3. Operadores, Expressões e Funções

Um programa tem como característica fundamental a capacidade de processar dados. Processar dados significa realizar operações com estes dados. As operações a serem realizadas com os dados podem ser determinadas por **operadores** ou **funções**. Os operadores podem ser de atribuição, aritméticos, de atribuição aritmética, incrementais, relacionais, lógicos e condicionais.

Exemplo: o símbolo `+` é um *operador* que representa a operação aritmética de adição. O identificador `sqrt()` é uma *função* que representa a operação de extrair a raiz quadrada de um número.

Uma **expressão** é um arranjo de operadores e operandos. A cada expressão válida é atribuído um valor numérico.

Exemplo: `4 + 6` é uma expressão cujo valor é `10`. A expressão `sqrt(9.0)` tem valor `3.0`.

3.1 Operador de Atribuição

A operação de atribuição é a operação mais simples do C. Consiste de atribuir valor de uma **expressão** a uma **variável**.

Sintaxe: A sintaxe da operação de atribuição é a seguinte:

identificador `=` *expressão*`;`

onde *identificador* é o nome de uma variável e *expressão* é uma expressão válida (ou outro identificador).

Exemplo: A seguir são mostradas algumas atribuições válidas:

```
a = 1;
delta = b * b - 4. * a * c;
i = j;
```

Observe que o símbolo de atribuição (`=`) **não tem** o mesmo significado que o usual da matemática que representa a igualdade de valores. Este símbolo, em C, representa a atribuição do valor

calculado em *expressão* a variável *identificador*. Em pseudo-linguagem o operador de atribuição é representado como ←. Também não se pode confundir o operador de atribuição (`=`) com o operador relacional de igualdade (`==`) que será visto na seção 3.5.1.

Observe-se também que o operando esquerdo **deve ser** um identificador de variável, isto é, **não pode** ser uma constante ou expressão.

Exemplo: A seguir são mostradas algumas atribuições inválidas:

```
1 = a;          // constante!
b + 1 = a;      // expressão!
```

3.1.1 Conversão de tipo.

Se os dois operandos de uma atribuição **não são** do mesmo tipo, o valor da expressão ou operador da direita **será convertido** para o tipo do identificador da esquerda.

Exemplo: Algumas atribuições com conversão de tipo:

```
int i;
float r;
i = 5;        // valor de i: 5
r = i ;       // valor de r: 5.0
```

A variável `i` foi inicializada com o valor `5`. Ao final da terceira instrução, `r` recebe o valor `5.0`.

Nestas conversões podem ocorrer alterações dos valores convertidos se o operando da esquerda for de um tipo que utilize **menor numero** de bytes que o operando da direita.

Exemplo: Algumas atribuições com conversão de tipo e perda de informação:

```
int i;
float r = 654.321;
i = r;               // truncamento!
```

Após a execução deste trecho de programa o valor da variável `i` será `654` pois seu valor foi truncado durante a conversão.

Pode-se dizer que as conversões potencialmente perigosas (onde há possibilidade de perda de informação) são:

```
char ← int ← float ← double
```

Observe que o compilador C ao encontrar esta operação **não gera** nenhum aviso de atenção para o programador. Assim este detalhe pode gerar um erro de programação (*bug*) que passe desapercebido ao programador inexperiente. É possível dizer que a linguagem C possui tipos "macios" (*soft types*) pois a operação com variáveis de tipos diferentes é perfeitamente possível. Esta característica do C se contrapõe a algumas linguagens em que isto não é possível (Fortran, por exemplo). Estas linguagens possuem tipos "duros" (*hard types*).

3.1.2 Limites do intervalo do tipo de dado.

Também é importante observar que os tipos em C tem intervalos bem definidos e os resultados das operações devem respeitar estes intervalos. Se a uma variável for atribuído um valor que esteja fora dos seus limites então este valor será alterado.

Exemplo: Observe as expressões abaixo, assuma que `i` seja uma variável do tipo `int`.

```
i = 4999;           // o valor de i e'   4999
i = 4999 + 1;       // o valor de i e'   5000
i = 5000 + 30000;   // o valor de i e' -30536
```

O valor de `35000` ultrapassou o limite superior do tipo `int` (`32767`).

É importante observar que em C, ao contrário de outras linguagens, a ultrapassagem do limite de um tipo **não é interpretado como erro.** Isto pode acarretar resultados inesperados para o programador desatento.

3.1.3 Atribuição múltipla.

E possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de **atribuição múltipla**.

Sintaxe: A sintaxe da atribuição múltipla é seguinte:

var_1 = [*var_2* = ...] *expressão*;

onde *var_1*, *var_2*, ... são os identificadores de variáveis e *expressão* é uma expressão válida.

Observe que na atribuição múltipla as operações ocorrem da **direita** para a **esquerda**, isto é, inicialmente o valor de *expressão* é atribuído a *var_2* e depois o valor de *var_2* é atribuído a *var_1*. Deve-se tomar cuidado com as conversões de tipo e limites de intervalo para atribuições de tipos diferentes.

Exemplo: Observe a instrução de atribuição múltipla abaixo: as variáveis inteiras `i, j` e `k` são todas inicializadas com o valor `1`. E as variáveis de dupla precisão `max` e `min` são inicializadas com o valor `0.0`:

```
int i, j, k;
double max, min;
i = j = k = 1;
max = min = 0.0;
```

Programa Exemplo: O arquivo `e0301.cpp` traz um programa para visualizar alguns aspectos relacionados com o operador de atribuição. Execute o programa passo-a-passo e observe o valor das variáveis.

3.2 Operadores Aritméticos

Existem cinco operadores aritméticos em C. Cada operador aritméticos está relacionado ao uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (`%`) chamado operador de **módulo** cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Operador	**Operação**
+	adição.
-	subtração.
*	multiplicação

`/`	divisão
`%`	módulo (resto da divisão inteira)

Sintaxe: A sintaxe de uma expressão aritmética é:

operando operador operando

onde *operador* é um dos **caracteres** mostrados acima e *operando* é uma **constante** ou um identificador de **variável**.

Exemplo: Algumas expressões aritméticas:

```
1+2   a-4.0   b*c   valor_1/taxa   num%2
```

Não existe em C, como existe em outras linguagens, um operador específico para a operação de potenciação (a^b). Existe, porém, uma função de biblioteca (**`pow()`**) que realiza esta operação. Veja a seção 3.7 adiante. Embora as operações do C sejam semelhantes as operações aritméticas usuais da matemática, alguns detalhes são específicos da linguagem, e devem ser observados.

3.2.1 Restrições de operandos.

Os operandos dos operadores aritméticos devem ser constantes numéricas ou identificadores de variáveis numéricas. Os operadores `+, -, *, /` podem operar números de todos os tipos (inteiros ou reais) porém o operador `%` somente aceita operandos **inteiros**.

Exemplo: Expressões válidas

Expressão	**Valor**
`6.4 + 2.1`	`8.5`
`7 - 2`	`5`
`2.0 * 2.0`	`4.0`
`6 / 3`	`2`
`47 % 2`	`1`

Uma restrição ao operador de divisão (`/`) é que o denominador **deve** ser diferente de zero. Se alguma operação de divisão por zero for realizada o correra um **erro de execução** do programa (*run-time error*), o programa será abortado e a mensagem `divide error` será exibida.

Exemplo: A expressão abaixo é inválida pois o primeiro operando não é um numero inteiro.

Expressão	**Valor**
`6.4 % 3`	`invalido!`

Podemos contornar o problema do operador inteiro da operação modulo usando o artifício da conversão de tipo (*casting*) mostrada na seção 2.3.4:

Exemplo: Observe o trecho de programa abaixo:

```
int num;
float valor = 13.0;
num = valor % 2;       // inválido!
num = (int)valor % 2;  // válido!
```

Observe que usamos a conversão de tipo para que o dado armazenado em `valor` fosse transformado no tipo `int` assim a operação módulo pode ser efetuada.

3.2.2 Conversão de tipo.

O resultado de uma operação aritmética **depende** dos tipos dos operandos. Se os operandos são do **mesmo** tipo o resultado será do **mesmo** tipo. Se os operando forem de tipos **diferentes** então haverá uma **conversão** de tipo tal que o tipo que ocupa **menos** espaço de memória será convertido para o tipo que ocupa **mais** espaço de memória e o resultado será deste tipo. Em geral:

`char` $\rightarrow$ `int` $\rightarrow$ `float` $\rightarrow$ `double`

Esta é uma regra geral, alguns compiladores podem ter outras regras de conversão.

Exemplo: Observe as conversões de tipo abaixo:

Expressão	**Valor**	**Conversão**
`6 + 2.0`	`8.0`	`int` $\rightarrow$ `float`
`7.000000 - 2.0000000000000`	`5.000000000000000`	`float` $\rightarrow$ `double`
`2 * 3.000000000000000`	`6.000000000000000`	`int` $\rightarrow$ `double`

Observe que estas conversões podem gerar resultados surpreendentes para o programador desatento.

Exemplo: Observe as expressões abaixo. Assuma que as variáveis `num_i, num_f, den_i` e `den_f` são inicializadas como:

```
int   num_i = 7  , den_i = 2  ;
float num_f = 7.0, den_f = 2.0;
```

Expressão	**Valor**	**Operandos**	**Resultado**
num_f / den_f	3.5	float / float	float
num_f / den_i	3.5	float / int	float
num_i / den_f	3.5	int / float	float
num_i / den_i	3	int / int	int

Observe que no exemplo acima o valor da última expressão é **3** e não **3.5**. Isto ocorre porque como os dois operandos são tipo `int` o resultado e convertido para o tipo `int` e ocorre o truncamento. O truncamento da divisão inteira é feito de modo a obter o menor valor absoluto.

Em C caracteres são armazenados na memória como números inteiros e por isso operações aritméticas são permitidas com tipos `char`. Os valores usados são os correspondentes da tabela ASCII.

Exemplo: Observe as expressões abaixo:

Expressão	**Valor**	**Conversão**
'A' + 1	'B'	65 + 1 → 66
'A' + 'B'	'â'	65 + 66 → 131
'A' + 32	'a'	65 + 32 → 97

3.2.4 Precedência de operadores.

Quando mais de um operador se encontram em uma expressão aritmética as operações são efetuadas uma de cada vez respeitando algumas regras de precedência: Estas regras de precedência são as mesmas da matemática elementar.

Os operadores de multiplicação (*), divisão (/) e módulo (%) tem precedência sobre os operadores de adição (+) e subtração (-). Entre operadores de mesma precedência as operações são efetuadas da **esquerda** para a **direita**. Veja a tabela 3.1.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
`1 + 2 - 3`	0	+ -
`24 - 3 * 5`	9	* -
`4 - 2 * 6 / 4 + 1`	2	* / - +
`6 / 2 + 11 % 3 * 4`	11	/ % * +

A ordem de precedência dos operadores pode ser quebrada usando-se parênteses: `( )`. Os parênteses são, na verdade, operadores de mais alta precedência e são executados primeiro. Parênteses internos são executados primeiro que parênteses externos.

Exemplo: Observe, nas expressões abaixo, o seu valor e a ordem das operações efetuadas:

Expressão	Valor	Ordem
`1 + (2 - 3)`	0	- +
`(24 - 3) * 5`	105	- *
`(4 - 2 * 6) / 4 + 1`	-1	* - / +
`6 / ((2 + 11) % 3) * 4`	24	+ % / *

Observe que os operadores e os operandos deste exemplo são os mesmos do exemplo anterior. Os valores, porém, são diferentes pois a ordem de execução das operações foi modificada pelo uso dos parênteses.

Programa Exemplo: O arquivo `e0302.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores aritméticos. Execute o programa passo-a-passo e observe o valor das variáveis.

3.3 Operadores de Atribuição Aritmética

Muitas vezes queremos alterar o valor de uma variável realizando alguma operação aritmética com ela. Como por exemplo: `i = i + 1` ou `val = val * 2`. Embora seja perfeitamente possível escrever estas instruções, foi desenvolvido na linguagem C uma instruções **otimizadas** com o uso de operadores ditos **operadores de atribuição aritmética**. Os símbolos usado são (`+=, -=, *=, /= , %=`). Deste modo as instruções acima podem ser rescritas como: `i += 1` e `val *= 2`, respectivamente.

Sintaxe: A sintaxe da atribuição aritmética é a seguinte:

var `+=` *exp*`;`

var `-=` *exp*`;`

```
var *= exp;
var /= exp;
var %= exp;
```

onde *var* é o identificador da variável e *exp* é uma expressão válida. Estas instruções são equivalentes as seguintes:

```
var = var + exp;
var = var - exp;
var = var * exp;
var = var / exp;
var = var % exp;
```

Exemplo: Observe as atribuições aritméticas abaixo e suas instruções equivalentes:

Atribuição aritmética	**Instrução equivalente**
`i += 1;`	`i = i + 1;`
`j -= val;`	`j = j - val;`
`num *= 1 + k;`	`num = num * (1 + k);`
`troco /= 10;`	`troco = troco / 10;`
`resto %= 2;`	`resto = resto % 2;`

O operador de atribuição aritmética tem precedência menor que os outros operadores até aqui discutidos. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0303.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores de atribuição aritmética. Execute o programa passo-a-passo e observe o valor das variáveis.

3.4 Operadores Incrementais

Em programação existem instruções muito comuns chamadas de **incremento** e **decremento**. Uma instrução de incremento **adiciona** uma unidade ao conteúdo de uma variável. Uma instrução de decremento **subtrai** uma unidade do conteúdo de uma variável.

Existem, em C, operadores específicos para realizar as operações de incremento (`++`) e decremento (`--`). Eles são genericamente chamados de **operadores incrementais**.

Sintaxe: A sintaxe dos operadores incrementais é a seguinte:

```
                instrução equivalente
++ var          var = var + 1
var ++          var = var + 1
-- var          var = var - 1
var --          var = var - 1
```

onde *var* é o nome da variável da qual se quer incrementar ou decrementar um unidade.

Observe que existe duas sintaxes possíveis para os operadores: pode-se colocar o operador **à esquerda** ou **á direita** da variável. Nos dois casos o valor da variável será incrementado (ou decrementado) de uma unidade. Porém se o operador for colocado **á esquerda** da variável, o valor da variável será incrementado (ou decrementado) **antes** que a variável seja usada em alguma outra operação. Caso o operador seja colocado **à direita** da variável, o valor da variável será incrementado (ou decrementado) **depois** que a variável for usada em alguma outra operação.

Exemplo: Observe o fragmento de código abaixo e note o valor que as variáveis recebem **após** a execução da instrução:

```
                                valor das variáveis
int a, b, c, i = 3;     // a: ?   b: ?   c: ?   i: 3
a = i++;                // a: 3   b: ?   c: ?   i: 4
b = ++i;                // a: 3   b: 5   c: ?   i: 5
c = --i;                // a: 3   b: 5   c: 4   i: 4
```

Os operadores incrementais são bastante usados para o controle de laços de repetição, que serão vistos na seção ??. É importante que se conheça exatamente o efeito sutil da colocação do operador, pois isto pode enganar o programador inexperiente.

Os operadores incrementais tem a mais alta precedência entre todos, sendo superados apenas pelos parênteses que tem precedência ainda maior. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0304.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores incrementais. Execute o programa passo-a-passo e observe o valor das variáveis.

3.5 Operadores Relacionais e Lógicos

A chave para a flexibilidade de um algoritmo é a tomada de decisões através da avaliação de condições de controle. Uma condições de controle é uma **expressão lógica** que é avaliadas como **verdadeira** ou **falsa**. Uma expressão lógica é construída com **operadores relacionais** e **lógicos**.

3.5.1 Operadores relacionais

Operadores relacionais verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais em C:

Operador	Significado
>	maior que
<	menor que
>=	maior ou igual a (não menor que)
<=	menor ou igual a (não maior que)
==	igual a
!=	não igual a (diferente de)

Sintaxe: A sintaxe das expressões lógicas é:

expressão_1 operador expressão_2

onde *expressão_1* e *expressão_2* são duas expressões numéricas quaisquer, e *operador* é um dos operadores relacionais.

Ao contrário de outras linguagens, em C **não existem** tipos lógicos, portanto o **resultado** de uma expressão lógica é um **valor numérico**: uma expressão avaliada **verdadeira** recebe o valor `1`, uma expressão lógica avaliada **falsa** recebe o valor `0`.

Se os operandos forem de **tipos diferentes** haverá uma conversão de tipo **antes** da avaliação da expressão. Esta conversão de tipo é feita de acordo com a regra mostrada na seção 3.2.2.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `i` e `j` são variáveis `int` cujos valores são `5` e `-3`, respectivamente. As variáveis `r` e `s` são `float` com valores `7.3` e `1.7`, respectivamente.

Expressão	Valor
i == 7	0
r != s	1
i > r	0
6 >= i	1
i < j	0
s <= 5.9	1

Os operadores relacionais de igualdade (== e !=) tem precedência **menor** que os de magnitude (>, <, >= e <=). Estes, por sua vez, tem precedência **menor** que os operadores aritméticos. Operadores relacionais de mesma precedência são avaliados da esquerda para a direita. Veja a tabela 3.1.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `m` e `n` são variáveis tipo `int` com valores `4` e `1`, respectivamente.

Expressão	**Valor**	**Ordem de Operação**
`m + n == 5`	1	`+ ==`
`m != 2 * n > m`	1	`* > !=`
`6 >= n < 3 - m`	0	`- >= <`
`m == n <= m > m`	0	`<= > !=`

3.5.2 Operadores lógicos

São três os operadores lógicos de C: `&&`, `||` e `!`. Estes operadores tem a mesma significação dos operadores lógicos Booleanos AND, OR e NOT.

Sintaxe: A sintaxe de uso dos operadores lógicos:

expr_1 && *expr_2*
expr_1 || *expr_2*
!*expr*

onde *expr_1*, *expr_2* e *expr* são expressões quaisquer.

Observe que os operadores lógicos atuam sobre expressões de quaisquer valores. Para estes operadores todo valor numérico diferente de `0` é considerado `1`.

Exemplo: A seguir é mostrado o valor lógico de uma expressão qualquer:

Expressão	***Valor lógico***
0	0
1	1
1.0	1
0.4	1
-5.2	1

onde *expr_1*, *expr_2* e *expr* são expressões quaisquer.

O resultado da operação lógica `&&` será `1` somente se os dois operandos forem `1`, caso contrário o resultado é `0`. O resultado da operação lógica `||` será `0` somente se os dois operandos forem `0`, caso contrário o resultado é `1`. O resultado da operação lógica `!` será `0` se o operandos for `1`, e `1` se o operando for `0`. Abaixo mostra-se o resultado das possíveis combinações entre os operandos para cada operador lógico:

Operador `&&`:	**op_1**	**op_2**	**Res**
op_1 && op_2	1	1	1
	1	0	0
	0	1	0
	0	0	0

Operador `||`:	**op_1**	**op_2**	**Res**
op_1 || op_2	1	1	1
	1	0	1
	0	1	1
	0	0	0

Operador `!`:	**op**	**Res**
!op	1	0
	0	1

O Operador && tem precedência sobre o operador `||`. Estes dois têm precedência menor que os operadores relacionais. O operador ! tem a mesma precedência que os operadores incrementais. Veja a tabela 3.1.

Exemplo: Observe as expressões lógicas abaixo e verifique o resultado de sua avaliação. Admita que `a`, `b` e `c` são variáveis tipo `int` com valores `0`, `1` e `2`, respectivamente.

Expressão	**Valor**	**Ordem de Operação**
a && b	0	
c > b || a < c	1	> < ||
a + b && !c - b	1	! + - &&
!b && c || a	0	! && ||

Programa Exemplo: O arquivo `e0305.cpp` traz um programa para visualizar alguns aspectos relacionados com os operadores lógicos e relacionais. Execute o programa passo-a-passo e observe o valor das variáveis.

3.6 Operador Condicional

O operador condicional (`?:`) é usado em expressões condicionais. Uma expressão condicional pode ter dois valores diferentes dependendo de uma condição de controle.

Sintaxe: A sintaxe de uma expressão condicional é:

condição `?` *expressão_1* `:` *expressão_2*

onde *expressão_1* e *expressão_2* são duas expressões quaisquer, e *condição* é uma expressão lógica que será avaliada primeiro. Se o valor de *condição* for `1`, isto é, verdadeiro, então a expressão condicional assumirá o valor de *expressão_1*. Caso contrario assumirá o valor de *expressão_2*. Uma expressão condicional é equivalente a uma estrutura de decisão simples:

```
se condição
   então expressao_1
   senão expressao_2
fim se
```

Exemplo: Observe as expressões condicionais abaixo e verifique o resultado de sua avaliação. Admita que `i`, `j` e `k` são variáveis tipo `int` com valores `1`, `2` e `3`, respectivamente.

Expressão	Valor
`i ? j : k`	2
`j > i ? ++k : --k`	4
`k == i && k != j ? i + j : i - j`	-1
`i > k ? i : k`	3

O operador condicional tem baixa precedência, precedendo, apenas, aos operadores de atribuição. Veja a tabela 3.1.

Programa Exemplo: O arquivo `e0306.cpp` traz um programa para visualizar alguns aspectos relacionados com o operador condicional. Execute o programa passo-a-passo e observe o valor das variáveis.

3.7 Funções de biblioteca

Uma função é um **sub-programa** (também chamado de rotina). Esta função *recebe* informações, *as processa* e *retorna* outra informação. Por exemplo, podemos ter uma função que receba um valor numérico, calcule seu logaritmo decimal e retorne o valor obtido. Existem dois tipos de funções: *funções de biblioteca* e *funções de usuário*. Funções de biblioteca são funções escritas pelos fabricantes do compilador e já estão pré-compiladas, isto é, já estão escritas em código de máquina. Funções de usuário são funções escritas pelo programador. Nesta seção trataremos somente das funções de biblioteca, funções de usuário serão vistas no capítulo ?.

3.7.1 O uso de funções

Antes de usar uma função é preciso saber como a função esta declarada, isto é, quais são os parâmetros que a função recebe e quais são os para metros que a função retorna. Estas informações estão contidas no manual do usuário do compilador ou em sua documentação *on-line*.

Sintaxe: A sintaxe de declaração de uma função é:

```
tipo_ret nome(tipo_1, tipo_2, ...)
```

onde `nome` é o nome da função, `tipo_1`, `tipo_2`, ... são os tipos (e quantidade) de parâmetros de entrada da função e `tipo_ret` é o tipo de dado de retorno da função. Além dos tipos usuais vistos na seção 2.3, existe ainda o tipo `void` (vazio, em inglês) que significa que aquele parâmetro é inexistente.

Exemplo: A função `cos()` da biblioteca `math.h` é declarada como:

```
double cos(double);
```

Isto significa que a função tem um parâmetro de entrada e um parâmetro de saída, ambos são do tipo `double`.

Exemplo: A função `getch()` da biblioteca `conio.h` é declarada como:

```
int getch(void);
```

Isto significa que a função não tem parâmetros de entrada e tem um parâmetro `int` de saída.

Para podermos usar um função de biblioteca devemos **incluir** a biblioteca na compilação do programa. Esta inclusão é feita com o uso da diretiva `#include` colocada antes do programa principal, como visto na secao 2.4.2.

Exemplo: Assim podemos usar a função no seguinte trecho de programa:

```
#include <math.h>            // inclusão de biblioteca
void main(){                 // inicio do programa principal
  double h = 5.0;            // hipotenusa
  double co;                 // cateto oposto
  double alfa = M_PI_4;      // angulo: π/4
  co = h * cos(alfa);        // calculo: uso da funcao cos()
}                            // fim do programa
```

As funções tem alta precedência, sendo mais baixa apenas que os parênteses. A tabela 3.1 mostra as precedências de todos os operadores estudados neste capítulo.

3.7.2 As bibliotecas disponíveis e algumas funções interessantes

A seguir segue uma lista de todas as bibliotecas disponíveis no compilador *Turbo C++ 3.0 Borland*: Ao longo do texto veremos o uso de muitas funções cobrindo uma boa parte destas bibliotecas, porém o leitor que desejar tornar-se "fluente" na linguagem C pode (e deve) estudá-las com profundidade.

```
alloc.h      assert.h     bcd.h        bios.h       complex.h
conio.h      ctype.h      dir.h        dirent.h     dos.h
errno.h      fcntl.h      float.h      fstream.h    generic.h
graphics.h   io.h         iomanip.h    iostream.h   limits.h
locale.h     malloc.h     math.h       mem.h        process.h
setjmp.h     share.h      signal.h     stdarg.h     stddef.h
stdio.h      stdiostr.h   stdlib.h     stream.h     string.h
strstrea.h   sys\stat.h   sys\timeb.h  sys\types.h  time.h
values.h
```

Vejamos algumas funcoes disponiveis nas bibliotecas C.

Biblioteca math.h

```
int abs(int i);
double fabs(double d);
```

Calcula o valor absoluto do inteiro `i` e do real `d`, respectivamente.

```
double sin(double arco);
double cos(double arco);
double tan(double arco);
double asin(double arco);
double acos(double arco);
double atan(double arco);
```

Funções trigonometricas do ângulo `arco`, em radianos.

```
double ceil(double num);
double floor(double num);
```

Funcoes de arredondamento para inteiro.

`ceil()` arredonda para cima. Ex. `ceil(3.2) == 3.0;`

`floor()` arredonda para baixo. Ex. `floor(3.2) == 4.0;`

```
double log(double num);
double log10(double num);
```

Funcoes logaritmicas: `log()` é logaritmo natural (base *e*), `log10()` é logaritmo decimal (base 10).

```
double pow(double base, double exp);
```

Potenciacao: `pow(3.2,5.6)` = $3.2^{5.6}$.

```
double sqrt(double num);
```

Raiz quadrada: `sqrt(9.0) = 3.0.`

Biblioteca stdlib.h

```
int random(int num);
```

Gera um número inteiro aleatório entre `0` e `num - 1`.

Programa Exemplo: O arquivo `e0307.cpp` traz um programa para visualizar alguns aspectos relacionados com funções de biblioteca. Execute o programa passo-a-passo e observe o valor das variáveis.

3.8 Precedência entre os operadores do C

A tabela 3.1 mostra a ordem de precedência de todos os operadores estudados neste capítulo. Os operadores de maior precedência são os **parênteses** e as chamadas de **funções**. Os operadores de menor precedência são os o operadores de **atribuição**.

Categoria	Operadores	Prioridade
parênteses	`( )`	interno → externo
função	`nome()`	E → D
incremental, lógico	`++ -- !`	E ← D
aritmético	`* / %`	E → D
aritmético	`+ -`	E → D
relacional	`< > <= >=`	E → D
relacional	`== !=`	E → D
lógico	`&&`	E → D
lógico	`||`	E → D
condicional	`?:`	E ← D
atribuição	`= += -= *= /= %=`	E ← D

***Tabela 3.1**: Precedência dos operadores. Maior precedência no topo, menor precedência na base.*