Capítulo 1

INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO C

1.1 Histórico da Linguagem C

A linguagem C foi desenvolvida em 1972 por D. Ritchie nos laboratórios da AT&T como uma linguagem para programação de sistemas operacionais e de outros programas básicos. A intenção de Ritchie foi criar uma linguagem que contivesse uma mistura de características de baixo nível, necessárias para acessar linguagem de máquina, com características de alto nível, necessárias para tornar os programas portáveis e fáceis de serem mantidos.

Antes do surgimento de C, sistemas operacionais eram tradicionalmente escritos em *assembly*. Isto deve-se ao fato de a velocidade 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) rapidez de implementação, (2) legibilidade, e (3) portabilidade. A portabilidade permite que o sistema operacional seja transportado para outro computador sem que o programa precise ser completamente reescrito: é necessário apenas que se tenha um compilador para a máquina desejada e que o programa seja recompilado. Por causa deste aspecto de portabilidade, as evoluções de C e de UNIX sempre caminharam juntas: cada implementação de UNIX para uma nova máquina requeria um compilador C para aquela máquina. Portanto, muitas pessoas imaginavam C como uma linguagem desenvolvida estritamente para implementação de UNIX. A partir de meados dos anos 80, C passou a ser reconhecida como uma linguagem de propósito geral.

Com o ganho de popularidade, C passou a contar com vários compiladores comerciais; muitos dos quais oferecendo características diferentes daquelas que faziam parte da especificação original da linguagem. Essas discrepâncias entre versões diferentes de C comprometiam uma das principais características desejáveis numa linguagem de alto nível: a portabilidade. Um programa é portável se cada declaração ou instrução utilizada para codificá-lo sob um compilador A pode também ser utilizada para codificá-lo sob um compilador B (ou, pelo menos, se as modificações necessárias possam ser facilmente isoladas e mantidas num nível mínimo). Para que haja esta compatibilidade entre compiladores, é necessário que exista uma padronização da linguagem que deva ser seguida pelos desenvolvedores de compiladores. A padronização da linguagem C (ou versão oficial de C) foi inicialmente especificada por um comitê do *American National Standards Institute* (*ANSI*) e, por causa disso, foi denominada de **ANSI C**, ou, simplesmente, **C padrão**. A versão mais recente de padronização de C foi homologada em 1999 e é conhecido como **C99**. Apesar de desenvolverem suas próprias versões de C, muitos fabricantes de compiladores C oferecem a opção ANSI C (ou ISO C) que pode ser escolhida pelo programador, por exemplo, através de uma caixa de diálogo ou diretiva de compilador. Em nome da portabilidade, é sempre bom utilizar a opção ANSI/ISO C.

A linguagem C 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 C; i.e., um compilador C não impõe regras que protejam o programador contra eventuais enganos, como o fazem outras linguagens (por exemplo, Pascal e Modula-2). Por exemplo, se o programador desejasse escrever um laço de repetição **for** num programa Pascal e, acidentalmente, digitasse ponto-e-vírgula logo após o início do laço, como:

```
for i := 1  to  10  do;
   (* corpo do laço *)
```

o compilador Pascal acusaria um erro, pois esta construção não é permitida. Entretanto, na construção equivalente na linguagem C:

```
for (i = 1; i < 10; i++);
   /* corpo do laço */
```

o compilador C não acusaria erro nenhum, e o efeito seria que o corpo do laço não seria executado 10 vezes conforme o programador esperaria, mas simplesmente uma vez.

A linguagem C foi desenvolvida para programadores experientes, e portanto, ela assume pouco a respeito daquilo que o programador deseja ou não fazer. C é uma linguagem extremamente poderosa e que dá muita liberdade ao programador para escrever programas que seriam muito difíceis de serem escritos em outra linguagem. Entretanto, programadores inexperientes abusam desta liberdade para escreverem programas que são desnecessariamente ilegíveis e difíceis de serem mantidos. O objetivo deste curso não é simplesmente ensinar a construir programas que funcionam, mas programas que além de funcionarem são bem construídos, fáceis de serem 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 Reservadas

Um **identificador** serve para nomear (ou rotular) objetos utilizados 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, dígitos e o caracter especial sublinha (_), sendo que o primeiro elemento constituinte não pode ser um dígito. O número de caracteres permitido para identificadores depende da implementação de C utilizada, mas o padrão ANSI/ISO requer que compiladores C aceitem, pelo menos, identificadores contendo 31 caracteres. Normalmente, compiladores modernos dão liberdade para programadores *sensatos* escreverem identificadores do tamanho desejado.

Uma característica importante da linguagem C é que, diferentemente de algumas outras linguagens (por exemplo, Pascal), ela 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 utilizados pela própria linguagem C (por exemplo, **while**, **for**) ou por bibliotecas de rotinas (por exemplo, **abs**, **cos**) e, portanto, não podem ser utilizados como identificadores pelo programador. Estas palavras são conhecidas como **palavras reservadas** da linguagem.

1.2.2 Tipos de Dados Elementares

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

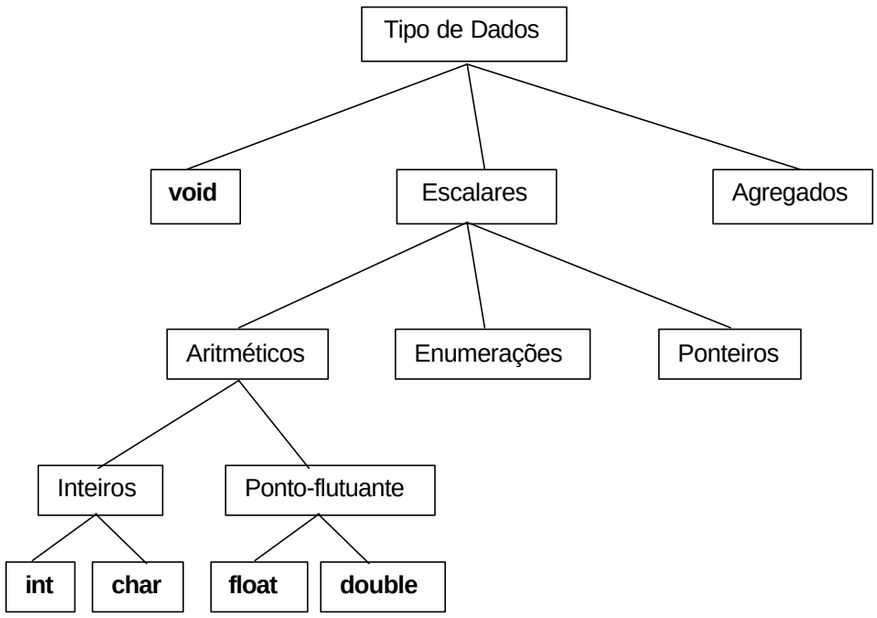

Figura 1: Tipos de Dados Básicos da Linguagem C

Nesta seção, os tipos aritméticos serão estudados. Existem dois tipos básicos de inteiros e dois tipos básicos de números de ponto flutuante. O espaço ocupado em memória por valores destes tipos, bem como as palavras reservadas que os identificam, estão apresentados na **Tabela 1**. Note, entretanto, que o espaço ocupado por cada um destes tipos (com exceção do tipo **char**, que ***sempre*** ocupa 1 byte) depende de implementação e que a **Tabela 1** refere-se apenas a uma implementação típica.

TIPO	PALAVRA-CHAVE	TAMANHO
Inteiro	**int**	4 bytes
Inteiro	**char**	1 byte
Ponto-flutuante	**float**	4 bytes
Ponto-flutuante	**double**	8 bytes

Tabela 1: Tamanhos Típicos de Tipos Aritméticos

É interessante notar que, em C, o tipo **char** não apenas é utilizado para representar caracteres (como em Pascal), 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 inteiro pode ainda ser **qualificado** por meio das seguintes palavras-chave: **short**, **long**, **signed** e **unsigned**. Os qualificadores **signed** e **unsigned** podem também ser utilizados para qualificar o tipo caractere, e suas interpretações referem-se ao fato de o tipo ser considerado com sinal (**signed**) ou sem sinal (**unsigned**). A qualificação de um tipo é feita antepondo-se o qualificador ao nome do tipo (por exemplo, **signed char**; **unsigned long int**). No caso dos inteiros, 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 se ter um tipo que seja **signed** e **unsigned** ao mesmo tempo).

Por padrão, o tipo inteiro apresenta-se com sinal, 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 aquele tipo pode conter. Por exemplo, um tipo **signed char**, que é representado por um byte, pode conter qualquer número no intervalo de -128 a 127, enquanto que o tipo **unsigned char** pode conter qualquer número no intervalo de 0 a 255.

As interpretações de **short** e **long** para inteiros 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 desta dependência com o compilador, **short int** deve sempre ocupar menos espaço do que **long int**.

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.

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 $2^{8} - 1$

Tabela 2: Intervalos Típicos de Valores de Tipos Aritméticos

O qualificador **long** pode também ser aplicado ao tipo **double**, mas (novamente) esta interpretação depende do compilador utilizado. Por exemplo, **long double** pode ser simplesmente equivalente a **double** ou a um tipo de precisão duas vezes maior do que **double**[1].

1.2.3 Constantes

Existem cinco tipos de constantes em C: **inteiras**, de **ponto flutuante**, **caracteres**, ***strings*** e **enumerações**. Estas duas últimas serão vistas mais adiante.

1.2.3.1 Constantes Inteiras

Constantes inteiras podem ser de três tipos, de acordo com a base de dígitos 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 mesmo começa com 0.

[1] Aqui, precisão refere-se ao número de casas decimais que o tipo pode conter.

• **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; portanto seria interpretado com decimal).

• **Hexadecimais**. Constantes inteiras hexadecimais representam números inteiros na base hexadecimal (16). Estas constantes devem começar com `0x` ou `0X`, e utilizar os dígitos de 0 a 9, e as letras de `A` a `F` (`A` representa 10, `B` representa 11, etc.). Exemplos válidos: `0x7FFA`, `0X230`.

O número de bytes alocados para uma constante inteira varia de implementação para implementação. Pode-se explicitamente especificar que uma constante tenha o tipo **long int** através do uso de **l** ou **L** no final da constante (por exemplo, `15L`, `0XAAB3FL`). Pode-se também especificar que uma constante seja do tipo **unsigned int** usando `u` ou `U` (por exemplo, `15U`, `0XAAB3FU`). Uma constante inteira sem sufixo (i.e., sem **l**, **L**, **u** ou **U**) tem tipo **int** se seu valor pode ser acomodado neste tipo; caso contrário, o tipo será **long int**; se este último tipo ainda não for suficiente para conter a constante, o tipo será **unsigned long int**. Finalmente, se este não for ainda suficiente, o resultado é imprevisível: alguns compiladores irão **truncar** (i.e., *cortar*) a constante para ela caber em memória, enquanto outros compiladores irão emitir uma mensagem de erro apontando uma condição de **overflow** (tamanho excessivo da constante). O compilador procede de forma similar para constantes com sufixo; i.e., ele tenta primeiro o menor tipo capaz de conter a constante especificada pelo sufixo. Por exemplo, o menor tipo capaz de conter uma constante com sufixo **U** (ou **u**) é **unsigned int**, enquanto que o maior tipo neste caso é **unsigned long int**. Os dois tipos capazes de conter uma constante com sufixo **L** (ou **l**) são **long int** e **unsigned long int**.

1.2.3.2 Constantes de Ponto Flutuante

Existem duas formas de se escreverem constantes de ponto flutuante. A primeira é simplesmente colocando-se um ponto decimal em um número, como por exemplo:

```
3.1415
.5
7.
```

A segunda forma de se escrever uma constante de ponto flutuante é através de **notação científica**. Nesta notação, um número de ponto flutuante consiste de 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`.

Constantes inteiras ou de ponto flutuante podem ser precedidas por um sinal de menos ou mais. Constantes precedidas pelo sinal de menos representam números negativos, enquanto que o sinal de mais serve para indicar números positivos e é redundante, pois um número diferente de zero e sem sinal é considerado positivo de qualquer forma. Na realidade, em C, números precedidos por sinal são considerados como *expressões*, e não como *constantes*.

1.2.3.3 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 contendo 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 o 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` (ao invés de `'A'`)[2].

1.2.3.4 Seqüências de Caracteres de Escape

Uma seqüência de caracteres de escape é uma seqüência de caracteres entre apóstrofos e cujo primeiro caractere é \. Seqüências de escape contendo dois caracteres representam caracteres não-imprimíveis que produzem certos efeitos especiais de saída (usualmente na tela do computador) quando utilizadas em instruções de saída. Algumas seqüências de escape comuns são as seguintes:

- `'\n'` - esta seqüência produz a impressão de uma nova linha no meio de saída
- `'\a'` - esta seqüência faz com que o computador emita um sinal sonoro
- `'\t'` - esta seqüência produz uma tabulação horizontal no meio de saída

Caracteres normais também podem ser representados por meio de seqüências de caracteres de escape. 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 não é usualmente necessária.

1.2.4 Declarações 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 **declarações de variáveis**.

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

Em C, toda variável precisa ser declarada antes de ser usada. As declarações de variáveis em um bloco[3] devem preceder qualquer instrução executável no bloco (o conceito de bloco será visto posteriormente). Uma declaração de variável em C consiste simplesmente de uma palavra-chave representando o tipo da variável seguida do nome da variável. Variáveis de um mesmo tipo podem ainda ser declaradas juntas e separadas por vírgulas. Por exemplo, a declaração:

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

declara 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 declarações:

---

[2] **Cuidado**: no código ASCII, caracteres representando dígitos não têm representações internas correspondentes aos próprios valores dos dígitos; por exemplo, as constantes `'5'` e `5` não são a mesma coisa, pois o código ASCII correspondente a `'5'` é 53.

[3] Um bloco consiste de uma seqüência de instruções delimitadas por chaves.

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

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*).

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[4]. Isto significa que não se deve fazer nenhuma 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 logo no início do programa (ou função). Esta **inicialização** pode ser feita em C combinando-se a declaração da variável com a atribuição do valor desejado. Por exemplo, suponha que se deseje atribuir o valor `0` a uma variável inteira `minhaVar`. Então isto poderia ser feito através da seguinte declaração:

```
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` aparece em vários pontos de seu programa. Se você quisesse modificar este valor (digamos para `3.14159`) você 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 através da **diretiva de pré-processador #define.** Por exemplo, suponha que se deseje denominar de `PI` o valor `3.14`, então a seguinte diretiva seria suficiente:

---

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

```
#define  PI  3.14
```

Quando o programa contendo a declaração acima é compilado (ou, mais precisamente, pré-processado), o compilador substitui todas as ocorrência 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 serão sempre escritos em letras maiúsculas.

1.2.7 Enumerações

Uma enumeração é útil quando se deseja utilizar um conjunto determinado de valores que podem estar associados a uma variável. Quando tenta-se atribuir a uma variável de um tipo enumeração um valor que não faz parte da própria enumeração, o compilador emite uma mensagem de erro. Uma declaração de variável de um tipo enumeração consiste da palavra reservada ***enum*** seguida de uma lista de identificadores entre chaves, e seguido finalmente pelo nome da variável. Por exemplo:

```
enum  {AZUL, VERMELHO, BRANCO, PRETO} cores;
```

declara a variável `cores` como sendo do tipo enumeração que consiste dos valores constantes `AZUL`, `VERMELHO`, `BRANCO` e `PRETO`; estes são os únicos valores que a variável `cores` pode assumir.

Valores inteiros são associados aos nomes de constantes na lista de enumeração de uma variável deste tipo. Caso não haja indicação em contrário, estes valores são baseados simplesmente nas posições das constantes na lista de enumeração, sendo que a primeira constante recebe o valor 0, a segunda constante recebe o valor 1, e assim por diante. No último exemplo, `AZUL` recebe o valor 0, `VERMELHO` recebe o valor 1, `BRANCO` recebe o valor 2, e `PRETO` recebe o valor 3. Muitas vezes, estes valores não são importantes, mas pode ser que, em algumas situações, seja necessário modificar estes valores *default*. Por isso, a linguagem C permite que se especifiquem valores de constantes numa enumeração. Se o valor de uma dada constante na enumeração não for definido explicitamente, seu valor será o valor da constante anterior na seqüência acrescido de 1; se o valor da primeira constante não for definido, este será zero (como antes). Por exemplo:

```
enum  {AZUL = -3, VERMELHO, BRANCO = 20, PRETO} cores;
```

No último exemplo, as constantes `VERMELHO` e `PRETO` terão valores `-2` e `21` respectivamente.

Note que não se requer que as atribuições de constantes sejam feitas em ordem crescente, como no exemplo acima, mas em nome da legibilidade, isto é recomendável.

1.3 Operadores e Expressões Aritméticos

Uma **expressão aritmética** é uma combinação ***legal*** de **operadores** e **operandos** que denotam o cálculo de um valor. Os componentes de uma expressão podem incluir variáveis, constantes ou chamadas de funções combinados por meio de operadores para formar a própria expressão. Como será visto ao longo do texto, a linguagem C possui muitos operadores. A seguir, os operadores aritméticos de C serão examinados.

1.3.1 Operadores Aritméticos

Nesta seção serão abordadas expressões aritméticas construídas pelos operadores aritméticos básicos, que são apresentados na **Tabela 3**.

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

Tabela 3: Operadores Aritméticos

A maioria dos operadores da **Tabela 3** funciona como 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 ainda que não existem símbolos distintos para denotar divisão inteira e real, 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 real; o compilador deduz do contexto se a divisão é inteira ou real (i.e., se os operandos são inteiros, a divisão será inteira).

Deve-se tomar cuidado com os operadores / e % quando se está lidando com inteiros negativos, pois o resultado não é especificado e pode variar de compilador para compilador. Por exemplo, `-5/2` pode resultar em `-2` ou `-3` dependendo do compilador. Da mesma forma, evite utilizar operandos negativos com o operador % (por exemplo, `-5%2` pode resultar em `1` ou `-1` dependendo do compilador).

1.3.2 Precedência e Associatividade

A maioria dos operadores aritméticos apresentados acima é conhecida de outras linguagens. Também como em outras linguagens, C avalia expressões de acordo com uma certa ordem de execução de operações. Esta ordem é determinada por duas propriedades de operadores: **precedência** e **associatividade**.

Operadores aritméticos são agrupados em grupos de precedência conforme mostra a **Tabela 4**.

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

Tabela 4: Precedências de Operadores Aritméticos

Operadores num mesmo grupo na **Tabela 4** 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.

Agora, operadores num mesmo grupo de precedência podem aparecer juntos numa expressão e a questão que surge é: *Se dois operadores numa expressão têm a mesma precedência, qual deles será aplicado primeiro?* A resposta para esta questão está no uso da segunda propriedade dos operadores: a associatividade. Existem dois tipos de associatividade de operadores: (1) **associatividade da esquerda para a direita** e (2) **associatividade da direita para a esquerda**. Operadores que apresentam associatividade da esquerda para a direita são ditos serem **associativos à esquerda**; caso contrário, eles são **associativos à direita**. Dentre os operadores apresentados acima, 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, o compilador começa a executar as operações 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[5].

Pode-se alterar a ordem de execução de uma expressão por meio do uso de parênteses. O compilador C executa operações entre parênteses antes de qualquer outra operação independentemente de precedência ou associatividade[6].

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.

1.3.3 Mistura de Tipos e Conversões Automáticas

C permite que tipos aritméticos (numéricos) 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 quatro tipos de conversões feitas implicitamente pelo compilador C. Três delas serão discutidas aqui. Elas são:

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

```
c = 882
```

---

[5] 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.

[6] 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.

pode resultar, na realidade, na atribuição do valor `114` a `c`. Por que isto ocorre? Primeiro, o valor `882` é grande demais para caber no tipo **char** (que ocupa apenas um byte). Segundo, pode-se observar que `882` é representado em 16 bits pela seqüência:

```
00000011  01110010
```

Mas, como o tipo **char** deve ser contido em apenas 8 bits, o compilador considera apenas os 8 bits mais baixos da seqüência:

```
01110010
```

que eqüivale a `114`.

• **Conversão de Alargamento de Inteiros**. Quando aparecem numa expressão, os tipos **signed char** e **short** são convertidos para **int**; enquanto que tipos **unsigned char** e **unsigned short** são convertidos para **int** (se couberem) ou **unsigned int**. Este tipo de conversão é inevitável, mas não produz nenhum efeito indesejável.

• **Conversão de Compatibilidade de Operandos**. Variáveis e constantes são convertidos de modo que um dado operador tenha como operandos valores do mesmo tipo. Esta conversão obedece a hierarquia de tipos apresentada na **Figura 2** a seguir.

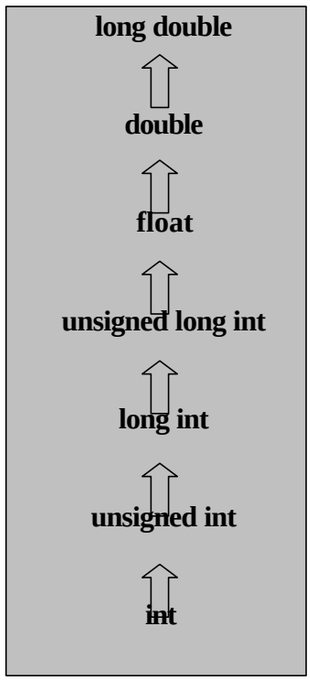

Figura 2: Hierarquia de Tipos Aritméticos

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, a variável `x` será convertida automaticamente para **float**.

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

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

Quando números de ponto flutuantes são convertidos numa atribuição, dois problemas podem acontecer: (1) **perda de precisão**, e (2) ***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 todas as casas decimais 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 `1.0123456789` e `f` é **float**, então após atribuição:

```
f  = d
```

`f` conterá o valor `1.012346`. 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 76 (i.e., `10E76`) e que o tipo **float** possa conter números com ordem de grandeza de no máximo 38. Então, uma atribuição:

```
f = 3.0E70
```

acarretaria em erro em tempo de execução do programa[7].

Conversões implícitas entre números de ponto flutuante e inteiros também podem acarretar em problemas. Claramente, a atribuição de um número de ponto flutuante a um inteiro pode gerar *overflow*, pois números de ponto flutuante são usualmente capazes de conter números maiores do que o permitido para inteiros. Também, apesar, de os tipos de ponto flutuante serem usualmente maiores 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 = 123456789
```

poderá causar a atribuição de `1.234568E8` a `f`.

Como exemplo final do que pode ocorrer com conversões implícitas, considere a seguinte expressão:

```
10u - 15
```

O resultado esperado para expressão acima seria certamente `-5`, mas isto não ocorre na realidade. Como a primeira constante (`10u`) é do tipo **unsigned int** 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 `-5` interpretado como **unsigned int**; supondo uma representação em 16 bits com complemento de dois para o tipo **int**, `-5` seria representado como:

---

[7] Compiladores C usualmente não fazem arredondamento ou truncamento em casos como este.

`11111111 11111011`. Se esta representação for interpretada como **unsigned int**, ter-se-á um enorme número inteiro (faça as contas: $2^{15} + 2^{14} + ... + 2^3 + 2^1 + 2^0$), ao invés de `-5`.

1.3.4 Conversão Explícita (*Casting*)

O programador pode também especificar conversões explicitamente em C (isto é chamado de *casting*). Para fazer uma conversão explícita de um tipo em outro, deve-se antepor o objeto do tipo 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 (convença-se de que entende por que isto ocorre). Entretanto, se um (ou ambos) 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;
```

Uma construção como ***(tipo)*** (como **(float)** acima) trata-se de mais um operador em C. Operadores de *casting* têm a mesma precedência dos operadores + e - unários. Portanto, a expressão acima é interpretada como: `((float) i1)/i2`[8].

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 essa conversão mais legível.

1.3.5 Operadores de Atribuição Aritmética

O sinal de igualdade utilizado em atribuição também é um operador. Este operador é associativo à direita. Por causa disso, 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` (i.e., `1`) a `y`, e `y` (i.e., `1`) 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:

```
int      j;
double   d;

j = d = 2.5;
```

Neste último exemplo, `d` recebe o valor `2.5`, mas `j` recebe o valor `2` (o valor de `d` convertido para **int**).

---

[8] Observe ainda 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`.

Exercício: Que valores receberiam `d` e `j` na atribuição `d = j = 2.5`?

Existem cinco outros operadores de atribuição que combinam operações aritméticas com atribuição em instruções únicas. Estes operadores, denominados de **operadores de atribuição aritmética**, e seus equivalentes funcionais são apresentadas na **Tabela 5**.

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 5: Operadores de Atribuição Aritmética

Todos os operadores de atribuição (incluindo `=`) fazem parte de um mesmo grupo de precedência baixíssima, e a associatividade destes operadores é da direita para a esquerda[9].

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

```
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[10].

1.3.6 Operadores de Incremento e de Decremento

Os operadores de incremento e de decremento são operadores unários que, quando aplicados a uma variável, adicionam ou subtraem 1, respectivamente. Estes operadores aplicam-se a variáveis numéricas ou do tipo ponteiro[11]. Os operadores de incremento e de decremento são representados respectivamente pelos símbolos `++` e `--`, e têm a mesma precedência dos operadores unários `+` e `-`.

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 ao valor) da

[9] Na realidade, os operadores de atribuição têm a precedência mais baixa do que as precedências de todos os outros operadores de C, com a exceção do operador "`,`", que será visto posteriormente.

[10] Alguns compiladores *inteligentes* são capazes de transformar, por exemplo, uma instrução como `a = a + b` em `a += b` quando esta última instrução é mais eficiente, mas nem todos os compiladores possuem esta capacidade.

[11] A interpretação de incremento e decremento de ponteiros é diferente da apresentada aqui, conforme será visto mais adiante.

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 resultam no ***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 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 que 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 dos operadores de incremento e de decremento podem ser resumidos na **Tabela 6**.

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 6: Operadores de Incremento e de Decremento

Muitas vezes (talvez, na maioria das 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. Por outro lado, não é recomendável o uso de uma variável afetada por um operador que produz efeito colateral na mesma expressão em que ocorre tal efeito. Esta recomendação deve-se ao fato de não ser especificado, nestes casos, quando o efeito colateral irá ocorrer. Por exemplo, considere o seguinte trecho de programa:

```
int  i, j = 4;

i = j * j++;
```

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. Se o primeiro `j` for avaliado primeiro, a expressão será resultará em `16` (i.e., `4*4`); se `j++` for avaliado primeiro, o resultado será `20` (i.e., `5*4`)[12].

1.3.7 O Operador sizeof

[12] Você pode ter ficado intrigado com este exemplo. Afinal, você pode pensar, o operador ++ tem precedência maior do que a precedência do operador `*` e, portanto, deve ser aplicado antes da multiplicação. Porém, acontece que precedência refere-se à ordem de avaliação de sub-expressões que ocorrem dentro de uma expressão e não a efeitos colaterais de operadores. A expressão `j++` resulta sempre em `4`, independentemente do compilador utilizado.

O operador **sizeof** é um operador prefixo de precedência igual à dos operadores + e - unários, e com associatividade da direita para a esquerda. Este operador pode receber como operando um tipo de dado (predefinido ou definido pelo programador) 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,

```
unsigned long int tamanhoDoTipoDouble = sizeof(double);
```

resulta na inicialização 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 ANSI/ISO, o resultado do operador **sizeof** deve ser **unsigned int** ou **unsigned long int**.

1.4 Operadores Relacionais e Lógicos

Operadores relacionais são operadores binários utilizados em **expressões de comparação**. Existem seis operadores relacionais em C, que são apresentados juntamente com seus possíveis resultados, na **Tabela 7** apresentada a seguir:

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 7: Operadores Relacionais

Os quatro primeiros operadores na **Tabela 7** têm a mesma precedência, que é menor do que as precedências dos operadores vistos antes, com exceção dos operadores de atribuição. Os dois últimos operadores estão uma classe de precedência abaixo dos quatro primeiros operadores[13].

Outras linguagens de programação, tais como **Pascal** e **Modula-2**, possuem operadores relacionais equivalentes aos operadores relacionais de C apresentados na **Tabela 7**. Entretanto, uma diferença fundamental é que outras linguagens possuem o tipo de dados booleano e expressões de comparação resultam num dos valores deste tipo (i.e., `TRUE` ou `FALSE`). C não possui o tipo booleano, de forma que os resultados de expressões de comparação são inteiros (`1` ou `0`, conforme mostra a **Tabela 7**).

Note ainda que o operador relacional de igualdade é representado pelo símbolo `==` e não por = como em Pascal. 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 (voltaremos a discutir este problema oportunamente).

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 <=) ao invés do operador de igualdade quando comparar números de ponto flutuante.

Operadores lógicos em C correspondem aos operadores de **negação** (`NOT`), **conjunção** (`AND`) e **disjunção** (`OR`) de outras linguagem, como, por exemplo, Pascal. Entretanto, como não existe 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 prioridade) na **Tabela 8**.

OPERADOR	SÍMBOLO
Negação	**!**
Conjunção	**&&**
Disjunção	**‖**

Tabela 8: Operadores Lógicos

O operador lógico **!** tem a mesma precedência dos operadores unários 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 aplicação de operadores lógicos sempre resulta em 0 ou 1, dependendo dos valores dos operandos. Os resultados dos valores resultantes da aplicação destes operadores de acordo

[13] O Apêndice B apresenta todos os operadores da linguagem C, bem como suas propriedades de precedência e de associatividade.

com os valores de seus operandos são resumidos na **Tabela 9** e na **Tabela 10** apresentadas a seguir[14].

X	!X
0	1
diferente de 0	0

Tabela 9: 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 10: Resultados dos Operadores && e ||

Para memorizar a **Tabela 10**, 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. Também, `X || Y` resulta em `0` apenas quando ambos os operandos são iguais a `0`.

A ordem de avaliação de operandos de expressões lógicas é especificada como sendo da esquerda para a direita. Além disso, o compilador não avalia operandos que não sejam necessários para a determinação do valor de uma expressão lógica. Por exemplo, quando o valor de `a` for zero na expressão:

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

o operando `a != 0` resulta em `0` e o compilador não avalia o operando `b/a > 4.0` pois sabe-se que para que toda a expressão resulte em `0`, basta que um dos operadores seja `0`[15].

O programador familiar com expressões relacionais em outras linguagens de programação (ou mesmo em Matemática) deve tomar cuidado ao escrever expressões desse tipo em C, pois o resultado pode não ser aquele desejado. Considere, por exemplo, a expressão `x < y < z` em Matemática. O aluno com algum conhecimento no assunto sabe que esta expressão é satisfeita quando um número `y` está entre os números `x` e `z`. A escrita desta expressão em Pascal seria sintaticamente ilegal; em C, esta expressão é legal, mas não tem o mesmo significado que teria em Matemática, visto que a expressão:

```
x < y < z
```

seria interpretada como:

```
(x < y) < z
```

Para entender por que esta expressão relacional não tem o mesmo significado matemático em C, atribua os valores `3` a `x`, `-2` a `y` e `7` a `z`. Isto resultaria na expressão: `3 < -2 < 7`, que, claramente, é falsa em Matemática, pois `-2` não está entre `3` e `7`. Entretanto, esta expressão resultaria em verdadeiro (i.e., 1) em C, de acordo com a seqüência de avaliações esquematizada a seguir:

```
3 < -2 < 7  →  (3 < -2) < 7   →  0 < 7 →  1
```

[14] Para adaptar-se a esta *nova lógica*, você pode raciocinar da seguinte maneira: *Em C, **FALSE** corresponde a zero, e **TRUE** corresponde a qualquer valor diferente de zero*.

[15] Esta é uma característica boa da linguagem C. Outras linguagens *menos inteligentes*, como **Visual Basic**, tentariam avaliar o segundo operando quando `a` fosse `0`, o que resultaria num erro em tempo de execução do programa.

Portanto, se a interpretação desejada fosse aquela de Matemática, o programador deveria rescrever a expressão como:

```
(x < y) && (y < z)
```

Neste último caso, considerando os valores atribuídos a `x`, a `y` e a `z` anteriormente, a expressão seria avaliada para falso (i.e., `0`) da seguinte maneira:

```
(3 < -2) && (-2 < 7) →   0 && 1   →   0
```

Note ainda que na expressão `(x < y) && (y < z)`, os parênteses ***não*** são necessários para assegurar a interpretação requerida, pois o operador **&&** tem precedência menor do que o operador <. O uso de parênteses, entretanto, melhora a legibilidade da expressão.

Uma observação final quanto ao uso de expressões relacionais e lógicas é que o programador deve evitar (ou, pelo menos, tomar bastante cuidado com) o uso de operadores com efeitos colaterais, como os operadores de incremento e decremento. O problema aqui é que, conforme visto acima, 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.

1.5 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 classificados em três categorias:

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

1.5.1 Seqüências de Instruções

Antes de prosseguir com a apresentação das estruturas de controle de C, é oportuno descrever aqui o conceito de **seqüência de instruções** (ou **instruções compostas**). Uma seqüência de instruções consiste de 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, declarações de variáveis.

1.5.2 Instruções Vazias

Outro aspecto interessante da linguagem C é que ela permite a escrita de **instruções vazias** (i.e., que não executam nenhuma tarefa) em qualquer local aonde pode-se colocar uma instrução normal. Uma instrução vazia em C é representada por ";" (ponto-e-vírgula), e é sempre uma boa idéia colocar instruções vazias em linhas separadas e acompanhadas de comentários[16] explicativos. Estas recomendações para escrita de instruções vazias evitam

[16] Comentários em C são escritos entre os símbolos `/*` e `*/`, conforme será visto mais adiante.

que instruções deste tipo que são ***propositais*** sejam confundidas com aquelas que são ***acidentais*** (i.e., ***bugs***). Como, isoladamente, "`;`" significa a instrução vazia, a colocação acidental deste símbolo em locais aonde 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).

1.5.3 Laços de Repetição

*1.5.3.1 Laço de Repetição **while***

A instrução **while** é uma estrutura de repetição que tem o seguinte formato em C:

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 denominado de **corpo do laço while** e pode ser representada por uma seqüência de instruções entre chaves. **Cuidado**: se você esquecer de colocar as chaves em torno de uma seqüência de instruções, 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 **while**.

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 seja avaliada como zero (falso). 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.

Exemplo de uso da instrução **while**:

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

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

Exercício: Quantas vezes o corpo do laço **while** do exemplo acima será executado?

De acordo com o que foi visto na seção anterior, deve-se tomar cuidado para não escrever "`;`" após a primeira linha de uma instrução **while** pois 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 loop */
            /* sem fim se o valor de x for diferente de 0         */
```

No exemplo acima, o programador 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 no 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.

*1.5.3.2 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 deve resultar em zero ou diferente de zero (falso ou verdadeiro, respectivamente). Em termos de interpretação, a única diferença entre **while** e **do-while** é o ponto aonde a condição de teste é avaliada. Na instrução **while**, a condição é avaliada no início do laço, enquanto que na instrução **do-while** a condição é avaliada no final do mesmo. Por causa disso, garante-se que o corpo de uma instrução **do-while** seja executado pelo menos uma vez. Portanto, a instrução **do-while** é indicada para situações aonde 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)
```

Exercício: Existe alguma diferença entre a instrução **do-while** do último exemplo e o exemplo de instrução **while** apresentado anteriormente? Quantas vezes o corpo do laço **do-while** do exemplo acima será executado?

*1.5.3.3 Laço de Repetição **for***

A instrução **for** é a última das três instruções de repetição de C e é um pouco mais complicada do que a instrução **for** de outras linguagem como Pascal. A sintaxe da instrução **for** segue o seguinte 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 condição de teste 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 no passo 3), <*expressão3*> é avaliada e retorna-se ao passo 2 acima.

Pode-se facilmente verificar que a instrução **for** é equivalente em termos funcionais à seguinte seqüência de instruções[17]:

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

Apesar desta eqüivalência, este conjunto de instruções não é indicado (por questão de legibilidade) para substituir instruções **for** aonde 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 Pascal, a instrução **for** de C é mais freqüentemente utilizada em laços de contagem como a instrução **for** de algumas outras linguagens. Por exemplo:

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

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

é eqüivalente à seguinte instrução Pascal:

```
for j := 1 to 10 do begin

    (* Seqüência de instruções a ser    *)
    (* executada 10 vezes repetidamente *)

end
```

Exercício: No último exemplo, `j++` poderia ser substituído por `++j`? Por que?

Um erro comum em instruções como a apresentada no último exemplo (em C, não em Pascal) 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 `<=`). Por exemplo, se a condição de teste utilizada no exemplo anterior fosse `j < 10`, ao invés de `j <= 10` como antes, o corpo do **for** seria executado apenas 9 vezes (e não 10 vezes, como antes). Este tipo de erro obviamente 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.

Conforme foi mencionado antes, qualquer das expressões entre parênteses pode ser omitida numa instrução **for**. Entretanto, os dois ponto-e-vírgulas devem sempre ser incluídos. Na prática, é comum omitir-se *<expressão1>* ou *<expressão3>*, mas nunca ambas ao mesmo tempo[18]. Normalmente, *<expressão2>* é sempre incluída, pois trata-se da condição de teste. Quando esta condição de teste é omitida, ela é considerada como se fosse igual a 1 (i.e., sempre verdadeira).

Evidentemente, 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.

---

[17] 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 que, no caso do laço **while**, o desvio seria feito para o topo deste laço (i.e., para *<expressão2>*).

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

1.5.3.4 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 realmente o programador deseja, mas muitas vezes, estas instruções contêm *bugs* que as impedem de terminarem 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:

```
while (1)
   <instrução>

for ( ; ; )
   <instrução>
```

Estas duas instruções são equivalentes, mas a primeira delas é mais utilizada (por questão de legibilidade) 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 *bug* porque ela contém 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.

1.5.4 Instruções de Desvios Condicionais

Instruções de desvios condicionais permitem o desvio do fluxo do programa para outras partes do programa dependendo do resultado da avaliação de uma expressão (**condição**). Essas instruções serão examinadas a seguir.

*1.5.4.1 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. Note ainda que, diferentemente de Pascal, não existe **then** na estrutura **if** de C. A interpretação do **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 que a instrução `z = x + y` é independente do **if** e é escrita no mesmo nível desta instrução. Endentação serve o 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 (ilusória) dependência 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>;
}
```

Na seção anterior, chamou-se a atenção para o perigo representado pela troca, por engano, do operador relacional de igualdade **==** pelo operador de atribuição **=**. Considere, agora, o seguinte exemplo concreto:

```
int x = 0;

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

A instrução acima contém com certeza um *bug*, 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**!)

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 um **if**). Obviamente, isto não é obrigatório, mas melhora a legibilidade de instruções **if** aninhadas.

Um fato importante em instruções aninhadas, e que acarreta algumas vezes em erro de programação, é o problema 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 primeira das instruções a seguir, o **else** refere-se ao segundo **if**, enquanto que na segunda instrução, o **else** refere-se ao primeiro **if**:

```
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 */
```

```
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  */
```

A segunda 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 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.

1.5.4.2 Operador Condicional

O operador condicional é o único operador **ternário** da linguagem C e, devido à sua semelhança com uma instrução condicional, ele será apresentado aqui. O operador condicional é representado pelos símbolos **?** e **:**, e aparece no seguinte formato:

```
x ? y : z
```

O primeiro operando (`x`) representa uma expressão condicional (i.e., um teste), enquanto que o segundo e o terceiro operandos representam o valor final da expressão, sendo que apenas um destes valores será escolhido, de acordo com a avaliação do teste. Mais especificamente, o resultado da expressão acima será `y` quando `x` for diferente de zero, e `z` quando `x` 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 e, portanto deve ser utilizado com parcimônia. Existem situações, entretanto, em que ele é bastante prático.

*1.5.4.3 Instrução **switch***

A instrução **switch** é uma instrução de **seleção múltipla** útil quando existem várias ramificações a serem seguidas num trecho de um programa. Neste caso, o uso de instruções **if** aninhadas tornam estas ramificações difíceis de serem lidas. Portanto, o uso de instruções **switch** não apenas melhora a legibilidade como também a eficiência do programa. Infelizmente, nem sempre uma instrução **switch** pode substituir instruções **if** aninhadas.

A instrução **switch** permite que caminhos múltiplos sejam escolhidos com base no valor de uma expressão. A sintaxe desta instrução é:

switch (<expressão>) {
    case <expressão constante 1> : <instrução 1>
    case <expressão constante 2> : <instrução 2>
                .
                :
    case <expressão constante N> : <instrução N>
    default : <instrução D>
}

A expressão que segue imediatamente **switch** deve ser uma expressão que resulte em um 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 cada **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 a instrução **case** de Pascal é que, na instrução **switch**, ***todas*** as instruções seguindo o **case** selecionado são executadas, mesmo que algumas destas instruções façam parte de um outro **case**. Este comportamento é evitado

(e usualmente é o que se deseja) através do uso das estruturas de controle (desvio incondicional) **break**, **goto** ou **return**. Usualmente, 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': /* Escreva aqui uma seqüência de instruções referentes */
             /* à opção A */
             break;
   case 'B': /* Escreva aqui uma seqüência de instruções referentes */
             /* à opção B */
             break;
   case 'C': /* Escreva aqui uma seqüência de instruções referentes */
             /* à opção C */
             break;
   default : /* Escreva aqui uma seqüência de 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 o **break** seguindo a seqüência de instruções associada a **default** não é realmente necessário pois não existe mais nenhuma instrução em seguida dentro do **switch**; entretanto, o uso deste **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' : /* Escreva aqui uma seqüência de instruções */
              /* referentes à opção a ou A */
              break;
   case 'b' :
   case 'B' : /* Escreva aqui uma seqüência de instruções */
              /* referentes à opção b ou B */
              break;
   case 'c' :
   case 'C' : /* Escreva aqui uma seqüência de instruções */
              /* referentes à opção c ou C */
              break;

   default : /* Escreva aqui uma seqüência de instruções */
             /* referentes à opção default */
             break;
}
```

1.5.5 Desvios Incondicionais

Desvios incondicionais permitem o desvio do fluxo do programa para outras partes do programa independentemente da avaliação de qualquer condição (como nos desvios condicionais vistos antes). Essas instruções serão examinadas a seguir[19].

*1.5.5.1 Instrução **break***

[19] Chamadas e retornos de funções são também considerados desvios incondicionais. Estas construções serão examinadas mais adiante.

A instrução **break** já foi apresentada anteriormente como um meio de impedir a passagem de uma instrução referente a um **case** para outras instruções pertencentes a outros **case**s 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 estruturas de repetição e esta segunda interpretação determina seu comportamento. Por exemplo:

```
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 serem 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.

*1.5.5.2 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 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:

```
#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 do for (ao invés da terceira, como realmente ocorre)?

A mesma recomendação feita anteriormente para a instrução **break** também vale aqui: tente não utilizar a instrução **continue**, pois seu uso tende a tornar o programa ilegível[20].

*1.5.5.3 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, que (diferentemente de outras linguagens) não precisa ser declarado, deve preceder a instrução rotulada e ser seguido por ":". Isto é uma instrução rotulada deve aparecer como:

<rótulo> : <instrução>

Deve-se observar que 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, não se pode desviar de uma função para outra. (Funções serão discutidas mais adiante. Aqui, é suficiente entender que funções em C são equivalentes a procedimentos e funções em Pascal.)

O uso abusivo da instrução **goto** é considerado uma má prática de programação e, por causa disso, algumas linguagens (por exemplo, **Modula-2**) até mesmo excluem esta estrutura de controle. Na realidade, raramente esta instrução precisa ser utilizada em linguagens estruturadas como C e Pascal e não existe condição geral na qual o uso de **goto** é indicado. Entretanto, existem algumas raras situações aonde o uso do mesmo melhora a eficiência e legibilidade do programa.

1.6 Operador Vírgula

O operador vírgula (,) é um operador que permite a avaliação de mais de uma expressão aonde 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. Também, 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 expressões **for** quando se precisa utilizar mais de uma expressão no lugar de <*expressão 1*> ou <*expressão 3*> (ver acima esquema sintático da expressão **for**). Por exemplo:

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

[20] No caso de laços aninhados, as instruções **break** e **continue** têm efeito apenas no laço que 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.

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

1.7 Funções Elementares de Entrada e Saída

C é uma linguagem relativamente pequena (i.e., com poucas construções válidas) porque parte de sua funcionalidade está contida em **funções de biblioteca** que residem fora da linguagem em si. A **biblioteca padrão** de C é dividida em grupos de funções que têm alguma afinidade entre si. Por exemplo, existem grupos de funções para entrada e saída, gerenciamento de memória, operações matemáticas, etc. Cada grupo de funções de biblioteca possui (pelo menos) dois arquivos: (1) um arquivo-objeto que contém as implementações das funções do grupo previamente compiladas (i.e., prontas para serem executadas); (2) um arquivo-fonte, denominado de **cabeçalho**, que contém ***definições parciais*** (**alusões**) legíveis das funções implementadas no arquivo-objeto[21]. Para utilizar um grupo de funções de biblioteca num programa, você deve incluir o arquivo de cabeçalho que contém as alusões das funções com uma diretiva #**include** (usualmente, no início do programa) com um dos seguintes formatos:

#**include** <<*nome do arquivo*>>

ou

#**include** "<*nome do arquivo*>"

A diferença entre os dois formatos é a forma como o compilador (ou, mais precisamente, o pré-processador) faz a busca do arquivo a ser incluído. O modo como os arquivos são procurados depende do compilador bem como do sistema operacional utilizados. O segundo formato é mais geral do que o primeiro no sentido de que a busca é mais extensa, mas, em compensação, a busca pode ser mais demorada. No caso de inclusão de arquivos de biblioteca, se a instalação do compilador foi feita de acordo com as recomendações do fabricante, não haverá problema de localização do arquivo com o primeiro formato acima. Se não conseguir incluir um arquivo com um dos formatos acima, consulte a documentação do compilador para saber aonde (i.e., em que diretório) estes arquivos podem ser localizados. Arquivos de cabeçalho normalmente têm a extensão `.h` (“h” vem de ***header***; i.e., cabeçalho em inglês).

Entrada e saída não fazem parte da linguagem C em si. As funções responsáveis pelas operações básicas de entrada e saída em C estão localizadas num grupo de funções de biblioteca denominado ***stdio*** (este nome vem de “***standard input/output***”; i.e., “entrada/saída padrão”). Para utilizar as funções de biblioteca de entrada/saída você deve, portanto, incluir o cabeçalho que contém as alusões destas funções, que é o arquivo `stdio.h`, através da diretiva:

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

Quando encontra uma diretiva #**include**, o pré-processador[22] C a substitui pelo arquivo que se deseja incluir. Por exemplo, quando a diretiva acima é processada, o conteúdo do arquivo `stdio.h` é inserido no local do programa aonde se encontra a diretiva. Desta maneira, quando o compilador encontrar uma referência (i.e., chamada) a uma função contida no arquivo `stdio.h` ele saberá que se trata realmente de uma função. Se você não incluir o arquivo de cabeçalho para as funções que deseja utilizar em seu programa,

[21] Um arquivo de cabeçalho pode conter ainda outras coisas, tais como definições de tipos e constantes.

[22] Por enquanto, não se preocupe em saber exatamente o que é o ***pré-processador C***: considere-o como um programa que prepara previamente um arquivo-fonte a ser em seguida processado pelo compilador. O pré-processador é responsável, entre outras coisas, pelo processamento de diretivas.

qualquer referência a estas funções será tratada pelo compilador como, por exemplo, *identificador não declarado*, visto que elas nem fazem parte da linguagem C em si nem você as declarou explicitamente.

Note que, como foi afirmado acima, um arquivo de cabeçalho contém apenas declarações parciais de funções; as declarações completas estão contidas no arquivo-objeto correspondente. Em resumo, as definições parciais de funções num arquivo de cabeçalho incluído servem apenas como **alusões** que indicam ao compilador que aquelas funções são definidas em outro arquivo. Para que o programa seja executado, é necessário que ele seja ligado com as definições completas das funções de biblioteca utilizadas. Esta ligação é feita por um **editor de ligações** (***link editor*** ou ***linker***).

A seguir serão apresentadas funções básicas de entrada e saída que possibilitarão, ao final deste capítulo, que você possa escrever programas elementares em C.

1.7.1 Função getchar()

A função **getchar()** é uma função de entrada que permite a entrada de um único caracter através do meio de entrada padrão (usualmente, o teclado do computador). A forma (*normal*) de uso desta função é[23]:

<variável do tipo **unsigned char**> = **getchar()**;

Quando o fluxo de execução do programa atinge uma instrução deste tipo, a execução é interrompida e espera-se que o usuário digite um caractere seguido de `<ENTER>` (ou `<RETURN>`). Então, a função **getchar()** retorna o inteiro correspondente ao caractere digitado (no intervalo entre 0 e 255). Por exemplo, se um programa contém os trechos abaixo:

```
unsigned  char  meuCaractere;

meuCaractere = getchar();
```

e o usuário digitar o caractere `A` na execução da instrução acima, à variável `meuCaractere` será atribuído o valor `65`, que é o valor inteiro decimal do caractere `A` no código ASCII (supondo que este código é utilizado). Note que, diferentemente de algumas outras linguagens, a chamada de uma função sem argumentos (parâmetros) como **getchar()**, requer que o nome da função seja seguida de abre-parênteses e fecha-parênteses.

1.7.2 Função putchar()

A função **putchar()** é uma função de saída que serve para impressão de caracteres no meio de saída padrão. A função **putchar()** recebe como entrada um parâmetro do tipo **int** e imprime o caractere correspondente a este valor (para ter controle do resultado, você deve passar um parâmetro do tipo **unsigned char**, ao invés do tipo **int**, que é permitido). A função **putchar()** retorna um inteiro que corresponde ao caractere impresso se a impressão for efetuada com êxito. Entretanto, este valor de retorno raramente é utilizado; de modo que usualmente a função **putchar()** é utilizada simplesmente como:

putchar(<variável/constante do tipo **unsigned char**>);

Por exemplo:

---

[23] Na realidade, o tipo de retorno de **getchar()** é **int**, mas você só precisa atribuir este valor retornado a uma variável do tipo **int** se estiver esperando ler, além de caracteres, o valor (`EOF`) que indica o final de um arquivo. Este valor não cabe numa variável do tipo **char**.

```
putchar('A');
```

resultaria na impressão de `A` no meio de saída padrão. A função retornaria o valor `65`, mas valor retornado não é utilizado[24]. Outra forma eqüivalente de imprimir o caractere `A` no meio de saída padrão é por meio da instrução (novamente, assumindo o uso de ASCII):

```
putchar(65);
```

Obviamente, quando se passa um valor inteiro diretamente, não se deve incluir aspas.

1.7.3 Função scanf()

A função **scanf()** é uma função de entrada mais geral do que **getchar()**, pois **scanf()** permite a entrada de valores de vários tipos simultaneamente. A função **scanf()** permite um número qualquer de argumentos. O primeiro argumento, que é obrigatório, é uma cadeia de caracteres denominado de **string de formatação**. Os argumentos seguintes são ***endereços*** de variáveis que irão conter os valores lidos no meio de entrada (mais precisamente, estes últimos parâmetros representam endereços aonde os dados lidos serão armazenados, conforme será visto mais adiante). Normalmente, uma chamada da função **scanf()** tem o seguinte formato:

scanf(<string de formatação>, <endereço de variável 1>, ..., <endereço de variável N>);

O string de formatação especifica justamente o formato dos dados a serem lidos e que serão atribuídos aos argumentos seguintes. Este string pode assumir formas bem variadas, mas, aqui, serão vistos apenas algumas formas mais comuns[25]. O string de formatação pode conter texto, mas normalmente, contém apenas **especificadores de formato**, que em sua forma mais simples, são constituídos do caractere **%** mais um caractere que informa como o dado correspondente sendo lido será interpretado. A **Tabela 11**, apresentada a seguir, enumera os especificadores de formato mais comuns.

ESPECIFICADOR DE FORMATO	INTERPRETAÇÃO DO DADO LIDO
%c	Caractere
%d	Inteiro em base decimal
%u	Inteiro sem sinal (*unsigned*)
%f	Número de ponto flutuante (**float**)
%e	Número de ponto flutuante com expoente (i.e., em notação científica)
%s	Cadeia de caracteres (*string*)
%lf	Número de ponto flutuante do tipo **double** ou **long double**

Tabela 11: Especificadores de Formato

O correto uso da função **scanf()** requer que haja um endereço de variável para cada especificação de formato no string de formatação e que o tipo de cada variável seja compatível com a especificação de formato correspondente. Por exemplo, a seguinte chamada da função **scanf()** lê três dados no meio de entrada:

```
long            inteiroLongo;
```

[24] Esta é mais uma diferença entre chamadas de funções em C e em Pascal: em Pascal, o valor retornado por uma função deve sempre ser utilizado; C não requer isto.
[25] O Apêndice C apresenta uma descrição completa desses especificadores de formato.

```
float           numeroDePontoFlutuante;
unsigned  char  meuCaractere;

scanf("%ld %f %c", &inteiroLongo, &numeroDePontoFlutuante, &meuCaractere);
```

Observe que os endereços das variáveis requeridos para os parâmetros finais da função **scanf()** são obtidos precedendo-se o nome da variável com o caractere &. Este caractere quando aplicado antes de uma variável representa o operador unário de endereço, e deve ser lido como "***o endereço de***". Cuidado para não esquecer de preceder cada variável na lista de entrada com o operador de endereço, pois a função **scanf()** não irá indicar (diretamente) o erro: simplesmente, ela não irá atribuir o valor esperado à variável.

A função **scanf()** retorna um valor inteiro que raramente é utilizado como deveria. Este valor de retorno indica o número de variáveis que tiveram valores atribuídos.

Quando caracteres que não são parte de especificadores de formato são incluídos no string de formatação, a função **scanf()** espera que o usuário digite cada caractere exatamente como ele aparece no string de formatação. Você deve tomar cuidado com esta característica da função **scanf()**, pois pode ser que isto não seja aquilo que você deseja. Por exemplo, se você incluir em seu programa a instrução:

```
scanf("Digite um numero  inteiro: %d", &meuInteiro);
```

a função **scanf()** espera que o usuário digite exatamente a cadeia de caracteres: "`Digite um numero inteiro: `" antes de introduzir o número inteiro esperado. Portanto, não se esqueça que a função **scanf()** é uma função *apenas de entrada*. Se você deseja informar ao usuário que seu programa está esperando uma entrada de dados (e isto é recomendável) preceda a instrução de entrada com uma instrução de saída que solicite a entrada do usuário. A função **printf()**, a ser apresentada em seguida, pode ser utilizada para este propósito.

1.7.4 Função printf()

A função **printf()** é uma função de saída com sintaxe similar àquela da função **scanf()**. A semelhança, entretanto, resume-se ao fato de ambas permitirem múltiplos argumentos, sendo que o primeiro (obrigatório) é um string de formatação. Os argumentos finais são variáveis (e não endereços de variáveis), constantes ou expressões, cujos valores serão apresentados no meio de saída. Além de especificadores de formato, o string de formatação pode ainda conter cadeias de caracteres que também são apresentados no meio de saída. Por exemplo, ao final da execução do trecho de programa a seguir:

```
int   n = 3;

printf("O quadrado de %d e' %d.", n, n*n);
```

seria impresso o seguinte no meio de saída:

```
O quadrado de 3 e' 9.
```

Especificadores de formato podem ser ainda mais específicos ao indicar como os argumentos finais de **printf**() devem ser impressos. Por exemplo, pode-se indicar que um número de ponto flutuante deve ser impresso com um total de cinco casas, sendo duas delas decimais, por meio do especificador `%5.2f` . Um estudo mais aprofundado de formatação de entrada/saída por meio de especificadores de formato encontra-se no **Apêndice C**.

1.8 Executando um Programa Completo

A construção de um programa (*de pequeno porte*) em qualquer linguagem algorítmica convencional segue, normalmente, a seqüência de tarefas descrita a seguir:

1. **Escrita do algoritmo** a ser seguido pelo programa numa linguagem algorítmica. Este passo corresponde ao projeto do programa.

2. **Tradução do algoritmo** na linguagem desejada. Se a linguagem algorítmica utilizada no passo 1 for próxima à linguagem de programação utilizada, este passo não oferece nenhuma dificuldade.

3. **Edição do programa** resultante do passo 2. A execução deste passo resulta num programa-fonte e pode ser considerada como parte do passo 2. Isto é, pode-se usualmente traduzir e editar um programa num único passo, sem que se tenha que fazer esta tradução à mão, por exemplo.

4. **Compilação e edição de ligações do programa**. Este passo resulta num arquivo-objeto (i.e., em código de máquina), mas este arquivo não consiste necessariamente num programa executável. O arquivo-objeto resultante da compilação precisa ser ligado a outros arquivos-objeto que porventura contenham funções ou variáveis utilizadas pelo programa. Isto é realizado através de um **editor de ligações** (***linker***). Muitos ambientes de programação possuem uma facilidade que processa compilação e edição de ligações ao mesmo tempo (esta facilidade, normalmente, é denominada **Make** ou **Build**). Alguns ambientes também oferecem uma facilidade com finalidade tripla que compila, faz ligações e executa um programa num único comando (normalmente, este comando é denominado **Run**).

5. **Teste e depuração do programa**. Neste passo, o programa deve ser executado para verificar-se se ele, efetivamente, funciona da forma como seria esperado. Deve-se examinar o programa sob várias circunstâncias (**casos de entrada**) e verificar se, em cada uma delas, o programa se comporta adequadamente. Se o programa apresenta um comportamento anormal, ele deve ser depurado (i.e., devem-se procurar as instruções responsáveis pelos erros e corrigi-las). Exceto para programas triviais, não se pode, usualmente, jamais provar que um programa é correto. Pode-se entretanto, provar que um programa é incorreto: é suficiente que ele não funcione numa dada circunstância.

Programas *triviais* não requerem os passos 1 e 2 acima. Isto é, programas triviais podem ser editados diretamente (passo 3) no editor do ambiente de programação utilizado. Evidentemente, o conceito de *trivial* aqui é relativo; isto é, ele depende da prática do programador. Se você não consegue fazer isto, não se sinta frustrado: siga toda a seqüência de passos acima, pois com alguma prática adquirida você conseguirá escrever cada vez mais programas sem ter que escrever seus algoritmos em linguagem algorítmica. Nunca esqueça, entretanto, que você não deve escrever programas complexos sem planejá-los previamente (passo 1 acima). O tempo gasto no planejamento da solução é recuperado na depuração do programa. Freqüentemente, a depuração de um programa mal planejado leva muito mais tempo do que a escrita do programa em si.

1.9 Anatomia de um Programa Simples em C

Um programa simples, consistindo de um único arquivo, em C possui o seguinte formato geral:

```
#include  <stdio.h>

/* Inclua aqui outros arquivos de biblioteca utilizados no programa */

/*    Inclua aqui declarações de constantes e tipos   */
/*    que serão necessários  no seu programa          */

int main(void)
{

      /*    Inclua aqui declarações de variáveis que  */
      /*    serão necessárias  no seu programa        */
```

```
      /*    Inclua aqui instruções que executem as ações          */
      /*    necessárias no seu programa.                          */
      /*    Isto é, traduza aqui, em C, seu algoritmo             */

      return 0; /* Informa ao sistema operacional que */
                /* o programa terminou normalmente    */
}
```

1.10 Exercícios de Revisão

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

(a) var; (b) VAR; (c) int; (d) $a; e) a$; (f) _10;
(g) double (h) VOID; (i) void; (j) struct; (l) structure; (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. Descreva as propriedades de **precedência** e **associatividade** de operadores.

5. 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.)

6. Assuma a existência das seguintes declarações 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`.

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

8. O que efeito colateral de um operador?

9. Dadas as seguintes declarações:

```
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.)

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

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

11. Aplique parênteses nas expressões abaixo que indiquem o modo como um compilador C executaria as mesmas:

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

12. 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**.

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

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

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

16. Rescreva o seguinte trecho de programa sem utilizar **goto**, **continue**, ou **break**:

```
main()
{
    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 */
}
```

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

1.11 Exercícios de Programação

EP1.1) Escreva um programa em C que imprima cem vezes a frase: `Programar em C e' otimo`.

EP1.2) (a) Utilize o operador **sizeof** em conjunto com a função **printf()** para determinar e imprimir o número de bytes dos seguintes tipos de dados:

(i) **short**;
(ii) **int**;
(iii) **long**;
(iv) **double**;
(v) **long double**.

A saída deste programa deve ser algo como: "`O tipo short ocupa 4 bytes`".

(b) Sabendo o número de bytes ocupado pelos tipos **short**, **int** e **long**, determine o intervalo de possíveis valores dos tipos:

(i) **short**;
(ii) **unsigned short**;
(iii) **int**;
(iv) **unsigned int**;
(v) **long int**;
(vi) **unsigned long**;

EP1.3) Escreva um programa em C que imprima as letras de `A` a `Z` (maiúsculas) e seus respectivos valores decimais.

EP1.4) Escreva um programa em C que receba um número inteiro como entrada e determine se o mesmo é par ou ímpar. O programa deve terminar quando um número inteiro negativo for introduzido. A saída do programa deve ser algo como: `O numero introduzido e' par`.