Declaração de Variáveis

No capítulo anterior aprendemos que as variáveis servem como nomes simbólicos que indicam unidades de armazenamento em um programa escrito em C. Uma variável possui um **rótulo** e um **conteúdo** (ou **valor**). Agora, introduzimos o **tipo de um dado**, que é a terceira característica da variável. Ainda existe o atributo de **escopo**, que é o tempo de vida da variável, conforme será estudado num capítulo posterior.

A variável é caracterizada por:
- Nome (rótulo)
- Tipo (domínio)
- Valor (conteúdo)
- Escopo (tempo de vida)

Tipos de Variáveis

O tipo determina o domínio, *i.e.* o conjunto de valores válidos, de uma variável e determina também as operações permitidas sobre a mesma. As operações disponíveis dependem de cada tipo. Na linguagem C, existem três famílias de tipos de variáveis: escalar, agregado e **`void`**. A família de escalares é formada por um grande número de tipos.

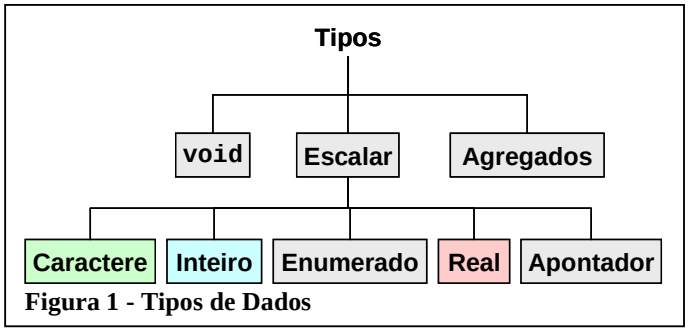

Figura 1 - Tipos de Dados

Os tipos **escalares** têm a capacidade de armazenar um único valor, que pode ser um número (inteiro ou fracionário), um caractere (que é um nome mais genérico para "símbolo"), um enumerador ou uma referência (ou apontador). Com exceção do apontador e do enumerador, estes tipos serão estudados neste capítulo.

Os tipos **agregados** são construções mais elaboradas, formadas pela composição de dois ou mais valores de um mesmo tipo (por exemplo, listas, vetores, matrizes, tabelas) ou de tipos diferentes (por exemplo, estruturas e uniões). Tipos agregados, apontadores e enumerações são estudados num capítulo mais à frente.

Por fim, existe o tipo *`void`*, que é um caso especial. Ele indica que não existe um tipo associado ao objeto correspondente ou o seu tipo é desconhecido. A rigor, não faz sentido declarar variáveis do tipo *`void`* . Ele será utilizado mais para o final do curso na declaração de funções e apontadores.

Declaração de Variáveis

Em um código escrito em C, *todas* as variáveis que serão utilizadas durante o programa precisam ser declaradas antes de seu primeiro uso no algoritmo. É através do tipo que o

compilador sabe quanta memória deve ser reservada para uma determinada variável (quantas células de memória são necessárias para armazenar o conteúdo da variável).

A declaração informa o nome da variável, o seu tipo e, opcionalmente, seu valor inicial. Com esta informação, o compilador destina automaticamente um número suficiente de células de memória, ou bytes, para armazenar esta variável.

A declaração de uma variável segue um dos dois modelos abaixo:

```
tipo nome = valor;
```

ou:

```
tipo nome;
```

O primeiro caso é a forma preferida para declaração de variáveis. Ela informa, em uma única declaração, as três características da variável: rótulo (nome), tipo (domínio de valores) e valor (conteúdo).

A segunda declaração é menos específica. Ela cria uma variável apenas com um nome e um tipo, mas sem um valor definido. Evidentemente, antes que o algoritmo utilize esta variável em algum cálculo, o programador deverá atribuir um valor à mesma em algum ponto anterior do código fonte.

Exemplo

Um programa deve calcular a média de um aluno em duas provas e em uma avaliação de laboratório. O programa precisará declarar as variáveis de forma semelhante ao exemplo a seguir:

```
float nota_prova_a = 8.0;
float nota_prova_b = 6.0;
float nota_laboratorio = 10.0;
float media;
```

São declaradas quatro variáveis: `nota_prova_a`, `nota_prova_b`, `nota_laboratório` e `media`. Todas elas são declaradas como sendo do tipo *`float`*, que será estudado neste capítulo. Somente as primeiras três recebem um valor na sua declaração. A variável `media` é declarada com valor indefinido, já que seu valor dependerá dos valores das três variáveis anteriores. O programa deve ser construído de maneira que, mais tarde, atribua um valor para a variável `media`.

Valor inicial das variáveis

Se a declaração já apresenta um valor inicial para a variável, o compilador o atribui à variável antes de iniciar a execução do algoritmo. Caso contrário, a variável conterá um valor arbitrário, impossível de se prever qual será, quando a execução do programa inicia. Não existe um “valor padrão” na linguagem C para variáveis que não recebem um valor inicial já na sua declaração.

É interessante entender o que acontece no computador como conseqüência da declaração de uma variável. Durante a execução do programa, a variável estará associada com posições reais de memória, que são um recurso compartilhado entre todos os programas em execução no computador.

Quando declaramos uma variável em um programa, ela será associada com uma (ou mais) célula de memória disponível logo antes de iniciar a execução do programa, *i.e.*, a variável não está associada a nenhum outro dado de nenhum outro programa que esteja executando no momento. Durante toda a execução do nosso programa, o computador garante que apenas nosso programa pode acessar e modificar o conteúdo dessa célula de memória. Porém, provavelmente em algum momento antes do início do nosso programa, esta célula foi utilizada por um outro programa que já terminou sua execução e deixou lá armazenado um valor qualquer. Note que, se a variável for declarada sem atribuição de valor inicial, a célula correspondente mantém o valor que já estava na memória quando nosso programa inicia sua execução, e que pode ser qualquer valor, imprevisível, deixado pelo outro programa. Se esquecermos de atribuir um valor à uma variável antes de usar seu conteúdo pela primeira vez no programa, então o valor lá encontrado será algo que certamente não desejamos usar em nosso algoritmo. Se usarmos esse valor arbitrário é certo que produziremos resultados errôneos. **Esquecer de atribuir um valor às variáveis antes de usar seu conteúdo pela primeira vez é uma causa comum de erros obscuros em C.**

Declaração múltipla

Caso desejemos declarar várias variáveis, todas do mesmo tipo, podemos listá-las em uma mesma linha, separando os nomes por vírgulas.

```
tipo nome1, nome2, nome3, ...;
```

Isso economiza alguns toques de digitação. Mesmo declarando em uma mesma linha, (algumas) variáveis podem ser inicializadas com valores.

```
tipo nome1 = valor, nome2, nome3 = valor3, ...;
```

Exemplo

O exemplo anterior poderia ser declarado assim:

```
float nota_prova_a = 8.0, nota_prova_b = 6.0;
float nota_laboratorio = 10.0;
float media;
```

A forma de declaração de variáveis é totalmente livre, e o programador deve utilizar aquela que deixe seu código mais legível.

Identificadores

Existem regras específicas que devem ser seguidas ao se escolher o nome de variáveis. Em C, os nomes (ou rótulos) das variáveis são denominados **identificadores**. As seguintes regras devem ser observadas ao se escolher um identificador para nomear uma variável:

- O identificador deve ser formado por uma seqüência de caracteres, os quais podem ser as letras maiúsculas ou minúsculas do alfabeto (a-z, A-Z), os dígitos numéricos (0-9), ou símbolo de sublinhado (_).
- Letras acentuadas não são permitidas.
- O identificador não pode começar com um dígito numérico.
- O identificador não pode ser uma das palavras reservadas da linguagem C (listadas mais adiante).

Exemplos

Identificadores que podem ser utilizados como nome de variáveis:
`contador`, `resto_divisao`, `media`, `nota1`, `nota2`, `PESO`

Maiúsculas/Minúsculas

O compilador distingue letras maiúsculas de minúsculas. Se dois identificadores diferenciam-se apenas pelo uso de maiúsculas e minúsculas, então o compilador C os tratará como duas variáveis diferentes. Por exemplo, `media` e `Media` são identificadores distintos. **Esta é uma causa freqüente de erros.**

Comprimento

Não existe limite para o comprimento máximo de um identificador. No entanto, a linguagem C padrão utiliza apenas os primeiros 31 símbolos para diferenciar os identificadores. Atualmente, a maioria dos compiladores modernos são mais flexíveis. Por exemplo, o compilador C da Microsoft trabalha com identificadores de até 247 símbolos.

Palavras reservadas da linguagem C (no padrão ANSI)

Algumas palavras não podem ser utilizadas como identificadores, pois são próprias da linguagem C. São elas:

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

Não é necessário decorá-las. Normalmente, um editor de código C apresenta as palavras reservadas em cor diferente, de forma que podemos saber imediatamente, por inspeção, se uma palavra é reservada ou se é um identificador.

Palavras reservadas na linguagem C (padrão Microsoft)

Além das palavras reservadas pela linguagem C (padrão ANSI), o compilador C da Microsoft reserva os seguintes identificadores:

__asm, **__based**, **__cdecl**, **__declspec**, **dllexport**, **dllimport**, **__except**, **__fastcall**, **__finally**, **__inline**, **__int16**, **__int32**, **__int64**, **__int8**, **__leave**, **naked**, **__stdcall**, **thread**, **__try**

Recomendações

A maioria dos programadores costuma seguir algumas convenções ao atribuir nomes às variáveis para tornar o código fonte mais uniforme. Tratam-se de boas práticas que melhoram substancialmente a legibilidade do código C. Cabe ao programador seguir estas recomendações ou não. Sugerimos:

- Utilizar sempre substantivos, evitar verbos.
- Quando o nome da variável é formado pela junção de duas palavras, usar o símbolo de sublinhado (_) entre as duas palavras consecutivas.
- Evitar identificadores que começam com o símbolo de sublinhado. Estes nomes são reservados para o compilador criar suas próprias variáveis, quando houver necessidade.

Tipos Inteiros

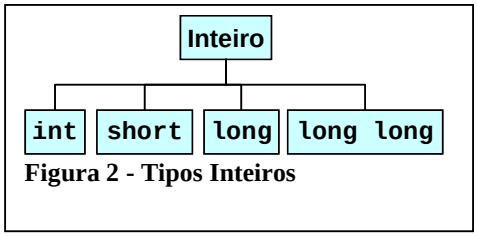

Figura 2 - Tipos Inteiros

O primeiro tipo que vamos estudar na linguagem C é usado para designar **números inteiros**. Eles caracterizam, portanto, variáveis cujo domínio será um intervalo dos números inteiros. Estas variáveis poderão conter números positivos ou negativos, desde que obrigatoriamente inteiros. Não é possível armazenar números com dígitos decimais em variáveis designadas como sendo do tipo inteiro. Devemos alertar, desde já, que a linguagem não suporta *todos* os inteiros. Isto é, vamos ver que existem limites para os valores absolutos que podem ser armazenados nas variáveis de tipos inteiros em C. O programador deve garantir que seu programa não extravasa esses valores, caso contrario resultados errôneos podem ser obtidos pelo programa.

O tipo inteiro comporta várias variantes. A diferença entre estas variantes está na quantidade de memória que é reservada para a variável e, portanto, estão diretamente relacionadas aos valores máximo e mínimo que elas poderão armazenar. A escolha do tipo para uma variável envolve um compromisso entre a amplitude dos números que ela poderá armazenar e o quanto de memória desejamos reservar para ela.

Os principais tipos inteiros são:

short short int	Números muito pequenos
short int	Números pequenos
long int	Números grandes
long long int	Números muito grandes

A linguagem C também define um tipo, denominado *int* (note a ausência de *short* ou *long*) que corresponde ao tipo inteiro que o processador consegue tratar com maior eficiência. Freqüentemente, chamamos *int* de **tipo inteiro padrão** da arquitetura do processador.

Exemplos

Declaração de variáveis para armazenar números pequenos:

```
short int numero_pequeno;
short int contador = 4;
```

Declaração de variáveis para armazenar números grandes:

```
long int quantidade_pecas;
long int numero_repeticoes = 3000000000;
```

Declaração de variáveis para armazenar números de tamanho padrão para a arquitetura do processador:

```
int contador;
int limite_tentativas = 100;
```

Intervalos de tipos inteiros

O que é um inteiro grande ou um inteiro pequeno? O tamanho do intervalo válido associado a cada um desta três variantes do tipo inteiro depende da arquitetura do computador, e também do compilador. Não existe uma padronização entre os compiladores C para esses valores.

Para escolher o tipo de número inteiro mais adequado para o seu programa, o programador precisa estudar seu algoritmo para saber qual o valor mínimo e máximo que cada variável, designada para conter números inteiros, poderá assumir durante a execução do programa. Em seguida, é necessário consultar a documentação do compilador, na qual encontra-se uma tabela com os valores mínimos e máximos de cada um destes tipos. O tipo desta variável deve ser aquele que comporta todo o intervalo de valores, do máximo ao mínimo, que poderão ser utilizados ou gerados pelo algoritmo.

A tabela a seguir apresenta os intervalos do domínio de cada um dos tipos de números inteiros, utilizando como referência a arquitetura Intel Pentium com compilador C da Microsoft.

Tipo	Descrição	Tamanho	Intervalo
int	Tamanho padrão	4 bytes	- 2.147.483.648 até 2.147.483.647
short short int	Números muito pequenos	1 byte	-256 até 257
short int	Números pequenos	2 bytes	-32.768 até 32.767
long int	Números grandes	4 bytes	- 2.147.483.648 até 2.147.483.647
long long int	Números muito grandes	8 bytes	$-9{,}223 \cdot 10^{15}$ até $9{,}223 \cdot 10^{15}$

Observação 1: Esta tabela refere-se ao compilador C da Microsoft e para uma arquitetura Intel Pentium. Outras implementações de compiladores C podem utilizar uma convenção diferente para tamanhos e intervalos definidos por estes tipos!

Observação 2: Especificamente para o compilador C da Microsoft, o tipo inteiro padrão e o número grande são equivalentes (respectivamente `int` e `long int`). Em outras palavras o número grande (`long int`) é justamente o tipo que o processador consegue tratar com maior eficiência.

Discussão sobre a escolha dos tipos

Para declarar variáveis de valor inteiro, a linguagem C dispõe de quatro tipos diferentes (`int`, `short int`, `long int` e `long long int`). A escolha do tipo mais adequado envolve um compromisso entre:

- O **valor mínimo** e **máximo** que a variável poderá representar (`short int` limita-se a números pequenos).
- A **quantidade de memória** ocupada (`short int` ocupa pouca memória e `long long int` ocupa mais memória).
- A **eficiência** de processamento (`int` é eficiente enquanto `short int` e `long long int` poderão implicar em alguma perda de eficiência).

A necessidade de distinguir entre tantos tipos inteiros fazia muito sentido nos computadores antigos, que tinham baixa capacidade de processamento e pouca disponibilidade de memória. Neles, a economia de memória e de processamento era uma questão importante para tornar viável a execução do algoritmo.

Os computadores atuais possuem uma quantidade de memória tão grande que, na prática, pode ser considerada infinita para a maioria dos programas. A preocupação de se utilizar o tipo `short int` para economizar memória perdeu muito de seu sentido prático.

A tecnologia dos processadores também evoluiu a ponto de realizar operações com a mesma eficiência para praticamente todos os tipos de números inteiros, independente se eles são `short int`, `long int` ou `long long int`.

> Exceto em aplicações onde o processamento ou o uso de memória são requisitos críticos de projeto, recomendamos utilizar sempre o tipo **`int`** para declarar variáveis cujo conteúdo será um número inteiro.

Escrever texto na tela

Escrever texto

Para escrever texto no monitor do computador utilizamos o comando `printf`. O texto é informado entre aspas duplas e entre parênteses após o `printf`, conforme o exemplo abaixo:

```
printf("mensagem");
```

Quando desejamos que a mensagem do próximo `printf` comece no início da próxima linha, a mensagem atual deve terminar com `\n`:

```
printf("mensagem\n");
```

Na verdade, a instrução `printf` não é primitiva na linguagem C. Ela é um comando mais complexo cuja descrição básica está no arquivo `stdio.h` , e que devemos incluir como diretiva para o compilador, sempre que nosso programa usar o `printf`. Programadores mais experientes já criaram o código para o comando `printf` e o disponibilizaram para uso em outros programas escritos em C. Assim, não precisamos nos preocupar com as idiossincrasias de saída de dados relativas ao computador para o qual estamos escrevendo nosso código C.

Escrever conteúdo de variáveis

Durante a execução do programa, a única forma para se conhecer o valor de uma variável é escrever seu conteúdo na tela. Para escrever o conteúdo das variáveis, o comando `printf` funciona de forma semelhante. Agora, a mensagem deve conter indicadores a serem substituídas pelo conteúdo das variáveis, cujos nomes devem ser passadas como os demais parâmetros no comando `printf`, conforme o exemplo abaixo:

```
printf("mensagem com indicações", variavel, variavel, ...);
```

Na mensagem entre aspas duplas, um indicador é um código iniciado por `%` (porcentagem). Para cada indicador que ocorre na mensagem deve haver a correspondente variável passada como parâmetro após a mensagem, e na mesma ordem em que os indicadores aparecem na mensagem.

Quando há um indicador, o comando `printf` consulta o valor da variável correspondente, formata-o usando a representação apropriada, especificada no indicador, e substitui o indicador pelo valor formatado ao imprimir a mensagem na tela.

Para cada tipo de variável (*`int`*, *`short int`*, *`long int`*, etc), existe um código apropriado para o indicador. Por exemplo, o indicador `%d` é substituído pelo conteúdo de uma variável de tipo *`int`*.

Exemplos

Vamos supor que o programa declarou as seguintes variáveis:

```
int quantidade = 10;
int nota1 = 6;
int nota2 = 7;
```

Alguns exemplos típicos de `printf` poderiam ser:

```
printf("Número de ítens comprados: %d\n", quantidade);
printf("Você comprou %d itens.\n", quantidade);
printf("O aluno tirou %d na primeira prova e %d na segunda.\n", nota1,
nota2);
```

O resultado na tela seria o seguinte:

```
Número de ítens comprados: 10
Você comprou 10 itens.
O aluno tirou 6 na primeira prova e 7 na segunda.
```

Tabela de indicações de formatação para escrita de valores inteiros (printf)

É necessário utilizar o indicador correto para cada tipo de variável. Caso contrário, o resultado poderá ser diferente do esperado. Todos os indicadores listados a seguir vão exprimir o conteúdo da variável usando a representação decimal, precedida por um sinal de menos (-) caso o número seja negativo. Existem os seguintes indicadores para os tipos inteiros:

%hd	ou %hi	para o tipo	*short int*
%d	ou %i	para o tipo	*int*
%ld	ou %li	para o tipo	*long int*
%lld	ou %lli	para o tipo	*long long int*

Controlando a forma da saída

O comando printf permite que se exerça algum controle sobre a forma da saída. O controle é indicado por símbolos colocados logo após o sinal de % nos indicadores de formatação.

Se nada for colocado, então o computador usa exatamente tantas posições para escrever o valor na tela quantas forem necessárias, incluindo uma posição para escrever o sinal de menos, se o valor for negativo. Por exemplo, o código abaixo

```
printf("Numero = '%d'", -850);
```

produziria a saída

```
Numero = '-850'
```

Note como foram usadas quatro posições entre as aspas: uma para o sinal e três para o valor. Note também que o valor a ser impresso pode ser informado como uma constante; no caso, -850. Como veremos mais adiante, o valor a ser impresso pode ser indicado também como o resultado de uma expressão cujo valor deve ser calculado.

Se colocarmos um número inteiro n após o sinal de % então o valor correspondente deve ser escrito em um campo de n posições. Se o valor correspondente ocupar mais do que as n posições reservadas, o computador ignora o número n e usa tantas posições quantas forem necessárias. Por outro lado, se número de posições for maior que o necessário para escrever o valor, então ele é escrito ajustado à direita. Por exemplo, o código abaixo

```
printf("Numero = '%8d'", -850);
```

produziria a saída

```
Numero = '    -850'
```

Nesse caso foram usadas oito posições entre as aspas: quatro brancos, uma para o sinal e três para o valor. E agora, o código

```
printf("Numero = '%1d'", -850);
```

produziria a saída

```
Numero = '-850'
```

Como um campo de uma posição é muito curto para se escrever o valor pretendido, o computador ignorou o tamanho do campo e usou as quatro posições necessárias.

Um sinal de menos logo após o sinal de % força o ajuste pela esquerda, e não pela direita. O código

```
printf("Numero = '%-8d'", -850);
```

produziria a saída

```
Numero = '-850    '
```

com oito posições, sendo que as quatro últimas são brancos.

Com esse tipo de controle, podemos gerar colunas bem alinhadas. O código

```
printf("Numeros\n");
printf(" %8d\n", -850);
printf(" %8d\n", 12345);
printf(" %8d\n", -4);
printf(" %8d\n", -90876);
```

produziria a seguinte saída alinhada pela direita

```
Numeros
    -850
   12345
      -4
  -90876
```

Existe também uma maneira de se controlar os tamanhos dos campos dinamicamente, quando o programa está executando. Para tal, colocamos um asterisco * logo após o sinal de %. Nesse caso, o tamanho do campo também deve ser informado como um parâmetro do comando printf, após as aspas duplas e logo antes do valor a ser impresso. Por exemplo, no comando abaixo

```
printf("Numeros = %d, %*d", 8, 5, -3);
```

o primeiro %d refere-se ao valor 8, que será impresso primeiro. Em seguida, na mensagem formatada, temos a vírgula e um branco, que são impressos logo após o valor 8. Depois, vemos o indicador %*d . Então, o próximo valor na lista após o 8, será usado para substituir o * nesse indicador, resultando em %5d . Este é o indicador que será usado para imprimir o próximo, e último, parâmetro na lista de valores, qual seja, o -3. Tudo se passa como se o comando printf acima fosse transformado para

```
printf("Numeros = '%d, %5d'", 8, -3);
```

produzindo como saída:

```
'8   -3'
```

Então o primeiro valor ocupou 1 posição (o suficiente para escrever 8) e o segundo valor ocupou 5 posições, sendo ajustado pela direita. Em geral, podemos associar variáveis ao símbolo *, tal como

```
printf("Numeros = '%d, %*d'", a, campo, b);
```

Nesse caso, o programa vai imprimir o valor da variável a com o número justo de posições, e vai usar o valor armazenado na variável campo como o tamanho do campo para imprimir o valor da variável b. Dependendo do valor armazenado na variável campo no momento da execução, o valor da variável b será impresso de uma forma ou de outra.

Leitura do teclado

O comando scanf armazena na posição de memória associada a uma variável o valor (conteúdo) digitado pelo usuário. Neste caso, o primeiro parâmetro do comando (escrito

entre aspas duplas) é formado por indicadores que determinam o tipo das variáveis que serão lidas. Os demais parâmetros são os nomes das variáveis (precedidos obrigatoriamente pelo símbolo &) que armazenarão os valores digitados. O comando scanf é utilizado conforme modelo a seguir:

```
scanf("indicadores", &variavel1, &variavel2, ...);
```

Os indicadores são semelhantes aos utilizados para a escrita com printf. Para ler um número inteiro e armazenar na variável v de tipo *int*:

```
int v;
scanf("%d", &v);
```

Para ler três números separados por espaços e armazená-los, respectivamente, nas variáveis x, y, z, de tipo *int*:

```
int x, y, z;
scanf("%d %d %d", &x, &y, &z);
```

O comando scanf também não é uma instrução primitiva da linguagem C . Sua descrição básica consta no arquivo stdio.h que deve ser incluído antes do código do programa, através de uma diretiva #include.

Tabela de indicações para leitura de valores inteiros (scanf)

Para ler números inteiros, utilize os seguintes indicadores no primeiro parâmetro do comando scanf. *i.e.*, entre as aspas duplas. Todos eles especificam que desejamos ler um número inteiro na representação decimal, podendo ser precedido por um sinal de menos (-) para números negativos ou um sinal de mais (+), este opcional, para números positivos.

%hd	ou %hi	para tipos	*short int*
%d	ou %i	para tipos	*int*
%ld	ou %li	para tipos	*long int*
%lld	ou %lli	para tipos	*long long int* (ANSI C)
%I64d	ou %I64i	para tipos	*long long int* (Microsoft C)

Exemplo: Leitura e escrita de dados inteiros

Este programa demonstra diferentes formas de uso dos comandos printf e scanf para escrever e imprimir variáveis do tipo *short int* e *long int*.

Código fonte:

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

int main(int argc, char* argv[]) {
     // Declarar variáveis
     short int pequenoA, pequenoB;
     long int grandeA, grandeB;

     // Ler um valor de cada vez:
     printf("Digite um número pequeno: ");
     scanf("%hd", &pequenoA);
     printf("Digite outro número pequeno: ");
     scanf("%hd", &pequenoB);

     // Ler dois valores em uma única operação:
```

```
        printf("Digite dois números grandes, separados por espaço: ");
        scanf("%ld %ld", &grandeA, &grandeB);

        // Escrever dois valores em uma única operação
        printf("Numeros: %hd e %hd\n", pequenoA, pequenoB);

        // Escrever um valor de cada vez
        printf("Numeros: %ld\n", grandeA);
        printf("Numeros: %ld\n", grandeB);

        return 0;
}
```

Consulte programas: TiposInteiros\LeituraEscrita\LeituraEscrita

Descrição passo a passo:

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

As primeiras duas linhas do programa são diretivas de compilador que fazem referência às bibliotecas padrão do C. Sua inclusão garante que os comandos `printf` e `scanf` estejam disponíveis para uso no programa.

```
short int pequenoA, pequenoB;
```

O programa começa declarando todas as variáveis que utilizará. Esta linha define duas variáveis, chamadas `pequenoA` e `pequenoB`. Elas armazenarão dois números pequenos que serão lidos do teclado. As duas são do tipo *`short int`*, mas não possuem um valor inicial. Como as características dessas variáveis são todas iguais, pode-se declará-las todas na mesma linha, separando seus nomes por vírgulas.

Repare que nenhuma dessas variáveis está recebendo um valor inicial. Então, quando executar, que valores conterão? É impossível prever. Felizmente este problema será resolvido quando o conteúdo das variáveis for lido com o comando `scanf`.

```
long int grandeA, grandeB;
```

A segunda declaração define dois números grandes que serão lidos do teclado.

```
printf("Digite um número pequeno: ");
scanf("%hd", &pequenoA);
printf("Digite outro número pequeno: ");
scanf("%hd", &pequenoB);
```

Para obter o valor das variáveis `pequenoA` e `pequenoB`, utilizamos duas vezes o comando `scanf` para ler o valor de cada uma. Utilizamos o indicador `%hd` para informar que desejamos ler um número na representação decimal e converter para uma variável de tipo *`short int`* . **Note a existência do `&` na frente dos nomes das variáveis para o funcionamento correto do `scanf`. Essa é uma fonte de erros em C.**

Note que a mensagem solicitando que o usuário digite os números é gerada pelo comando `printf`, não por `scanf`! Como o comando `printf` não termina com `\n`

então o cursor não é deslocado para a próxima linha após a mensagem ser impressa. Assim, o usuário continua digitando os valores de entrada logo após a mensagem solicitando o dado. Porém, como o usuário deve apertar a tecla "enter" (ou "return") logo após digitar o primeiro valor (que vai ser armazenado em pequenoA), o cursor é imediatamente deslocado para a próxima linha, preparando o terreno para que a mensagem do segundo `printf` comece logo na primeira coluna da nova linha. Por fim, note que o computador automaticamente imprime na tela os dígitos que vão sendo teclados, mesmo o programa *não tendo solicitado* isso explicitamente. Essa impressão dos dados digitados é chamada de *eco na tela*.

```
printf("Digite dois números grandes, separados por espaço: ");
scanf("%ld %ld", &grandeA, &grandeB);
```

Para as variáveis `grandeA` e `grandeB`, utilizamos o comando `scanf` de uma forma diferente. As variáveis são lidas de uma só vez, ou seja, o usuário precisa escrever os dois valores na mesma linha e o `scanf` reconhecerá os dois números. Utilizamos o indicador `%ld` para informar que desejamos ler um número na representação decimal e armazenar numa variável de tipo `long int`. Observe o espaço entre os dois indicadores `%ld`. Isso indica que os números devem vir separados por um ou mais *espaços em branco* (que, para a rotina `scanf` podem ser tanto o "branco" comun, o "tab", ou o "return"). Porém, se colocarmos outros caracteres na mensagem do comando `scanf` (que vem entre aspas duplas) entre os dois indicadores `%ld`, então o computador exigirá um casamento perfeito entre a entrada de dados e o formato indicado no comando `scanf`. Por exemplo, se escrevermos

```
scanf("%ld A %ld", &grandeA, &grandeB);
```

então os dados deverão estar separados por um ou mais espaços em branco, seguidos do caractere "A", seguido de um ou mais espaços em branco. Se a forma de entrada dos dados não seguir a especificação do comando, então resultados inesperados podem ocorrer, dependendo de cada compilador. Para segurança, consulte o manual do seu compilador para ver como o comando `scanf` é lá descrito.

Continuamos agora com o programa original.

```
printf("Numeros: %hd e %hd\n", pequenoA, pequenoB);
```

A mensagem do comando `printf` contém dois indicadores `%hd`, ou seja, ele irá substituir o primeiro pelo valor da variável `pequenoA` e o segundo pelo valor da variável `pequenoB` . Antes de realizar a substituição, ele formata os dois valores para a representação decimal. O símbolo `\n` indica que o próximo `printf` escreverá no início da linha seguinte.

```
printf("Numeros: %ld\n", grandeA);
printf("Numeros: %ld\n", grandeB);
```

Eis uma forma alternativa de escrever números, utilizando um comando `printf` para cada variável. Note que, como as variáveis são do tipo `long int`, necessitamos do indicador `%ld` para realizar a substituição e formatação.

```
return 0;
```

Para terminar o programa, devemos sempre utilizar o comando `return 0`.

Exemplo de Execução:

Os valores digitados pelo usuários estão em *itálico*.

```
Digite um número pequeno: 300
Digite outro número pequeno: 200
Digite dois números grandes, separados por espaço: 100000 300200
Números: 300 e 200
Números: 100000
Números: 300200
```

Observações importantes

Quando a execução do programa chega ao comando `scanf`, o programa permanece suspenso até que o usuário digite todos os valores esperados pelo `scanf` e pressione a tecla *enter*.

Se um comando `scanf` requer mais de um valor, como no exemplo da leitura de `x`, `y` e `z` abaixo, isto não implica que o usuário deve digitar os valores na mesma linha. Por exemplo:

```
scanf("%d %d %d", &x, &y, &z);
```

Tanto faz o usuário digitar:

```
3 4 5
```

Ou:

```
3
4
5
```

Se o usuário digitar mais números que o esperado pelo `scanf`, os números em excesso ficarão armazenados na fila de entrada para serem lidos automaticamente pelos próximos comandos `scanf`.

O primeiro parâmetro do `scanf` contém somente indicadores. O programa tentará reconhecer o texto digitado pelo usuário de acordo com o indicador. Para escrever uma mensagem informando ao usuário o que ele deve digitar, utilize o comando `printf` antes do `scanf`.

```
printf("Por favor, digite três números: ");
scanf("%d %d %d", &x, &y, &z);
```

O resultado será (a parte digitada pelo usuário está em *itálico*):

```
Por favor, digite três números: 3 4 5
```

O exemplo a seguir está **errado**:

```
scanf("Por favor, digite três números: %d %d %d", &x, &y, &z); (errado!)
```

Se, por algum motivo, o usuário não digitar texto no formato correto conforme indicado exatamente pela mensagem entre aspas, com os indicadores substituídos pelos valores a serem lidos, então o comando `scanf` interromperá a leitura sem preencher o conteúdo das variáveis.

Introdução aos Operadores

Agora vamos estudar algumas operações básicas que podem ser realizadas sobre as variáveis de tipo inteiro.

Atribuição

Após a declaração de uma variável, é possível atribuir-lhe um novo valor. Esta operação é indicada pelo operador "=".

Note que, em C, o sinal = não significa o mesmo que igualdade matemática!

Para atribuir um valor a uma variável, a sintaxe é bem simples:

```
variavel = valor;
```

Na atribuição, o operador = aparece entre os dois lados da operação. O lado esquerdo deve ser obrigatoriamente uma variável. O lado direito deve ser um número, uma outra variável ou uma expressão matemática cujo resultado final seja um número.

O efeito da atribuição é simples. Primeiro, calcula-se o resultado (valor) da expressão à direita. Depois, armazena-se o resultado na variável à esquerda do operador =.

É importante garantir que o tipo da variável à esquerda seja compatível com o gerado pela expressão à direita. Caso contrário, a atribuição poderá ocorrer de forma diferente da esperada. **Este é um erro freqüente**.

Exemplo

Suponha que existam variáveis chamadas quantidade, nota1 e nota2, todas declaradas como *int* . Vamos atribuir, respectivamente, a cada uma dessas variáveis, os novos valores 10, 6 e 7:

```
quantidade = 10;
nota1 = 6;
nota2 = 7;
```

A atribuição pode ser também o resultado de uma expressão matemática:

```
soma = nota1 + nota2;
contador_novo = contador_velho + 1;
```

Outros Operadores

Para números inteiros, a linguagem C define um operador para cada uma das quatro operações básicas (adição, subtração, multiplicação, divisão). No caso da divisão, existe um operador para se obter o quociente e outro para se obter o resto. Também há um operador para se obter o negativo de um valor.

Expressões envolvendo números inteiros, em C, são muito semelhantes à expressões matemáticas tradicionais com as quais já estamos familiarizados.

Os operadores realizam as operações sobre dois valores, que devem ser resultados da avaliação de expressões de tipos compatíveis.

Para os tipos inteiros, a linguagem C define os seguintes operadores:

Soma	+	(símbolo de adição)
Subtração	-	(símbolo de subtração, ou de negativo)
Multiplicação	*	(asterisco)
Divisão	/	(barra)
Resto	%	(porcento)

Os operadores matemáticos normalmente são utilizados em conjunto com o operador de atribuição para armazenar o resultado em uma variável, como podemos observar nos próximos exemplos.

Também é permitido utilizar parênteses para agrupar operadores de uma expressão, indicando a ordem na qual as operações devem ser realizadas. As regras para o uso de parênteses são as mesmas a que já estamos acostumados na aritmética.

Exemplos

Soma: operador +:

```
int parcela1 = 10;
int parcela2 = 16;
int soma;
soma = parcela1 + parcela2;
printf("Soma: %d mais %d é %d", parcela1, parcela2, soma);
```

Imprime:

```
Soma: 10 mais 16 é 26
```

Note que primeiro será realizada a soma, depois a atribuição.

Subtração: operador -:

```
int parcela1 = 10;
int parcela2 = 16;
int subtracao;
subtracao = parcela1 - parcela2;
printf("Subtração: %d menos %d é %d", parcela1, parcela2, subtracao);
```

Imprime:

```
Subtração: 10 menos 16 é -6
```

Multiplicação: operador * (asterisco):

```
int fator_a = 4;
int fator_b = 6;
int produto;
produto = fator_a * fator_b;
printf("Multipliação: %d vezes %d é %d", fator_a, fator_b, produto);
```

Imprime:

```
Multiplicação: 4 vezes 6 é 24
```

Divisão: operador /(barra):

Quando aplicado a números inteiros, o operador de divisão não realiza a operação exata, mas gera o quociente da divisão.

```
int dividendo = 46;
int divisor = 6;
int quociente;
quociente = dividendo / divisor;
printf("Divisão: %d dividido por %d é %d", dividendo, divisor, quociente);
```

Imprime:

```
Divisão: 46 dividido por 6 é 7
```

Resto: operador % (porcento):

Código em C:

```
int dividendo = 46;
int divisor = 6;
int quociente;
int resto;
quociente = dividendo / divisor;
resto = dividendo % divisor;
printf("Divisão: %d dividido por %d é %d, resto %d", dividendo, divisor,
quociente, resto);
```

Imprime:

```
Divisão: 46 dividido por 6 é 7, resto 3
```

Exemplo:

Este programa pergunta o tempo decorrido entre dois eventos, que deve ser dado em segundos. O programa calcula, então, quantas horas, minutos e segundos correspondem a este intervalo de tempo.

Código fonte:

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

int main(int argc, char* argv[]) {
	// Declarar variáveis
	int horas, minutos, segundos;
	int total_segundos;

	// Pedir ao usuário para escrever intervalo de tempo (em segundos)
	printf("Digite o intervalo de tempo (segundos): ");
	scanf("%d", &total_segundos);

	// Cálculos;
	horas    = (total_segundos / 60) / 60;
	minutos  = (total_segundos / 60) % 60;
	segundos = total_segundos % 60;

	// Imprimir resultados
	printf("\n");
	printf("Total de segundos: %d \n", total_segundos);
	printf("Tempo: %d:%d:%d\n", horas, minutos, segundos);

	return 0;
}
```

Consulte: IntroducaoOperadores\Horario01\Horario01.vcproj

Descrição passo a passo:

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

As primeiras duas linhas do programa são diretivas de compilador para que este inclua as bibliotecas padrão da linguagem C. Elas garantirão que os procedimentos `printf` e `scanf` estarão disponíveis para serem usados no programa.

```
int horas, minutos, segundos;
```

O programa começa declarando todas as variáveis que utilizará. Esta linha define três variáveis, chamadas `horas`, `minutos` e `segundos`. Elas armazenarão o resultado dos cálculos realizados. Todas elas são do tipo `int` e não possuem um valor inicial. Como as características dessas variáveis são todas iguais, pode-se declará-las todas na mesma linha, separando seus nomes com vírgulas.

Como nenhuma dessas variáveis está recebendo um valor inicial, devemos tomar o cuidado de atribuir um valor a elas, antes de utilizá-las para cálculos no programa.

```
int total_segundos;
```

A segunda declaração é o valor que será digitado do usuário, que ficará armazenado na variável chamada `total_segundos`.

```
printf("Digite o intervalo de tempo (segundos): ");
scanf("%d", &total_segundos);
```

Para obter o valor da variável `total_segundos`, utilizamos o procedimento `scanf` para ler seu valor. O indicador `%d` informa que desejamos ler um número na representação decimal e converter para uma variável de tipo `int`.

Note a presença do `&` na frente do nome da variável `total_segundos` para o funcionamento correto do `scanf`. Nova fonte de erros.

```
horas    = (total_segundos / 60) / 60;
```

Esta linha é uma atribuição. O lado direito do operador `=` é uma expressão matemática que calcula o número de horas, dado o valor em segundos. O símbolo `/` indica operador de divisão inteira. Note que dividimos duas vezes por 60 (uma vez para transformar segundos em minutos, e uma segunda vez para transformar minutos em horas). O valor resultante será armazenado na variável `horas`. O uso dos parênteses, a rigor, é opcional, já que a divisões sucessivas são realizadas da esquerda para direita.

```
minutos  = (total_segundos / 60) % 60;
segundos = total_segundos % 60;
```

Estas duas linhas também são atribuições. Dividimos o total de segundos por 60, para obter o número de minutos. Depois, calculamos o resto da divisão por 60, para

saber o número de minutos que excedem as horas. Note que, neste caso, o uso de parênteses é obrigatório, para garantir que primeiro será realizada a divisão de `total_segundos` por 60, e somente depois será realizado o cálculo do resto.

O número de segundos é obtido de forma semelhante, calculando o resto da divisão por 60, para obter o número de segundos que excedem os minutos.

```
printf("\n");
printf("Total de segundos: %d \n", total_segundos);
printf("Tempo: %d:%d:%d\n", horas, minutos, segundos);
```

Por fim, o programa não seria de utilidade se não informasse o resultado de forma visível para o usuário. O primeiro comando `printf` escreve uma linha em branco, apenas para tornar a saída do texto mais organizada. O segundo `printf` escreve novamente o total de segundos digitado pelo usuário. Note que utilizamos o indicador `%d` para informar que ele será substituído pela representação decimal do valor da variável `total_segundos`.

```
return 0;
```

Para terminar o programa, devemos sempre utilizar `return 0`.

Exemplo de Execução:

```
Digite o intervalo de tempo (segundos): 1085

Total de segundos: 1085
Tempo: 0:18:5
```

Melhorias no código:

Podemos aplicar uma pequena melhoria no código do cálculo das horas, minutos e segundos. Note que, para determinar tanto o número de horas quanto o número de minutos realizamos divisões por 60. Para evitar a repetição desta operação, poderíamos guardar o resultado da divisão em uma variável auxiliar.

```
int total_minutos;
.
.
.
segundos = total_segundos % 60;
total_minutos = total_segundos / 60;

minutos = total_minutos % 60;
horas = total_minutos / 60;
```

Consulte: IntroducaoOperadores\Horario02\Horario02.vcproj

Note como a variável `total_minutos` foi reutilizada para armazenar o valor intermediário dos minutos. Com isso, economizamos um cálculo de divisão.

Este exemplo comprova que não existe uma única solução para escrever um algoritmo correto para uma tarefa dada. Não existe um algoritmo “mais correto” que outro. O programador deve preferir aquele mais simples de entender.

Tipo Caractere

Para realizar processamento de texto e trabalhar com letras do alfabeto e outros caracteres como, por exemplo, símbolos de pontuação, a linguagem C também representa s caracteres como números inteiros, com algumas facilidades específicas. Desta forma, **tudo que aprendemos para números inteiros pode ser aplicado para caracteres**.

As variáveis que armazenam um caractere são declaradas como sendo do tipo `char`. Esse tipo nada mais é que uma outra denominação para o tipo `short short int`. Uma variável desse tipo usa apenas um byte da memória, podendo acomodar valores inteiros desde -256 até +255, como podemos extrair da nossa tabela de valores para inteiros apresentada anteriormente. No entanto, quando uma variável é declarada como do tipo `char` o programador já indica sua intenção de utilizar a variável para armazenar símbolos.

A correspondência entre símbolos e números é dada pela tabela ASCII (American Standard Code for Information Interchange). Esta tabela utiliza os números de 0 até 127 para letras do alfabeto inglês e para os sinais de pontuação mais comuns. Por exemplo, na entrada 65 da tabela ASCII encontramos a letra 'A'. Assim, uma variável do tipo `char` cujo conteúdo seja 65 representaria também a letra 'A'. Infelizmente não existe consenso de como tratar os caracteres numerados a partir de 128, onde estariam os caracteres acentuados, e isso é causa de incompatibilidades entre programas que compilam e executam em plataformas diferentes.

Uma das facilidades da linguagem C para manipular caracteres é a conversão automática entre letras e números da tabela ASCII. Assim, para atribuir uma letra a uma variável do tipo `char`, basta escrever a letra entre aspas simples. Por exemplo:

```
char letra = 'A';
```

O resultado dessa atribuição é que o conteúdo da variável `letra` será uma seqüência de 8 bits (que é o tamanho de um `short short int`) que, se interpretada como um número resultará em 65. De forma semelhante, podemos atribuir a mesma letra ‘A’ à variável `letra` diretamente, usando seu respectivo código ASCII que é 65:

```
char letra = 65;
```

A primeira atribuição, entretanto, é bem mais legível. Não precisamos conhecer a tabela ASCII. Simplesmente escrevemos a letra desejada (entre aspas simples) e deixamos o compilador realizar a conversão.

Uma informação útil, entretanto, é que as letras a-z têm numeração *consecutiva* na tabela ASCII. O mesmo vale para as letras A-Z e para os dígitos 0-9.

Escrita

Para escrever um caractere, utiliza-se o comando `printf` com um dos seguintes indicadores:

`%c`	`char`	escreve caractere
`%hhd, %hhi`	`char`	escreve o código ASCII do caractere

Exemplo

```
char letra_a = 'a', letra_W = 'W';
printf("Letra %c tem código %hhd", letra_a, letra_a);
printf("Letra %c tem código %hhd", letra_W, letra_W);
```

Imprime:

```
Letra a tem código 97
Letra W tem código 87
```

Observe como o indicador %c causa a impressão do símbolo correspondente ao valor que está armazenado na variável. O compilador faz a conversão automaticamente. Já o indicador %hhd causa a impressão do valor numérico e na representação decimal.

Leitura

A leitura de caractere é realizada com o comando scanf, usando os seguintes indicadores de formatação :

%c	*char*	lê um caractere
%hhd, %hhi	*char*	lê o código ASCII do caractere
%c (espaço em branco na frente)	*char*	lê próximo caractere que não é caractere de espaçamento

O primeiro indicador (%c) encontrado na mensagem, força a leitura do próximo caractere, independente da letra ou pontuação que ele representa. O último indicador (%c com espaço em branco na frente) significa que desejamos ler a próxima letra, ignorando caracteres de espaçamento.

Se desejamos ler uma letra digitada pelo usuário, devemos usar scanf(" %c"), caso contrário, o programa vai ler o próximo caractere da entrada, mesmo que ele seja um espaço em branco como, por exemplo, o último return que foi teclado. Note que espaços em branco também correspondem a símbolos (existe uma tecla para eles!) e, assim, eles também têm um índice na tabela ASCII (usualmente é o de número 40).

No início as operações de leitura e escrita usando caracteres, especialmente as operações de leitura, podem parecer confusas. Se tal é o caso, tente evitá-las enquanto ganha mais proficiência com a linguagem C. Uma boa forma remover confusões é imaginar que tudo que teclamos (inclusive a tecla “espaço”!) é um caractere que é colocado numa *fila de entrada*. Quando invocamos um comando scanf, o computador acessa essa fila e retira de lá exatamente quantos caracteres sejam necessários para obter os valores de que precisa. Os demais caracteres são deixados na fila (inclusive aqueles correspondentes aos return que digitamos!). Se teclarmos mais, os novos caracteres entram *todos* na fila, na ordem em que foram teclados. Se invocarmos novos comandos scanf os caracteres mais no início da fila são removidos até satisfazer os comandos scanf. Uma boa maneira de verificar quais caracteres um comando scanf leu seria imprimir o conteúdo das variáveis onde esses caracteres foram armazenados, mas agora usando indicadores de formatação %d, que informam os valores decimais contidos nas variáveis.

Os caracteres de espaçamento correspondem ao "branco", ao "tab" e ao "return". Podemos fazer esses caracteres aparecerem nas mensagens impressas de várias formas. Eis duas delas:

1. No comando `printf` podemos usar o espaço, ou o '\t' ou o '\n' para causar a impressão de cada um deles, respectivamente.
2. Podemos armazenar numa variável tipo `char` o seu valor numérico e usar o indicador `%c` no comando `printf` para imprimi-los.

Em resumo, na linguagem C símbolos são, na realidade, tratados como valores inteiros pequenos. O programador pode manipular o conteúdo dessas variáveis como se fossem inteiros, inclusive usando todas as operações sobre inteiros. O compilador, porém, aceita a sintaxe especial das aspas simples que permite escrever o símbolo diretamente ao invés de seu valor numérico correspondente. Além disso, o compilador faz a conversão automática entre o valor numérico e o símbolo correspondente, e vice-versa.

Você já tem condições de fazer um programa para descobrir qual é o símbolo associado a qualquer inteiro pequeno. Tente!

Outros Tipos Inteiros

Já estudamos como a linguagem C utiliza as palavras `short` e `long` para definir tipos de números inteiros com diferentes intervalos para os possíveis valores. A linguagem C também permite informar se a variável aceita ou não um sinal, ou seja, se ela aceita números negativos.

Sobre a diversidade de tipos em C

A rigor, atualmente, o programador não precisa se preocupar com o intervalo dos números usados na maioria dos programas. Todos os tipos numéricos são processados com praticamente a mesma eficiência e a memória disponível é tão grande que pode ser considerada infinita. Não faz mais muito sentido optar por declarar uma variável como `short int` apenas para economizar memória ou como `int` para tornar o programa mais rápido.

Da mesma forma, para os programas mais comuns não há necessidade em se preocupar se uma variável possui ou não sinal. Declarando todas as variáveis inteiras como do tipo `int`, o programa poderá operar tanto com números positivos como com números negativos.

Assim, para contornar a complexidade gerada pelo excesso de tipos de números inteiros, recomenda-se utilizar somente o tipo `int`, que é o tipo inteiro mais genérico.

No entanto, ainda existe muito código fonte C antigo e é necessário pelo menos conhecer todos os tipos da linguagem C.

Tipos com sinal

Até o momento, encontramos os tipos inteiros `int`, `short int`, `long int` e `long long int`. O tipo `char` também pode ser utilizado como um caso especial de tipo inteiro que representa um símbolo. Todos estes tipos representam números inteiros, cujo domínio é um intervalo (quase) simétrico de números negativos e positivos. Por este motivo, estes tipos são também denominados de tipos **com sinal**. Para ressaltar esta propriedade, a declaração das variáveis pode ser realizada adicionando-se opcionalmente a palavra `signed` ao nome do tipo. Normalmente, para deixar o programa mais enxuto e mais fácil de ler, costuma-se omitir a palavra `signed`.

Os seguintes tipos são equivalentes (declaram o mesmo tipo de variável):

Tipo com sinal	Tipo equivalente com sinal
`char`	`signed char`
`int`	`signed int`
`short short int`	`signed short short int`
`short int`	`signed short int`
`long int`	`signed long int`
`long long int`	`signed long long int`

Tipos sem sinal

Para cada um destes tipos inteiros, C oferece um tipo correspondente que representa um número não negativo. Ao invés do domínio do tipo conter tanto números inteiros negativos como positivos, estes tipos representam apenas números positivos. Por este motivo, estes tipos são também denominados de tipos **sem sinal**. Para ressaltar esta propriedade, a declaração das variáveis deve ser realizada adicionando-se a palavra obrigatória `unsigned` ao nome do tipo. Todos os conceitos aprendidos para variáveis com sinal também são válidos para variáveis sem sinal.

A tabela a seguir apresenta estes novos tipos.

Tipo com sinal	Tipo equivalente com sinal	Equivalente sem sinal
`char`	`signed char`	`unsigned char`
`int`	`signed int`	`unsigned int`
`short short int`	`signed short short int`	`unsigned short short int`
`short int`	`signed short int`	`unsigned short int`
`long int`	`signed long int`	`unsigned long int`
`long long int`	`signed long long int`	`unsigned long long int`

A próxima tabela mostra os domínios de cada um dos tipos inteiros para o compilador C da Microsoft. Repare como os tipos sem sinal comportam valores positivos além daqueles que podem ser armazenados em tipos com sinal.

Tipo	Tamanho	Domínio
char ou *signed char*	1 byte	- 128 até 127
int ou *signed int*	4 bytes	- 2.147.483.648 até 2.147.483.647
short int ou *signed short int*	2 bytes	- 32.768 até 32.767
long int ou *signed long int*	4 bytes	- 2.147.483.648 até 2.147.483.647
long long int ou *signed long long int*	8 bytes	$-9{,}223 \cdot 10^{15}$ até $9{,}223 \cdot 10^{15}$
unsigned char	1 byte	0 até 255
unsigned int	4 bytes	0 até 4.294.967.296
unsigned short int	2 bytes	0 até 65.536
unsigned long int	4 bytes	0 até 4.294.967.296
unsigned long long int	8 bytes	0 até $18{,}446 \cdot 10^{15}$

Escrita

A tabela a seguir resume todos os indicadores para todos os tipos encontrados, para uso com o comando `printf`:

`%hhd, %hhi`	*char* ou *signed char*
`%d, %i`	*int* ou *signed int*
`%hd, %hi`	*short int* ou *signed short int*
`%ld, %li`	*long int* ou *signed long int*
`%lld, %lli, %I64i`	*long long int* ou *signed long long int*
`%hhu`	*unsigned char*
`%u`	*unsigned int*
`%hu`	*unsigned short int*
`%lu`	*unsigned long int*
`%llu, %I64u`	*unsigned long long int*

Observação: O compilador C da Microsoft não aceita os indicadores `%lld`, `%lli` e `%llu`. Deve-se utilizar, respectivamente, `%I64d`, `% I64i` e `%I64u`.

Leitura

Para o comando `scanf`, os indicadores disponíveis são:

`%hhd, %hhi`	*`char`* ou *`signed char`*
`%d, %i`	*`int`* ou *`signed int`*
`%hd, %hi`	*`short int`* ou *`signed short int`*
`%ld, %li`	*`long int`* ou *`signed long int`*
`%lld, %lli, %I64i`	*`long long int`* ou *`signed long long int`*
`%hhu`	*`unsigned char`*
`%u`	*`unsigned int`*
`%hu`	*`unsigned short int`*
`%lu`	*`unsigned long int`*
`%llu, %I64u`	*`unsigned long long int`*

Tipos Ponto Flutuante (Fracionários)

Além dos números inteiros, a linguagem C define três tipos de números cujos domínios são números fracionários: são tambémchamados de números de tipo **ponto flutuante**.

Os três tipos de ponto flutuante são:

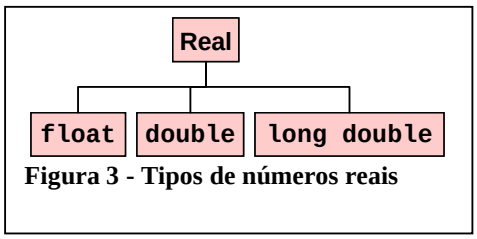

Figura 3 - Tipos de números reais

`float`	Pouca precisão, baixa magnitude
`double`	Muita precisão, alta magnitude
`long double`	Precisão ainda maior, altíssima magnitude

Tal como para as variáveis inteiras, quanto maior a precisão e magnitude, mais memória é necessária para armazenar o conteúdo de variáveis daquele tipo. No computadores antigos, as variáveis de tipo *`float`* eram processadas com maior eficiência. Hoje, não existe um tipo de ponto flutuante preferido para cada processador. A arquitetura Intel Pentium implementa as operações sobre *`float`* e *`double`* praticamente com a mesma eficiência.

Exemplos

Declaração de variáveis de pouca precisão:

```
float raio = 5.4;
float area = 50040.22;
```

Declaração de variáveis com alta precisão:


```
double velocidade = 5.333222567854;
```

Intervalos de ponto flutuante

Tipo	Tamanho	Precisão aproximada	Magnitude aproximada
float	4 byes	7 dígitos	$-3{,}4\cdot10^{38}$ até $3{,}4\cdot10^{38}$
double	8 byes	15 dígitos	$-1{,}7\cdot10^{308}$ até $1{,}7\cdot10^{308}$
long double	10 bytes	19 dígitos	$-1{,}2\cdot10^{4932}$ até $1{,}2\cdot10^{4932}$

Escrita

Para escrever números de ponto flutuante, utiliza-se o comando `printf`, de forma muito semelhante àquela vista para valores inteiros. São os seguintes os indicadores de formatação para lidar com valores fracionários:

%e	*double/float*	usa notação científica (ex.: 4.56e-3)
%f	*double/float*	usa representação decimal (ex.: 0.00456)
%g	*double/float*	escolhe entre notação científica e representação decimal, escolhendo a forma mais compacta.
%Le	*long double*	usa notação científica (ex.: 4.56e-3)
%Lf	*long double*	usa representação decimal (ex.: 0.00456)
%Lg	*long double*	escolhe entre notação científica e representação decimal, escolhendo a forma mais compacta.

Exemplo

```
float numero   = 156.40;
printf("O número: %f %e %g\n", numero, numero, numero);
numero = 1560000.0;
printf("O número: %f %e %g\n", numero, numero, numero);
```

Produzirá:

```
O número: 156.399994 1.564000e+02 156.4
O número: 1560000.000000 1.560000e+06 1.56e+06
```

Ao invés de imprimir 156.4, o programa imprime 156.399994. A explicação está na próxima observação.

Observação

O tipo ponto flutuante não é capaz de representar qualquer número real com precisão absoluta, ele os aproxima para frações que satisfazem a seguinte expressão:

$$valor = sinal \cdot mantissa \cdot 2^{expoente}.$$

E esta aproximação é boa o suficiente para praticamente todas as aplicações. No entanto, para aplicações financeiras que requerem apuração rigorosa de valores, não devemos utilizar variáveis de tipo ponto flutuante. Basta multiplicar os valores por 100 e usar inteiros.

Leitura

A leitura de valores em ponto flutuante é realizada com o comando scanf, de forma muito semelhante àquela vista para valores inteiros. Utiliza-se os seguintes indicadores de formatação:

%e	*float*	lê em notação científica
%f	*float*	lê em representação decimal
%g	*float*	lê em qualquer uma das duas formas
%le	*double*	lê em notação científica
%lf	*double*	lê em representação decimal
%lg	*double*	lê em qualquer uma das duas formas
%Le	*long double*	lê em notação científica
%Lf	*long double*	lê em representação decimal
%Lg	*long double*	lê em qualquer uma das duas formas

Exemplo: operações com números fracionários

Calcular o perímetro e área de um círculo, cujo raio foi informado pelo usuário.

Código fonte

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

int main(int argc, char* argv[]) {
     // Declarar variáveis
     double pi = 3.141592;
     double raio, area, perimetro;

     // Pedir ao usuário escrever o raio
     printf("Digite o raio: ");
     scanf("%lf", &raio);

     // Cálculos
     area = pi * (raio * raio);
     perimetro = 2.0 * pi * raio;

     // Imprimir resultados
     printf("\n");
     printf("Raio: %lf \n", raio);
     printf("Área: %lf \n", area);
     printf("Perímetro: %lf \n", perimetro);

     return 0;
}
```

Consulte: TiposPontoFlutuante\Circulo01\Circulo01.vcproj

Descrição passo a passo

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

As primeiras duas linhas do programa são diretivas de compilador para incluir as bibliotecas padrão da linguagem C. Elas garantem que os comandos `printf` e `scanf` estejam disponíveis no instante da compilação do programa.

```
double pi = 3.141592;
```

O programa começa delcarando todas as variáveis que utilizará. Declara-se a variável de nome `pi`, do tipo `double`, que já armazena o valor 3.141592.

```
double raio, area, perimetro;
```

Esta linha é outra declaração em C, que define três variáveis, chamadas `raio`, `area` e `perimetro`. Todas elas são do tipo `double` e não possuem um valor inicial. Como as características dessas variáveis são todas iguais, pode-se declará-las todas na mesma linha, separando seus nomes com vírgula. **Como nenhuma dessas variáveis está recebendo um valor inicial, devemos tomar o cuidado de atribuir um valor a elas antes de utilizar seus valores para cálculos no programa**.

```
printf("Digite o raio: ");
scanf("%lf", &raio);
```

Para obter o valor da variável `raio`, utilizamos o comando `scanf` para ler o seu valor. Note que utilizamos o indicador `%lf` para informar que desejamos ler um número na representação decimal e converter para uma variável de tipo `double`. **Note a existência do `&` na frente da variável `raio` para o funcionamento correto do `scanf`.**

```
area = pi * (raio * raio);
```

Essa linha é uma atribuição. O lado direito do operador `=` é uma expressão matemática que calcula a área do círculo a partir do raio, usando a fórmula que aprendemos em geometria ($A = \pi R^2$). O símbolo `*` indica o operador de multiplicação. Como não existe um operador de potência, multiplicamos duas vezes pelo raio. O valor resultante será armazenado na variável `area`.

```
perimetro = 2.0 * pi * raio;
```

De forma semelhante, esta linha é uma atribuição que calcula o valor do perímetro e o armazena na variável `perimetro`.

```
printf("\n");
printf("Raio: %lf \n", raio);
printf("Área: %lf \n", area);
printf("Perímetro: %lf \n", perimetro);
```

Por fim, o programa não seria de utilidade se ele não informasse o resultado de forma visível para o usuário. O primeiro `printf` escreve uma linha em branco,

apenas para tornar o texto mais organizado. O segundo `printf` escreve novamente o raio digitado pelo usuário. Note que utilizamos o indicador `%lf` para informar que ele será substituído pela representação decimal do valor da variável `raio`, que é do tipo `double`. O mesmo vale para os demais `printf`s, que escrevem o valor calculado para a área e para o perímetro.

```
return 0;
```

Para terminar o programa, devemos sempre utilizar `return 0`.

Exemplo de execução

```
Digite o raio: 3.5

Raio: 3.500000
Área: 38.484502
Perímetro: 21.991144
```