Introdução à linguagem C

Rodolfo Riyoei Goya

1. Introdução à programação em linguagem C

1.1. Características da linguagem C

A linguagem C foi criada em 1972 por Dennis Ritchie nos laboratórios da Bell. Desde então e em muito pouco tempo, tornou-se uma das linguagens mais populares do mundo entre programadores profissionais.

A razão dessa popularidade é que a linguagem C foi concebida por programadores para ser uma ferramenta útil para programadores e proporcionar as conveniências de uma linguagem de alto nível, mas, ao mesmo tempo, um controle muito próximo ao hardware e periféricos.

A linguagem C foi concebida para ser a ferramenta de desenvolvimento do sistema operacional UNIX, este mesmo concebido para ser um sistema facilmente transportável para qualquer plataforma de hardware. Deste modo, em todas as plataformas onde o UNIX possa ser encontrado se encontra uma versão do compilador C.

Esta característica de portabilidade é peculiar da linguagem C. A linguagem é especificada de tal modo que requer um mínimo de ajustes para ser executada em qualquer máquina, seja um microcontrolador, microcomputador pessoal, mainframe ou supercomputador.

As implementações atualmente existentes de compiladores de linguagem C produzem código de tal eficiência em espaço e velocidade que chegam a rivalizar programas codificados diretamente em linguagem assembly (linguagem de máquina nativa do computador).

Todas estas razões tornam a linguagem C tão popular que a maioria dos programas comerciais atualmente desenvolvidos (incluindo sistemas operacionais, compiladores e interpretadores para outras linguagens) é implementada em linguagem C. Assim, sua popularidade tornou-a a linguagem comum para divulgação e comunicações em revistas especializadas.

1.2. Arquivos usados no desenvolvimento de um programa em C

A linguagem C é uma linguagem compilada. Isto quer dizer que para executar um programa escrito em linguagem C (programa-fonte), deve-se primeiro traduzi-lo (essa tarefa é executada pelo compilador) para a linguagem nativa do computador para que possa ser posteriormente executado. Uma vez obtido o programa compilado (programa executável), este pode voltar a ser executado quantas vezes se desejar sem a necessidade de uma nova compilação. O programa executável pode ser distribuído para execução sem a necessidade de se distribuir o programa-fonte.

No processo de desenvolvimento de um programa escrito em linguagem C, normalmente são utilizados 3 arquivos:

Em UNIX:

nome.C → nome.o → a.out

(Executa-se o programa teclando o nome do Arquivo "a.out").

Em MS-DOS e WINDOWS:

nome.C → nome.OBJ → nome.EXE

(Executa-se o programa teclando "nome").

O primeiro dos arquivos é denominado de programa-fonte e é produzido pelo programador com o uso de um editor de texto. O arquivo intermediário é denominado de programa-objeto e é produzido pelo compilador. O arquivo final é denominado de programa executável e é produzido pelo linker. O programa-objeto é um módulo que o linker reúne a outros módulos (produzidos pela compilação de outros programas fontes produzidos pelo programador ou contendo rotinas de biblioteca que desempenham funções oferecidas com a linguagem) para produzir o programa final executável.

1.3. Estrutura básica de um programa em linguagem C

Um programa escrito em linguagem C tem uma estrutura geral da forma abaixo:

```
#include <stdio.h>                  Cabeçalho (header)

int main() /* Um programa bem simples. */
{
  int num;                    Corpo do programa (body)

  num = 1;
  printf( "%d eh um belo numero.\n", num );
  return 0;
}
```

Este é um programa que, ao ser executado, imprime a frase: *1 eh um belo número*. É um programa simples, que ilustra as características de um programa escrito em C.

Todo o programa escrito em linguagem C constitui-se de uma coleção de uma ou mais funções, onde uma delas deve se chamar `main()`. A função `main()` é particularmente importante pois é a função chamada primeiro quando o programa é executado (não importando em que parte do programa ela esteja).

Todo programa consiste de um cabeçalho (header) e de um corpo (body). O cabeçalho contém todas as diretivas de pré-processamento (tal como o `#include` do exemplo acima). O corpo do programa contém as funções, que podem ser reconhecidas como um nome seguido por um par de parênteses, podendo haver ou não parâmetros dentro dos parênteses. (No caso do exemplo o nome da função é `main()`).

No caso do exemplo, o arquivo de funções de biblioteca chamado `stdio.h` passa a fazer parte do programa pelo uso da diretiva `#include`. Este arquivo é fornecido como parte do compilador C. Às vezes ele é necessário, às vezes não. No caso do exemplo ele é necessário por conter dados descrevendo a função `printf` utilizada no programa (`stdio` é abreviatura de **standard input/output** - entrada/saída padrão). Outros arquivos, contendo descrições de outros tipos de funções também fazem parte dos pacotes que compõem um compilador C.

Já o corpo da função é delimitado por chaves {} e constitui-se de uma série de comandos (“statements”), cada comando terminado (separado um do outro) com um ponto e vírgula (`;`).

No exemplo, o corpo do programa contém quatro comandos: o primeiro é um comando de declaração de variável, definindo os nome e tipos de variáveis que estão sendo usadas. Variáveis são locais onde dados são guardados em memória. O processamento de dados em um computador constitui-se normalmente na manipulação no valor de variáveis. Todas as variáveis utilizadas em um programa devem ser declaradas para especificar seu nome e seu tipo.

O segundo é um comando de atribuição onde uma variável tem seu valor modificado e um terceiro comando que consiste na chamada da função `printf()`. Os parâmetros da função `printf()` tais como o %d e o \n serão explicados mais adiante.

O quarto é o comando `return 0` que finaliza a função.

No exemplo pode-se notar também que existe um trecho com o formato abaixo:

```
/* Um programa bem simples. */
```

Este trecho é um comentário. Os comentários são iniciados com `/*` e terminados com `*/` e servem apenas para facilitar a leitura do programa, pois são ignorados pelo compilador. Comentários são observações inseridas pelo programador para tornar um programa mais claro.

O compilador C ignora a quantidade de linhas e espaços em branco colocados entre os comandos. Deste modo, o programa seguinte tem o mesmo efeito final que o do exemplo:

```
#include <stdio.h> int main()


/* Um programa bem simples. */

{ int num; num

=

1;


printf(

"%d é um belo número.\n",

num); }
```

Apesar de serem equivalentes, nota-se que o exemplo inicial apresenta-se de uma forma bem mais clara. De fato, o modo como o programa é escrito pode torná-lo bem mais legível. Assim, um bom hábito de programação segue algumas regras bastante simples como as relacionadas abaixo:

```
#include <stdio.h>

int main( int argc, char *argv[] ) /* Imprime pernas e dedos. */
                                   Use Comentários.
{
  int pernas, dedos;               Escolha nomes convenientes.
                                   Use linhas em branco.
  pernas = 4;                      Coloque um comando por linha.
  dedos = 20;
  printf( "Duas pessoas têm %d pernas e %d dedos.\n", pernas, dedos );
}
```

Outro aspecto importante é que a linguagem C diferencia letras maiúsculas e minúsculas na formação de nomes. Assim, uma variável chamada `pernas` não será a mesma que outra variável chamada de `PERNAS` ou de `Pernas` (esta característica é denominada de “case-sensitive”, diferentemente do que ocorre em muitas outras linguagens de programação).

Na escolha de nomes para funções e variáveis o programador pode-se utilizar de letras minúsculas e maiúsculas (sem acentuação ou cedilha), dígitos numéricos e o símbolo de underscore ("_").

Os nomes podem ser formados por qualquer combinação destes caracteres somente com a restrição de não poderem começar com dígito numérico. Os nomes podem ter qualquer comprimento embora alguns compiladores só utilizem os oito primeiros (assim uma variável ou função chamada nome_longo1 pode ser tida como a mesma que uma outra chamada nome_longo2 pois ambas têm os oito primeiros caracteres idênticos, em alguns compiladores).

Nomes válidos:

contagem
vai_um _ehbom
temp1

Nomes inválidos:

%descont
vai-um
e'bom
1temp

Os três primeiros são inválidos por não usarem caracteres válidos (%, - e ') e o último por iniciar com caracter numérico.

2. Blocos construtivos da linguagem C

Funções são construídas encadeando-se comandos. Por sua vez, os comandos operam sobre dados. Fornece-se números, letras e palavras ao computador e espera-se obter algo com estes dados. A linguagem C oferece ao programador diversos tipos de dados para que este possa construir seus programas.

Por outro lado, para que um programa se utilize de dados de uma maneira útil é necessário que ele seja capaz de interagir com o meio exterior (frequentemente o usuário do programa) recebendo dados para serem processados e devolvendo-lhe os valores processados.

Porém não basta o computador devolver ao usuário os dados por ele enviados. A linguagem deve dispor de recursos que permitam ao programador efetuar manipulações e transformações nos dados recebidos.

Esta seção mostra como a linguagem C lida com estes conceitos.

2.1. Tipos de Dados

Os tipos básicos de dados oferecidos pela linguagem C estão divididos em três classes:

Números Inteiros:	`short`, `int`, `long`, `long long`, e `unsigned`
Caracteres:	`char`
Números em Ponto Flutuante:	`float`, `double` e `long double`

Os cinco primeiros são usados para representar números inteiros, ou seja, números sem parte fracionária. Números tais como 5, -33 e 2719 são inteiros enquanto números como 3.14, 1/2 e $6{,}023 \times 10^{23}$ não o são.

O segundo tipo é denominado de caracter e consiste em letras do alfabeto (maiúsculas e minúsculas) e outros caracteres tais como ‘#’, ‘$’, ‘%’ e ‘&’ além dos próprios dígitos numéricos.

Os outros dois tipos são utilizados para representar números que podem ter parte decimal. Estes dois últimos são também denominados tipos em ponto flutuante.

2.1.1. Dado do tipo inteiro

Os diversos tipos possíveis de inteiro relacionados se devem à possibilidade de escolha que a linguagem oferece para as faixas de valores que um dado pode assumir. Ao contrário do que ocorre na vida real, dentro de um computador os números têm valores mínimos e máximos e podem ser representados com precisão limitada. Assim os tipos `short`, `int`, `unsigned`, `long` e `long long` são tipos inteiros que permitem diferentes faixas de valores.

As faixas para cada tipo variam para cada computador onde a linguagem é implementada. No caso dos processadores de 16 bits, por exemplo, variáveis do tipo `short` ocupam um byte de memória (8 bits) e seus valores vão de -128 a +127 (entre -2^7 e 2^7-1), variáveis do tipo `int` ocupam dois bytes de memória (16 bits) e seus valores vão de -32768 a +32767 (entre -2^{15} e $2^{15}-1$), variáveis do tipo `unsigned` ocupam dois bytes de memória (16 bits) e seus valores vão de 0 a +65535 (entre 0 e $2^{16}-1$) e variáveis do tipo `long` ocupam quatro bytes de memória (32 bits) e seus valores vão de -2147483648 a +2147483647 (entre -2^{31} e $2^{31}-1$).

No caso do compilador MinGW usado neste curso (produz programas para serem executados em Microsoft Windows):

- Variáveis do tipo `short` são armazenadas em 16 bits de memória e seus valores vão de -32768 a +32767 (entre -2^{15} e $2^{15}-1$).
- Variáveis do tipo `int` e `long` são armazenadas em 32 bits de memória e seus valores vão de -2147483648 a +2147483647 (entre -2^{31} e $2^{31}-1$).
- Variáveis do tipo `unsigned` e `unsigned int` são armazenadas em 32 bits de memória e seus valores vão de 0 a +4294967295 (entre 0 e $2^{32}-1$).
- Variáveis do tipo `long long` são armazenadas em 64 bits de memória e seus valores vão de - 9223372036854775808 a + 9223372036854775807 (entre -2^{63} e $2^{63}-1$).

Variáveis de tipo inteiras em C são declaradas simplesmente especificando-se o tipo seguindo-se de uma lista com os nomes das variáveis.

```
int pontos;
short dia;
long populacao;
long long divida;
short hora, minuto, segundo;
```

Uma lista de variáveis é feita com uma relação dos nomes das variáveis separados por vírgulas e terminada com ponto e vírgula.

A faixa finita na qual os números são representados se deve o fato de que cada tipo de dado reserva uma porção finita de memória para armazenamento de cada tipo. Por exemplo, nos processadores de 16 bits, dados de tipo `short` são guardados em dois bytes de memória, dados de tipo `int` são guardados em quatro bytes de memória, do tipo `long` em quatro bytes e do tipo `long long` em oito bytes (nos processadores pentium, dados de tipo `short` são guardados em dois bytes de memória e dados de tipo `int` e `long` em quatro bytes).

A palavra `unsigned` é utilizada para modificar a faixa de definição de um dado para valores apenas positivos. Assim, no caso dos processadores de 16 bits, um dado de tipo `unsigned short` tem valores entre 0 e 255, `unsigned int` tem valores entre 0 e 65535 e um dado do tipo `unsigned long` pode assumir valores entre 0 e 4294967295 (nos processadores pentium, um dado de tipo `unsigned short` tem valores entre 0 e 65535, e dados de tipo `unsigned int` e `unsigned long` podem assumir valores entre 0 e 4294967295).

```
unsigned int ano;
unsigned short dia, mes;
unsigned long porcos, bois, gansos;
```

Outro tipo de dado importante são as constantes inteiras. A linguagem C reconhece um número sem um ponto decimal como sendo um inteiro. Assim, números como 2 ou -150 são constantes inteiras. As constantes inteiras podem ser usadas para iniciar o valor de variáveis:

```
dia = 1;
mes = 5;
pontos = -20;
```

A inicialização de variáveis também pode ser feita no comando de declaração. Como no exemplo abaixo:

```
int creditos, pontos = 0;
short hora = 12, minutos = segundos = 0;
```

No primeiro exemplo apenas a variável `pontos` é inicializada. Enquanto não receber nenhum valor durante a execução do programa o valor da variável `creditos` deve ser tomado como indeterminado. No segundo exemplo, a variável `hora` é iniciada com o valor 12 enquanto as variáveis `minutos` e `segundos` são iniciadas com 0.

Constantes de tipo `long` são definidas com valores numéricos finalizados com a letra `'l'`:

```
long populacao = 1000000000l; /* apesar de parecer com '1' (um)
                                 é uma letra 'l' (ele) no final. */
```

Variáveis de tipo `long` ocupam mais espaço e tomam mais tempo de processamento que variáveis de tipo `int`.

Constantes de tipo `long long` são definidas com valores numéricos finalizados com a letra `'l'`:

```
long long populacao = 10000000000ll; /* parece com '11' (um um)
                                        mas é 'll' (ele ele) no final. */
```

Variáveis de tipo `long long` ocupam mais espaço e tomam mais tempo de processamento que variáveis de tipo `long`.

2.1.2. Dado do tipo caracter

Normalmente um dado do tipo `char` é definido como um número inteiro na faixa entre -128 e 127 armazenado em 8 bits (1 byte), enquanto um dado de tipo `unsigned char` é definido como um número inteiro na faixa entre 0 e 255 armazenado em 8 bits (1 byte). Os computadores normalmente usam um código para traduzir estes números em caracteres (os mais conhecidos são o ASCII, EBCDIC e o UNICODE). A linguagem C usa o código ASCII (American Standard Coding for Information Interchange) para traduzir caracteres em códigos. No código ASCII, por exemplo, a letra 'A' é codificada em binário como 01000001 (65 em decimal), a letra 'a' como 01100001 (97 em decimal) e o símbolo '?' como 00111111 (63 em decimal).

Caracteres numéricos não têm necessariamente um código correspondente a seu valor. Assim, por exemplo, o código ASCII que corresponde ao caracter '0' é 00110000, 48 em decimal.

As variáveis de tipo caracter são declaradas utilizando-se a palavra chave `char`:

```
char letra, resposta;
```

Um caracter constante é definido como um símbolo entre apóstrofos. Deste modo podemos usar tais constantes para atribuir um valor a uma variável de tipo caracter.

```
letra = 97;     /* coloca ‘a’ na variável letra.    */
resposta = 'S'; /* coloca ‘S’ na variável resposta. */
```

Se observarmos uma tabela ASCII veremos que certos caracteres são denominados de não-imprimíveis. Tais caracteres são usados normalmente para o controle de impressão de saída. Para a representação destes caracteres, a linguagem C utiliza certas sequências especiais de caracteres. Tais sequências são denominadas de sequências de escape com alguns exemplos relacionados abaixo:

\n	newline
\t	tabulação
\b	backspace
\r	retorno de carro - carriage return
\f	form feed
\nnn	Número codificado em octal
\xnnnn	Número codificado em hexadecimal
\\	barra invertida (\) - backslash
\'	Apóstrofo (')
\"	Aspas (")

Tais sequências também devem ser colocadas entre aspas para definir uma constante. Por exemplo:

```
char caracter = '\n';
```

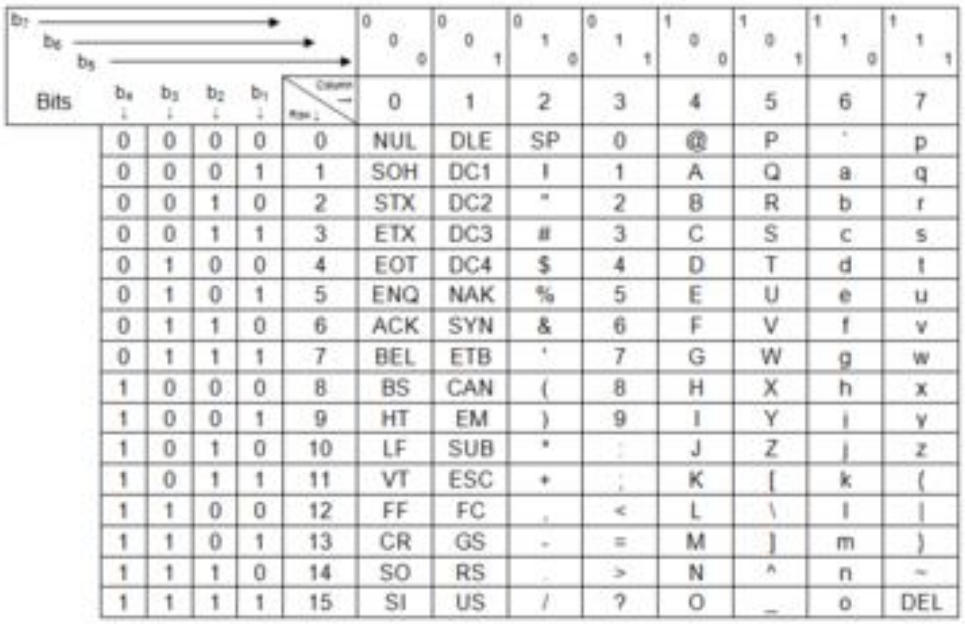
Ao se imprimir a variável `caracter`, a cabeça de impressão da impressora ou o cursor na tela avançam para o começo da próxima linha.

Na relação dada, o newline avança para o início da próxima linha, um tab move o cursor ou a cabeça de impressão de uma quantidade fixa de posições (normalmente múltipla de 5 ou 8). Um backspace move para trás de um caracter. Um retorno de carro move para o início da mesma linha e um form feed faz com que a impressora avance o papel de uma página. Os três últimos permitem a definição de constantes de caracter para os caracteres \' e ".

Sequência	Símbolo	Codificação binária
\n	Newline	00001010
\t	Tabulação	00001001
\b	Backspace	00001000
\r	retorno de carro - carriage return	00001101
\f	form feed	00001100
\\	barra invertida (\) - backslash	01011100
\'	Apóstrofo (')	00100111
\"	Aspas (")	00100010

Deste modo, as declarações:
```
char caracter = '\n';
```
e
```
char caracter = 10;
```
têm o mesmo significado.

A codificação em binário completa de todos os caracteres segundo a tabela ASCII é apresentada na figura abaixo:



Bits	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4 b3 b2 b1	0	1	2	3	4	5	6	7
0 0 0 0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	STX	DC2	"	2	B	R	b	r
0 0 1 1	ETX	DC3	#	3	C	S	c	s
0 1 0 0	EOT	DC4	$	4	D	T	d	t
0 1 0 1	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	ACK	SYN	&	6	F	V	f	v
0 1 1 1	BEL	ETB	'	7	G	W	g	w
1 0 0 0	BS	CAN	(	8	H	X	h	x
1 0 0 1	HT	EM	)	9	I	Y	i	y
1 0 1 0	LF	SUB	*	:	J	Z	j	z
1 0 1 1	VT	ESC	+	;	K	[	k	{
1 1 0 0	FF	FC	,	<	L	\	l	|
1 1 0 1	CR	GS	-	=	M	]	m	}
1 1 1 0	SO	RS	.	>	N	^	n	~
1 1 1 1	SI	US	/	?	O	_	o	DEL

Uma tabela de codificação ASCII para binário completa pode ser achada em:
http://pt.wikipedia.org/wiki/ASCII

2.1.3. Dados do tipo ponto flutuante

Os tipos de dados inteiros e caracteres servem muito bem para a maioria dos programas. Contudo, programas orientados para aplicações matemáticas frequentemente fazem uso de números que apresentam parte decimal ou são grandes demais para serem representados em tipo inteiro (acima de 10^{10} em `int` e 10^{20} em `long long`). Para este tipo de dados, em C, existem os tipos denominados de ponto flutuante: `float`, `double` e `long double`.

A representação de números em ponto flutuante é análoga a notação científica e permite a representação de uma faixa bastante grande de números, incluindo os números fracionários. Por exemplo:

Número	Notação Científica	Notação em Linguagem C
1000000000	$1,0x10^{9}$	1.0e9
123000	$1,23x10^{5}$	1.23e5
-786,89	$-7,8689x10^{2}$	-7.8689e2
322,56	$3,2256x10^{2}$	3.2256e2
0,000056	$5,6x10^{-5}$	5.6e-5

Na primeira coluna é exibida a notação usual. Na segunda coluna a notação científica e na terceira o modo como a notação científica é representada em um computador. Em linguagem C, de modo diferente que o usual em português, usa-se '.' ao invés de ',' para separar a parte inteira da decimal.

Do mesmo modo que ocorre com os inteiros, números em ponto flutuante são armazenados em porções finitas de memória de modo que a sua precisão é limitada. O modo como são codificados, seguem os padrões IEEE 754-1985 e IEEE 754-2008.

O tipo `float` é armazenado em 32 bits de memória e tem precisão de 7 dígitos com o expoente indo de 10^{-38} a 10^{+38}.
O tipo `double` é armazenado em 64 bits de memória e tem precisão de 14 dígitos com expoente indo de 10^{-308} a 10^{+308}.
O tipo `long double` é armazenado em 80 bits de memória e tem precisão de 18 dígitos com expoente indo de 10^{-4932} a 10^{+4932}.

Isto quer dizer que um número como 4345345.8476583746123 será armazenado como:
4.345345e+6 em uma variável do tipo `float`.
4.3453458476584e+6 em uma variável do tipo `double`.
4.34534584765837461e+6 em uma variável do tipo `long double`.

Constantes em ponto flutuante podem ser definidas de diversas formas. A mais geral é uma série de dígitos com sinal, incluindo um ponto decimal, depois um e ou E seguido de um expoente de uma potência de dez com sinal. Por exemplo:

-1.609E-19
+6.03e+23.

Na definição de constantes, podem-se omitir sinais positivos, a parte de expoente e a parte inteira ou fracionária. Por exemplo:

3.14159
.2
4e16
.8e-5
100.

Não se podem usar espaços dentro de um número em ponto flutuante:
O número 3.34 E+12 está errado (tem um espaço entre o 4 e o E).

Constantes de tipo `long double` são criadas acrescentando-se 'L' ou 'l' no final: 4.34534584765837461e+6L ou 4.34534584765837461e+6l ("ele").

Constantes de tipo `float` são criadas acrescentando-se 'F' ou 'f' no final:
4. 345346e+6F ou 4.345346e+6f.

Variáveis de tipo `long double` ocupam mais espaço e tomam mais tempo de processamento que variáveis de tipo `double` e variáveis de tipo `double` ocupam mais espaço e tomam mais tempo de processamento que variáveis de tipo `float`.

2.2. Funções de Entrada/Saída

Nesta seção são examinadas algumas das funções existentes na biblioteca da linguagem C utilizadas para entrada e saída de dados: as funções `printf()`, `scanf()`, `getche()`, `getchar()`, `getch()`, `putchar()` e `putch()`.

Estas funções são necessárias para que o usuário do programa possa introduzir dados no programa e que este possa devolver os resultados processados.

Estas funções pertencem às bibliotecas stdio.h e conio.h, oferecidas juntamente com todo pacote comercialmente disponível de compilador de linguagem C. Para que estas funções possam ser incluídas em um programa em C torna necessária a inclusão da diretiva no cabeçalho do programa:

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

2.2.1. Função printf()

A função `printf()` é uma função de impressão em tela bastante flexível em seu uso.

A função `printf()` tem o formato geral conforme o descrito abaixo:

```
#include <stdio.h>
main()
{
  int idade = 30;
  float altura = 1.75;

  printf( "Ricardo tem:\n%f m de altura e %d anos.", altura, idade );
}
```

Nos parâmetros de uma função `printf()` encontramos primeiro uma parte entre aspas que contém os caracteres que serão impressos pelo comando (o que pode incluir sequências de escape como no exemplo acima).

Na segunda parte temos uma lista que pode ser formada por qualquer combinação de variáveis, constantes e expressões.

Na primeira parte algumas sequências iniciadas com o símbolo '%' (denominadas de especificadores de formato) não são impressas, mas sim substituídas por valores que são dados pelos elementos da segunda parte.

Deste modo, o exemplo resultará na saída abaixo:

```
Ricardo tem:
1.750000 m de altura e 30 anos
```

Existem outros especificadores de formato. Por exemplo, para caracteres é %c:

```
printf( "A letra %c eh a primeira de Ricardo.\n", 'R' );
```

Relacionamos abaixo alguns especificadores de formato:

Sequência	Uso
%d	Inteiros em decimal.
%ld	Inteiros `long` em decimal.
%lld	Inteiros `long long` em decimal.
%c	Caracter traduzido segundo a tabela ASCII.
%e	Número em ponto flutuante - notação usando exponencial.
%f	Número em ponto flutuante - notação decimal.
%g	Número em ponto flutuante - notação usando exponencial ou decimal.
%Le	Número `long double` - notação usando exponencial.
%Lf	Número `long double` - notação decimal.
%Lg	Número `long double` - notação usando exponencial ou decimal.
%%	O símbolo '%'.

Além do modo descrito, os especificadores de formato para valores numéricos podem ser alterados para ajustar o formato dos valores a serem impressos.

Modificação	Significado
digito	Indica a largura mínima do número a ser impresso. Por exemplo: %4d quer dizer que o número será impresso com quatro dígitos (maior se necessário).
.digito	Indica precisão. Para números em ponto flutuante quantas casas depois da vírgula serão usadas. Por exemplo: %6.2f indica um número que será impresso com um total de seis caracteres (incluindo o sinal e o ponto) com indicação de duas casas depois do ponto decimal.
-	Justificação. Indica que o número será impresso encostado do lado esquerdo do campo. O normal é o número ser impresso terminando no final do campo especificado pelo tamanho. Por exemplo: %-10d
l	Especifica que o valor a ser impresso é do tipo long. Por exemplo: %6ld
ll	Especifica que o valor a ser impresso é do tipo long long. Por exemplo: %6lld
L	Especifica que o valor a ser impresso é do tipo long double. Por exemplo: %6Lf

2.2.2. Função getche()

A função `getche()` (que é uma abreviatura de get character with echo - lê um caracter e ecoa na tela) faz a entrada de um caracter teclado. Por exemplo:

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

main()
{
  char ch;

  printf( "Pressione uma tecla: " );
  ch = getche();
  printf( "\nA tecla pressionada foi %c.", ch );
}
```

2.2.3. Funções getch() e getchar()

A função `getch()` (que é uma abreviatura de get character without echo - lê um caracter sem ecoar na tela) faz a entrada de um caracter teclado. Por exemplo:

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

main()
{
  char ch;

  printf( "Pressione uma tecla: " );
  ch = getch();
  printf( "\nA tecla pressionada foi %c.", ch );
}
```

2.2.4. Função scanf()

A função `getche()` devolve apenas valores de tipo caracter. Para se fazer a entrada de dados de outros tipos (inclusive de caracter) pode-se utilizar a função `scanf()`.

A estrutura de uma chamada de função `scanf()` é semelhante a da `printf()`.

```
#include <stdio.h>

main()
{
  float anos;

  printf( "Entre sua idade em anos: " );
  scanf( "%f", &anos );
  printf( "Voce tem %.0f dias de idade.\n", anos * 365 );
}
```

Os especificadores de formato usados são os mesmos. A única diferença significativa é o símbolo '&' precedendo o nome da variável (este símbolo, na realidade, representa um operador denominado de operador endereço, o que significa que passamos para a função `scanf()` o endereço das variáveis a serem lidas pela função e veremos mais a respeito deste operador na seção sobre ponteiros, por enquanto fixemos apenas que se deve colocar um '&' antes do nome da variável ao se usar o `scanf()`). A função `scanf()` pode ser usada para a leitura de vários números:

```
#include <stdio.h>

main()
{
  float x, y;

  printf( "Tecle dois numeros: " );
  scanf( "%f %f", &x, &y );
  printf( "%f + %f = %f.\n", x, y, x + y );
}
```

2.2.5. Função putch() e putchar()

A função `putch()` (que é uma abreviatura de print character - imprime um caracter) faz a saida de um caracter. Por exemplo:

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

main()
{
  char ch;

  printf( "Pressione uma tecla: " );
  ch = getche();
  printf( "\nA tecla pressionada foi " );
  putch( ch );
}
```

2.3. Operadores

Um dos tipos de comando mais importantes para a construção de programas em linguagem C são as expressões. Expressões são construídas através do uso de variáveis e constantes relacionadas através de operadores. Veremos nesta seção alguns operadores elementares: Atribuição, Adição, Subtração, Divisão, Multiplicação, Resto, Sinal, Incremento e Decremento, Atribuição Aritmética e Operadores Relacionais.

A importância dos operadores vem do fato de que estes se constituem nas ferramentas para a manipulação de dados e variáveis em um programa.

A combinação de diversos operadores, constantes e variáveis em um mesmo comando forma uma expressão. Uma expressão com diversos operadores é avaliada em uma ordem univocamente determinada. A ordem com a qual uma combinação de operadores agrupados em uma expressão é avaliada é denominada de precedência.

2.3.1. Atribuição

Um operador de atribuição se apresenta na forma de uma variável seguida de um caracter ‘=’ e, em seguida de outra variável, constante ou uma expressão que pode incluir outros operadores.

Em C, um operador de atribuição avalia o valor de uma expressão e faz com que uma variável assuma este valor. Assim, em uma atribuição do tipo:

```
ano = 2009;
```

O símbolo ‘=’ não significa "igual" no seu sentido matemático. Em C o símbolo "=" é denominado de operador de atribuição e quer dizer o seguinte: atribua o valor 2009 à variável ano. Tal distinção faz muito sentido quando se interpreta o significado do comando abaixo:

```
i = i + 1;
```

Do ponto de vista matemático, tal identidade não teria sentido. Em linguagem C um comando deste tipo significaria o seguinte: "Tome o valor atual da variável `i`, some 1 a este valor e atribua o resultado à variável `i`".

O operador de atribuição resulta, por si só em um valor. Assim, no exemplo abaixo:

```
a = b = 1;
```

O operador de atribuição `b = 1` tem o valor 1 como resultado e este valor é atribuído, então, para a variável `a` (a avaliação é feita da direita para a esquerda).

2.3.2. Operador de Adição: +

O operador de adição faz com os dois valores a cada lado do operador sejam somados. Por exemplo, o comando:

```
printf( "%d", 5 + 5 );
```

Imprimirá o número 10 e não a expressão 5 + 5.

O operador de adição pode ser utilizado tanto com variáveis como com constantes. Assim, o comando:

```
preco = custo + lucro;
```

Fará com que o computador determine o valor atual de cada variável, some tais valores e atribua o resultado à variável `preco`. Este comando não altera o valor das variáveis `custo` e `lucro`.

2.3.3. Operador de Subtração: -

O operador de subtração faz com o valor à direita do operador seja subtraído do valor à esquerda. Por exemplo, o comando:

```
idade = 2009 – ano_de_nascimento;
```

Fará com que o computador determine o valor atual da variável `ano_de_nascimento`, subtraia este valor de 2009 e atribua o resultado à variável `idade`.

2.3.4. Operador de Multiplicação: *

O operador de multiplicação faz com os dois valores a cada lado do operador sejam multiplicados. Por exemplo, o comando:

```
imposto = receita * 0.15;
```

Fará com que o computador determine o valor atual da variável `receita`, multiplique este valor por 0.15 e atribua o resultado à variável `imposto`.

2.3.5. Operador de Divisão: /

O operador de divisão faz com o valor à esquerda do operador seja dividido pelo valor à direita. Por exemplo, o comando:

```
premio = rateio / ganhadores;
```

Fará com que o computador determine o valor atual das variáveis `rateio` e `ganhadores` divida o primeiro pelo segundo e atribua o resultado à variável `premio`.

A operação de divisão funciona de modo diferente quando os valores usados são inteiros quando comparados à divisão em ponto flutuante. A divisão de valores em ponto flutuante resulta em valores em ponto flutuante, mas a divisão de valores inteiros também resulta em valores inteiro. O fato de um número inteiro não possuir parte fracionária faz com que uma divisão do tipo 29 / 10 tenha sua parte fracionária descartada (o resultado dessa divisão é 2). Este processo de se remover a parte fracionária do resultado é denominado de truncamento.

2.3.6. Operador de Resto: %

O operador de resto resulta do resto da divisão inteira do valor à esquerda do operador seja dividido pelo valor à direita. Por exemplo, o comando:

```
resto = 39 % 10;
```

Fará com que o computador atribua o valor 9 à variável `resto`.

2.3.7. Operador de Sinal: -

O operador de sinal é usado para indicar ou modificar o sinal de um valor. Por exemplo:

```
lucro = -25;
prejuizo = -lucro;
```

Atribui o valor 25 à variável prejuízo. Este tipo de operador é denominado de unário porque manipula um único operando, ao contrário dos demais vistos até aqui.

2.3.8. Operador de incremento e decremento: ++ e --

O operador de incremento desempenha uma tarefa simples; ele incrementa (soma um) ao valor de seu operando (é um operador unário).

Este operador pode ser usado com o ++ antes da variável afetada ou com o ++ depois da variável afetada. A diferença entre ambos se refere ao momento em que o incremento é efetuado. No primeiro caso, o incremento ocorre no início do comando, antes de qualquer outra operação ser executada. No segundo caso, o incremento é feito após a execução de todas as operações no comando. Assim:

```
i = 5;
j = 8;
k = ++i * j++;
```

Resultará nos valor 6, 9 e 48 para `i`, `j` e `k`. A variável `k` ficou com o valor 48 como resultado da multiplicação de 6 e 8 pois `i` já sofreu o incremento antes do produto e `j` depois.

O operador de decremento opera de modo semelhante. Com a diferença de que a operação feita é de decremento, ou seja, de se subtrair um da variável.

2.3.9. Precedência de operadores

A precedência de operadores é a regra que permite definir qual a ordem com que as operação são realizadas dentro de uma expressão que reúna diversos operadores.

Por exemplo, a expressão abaixo:

```
x = 3 + y * 4;
```

Possui dois operadores: + e *. A precedência permite que se resolva qual será avaliado primeiro: se primeiro fazemos a soma e em seguida multiplicamos o resultado por 4 ou se fazemos primeiro o produto e depois somamos 3 ao resultado.

Na expressão em questão, a multiplicação será feita primeiro e depois a soma. Caso se quisesse fazer primeiro a soma devem ser usados os parênteses:

```
x = (3 + y) * 4;
```

Deste modo, a ordem de precedência dos operadores aritméticos e de incremento na linguagem C está tabelada abaixo:

Operador	Associatividade
()	Esquerda para direita.
++ --	Esquerda para direita.
- (unário)	Esquerda para direita.
* /	Esquerda para direita.
+ - (binário)	Esquerda para direita.
=	Direita para esquerda.

2.3.10. Atribuição Aritmética

Um operador de atribuição aritmética é uma forma compacta de operação de soma, subtração, multiplicação, divisão ou resto. Com o uso de atribuição aritmética o comando:

```
i = i + j;
```

Fica:

```
i += j;
```

Do mesmo modo, os operadores relacionados acima podem ser usados na construção de operadores de atribuição aritméticos:

```
+= atribuição com soma:            i+= j;      equivale a i= i + j;
-= atribuição com subtração:       i-= j;      equivale a i= i - j;
*= atribuição com multiplicação:   i*= j;      equivale a i= i * j;
/= atribuição com divisão:         i/= j;      equivale a i= i / j;
%= atribuição com resto:           i%= j;      equivale a i= i % j;
```

A atribuição aritmética tem a mesma precedência que o operador de atribuição.

2.3.11. Operadores Relacionais

Os operadores relacionais efetuam comparações entre dois valores (constantes, variáveis ou expressões). Na linguagem C, se uma comparação resultar em verdadeiro, o operador relacional correspondente resulta em 1. Do contrário, resulta em 0.

Na linguagem C existem 6 operadores relacionais conforma a tabela abaixo:

Operador	Significado
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a
= =	Igual a
!=	Diferente de

Os operadores relacionais são os de menor precedência dentre todos os operadores. Assim, a expressão:

```
1 < 2 + 4;
```

Resulta no valor 1 (correspondente a verdadeiro) pois 1 é menor que 6.

2.3.12. Erros de arredondamento e overflow

Na execução de operações em ponto flutuante é possível que ocorram erros devido ao arredondamento que ocorre com a precisão finita com que os números são representados em um computador. Deste modo, uma operação do tipo:

```
main()
{
  float a;

  a = 1.0 + 1.0E30;
  a = a - 1.0E30;
  printf( "%f\n", a );
}
```

imprimirá 0.000000 como resposta porque o primeiro comando de atribuição arredondará o resultado da operação para a precisão de 7 dígitos de uma variável do tipo float (muito aquém dos 30 dígitos necessários para a operação acima).

É possível que durante uma operação resulte em um valor fora da faixa para a qual o tipo do dado é definido. Tal situação é denominada de overflow.

```
main()
{
  float a;

  a = 1.0E30 * 1.0E30;
  printf( "%f\n", a );
}
```

O resultado decorrente desta operação dependerá da implementação de cada compilador. É responsabilidade do programador prevenir-se contra a ocorrência ou contra os efeitos de tais situações.

3. Loops

Quase sempre, o que vale a pena ser feito, vale a pena ser feito mais que uma vez. Em programação, uma determinada ação deve ser executada repetidas vezes e, frequentemente, com alguma variação nos detalhes a cada vez. O mecanismo com o qual realizamos estas repetições é denominado de "loop".

Existem, basicamente, três tipos de implementações de loops em C:

Loops do tipo "`for`".
Loops do tipo "`while`".
Loops do tipo "`do while`".

3.1. Loops do tipo "`for`"

Um loop do tipo "`for`" apresenta a estrutura abaixo:

```
for( exp1; exp2; exp3 )
  comando;
```

A expressão `exp1` é uma inicialização. É executada apenas uma vez, exatamente no momento em que se vai iniciar o processamento do loop. A segunda expressão (`exp2`) é uma condição de teste. É avaliada antes de cada possível passagem do loop. Quando esta expressão resultar em valor falso o loop é terminado. A terceira expressão (`exp3`) é avaliada ao final de cada loop. A cada passagem do loop o comando é executado uma vez.

O loop "`for`" é próprio para quando se deseja executar o loop um número determinado de vezes. No exemplo abaixo, o comando no loop é executado 256 vezes.

```
#include <stdio.h>

main()
{
  int n;

  printf("Tabela ASCII:\n");
  for( n = 0; n < 256; n++ )
    printf( " Caracter : %c Codigo: %d\n", n, n );
}
```

Caso seja executado mais de um comando dentro do loop, estes ficam delimitados por chaves {}:

```
#include <stdio.h>

main()
{
  int i, ano;
  float atual, taxa;

  printf( "Estima a populacao nos proximos anos..\n" );
  printf( "Entre com a populacao atual: " );
  scanf( "%f", &atual );
  printf( "Entre com a taxa de crescimento: " );
  scanf( "%f", &taxa );
  printf( "Entre com a ano atual: " );
  scanf( "%d", &ano );

  for( i = 1; i <= 10; i++ )
  {
    ano++;
    populacao *= taxa;
    printf( "ano %d populacao= %.0f\n", ano, atual );
  }
}
```

Os loops "`for`" podem ser aninhados (um loop colocado dentro do outro):

```
#include <stdio.h>

main()
{
  int i, j;

  printf( "Tabuada do 1 ao 10:\n" );
  for( i = 1; i <= 10; i++ )
  {
    for( j = 1; j <= 10; j++ )
      printf( " %2d*%2d= %3d", i, j, i * j );

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

O loop "**for**" é bem flexível. Pode-se decrementar a contagem ao invés de incrementar:

```
printf( "Contagem regressiva...\n" );
for( n = 10; n > 0; n-- )
 printf( "%d segundos\n", n );
```

Pode-se ter a contagem em qualquer passo:

```
 for( n = 10; n < 230; n = n + 15 )
   printf( "%d \n", n );
```

Pode-se contar letras ao invés de números:

```
printf( "Alfabeto minusculo:\n" );
for( ch = 'a'; ch <= 'z'; ch++ )
  printf( "%c ", ch );
```

3.2. Loops do tipo "`while`"

Os loops do tipo "`while`" apresentam a forma abaixo:

```
while( expresão )
  comando;
```

No comando "`while`", a expressão é avaliada ao início do loop. Se esta resultar verdadeira, então o comando é efetuado e, então, a expressão é reavaliada reiterando o processo que só terminará quando a expressão for avaliada e se tornar falsa.

O loop "`while`" é próprio para quando se deseja executar o loop um número indeterminado à priori de vezes.

Por exemplo:

```
#include <stdio.h>

main()
{
  char ch;

  printf( "Termine pressionando Enter.\n" );
  while ( ch = getche() != \n )
    printf( " Caracter teclado: %c Codigo: %d\n", ch, ch );
}
```

3.3. Loops do tipo "do while"

Os loops do tipo "do while" apresentam a forma abaixo:

```
do
  comando;
while (expresão);
```

A principal diferença entre um loop "while" e um "do while" é que, no segundo caso, o comando é sempre executado pelo menos uma vez independentemente da possibilidade da expressão ser avaliada como falsa já ao início do loop.

O loop do tipo "do while" é próprio para quando se deseja executar o loop um número indeterminado de vezes, porém pelo menos uma vez.

Por exemplo:

```
#include <stdio.h>

main()
{
  char ch;
  int n = 1;

  do
    printf( "\nVoltas no loop: %d Continua ? (s/n) ", n++ );
  while( ch = getche() != 's' )
}
```

4. Decisões

Um tipo de comando necessário para a criação de programas úteis, versáteis e inteligentes são os de decisão e escolha de alternativas.

Na linguagem C são definidos comandos de controle que permitem desviar o fluxo de execução de um programa de acordo com o resultado de testes ou resultados obtidos para operações.

As palavras-chave para estes comandos estão relacionadas abaixo:

if
if-else
switch-case-break-default

4.1. Decisões do tipo "`if`"

O comando de controle "`if`" tem a estrutura abaixo:

```
if( expressão )
  comando;
```

Durante a execução do programa a expressão é avaliada e, se o resultado for verdadeiro, somente então o comando é executado.

Embora qualquer tipo de expressão possa ser empregado (neste caso o resultado falso corresponde a um valor inteiro igual a zero e verdadeiro corresponde a um valor inteiro diferente de zero), normalmente são utilizadas expressões com operadores relacionais e lógicos.

No caso de se desejar condicionar a execução de uma sequência de comandos a uma expressão, define-se uma relação de comandos a serem executados delimitando-os com chaves {}.

```
if( expressão )
{
  comando1;
  comando2;
  comando3;
}
```

Se, ao avaliar-se a expressão resultar em verdadeiro (um) os comandos comando1 comando2 e comando3 serão executados. Senão, nenhum deles será executado.

4.2. Decisões do tipo "`if-else`"

O comando de controle "**`if-else`**" têm a estrutura abaixo:

```
if( expressão )
  comando1;
else
  comando2;
```

Neste comando de controle, a expressão é avaliada. Se o resultado for verdadeiro (um), o comando1 é executado. Se o resultado for falso (zero) o comando2 é executado. A execução dos dois comandos é mutuamente exclusiva. Apenas um dos dois comandos será executado.

Caso seja necessário o uso de mais comandos, tanto para o caso da expressão ser verdadeira como no caso da expressão ser falsa, as chaves {} também podem ser empregadas.

No caso de se encadear diversos comandos de controle "**`if`**" em sequência, o comando "**`else`**" corresponderá à condição falsa do comando "if" anterior mais próximo que não tenha um "**`else`**" ainda associado:

```
if( exp1 )
  if( exp2 )
    comando_a;    /* exp1 == verdadeiro && exp2 == verdadeiro   */
  else            /* corresponde à negação do if( expr2) acima. */
    comando_b;    /* exp1 == verdadeiro && exp2 == falso        */
else              /* corresponde à negação do if( expr2) acima. */
  comando_c;      /* exp1 == falso */
```

O `comando_a` será executado se a expressão `exp1` for verdadeira e a expressão `exp2` também for verdadeira. O `comando_b` será executado se a expressão `exp1` for verdadeira e a expressão `exp2` for falsa. Para que o `comando_c` seja executado basta que a expressão `exp1` seja falsa (`exp2` sequer é avaliada).

4.3. Decisões do tipo "`switch`"

Nos comandos de controle "**`if`**" e "**`if-else`**" a decisão tomada permitia a escolha dentre duas possíveis alternativas. Com o comando de controle "**`switch`**" pode-se escolher um comando a ser executado dentre diversos possíveis valores de uma expressão. A estrutura de um comando de controle "**`switch`**" é descrito abaixo:

```
switch( exp )
{
  case x1: comando_x1;
  break;
  case x2: comando_x2;
  break;
  case x3: comando_x3;
  break;
  case x4: comando_x4;
  break;
  default: comando_x5;
}
```

O comando **`switch`** mostrado é equivalente ao comando abaixo:

```
if( exp == x1 )
  comando_x1;
else
  if( exp == x2 )
    comando_x2;
  else
    if( exp == x3 )
      comando_x3;
    else
      if( exp == x4 )
        comando_x4;
      else
        comando_x5;
```

A expressão `exp` é avaliada. Se o resultado for igual a `x1`, então `comando_x1` é executado. Se o resultado for igual a `x2`, então `comando_x2` é executado. Se o resultado for igual a `x3`, então `comando_x3` é executado. Se o resultado for igual a `x4`, então `comando_x4` é executado. Se o resultado for diferente de todos eles, então `comando_x5` é executado.

Não há limites para o número de "**`cases`**" que podem ser usados em um comando "**`switch`**". Os valores x1, x2, x3 e x4 usados no comando são constantes inteiras ou caracteres (valores em ponto flutuante não são permitidos). Caso haja valores iguais em dois ou mais diferentes "**`cases`**" será executado o comando ligado ao primeiro "**`case`**".

Um "**case**" em particular recebe o nome de "**default**" e, normalmente, é colocado no final do comando "**switch**" (embora possa ser colocado em qualquer posição dentro do comando "**switch**"). O comando associado ao "**default**" é executado sempre que nenhum "**case**" corresponda ao valor associado à expressão do "**switch**".

Um comando adicional associado ao comando de "**switch**" é o comando "**break**" que determina o término da sequência de comandos associados ao "**case**". Assim, na ausência de um "**break**", a execução de um "**case**" pode prosseguir no "**case**" seguinte.

Por exemplo:

```
switch( getche() )
{
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u': printf("Você teclou uma vogal.\n");
            break;
  default: printf("Você teclou uma consoante.\n");
}
```

4.4. O operador condicional

O operador condicional é um modo abreviado para um comando "**`if-else`**" com uso para comando de atribuição. A estrutura para o operador condicional é descrita abaixo:

```
x = exp1 ? exp2 : exp3;
```

Se `exp1` for verdadeiro, a variável `x` recebe o valor de `exp2`. Se for falso de `exp3`.

A expressão acima é equivalente ao comando abaixo:

```
if( exp1 )
  x = exp2;
else
  x = exp3;
```

Por exemplo, o comando abaixo atribui à variável `x` o valor absoluto da variável `y`:

```
x = (y < 0) ? -y : y;
```

5. Funções

Um programa em linguagem C constitui-se em uma coleção de funções (onde pelo menos uma delas é chamada de `main()` e é a função executada ao iniciar o programa). Algumas funções (como o `printf()` e o `scanf()`) são oferecidas como parte da biblioteca da linguagem C. Descreveremos, nesta seção, como que o programador pode especificar e usar suas próprias funções.

O programador pode definir quantas funções desejar em seus programas e chamá-las de dentro de qualquer função que queira. Não há qualquer restrição de ordem ou hierarquia na definição das funções para o programador.

O uso de uma função passa por duas etapas. Na primeira nós definimos a função. Nesta fase nós devemos declarar seu protótipo no cabeçalho do programa e definir a função relacionando os comandos que fazer sua implementação.

O protótipo da função corresponde a sua declaração. Deste modo, informa ao compilador o seu nome, o tipo de dado que retorna (caso retorne algum) e que tipo de parâmetros ele recebe (caso receba algum).

Na segunda fase, a função é executada. Uma vez definida a função, esta pode ser chamada de diferentes pontos do programa. Este é um dos motivos para se utilizar funções, pois deste modo economiza-se espaço de memória.

Contudo, mesmo que seja para se chamar uma função apenas de um ponto do programa, vale a pena utilizar-se de funções para implementar trechos de programa pois isto torna o programa mais estruturado e modular.

Por exemplo, um programa que faça o seguinte:

leia uma lista de números
ordene os números
encontre sua média
desenhe um histograma

Pode ser feito do seguinte modo:

```
int main()
{
  leia_lista();
  ordena_lista();
  faz_media();
  faz_histograma();
}
```

Onde o programador decompõe sua tarefa em tarefas mais simples.

Uma função pode conter uma chamada para outra função. Esta característica é denominada de reentrância. Este é um modo comum para criar funções progressivamente mais sofisticadas a partir de funções mais simples.

Uma função pode conter chamadas para a própria função (uma situação semelhante aos loops). Esta situação é denominada de recursão. Recursão pode ser obtida em situações onde há uma cadeia fechada de funções que se chamam. Recursão é um recurso muito frequentemente usado em programação.

5.1. Funções simples

A estrutura para definir uma função simples está descrita abaixo:

```
void nome(); /* Prototipo. */

main()
{
  .
  .
  nome(); /* Chamada da função. */
  .
  nome(); /* Chamada da função. */
  .
  nome(); /* Chamada da função. */
  .
  .
}

void nome() /* Definição da função. */
{
  .
  .
}
```

Na declaração da função (protótipo) o nome "`void`" define que a função não retorna dado de nenhum tipo. Especifica-se o nome da função (nome) e que não se utiliza de parâmetros.

A cada chamada da função ela é executada. Na definição da função, são relacionados os comando que a compõe. Por exemplo:

```
#include <stdio.h>

void separador(); /* Prototipo da funcao. */

main()
{
  separador(); /* Chamada da funcao. */
  printf( "EU GOSTO DE LINGUAGEM C\n" );
  printf( "Departamento de Engenharia\n" );
  separador(); /* Chamada da funcao. */
}

void separador() /* Definicao da funcao. */
{
  int i;

  for ( i = 1; i <= 60; i++ )
    printf( "%c", '*' );
  printf( "\n" );
}
```

5.2. Funções que retornam um valor

Podem-se definir funções que retornam valores. Para tanto basta declarar a função com o tipo de dado que se deseja retornar (diferente do tipo `void`, usado quando não se deseja retornar dados) e terminar a execução da função com o comando `return`, conforme a estrutura descrita abaixo:

```
tipo nome(); /* Prototipo. */

main()
{
  tipo a, b, c;
  .
  .
  a = nome(); /* Chamada da função. */
  .
  b = nome(); /* Chamada da função. */
  .
  c = nome(); /* Chamada da função. */
  .
  .
}

tipo nome() /* Definição da função. */
{
  .
  .
  .
  return x;
}
```

Na estrutura acima, o tipo pode ser qualquer um dos existentes para a linguagem C (`int`, `char`, `float`, etc) e o comando `return` que aparece na definição da função faz com que seja o valor da expressão representada por `x` seja avaliado e este valor é retornado para o ponto onde a função foi chamada.

Por exemplo:

```
#include <stdio.h>

char get_numero(); /* Prototipo da funcao. */

main()
{
  char a;

  printf( "Tecle um digito numerico: " );
  a = get_numero(); /* Chamada da funcao. */
  printf( "O codigo do digito teclado (%c) eh: %d\n", a, a );
}

char get_numero() /* Definicao da funcao. */
{ /* Le o teclado ate o usuario teclar um caracter numerico. */
  char a;

  while( 1 )
  switch ( a = getch() )
  {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9': return a;
    default:
  }
}
```

5.3. Uso de argumentos em funções

Podem ser definidas funções que recebem argumentos ao serem chamadas. Os argumentos são passados dentro dos parênteses colocados após o nome da função. No momento em que a função é chamada, os valores colocados nos parênteses substituem os declarados na definição da função, de modo que a função pode utilizá-los.

Por exemplo:

```
#include <stdio.h>

void barra( int ); /* Prototipo da funcao. Recebe um parametro. */

main()
{
  printf( "Resultado final do campeonato:\n" );
  printf( "Goya\n" );
  barra( 35 ); /* Chamada da funcao. */
  printf( "Ulisses\n" );
  barra( 31 ); /* Chamada da funcao. */
  printf( "Julio\n" );
  barra( 22 ); /* Chamada da funcao. */
}

void barra( int tamanho ) /* Definicao da funcao. */
{
  int i;

  for( i = 1; i <= tamanho; i++ )
    printf( "%c",'=' );
  printf( "\n\n" );
}
```

No exemplo, a função `barra()` foi definida com um parâmetro denominado de tamanho. A cada vez que a função foi chamada, o valor do tamanho foi substituído (por 35, 31 e 22 para cada chamada no programa acima).

5.4. Funções e variáveis globais

Os comandos de declaração executados no interior de funções definem variáveis cuja validade se restringe ao interior da função e por isso são invisíveis para outras funções. Tais variáveis são denominadas de locais e nos permitem definir variáveis com o mesmo nome, mas com diferentes usos em diferentes funções.

Contudo, podem-se declarar variáveis que sejam visíveis e modificáveis por todas as funções do programa sem que seja necessário passá-las como parâmetro para todas as funções. Tais variáveis são ditas "globais" ou "externas".

O uso de variáveis globais é irrestrito. Contudo, o uso indiscriminado de variáveis globais é perigoso. Torna o programa difícil de modificar e corrigir. Como são visíveis por todas as funções do programa, estão sujeitas a modificação por qualquer uma delas e, além disso, usam a memória de modo menos eficiente que o usado com variáveis locais.

Por exemplo:

```
#include <stdio.h>

void impar();    /* Prototipo. Testa se Numero eh Impar */
void negativo(); /* Prototipo. Testa se Numero eh Positivo */

int numero;      /* Variavel global. */

int main()
{
  printf( "Tecle um numero: " );
  scanf( "%d", &numero );
  impar();
  negativo();

  return 0;
}

void impar()
{
  if( numero % 2 ) /* Manipula uma variavel global. */
    printf( "O numero eh impar.\n" );
  else
    printf( "O numero eh par.\n" );
}

void negativo()
{
 if( numero >= 0 ) /* Manipula uma variavel global. */
   printf("O numero eh positivo.\n");
 else
   printf("O numero eh negativo.\n");
}
```

5.5. Classes de armazenamento

As funções em execução são entidades que são criadas quando são chamadas e deixam de existir quando retornam ao ponto de onde foram chamadas. Quando, dentro de uma função há uma chamada para outra função (reentrância) ou para ela mesma (recursão), uma nova entidade é criada. O mesmo, normalmente, ocorre com variáveis locais: elas são criadas quando a função é chamada e disponibilizadas para reuso quando a função é terminada.

5.5.1. Variáveis estáticas

Uma variável pode ter sua existência desvinculada da função onde está declarada e valer durante toda a execução do programa se for declarada como estática. Para isso, é necessário usar o modificador "`static`" na sua declaração. Quando isso ocorre, ela preserva o seu valor, mesmo quando a função retorna, podendo ser reutilizada quando a função é novamente chamada.

Isso não quer dizer que a variável estática possa ser manipulada por outras funções. O caráter estático não muda a característica de visibilidade de uma variável local.

Apenas um exemplar da variável estática é criado para a função. Assim, em caso de recursão, uma variável estática é compartilhadas entre as diferentes instâncias da mesma função.

5.5.2. Variáveis automáticas

Uma variável pode ter sua existência vinculada da função onde está declarada e valer enquanto a função está sendo executada se for declarada como automática. Para isso, pode-se usar o modificador "`auto`" na sua declaração (ou não usar modificador nenhum). Em caso de recursão, um novo exemplar da variável é criado a cada chamada da função, assim, variáveis automáticas não são compartilhadas entre diferentes instâncias de chamadas de uma função recursiva.

5.5.3. Constantes

Uma variável que não deve ser modificada durante a execução da função pode ser declarada como constante. Para isso, pode-se usar o modificador "`const`" na sua declaração. O uso de expressões que alteram o valor de uma variável deste tipo resulta em erro de compilação. Constantes podem ser estáticas ou automáticas, ou seja, os modificadores podem ser acumulados.

Constantes devem ser inicializadas na declaração. A ausência desta inicialização também é um erro de sintaxe.

5.5.4. Registradores

Uma variável pode ser armazenada em um registrador do processador, ao invés de uma posição de memória. Para isso, pode-se usar o modificador "**`register`**" na sua declaração. O uso de variável deste tipo pode resultar em melhor desempenho. Os compiladores modernos fazem alocação de registradores automaticamente, o que torna o uso deste modificador quase desnecessário. O uso deste tipo de modificador pode ter algum benefício quando se produz código para ser executado em micro-controladores.

5.5.7. Inicialização de variáveis da declaração

Uma variável pode ser inicializada na declaração no programa. Em tempo de execução, isso significa que o valor inicial é colocado na variável no momento em que a variável é criada.

Variáveis automáticas são criadas cada vez que a função onde estão declaradas é chamada. Assim, havendo inicialização na declaração, a variável automática é sempre inicializada no momento da chamada da função.

Variáveis estáticas não são recriadas a cada vez que a função onde estão declaradas é chamada. Assim, havendo inicialização na declaração, a variável estática é inicializada apenas uma vez no início do programa.

Por exemplo:

```
#include <stdio.h>

void f();

int main()
{
  int i;

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

  return 0;
}

void f()
{
  static int c = 1;

  printf( "esta funcao foi chamada %d vezes.\n", c++ );
}
```

5.6. Funções de biblioteca

Junto com as ferramentas de desenvolvimento (como o MingW) são oferecidas bibliotecas com funções desenvolvidas e testadas para uso pelo desenvolvedor e incluem funções de Entrada/Saída (stdio.h), Matemáticas (math.h), Teste de tipos de caracteres (ctype.h), Entrada/Saída de caracteres (conio.h), Manipulação de strings (strings.h), Tempo (time.h) entre muitas outras.

Estas funções permitem que o programador poupe trabalho no desenvolvimento de programas.

Funções matemáticas(ctype.h):

Trigonométricas (argumento em radianos):

double acos(double)	double asin(double)	double atan(double)
double cos(double)	double sin(double)	double tan(double)

Manipulação e arredondamento:

double atof(char *)	int abs(int)	double ceil(double)
double fabs(double)	double floor(double)	double trunc (double)
double round(double)		

Exponencial, logarítimo e funções hiperbólicas:

double cosh(double)	double sinh(double)	double tanh(double)
double exp(double)	double log(double)	double log10(double)
double pow10(double)	double pow(double, double)	

Teste de Tipos (ctype.h)

int isalpha(char)	‘a’-‘z’ ou ‘A’-‘Z’
int isupper(char)	‘A’-‘Z’
int islower(char)	‘a’-‘z’
int isdigit(char)	‘0’-‘9’
int isxdigit(char)	‘0’-‘9’ / ‘a’-‘f’ / ‘A’-‘F’
int isspace(char)	‘\0x9’-’\0xD’,’ \0x20‘
int ispunct(char)	pontuação
int isalnum(char)	‘a’-‘z’ / ‘A’-’Z’ / ‘0’-‘9’
int isprint(char)	imprimível
int isgraph(char)	caractere gráfico
int iscntrl(char)	‘\0’-’\0x1F’
int isascii(char)	‘\0’-’\0x7F’

6. Vetores e Strings

Vetores e strings são nomes dados a arranjos de variáveis onde diversas variáveis de um mesmo tipo e formando um conjunto são tratados coletivamente.

Os vetores são declarados conforme a estrutura abaixo:

```
tipo nome[tamanho];
```

Onde o tipo e o nome seguem as regras já conhecidas para declaração de variáveis e o tamanho nos dá o número de elementos que compõem o conjunto.

Exemplo:

```
#include <stdio.h>

main()
{
  float soma;
  float temperatura[7];
  int i;

  for( i = 0; i < 7 ; i++ )
  {
    printf( "Entre com a temperatura do dia %d:", i );
    scanf( "%f", &temperatura[i] );
  }

  for( soma = 0, i = 0; i < 7 ; i++ )
    soma += temperatura[i];

  printf( "Temperatura media eh de %f:", soma / 7 );
}
```

É importante notar que, na linguagem C, todos os vetores têm sua numeração iniciando em 0 (ou seja, no exemplo acima, o vetor contém posições cuja enumeração vai de 0 a 6).

6.1. Iniciando os dados em um vetor

Os dados contidos em um vetor podem ser inicializados diretamente na sua declaração. Por exemplo:

```
int dias_por_mes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Inicia um vetor de 12 posições com cada posição contendo o número de dias que o mês tem.

Ao iniciarem-se os dados de um vetor não é necessário definir o tamanho do vetor:

```
float precos[] = {20.1, 12.5, 8.8, 6.9, 13.5};
```

O compilador é esperto o suficiente para contar o número de elementos utilizados na definição e criar um vetor do tamanho apropriado.

Contudo, o uso de inicialização de vetores é restrito a variáveis globais, não podendo ser usado em variáveis locais.

6.2. Vetores de duas ou mais dimensões

Em linguagem C podemos definir vetores de duas dimensões:

```
int tabuleiro[8][8];
```

Ou mais dimensões:

```
int cubo[3][2][4];
```

Todas as considerações feitas para vetores valem para os vetores multi-dimensionais. Um aspecto importante é o da iniciação de um vetor multidimensional. Neste caso, a lista de valores empregada para a inicialização deve ser construída de modo que o índice da direita varia mais rapidamente que o índice da esquerda.

Por exemplo:

```
int Cubo[3][2][4] =
 { { { 1, 2, 3, 4},
     { 5, 6, 7, 8} },
   { { 9,10,11,12},
     {13,14,15,16} },
   { {17,18,19,20},
     {21,22,23,24} } };
```

Os valores 3, 2 e 4 são desnecessários para especificar o tamanho do vetor. Foram colocados para maior clareza do que ocorre no exemplo. Por ele, notamos que a posição Cubo[2][1][0] foi inicializada com o valor 21.

6.3. Passando vetores como parâmetro de funções

Para passar um vetor como parâmetro de uma função, basta definir em seu protótipo e cabeçalho a especificação do tipo com o uso dos colchetes:

```
int maximo( int[],int ); /* Prototipo de maximo. */
```

No protótipo do exemplo, a função definida retorna um valor inteiro e tem dois parâmetros. O primeiro parâmetro é um vetor e o segundo é um inteiro.

Por exemplo, para o protótipo acima, podemos escrever a seguinte função:

```
int maximo( int[] lista, int tamanho )
{
  int teste;
  int i;

  teste = lista[0];

  for(i = 0; i < tamanho; i++ )
    if( lista[i] > teste )
      teste = lista[i];

  return teste;
}
```

6.4. Strings

Strings são um caso particular de vetor, são vetores de caracteres. Os strings são utilizados para a manipulação de textos tais como palavras, nomes e frases.

Constantes contendo strings normalmente são definidas na forma de textos entre aspas: "EU GOSTO DE C".

Variáveis contendo strings são declaradas na formas de vetores de variáveis de tipo `char`:

```
char nome[20];
```

Contudo, a inicialização de strings pode ser facilitada pelo compilador.

```
char cumprimento[20] = "Bom Dia";
```

Todo string acaba por representar um bloco de dados em memória onde, em cada posição, é colocado o código correspondente a um caracter do string. Após o último caracter do string, é colocado um valor 0 sinalizando o término do string. (Este formato de string é conhecido como ASCIIZ – ASCII seguido de um zero).

De fato, a inicialização acima é equivalente à inicialização descrita abaixo:

```
char cumprimento[] = {'B','o','m',' ','D','i','a','\0'};
```

Onde o '\0' corresponde ao valor zero que indica o final do string.

6.5. Funções de manipulação de strings

Vamos ver, nesta seção, como são algumas funções para manipulação de strings:

Para manipular strings, as funções `scanf()` e `printf()` dispõem de um especificador de formato apropriado: %s.

```
char nome[20] = "Programa Teste.C"

printf( "Nome atual: %s\n", nome );
printf( "Entre com novo nome: " );
scanf( "%s", nome );
```

O uso do especificador de formato (%s) é muito semelhante ao usado para outros tipos de dados. Com relação à função `scanf()` uma diferença importante surge pois a variável surge sem o símbolo '&' e isto se deve ao fato de que a linguagem C subentende que o uso do nome de um vetor ou string sem o índice significa que o valor indicado é o seu endereço.

De fato `nome` é equivalente a `&nome[0]`.

Além do `scanf()` e `printf()` existem mais duas outras funções apropriadas para manipulação de strings: `gets()` e `puts()`. Seu uso é bastante simples:

```
puts( "Entre com seu nome: " );
gets( Nome );
puts( "Como vai" );
puts( Nome );
```

Como pode ser visto, as funções `gets()` e `puts()` movimentam strings entre variáveis e tela de modo bastante simples.

7. Ponteiros

Outro tipo de dado presente na linguagem C é o ponteiro (pointer). Neste tipo de dado manipula-se não um valor, mas sim um endereço. Os valores são armazenados em memória. Deste modo, variáveis distintas são armazenados em diferentes posições de memória. A cada posição corresponde um endereço diferente e único.

Um ponteiro proporciona um meio de acesso a uma variável sem referenciar esta variável diretamente.

Ponteiros são empregados principalmente para retornar mais de um valor de uma função e para passar vetores e strings de ou para uma função.

O comando de declaração para uma variável do tipo ponteiro tem a estrutura abaixo:

```
tipo *nome;
```

Onde o tipo e o nome seguem as regras para definição já conhecidas. O que especifica que a variável é de tipo ponteiro é o asterisco colocado na frente do nome.

O uso de variáveis de tipo ponteiro é bastante simples:

```
int x,y;
int *px,*py;

x = 1; /* atribui à variável x o valor 1. */
y = 2; /* atribui à variável y o valor 2. */
px= &x; /* atribui ao ponteiro px o endereco da variavel x. */
py= &y; /* atribui ao ponteiro py o endereco da variavel y. */
```

O significado das atribuições acima é bastante simples. O operador ‘&’ é um operador unário que retorna o endereço de seu operando. Assim, podemos manipular o endereço apontado por uma variável de tipo ponteiro.

Outro operador importante é o operador '*' que indica o valor contido em uma posição de memória apontada por uma variável de tipo ponteiro.

Por exemplo:

```
int x,y;
int *px,*py;

x = 1; /* atribui à variável x o valor 1. */
y = 2; /* atribui à variável y o valor 2. */

px = &x; /* atribui ao ponteiro px o endereco da variavel x. */
py = &y; /* atribui ao ponteiro py o endereco da variavel y. */

printf( "O endereco de x eh %d e o valor de x eh %d", px, *px );
```

onde o `*px` significa, literalmente, o valor da posição de memória apontado por `px`.

7.1. Ponteiros e funções

Um dos usos mais apropriados para variáveis de tipo ponteiro é o de passar mais de um parâmetro de retorno para funções. O modo como este mecanismo é implementado é descrito abaixo:

```
void troca( int *px, int *py ); /* Prototipo. */

main()
{
  int x = 4, y = 7;

/*
 * Ao chamar a funcao passamos o endereco
 * das variaveis x e y como parametro.
 */
  printf( "x vale %d e y vale %d\n", x, y );
  troca( &x, &y );
  printf( "x vale %d e y vale %d\n", x, y );
}

void troca( int *px, int *py )
/* Troca os valores das variaveis apontadas por px e py. */
{
  int n;

  n = *px;
  *py = *px;
  *px = n;
}
```

Ao examinarmos a função, verificamos que houve manipulação nos valores das posições cujos endereços foram passados como parâmetro. Deste modo é que conseguimos implementar funções cuja ação é estendido a mais de uma variável (ou seja, retorna mais de um valor).

7.2. Ponteiros e vetores

Ponteiros também oferecem meios bastante convenientes para manipulação de vetores. Por exemplo, o programa abaixo utiliza ponteiros para percorrer um vetor:

```
main()
{
 int n[20];
 int *p;
 int i;
 int soma;

 for( i = 0, p = n, soma = 0; i < 20; i++; p++ )
   soma += *p;
}
```

No exemplo, o programa percorre todo o vetor, acumulando a soma de seus elementos na variável soma. Um detalhe importante do exemplo é a operação `p++` dentro do comando for. Dado que o a variável `p` foi definida como um ponteiro para um inteiro, `p++` significa que o endereço contido em `p` foi acrescido de uma quantidade que corresponde ao número de posições de memória ocupadas pela variável do tipo especificado quando da declaração do ponteiro (o que equivale a dizer que, após o incremento, `p` passa a apontar para a próxima posição dentro do vetor).

Outro aspecto importante é o de como as variáveis de tipo ponteiro são aplicáveis para a passagem de vetores como parâmetros para funções. Ao se passar um ponteiro para um vetor como parâmetros para uma função, trabalhamos sobre a mesma instância dos dados do vetor, ou seja, não criamos uma cópia dos dados para uso dentro da função.

7.3. Ponteiros e strings

Ponteiros também oferecem meios bastante convenientes para manipulação de strings. Por exemplo, o programa abaixo utiliza ponteiros para percorrer um string:

```
main()
{
  char frase[20];
  int *p;
  int contchar;
  int contspaces;

  p = frase;    /* p aponta para o inicio da frase.  */
  contchar = contspaces = 0;
  while ( !*p ) /* Um string termina com um valor 0. */
  {
    if( *p == ' ' )
      contspaces++;
    contchar++;
    p++;
  }
}
```

No exemplo, o programa percorre todo o string, contando o número de espaços em branco e caracteres presentes nele. No final de um string sempre encontramos o valor 0.

8. Arquivos

A linguagem C oferece em sua biblioteca uma série de funções próprias para manipulação de dados guardados em disco (sejam os chamados discos removíveis ou disquetes ou os chamados discos fixos ou rígidos). Nestes tipos de meio, os dados são guardados em blocos denominados de arquivos.

Como os dados guardados em arquivos têm característica de persistência (sua duração se estende além da execução do próprio programa e sobrevive ao desligamento e religamento da máquina), a manipulação de arquivos é apropriada para manuseio e manutenção de bancos de dados entre outras aplicações.

Existem, basicamente, dois tipos de entrada/saída para disco: o chamado modo texto (ou modo padrão) e o modo binário. Ambos os modos serão descritos nas sessões seguintes.

8.1. Fases na manipulação de arquivos

Ambos os modos de acesso a arquivos passam por três etapas: abertura, manipulação (leitura ou escrita) e fechamento.

Na fase de abertura, o programa especifica o nome do arquivo que se deseja manipular, o tipo de entrada/saída a ser efetuado (se no tipo texto ou binário) e se o acesso vai ser de leitura ou escrita.

Na fase de manipulação, o programa pode enviar dados para o disco (escrita em arquivo) ou receber dados vindos do disco (leitura de arquivo).

No fechamento, o programa informa ao sistema operacional que concluiu o uso do arquivo. Nesta fase qualquer dado que esteja em buffer na memória é finalmente escrito no disco.

8.2. Entrada/Saída padrão

A linguagem C oferece uma série de funções para manipulação de arquivos.

Para a manipulação de arquivos, usam-se as funções abaixo:

fopen	Abre arquivo.
fclose	Fecha arquivo.
fcloseall	Fecha todos os arquivos abertos.
ferror	Sinaliza o status atual do arquivo.
perror	Imprime mensagem de erro.

As funções para manipulação de entrada e saída no modo padrão são relacionadas abaixo:

putc	Escreve caracter em arquivo.
getc	Lê caracter de arquivo.
fputs	Escreve string em arquivo.
fgets	Lê string de arquivo.
fprintf	Escreve formatado em arquivo.
fscanf	Lê formatado de arquivo.
fwrite	Escreve bloco em arquivo.
fread	Lê bloco de arquivo.

O controle de acesso ao arquivo pode ser realizado com as funções abaixo:

fgetpos	Indica a posição do ponteiro de acesso ao arquivo.
fseek	Posiciona o ponteiro de acesso em local desejado.
rewind	Posiciona o ponteiro de acesso no início do arquivo.
fflush	Descarrega o stream para o arquivo em disco.
flushallf	flush para todos os arquivos abertos.
feof	Indica se o ponteiro do arquivo atingiu o final.

8.2.1. Abertura de arquivo

A abertura de arquivo é um procedimento de comunicação entre o programa e o sistema operacional. Ela é necessária para que o sistema verifique o acesso ao arquivo, onde ele está localizado, qual o tipo de acesso desejado e definir as áreas de buffers necessárias para a operação do arquivo.

Para abrirmos um arquivo, é necessário declarar-se uma variável de tipo ponteiro para arquivo com o comando:

```
FILE *arquivo;
```

Onde arquivo é um nome de identificador para a variável. De posse da variável, a abertura pode ser efetuada com o seguinte procedimento:

```
arquivo = fopen( "NOME.EXT", CodigoDeAcesso );
```

O "NOME.EXT" é o nome do arquivo a ser aberto. Está localizado no diretório atual e não á procurado no path. Se estiver em outro disco ou diretório deve ser especificado por completo (pro exemplo: "C:\\QC25\\SOURCE\\EXEMPLO.TXT").

O código de acesso é um dentre seis possíveis strings:

`"r"`	Aberto para leitura. O arquivo já deve existir. Se não existir devolve NULL.
`"w"`	Aberto para escrita. Se o arquivo já existe seu conteúdo é perdido. Se não, um arquivo novo é criado.
`"a"`	Append: Se o arquivo já existe, ele é aberto para escrita após o final. Se não, um arquivo para escrita novo é criado.
`"r+"`	Aberto para leitura e escrita. O arquivo já deve existir. Se não existir devolve NULL.
`"w+"`	Aberto para leitura e escrita. Se o arquivo já existisse seu conteúdo original á perdido.
`"a+"`	Abertura para leitura e escrita no final. Se o arquivo não existisse seria criado.

No caso de ocorrem um erro na operação de abertura de arquivo, a função retorna com o valor NULL.

8.2.2. Fechamento de arquivo

O fechamento de um arquivo é feito através da função `fclose()` ou `fcloseall()` na forma abaixo:

```
fclose( arquivo );
fcloseall();
```

No primeiro caso é fechado um arquivo específico. No segundo todos os arquivos que estejam abertos.

8.2.3. Mensagem de erro

A ocorrência de erro pode ser testada com o uso da função `ferror()`. Caso ocorra algum erro ela retorna um valor não nulo. A função `perror()` imprime uma mensagem de erro com um string dado pelo usuário e a mensagem correspondente ao erro ocorrido vinda do sistema:

```
if( ferror( arquivo ) )
 perror( "O erro ocorrido foi" );
```

8.2.4. Entrada e saída de caracteres e strings para arquivos

As funções básicas de entrada e saída de caracteres em arquivos são `getc()` e `putc()` e são usadas da seguinte forma:

```
putc( ch, arquivo );
ch = getc( arquivo );
```

Com `ch` de tipo `char` ou `int`. (Como `int` pode-se usar o fato de que a função retorna o valor EOF (-1) caso o arquivo tenha chegado ao final) e arquivo de tipo ponteiro para arquivo com endereço de arquivo já aberto.

As funções para entrada e saída de strings são usadas conforme descrito abaixo:

```
fgets( string, arquivo );
fputs( string, arquivo );
```

Com string como ponteiro para caracter e arquivo como ponteiro para arquivo apontando para arquivo já aberto.

8.2.5. Entrada e saída formatada para arquivos

Para entrada e saída formatada para arquivos usamos as funções `fprintf()` e `fscanf()`. O uso destas funções é idêntico ao empregado para console (`printf()` e `scanf()`) com acréscimo para a identificação do arquivo em acesso:

```
fprintf( arquivo, especificador de formato, lista de variáveis );
fscanf( arquivo, especificador de formato, lista de variáveis );
```

8.2.6. Entrada e saída de blocos de dados com arquivos

As funções `fread()` e `fwrite()` podem ser usadas para leitura e escrita de blocos de dados para arquivos. Seu formato de uso está descrito abaixo:

```
fread( ponteiro, numero de itens, tamanho do item, arquivo );
fwrite( ponteiro, numero de itens, tamanho do item, arquivo );
```

Onde o ponteiro é o endereço inicial na memória onde vai ser colocado o dado lido ou de onde vem o dado a ser escrito, o número de itens especifica a quantidade de dados a serem movidos e o tamanho do item especifica o número de bytes para representar cada item (1 para `char`, 4 para `int` e assim por diante) e arquivo é um ponteiro para arquivo apontando para um arquivo aberto.

Com o uso de `fread()` e `fwrite()` pode-se escrever estruturas complexas tais como vetores e vetores de strings em disco com um acesso bastante simplificado.

8.3. O controle de ponteiro de acesso a arquivos

A função `fgetpos()` devolve a posição do ponteiro de acesso ao arquivo. É utilizada do seguinte modo:

```
posicao = fgetpos( arquivo ); /* Onde arquivo é um ponteiro para um
                                 arquivo aberto. */
```

A função `fseek()` posiciona o ponteiro de acesso em local desejado. É utilizada do seguinte modo:

```
fseek( arquivo, deslocamento, posicao );
```

Onde o arquivo é o ponteiro para o arquivo aberto, o deslocamento é a distância (em uma variável tipo `long`) em relação à posição especificada. Para especificar a posição são possíveis três valores: SEEK_SET para o início do arquivo, SEEK_CUR para a posição atualmente apontada e SEEK_END para o final do arquivo. O deslocamento pode ser positivo ou negativo.

A função `rewind()` posiciona o ponteiro de acesso no início do arquivo. Uso:

```
rewind();
```

As funções `fflush()` (descarrega o stream para o arquivo em disco) e `fflushall()` (descarrega o conteúdo de todos os arquivos abertos para os seu respectivos arquivos em disco) controlam os buffers em memória permitindo assegurar a integridade dos dados gravados em arquivo para o caso do programa ser terminado por alguma razão exepcional. Uso:

```
fflush( arquivo );
fflushall()
```

A função `feof( arquivo )` indica, com um valor retornado não nulo, que o ponteiro do arquivo atingiu o final.

```
feof( arquivo );
```

9. Estruturas

Estruturas (struct) são ferramentas da linguagem C que permitem que o programador defina tipos de variáveis que são o agrupamento de variáveis (como nos vetores), mas com diferentes tipos.

Por exemplo, suponha que um determinado programa necessite em sua estrutura de dados de uma entidade para armazenar o nome (um vetor de caracteres), número do departamento (um inteiro) e o salário (um número em ponto flutuante) de um funcionário. São dados de tipos diferentes e poderia ser necessário criar vetores com estas estruturas.

Para tal a linguagem C definiria uma estrutura como a abaixo:

```
struct fichaTipo /* Definicao da estrutura. */
{
  char   nome[40];
  int    depto;
  double salario;
}

struct fichaTipo ficha; /* Declaracao da variavel. */
```

O uso da variável é bastante simples.

```
ficha.depto = 1;
ficha.salario = 12000.0;
```

Estruturas podem formar vetores:

```
struct fichaTipo fichario[1000];
```

Ponteiros para estruturas também podem ser definidos.

```
struct fichaTipo *pficha;
```

Para identificar o elemento da estrutura apontado por um ponteiro deste tipo usamos outro operador: ->

```
Departamento = pficha->depto;
```

Bibliografia

Barkakati, N. - The Waite Group's Microsoft C Bible
Howard W. Sams & Co

É uma referência completa. Própria para um guia de consulta para programadores que trabalhem com a linguagem C. Contém uma relação e descrição extensa das funções de biblioteca oferecidas pela linguagem incluindo um guia de compatibilidade com outras versões de compiladores, com a versão C do sistema UNIX e com o padrão ANSI. Descreve, com exemplos, o uso das funções. Não é muito apropriado para iniciantes ou pessoas que desejam se um livro para o aprendizado da linguagem.

Waite, M. e Prata, S. - The New Waite Group's C Primer Plus
Howard W. Sams & Co

É um livro didático da melhor qualidade. Apresenta conceitos e filosofias próprias da linguagem C de forma simples e repleta de exemplos.

Jansa, K. - Microsoft C. Dicas, Segredos e Truques.
Makrom Books

É um livro muito bom. Em português, constitui-se em um guia muito bom para iniciantes. Introduz o programador de uma forma didática inclusive apresentando dicas e os erros mais frequentemente cometidos por programadores iniciantes. Sua falha mais importante é concentrar-se principalmente na linguagem C da Microsoft sem comparar com outras implementações e dedicar espaço em excesso aos detalhes específicos ao IBM-PC.

Swan, T. – Tecle e Aprenda C