteiros

Em C, ponteiros e matrizes estão intimamente relacionados. Como você sabe, um nome de matriz sem um índice é um ponteiro para o primeiro elemento da matriz. Por exemplo, considere a seguinte matriz.

```
char p[10];
```

As seguintes sentenças são iguais:

```
p
&p[0]
```

Colocando de outra forma,

```
p == &p[0]
```

é avaliado como verdadeiro, porque o endereço do primeiro elemento de uma matriz é o mesmo que o da matriz.

Reciprocamente, qualquer ponteiro pode ser indexado como se uma matriz fosse declarada. Por exemplo, considere este fragmento de programa:

```
int *p, i[10];
p = i;
p[5] = 100; /* atribui usando o índice */
*(p+5) = 100; /*atribui usando aritmética de ponteiros */
```

Os dois comandos de atribuição colocam o valor 100 no sexto elemento de **i**. O primeiro elemento indexa **p**; o segundo usa aritmética de ponteiro. De qualquer forma, o resultado é o mesmo. (O Capítulo 5 discute ponteiros e aritmética de ponteiros.)

Esse processo também pode ser aplicado a matrizes de duas ou mais dimensões. Por exemplo, assumindo que **a** é uma matriz de inteiros 10 por 10, estas duas sentenças são equivalentes:

```
a
&a[0][0]
```

Além disso, o elemento 0,4 de **a** pode ser referenciado de duas formas: por indexação de matriz, **a[0][4]**, ou por ponteiro, ***(a+4)**. Similarmente, o elemento 1,2 é **a[1][2]** ou ***(a+12)**. Em geral, para qualquer matriz bidimensional

a[j][k] é equivalente a *(a+(j*comprimento das linhas)+k)

Às vezes, ponteiros são usados para acessar matrizes porque a aritmética de ponteiros é geralmente mais rápida que a indexação de matrizes.

De certa forma, uma matriz bidimensional é semelhante a uma matriz de ponteiros que apontam para matrizes de linhas. Por isso, usar uma variável de ponteiro separada torna-se uma maneira fácil de utilizar ponteiros para acessar os elementos de uma matriz bidimensional. A função seguinte imprimirá o conteúdo da linha especificada da matriz global inteira **num**:

```
int num[10][10];
  .
  .
  .
void pr_row(int j)
{
  int *p, t;

  p = &num[j][0]; /* obtém o endereço do primeiro
                     elemento na linha j */

  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

Você pode generalizar essa rotina usando como argumentos de chamada a linha, o comprimento das linhas e um ponteiro para o primeiro elemento da matriz, como mostrado aqui:

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;

  p = p + (j * row_dimension);

  for(t=0; t<row_dimension; ++t)
    printf("%d ", *(p+t));
}
```

Matrizes de dimensões maiores que dois podem ser reduzidas de forma semelhante. Por exemplo, uma matriz tridimensional pode ser reduzida a um ponteiro para uma matriz bidimensional, que pode ser reduzida a um ponteiro para uma matriz unidimensional. Genericamente, um matriz *n*-dimensional pode ser reduzida a um ponteiro para uma matriz (*n*-1)-dimensional. Essa nova matriz pode ser reduzida novamente com o mesmo método. O processo termina quando uma matriz unidimensional é produzida.

Inicialização de Matriz

C permite a inicialização de matrizes no momento da declaração. A forma geral de uma inicialização de matriz é semelhante à de outras variáveis, como mostrado aqui:

especificador_de_tipo nome_da_matriz[tamanho1]...[tamanhoN] = {lista_valores};

A *lista_valores* é uma lista separada por vírgulas de constantes cujo tipo é compatível com *especificador_de_tipo*. A primeira constante é colocada na primeira posição, da matriz, a segunda, na segunda posição e assim por diante. Observe o ponto-e-vírgula que segue o }.

No exemplo seguinte, uma matriz inteira de dez elementos é inicializada com os números de 1 a 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Isso significa que **i[0]** terá o valor 1 e **i[9]** terá o valor 10.

Matrizes de caracteres que contêm strings permitem uma inicialização abreviada que toma a forma:

char *nome_da_matriz[tamanho] = "string";*

Por exemplo, este fragmento de código inicializa **str** com a frase "Eu gosto de C".

```
char str[14] = "Eu gosto de C";
```

Isso é o mesmo que escrever

```
char str[14]= {'E','u',' ','g','o','s','t','o',' ','d','e',' '
               ,'C','\0'};
```

Como todas as strings em C terminam com um nulo, você deve ter certeza de que a matriz a ser declarada é longa o bastante para incluir o nulo. Isso explica porque **str** tem comprimento de 14 caracteres, muito embora "Eu gosto de C" tenha apenas 13. Quando você usa uma constante string, o compilador automaticamente fornece o terminador nulo.

Matrizes multidimensionais são inicializadas da mesma forma que matrizes unidimensionais. Por exemplo, o código seguinte inicializa **sqrs** com os números de 1 a 10 e seus quadrados.

```
int sqrs[10][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

Inicialização de Matrizes Não-Dimensionadas

Imagine que você esteja usando inicialização de matrizes para construir uma tabela de mensagens de erro, como mostrado aqui:

```
char e1[17] = "erro de leitura\n";
char e2[17] = "erro de escrita\n";
char e3[29] = "arquivo não pode ser aberto\n";
```

Como você poderia supor, é tedioso contar os caracteres em cada mensagem, manualmente, para determinar a dimensão correta da matriz. Você pode deixar C calcular automaticamente as dimensões da matriz, usando matrizes não dimensionadas. Se, em um comando de inicialização de matriz, o tamanho da matriz não é especificado, o compilador C cria uma matriz grande o bastante para conter todos os inicializadores presentes. Isso é chamado de *matriz não-dimensionada*. Usando essa abordagem, a tabela de mensagens torna-se

```
char e1[] = "Erro de leitura\n";
char e2[] = "Erro de escrita\n";
char e3[] = "Arquivo não pode ser aberto\n";
```

Dadas essas inicializações, este comando

```
printf("%s tem comprimento %d\n", e2, sizeof e2);
```

mostrará

erro de escrita tem comprimento 17

Além de ser menos tediosa, a inicialização de matrizes *não-dimensionadas* permite a você alterar qualquer mensagem sem se preocupar em usar uma matriz de dimensões incorretas.

Inicializações de matrizes não-dimensionadas não estão restritas a matrizes unidimensionais. Para matrizes multidimensionais, você deve especificar todas, exceto a dimensão mais à esquerda, para que o compilador possa indexar a matriz de forma apropriada. Desta forma, você pode construir tabelas de comprimentos variáveis e o compilador aloca automaticamente armazenamento suficiente para guardá-las. Por exemplo, a declaração de **sqrs** como uma matriz não-dimensionada é mostrada aqui:

```
int sqrs[][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

A vantagem dessa declaração sobre a versão que especifica o tamanho é que você pode aumentar ou diminuir a tabela sem alterar as dimensões da matriz.

Um Exemplo com o Jogo-da-Velha

O exemplo que segue ilustra muitas das maneiras pelas quais você pode manipular matrizes em C. Matrizes multidimensionais são comumente usadas para simular as matrizes de jogos de tabuleiro. Essa seção desenvolve um programa simples de jogo da velha.

O computador joga de forma simples. Quando é a vez do computador, ela usa **get_computer_move()** para varrer a matriz, procurando por uma célula desocupada. Quando encontra uma, ele põe um O nesta posição. Se ele não pode encontrar uma célula vazia, ele indica um jogo empatado e termina. A função **get_player_move()** pergunta-lhe onde você quer colocar um X. O canto esquerdo superior é a posição 1,1; o canto direito inferior é a posição 3,3.

A matriz do tabuleiro é inicializada para conter espaços. Isso torna mais fácil apresentar a matriz na tela.

Toda vez que é feito um movimento, o programa chama a função **check()**. Essa função devolve um espaço se ainda não há vencedor, um X se você ganhou ou um O se o computador ganhou. Ela varre as linhas, as colunas e, em seguida, as diagonais, procurando uma que contenha tudo X's ou tudo O's.

A função **disp_matrix()** apresenta o estado atual do jogo. Observe como a inicialização com espaços simplifica essa função.

Todas as rotinas, neste exemplo, acessam a matriz **matrix** de forma diferente. Estude-as para ter certeza de que você compreendeu cada operação com matriz.

```
/* Um exemplo de jogo-da-velha simples. */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3]; /* a matriz do jogo */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void));

void main(void)
{
  char done;

  printf("Este é o jogo-da-velha.\n");
  printf("Você estará jogando contra o computador.\n");

  done = ' ';
  init_matrix();
  do{
    disp_matrix();
    get_player_move();
    done = check();  /* verifica se há vencedor */
    if(done!=' ') break;  /* vencedor! */
    get_computer_move();
    done = check();  /* verifica se há vencedor */
  } while(done== ' ');
  if(done=='X') printf("Você ganhou!\n");
  else printf("Eu ganhei!!!!\n");
  disp_matrix();  /* mostra as posições finais */
}

/* Inicializa a matriz. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
   for(j=0; j<3; j++) matrix[i][j] = ' ';
}

/* Obtém a sua jogada. */
void get_player_move(void)
{
  int x, y;

  printf("Digite as coordenadas para o X: ");
  scanf("%d%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!=' ') {
    printf("Posição inválida, tente novamente. \n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}

/* Obtém uma jogada do computador. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++) {
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }

  if(i*j==9) {
    printf("empate\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
```

```
}

/* Mostra a matriz na tela. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c ¦ %c ¦ %c ", matrix[t][0], matrix[t][1],
         matrix[t][2]);
    if(t!=2) printf("\n---¦---¦---\n");
  }
  printf("\n");
}

/* Verifica se há um vencedor. */
char check(void)
{
  int i;

  for(i=0; i<3; i++) /* verifica as linhas */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matriz[i][2]) return matrix[i][0];

  for(i=0; i<3; i++) /* verifica as colunas */
    if(matrix[0][i]==matrix[1][i] &&
       matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* testa as diagonais */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matriz[2][2])
        return matrix[0][0];
  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matriz[2][0])
        return matrix[0][2];

  return ' ';
}
```

MAKRON Books

5

Ponteiros

O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C. Há três razões para isso: primeiro, ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos; segundo, eles são usados para suportar as rotinas de alocação dinâmica de C, e terceiro, o uso de ponteiros pode aumentar a eficiência de certas rotinas.

Ponteiros são um dos aspectos mais fortes e mais perigosos de C. Por exemplo, ponteiros não-inicializados, ou *ponteiros selvagens*, podem provocar uma quebra do sistema. Talvez pior, é fácil usar ponteiros incorretamente, ocasionando erros que são muito difíceis de encontrar.

O Que São Ponteiros?

Um *ponteiro* é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável na memória. Se uma variável contém o endereço de uma outra, então a primeira variável é dita para *apontar* para a segunda. A Figura 5.1 ilustra essa situação.

Variáveis Ponteiros

Se uma variável irá conter um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste no tipo de base, um * e o nome da variável. A forma geral para declarar uma variável ponteiro é

*tipo *nome;*

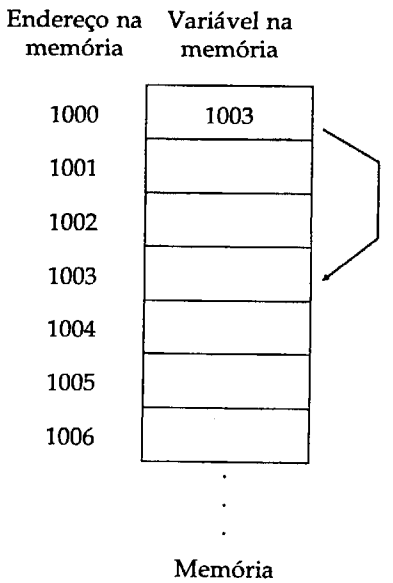


Figura 5.1 Uma variável aponta para outra.

onde *tipo* é qualquer tipo válido em C e *nome* é o nome da variável ponteiro.

O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar. Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. No entanto, toda a aritmética de ponteiros é feita por meio do tipo base, assim, é importante declarar o ponteiro corretamente. (A aritmética de ponteiros é discutida mais adiante neste capítulo.)

Os Operadores de Ponteiros

Existem dois operadores especiais para ponteiros: * e &. O & é um operador unário que devolve o endereço na memória do seu operando. (Um operador unário requer apenas um operando.) Por exemplo,

```
m = &count;
```

coloca em **m** o endereço da memória que contém a variável **count**. Esse endereço é a posição interna ao computador da variável. O endereço não tem relação alguma com o valor de **count**. O operador **&** pode ser imaginado como retornando

"o endereço de". Assim, o comando de atribuição anterior significa "**m** recebe o endereço de **count**".

Para entender melhor a atribuição anterior, assuma que a variável **count** usa a posição de memória 2000 para armazenar seu valor. Assuma também que **count** tem o valor 100. Então, após a atribuição anterior, **m** terá o valor 2000.

O segundo operador de ponteiro, *, é o complemento de &. É um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se **m** contém o endereço da variável **count**,

```
q = *m;
```

coloca o valor de **count** em **q**. Portanto, **q** terá o valor 100 porque 100 estava armazenado na posição 2000, que é o endereço que estava armazenado em **m**. O operador * pode ser imaginado como "no endereço". Nesse caso, o comando anterior significa "**q** recebe o valor que está no endereço **m**".

Alguns iniciantes em C podem confundir-se porque o sinal de multiplicação e o símbolo de "no endereço" são idênticos, como também o AND bit a bit é igual ao símbolo de "endereço de". Esses operadores não têm nenhuma relação um com o outro. Tanto & como * têm uma precedência maior do que todos os operadores aritméticos, exceto o menos unário, com o qual eles se parecem.

As variáveis ponteiros sempre devem apontar para o tipo de dado correto. Por exemplo, quando um ponteiro é declarado como sendo do tipo **int**, o ponteiro assume que qualquer endereço que ele contenha aponta para uma variável inteira. Como C permite a atribuição de qualquer endereço a uma variável ponteiro, o fragmento de código seguinte compila sem nenhuma mensagem de erro (ou apenas uma advertência, dependendo do seu compilador), mas não produz o resultado desejado:

```
void main(void)
{
  float x, y;
  int  *p;

  /* O próximo comando faz com que p (que é ponteiro
     para inteiro) aponte para um float. */
  p = &x;

   /* O próximo comando não funciona como esperado. */
  y = *p;
}
```

Isso não irá atribuir o valor de x a y. Já que p é declarado como um ponteiro para inteiros, apenas dois bytes de informação são transferidos para y, não os 8 bytes que normalmente formam um número em ponto flutuante.

Expressões com Ponteiros

Em geral, expressões envolvendo ponteiros concordam com as mesmas regras de qualquer outra expressão de C. Nesta seção uns poucos aspectos especiais de expressões com ponteiros serão examinados.

Atribuição de Ponteiros

Como é o caso com qualquer variável, um ponteiro pode ser usado no lado direito de um comando de atribuição para passar seu valor para um outro ponteiro. Por exemplo,

```
#include <stdio.h>

void main(void)
{
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf("%p", p2); /* escreve o endereço de x, não seu
                       valor! */
}
```

Agora, tanto **p1** quanto **p2** apontam para **x**. O endereço de **x** é mostrado, usando o modificador de formato de **printf()** **%p**, que faz com que **printf()** apresente um endereço no formato usado pelo computador host.

Aritmética de Ponteiros

Existem apenas duas operações aritméticas que podem ser usadas com ponteiros: adição e subtração. Para entender o que ocorre na aritmética de ponteiros, consideremos **p1** um ponteiro para um inteiro com o valor atual 2000. Assuma, também, que os inteiros são de 2 bytes. Após a expressão


```
p1++;
```

p1 contém 2002, não 2001. Cada vez que **p1** é incrementado, ele aponta para o próximo inteiro. O mesmo é verdade nos decrementos. Por exemplo, assumindo que **p1** tem o valor 2000, a expressão

```
p1--;
```

faz com que **p1** receba o valor 1998.

Generalizando a partir do exemplo anterior, as regras a seguir governam a aritmética de ponteiros. Cada vez que um ponteiro é incrementado, ele aponta para a posição de memória do próximo elemento do seu tipo base. Cada vez que é decrementado, ele aponta para a posição do elemento anterior. Com ponteiros para caracteres, isso freqüentemente se parece com a aritmética "normal". Contudo, todos os outros ponteiros incrementam ou decrementam pelo tamanho do tipo de dado que eles apontam. Por exemplo, assumindo caracteres de 1 byte e inteiros de 2 bytes, quando um ponteiro para caracteres é incrementado, seu valor aumenta em um. Porém, quando um ponteiro inteiro é incrementado, seu valor aumenta em dois. Isso ocorre porque os ponteiros são incrementados e decrementados relativamente ao tamanho do tipo base de forma que ele sempre aponta para o próximo elemento. De maneira mais geral, toda a aritmética de ponteiros é feita relativamente ao tipo base do ponteiro, para que ele sempre aponte para o elemento do tipo base apropriado. A Figura 5.2 ilustra esse conceito.

Você não está limitado a apenas incremento e decremento. Você pode somar ou subtrair inteiros de ponteiros. A expressão

```
p1 = p1 + 12;
```

faz **p1** apontar para o décimo segundo elemento do tipo **p1** adiante do elemento que ele está atualmente apontando.

Além de adição e subtração entre um ponteiro e um inteiro, nenhuma outra operação aritmética pode ser efetuada com ponteiros. Especificamente, você não pode multiplicar ou dividir ponteiros; não pode aplicar os operadores de deslocamento e de mascaramento bit a bit com ponteiros; e não pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.

```
char *ch=3000;
int *i=3000;
```

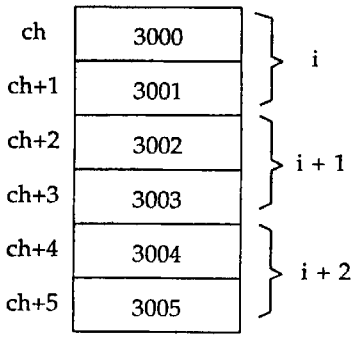


Figura 5.2 Toda aritmética de ponteiros é relativa a seu tipo base.

Comparação de Ponteiros

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros **p** e **q**, o fragmento de código seguinte é perfeitamente válido.

```
if(p<q) printf("p aponta para uma memória mais baixa que q\n");
```

Geralmente, comparações de ponteiros são usadas quando dois ou mais ponteiros apontam para um objeto comum. Como exemplo, um par de rotinas de pilha são desenvolvidas de forma a guardar valores inteiros. Uma pilha é uma lista em que o primeiro acesso a entrar é o último a sair. É freqüentemente comparada a uma pilha de pratos em uma mesa — o primeiro prato colocado é o último a ser usado. Pilhas são freqüentemente usadas em compiladores, interpretadores, planilhas e outros softwares relacionados com o sistema. Para criar uma pilha, são necessárias duas funções: **push()** e **pop()**. A função **push()** coloca os valores na pilha e **pop()** retira-os. Essas rotinas são mostradas aqui com uma função **main()** bem simples para utilizá-las. Se você digitar **0**, um valor será retirado da pilha. Para encerrar o programa, digite **-1**.

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

# define SIZE 50
```



```
void push(int i);
int pop(void);

int *tos, *p1, stack[SIZE];

void main(void)
{
  int value;

  tos = stack; /* faz tos conter o topo da pilha */
  p1 = stack; /* inicializa p1 */

  do {
    printf("Digite o valor: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("valor do topo é %d\n", pop());
  } while(value!=-1);
}

void push(int i)
{
  p1++;
  if(p1==(tos+SIZE)) {
    printf("Estouro da pilha");
    exit(1);
  }
  *p1 = i;
}

pop(void)
{
  if(p1==tos) {
    printf("Estouro da pilha");
    exit(1);
  }
  p1--;
  return *(p1+1);
}
```

Você pode ver que a memória para a pilha é fornecida pela matriz **stack**. O ponteiro **p1** é ajustado para apontar para o primeiro byte em **stack**. A pilha é realmente acessada pela variável **p1**. A variável **tos** contém o endereço do topo da pilha. O valor de **tos** evita que se retirem elementos da pilha vazia. Uma vez

que a pilha tenha sido inicializada, **push()** e **pop()** podem ser usadas como uma pilha de inteiros. Tanto **push()** como **pop()** realizam um teste relacional com o ponteiro **p1** para detectar erros de limite. Em **push()**, **p1** é testado junto ao final da pilha adicionando-se **SIZE** (o tamanho da pilha) a **tos**. Em **pop()**, **p1** é verificado junto a **tos** para assegurar que não se retirem elementos da pilha vazia.

Em **pop()**, os parênteses são necessários no comando **return**. Sem eles, o comando seria

```
return *p1 +1;
```

que retornaria o valor da posição **p1** mais um, não o valor da posição **p1+1**. Você deve usar parênteses para garantir a ordem correta de avaliação quando usar ponteiros.

Ponteiros e Matrizes

Há uma estreita relação entre ponteiros e matrizes. Considere este fragmento de programa:

```
char str[80], *p1;
p1 = str;
```

Aqui, **p1** foi inicializado com o endereço do primeiro elemento da matriz **str**. Para acessar o quinto elemento em **str**, teria de ser escrito

```
str[4]
```

ou

```
*(p1+4)
```

Os dois comandos devolvem o quinto elemento. Lembre-se de que matrizes começam em 0, assim, deve-se usar 4 para indexar **str**. Também deve-se adicionar 4 ao ponteiro **p1** para acessar o quinto elemento porque **p1** aponta atualmente para o primeiro elemento de **str**. (Recorde-se que um nome de uma matriz sem um índice retorna o endereço inicial da matriz, que é o primeiro elemento.)

C fornece dois métodos para acessar elementos de matrizes: aritmética de ponteiros e indexação de matrizes. Aritmética de ponteiros pode ser mais rápida que indexação de matrizes. Já que velocidade é geralmente uma consideração em programação, programadores em C normalmente usam ponteiros para acessar elementos de matrizes.

Essas duas versões de **putstr()** — uma com indexação de matrizes e uma com ponteiros — ilustram como você pode usar ponteiros em lugar de indexação de matrizes. A função **putstr()** escreve uma string no dispositivo de saída padrão.

```
/* Indexa s como uma matriz. */
void puts(char *s)
{
  register int t;
  for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Acessa s como um ponteiro. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

A maioria dos programadores profissionais em C acharia a segunda versão mais fácil de ler e entender. Na realidade, a versão com ponteiros é a forma pela qual rotinas desse tipo são normalmente escritas em C.

Matrizes de Ponteiros

Ponteiros podem ser organizados em matrizes como qualquer outro tipo de dado. A declaração de uma matriz de ponteiros **int**, de tamanho 10, é

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira, chamada **var**, ao terceiro elemento da matriz de ponteiros, deve-se escrever

```
x[2] = &var;
```

Para encontrar o valor de **var**, escreve-se

```
*x[2]
```

Se for necessário passar uma matriz de ponteiros para uma função, pode ser usado o mesmo método que é utilizado para passar outras matrizes — simplesmente chame a função com o nome da matriz sem qualquer índice. Por exemplo, uma função que recebe a matriz **x** se parece com isto:

```
void display_array(int *q[])
{
  int t;

  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```

Lembre-se de que **q** não é um ponteiro para inteiros; **q** é um ponteiro para uma matriz de ponteiros para inteiros. Portanto, é necessário declarar o parâmetro **q** como uma matriz de ponteiros para inteiros, como mostrado no código anterior. Ela não pode ser simplesmente declarada como um ponteiro para inteiros, porque não é isso o que ela é.

Matrizes de ponteiros são usadas normalmente como ponteiros para strings. Você pode criar uma função que exiba uma mensagem de erro, quando é dado seu número de código, como mostrado aqui:

```
void syntax_error(int num)
{
 static char *err[] = {
    "Arquivo não pode ser aberto\n",
    "Erro de leitura\n",
    "Erro de escrita\n",
    "Falha da mídia\n"
  };

  printf("%s", err[num]);
}
```

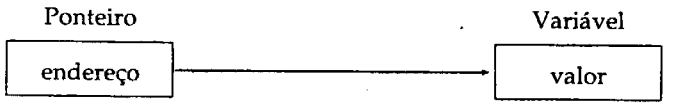
A matriz **err** contém ponteiros para cada string. Como você pode ver, **printf()** dentro de **syntax_error()** é chamada com um ponteiro de caracteres que aponta para uma das várias mensagens de erro indexadas pelo número de erro passado para a função. Por exemplo, se for passado o valor 2, a mensagem **Erro de escrita** é apresentada.

Observe que o argumento da linha de comandos **argv** é uma matriz de ponteiros a caracteres. (Veja o Capítulo 6.)

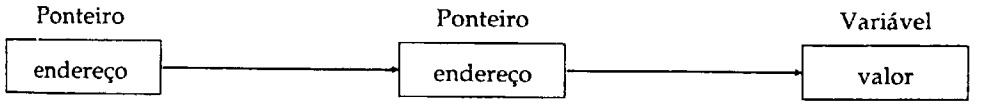
Indireção Múltipla

Você pode ter um ponteiro apontando para outro ponteiro que aponta para o valor final. Essa situação é chamada *indireção múltipla*, ou *ponteiros para ponteiros*.

Ponteiros para ponteiros podem causar confusão. A Figura 5.3 ajuda a esclarecer o conceito de indireção múltipla. Como você pode ver, o valor de um ponteiro normal é o endereço de uma variável que contém o valor desejado. No caso de um ponteiro para um ponteiro, o primeiro ponteiro contém o endereço do segundo, que aponta para a variável que contém o valor desejado.



Indireção Simples



Indireção Múltipla

Figura 5.3 Indireção simples e múltipla.

A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. De fato, indireção excessiva é difícil de seguir e propensa a erros conceituais. (Não confunda indireção múltipla com listas encadeadas.)

NOTA: Não confunda a múltipla indireção com estruturas de dados de alto nível, tais como listas ligadas, que utilizam ponteiros. Estes são dois conceitos fundamentalmente diferentes.

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um * adicional na frente do nome da variável. Por exemplo, a seguinte declaração informa ao compilador que **newbalance** é um ponteiro para um ponteiro do tipo **float**:

```
float **newbalance;
```

É importante entender que **newbalance** não é um ponteiro para um número em ponto flutuante, mas um ponteiro para um ponteiro **float**.

Para acessar o valor final apontado indiretamente por um ponteiro a um ponteiro, você deve utilizar o operador asterisco duas vezes, como neste exemplo:

```
#include <stdio.h>

void main(void)
{
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* imprime o valor de x */
}
```

Aqui, **p** é declarado como um ponteiro para um inteiro e **q**, como um ponteiro para um ponteiro para um inteiro. A chamada a **printf()** imprime **10** na tela.

Inicialização de Ponteiros

Após um ponteiro ser declarado, mas antes que lhe seja atribuído um valor, ele contém um valor desconhecido. Se você tentar usar um ponteiro antes de lhe dar um valor, provavelmente quebrará não apenas seu programa como também o sistema operacional de seu computador — um tipo de erro muito desagradável!

Há uma importante convenção que a maioria dos programadores de C segue quando trabalha com ponteiros: um ponteiro que atualmente não aponta para um local de memória válido recebe o valor nulo (que é zero). Por convenção, qualquer ponteiro que é nulo implica que ele não aponta para nada e não deve ser usado. Porém, apenas o fato de um ponteiro ter um valor nulo não o torna "seguro". Se você usar um ponteiro nulo no lado esquerdo de um comando de atribuição, ainda correrá o risco de quebrar seu programa ou o sistema operacional.

Como um ponteiro nulo é assumido como sendo não usado, você pode utilizar o ponteiro nulo para tornar fáceis de codificar e mais eficientes muitas rotinas. Por exemplo, você poderia usar um ponteiro nulo para marcar o final de uma matriz de ponteiros. Uma rotina que acessa essa matriz sabe que chegará ao final ao encontrar o valor nulo. A função **search()**, mostrada aqui, ilustra esse tipo de abordagem.


```
/* procura um nome */
search(char *p[], char *name)
{
  register int t;

  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;

    return -1; /* não encontrado */
}
```

O laço **for** dentro de **search()** é executado até que seja encontrada uma coincidência ou um ponteiro nulo. Como o final da matriz é marcado com um ponteiro nulo, a condição de controle do laço falha quando ele é atingido.

É uma prática comum entre programadores em C inicializar strings. Você viu um exemplo disso na função **syntax_error()**, na seção "Matrizes de Ponteiros". Uma outra variação no tema de inicialização é o seguinte tipo de declaração de string:

```
char *p = "alo mundo";
```

Como você pode observar, o ponteiro **p** não é uma matriz. A razão pela qual esse tipo de inicialização funciona deve-se à maneira como o compilador opera. Todo compilador C cria o que é chamada de *tabela de string*, que é usada internamente pelo compilador para armazenar as constantes string usadas pelo programa. Assim, o comando de declaração anterior coloca o endereço de **"alo mundo"**, armazenado na tabela de strings no ponteiro **p**. **p** pode ser usado por todo o programa como qualquer outra string. Por exemplo, o programa que segue é perfeitamente válido:

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

char *p = "alo mundo";

void main(void)
{
  register int t;

  /* imprime o conteúdo da string de trás para frente */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
}
```

Ponteiros para Funções

Um recurso confuso, mas poderoso de C, é o *ponteiro para função*. Muito embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. O endereço de uma função é o ponto de entrada da função. Portanto, um ponteiro de função pode ser usado para chamar uma função.

Para entender como funcionam os ponteiros de funções, você deve conhecer um pouco como uma função é compilada e chamada em C. Primeiro, quando cada função é compilada, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido. Quando é feita uma chamada à função, enquanto seu programa está sendo executado, é efetuada uma chamada em linguagem de máquina para esse ponto de entrada. Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, ele pode ser usado para chamar essa função.

O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos. (Isso é semelhante à maneira como o endereço de uma matriz é obtido quando apenas o nome da matriz, sem índices, é usado.) Para ver como isso é feito, estude o programa seguinte, prestando bastante atenção às declarações:

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

void check(char *a, char *b,
           int (*cmp) (const char *, const char *));

void main(void)
{
  char s1[80], s2[80];
  int (*p)();

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);
}

void check(char *a, char *b,
           int (*cmp) (const char *, const char *))
{
```



```
  printf("testando igualdade\n");
  if(!(*cmp)(a, b)) printf("igual");
  else printf("diferente");
}
```

Quando a função **check()** é chamada, dois ponteiros de caractere e um ponteiro para função são passados como parâmetros. Dentro da função **check()**, os argumentos são declarados como ponteiros de caractere e um ponteiro para função. Note como o ponteiro para função é declarado. Você deve usar uma forma semelhante para declarar outros ponteiros para funções, embora o tipo de retorno possa ser diferente. Os parênteses ao redor de ***cmp** são necessários para que o compilador interprete o comando corretamente.

Dentro de **check()**, o comando

```
(*cmp)(a, b)
```

chama **strcmp()**, que é apontado por **cmp** com os argumentos **a** e **b**. Novamente, os parênteses ao redor de ***cmp** são necessários. Esse exemplo também ilustra o método geral a se usar com um ponteiro para função para chamar a função que ele aponta.

Observe que você pode chamar **check()** usando **strcmp()** diretamente, como mostrado aqui:

```
check(s1, s2, strcmp);
```

Isso elimina a necessidade de um ponteiro adicional.

Você pode estar-se perguntando por que alguém escreveria um programa dessa forma. Obviamente, nesse exemplo, nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos ou manter uma matriz de funções. Por exemplo, quando um compilador é escrito, o analisador (*parser*, a parte que avalia expressões) geralmente faz chamadas a funções para várias rotinas de suporte — por exemplo, as funções seno, co-seno e tangente. Em vez de ter um grande comando **switch** com todas essas funções listadas, uma matriz de ponteiros para funções pode ser usada com a função apropriada sendo selecionada pelo seu índice. Você pode sentir a essência desse tipo de uso estudando uma versão expandida do exemplo anterior. Nesse programa, **check()** pode testar igualdade numérica ou alfabética simplesmente chamando-a com uma função de comparação diferente.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include "string.h"

void check(char *a, char *b,
           int (*cmp)  (const char *, const char*));
int numcmp(const char *a, const char *b);

void main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  if(isalpha(*s1))
        check(s1, s2, strcmp);
  else
        check(s1, s2, numcmp);
}

void check(char *a, char *b,
           int (*cmp)  (const char *, const char*))
{
  printf("testando igualdade\n");
  if(!(*cmp)(a, b)) printf("igual");
  else printf("diferente");
}

numcmp(const char *a, const char *b)
{
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

As Funções de Alocação Dinâmica em C

Ponteiros fornecem o suporte necessário para o poderoso sistema de alocação dinâmica de C. *Alocação dinâmica* é o meio pelo qual um programa pode obter memória enquanto está em execução. Como você sabe, variáveis globais têm o armazenamento alocado em tempo de compilação. Variáveis locais usam a pilha.

No entanto, nem variáveis globais nem locais podem ser acrescentadas durante o tempo de execução. Porém, haverá momentos em que um programa precisará usar quantidades de armazenamento variáveis. Por exemplo, um processador de texto ou um banco de dados aproveita toda a RAM de um sistema. Porém, como a quantidade de RAM varia entre computadores esses programas não poderão usar variáveis normais. Em vez disso, esses e outros programas alocam memória, conforme necessário, usando as funções do sistema de alocação dinâmica de C.

A memória alocada pelas funções de alocação dinâmica de C é obtida do *heap* — a região de memória livre que está entre seu programa e a área de armazenamento permanente e a pilha. Embora o tamanho do heap seja desconhecido, ele geralmente contém uma quantidade razoavelmente grande de memória livre.

O coração do sistema de alocação dinâmica de C consiste nas funções **malloc()** e **free()**. (Na verdade, C tem diversas outras funções de alocação dinâmica, mas essas duas são as mais importantes.) Essas funções operam em conjunto, usando a região de memória livre para estabelecer e manter uma lista de armazenamento disponível. A função **malloc()** aloca memória e **free()** a libera. Isto é, cada vez que é feita uma solicitação de memória por **malloc()**, uma porção da memória livre restante é alocada. Cada vez que é efetuada uma chamada a **free()** para liberação de memória, a memória é devolvida ao sistema. Qualquer programa que use essas funções deve incluir o cabeçalho **STDLIB.H**.

A função **malloc()** tem este protótipo:

void *malloc(size_t *número_de_bytes*);

Aqui, *número_de_bytes* é o número de bytes de memória que você quer alocar. (O tipo size_t é definido em **STDLIB.H** como — mais ou menos — um inteiro sem sinal.) A função **malloc()** devolve um ponteiro do tipo **void**, o que significa que você pode atribuí-lo a qualquer tipo de ponteiro. Após uma chamada bem-sucedida, **malloc()** devolve um ponteiro para o primeiro byte da região de memória alocada do heap. Se não há memória disponível para satisfazer a requisição de **malloc()**, ocorre uma falha de alocação e **malloc()** devolve um nulo.

O fragmento de código mostrado aqui aloca 1000 bytes de memória.

```
char *p;
p = malloc(1000); /* obtém 1000 bytes */
```

Após a atribuição, p aponta para o primeiro dos 1000 bytes de memória livre.

O próximo exemplo aloca espaço para 50 inteiros. Observe o uso de **sizeof** para assegurar portabilidade.

```
int *p;
p = malloc(50*sizeof(int));
```

Como o heap não é infinito, sempre que alocar memória, você deve testar o valor devolvido por **malloc()**, antes de usar o ponteiro, para estar certo de que não é nulo. Usar um ponteiro nulo quase certamente travará o computador. A maneira adequada de alocar memória é ilustrada neste fragmento de código:

```
if(!(p=malloc(100)) {
  printf("sem memória.\n");
  exit(1);
}
```

Obviamente, você pode substituir algum outro tipo de manipulador de erro em lugar do **exit()**. Apenas tenha certeza de não usar o ponteiro **p** se ele for nulo.

A função **free()** é o oposto de **malloc()**, visto que ela devolve memória previamente alocada ao sistema. Uma vez que a memória tenha sido liberada, ela pode ser reutilizada por uma chamada subseqüente a **malloc()**. A função **free()** tem este protótipo:

void free(void *p);

Aqui, *p* é um ponteiro para memória alocada anteriormente por **malloc()**. É muito importante que você *nunca* use **free()** com um argumento inválido; isso destruiria a lista de memória livre.

O subsistema de alocação dinâmica de C é usado em conjunção com ponteiros para suportar uma variedade de construções de programação importantes, como listas encadeadas e árvores binárias. Você verá diversos exemplos disso na Parte 3. Um outro uso importante de alocação dinâmica é a matriz dinâmica, discutida a seguir.

Matrizes Dinamicamente Alocadas

Algumas vezes você terá de alocar memória, usando **malloc()**, mas operar na memória como se ela fosse uma matriz, usando indexação de matrizes. Em essência, você pode querer criar uma *matriz dinamicamente alocada*. Como qualquer ponteiro pode ser indexado como se fosse uma matriz unidimensional, isso não representa nenhum problema. Por exemplo, o programa seguinte mostra como você pode usar uma matriz alocada dinamicamente:

```
/* Aloca espaço para uma string dinamicamente, solicita
   a entrada do usuário e, em seguida, imprime a string de
   trás para frente. */

#include <stdlib.h>
```



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

void main(void)
{
  char *s;
  register int t;

  s = malloc(80);

  if(!s) {
    printf("Falha na solicitação de memória.\n");
    exit(1);
  }

  gets(s);
  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
  free(s);
}
```

Como o programa mostra, antes de seu primeiro uso, **s** é testado para assegurar que a solicitação de alocação foi bem-sucedida e um ponteiro válido foi devolvido por **malloc()**. Isso é absolutamente necessário para evitar o uso acidental de um ponteiro nulo, que, como exposto anteriormente, quase certamente provocaria um problema. Observe como o ponteiro **s** é usado na chamada a **gets()** e, em seguida, indexado como uma matriz para imprimir a string de trás para frente.

Acessar memória alocada como se fosse uma matriz unidimensional é simples. No entanto, matrizes dinâmicas multidimensionais levantam alguns problemas. Como as dimensões da matriz não foram definidas no programa, você não pode indexar diretamente um ponteiro como se ele fosse uma matriz multidimensional. Para conseguir uma matriz alocada dinamicamente, você deve usar este truque: passar o ponteiro como um parâmetro a uma função. Dessa forma, a função pode definir as dimensões do parâmetro que recebe o ponteiro, permitindo, assim, a indexação normal de matriz. Para ver como isso funciona, estude o exemplo seguinte, que constrói uma tabela dos números de 1 a 10 elevados a primeira, à segunda, à terceira e à quarta potências:

```
/* Apresenta as potências dos números de 1 a 10.
   Nota: muito embora esse programa esteja correto,
   alguns compiladores apresentarão uma mensagem de
   advertência com relação aos argumentos para as funções
   table() e show(). Se isso acontecer, ignore. */
```

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

int pwr(int a, int b);
void table(int p[4][10]);
void show(int p[4][10]);

void main(void)
{
  int *p;

  p = malloc(40*sizeof(int));

  if(!p) {
    printf("Falha na solicitação de memória.\n");
    exit(1);
  }

  /* aqui, p é simplesmente um ponteiro */
  table(p);
  show(p);
}

/* Constrói a tabela de potências. */
void table(int p[4][10]) /* Agora o compilador tem uma matriz
                            para trabalhar. */
{
  register int i, j;

  for(j=1; j<11; j++)
    for(i=1; i<5; i++) p[i-1][j-1] = pwr(j, i);
}

/* Exibe a tabela de potências inteiras. */
void show(int p[4][10]) /* Agora o compilador tem uma matriz
                           para trabalhar. */
{
  register int i, j;

  printf("%10s %10s %10s %10s\n",
         "N", "N^2", "N^3", "N^4");
  for(j=1; j<11; j++) {
    for(i=1; i<5; i++) printf("%10d ", p[i-1][j-1]);
    printf("\n");
  }
```



```
}

/* Eleva um inteiro a uma potência especificada. */
pwr(int a, int b)
{
  register int t=1;

  for(; b: b--) t = t*a;
  return t;
}
```

A saída produzida por esse programa é mostrada na Tabela 5.1.

Tabela 5.1 A saída do programa de potências.

N	N^2	N^3	N^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

Como o programa anterior ilustra, ao definir um parâmetro de função com as dimensões da matriz desejada, você pode enganar o compilador C para manipular matrizes dinâmicas multidimensionais. De fato, no que diz respeito ao compilador C, você tem uma matriz 4,10 dentro das funções **show()** e **table()**. A diferença é que o armazenamento para a matriz é alocado manualmente usando o comando **malloc()**, em lugar de usar o comando normal de declaração de matriz. Além disso, note o uso de **sizeof** para calcular o número de bytes necessários a uma matriz inteira de 4,10. Isso garante a portabilidade desse programa para computadores com inteiros de tamanhos diferentes.

Problemas com Ponteiros

Nada pode trazer mais problemas do que um ponteiro selvagem! Ponteiros são uma bênção que pode trazer problemas. Eles lhe dão uma capacidade formidável