CAPÍTULO 1

INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO C

1.1 Histórico da Linguagem C

A linguagem C foi desenvolvida entre os anos 1971 e 1973 por Dennis Ritchie, nos laboratórios da AT&T, como uma linguagem para implementação de sistemas operacionais e outros programas básicos. A intenção de Ritchie era criar uma linguagem que incorporasse características de baixo nível, necessárias para produzir programas eficientes, com características de alto nível, necessárias para tornar os programas legíveis, portáveis e fáceis de ser mantidos.

Antes do surgimento de C, sistemas operacionais eram tradicionalmente escritos em assembly. Isso deve-se ao fato de a rapidez de execução destes programas ser um fator crítico e também porque o programador de sistemas precisa ter acesso a certas facilidades de baixo nível que não são comuns em outras linguagens de alto nível. Em 1973, Ritchie demonstrou o poder e a flexibilidade de C para este tipo de tarefa ao escrever nesta linguagem a maior parte do sistema operacional Unix.

Escrever um sistema operacional numa linguagem de alto nível tem três vantagens principais com relação a assembly: (1) maior rapidez de implementação, (2) legibilidade e (3) portabilidade. A portabilidade permite que o sistema operacional seja transportado para outro computador sem que ele precise ser completamente reescrito: é necessário apenas que se tenha um compilador para a máquina desejada e que o

sistema operacional seja recompilado. Por causa disso, as evoluções de C e de Unix caminharam juntas inicialmente: cada implementação de Unix para uma nova máquina requeria um compilador de C para essa máquina. Assim, muitos imaginavam C como uma linguagem desenvolvida estritamente para implementação de Unix.

A partir de meados dos anos 1980, C passou a ser reconhecida como uma linguagem de propósito geral e, com o ganho de popularidade, a contar com vários compiladores comerciais. Muitos compiladores apresentavam características diferentes daquelas que faziam parte da especificação original da linguagem e essas discrepâncias comprometiam uma das principais características desejáveis numa linguagem de alto nível: a portabilidade. Para que houvesse compatibilidade entre compiladores, foi necessário criar uma padronização da linguagem que fosse seguida pelos desenvolvedores de compiladores.

A padronização da linguagem C foi inicialmente realizada por um comitê do *American National Standards Institute* (ANSI) e, por causa disso, foi denominada de **ANSI C**, ou, simplesmente, **C padrão**. Em 1990, a organização ISO assumiu a responsabilidade pela padronização de C e aprovou a padronização ANSI. A versão mais recente de padronização de C foi homologada em 1999 e é conhecida como **C99**. A **Tabela 1** apresenta resumidamente os principais marcos evolutivos da linguagem C.

ANO	MARCO
1969	Ken Thompson cria Unix e a linguagem B baseada na linguagem BCPL.
1971	Dennis Ritchie inicia o desenvolvimento de C como uma evolução da linguagem B.
1978	Kernighan e Ritchie lançam o livro *A Linguagem de Programação C* que, durante anos, foi considerado a definição oficial da linguagem (este livro é popularmente conhecido como K&R).
1983	ANSI começa a padronização de C.

1989	A padronização ANSI de C é homologada. Esta padronização passou a ser conhecida inicialmente como ANSI C e, posteriormente, como **C89**.
1990	ISO aprova a padronização de ANSI C. Popularmente, esta padronização é denominada **C90**, que é essencialmente a mesma padronização C89; oficialmente, esta padronização é conhecida como ISO/IEC 9899:1990.
1994	ISO apresenta uma correção ao padrão C90, oficialmente conhecida como ISO/IEC 9899/COR1:1994.
1995	ISO apresenta uma emenda ao padrão C90, oficialmente conhecida como ISO/IEC 9899/AM1:1995. O padrão C90 acrescido por esta emenda passou a ser conhecido como **C95**. Esta emenda é importante, pois introduziu suporte a caracteres extensos e multibytes na biblioteca padrão de C.
1995	Começam os trabalhos para uma nova padronização de C.
1996	ISO apresenta uma correção ao padrão C95, oficialmente conhecida como ISO/IEC 9899/COR2:1996.
1999	ISO aprova o mais novo padrão de C, denominado popularmente **C99** e oficialmente ISO/IEC 9899:1999.
2001	ISO apresenta uma correção ao padrão C99, oficialmente conhecida como ISO/IEC 9899 Cor. 1: 2001.
2004	ISO apresenta uma correção ao padrão C99, oficialmente conhecida como ISO/IEC 9899 Cor. 2: 2004.

Tabela 1: Marcos Evolutivos da Linguagem C

Apesar de desenvolverem suas próprias versões de C, muitos fabricantes de compiladores de C oferecem a opção ANSI C (ou ISO C), que pode ser escolhida pelo programador, por exemplo, por meio de uma

caixa de diálogo ou diretiva. Em nome da portabilidade, é sempre bom utilizar a opção ANSI/ISO C[1].

Não obstante todos os esforços de padronização, muitas construções da linguagem C ainda têm interpretações que variam de compilador para compilador. Isto ocorre devido a ambigüidades e omissões ora na definição original da linguagem ora em versões de padronização. Esses problemas, apesar de reconhecidos, são perpetuados a cada novo padrão para garantir compatibilidade histórica; isto é, para garantir que programas antigos continuem podendo ser compilados num padrão mais recente. Ao longo do texto, cada característica de C que pode ter mais de uma interpretação é identificada como *dependente de implementação*. Este termo é mais preciso do que *dependente de compilador*, pois diferentes versões (i.e., implementações) de um mesmo compilador podem dar interpretações diferentes para uma dada construção da linguagem.

Apesar do sucesso obtido, a linguagem C também adquiriu a reputação de ser uma linguagem intrinsecamente ilegível e que promove maus hábitos de programação. Outra reclamação freqüente diz respeito à condescendência dos compiladores de C com relação a algumas construções da linguagem; i.e., um compilador de C não impõe regras mais rígidas que protejam o programador contra eventuais erros, como o fazem outras linguagens (por exemplo, Modula-2 e Java). Defensores de C, no entanto, alegam que esta linguagem foi desenvolvida para programadores experientes e, portanto, ela assume pouco a respeito daquilo que o programador deseja ou não fazer.

Em resumo, C é uma linguagem extremamente poderosa e que dá muita liberdade ao programador para escrever programas que seriam muito difíceis de ser escritos em outra linguagem. Entretanto, programadores inexperientes abusam desta liberdade e escrevem programas que são desnecessariamente ilegíveis e difíceis de ser mantidos. O objetivo deste livro não é simplesmente ensinar a construir programas que funcionam, mas programas que, além de funcionarem, são bem construídos, fáceis

[1] Alguns compiladores, como gcc, até mesmo oferecem opção para escolha entre vários padrões ISO: C90, C95 etc.

de ser lidos e mantidos. Portanto, lembre-se que, apesar de poderosa, C é uma linguagem que exige muita disciplina por parte do programador para que este objetivo seja alcançado.

1.2 Aspectos Fundamentais

1.2.1 Identificadores e Palavras-chave

Um **identificador** serve para nomear componentes utilizados num programa escrito numa dada linguagem de programação. Cada linguagem possui regras próprias para formação de seus identificadores. Um identificador em C deve ser constituído apenas de letras[2], dígitos e o caractere especial sublinha (_), sendo que o primeiro elemento constituinte não pode ser um dígito. O número de caracteres permitido para identificadores depende da versão do padrão de C utilizada. O padrão ISO C90 requer que compiladores de C aceitem, pelo menos, identificadores contendo 31 caracteres, mas, no padrão C99, este número subiu para 63. Normalmente, compiladores modernos dão liberdade para programadores *sensatos* escreverem identificadores do tamanho desejado.

Uma característica importante da linguagem C é que ela, diferentemente de algumas outras linguagens (por exemplo, Pascal), faz distinção entre letras maiúsculas e minúsculas. Isto significa, por exemplo, que duas variáveis com nomes `minhaVar` e `MinhaVar` são diferentes.

Existem algumas palavras que já são utilizadas pela própria linguagem C ou por sua biblioteca padrão. Palavras que possuem significado especial na linguagem (por exemplo, **while**, **for**) e que *não podem* ser utilizadas como identificadores pelo programador são conhecidas como **palavras-chave** da linguagem. Identificadores exportados pela bibliotecas padrão (por exemplo, **printf**, **NULL**) são conhecidos como **identificadores reservados** e o programador deve evitar redefini-los como identificadores em seus programas. Também não é recomendado que se escrevam identificadores começando por sublinha, pois o uso deste caractere no início de identificadores é reservado pela linguagem.

---

[2] Letra aqui não inclui ç ou caracteres acentuados de Português, embora, de acordo com o padrão C99, estes caracteres possam ser utilizados em identificadores (v. **Seção 8.7**).

1.2.2 Tipos de Dados Primitivos

Os tipos de dados de C podem ser classificados de acordo com a hierarquia apresentada na **Figura 1**.

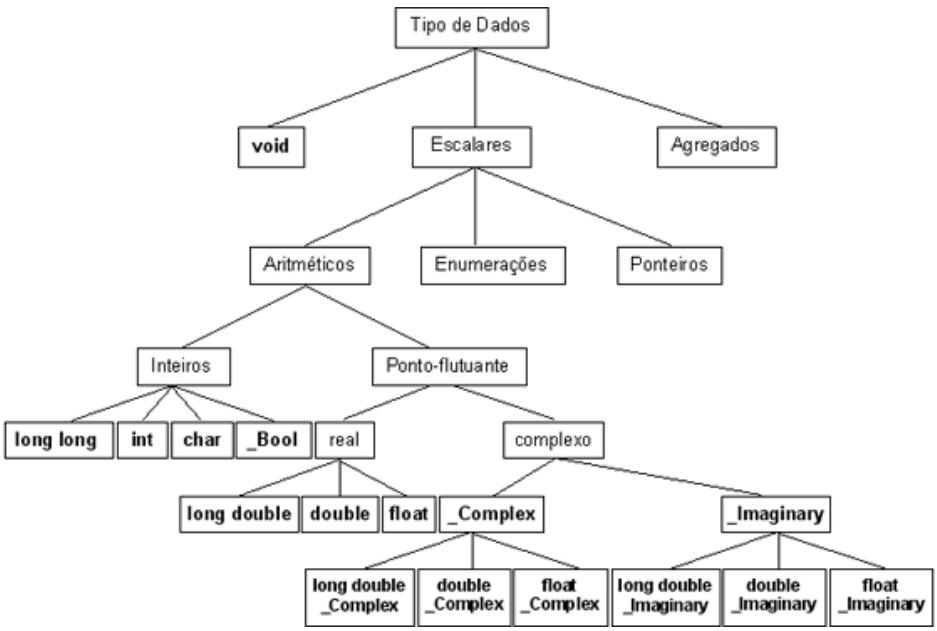

Figura 1: Tipos de dados da Linguagem C

Um tipo **primitivo** (ou **embutido**) é um tipo incorporado na própria linguagem e representado por uma palavra-chave. Os tipos de dados primitivos de C aparecem em negrito na **Figura 1**. A linguagem C oferece ainda a possibilidade de criação de inúmeros outros tipos. Estes tipos, denominados **tipos derivados**, podem ser criados pelo próprio programador (v. **Seção 4.6**) ou providos pela biblioteca padrão de C. Nesta seção, os tipos aritméticos primitivos serão estudados.

► **Tipos Inteiros**

Conforme pode-se verificar na **Figura 1**, existem quatro tipos básicos de inteiros em C: **int**, **char**, **_Bool** e **long long int** (que pode ser abreviado para **long long**). O espaço ocupado em memória por valores destes tipos é

definido apenas para os tipos **char**, que sempre ocupa um byte, e **long long** (C99), que sempre ocupa 8 bytes. Os tamanhos ocupados por valores dos outros tipos são dependentes de implementação.

É interessante notar que, em C, o tipo **char** não apenas é utilizado para representar caracteres como em muitas linguagens, mas também pode representar inteiros que requerem apenas um byte de memória. Em outras palavras, uma variável do tipo **char** pode representar tanto um caractere quanto um inteiro, desde que este último caiba em um byte.

O tipo **int** pode ainda ser qualificado com as seguintes palavras-chave:

· **short** ou **long**, que se referem ao tamanho do tipo.

· **signed** ou **unsigned**, que se referem ao fato de o tipo ser considerado com sinal ou sem sinal, respectivamente.

Os qualificadores **signed** e **unsigned** podem também ser utilizados para qualificar os tipos **char** e **long long**[3]. A qualificação de um tipo é feita antepondo-se o qualificador ao nome do tipo (por exemplo, **signed char**, **unsigned long int**). No caso do tipo **int**, pode-se ainda abreviar **short int** e **long int** para simplesmente **short** e **long**; **signed** e **unsigned** podem ainda ser utilizados para abreviar **signed int** e **unsigned int**, respectivamente. No caso de uso de dois qualificadores, **signed** ou **unsigned** deve preceder **short** ou **long**. Neste último caso, pode-se também abreviar deixando a palavra **int** de fora (por exemplo, **unsigned long** é o mesmo que **unsigned long int**). Note que os pares **signed** e **unsigned**, e **short** e **long** são mutuamente exclusivos (por exemplo, não faz sentido ter um tipo que seja **signed** e **unsigned** ao mesmo tempo).

Por padrão, os tipos **int** e **long long** apresentam-se com sinal[4], enquanto que o tipo **char** pode ter sinal ou não como padrão, dependendo do compilador utilizado. Note que o uso de um tipo sem sinal aproximadamente duplica o maior valor positivo que o tipo poderia conter

---

[3] Nenhuma das palavras *long* do tipo **long long** é considerada um qualificador. Ou seja, elas fazem parte do próprio nome do tipo.

[4] Uma implicação óbvia desse fato é que o uso de **signed** com **int** e **long long** é redundante.

se incluísse sinal. Por exemplo, o tipo **signed char**, que é representado em um byte, pode acomodar qualquer número no intervalo de -128 a 127, enquanto o tipo **unsigned char** pode acomodar qualquer número no intervalo de 0 a 255.

As interpretações de **short** e **long** para o tipo **int** variam de compilador para compilador. Por exemplo, num dado compilador **short int** e **int** significam a mesma coisa (i.e., ocupam o mesmo espaço em memória), enquanto **long int** ocupa o dobro deste espaço; em outro compilador, **long int** e **int** ocupam o mesmo espaço em memória, enquanto **short int** ocupa metade deste espaço. Apesar dessa dependência de implementação, de acordo com o padrão de C, **short int** deve sempre ocupar menos espaço do que **long int**.

Conforme você deve ter percebido, os tipos inteiros de C, com exceção de **long long**, que foi introduzido em C99, não são portáveis. Reconhecidamente, esta é uma falha da especificação original da linguagem que se perpetuou em subseqüentes padronizações devido a razões de compatibilidade histórica. Portanto, o programador deve ter muita cautela quando utilizar estes tipos em situações práticas. Ao longo deste livro, estes tipos são utilizados em diversos exemplos, pois, dada a natureza didática e a relativa simplicidade destes exemplos, existe pouca possibilidade de ocorrerem problemas de portabilidade. Para resolver estes problemas de portabilidade, o padrão C99 introduziu o arquivo de cabeçalho `<stdint.h>`, onde são definidos vários tipos inteiros de tamanho definido. O uso desses tipos em vez dos tipos inteiros primitivos de C é altamente recomendável para evitar qualquer problema de portabilidade dessa natureza.

De qualquer modo, a **Tabela 2** apresentada a seguir pode ser utilizada como guia para a escolha do tipo de dado inteiro adequado para as necessidades de seus programas, desde que o uso destes tipos não acarrete problemas de portabilidade. Note, entretanto, que, exceto para os tipos **long long** e **char**, os tamanhos e respectivos intervalos de valores dos tipos apresentados na **Tabela 2** são apenas *típicos* (i.e., prováveis).

Não existe nenhuma garantia de que um dado compilador realmente utilize estes tamanhos e intervalos de valores.

TIPO	TAMANHO (em bytes)	INTERVALO DE VALORES
int	4	-2^{31} a $2^{31} - 1$
short int	2	-2^{15} a $2^{15} - 1$
long int	4	-2^{31} a $2^{31} - 1$
unsigned int	4	0 a $2^{32} - 1$
unsigned short int	2	0 a $2^{16} - 1$
unsigned long int	4	0 a $2^{32} - 1$
signed char	1	-2^{7} a $2^{7} - 1$
unsigned char	1	0 a $2^{8} - 1$
long long int	8	-2^{63} a $2^{63} - 1$
unsigned long long int	8	0 a $2^{64} - 1$

Tabela 2: Intervalos típicos de valores de tipos inteiros

▶ **Tipos de Ponto-flutuante Reais**

Os tipos de ponto-flutuante em C são divididos em duas categorias: **números reais** e **números complexos**, sendo que esta segunda categoria foi introduzida pelo padrão C99. Existem três tipos de ponto-flutuante reais: **float**, **double** e **long double**. O padrão de C não especifica os tamanhos destes tipos, mas especifica que o conjunto de valores do tipo **float** é um subconjunto do conjunto de valores do tipo **double**, que, por sua vez, é um subconjunto do conjunto de valores do tipo **long double**.

▶ **Tipos de Ponto-flutuante Complexos (C99)**

O padrão C99 introduziu uma abstração que representa os números complexos de matemática. Para tanto, foram incorporadas duas novas palavras-chave: **_Complex** e **_Imaginary**[5], e criados os tipos de dados

[5] A razão pela qual o padrão C99 utiliza **_Complex** e **_Imaginary** em vez de **complex** e **imaginary**, que seriam identificadores mais naturais, é o fato de muitos programas já existentes à época de publicação deste padrão possuírem suas próprias implementações de números complexos utilizando os identificadores **complex** e **imaginary**. Assim, a utilização destes mesmos identificadores como palavras-chave no novo padrão certamente traria problemas para estes programas. Novos programas, que não são sensíveis a este problema, podem usar as macros **complex** e **imaginary** definidas no arquivo de cabeçalho `<complex.h>` da biblioteca padrão de C.

primitivos apresentados na **Tabela 3** com suas respectivas interpretações.

TIPO	INTERPRETAÇÃO
float _Complex	Tipo complexo cujas partes real e imaginária são do tipo **float**.
double _Complex	Tipo complexo cujas partes real e imaginária são do tipo **double**.
long double _Complex	Tipo complexo cujas partes real e imaginária são do tipo **long double**.
float _Imaginary	Tipo imaginário puro cuja parte imaginária é do tipo **float**.
double _Imaginary	Tipo imaginário puro cuja parte imaginária é do tipo **double**.
long double _Imaginary	Tipo imaginário puro cuja parte imaginária é do tipo **long double**.

Tabela 3: Tipos complexos introduzidos pelo padrão C99

Os tipos apresentados na **Tabela 3** também podem, alternativamente, ser denominados como:

- **_Complex float**
- **_Complex double**
- **_Complex long double**
- **_Imaginary float**
- **_Imaginary double**
- **_Imaginary long double**

Os tipos apresentados na **Tabela 3** são considerados tipos de ponto-flutuante. Para cada tipo de ponto-flutuante preexistente em C (i.e., tipos reais), o padrão C99 introduziu um novo tipo complexo (**_Complex**) correspondente, e o mesmo ocorreu com relação ao tipos imaginários puros (**_Imaginary**).

De acordo com o padrão C99, o suporte para os tipos imaginários puros (i.e., os três últimos tipos enumerados na **Tabela 3**) é opcional. O padrão C99 também introduz inúmeras funções que executam operações sobre números complexos. Estas funções são descritas no **Volume II**.

Números complexos têm importância crucial em várias áreas de conhecimentos, como, por exemplo, Física, Engenharia e Eletrônica. Para ser capaz de aproveitar bem as facilidades introduzidas por C99 para processamento de números complexos, é necessário conhecimento matemático que vai além do conhecimento básico adquirido sobre o assunto. Portanto, se você não possui o necessário conhecimento e não constrói programas em áreas onde números complexos são importantes, não se preocupe em conhecer bem a implementação de números complexos em C.

1.2.3 Constantes

Existem cinco tipos de constantes em C: **inteiras**, de **ponto-flutuante**, **caracteres**, ***strings*** e de **enumerações**. Constantes deste último tipo serão vistas na **Seção 9.7**, enquanto as demais serão apresentadas a seguir.

► **Constantes Inteiras**

Constantes inteiras podem ser de três tipos, de acordo com a base numérica utilizada:

• **Decimais**. São as constantes inteiras mais utilizadas em programação de alto nível e são escritas utilizando-se os dígitos de 0 a 9, sendo que o primeiro dígito não pode ser zero, a não ser, é claro, quando o próprio valor é zero. Exemplos válidos de constantes inteiras decimais: `0`, `12004`, `-67`; um exemplo inválido seria `09`, pois o número começa com 0.

• **Octais**. Constantes inteiras octais representam números inteiros na base octal. Estas constantes devem começar com `0` (zero) e utilizar os dígitos de 0 a 7. Exemplos válidos: `00` (representa o número 0 em octal), `07`, `07744`. Exemplos inválidos: `085` (8 não é um dígito octal), `77` (não começa com 0 e, portanto, seria interpretado com decimal).

• **Hexadecimais**. Constantes inteiras hexadecimais representam números inteiros na base hexadecimal. Estas constantes devem começar com `0x` ou `0X` e utilizar os dígitos de 0 a 9 e as letras de `A` a `F` (ou `a` a `f`). Exemplos válidos: `0x7FFA`, `0X230`, `0xf2de`.

Pode-se explicitamente sugerir que uma constante inteira seja interpretada como sendo de um determinado tipo primitivo por meio do uso de sufixos. Os sufixos permitidos para constantes inteiras são apresentados na **Tabela 4**.

SUFIXO	SUGERE QUE A CONSTANTE SEJA...	EXEMPLOS
l ou **L**	long	`15L, 0xAAB3FL`
ll ou **LL**	long long	`465LL`
u ou **U**	unsigned int	`5U, 0xAEF3FU`
ul, **uL**, **Ul** ou **UL**,	unsigned long	`15UL`
ull, **uLL**, **Ull** ou **ULL**	unsigned long long	`32ULL`

Tabela 4: Sufixos utilizados com constantes inteiras

Antes da introdução do tipo **long long** pelo padrão C99, o tipo inteiro de maior capacidade era **long**, o que significava dizer que qualquer valor inteiro com sinal representável em C podia ser armazenado como um valor deste tipo. O tipo **long long** introduziu uma exceção a esta regra.

Na versão anterior de C99, uma constante inteira sem sufixo `u` ou `U` seria considerada do primeiro tipo em que coubesse na seqüência: **int** ⟶ **long** ⟶ **unsigned long**. Com a introdução do tipo **long long**, as regras para constantes inteiras passaram a ser as seguintes:

• Constantes inteiras na base decimal sem sufixo `u` ou `U` são sempre consideradas **signed**.

• Constantes inteiras na base octal ou hexadecimal podem ser consideradas **signed** ou **unsigned** dependendo do primeiro tipo que a constante couber na seqüência: **int** ⟶ **unsigned int** ⟶ **long** ⟶ **unsigned long** ⟶ **long long** ⟶ **unsigned long long**.

As novas regras de interpretação de constantes inteiras introduzidas em C99 são resumidas na **Tabela 5**.

SUFIXO	CONSTANTE DECIMAL	CONSTANTE OCTAL OU HEXADECIMAL
Nenhum	**int → long → long long**	**int → unsigned → long → unsigned long → long long → unsigned long long**
U ou u	**unsigned int → unsigned long → unsigned long long**	**unsigned int → unsigned long → unsigned long long**
L ou l	**long → long long**	**long → unsigned long → long long → unsigned long long**
U ou u e L ou l	**unsigned long → unsigned long long**	**unsigned long → unsigned long long**
LL ou ll	**long long**	**long long → unsigned long long**
U ou u e LL ou ll	**unsigned long long**	**unsigned long long**

Tabela 5: Regras para acomodação de constantes inteiras (C99)

Na **Tabela 5**, "→" significa "se não couber no tipo anterior". Por exemplo, a seqüência **int → long → long long** deve ser interpretada como: se couber no tipo **int**, a constante será **int**; se não couber neste tipo, será **long**; se não couber neste tipo, será **long long**.

▶ **Constantes de Ponto-Flutuante**

Existem dois tipos de constantes de ponto-flutuante: reais e complexas. Constantes de ponto-flutuante reais podem ser escritas de três maneiras. A primeira delas é simplesmente colocando-se um ponto decimal separando as partes inteira e decimal do número, como, por exemplo:

```
3.1415
.5
7.
```

A segunda forma de se escrever uma constante de ponto flutuante é por meio de **notação científica**. Nesta notação, um número de ponto flutuante consiste em duas partes: (1) **mantissa** e (2) **expoente**; esta última representa uma potência de 10. Estas duas partes são separadas pela letra **e** ou **E**. Por exemplo, o número `2E4` deve ser interpretado como $2x10^4$ e lido como "dois vezes 10 elevado à quarta potência". A mantissa de um número de ponto flutuante pode ser inteira ou decimal; o expoente pode ser positivo ou negativo, mas *deve* ser inteiro. Por exemplo, `2.5E-3` representa o número $2.5x10^{-3}$, ou seja, `0.0025`.

No padrão C99, constantes de ponto-flutuante podem ainda ser representadas em **formato hexadecimal**. Este formato permite que constantes de ponto-flutuante sejam especificadas com o máximo de precisão. Uma constante de ponto-flutuante em formato hexadecimal consiste em:

- Um prefixo hexadecimal (`0x` ou `0X`).
- Algarismos hexadecimais significativos representando a parte inteira e a parte fracionária; ambas as partes são opcionais, mas não ao mesmo tempo; a parte fracionária, quando presente, começa com ponto.
- Um expoente binário representando uma potência de 2 que deve ser multiplicada pela parte significativa; este valor é um inteiro na base decimal com ou sem sinal.
- Uma letra `P` ou `p` separando os algarismos significativos do expoente.
- Um sufixo (opcional), que pode ser `f`, `F`, `l` ou `L`, com o mesmo significado (**float** ou **long double**) visto anteriormente para constantes de ponto-flutuante decimais.

Por exemplo, a constante de ponto-flutuante em formato hexadecimal:

```
0x1P-1
```

corresponde a:

```
0.5
```

pois:

$1 * 2^{-1} = 0.5$

Note que, diferentemente do que ocorre com constantes de ponto-flutuante em formato decimal, o expoente não é opcional[6].

A **Tabela 6** apresenta alguns outros exemplos de constantes de ponto-flutuante em formato decimal com os respectivos valores correspondentes em formato hexadecimal:

FORMATO HEXADECIMAL	FORMATO DECIMAL
0x1P2	4.0
0x56.46P-3	10.784180
-0x5.3456P7f	-666.167969f

Tabela 6: Exemplos de constantes de ponto-flutuante em formato hexadecimal

O padrão C99 não introduziu nenhuma nova sintaxe para a escrita de constantes complexas, de modo que, para ser capaz de escrever tais constantes, é necessário incluir o arquivo de cabeçalho `<complex.h>` e utilizar a constante **_Complex_I**[7] ou, mais naturalmente, **I**. Ambas representam $\sqrt{-1}$ em Matemática. Por exemplo, a constante:

```
1 + 2.5*_Complex_I
```

é o mesmo que:

```
1 + 2.5*I
```

Constantes complexas podem utilizar em suas partes real e imaginária os mesmos formatos de constantes de ponto-flutuante vistas antes, incluindo os mesmos sufixos.

---

[6] Este requisito é necessário porque, caso contrário, haveria ambigüidade com a letra f, que tanto pode representar um dígito hexadecimal quanto o sufixo de constantes do tipo float. Por exemplo, a constante 0x1.0f poderia representar 1.0f (i.e., um valor do tipo float em formato decimal) ou 1.9375 (i.e., um valor do tipo double em formato hexadecimal).

[7] A razão pela qual utiliza-se a constante _Complex_I em vez de apenas I é o fato de muitos programas já existentes à época de publicação deste padrão possuírem suas próprias implementações de números complexos utilizando o identificador I.

▶ Caracteres Constantes

Uma constante do tipo **char** pode ser representada tanto por um caractere entre apóstrofos (por exemplo, `'A'`), quanto por um número inteiro capaz de ser contido em um byte. Uma constante composta de um caractere entre apóstrofos apenas informa ao compilador que ele deve considerar o valor correspondente ao caractere no código de caracteres ora utilizado. Por exemplo, se o código de caracteres utilizado é ASCII, quando o compilador encontra a constante `'A'`, ele a interpreta como o valor `65` (este valor corresponde ao código ASCII do caractere `A`). Este mesmo valor poderia também ser obtido utilizando uma constante do tipo **char** com o valor `65` em vez de `'A'`[8].

Uma outra maneira de se escreverem caracteres constantes é por meio de **seqüências de escape**. Uma seqüência de escape começa sempre com o caractere \ (denominado **caractere de escape**) seguido de (1) um caractere com significado especial ou (2) um número em formato octal ou hexadecimal. Em ambos os casos, a seqüência de escape deve ser estar entre apóstrofos.

O primeiro formato de seqüências de escape deve ser utilizado para representar caracteres não-imprimíveis, i.e., caracteres que não possuem representação gráfica. Estes caracteres tipicamente produzem certos efeitos especiais no meio de saída quando utilizados em instruções de saída. O primeiro formato de seqüências de escape também deve ser utilizado para a escrita de caracteres constantes que possuem significados especiais em C. A **Tabela 7** apresenta as seqüências de escape deste formato que podem ser utilizados em C.

[8] Não é uma boa idéia representar caracteres num programa por constantes inteiras, por duas razões. A primeira razão é legibilidade (por exemplo, será que alguém vai entender o que significa 65 quando ler seu programa?). Requerer que alguém use uma tabela de caracteres para entender seu programa não é sensato. A segunda é que o programa pode ter sua portabilidade comprometida, pois o padrão de C não especifica que o código de caracteres utilizado por um compilador deva ser ASCII ou qualquer outro.

SEQÜÊNCIA DE ESCAPE	DESCRIÇÃO
\a	Campainha (alerta)
\b	Backspace
\t	Tabulação horizontal
\v	Tabulação vertical
\n	Quebra de linha
\f	Quebra de página
\r	Retorno de carro
\e	Escape
\0	Caractere nulo
\\	Barra invertida
\?	Interrogação
\'	Apóstrofo
\"	Aspas

Tabela 7: Seqüências de escape e seus significados

O segundo formato de seqüências de escape pode ser utilizado para representar qualquer caractere do conjunto básico de caracteres, inclusive aqueles que podem ser representados nas formas apresentadas acima. Neste caso, deve-se escrever o valor correspondente ao código do caractere em notação octal ou hexadecimal (neste último caso, precedido por `x` minúsculo). Por exemplo, se o código utilizado é o ASCII, a letra `Z` poderia ser escrita em seqüência de escape como `'\132'` (octal) ou `'\x5A'` (hexadecimal). Esta forma de representação de caracteres é indicada para representar caracteres que não podem ser representados nos formatos anteriores. Ela não é usualmente necessária nem é portável.

► **Strings Constantes**

Um **string constante** consiste em uma seqüência de caracteres constantes. Em C, um *string* constante pode conter caracteres constantes em qualquer dos formatos apresentados acima e deve ser envolvido entre aspas. Exemplos de *strings* constantes são apresentados a seguir:

```
"bola"
"Dia\tMes\tAno\n"
"\x62\x6F\x6C\x61"
```

Strings constantes separados por espaços em branco são automaticamente concatenados pelo compilador. Isto é útil quando se tem um *string* constante muito grande e deseja-se escrevê-lo em duas linhas. Por exemplo:

```
"Este e' um string constante muito grande para "
"ser contido numa unica linha do meu programa"
```

Os dois *strings* constantes do último exemplo serão concatenados pelo compilador para formar um *string* constante:

```
"Este e' um string constante muito grande para ser
contido numa unica linha do meu programa"
```

▶ **Caracteres e Strings Extensos**

Caracteres extensos são caracteres que ocupam mais de um byte. Portanto, estes caracteres não são valores do tipo **char**, que ocupam apenas um byte, mas sim do tipo derivado **wchar_t**, encontrado na biblioteca padrão de C. Num programa em C, um caractere extenso constante tem sintaxe semelhante aos caracteres comuns, mas é precedido por L (por exemplo, `L'z'`).

Um **string extenso** é um *string* composto de caracteres extensos. Num programa em C, um *string* extenso constante tem sintaxe semelhante aos *strings* comuns, mas é precedido por L (por exemplo, `L"bola"`).

O **Volume II** apresenta um estudo aprofundado de caracteres extensos e *strings* extensos.

1.2.4 Definições de Variáveis

A memória de um computador pode ser vista simplesmente como um grande agrupamento de bits dividido em agrupamentos menores que são endereçáveis. Uma **variável** em programação de alto nível representa uma ou mais posições endereçáveis de memória.

Agrupamentos endereçáveis de bits podem representar quaisquer tipos de dados ou até mesmo instruções. Então, como é que o computador sabe o que representa uma dada seqüência de bits? Isto é, como é, por exemplo, que o computador sabe que uma dada seqüência de bits

representa um número inteiro e não um número de ponto flutuante ou uma seqüência de caracteres? A resposta simplesmente é: em princípio, o computador não sabe fazer este tipo de interpretação sozinho. Isto é, ele precisa ser informado pelo programador como as várias porções de memória utilizadas por um programa devem ser interpretadas. O programador faz isso por meio de **definições de variáveis**[9].

Além de prover uma interpretação para uma porção de memória, uma definição de variável também possui duas outras finalidades: (1) fazer com que seja alocado espaço suficiente para conter a variável e (2) identificar este espaço por meio de um símbolo (**identificador**) humanamente legível.

Em C, toda variável precisa ser definida antes de ser usada. Uma definição de variável em C consiste simplesmente em uma palavra-chave representando o tipo da variável seguida do nome da variável. Variáveis de um mesmo tipo podem ainda ser definidas juntas e separadas por vírgulas. Por exemplo, a definição:

```
int minhaVar, i, j;
```

define as variáveis `minhaVar`, `i` e `j` como sendo do tipo **int**.

É interessante observar que, quando utiliza-se um qualificador com o tipo **int** como tipo de uma variável, a palavra **int** pode ser omitida. Por exemplo, as definições:

```
unsigned  long  int varInt;
short  int          sInteiro;
```

[9] Os termos *declaração de variável* e *definição de variável* são, com freqüência, usados indistintamente. Em C, no entanto, estes termos representam conceitos diferentes. *Definição de variável* é exatamente o que está sendo descrito aqui, nesta seção. Uma declaração de variável, por outro lado, apenas informa o compilador da existência de uma definição de variável em algum lugar do programa. Em outras palavras, declaração de variável é o mesmo que alusão de variável (v. **Capítulo 4**). A mesma confusão ocorre com os termos *declaração de função* e *definição de função* (v. **Capítulo 3**). Em qualquer dos casos, utilize a seguinte regra prática para fazer a distinção: definição gera código em linguagem de máquina quando o programa é compilado, enquanto declaração não provoca geração de código.

podem ser abreviadas para:

```
unsigned  long  varInt;
short           sInteiro;
```

Seguem algumas recomendações para a escolha de identificadores para variáveis:

- Escolha nomes que sejam significativos (por exemplo, `matricula` é mais significativo do que `x` ou `m`).
- Evite utilizar nomes de variáveis que sejam muito parecidos ou que difiram em apenas um caractere (por exemplo, `primeiraNota` e `segundaNota` são melhores do que `nota1` e `nota2`).
- Evite utilizar `l` (*ele*) e `o` (*ó*) como nomes de variáveis, pois são facilmente confundidos com `1` (*um*) e `0` (*zero*).

Essas recomendações têm como objetivo melhorar a legibilidade dos programas. No **Capítulo 6** você encontrará outros conselhos que ajudarão a tornar seus programas mais legíveis.

1.2.5 Atribuição

Uma das instruções mais simples em C é a instrução de **atribuição**. Uma instrução deste tipo preenche o espaço em memória representado por uma variável com um valor determinado, e sua sintaxe tem a seguinte forma geral:

```
variável = expressão;
```

A interpretação para uma instrução de atribuição é a seguinte: a expressão (lado direito) é avaliada e o valor resultante é armazenado no espaço de memória representado pela variável (lado esquerdo).

Quando um espaço em memória é alocado para conter uma variável, o conteúdo deste espaço (i.e., o valor da própria variável) *pode ser* indeterminado[10]. Isto significa que não se deve fazer nenhuma

[10] Existem situações nas quais o conteúdo de uma variável é automaticamente preenchido com zeros, conforme será visto mais adiante.

suposição sobre o valor de uma variável antes que a mesma assuma um valor *explicitamente* atribuído. Às vezes, é desejável que uma variável assuma um certo valor no instante de sua definição. Esta **iniciação** pode ser feita em C combinando-se a definição da variável com a atribuição do valor desejado. Por exemplo, suponha que se deseje atribuir o valor inicial `0` a uma variável inteira `minhaVar`. Então isto poderia ser feito por meio da seguinte iniciação:

```
int  minhaVar = 0;
```

1.2.6 Constantes Simbólicas

Assim como outras linguagens de programação, C permite que se associe um identificador com um valor constante. Tal identificador é denominado **constante simbólica**. O uso de constantes simbólicas em um programa tem dois objetivos principais: (1) tornar o programa mais legível (por exemplo, `PI` é mais legível do que o valor `3.14`) e (2) tornar o programa mais fácil de ser modificado. Como ilustração deste segundo ponto, suponha que o valor constante `3.14` apareça em vários pontos de seu programa. Se você quisesse modificar este valor (digamos para `3.14159`) teria de encontrar todos os valores antigos e substituí-los. Entretanto, se este valor fosse representado por uma constante simbólica declarada no início do programa, você precisaria fazer apenas uma modificação na própria declaração da constante.

Uma constante simbólica pode ser definida em C por meio da **diretiva de pré-processador #define.** Por exemplo, suponha que se deseje denominar de `PI` o valor `3.14`. Neste caso, a seguinte diretiva seria utilizada:

```
#define  PI  3.14
```

Quando o programa contendo a declaração acima é compilado (ou, mais precisamente, pré-processado), o compilador substitui todas as ocorrências de `PI` por seu valor declarado (i.e., `3.14`).

As regras para escrita de constantes simbólicas são as mesmas daquelas utilizadas para variáveis, mas sugere-se que o programador adote

uma notação diferente da utilizada em nomes de variáveis. Aqui, nomes de constantes simbólicas serão sempre escritos em letras maiúsculas.

1.2.7 Comentários

Comentários em C são quaisquer seqüências de caracteres colocadas entre os **delimitadores de comentários** `/*` e `*/`. Os caracteres entre os delimitadores de comentários são totalmente ignorados; portanto, não existe nenhuma restrição com relação a estes caracteres. Por exemplo:

```
x = 10; /* Isto é um comentário */
```

De acordo com o padrão ISO, não é permitido o uso aninhado de comentários deste tipo; i.e., um comentário dentro de outro comentário. Entretanto, algumas implementações de C aceitam isso, o que pode ser bastante útil na depuração de programas, pois permite comentar trechos de programa que já contenham comentários.

O padrão C99 introduz o delimitador de comentário `//`. Caracteres que seguem este delimitador até o final da linha que o contém são ignorados pelo compilador. Exemplo:

```
x = 10; // Isto é um comentário
```

Este tipo de comentário pode ser aninhado em comentários do tipo anterior:

```
/*
x = 10; // Isto é um comentário
*/
```

E vice-versa:

```
x = 10; // Isto /* é um */ comentário
```

Um comentário começando com // pode ser continuado na próxima linha utilizando o caractere de continuação de linha \, conforme mostrado a seguir[11]:

[11] A barra invertida só é interpretada como caractere de continuação de linha se ela for seguida por uma quebra de linha introduzida por `[ENTER]` ou `[RETURN]`.

```
// Este é um comentário que \
   continua na próxima linha
```

Esta prática, no entanto, não apenas é desnecessária como também pode trazer alguns problemas. Portanto, seria mais recomendável escrever o comentário do último exemplo como:

```
// Este é um comentário que
// continua na próxima linha
```

Recomendações sobre uso prático de comentários serão apresentadas no **Capítulo 6**.

1.3 Operadores e Expressões

1.3.1 Operadores

Informalmente, um **operador** representa uma operação elementar da linguagem C. Esta operação é, dependendo da mesma, aplicada sobre um ou mais valores denominados **operandos**.

1.3.2 Expressões

Uma expressão é uma combinação legal de operadores e operandos. Aquilo que constitui uma combinação legal de operadores e operandos é precisamente definido para cada operando da linguagem C.

1.3.3 Propriedades dos Operadores

Além de produzirem valores como resultados de suas aplicações, operadores de C possuem várias outras propriedades. Como C é uma linguagem intensivamente baseada no uso de operadores, o entendimento destas propriedades é fundamental para a aprendizagem da própria linguagem. Portanto, certifique-se de que realmente entende todos os conceitos apresentados nesta seção antes de prosseguir e refira-se a ela quando tiver dificuldades em entender algum operador apresentado ao longo do texto.

As propriedades dos operadores são divididas em duas categorias: (1) propriedades que **todos os operadores possuem** e (2) propriedades que **alguns operadores possuem**. Estas propriedades são apresentadas a seguir.

▶ Propriedades que Todos os Operadores Possuem

Qualquer operador de C possui as propriedades apresentadas a seguir.

Resultado

O **resultado** de um operador é o valor resultante da aplicação do operador sobre seus operandos. Por exemplo: a aplicação do operador "+" na expressão "2 + 3" resulta em 5.

Aridade

A **aridade** de um operador é o número de operandos sobre os quais o operador atua. De acordo com esta propriedade, os operadores de C são divididos em três categorias:

- **Operadores unários** – são operadores que requerem um operando
- **Operadores binários** – são operadores que requerem dois operandos
- **Operador ternário** – é um operador que requer três operandos

O operador de soma (+), por exemplo, tem aridade 2 (i.e., ele é um operador binário).

Precedência

A **precedência** (ou **prioridade**) de um operador determina a ordem relativa com que ele é aplicado. Isto é, um operador de maior precedência é aplicado antes de um operador de menor precedência. Portanto, precedência é um conceito relativo entre operadores. Em C, operadores são agrupados em **grupos de precedência** que satisfazem as seguintes propriedades:

- Todos os operadores de um grupo de precedência possuem a mesma precedência.
- Operadores que pertencem a diferentes grupos de precedência possuem precedências diferentes.

Por exemplo, os operadores + e - (soma e subtração, respectivamente) fazem parte de um mesmo grupo de precedência, e o mesmo ocorre com os operadores * e / (multiplicação e divisão, respectivamente). No entanto, o grupo de multiplicação e divisão possui precedência maior do que o grupo de soma e subtração. Assim, na expressão `2*3 + 5`, o operador * é aplicado antes do operador +, pois aquele tem maior precedência que este.

Associatividade

Associatividade é um conceito semelhante ao de precedência no sentido de que ambos são utilizados para decidir a ordem de aplicação de operadores numa expressão. Entretanto, diferentemente do que ocorre com precedência, a associatividade é utilizada com operadores de mesma precedência (i.e., operadores que fazem parte de um mesmo grupo de precedência). Existem dois tipos de associatividade em C:

- **Associatividade à esquerda**: o operador à esquerda predomina sobre o operador da direita; i.e., o operador da esquerda é aplicado antes do operador da direita.
- **Associatividade à direita**: o operador à direita predomina sobre o operador da esquerda; i.e., o operador da direita é aplicado antes do operador da esquerda.

Por exemplo, na expressão "8/2/2" o primeiro operador é aplicado antes do segundo, pois o operador "/" têm associatividade à esquerda.

▶ Propriedades que Alguns Operadores Possuem

Além das propriedades apresentadas acima, *alguns operadores* possuem as propriedades adicionais apresentadas a seguir.

Efeito colateral

Esta propriedade é análoga ao efeito colateral (muitas vezes indesejável) provocado por um medicamento. Nesta analogia, o efeito principal do medicamento corresponde ao resultado produzido pelo operador, e o efeito colateral é alteração de valor produzido pelo operador em um de seus operandos. Entretanto, no caso de operadores com efeito

colateral, muitas vezes, o efeito colateral produzido pelo operador é o único proveito desejado. Em resumo, o efeito colateral de um operador consiste em alterar o valor de um de seus operandos; portanto, o operando sujeito a este efeito colateral deve ser uma variável. Um exemplo comum de operador com efeito colateral é o operador de atribuição, que já foi parcialmente apresentado na **Seção 1.2.5**.

Ordem de avaliação

A ordem de avaliação de um operador indica qual dos operandos sobre os quais ele atua é avaliado primeiro. Portanto, esta propriedade só faz sentido para operadores com aridade maior do que um (i.e., operadores binários e ternários). A maioria dos operadores binários de C não possui esta propriedade e este fato, muitas vezes, acarreta expressões capazes de produzir dois resultados válidos possíveis. Os operadores de conjunção **&&** e disjunção lógicas **||**, apresentados na **Seção 1.3.6**, são alguns dos poucos operadores que possuem ordem de avaliação definida.

▶ Uso de Parênteses

Pode-se alterar a ordem de aplicação de operadores numa expressão por meio do uso de parênteses. O compilador de C executa operações entre parênteses antes de qualquer outra operação independentemente de precedência ou associatividade[12].

Mesmo quando são desnecessários (i.e., redundantes para o compilador), parênteses também são muito úteis para tornar expressões complexas mais legíveis. É sempre uma boa idéia colocar expressões complexas entre parênteses, pois isto não apenas assegura que a expressão seja avaliada corretamente como também permite que a mesma seja lida sem a necessidade de referência a tabelas de precedência e associatividade.

[12] Obviamente, se existem vários operadores dentro de uma expressão entre parênteses, as regras de precedência e associatividade serão aplicadas para estes operadores para determinar a ordem de operações dentro dos parênteses.

É importante salientar que o uso de parênteses influencia apenas as propriedades de precedência e associatividade (e, por conseguinte, o resultado). Nenhuma outra propriedade de operadores sofre influência do uso de parênteses.

1.3.5 Operadores e Expressões Aritméticas

Uma **expressão aritmética** é uma combinação de operadores e operandos aritméticos que, quando avaliada, resulta num valor aritmético (i.e., numérico). Os operandos de uma expressão aritmética podem incluir variáveis, constantes ou chamadas de funções, que resultem num valor numérico. Os operadores aritméticos básicos de C são apresentados na **Tabela 8**.

OPERADOR	OPERAÇÃO
-	Menos unário (inversão de sinal)
+	Mais unário
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira

Tabela 8: Operadores aritméticos

A maioria dos operadores da **Tabela 8** funciona como em aritmética e em outras linguagens de programação, mas, em C, existem algumas diferenças. O operador + unário tem pouca utilidade prática e na maioria das vezes é redundante, uma vez que ele não produz nenhuma mudança de valor (embora ele possa causar conversão implícita, como será visto mais adiante).

Note que não existem símbolos distintos para denotar divisão inteira e de ponto-flutuante, como em algumas outras linguagens de programação (por exemplo, **div** e / em Pascal). Isto é, o símbolo / serve tanto para divisão inteira quanto para divisão de ponto-flutuante e o

compilador deduz do contexto se a divisão é inteira ou de ponto-flutuante. Ou seja, se ambos os operandos forem inteiros, a divisão será inteira; caso contrário, será de ponto-flutuante.

De acordo com o padrão C99, divisão e resto de divisão inteiros devem ter resultados truncados em direção a zero. Isto só influencia o resultado destas operações quando um ou ambos os valores envolvidos são negativos. Por exemplo, antes da homologação do padrão C99, a divisão e o resto da divisão de -17 por 5 poderia resultar em (truncamento em direção a zero):

```
-17/5 = -3
-17%5 = -2
```

ou (truncamento em direção a -∞):

```
-17/5 = -4
-17%5 = 3
```

pois, antes de C99, tudo que era requerido era que a equação `a = (a/b)*b + a%b` fosse satisfeita e, de fato, no exemplo acima, os pares de resultados satisfazem esta equação (verifique isto). Mas, de acordo com padrão C99, apenas o primeiro resultado é correto.

Operadores aritméticos são agrupados em grupos de precedência, conforme mostra a **Tabela 9**.

GRUPO DE OPERADORES	PRECEDÊNCIA
+, - (unários)	Mais alta
*, /, %	
+, - (binários)	Mais baixa

Tabela 9: Precedências de operadores aritméticos

Operadores num mesmo grupo na **Tabela 9** têm a mesma precedência. Quando, numa expressão, aparecem operadores de grupos de precedências diferentes, os operadores de mais alta precedência são

aplicados antes dos operadores de precedência mais baixa. Por exemplo, a expressão:

```
4 + 3*2
```

resulta em `10` (e não em `14`), uma vez que o operador * tem precedência maior do que o operador + e, portanto, é aplicado primeiro.

Dentre os operadores apresentados na **Tabela 9**, apenas os operadores unários são associativos à direita; todos os outros são associativos à esquerda. A associatividade à esquerda agrupa operadores de mesma precedência da esquerda para a direita (ou melhor, as operações são executadas da esquerda para a direita). Por exemplo, a expressão:

```
8/2/2
```

é interpretada como `(8/2)/2` [e não como `8/(2/2)`], uma vez que o operador / é associativo à esquerda e resulta, portanto, em `2` (e não em `8`). Por outro lado, a expressão:

```
- -2
```

seria interpretada como `-(-2)`, visto que o operador - (unário) é associativo à direita[13].

1.3.6 Operadores e Expressões Relacionais

Operadores relacionais são operadores binários utilizados em expressões de comparação. Os operandos de um operador relacional podem ser de qualquer tipo aritmético, exceto do tipo complexo. Existem seis operadores relacionais em C, que são apresentados juntamente com seus possíveis resultados, na **Tabela 10,** apresentada a seguir:

[13] Note que há um espaço em branco entre os dois traços na expressão acima. Isto evita que o compilador interprete os dois traços como o operador de decremento que será visto mais adiante. Um ponto similar vale para o mais unário: se mais de um operador "+" devem aparecer juntos, estes símbolos devem ser separados por um espaço em branco, para evitar confusão com o operador de incremento.

OPERADOR	DENOMINAÇÃO	APLICAÇÃO	RESULTADO
>	maior do que	**a > b**	1 se a é maior do b; 0, caso contrário
>=	maior do que ou igual a	a >= b	1 se a é maior do que ou igual a b; 0, caso contrário
<	menor do que	a < b	1 se a é menor do b; 0, caso contrário
<=	menor do que ou igual a	a <= b	1 se a é menor do que ou igual a b; 0, caso contrário
==	igual a	a == b	1 se a é igual a b; 0, caso contrário
!=	diferente de	a != b	1 se a é diferente de b; 0, caso contrário

Tabela 10: Operadores relacionais

Os quatro primeiros operadores na **Tabela 10** têm a mesma precedência, que é menor do que as precedências dos operadores aritméticos vistos antes. Os dois últimos operadores estão uma classe de precedência abaixo dos quatro primeiros operadores[14]. Em resumo, a **Tabela 9** exibe os grupos de precedência dos operadores relacionais.

GRUPO DE OPERADORES	PRECEDÊNCIA
>, >=, <, <=	Mais alta
==, !=	Mais baixa

Tabela 11: Precedências dos operadores relacionais

[14] O **Apêndice A** apresenta um quadro-resumo com todos os operadores da linguagem C, bem como suas propriedades de precedência e associatividade.

Outras linguagens de programação, tais como Pascal e Modula-2, possuem operadores relacionais equivalentes aos operadores relacionais de C apresentados na **Tabela 10**. Entretanto, uma diferença fundamental é que outras linguagens possuem o tipo de dado booleano e expressões de comparação resultam num dos valores deste tipo (i.e., `TRUE` ou `FALSE`). Apesar de o padrão C99 ter introduzido o tipo booleano em C, os resultados de expressões de comparação ainda são inteiros como eram quando a linguagem C foi criada.

Note ainda que o operador relacional de igualdade é representado pelo símbolo "==" e não por "=", como em Pascal e Basic. O programador deve se prevenir contra este tipo de engano, pois ele é uma fonte muito comum de erros de programação que não são apontados pelo compilador. Este erro é muitas vezes difícil de ser localizado, pois, diferentemente de Pascal, o uso por engano do operado = ao invés de == é *sintaticamente* legal em C, embora, freqüentemente, resulte numa interpretação diferente da pretendida. Um exemplo concreto dessa situação será apresentado mais adiante.

Outro cuidado que o programador deve tomar com o uso do operador de igualdade é que o mesmo não deve ser utilizado para comparar números de ponto flutuante, pois o computador pode armazenar estes números de forma aproximada. Por exemplo, a expressão relacional:

```
(2.0/3 + 2.0/3 + 2.0/3) == 2.0
```

usualmente não resulta em `1`, conforme seria esperado empregando um raciocínio matemático. Isto deve-se ao fato de cada termo entre parênteses resultar numa dízima infinita (`0.6666...`), que não pode ser completamente armazenada no computador. Por causa disso, a expressão entre parênteses poderia resultar, por exemplo, em `2.00000001`, que não é igual a `2.0`. Para contornar este problema, o programador poderia utilizar outro operador relacional (por exemplo, **>=** ou **<=**), em vez do operador de igualdade, quando comparar números de ponto flutuante.

1.3.7 Operadores e Expressões Lógicas

Operadores lógicos em C correspondem aos operadores de **negação** (`NOT`), **conjunção** (`AND`) e **disjunção** (`OR`) de algumas outras linguagens. Entretanto, diferentemente de outras linguagens e apesar de o padrão C99 ter introduzido o tipo booleano em C, os operadores lógicos em C podem receber como operandos quaisquer valores escalares (v. **Figura 1**). Os operadores lógicos de C são apresentados em ordem decrescente de precedência, na **Tabela 12**.

Operador	Símbolo	Precedência
Negação	!	Mais alta
Conjunção	&&	↓
Disjunção	||	Mais baixa

Tabela 12: Operadores lógicos em ordem decrescente de precedência

A aplicação de um operador lógico sempre resulta em 0 ou 1, dependendo dos valores dos seus operandos. Os valores resultantes da aplicação destes operadores de acordo com os valores de seus operandos são resumidos na **Tabela 13** e na **Tabela 14,** apresentadas a seguir.

X	!X
0	1
diferente de 0	0

Tabela 13: Resultados do operador !

X	Y	X && Y	X || Y
0	0	0	0
0	diferente de 0	0	1
diferente de 0	0	0	1
diferente de 0	diferente de 0	1	1

Tabela 14: Resultados dos operadores && e ||

Para memorizar a **Tabela 14**, você precisa apenas considerar que a aplicação de **&&** resulta em 1 apenas quando os dois operandos são diferentes de zero; caso contrário, o resultado é 0. De modo semelhante, `X || Y` resulta em 0 apenas quando ambos os operandos são iguais a 0.

O operador lógico **!** tem a mesma precedência dos operadores aritméticos unários + e –, vistos anteriormente. Os operadores lógicos **&&** e **||** têm precedências mais baixas do que operadores aritméticos e relacionais, mas não fazem parte de um mesmo grupo de precedência: o operador **&&** tem precedência mais alta do que o operador **||**. A **Tabela 15** apresenta as precedências relativas entre todos os operadores vistos até aqui.

GRUPO DE OPERADORES	PRECEDÊNCIA
!, +, - (unários)	Mais alta
*, /, %	↓
+, - (binários)	↓
>, >=, <, <=	↓
==, !=	↓
&&	↓
||	Mais baixa

Tabela 15: Precedências relativas entre operadores aritméticos, relacionais e lógicos

A ordem de avaliação de operandos dos operadores **&&** e **||** é especificada como sendo da esquerda para a direita. Além disso, o compilador não promove a avaliação de operandos que não sejam necessários para a determinação do valor de uma expressão lógica[15]. Por exemplo, quando o valor de `a` for zero na expressão:

```
(a != 0) && (b/a > 4.0)
```

[15] Este modo de avaliação de operandos numa expressão lógica é denominado **avaliação com curto-circuito**.

o operando `a != 0` resulta em 0 e o operando `b/a > 4.0` não é avaliado, pois considera-se que, para que toda a expressão resulte em 0, basta que um dos operandos seja 0.

1.4 Os Tipos _Bool e bool

O tipo **_Bool** é um tipo inteiro sem sinal introduzido pelo padrão C99. Variáveis deste tipo podem apenas assumir 0 ou 1. O padrão C99 também introduziu o arquivo de cabeçalho `<stdbool.h>` (v. **Volume II**) que define o tipo derivado **bool** (equivalente a **_Bool**) e as macros **true** (equivalente a 1) e **false** (equivalente a 0). Este tipo e estas macros permitem a escrita de expressões em C semelhantes a expressões booleanas de outras linguagens (por exemplo, C++ e Java), como, por exemplo:

```
#include <stdbool.h>
...
bool  b = false;
...
if (...)
   b = true;
```

Na prática, o tipo **bool** e as constantes **true** e **false** servem apenas para melhorar a legibilidade e permitir que trechos de programas escritos em C possam ser aproveitados em linguagens como C++ e Java.

1.5 Mistura de Tipos e Conversões Automáticas

C permite que tipos aritméticos sejam misturados em expressões. Entretanto, para que tais expressões façam sentido, o compilador executa **conversões automáticas** (ou **implícitas**). Estas conversões muitas vezes são responsáveis por resultados inesperados e, portanto, o programador deve estar absolutamente ciente das transformações feitas implicitamente pelo compilador antes de misturar tipos numa expressão aritmética; caso contrário, é melhor evitar definitivamente a mistura de tipos.

Existem cinco situações nas quais conversões são feitas implicitamente pelo compilador de C. Três delas serão discutidas a seguir, enquanto as demais serão discutidas na **Seção 3.3.5**.

1.5.1 Conversão de Atribuição

Neste tipo de conversão, o valor do lado direito de uma atribuição é convertido para o tipo da variável do lado esquerdo. O problema que pode ocorrer com este tipo de conversão surge quando o tipo do lado esquerdo é *mais curto* do que o tipo do lado direito da atribuição. Por exemplo, se `c` é uma variável do tipo **unsigned char**, a atribuição:

```
c = 936;
```

pode resultar, na realidade, na atribuição do valor `168` a `c`. Por que isto ocorre? Primeiro, o valor `936` é grande demais para caber numa variável do tipo **char** (que ocupa apenas um byte). Segundo, pode-se observar que `936` é representado em 16 bits pela seqüência:

```
00000011  10101000
```

Mas, como valores do tipo **char** devem ser contido em apenas 8 bits, o compilador considera apenas os 8 bits de menor ordem:

```
10101000
```

que equivalem a `168`.

Qualquer valor de um tipo escalar atribuído a uma variável do tipo **_Bool** que não seja deste tipo é convertido para 0 quando o valor é 0 e 1 quando o valor é diferente de 0. Por exemplo:

```
_Bool   b = 0;
double  x = 0.0;
long    y = -3;

b = x; // b recebe 0
b = y; // b recebe 1
```

Quando um valor de um tipo complexo é convertido para um tipo de ponto-flutuante, a parte imaginária desaparece. Quando um valor de um tipo de ponto-flutuante é convertido para um valor de tipo complexo, a parte imaginária é zero.

1.5.2 Conversão de Alargamento (Promoção)

Quando aparecem numa expressão, valores dos tipos **signed char**, **short** e **_Bool** são convertidos para **int**, enquanto valores dos tipos **unsigned char** e **unsigned short** são convertidos para **int** (se couberem) ou **unsigned int**. Este tipo de conversão é denominado **alargamento** (ou **promoção**) e é inevitável, mas não produz nenhum efeito indesejável[16].

1.5.3 Conversão Aritmética Usual

Quando variáveis e constantes de tipos diferentes aparecem numa expressão, elas são convertidas de modo que um dado operador tenha como operandos valores do mesmo tipo. Esta conversão, denominada **conversão aritmética usual,** no padrão C99, obedece à hierarquia de tipos apresentada na **Figura 2,** a seguir.

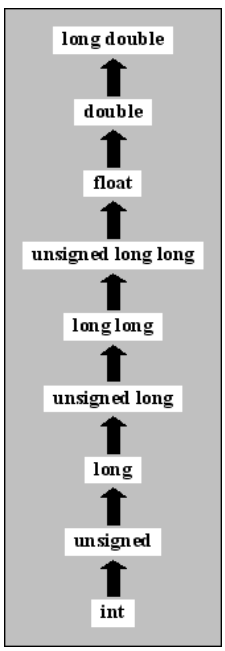

Figura 2: Hierarquia de tipos aritméticos

[16] A promoção de inteiros é especificada mais precisamente pelo padrão C99, conforme apresentado no Volume II, mas a descrição apresentada aqui é suficiente para propósitos práticos.

Na hierarquia representada na **Figura 2**, a seta deve ser interpretada como "*é convertido para*". Por exemplo, se numa expressão com dois operandos de tipos diferentes um deles é do tipo **long double**, o outro operando é convertido para **long double**. Em outras palavras, a hierarquia de tipos acima deve ser entendida da seguinte maneira:

Sempre que um operador tem dois operandos de tipos diferentes, o operando cujo tipo é de mais baixa hierarquia é convertido para o tipo de mais alta hierarquia.

Por exemplo, se `x` é do tipo **int** e `y` é do tipo **float** na expressão abaixo:

```
x/y
```

então, o valor da variável `x` será convertido automaticamente para **float**.

Deve-se notar que os tipos **short**, **char** e **_Bool** não aparecem na hierarquia de tipos da **Figura 2**. Isto deve-se ao fato de as conversões de promoção de inteiros discutidas anteriormente ocorrerem antes das conversões que utilizam esta hierarquia.

Os tipos complexos não aparecem na **Figura 2** por simplicidade, pois, afinal, estes tipos não são tão comumente utilizados quanto os demais tipos. Quando for o caso, conversões aritméticas entre tipos complexos são as mesmas que ocorrem com os respectivos tipos reais. Por exemplo, as partes real e imaginária de um valor do tipo **float _Complex** estão sujeitas às mesmas conversões que um valor real do tipo **float**. Além disso, numa expressão, se um dos operandos é complexo, o resultado é complexo e, se for o caso, o operando que possui tipo de menor capacidade é promovido para o tipo do outro operando. Por exemplo, a soma de um valor do tipo **double _Complex** com um valor do tipo **float _Complex**, faz com que o valor do tipo **float _Complex** seja convertido em **double _Complex** e o resultado da soma seja deste último tipo.

1.5.4 Problemas Causados por Conversões Automáticas

Conforme foi antecipado, conversões implícitas podem gerar alguns resultados inesperados. A seguir serão apresentados alguns exemplos de conversões inusitadas.

Quando números de ponto flutuante são convertidos numa atribuição, dois problemas podem acontecer: (1) **perda de precisão** e (2) **perda por excesso** (*overflow*). Ambos os problemas ocorrem quando se atribui a uma variável um valor de um tipo com capacidade de armazenamento maior do que a capacidade de armazenamento da variável. No primeiro caso, pode ser que o tipo da variável, apesar de ser capaz de representar a ordem de grandeza do número atribuído, não seja capaz de conter todos os algarismos significativos do tipo maior. Por exemplo, suponha que, numa dada implementação de C, o tipo **double** seja capaz de representar 10 casas decimais, enquanto que o tipo **float** seja capaz de representar apenas seis casas decimais. Então, haverá *arredondamento* para que o número possa ser contido no tipo menor. Por exemplo, se `d` é **double** e contém o valor `4.1021456789654` e `f` é **float**, então após atribuição:

```
f  = d
```

`f` conterá o valor `4.10214567`, Havendo, portanto, uma perda de precisão. Se não houver problema com esta perda de precisão, estará tudo bem; caso contrário, você deve declarar o tipo de `f` como sendo **double**.

O segundo problema ocorre quando o número sendo atribuído é muito grande para ser contido no tipo menor. Por exemplo, suponha que o tipo **double** possa conter um número com ordem de grandeza de 308 (i.e., `10E308`) e que o tipo **float** possa conter números com ordem de grandeza de no máximo 38. Então, uma atribuição:

```
f = 2.4E95
```

acarretaria erro em tempo de execução do programa[17].

[17] Compiladores de C padrão não fazem arredondamento ou truncamento em casos como este (v. detalhes no **Volume II**).

Conversões implícitas entre números de ponto flutuante e inteiros também podem acarretar problemas. Claramente, a atribuição de um número de ponto flutuante a um inteiro pode causar perda por excesso, pois números de ponto flutuante são usualmente capazes de conter números com ordens de grandeza maiores do que o permitido para inteiros. Também, apesar de os tipos de ponto flutuante terem usualmente maiores capacidades de armazenamento do que os tipos inteiros, a atribuição de um inteiro a um número de ponto flutuante pode causar perda de precisão, pois o número de ponto flutuante poderá não ser capaz de representar todos os algarismos significativos do inteiro. Neste caso, haverá arredondamento. Por exemplo, se `f` é do tipo **float**, a atribuição:

```
f = 556678558985867854L;
```

poderá causar a atribuição de `556678545254907900.0` a `f`.

Como exemplo final do que pode acontecer de inusitado com conversões implícitas, considere a seguinte expressão:

```
12u - 20
```

O resultado esperado para a expressão acima certamente seria `-8`, mas isto pode não ocorrer na realidade. Como a primeira constante (`10u`) é do tipo **unsigned** e a segunda constante (`15`) é do tipo **int**, esta última será implicitamente convertida para **unsigned int** e o resultado será também interpretado como **unsigned int**. Isto é, o resultado da expressão será a representação do valor `-8` interpretado como **unsigned**. Supondo uma representação em 32 bits com complemento de dois para o tipo **int**, `-8` seria representado como:

```
11111111 11111111 11111111 11111000
```

Se esta seqüência de bits for interpretada como **unsigned int**, ter-se-á, em vez de `-8`, um enorme número inteiro:

```
4294967288
```

1.6 Outros Operadores

1.6.1 Conversão Explícita

O programador pode especificar conversões explicitamente em C. Para fazer uma conversão explícita de um tipo em outro, deve-se antepor o valor que se deseja transformar pelo nome do tipo desejado entre parênteses. Por exemplo, suponha que se tenham as seguintes linhas de programa em C:

```
int   i1 = 3, i2 = 2;
float f;

f = i1/i2;
```

Neste caso, devido a conversões implícitas, `f` receberá o valor `1.0`, o que talvez não fosse o esperado. Entretanto, se um dos inteiros for promovido explicitamente a **float**, o resultado será `1.5` (o que talvez fosse mais esperado). Isto pode ser feito do seguinte modo:

```
f = (float) i1/i2;
```

A construção **(*tipo*)** [como **(float)** acima] trata-se de mais um operador em C, denominado **operador de conversão explícita** (ou **operador de *casting***). O operador de *casting* tem a mesma precedência dos outros operadores unários. Portanto, a expressão anterior é interpretada como:

```
((float) i1)/i2
```

Observe que, se toda a expressão `i1/i2` for colocada entre parênteses, a conversão será feita, de forma redundante, sobre o resultado da expressão, e o resultado será novamente `1.0`. O uso de *casting*, mesmo quando desnecessário, melhora a legibilidade dos programas. Por exemplo, suponha que `d` é do tipo **double** e `i` é do tipo **long int**. Então, na atribuição:

```
i = (long int) d;
```

o operador **(long int)** é utilizado apenas para enfatizar que ocorre uma conversão para **long int**. Funcionalmente, este operador é redundante, uma vez que esta transformação ocorreria implicitamente se o operador não fosse incluído. No entanto, o uso do operador torna clara esta conversão.

1.6.2 O Operador de Atribuição

A instrução de atribuição foi apresentada na **Seção 1.2.5**, mas não foi mencionado que, na verdade, uma instrução de atribuição representa uma expressão e que o sinal de igualdade utilizado em atribuição representa o operador principal desta expressão. Este operador faz parte de um grupo de operadores que têm uma das mais baixas precedências dentre todos os operadores de C[18]. Este operador possui efeito colateral que consiste exatamente na alteração de valor causada na variável do lado esquerdo da expressão, conforme foi descrito na **Seção 1.2.5**.

O grupo de operadores a que pertence o operador de atribuição possui associatividade à direita e o resultado da aplicação deste operador é o valor recebido pela variável no lado esquerdo da expressão de atribuição[19]. Devido às suas propriedades, o operador de atribuição pode ser utilizado para a execução de múltiplas atribuições numa única linha de instrução. Por exemplo, se `x`, `y` e `z` são do tipo **int**, a atribuição composta:

```
x = y = z = 1;
```

resulta na atribuição de `1` a `z`, `z` a `y` e `y` a `x`, nesta ordem. Neste caso, `x`, `y` e `z` terão, no final da execução da instrução, o mesmo valor, mas isto nem sempre acontece numa atribuição múltipla, pois podem ocorrer conversões implícitas. Considere o seguinte exemplo:

---

[18] De fato, apenas o operador vírgula possui precedência menor do que este grupo de operadores.

[19] Cuidado: alguns textos afirmam que o valor resultante é o resultado da avaliação do operando direto da expressão, o que é verdade apenas quando não ocorre conversão de atribuição (v. **Seção 1.5**)

```
int      j;
double   d;

j = d = 2.5;
```

Neste exemplo, `d` recebe o valor `2.5`, mas `j` recebe o valor `2` (o valor de `d` convertido para **int**). (**Exercício**: Que valores receberiam `d` e `j` na atribuição `d = j = 2.5`?)

Como a maioria dos operadores de C, o operador de atribuição não possui ordem de avaliação de operandos definida. Isto, aliado ao fato de o operador de atribuição possuir efeito colateral, pode dar origem a expressões capazes de produzir dois resultados aceitáveis. Por exemplo, a expressão:

```
(x = 2)*(x + 1)
```

pode produzir dois resultados diferentes, dependendo de qual dos operandos [`(x = 2)` ou `(x + 1)`] é avaliado primeiro. Isto é, supondo que a variável `x` é **int** e foi iniciada com 0, se o primeiro operando for avaliado antes do segundo, o resultado será 6, enquanto se o segundo operando for avaliado antes do primeiro, o resultado será 2. Qualquer dos dois é válido porque o padrão da linguagem C não especifica qual dos dois operandos deve ser avaliado primeiro. Portanto, a expressão `(x = 2)*(x + 1)` não é portável em C.

É importante salientar que, de acordo com a **Seção 1.2.5**, atribuição é uma instrução; aqui, nesta seção, também foi visto que atribuição é uma expressão. Em geral, qualquer expressão pode ser utilizada como uma instrução em C.

1.6.3 Operadores de Atribuição Aritmética

Existem cinco outros operadores de atribuição que combinam operações aritméticas com atribuição em operações únicas. Estes operadores, denominados de **operadores de atribuição aritmética**, e seus equivalentes funcionais são apresentados na **Tabela 16**.

OPERADOR	EQUIVALENTE A
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

Tabela 16: Operadores de atribuição aritmética

Todos os operadores de atribuição (incluindo =) fazem parte de um mesmo grupo de precedência. Os operadores deste grupo têm precedência baixíssima e associatividade da direita para a esquerda.

Levando em conta a baixa precedência dos operadores de atribuição, a expressão:

```
j *= 2 + 3
```

seria interpretada como:

`j = j*(2 + 3)` (e não como `j = j*2 + 3`).

O uso de operadores de atribuição aritmética apresenta algumas vantagens, como:

- Algumas vezes, eles tornam o programa mais legível e menos sensível a erros de digitação, principalmente quando o identificador do lado esquerdo da atribuição apresenta grande comprimento.
- Alguns computadores possuem instruções de máquina capazes de executar estas operações diretamente, de modo que o código escrito desta maneira torna-se mais eficiente.

1.6.4 Operadores de Incremento e Decremento

Os operadores de **incremento** e **decremento** são operadores unários que, quando aplicados a uma variável, adicionam-lhe ou subtraem-lhe 1, respectivamente. Estes operadores aplicam-se a variáveis numéricas

ou do tipo ponteiro. Os operadores de incremento e decremento são representados respectivamente pelos símbolos **++** e **--** e têm a mesma precedência de todos os operadores unários.

Existem duas versões para cada um destes operadores: (1) **prefixa** e (2) **sufixa**. Esta classificação refere-se à posição do operador com relação ao operando. Se o operador aparece antes do operando (por exemplo, `++x`), ele é um operador prefixo; caso contrário (por exemplo, `x++`), ele é sufixo. Todos estes operadores produzem efeitos colaterais nas variáveis sobre as quais atuam. Estes efeitos colaterais correspondem exatamente ao incremento (i.e., acréscimo de 1 ao valor) ou decremento (i.e., subtração de 1 do valor) da variável.

Com relação a efeitos colaterais, não existe diferença quanto ao uso da forma prefixa ou sufixa de cada um desses operadores. Isto é, qualquer um dos dois operadores de incremento produz o mesmo efeito de incremento e qualquer um dos dois operadores de decremento produz o mesmo efeito de decremento.

A diferença entre as versões prefixa e sufixa de cada um desses operadores está no resultado da operação. Os operadores sufixos produzem como resultado o *próprio valor da variável* sobre a qual atuam; por outro lado, operadores prefixos resultam no valor da variável *após o efeito colateral* (i.e., incremento ou decremento) ter ocorrido. Portanto, se o resultado de uma operação de incremento ou de decremento não for utilizado, não faz nenhuma diferença se o operador utilizado é prefixo ou sufixo. Por exemplo, no trecho de programa a seguir:

```
int  y, x = 2;

y = 5*x++;
```

`y` recebe o valor `10`, enquanto se a instrução de incremento fosse:

```
y = 5*++x
```

`y` receberia o valor `15`. Em ambos os casos, entretanto, a variável `x` seria incrementada para `3` (i.e., o efeito colateral seria o mesmo).

Supondo que `x` é uma variável de um tipo escalar, os resultados e efeitos colaterais dos operadores de incremento e decremento são resumidos na **Tabela 17**.

Operação	Denominação	Valor da Expressão	Efeito Colateral
x++	incremento sufixo	o mesmo de x	adiciona 1 a x
x--	decremento sufixo	o mesmo de x	subtrai 1 de x
++x	incremento prefixo	o valor de x mais 1	adiciona 1 a x
--x	decremento prefixo	o valor de x menos 1	subtrai 1 de x

Tabela 17: Operadores de incremento e decremento

Muitas vezes, o programador está interessado apenas no efeito colateral de um operador de incremento ou de decremento. O programador deve preocupar-se com a diferença entre operadores prefixo e sufixo apenas quando os valores resultantes de expressões formadas por estes operadores forem utilizados.

Do mesmo modo como ocorre com operadores de atribuição, não é recomendável o uso de uma variável afetada por um operador de incremento ou decremento na mesma expressão em que ocorre tal efeito. Por exemplo, considere o seguinte trecho de programa:

```
int  i, j = 4;

i = j * j++;
```

Conforme foi visto antes, a linguagem C não especifica qual dos operandos da multiplicação é avaliado primeiro e, portanto, o resultado a ser atribuído a `i` irá depender da interpretação do compilador.

O programador também deve evitar (ou, pelo menos, tomar bastante cuidado com) o uso de operadores com efeitos colaterais, como os operadores de incremento e decremento, com os operadores lógicos **&&** e **||**. O problema é que, conforme visto na **Seção 1.3.6**, o compilador nem sempre avalia completamente algumas expressões lógicas. Considere o seguinte exemplo:

```
(a > b) && (c == d++)
```

Nesta situação, a variável `d` seria incrementada apenas quando `a` fosse maior do que `b` e talvez o programador desejasse que a mesma variável fosse incrementada *sempre* que esta instrução fosse executada.

1.6.5 O Operador sizeof

O operador **sizeof** é um operador unário, prefixo e de precedência e associatividade iguais às dos outros operadores unários, e com associatividade da direita para a esquerda. Este operador pode receber como operando um tipo de dado ou uma expressão.

Quando aplicado a um tipo de dados, o operador **sizeof** resulta no número de bytes necessários para alocar uma variável daquele tipo. Neste caso, o operando deve ser colocado entre parênteses. Por exemplo,

```
size_t tamanhoDoTipoDouble = sizeof(double);
```

resulta na iniciação da variável `tamanhoDoTipoDouble` com o número de bytes necessários para conter uma variável do tipo **double** na implementação de C utilizada.

Quando aplicado a uma expressão, o operador **sizeof** resulta no número de bytes que seriam necessários para conter o resultado da expressão se a mesma fosse avaliada. A expressão em si *não é avaliada*. Quando o operando do operador **sizeof** é uma expressão, a mesma não precisa ser colocada entre parênteses, porém o uso destes é sempre recomendável.

Segundo o padrão ISO, o resultado do operador **sizeof** deve ser do tipo **size_t**, que é um tipo inteiro sem sinal (por exemplo, **unsigned long**) definido na biblioteca padrão de C.

1.7 Estruturas de Controle

Normalmente, um programa é executado seqüencialmente, da primeira à última instrução. Usualmente, entretanto, o fluxo normal de

execução de um programa é alterado por meio de **estruturas de controle** que provocam desvios e repetições de certas instruções. As estruturas de controle em C podem ser classificadas em três categorias:

- **Desvios condicionais** que permitem decidir, por meio da avaliação de uma condição, se uma porção do programa será executada ou não.
- **Desvios incondicionais** que indicam incondicionalmente qual instrução será executada em seguida.
- **Repetições** (ou **iterações**) que permitem a execução de uma ou mais instruções repetidamente até que uma condição seja satisfeita.

Antes de apresentar as estruturas de controle de C, serão apresentadas algumas considerações importantes sobre instruções em C. Em primeiro lugar, é necessário salientar que toda instrução ou declaração em C *termina* com ponto-e-vírgula.

1.7.1 Seqüências de Instruções

Uma **seqüência de instruções** consiste em mais de uma instrução e pode ser inserida em qualquer local em um programa onde uma única instrução é permitida. Seqüências de instruções devem ser colocadas entre chaves (i.e., entre "{" e "}"). Uma seqüência de instruções é também conhecida por **bloco** e pode conter, além de instruções, definições de variáveis e tipos (v. **Seção 4.6**). Quando ocorrem num bloco, essas definições são conhecidas como **locais** ao bloco e têm validade apenas em seu interior.

O padrão C99 permite que instruções e declarações apareçam em qualquer ordem num bloco desde que sejam respeitadas as regras de escopo vigentes. Por exemplo, o bloco esquematizado a seguir é perfeitamente legal:

```
{
   ...
   int i;
   i = ...;
   ...
}
```

Enquanto isso, o bloco ligeiramente diferente do anterior apresentado a seguir é ilegal:

```
{
   ...
   i = ...;
   int i;
   ...
}
```

A ilegalidade apresentada no último exemplo é decorrente do fato de a variável `i` estar sendo usada antes de sua definição (i.e., fora de seu escopo) e não pelo fato de sua definição suceder uma instrução. No primeiro exemplo, isto não ocorre e, portanto, não há ilegalidade.

Blocos podem ainda ser aninhados dentro de outros blocos, mas o aninho de mais de um nível não é recomendado, pois prejudica a legibilidade do programa.

1.7.2 Instruções Vazias

Um aspecto interessante da linguagem C é que ela permite a escrita de **instruções vazias** (i.e., que não executam nenhuma tarefa) em qualquer local onde pode-se colocar uma instrução normal. Uma instrução vazia em C é representada por "**;**" (ponto-e-vírgula).

É sempre uma boa idéia colocar instruções vazias em linhas separadas e acompanhadas de comentários explicativos. Esta recomendação para escrita de instruções vazias evita que instruções deste tipo que são *propositais* sejam confundidas com aquelas que são *acidentais* (i.e., erros de programação). Como, isoladamente, "*;*" significa a instrução vazia, a colocação acidental deste símbolo em locais onde espera-se uma instrução normal será interpretada pelo compilador como uma construção válida, mas pode ser que isto não seja o desejado pelo programador (um exemplo disto será apresentado na seção a seguir).

1.7.3 Laços de Repetição: while, do-while e for

Laços de repetição permitem controlar o número de vezes que uma instrução ou uma seqüência de instruções é executada. A seguir as estruturas de repetição de C serão detalhadas.

► Laço de Repetição while

A instrução **while** (ou **laço while**) é uma estrutura de repetição que tem o seguinte formato:

while (*expressão*)
instrução;

A expressão entre parênteses é uma condição de teste e é muitas vezes (mas não necessariamente) uma expressão relacional. A instrução endentada na linha seguinte no esquema acima é denominada de **corpo do laço while** e pode ser representada por uma seqüência de instruções entre chaves.

A instrução **while** é interpretada conforme descrito a seguir. A expressão entre parênteses é avaliada e, se o resultado desta expressão for diferente de zero, o corpo do laço será executado. Então, o controle do programa retorna para o topo da instrução **while** e o processo é repetido até que a expressão resulte em zero. Quando isto ocorre, o controle passa para a instrução que segue toda a instrução **while**. Note que, se inicialmente a expressão resultar em zero, o corpo do laço não será executado nenhuma vez.

Como exemplo de uso da instrução **while** considere:

```
long   x = 0L, y = 10L;

while (x < y) {
       x++;
       y--;
}
```

Cuidado para não se esquecer de colocar as chaves em torno de uma seqüência de instruções; caso contrário, apenas a primeira instrução será considerada como sendo o corpo do **while**. Por isso, mesmo quando o

corpo da instrução **while** é constituído por uma única instrução, é sempre uma boa idéia colocá-lo entre chaves. Isto evita que você esqueça de incluí-las se, por acaso, quiser acrescentar alguma instrução ao corpo do **while**. Esta última recomendação é válida para outras estruturas de controle e não apenas para o laço **while**. Também deve-se tomar cuidado para não escrever "`;`" após a primeira linha de uma instrução **while**, pois, assim, o corpo da mesma será interpretada como sendo a instrução vazia. Por exemplo:

```
while (x);
x--;   /* Esta instrução será executada exatamente uma vez */
       /* se o valor de x for 0 e o programa entrará em laço */
       /* sem fim se o valor de x for diferente de 0 */
```

No exemplo acima, o programador, muito provavelmente, pretendia que o corpo do laço fosse `x--` (conforme sugerido pela endentação), mas, na realidade, o corpo será a instrução vazia (i.e., o ponto-e-vírgula ao final da primeira linha). A recomendação de precaução para não escrever ponto-e-vírgula em locais indevidos vale para outras estruturas de controle vistas a seguir.

► Laço de Repetição do-while

A instrução **do-while** é outra estrutura de repetição de C cuja sintaxe segue o esquema a seguir:

do
    instrução;
while (*expressão*);

Como na instrução **while** vista antes, *instrução* representa o corpo do laço e *expressão* é uma expressão que, quando resulta em zero, encerra o laço. Em termos de interpretação, a única diferença entre **while** e **do-while** é o ponto onde a condição de teste é avaliada. Na instrução **while**, a condição é avaliada no início do laço, enquanto na instrução **do-while** a condição é avaliada ao final. Por causa disso, garante-se que o corpo

de uma instrução **do-while** seja executado pelo menos uma vez. Assim, a instrução **do-while** é indicada para situações nas quais deseja-se que o corpo do laço seja executado pelo menos uma vez.

Exemplo de uso da instrução **do-while**:

```
long  x = 0L, y = 10L;

do {
  x++;
  y--;
} while (x < y);
```

► Laço de Repetição for

A instrução **for** é um pouco mais complicada do que a equivalente de algumas outras linguagens de programação. A sintaxe da instrução **for** segue este esquema:

for (*expressão1*; *expressão2*; *expressão3*)
instrução;

Qualquer das expressões entre parênteses é opcional, mas usualmente todas as três são utilizadas. Uma instrução **for** é interpretada conforme a seguinte seqüência de passos:

1. *expressão1* é avaliada. Usualmente, esta é uma ou expressões de atribuição para uma ou mais variáveis.

2. *expressão2*, que é a expressão condicional da estrutura **for**, é avaliada.

3. Se *expressão2* resultar em zero, a instrução **for** é encerrada e o controle do programa passa para a instrução seguinte a toda instrução **for**. Se *expressão2* resultar num valor diferente de zero, o corpo do laço (representado por *instrução* no esquema acima) é executado.

4. Após a execução do corpo do laço (se for o caso, de acordo com o passo 3), *expressão3* é avaliada e retorna-se ao passo 2.

Pode-se facilmente verificar que a instrução **for** é equivalente em termos funcionais à seguinte seqüência de instruções[20]:

```
expressão1;
while (expressão2){
    instrução;
    expressão3;
}
```

Apesar desta equivalência, este conjunto de instruções não é indicado (por questão de legibilidade) para substituir instruções **for** onde a utilização desta instrução parece ser a escolha mais natural. Em particular, a instrução **for** torna mais fácil verificar as mudanças de valor de variáveis usualmente contidas em *expressão3*.

Embora seja um pouco mais complexa do que a instrução **for** de outras linguagens, a instrução **for** de C é mais freqüentemente utilizada em laços de contagem como a instrução **for** dessas linguagens. Por exemplo:

```
for  (j = 1; j <= 10; j++) {

    /* Seqüência de instruções a ser    */
    /* executada 10 vezes repetidamente */
}
```

Um erro comum em instruções como a apresentada no último exemplo é executar o corpo do **for** um número de vezes diferente do pretendido devido ao uso de um operador relacional inadequado (por exemplo, utilizar < em vez de <= ou vice-versa). Por exemplo, se a condição de teste utilizada no exemplo anterior fosse `j < 10`, em vez de `j <= 10`, o corpo do **for** seria executado apenas 9 vezes (e não 10 vezes, como no exemplo). Este tipo de erro não é indicado pelo compilador e é muitas vezes difícil de ser detectado. A melhor forma de prevenir este tipo de erro é testar cada laço de contagem até convencer-se de que ele realmente funciona conforme o esperado.

[20] Existe uma exceção para esta eqüivalência. Se houver uma instrução continue dentre as instruções que constituem o corpo do laço for, essa instrução causará o desvio para <*expressão3*>, enquanto no caso do laço while o desvio seria feito para o topo deste laço (i.e., para <*expressão2*>).

Conforme foi mencionado antes, qualquer das expressões entre parênteses pode ser omitida numa instrução **for**. Entretanto, os dois pontos-e-vírgulas devem sempre ser incluídos. Na prática, é comum omitir-se *expressão1* ou *expressão3*, mas não ambas ao mesmo tempo[21]. Normalmente, *expressão2* é incluída, pois trata-se da condição de teste. Quando esta condição de teste é omitida, ela é considerada como se fosse igual a 1.

Pode-se também utilizar uma instrução nula como corpo de um laço **for**. Este tipo de construção é utilizado quando a tarefa desejada é executada pelas próprias expressões entre parênteses.

Em C99, é permitido o uso de definições de variáveis no lugar da expressão identificada na sintaxe de **for** como *expressão1*. Variáveis declaradas desta maneira têm validade apenas no laço for correspondente. Por exemplo:

```
for (int i = 0; i < 10; ++i) {
   ...
}

i = 0; // Erro: a variável i não é mais válida aqui
```

Neste caso, o escopo da variável é o interior do laço **for** no qual a mesma é definida e, portanto, o laço do último exemplo é equivalente a:

```
{
   int i = 0
   for (; i < 10; ++i) {
      ...
   }
{
```

▶ Laços de Repetição Infinitos

Laços de repetição infinitos são aqueles cujos corpos são executados indefinidamente. Algumas vezes, um laço de repetição infinito é aquilo que

[21] Omitir simultaneamente <*expressão1*> e <*expressão3*> torna a expressão **for** eqüivalente a uma única instrução **while** (v. relação entre **for** e **while** apresentada anteriormente).

realmente o programador deseja, mas, em outras, estas instruções contêm erros de programação que as impedem de terminar apropriadamente.

Repetição infinita é provocada por dois tipos de laços de repetição:

(1) laço de repetição que não contém uma condição de término; e

(2)laço de repetição que contém uma condição de término que nunca é atingida.

Como exemplos de laços de repetição infinitos do tipo (1), têm-se, esquematicamente:

```
while (1)
   instrução;
```

e

```
for ( ; ; )
   instrução;
```

Estas duas instruções são equivalentes e constituem jargões (v. **Seção 6.7**) comumente utilizados para indicar repetição infinita intencional.

Como exemplo de laço de repetição infinito do tipo (2) tem-se o seguinte:

```
for (i = 1; i <= 10; i++){
   ...
   i = 2;
}
```

Certamente a instrução **for** do último exemplo contém um erro de programação porque ela possui um teste (`i <= 10`) que *sempre* é satisfeito, pois sempre que esta condição é testada, `i` é menor do que 10 (verifique isso). Portanto, a condição (*implícita*) de parada (`i > 10`) jamais será atingida.

1.7.4 Desvios Condicionais: if-else e switch-case

Instruções de desvios condicionais permitem o desvio do fluxo de execução do programa para outras partes do programa dependendo do

resultado da avaliação de uma expressão (**condição**). Essas instruções serão examinadas a seguir.

▶ Instrução *if-else*

A principal instrução condicional em C é a instrução **if-else,** que tem a seguinte sintaxe:

if (*expressão*)
    instrução1;
else
    instrução2;

Como em outras linguagens, a parte **else** é opcional. A interpretação da instrução **if** é a seguinte: a expressão entre parênteses é avaliada; se o resultado da mesma for diferente de zero, *instrução1* será executada; caso contrário, se houver uma parte else, *instrução2* será executada. As instruções *instrução1* e *instrução2* podem, conforme visto anteriormente, ser substituídas por seqüências de instruções.

Exemplos de usos da instrução **if**:

```
if (x)
   y++;       /* Executada quando x for diferente de zero*/
else
   y--;       /* Executada quando x for igual a zero */
z = x + y;    /* Executada sempre        */


if (x)
   y++;     /* Executada quando x for diferente de zero */
y--;         /* Executada sempre */
```

As endentações nos exemplos acima refletem relações de dependência entre instruções. Por exemplo, no primeiro caso acima, as instruções `y++` e `y--` pertencem ao **if** e são endentadas com relação a esta última instrução, enquanto a instrução `z = x + y` é independente do **if** e é escrita no mesmo

nível desta instrução. Endentação serve ao único propósito de melhorar a legibilidade do programa e não faz a menor diferença para o compilador. Para ilustrar este ponto, suponha, por exemplo, que você deseja executar as instruções *instrução1* e *instrução2* apenas quando `x` é diferente de zero na instrução a seguir:

```
if (x)
   instrução1;   /* Executada quando x for diferente de zero */
   instrução2;  /* Executada sempre  */
```

Apesar de a endentação indicar uma dependência ilusória de *instrução2* com o **if**, na realidade esta instrução será sempre executada, independentemente do valor de `x`. Para executar ambas as instruções acima quando `x` é diferente de zero, você teria que juntá-las numa seqüência de instruções como:

```
if (x) {
   instrução1;
   instrução2;
}
```

Deve-se ainda chamar a atenção para o perigo representado pela troca, por engano, do operador relacional de igualdade **==** pelo operador de atribuição **=**. Como exemplo, considere o seguinte:

```
int x = 0;

if  (x = 10)
   y += 1;     /* Esta instrução será sempre executada  */
```

A instrução acima contém, com certeza, um erro de programação, pois, como a expressão `x = 10` resulta sempre em 10 (por que?), a instrução `y += 1` será sempre executada. Se esta fosse realmente a intenção do programador, não faria sentido escrever uma instrução **if**.

Para prevenirem-se contra esse tipo de erro, alguns programadores adotam a disciplina de escreverem sempre expressões de igualdade, como aquela do último exemplo, com a constante do lado esquerdo. Assim, se

o operador de igualdade for acidentalmente trocado pelo operador de atribuição, o compilador indicará o erro. Por exemplo:

```
int x = 0;

if  (10 = x) /* O compilador indicará um erro nesta instrução */
   y += 1;
```

Instruções **if** podem ser aninhadas de modo a representar desvios múltiplos. Uma instrução **if** é aninhada quando a instrução seguindo o **if** ou a instrução seguindo o **else** também é uma instrução **if**. Por exemplo, a instrução a seguir é uma instrução **if** aninhada:

```
int a, b, c, x;
   ...
if (a < b)
   if  (a < c)
      x = a;
   else
      x = c;
else  if  (b < c)
   x = b;
else
   x = c;
```

A execução da instrução **if** aninhada do exemplo acima atribuirá a `x` o menor valor entre os números `a`, `b` e `c` (convença-se de que realmente entende isto).

Note que, quando um **if** segue imediatamente um **else**, coloca-se o **if** na mesma linha do **else** (i.e., não há endentação como seria o caso se a instrução seguindo o **else** fosse de outro tipo que não **if**). Obviamente, isto não é obrigatório, mas melhora a legibilidade de instruções **if** aninhadas.

Um aspecto importante em instruções aninhadas e que acarreta algumas vezes erro de programação é o de casamento de cada **else** com o respectivo **if**. Especificamente no último exemplo acima, o primeiro **else** casa com o segundo **if**; o segundo **else** casa com o primeiro **if** e o terceiro **else** casa com o terceiro **if** (escrito como **else if**). Em geral, deve-se adotar a seguinte regra de casamento:

Um else está sempre associado ao if mais próximo que não tenha ainda um else associado.

A regra acima deixa de ser válida se o **if** mais próximo do **else** estiver isolado entre chaves. Por exemplo, na instrução a seguir:

```
if (x)
  if  (y)
    y += x; /* Executada quando x e y são ambos diferentes de 0 */
  else
    x += y; /* Executada quando x é diferente de 0 e y é igual a 0 */
```

else refere-se ao segundo **if**, enquanto na instrução:

```
if (x) {
  if  (y)
    y += x; /* Executada quando x e y são ambos diferentes de 0 */
} else
    x += y;  /* Executada quando x é igual a 0; */
             /* independentemente do valor de y */
```

else refere-se ao primeiro **if**.

A última instrução acima também poderia ser escrita colocando-se uma instrução vazia (i.e., ";") num **else** pertencente ao segundo **if**:

```
if (x)
   if  (y)
      y += x;
   else
      ;        /* Instrução vazia  */
else
   x += y;     /* Executada quando x é igual a zero  */
```

Lembre-se de que é sempre boa prática de programação indicar por meio de comentários quando uma instrução vazia está sendo deliberadamente utilizada, como no último exemplo. Observe ainda que a colocação do **else** no mesmo nível de endentação de seu **if** associado, como nos exemplos acima, ajuda a identificar possíveis casamentos errôneos entre **else**s e **if**s.

► Instrução switch

A instrução **switch** é uma instrução de **seleção múltipla** útil quando existem várias ramificações possíveis a serem seguidas num trecho de programa. Neste caso, o uso de instruções **if** aninhadas tornam o programa difícil de ser lido. O uso de instruções **switch** nesses casos não apenas melhora a legibilidade como também a eficiência do programa. Infelizmente, nem sempre uma instrução **switch** pode substituir instruções **if** aninhadas. Mas, quando possível utilizar, uma instrução **switch** permite que caminhos múltiplos sejam escolhidos com base no valor de uma expressão. A sintaxe dessa instrução é:

```
switch (expressão) {
      case expressão-constante1 :  instrução1;
      case expressão-constante2 :  instrução2;
                        …
      case expressão- constanteN :  instruçãoN;
      default                    :  instruçãoD;
}
```

A expressão entre parênteses que segue imediatamente a palavra **switch** deve resultar num valor inteiro (por exemplo, **char**, **int**, **long**); esta expressão não pode resultar em **float**, por exemplo.

A instrução **switch** é interpretada da seguinte maneira: *expressão* é avaliada e, se ela for igual a alguma expressão constante seguindo uma ramificação **case**, *todas* as instruções que seguem esta expressão serão executadas. Se *expressão* não for igual a nenhuma instrução constante seguindo um **case** e houver uma parte **default**, que é opcional, a instrução seguindo esta parte será executada.

Uma diferença importante entre a instrução **switch** de C e instruções análogas em outras linguagens é que, na instrução **switch**, *todas* as instruções seguindo a ramificação **case** selecionada são executadas, mesmo que algumas destas instruções façam parte de outra ramificação **case**. Este comportamento é evitado (e usualmente é o que se deseja) por meio

do uso de desvios incondicionais: **break**, **goto** ou **return**. Tipicamente, a instrução **break** é utilizada para fazer com que a instrução **switch** seja encerrada e o controle passe para a instrução seguinte a esta instrução. Portanto, na grande maioria das vezes, deve-se utilizar um **break** no final de cada instrução (ou seqüência de instruções) seguindo cada **case**. Por exemplo,

```
char  meuCaractere;
   ...
switch (meuCaractere) {
   case 'A': /* Instruções referentes à opção A */
             break;
   case 'B': /* Instruções referentes à opção B */
             break;
   case 'C': /* Instruções referentes à opção C */
             break;
   default : /* Instruções referentes à opção default */
             break;
}
```

No exemplo acima, existe a possibilidade de execução de quatro seqüências de instruções referentes ao valor correntemente assumido pela variável `meuCaractere`: uma para cada um dos valores `'A'`, `'B'` e `'C'` e a última para quando o valor de `meuCaractere` for diferente desses valores (representado pela parte **default**). Note que a instrução **break** seguindo a seqüência de instruções associada a **default** não é realmente necessária, pois não existe mais nenhuma instrução em seguida dentro do **switch**; entretanto, o uso desse **break** constitui boa prática de programação.

Quando mais de uma opção **case** num **switch** corresponde a uma mesma instrução, pode-se colocar estas opções juntas e seguidas pela instrução comum. Por exemplo:

```
switch (meuCaractere) {
   case 'a' :
   case 'A' : /* Instruções referentes à opção a ou A */
              break;
   case 'b' :
   case 'B' : /* Instruções referentes à opção b ou B */
```

```
            break;
   case 'c' :
   case 'C' : /* Instruções referentes à opção c ou C */
            break;

   default : /* Instruções referentes à opção default */
            break;
}
```

De acordo com o padrão C99, uma variável pode ser definida no início de uma instrução **switch** (i.e., fora de qualquer **case** da instrução) e tem validade desde o ponto de sua definição até o final da instrução. Uma tal variável pode ser iniciada, mas esta iniciação não terá nenhum efeito. Do mesmo modo, o início de uma instrução **switch** pode conter instruções, mas estas não terão nenhum efeito, pois nunca são executadas. Como ilustração dessas características, considere o seguinte exemplo[22]:

```
int main(void)
{
   int x = 0;

   switch (x) {
      int i = 4; // Definição vale, mas iniciação é inútil
        printf("Na entrada de switch: i = %d\n", i); // Não
chamada

      case 0:
                // O valor e i é indefinido aqui
       printf("No inicio do primeiro case: i = %d\n", i);
         i = 17; // Variável i é iniciada aqui
      default:
       printf("Na parte default do switch: i = %d\n", i);
   }

       // Se o comentário  a seguir for removido, o compilador
       // indicará um erro, pois a variável i não é mais
       // reconhecida aqui
//   i = 0;

   return 0;
}
```

[22] Se você for totalmente novato em programação em C, retorne a esta seção após estudar o **Capítulo 2** e entender como funciona um programa simples escrito nesta linguagem.

O programa do exemplo anterior produzirá como saída:

```
No inicio do primeiro case: i = 42
Na parte default do switch: i = 17
```

Note que a primeira instrução **printf()** do programa anterior não é executada. Observe ainda que o valor `42` impresso pelo programa é um valor aleatório; i.e., o valor encontrado na posição de memória onde a variável `i` foi alocada[23]. Numa situação como esta apresentada pelo programa do último exemplo, um bom compilador poderia apresentar uma mensagem de advertência similar a[24]:

```
[Warning] unreachable code at beginning of switch statement
```

Esta mensagem de advertência indica que nem a iniciação de `i` nem a primeira instrução **printf()** serão jamais executadas.

1.7.5 Desvios Incondicionais: break, continue e goto

Desvios incondicionais permitem o desvio do fluxo do programa para outras partes do programa independentemente da avaliação de qualquer condição (como ocorre com os desvios condicionais vistos antes). Essas instruções serão examinadas a seguir.

▶ Instrução break

A instrução **break** já foi apresentada anteriormente como um meio de impedir a passagem de uma instrução referente a uma ramificação **case** para outras instruções pertencentes a outras ramificações **case** de uma instrução **switch**. Pode-se, entretanto, interpretar o comportamento de **break** mais genericamente como uma forma de causar o término prematuro de uma estrutura de controle. A instrução **break** pode também ser utilizada em qualquer estrutura de repetição com a finalidade de causar o término imediato da estrutura. Por exemplo:

---

[23] Se você executar este programa, provavelmente obterá outro valor.

[24] Esta mensagem de advertência foi gerada pelo compilador gcc versão 3.4.2 usando as opções: `-std=c99 -Wall`.

```
while (x){
      ...
     if (y)
        break;
      ...
}
```

No exemplo acima, o laço **while** terminará imediatamente após a execução da instrução **break**.

O uso de desvios incondicionais, como **break**, pode tornar os programas difíceis de ser lidos. Portanto, estes desvios devem ser usados com cautela. Usualmente, sempre existe uma maneira mais elegante de se escrever um trecho de programa sem o uso de **break**. Uma exceção a esta regra é o uso de **break** em instruções **switch**, conforme visto acima.

▶ Instrução continue

Como **break**, a instrução **continue** provoca desvios incondicionais em laços de repetição. Entretanto, diferentemente do **break**, **continue** não provoca o término do laço, mas sim o retorno ao topo do mesmo. Portanto, esta instrução é útil quando se deseja desviar um trecho do corpo de um laço por alguma razão. Por exemplo:

```
while (x){
   ...
   if (y)
      continue;
   ... /* Essas instruções não são executadas quando y ≠ 0 */
}
```

No último exemplo, após a execução de **continue**, o controle do programa retorna ao topo da instrução **while** (i.e., o teste da condição `x`). Nesta situação, as instruções que seguem o **continue** (indicadas por três pontos no exemplo) não serão executadas.

No caso do laço **for**, a instrução **continue** causa o desvio para a avaliação da terceira expressão do laço (e não para a segunda, como se poderia supor). Por exemplo, após ser executado, o programa a seguir[25]:

[25] Se você não conseguir entender esse programa, retorne a ele depois de estudar o **Capítulo 2**.

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 1; printf("\nExpressao 2"); printf("\nExpressao 3"))
      if (i++ < 2)
         continue;
      else
         break;

   return 0;
}
```

resulta na impressão de:

```
Expressao 2
Expressao 3
Expressao 2
```

Exercício: Qual seria o valor impresso pelo programa do último exemplo se a instrução **continue** causasse o desvio para a segunda expressão da instrução **for** (em vez da terceira, como realmente ocorre)?

A mesma recomendação feita anteriormente para a instrução **break** também vale aqui: seja comedido no uso da instrução **continue**, pois o uso excessivo desta instrução tende a tornar o programa ilegível.

► Instrução goto

A instrução **goto** é uma instrução de desvio incondicional que assume a seguinte forma:

goto *rótulo*;

onde *rótulo* é um identificador que indica a instrução para onde o fluxo do programa será desviado após a execução do **goto** correspondente. O rótulo é declarado precedendo uma instrução com ":" entre estes. Ou seja, uma instrução rotulada deve aparecer como:

rótulo : *instrução*

Deve-se observar que, numa mesma função, duas instruções não podem possuir o mesmo rótulo e que uma instrução rotulada deve estar situada na mesma função que o **goto** que a referencia. Em outras palavras, usando **goto** não se pode desviar de uma função para outra[26].

O uso abusivo da instrução **goto** é considerado uma má prática de programação e, por causa disso, algumas linguagens (por exemplo, Java) até mesmo excluem esta estrutura de controle. Na realidade, raramente esta instrução precisa ser utilizada em linguagens estruturadas como C e não existe condição geral na qual o uso de **goto** seja indicado. Entretanto, existem algumas raras situações nas quais seu uso melhora a eficiência e a legibilidade do programa, como num dos exemplos apresentados na **Seção 2.6**.

► **Uso de break e continue em Laços Aninhados**

No caso de **laços aninhados** (i.e., laço de repetição contido em outro laço), as instruções **break** e **continue** têm efeito apenas no laço que imediatamente as contém. Isto que dizer que, se um laço interno contém uma instrução **break** (ou **continue**), esta instrução não tem nenhum efeito sobre o laço externo. Por exemplo:

```
while (x > 10) {
   do {
      ...
      if (z == 0)
         break;
      ...
   } while (y <= 15)
   ...
}
```

No exemplo acima, a execução da instrução **break** causa a saída do laço **do-while** interno e não do laço **while** externo. Raciocínio análogo

[26] Funções serão discutidas mais adiante. Aqui, é suficiente entender que funções em C são equivalentes a rotinas ou procedimentos em outras linguagens.

aplica-se ao uso de **continue** em laços aninhados. Idem para aninho entre laços de repetição e **switch-case**, no caso de **break**.

1.8 O Operador Condicional

O operador condicional é o único operador **ternário** da linguagem C e é representado pelos símbolos **?** e **:**. Este operador aparece no seguinte formato:

operando1 **?** *operando2* **:** *operando3*

O primeiro operando representa tipicamente uma expressão condicional (i.e., um teste), enquanto o segundo ou o terceiro operando representa o resultado final da expressão, sendo que apenas um destes valores será escolhido, de acordo com o resultado da avaliação do teste. Mais especificamente, o resultado da expressão acima será *operando2* quando *operando1* for diferente de zero e *operando3* quando *operando1* for zero. Os operandos podem ser de quaisquer tipos escalares, mas se o segundo e o terceiro operandos forem de tipos diferentes haverá conversão implícita de acordo com as regras vistas anteriormente.

O operador condicional representa uma abreviação para instruções **if-else** de um formato específico, como mostrado no exemplo a seguir:

```
if (x)
   w = y;
else
   w = z;
```

A instrução **if** do exemplo acima pode ser substituída por:

```
w = x ? y : z;
```

Assim como os operadores **&&** e **||**, a ordem de avaliação de operandos do operador condicional é bem definida: o primeiro operando

é *sempre* avaliado em primeiro lugar, em seguida é avaliado o segundo ou o terceiro operando, de acordo com o resultado da a avaliação do primeiro operando.

A precedência do operador condicional é maior apenas do que a dos operadores de atribuição e do operador "," (ainda não visto) e sua associatividade é da direita para a esquerda. Este operador é algumas vezes difícil de ser lido; portanto, deve ser utilizado com parcimônia. Existem situações, entretanto, em que seu uso é bastante conveniente.

1.9 O Operador Vírgula

O operador vírgula (",") é um operador que permite a avaliação de mais de uma expressão onde uma única expressão seria normalmente permitida. O resultado deste operador é a expressão (operando) mais à direita. O operador vírgula é o operador de mais baixa precedência de toda a linguagem C e sua associatividade é da esquerda para a direita. A ordem de avaliação de operandos do operador vírgula é especificada como sendo da esquerda para a direita.

O uso deste operador muitas vezes prejudica a legibilidade do programa e seu uso deve ser evitado na maioria dos casos. O uso mais comum do operador vírgula é em instruções **for** quando se precisa utilizar mais de uma expressão no lugar de *expressão1* ou *expressão3* (ver esquema sintático da expressão **for** na **Seção 1.7.3**). Por exemplo:

```
for (j = 10, k = 1; j - k > 0; j--, k++){
      .
      :
}
```

Em situações diferentes da exemplificada acima, é melhor evitar o uso do operador vírgula e, em vez disso, escrever as expressões em instruções separadas.

1.10 Exercícios de Revisão

1. Quais dos seguintes nomes não podem ser utilizados como identificadores em C? Explique por quê.

(a) var;	(b) VAR;	(c) int;	(d) $a;	e) a$;	(f) _10;
(g) structure;	(h) double	(i) VOID;	(j) void;	(l) struct;	(m) default

2. Seja `a` uma variável do tipo **char**. Suponha que o código de caracteres ASCII seja utilizado. Que valor (inteiro decimal) será armazenado em `a` após a execução de cada uma das seguintes instruções:

(a) a = 'G';	(b) a = 9;	(c) a = '9';	(d) a = '1' + '9';

(**Sugestão**: Consulte uma tabela do código ASCII.)

3. Diga de que tipo são as seguintes constantes numéricas:

(a) 10;	(b) 3.14;	(c) 2.5f;	(d) 1.6e10L;	(e) 0L;	(f) 0.

4. (a) Mostre que a representação em formato decimal de cada constante em formato hexadecimal na primeira coluna da **Tabela 6** (**Seção 1.2.3**) corresponde àquela apresentada na segunda coluna desta tabela. (b) Mostre que a representação em formato hexadecimal de cada constante em formato decimal na segunda coluna da **Tabela 6** corresponde àquela apresentada na primeira coluna desta tabela.

5. Descreva as propriedades de **precedência** e **associatividade** de operadores.

6. Qual é a diferença entre os tipos **_Bool** e **bool**?

7. Sejam `f` uma variável do tipo **float**, `u` uma variável do tipo **unsigned int**, `i` uma variável do tipo **int**, `c` uma variável do tipo **signed char**. Que valores serão atribuídos a `f`, `u`, `i` e `c` após a execução de cada uma das seguintes instruções:

(a) `c = i = f = u = 23.5;`
(b) `i = u = f = c = 435;`
(c) `f = c = i = u = -10;`

(**Sugestão**: Você certamente precisará representar alguns números como seqüências de bits para entender como os mesmos serão interpretados. Suponha que números negativos são representados em complemento de dois.)

8. Assuma a existência das seguintes definições de variáveis num programa em C:

```
int    m = 5, n = 4;
float  x =2.5, y = 1.0;
```

Quais serão os valores das seguintes expressões?

(a) `m + n + x - y`
(b) `m + x - (n + y)`
(c) `x - y + m + y / n`
(d) `m += n + x - y`
(e) `m /= x*n + y`
(f) `n %= y + m`
(g) `x += y -= m`

9. Explique a diferença entre os operadores prefixo e sufixo de incremento.

10. O que é efeito colateral de um operador?

11. Quais são os operadores da linguagem C que possuem ordem de avaliação de operandos definida?

12. Dadas as seguintes iniciações:

int j = 0, m = 1, n = -1;

Quais serão os resultados de avaliação das seguintes expressões em C?

(a) `m++ - --j`
(b) `m += ++j*2`
(c) `m * m++` (O resultado desta expressão é dependente do compilador; apresente os dois resultados possíveis.)

13. Explique por que a instrução abaixo não é portável (apesar de ser aceitável em C):

```
j = (i + 1) * (i = 1)
```

14. Considere o seguinte trecho de programa:

```
int  i, j = 4;

i = j * j++;
```

Mostre que, se `j` for avaliado primeiro, a expressão `j * j++` resultará em `16` e, se `j++` for avaliado primeiro, o resultado será `20`.

15. Aplique parênteses nas expressões a seguir que indiquem o modo como um compilador de C executaria as mesmas:

(a) `a = b*c == 2`
(b) `a = b && x != y`
(c) `a = b += c + a`

16. A instrução a seguir é legal em C? Explique.

```
(1 + 2) * 4;
```

17. Descreva a sintaxe (i.e., o formato) e a semântica (i.e., o funcionamento) das seguintes estruturas de controle de C:

(a) **while**;
(b) **do-while**;
(c) **for**.

18. (a) Quantas vezes o corpo do laço **while** do exemplo a seguir será executado? (b) Quais serão os valores de `x` e `y` imediatamente após a saída desse laço?

```
long   x = 0L, y = 10L;

while (x < y) {
       x++;
       y--;
}
```

19. (a) Existe alguma diferença entre a instrução **while** do exercício anterior e a instrução **do-while** a seguir? (b) Quantas vezes o corpo do laço **do-while** será executado? (c) Quais serão os valores de `x` e `y` imediatamente após a saída desse laço?

```
long  x = 0L, y = 10L;

do {
  x++;
  y--;
} while (x < y);
```

20. Na instrução for a seguir, a expressão `j++` pode ser substituída por `++j` sem alterar a funcionalidade da instrução? Por quê?

```
for  (j = 1; j <= 10; j++) {
   /* Seqüência de instruções a ser    */
   /* executada 10 vezes repetidamente */
}
```

21. Para que serve o uso de instruções **break** dentro de uma instrução **switch**?

22. (a) Compare as instruções de controle **break** e **continue**. (b) Dentro de que estruturas de controle estas instruções podem ser incluídas?

23. (a) Descreva o funcionamento da instrução **goto**. (b) Por que usualmente o uso de **goto** não é incentivado?

24. Ambos os trechos de programa a seguir contêm erros:

Trecho de programa 1	Trecho de programa 2

`int x = 0;`
 	
`if (x = 10)`
`  y += 1;`
	`int x = 0;`
	
	`if (10 = x)`
	`  y += 1;`

Por que o compilador indica uma ocorrência de erro no trecho de programa 2, mas o mesmo não ocorre com o trecho de programa 1?

25. Reescreva o seguinte trecho de programa sem utilizar **goto**, **continue**, ou **break**:

```
    int    num = 0;
    char   c;

    c = getchar();
    while (1){
       if (c == '\n')
          break;
       if (isdigit(c))
          continue;
       if (c == 'a')
          goto  somaNumero;

proximoCaracter:
       c = getchar();
       goto  finalDoLaco;

somaNumero:
       num++;
       goto  proximoCaracter;

finalDoLaco:
       ;
    }  /* Final do while */
```

26. Suponha que `soma` e `x` sejam variáveis do tipo **int** iniciadas com 0. A instrução a seguir é portável? Explique.

```
soma = (x = 2) + (++x);
```

27. Suponha que `soma` e `x` sejam variáveis do tipo **int** iniciadas com 0. A instrução a seguir é portável? Explique.

```
soma = (x = 2, ++x);
```