Métodos e Técnicas de Programação
.:: Estruturas de programação e tipos básicos do C ::.

Prof. Igor Peretta

2018-2

Contents

1 Visualização e captura

De maneira bastante simplificada, podemos visualizar informações sobre as variáveis dentro de nossos programas através da instrução `printf` (*PRINT Formatted data*). Esta instrução envia uma sequência de caracteres (*string*) para o buffer padrão de saída (`stdout`), ou seja, "escreve" no monitor. Quando temos dentro da string uma subsequência de pelo menos dois caracteres que começa com "%", chamada aqui de especificador de formato, a mesma é trocada pelo valor especificado fora da string na ordem de encontro das constantes e/ou variáveis.

Note que, para utilizar o comando `printf` ou `scanf` é necessário incluir a biblioteca `stdio.h` (*Stardard Input/Output*):

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

1.1 Entrada e Saída de dados

`printf("X = %d\n",x);` imprime a mensagem determinada pela *string* (entre aspas), substituindo os conjuntos de caractere que começam por "%" (chamados especificadores) pelos valores informados nos outros argumentos (nesse caso, o buffer de saída `stdout` é o monitor).

`scanf("%d",&x);` aguarda dado de entrada do usuário e armazena no endereço da variável em questão. Para determinar o endereço de memória a partir do nome da variável de interesse, basta acrescentar `&` na frente do nome.

1.2 Especificadores mais comuns

%d ou %i	É trocado por um número inteiro, base decimal (int)
%u	É trocado por um número inteiro não-negativo, base decimal (int)
%x	É trocado por un número inteiro, base hexadecimal (int)
%c	É substituído por um caractere (char)
%s	É substituído por uma sequência de caracteres (char*)
%f	Substituído por um número com ponto flutuante (float)
%lf	Substituído por um número com ponto flutuante (double)

Nota " não é um especificador, mas seu uso dentro da *string* do `printf` produzirá uma "nova linha".

1.3 Exemplos

Exemplo `printf`

```
#include <stdio.h>
int main() {
    char nome[] = "Alfazema";
    float preco = 2.19;
    int qtde = 7;
    printf("%s compra %d cadernos (R$ %.2f a unidade), "
    "gastando um total de R$ %.2f\n", nome, qtde,
    preco, preco*qtde);
    return 0;
}
```

```
Alfazema compra 7 cadernos (R$ 2.19 a unidade), gastando um total de R$ 15.33
```

Exemplo `scanf`

```
#include <stdio.h>
int main() {
    char nome[256];
    float preco;
    int qtde;
    printf("Nome: ");
    scanf("%s", nome);
    printf("Preco: ");
    scanf("%f", &preco);
    printf("Quantidade: ");
    scanf("%d", &qtde);
    printf("%s compra %d cadernos (R$ %.2f a unidade),"
    "gastando um total de R$ %.2f\n", nome, qtde,
    preco, preco*qtde);
    return 0;
}
```

```
Nome: Preco: Quantidade:  compra 32733 cadernos (R$ -0.00 a unidade),gastando um total
```

2 Recordando estruturas de programação

Seguem as estruturas de programação em C. Em geral, dentro da mesma estrutura temos comandos distintos que podem vir a ter a mesma tarefa. Mas

cada qual deles possui características que podem ser vantajosas dependendo do problema.

2.1 Sequência

Uma estrutura sequencial é uma estrutura de desvio do fluxo de controle presente em linguagens de programação que realiza um conjunto predeterminado de comandos de forma sequencial, de cima para baixo, na ordem em que foram declarados. Note que o compilador de uma linguagem estrutural não saberá o que significa um identificador (que não os reservados pela linguagem) a não ser que o mesmo tenha sido definido ou iniciado antes do seu uso efetivo. Além disso, a redefinição de um identificador antes do seu uso com a definição anterior fará com que a definição original seja perdida.

```
int numero;
numero = 109;
printf("numero = %d\n", numero);
```

Figure 1: **Código e execução:** a declaração e inicialização de `numero` **deve** ocorrer antes do seu uso efetivo

```
numero = 109
```

```
int numero = 9;
int numero_ao_quadrado;
numero_ao_quadrado = numero*numero;
printf("O quadrado de %d = %d\n", numero, numero_ao_quadrado); //
numero = 100;
```

Figure 2: **Código e execução:** a nova atribuição da variável `numero` anterior ao uso efetivo do seu conteúdo faz com que a informação para o usuário esteja errada; você conseguiria consertar?

```
O quadrado de 9 = 81
```

2.1.1 Operadores matemáticos

Operador	Caractere	Prioridade
soma	+	2
subtração	-	2
multiplicação	*	1
divisão (reais) ou quociente (inteiros)	/	1
módulo ou resto (inteiros)	%	1
para prioridade de resolução	( )	0

Quanto **menor** a prioridade, primeiro o operador será avaliado. Quando dois ou mais operadores estão na mesma prioridade, serão avaliados da esquerda para a direita.

Qual o valor da seguinte expressão?

$$(3 - 4 \cdot 5) - (5 \cdot 3^2 + 5)/2$$

```
printf("Expressao = %d\n", (3-4*5) - (5*(3*3) + 5) / 2);
```

```
Expressao = -42
```

```
int x = 129, y = 7;
printf("quociente %d, resto %d\n", x/y, x%y);
printf("quociente %d, resto %d\n", 15/4, 15%4);
printf("divisão %d\n", x/13);
printf("divisão %f\n", x/13.0);
printf("divisão %f\n", (float)x/13);
```

Figure 3: **Código e execução:** a barra de divisão entre inteiros é o quociente da divisão (também um inteiro) com outra operação para resto; a barra de divisão onde ao menos um dos operandos é um número com ponto flutuante é a divisão efetiva com resultado também em ponto flutuante

```
quociente 18, resto 3
quociente 3, resto 3
divisão 9
divisão 9.923077
divisão 9.923077
```

2.2 Decisão/Seleção

Estrutura de seleção (decisão, expressão condicional ou ainda construção condicional) é uma estrutura de desvio do fluxo de controle presente em linguagens de programação que realiza diferentes computações ou ações dependendo se a seleção (ou condição) é verdadeira ou falsa, em que a expressão é processada e transformada em um valor booleano (verdadeiro ou falso).

No caso do C, o tipo **bool** não é primitiva. Para inferência lógica, o valor 0 (todos os bits do tipo igual a 0) é considerado falso e qualquer coisa diferente é considerado verdadeiro.

2.2.1 Operadores booleanos

O resultado desses operadores é 0 (falso) ou qualquer número (verdadeiro, normalmente 1):

operador	nome
==	igual
!=	diferente
<	menor que
>	maior que
<=	menor ou igual que
>=	maior ou igual que
&&	AND, "e" lógico
||	OR, "ou" lógico
!	NEG, negação lógica

```
int x = 3, y = 5;
if(x == 4 || y == 5) printf("Um (ou ambos) iguais\n");
if(x != 4 || y != 5) printf("Um (ou ambos) diferentes\n");
if(x == 3 && y == 5) printf("Todos iguais\n");
if(x != 4 && y != 6) printf("Todos diferentes\n");
```

```
Um (ou ambos) iguais
Um (ou ambos) diferentes
Todos iguais
Todos diferentes
```

---

1. Tabelas da verdade

A	B	A && B	A B
Falso	Falso	Falso	Falso
Falso	Verdadeiro	Falso	Verdadeiro
Verdadeiro	Falso	Falso	Verdadeiro
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro

A	!A
Falso	Verdadeiro
Verdadeiro	Falso

2.2.2 Seleção simples (if) e composta (if-else)

O `if` é usado para executar um comando ou um bloco de comandos (*corpo com código*), somente se uma dada *condição* for verdadeira. Sua sintaxe:

`if (` *condição* `)` *corpo com código* `;`

A *condição* é uma expressão (em geral, booleana) que está sendo avaliada. Se a condição é verdadeira, o *corpo* é executado. Se for falsa, o *corpo* é simplesmente ignorado e o programa continua após a seleção do `if`.

As seleções onde uma alternativa pode ser executada se a condição é falsa precisam da palavra `else` para introduzir o comando ou um bloco de comandos a serem executados. Sua sintaxe fica:

`if (` *condição* `)` *corpo, se verdade* `else` *corpo, se falso* `;`

```
int x = 3;
if(x%2 == 0)
  printf("PAR\n");
else
  printf("IMPAR\n");
```

Figure 4: **Código e execução:** uso do `if` para simular o resultado do lançamento de uma moeda

```
IMPAR
```

2.2.3 Operador ternário

O operador ternário funciona como uma seleção *inline*, ou seja, pode ser inserido dentro de outros comandos ou mesmo em atribuições. Sua sintaxe:

condição `?` *expressão, se verdade* `:` *expressão, se falso* `;`

Note que o operador ternário retorna valores ou resultados de expressões, nunca blocos de comando.

```
int y = 9090;
int x = y ? 8 : 3;
printf("%s\n",(x%2)? "IMPAR" : "PAR");
```

Figure 5: **Código e execução:** dois exemplos em um para o operador ternário

```
PAR
```

2.2.4 Seleção de múltipla escolha

Note que as únicas condições tratáveis pelo `switch` são as do tipo "igual a certo valor".

```
int x = 3;
switch(x%2) {
 case 0:
   printf("PAR\n");
   break;
 case 1:
   printf("IMPAR\n");
   break;
 default:
   printf("Rever logica do codigo\n");
}
```

Figure 6: **Código e execução:** a seleção de `switch` para mais de uma possibilidade com `default`

`IMPAR`

2.3 Iteração

Uma estrutura de iteração (repetição) realiza e repete diferentes computações ou ações dependendo se uma condição é verdadeira ou falsa, condição essa que é um expressão processada e transformada em um valor booleano. Está associado a ela além da condição (também chamada expressão de controle ou condição de parada) o bloco de código: verifica-se a condição, e caso seja verdadeira, o bloco é executado. Após o final da execução do bloco, a condição é verificada novamente e, caso ela ainda seja verdadeira, o código é executado mais tantas vezes enquanto a condição seja verdadeira.

2.3.1 Pré-testada

A sintaxe do `while` é:

`while (` *condição* `)` *corpo com código* `;`

O laço do `while` simplesmente repete o *corpo com código* enquanto a *condição* for verdadeira. Se após qualquer execução do *corpo* a *condição* não for mais verdadeira, o laço se encerra e o programa continua logo depois do laço.

```
int x = 6;
while(x < 5) {
  printf("%i; ", x);
  x++;
}
```

Figure 7: **Código e execução:** uso ordinário do `while` com variável auxiliando no controle

---

```
int x = 42;
while(x < 5) {
  printf("%i; ", x);
  x++;
}
printf("\nAlgo até aqui?\n");
```

Figure 8: **Código e execução:** se a condição chegar ao `while` sendo falsa, o *corpo com código* nunca será executado

```
Algo até aqui?
```

2.3.2 Pós-testada

A sintaxe do `do-while` é:

`do` *corpo com código* `while(` *condição* `);`

Ele se comporta como o laço com `while`, exceto que a condição é avaliada depois da execução do *corpo com código*, garantindo assim ao menos uma execução do mesmo, ainda que a a condição nunca seja verdadeira.

```
int x = 6;
do {
  printf("%d; ", x);
  x++;
} while(x < 5);
```

Figure 9: **Código e execução:** uso ordinário do `do-while` com variável auxiliando no controle

```
6;
```

```
int x = 42;
do {
  printf("%d; ", x);
  x++;
} while(x < 5);
printf("\nAlgo até aqui?\n");
```

Figure 10: **Código e execução:** exemplo onde o corpo só é executado uma única vez (a condição, quando testada, é falsa)

```
42;
Algo até aqui?
```

2.3.3 Com variável de controle

O laço `for` foi projetado para iterar um certo número de vezes. Sua sintaxe é:

`for (` *inicialização* `;` *condição* `;` *incremento* `)` *corpo com código* `;`

O `for` repetirá o *corpo com código* enquanto a *condição* for verdadeira. Note que, como o `while`, se a condição chegar sendo falsa, o *corpo com código* nunca será executado.

```
int i;
for(i = 0; i < 5 ; i++) {
  printf("%d; ", i);
}
```

Figure 11: **Código e execução:** uso ordinário do `for`

```
0; 1; 2; 3; 4;
```

```
int i, j;
for(i = 0, j = 5; i <= j ; i++, j--) {
  printf("%d, %d ; ", i, j);
}
```

Figure 12: **Código e execução:** diversas variáveis podem participar dos parâmetros do `for`

```
0, 5 ; 1, 4 ; 2, 3 ;
```

```
srand(time(0));
int x = 0;
for(;;) {
  if(rand()%5 == 0)
    break;
  x++;
  printf("%d ", x);
}
printf("se nao fosse o break, o laço se repetiria infinitamente...\n");
```

Figure 13: **Código e execução:** laço infito com `for`

```
1 se nao fosse o break, o laço se repetiria infinitamente...
```

```
int i, j;
for(i = 1, j = 1; i < 7 ; i++, j*=i);
printf("%d! = %d\n",i,j);
```

Figure 14: **Código e execução:** fatorial feito apenas com parâmetros do `for`

```
7! = 5040
```

Tente você a programar o número de Fibonacci dentro do `for()`:

$$Fib(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ Fib(n-1) + Fib(n-2), & n \geq 2 \end{cases}$$

```
int i, n_1, n_2, f;
for(i = 1, n_1 = 1, n_2 = 1, f = 1; i < 7 ; i++, f = n_1 + n_2,
    n_2 = n_1, n_1 = f);
printf("fib(%d) = %d\n",i,f);
```

```
fib(7) = 21
```

3 Primitivas ou tipos básicos

A linguagem C provê especificadores para quatro tipos básicos aritméticos: `char`, `int`, `float` e `double`, assim como os modificadores `signed`, `unsigned`, `short` e `long`.

```
char c = 0xFF;
unsigned char d = 0xFF;
printf("%hhd, %hhu\n", c, d);
```

```
-1, 255
```

```
char C; //signed
unsigned char uC;
short int S; //signed
unsigned short int uS;
int I; //signed
unsigned int uI;
long int L; //signed
unsigned long int uL;
long long int LL; //signed
unsigned long long int uLL;
float F;
double D;
long double LD;
printf("C  : %d byte;  uC  : %d byte\n", sizeof(C), sizeof(uC));
printf("S  : %d bytes; uS  : %d bytes\n", sizeof(S), sizeof(uS));
printf("I  : %d bytes; uI  : %d bytes\n", sizeof(I), sizeof(uI));
printf("L  : %d bytes; uL  : %d bytes\n", sizeof(L), sizeof(uL));
printf("LL : %d bytes; uLL : %d bytes\n", sizeof(LL), sizeof(uLL));
printf("F  : %d bytes; D   : %d bytes\n", sizeof(F), sizeof(D));
printf("LD : %d bytes\n", sizeof(LD));

C  : 1 byte;  uC  : 1 byte
S  : 2 bytes; uS  : 2 bytes
I  : 4 bytes; uI  : 4 bytes
L  : 8 bytes; uL  : 8 bytes
LL : 8 bytes; uLL : 8 bytes
F  : 4 bytes; D   : 8 bytes
LD : 16 bytes

char Ci = 0x80, Cs = 0x7F; //signed
unsigned char uCi = 0x00, uCs = 0xFF;
short int Si = 0x8000, Ss = 0x7FFF; //signed
unsigned short int uSi = 0x0000, uSs = 0xFFFF;
int Ii = 0x80000000, Is = 0x7FFFFFFF; //signed
unsigned int uIi = 0x00000000, uIs = 0xFFFFFFFF;
long int Li = 0x8000000000000000, Ls = 0x7FFFFFFFFFFFFFFF; //signed
unsigned long int uLi = 0x0000000000000000, uLs = 0xFFFFFFFFFFFFFFFF; //signed;
long int LLi = 0x8000000000000000, LLs = 0x7FFFFFFFFFFFFFFF; //signed
unsigned long int uLLi = 0x0000000000000000, uLLs = 0xFFFFFFFFFFFFFFFF; //signed;
printf("C  : %hhd ~ %hhd min-max; uC  : %hhu ~ %hhu min-max\n", Ci, Cs, uCi, uCs);
```

```
printf("S  : %hd ~ %hd min-max; uS  : %hu ~ %hu min-max\n", Si, Ss, uSi, uSs);
printf("I  : %d ~ %d min-max; uI  : %u ~ %u min-max\n", Ii, Is, uIi, uIs);
printf("L  : %ld ~ %ld min-max; uL  : %lu ~ %lu min-max\n", Li, Ls, uLi, uLs);
printf("LL : %lld ~ %lld min-max; uLL : %llu ~ %llu min-max\n", LLi, LLs, uLLi, uLLs);
```

```
C  : -128 ~ 127 min-max; uC  : 0 ~ 255 min-max
S  : -32768 ~ 32767 min-max; uS  : 0 ~ 65535 min-max
I  : -2147483648 ~ 2147483647 min-max; uI  : 0 ~ 4294967295 min-max
L  : -9223372036854775808 ~ 9223372036854775807 min-max; uL  : 0 ~ 184467440737095516
LL : -9223372036854775808 ~ 9223372036854775807 min-max; uLL : 0 ~ 184467440737095516
```

3.1 Ponto flutuante

```
printf("F  : %e ~ %e min-max\n", FLT_MIN, FLT_MAX);
printf("D   : %le ~ %le min-max\n", DBL_MIN, DBL_MAX);
printf("LD  : %Le ~ %Le min-max\n", LDBL_MIN, LDBL_MAX);
```

```
F  : 1.175494e-38 ~ 3.402823e+38 min-max
D   : 2.225074e-308 ~ 1.797693e+308 min-max
LD  : 3.362103e-4932 ~ 1.189731e+4932 min-max
```

FLOAT > Especificador (formato): %f ou %F, %g ou %G, %e ou %E (notação científica) DOUBLE > Especificador (formato): %lf ou %lF, %lg ou %lG, %le ou %lE LONG DOUBLE > Especificador (formato): %Lf ou %LF, %Lg ou LG, %Le ou %LE

```
float pf = 2.0/3.0;
double pd = 2.0/3.0;
long double pld = 2.0L/3.0L;
printf("Float       %.60e\n", pf);
printf("Double      %.60le\n", pd);
printf("Long Double %.60lle\n", pld);
```

```
Float       6.666666865348815917968750000000000000000000000000000000e-01
Double      6.666666666666666296592325124947819858789443969726562500000000e-01
Long Double 6.666666666666666666847367028747584072334575466811656951904297e-01
```

3.2 Tipo BOOL (Booleano)

Em C, se a variável conter todos os seus 8 primeiros bits igual a 0, é considerada `false`; qualquer outra configuração é `true`. Isso pode gerar problemas:

```
unsigned char num = 0;
if (num == 0)
    printf("Verdadeiro %08X\n", num);
else
    printf("Falso %08X\n",num);
```

Figure 15: **Código e execução:** problema de não se ter um tipo booleano, se não houver respeito pelo tamanho máximo armazenado pelo tipo, poderemos ter distorções do desejado (256 dec = 1 0 0 0 0 0 0 0 0 bin)

```
Verdadeiro 00000000
```

Na biblioteca `<stdbool.h>` (C99) foi definido o tipo `_Bool`:

```
_Bool num = 256;
if(num) printf("Verdadeiro\n"); else printf("Falso\n");
```

Figure 16: **Código e execução:** usando `#include <stdbool.h>` e o tipo booleano definido na mesma (problema anterior resolvido)

```
Verdadeiro
```

Ainda, se necessário, podemos criar o tipo `bool` usando `typedef` e `enum`:

```
typedef enum { false = 0, true = 1 } bool;
bool cond = true;
if(cond) printf("Verdadeiro\n"); else printf("Falso\n");
```

Figure 17: **Código e execução:** criação do tipo `bool`

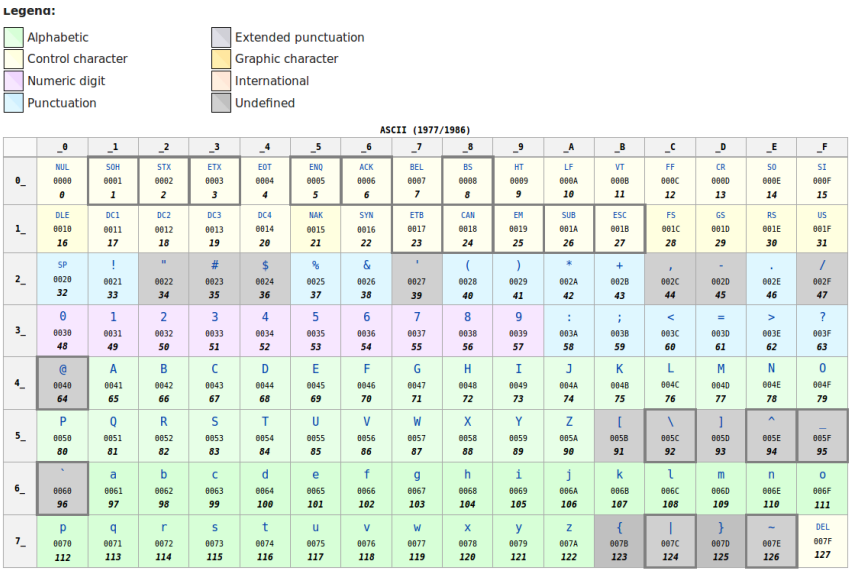
```
Verdadeiro
```

3.3 Tipo char, tratamento especial

Considerados os "átomos" da comunicação por mensagens com o usuário, o tipo `char` recebe tratamento especial da linguagem C, em especial quando formam *strings* (sequências de caracteres).

Por hora não vamos tratar de *strings*, mas vamos entender qual é a relação do `char` com o C. A codificação básica usada para traduzir os bits armazenados em caracteres é o que chamamos de código ASCII (ou ASCII

7-bits). Criado no final da década de 1960, atendia os caracteres necessários para se escrever em inglês. Somente 7 bits eram necessários para sua codificação (128 caracteres no total, dentre visíveis ou não) e sua herança afetou como a linguagem C foi criada.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000 0	SOH 0001 1	STX 0002 2	ETX 0003 3	EOT 0004 4	ENQ 0005 5	ACK 0006 6	BEL 0007 7	BS 0008 8	HT 0009 9	LF 000A 10	VT 000B 11	FF 000C 12	CR 000D 13	SO 000E 14	SI 000F 15
1_	DLE 0010 16	DC1 0011 17	DC2 0012 18	DC3 0013 19	DC4 0014 20	NAK 0015 21	SYN 0016 22	ETB 0017 23	CAN 0018 24	EM 0019 25	SUB 001A 26	ESC 001B 27	FS 001C 28	GS 001D 29	RS 001E 30	US 001F 31
2_	SP 0020 32	! 0021 33	" 0022 34	# 0023 35	$ 0024 36	% 0025 37	& 0026 38	' 0027 39	(0028 40	) 0029 41	* 002A 42	+ 002B 43	, 002C 44	- 002D 45	. 002E 46	/ 002F 47
3_	0 0030 48	1 0031 49	2 0032 50	3 0033 51	4 0034 52	5 0035 53	6 0036 54	7 0037 55	8 0038 56	9 0039 57	: 003A 58	; 003B 59	< 003C 60	= 003D 61	> 003E 62	? 003F 63
4_	@ 0040 64	A 0041 65	B 0042 66	C 0043 67	D 0044 68	E 0045 69	F 0046 70	G 0047 71	H 0048 72	I 0049 73	J 004A 74	K 004B 75	L 004C 76	M 004D 77	N 004E 78	O 004F 79
5_	P 0050 80	Q 0051 81	R 0052 82	S 0053 83	T 0054 84	U 0055 85	V 0056 86	W 0057 87	X 0058 88	Y 0059 89	Z 005A 90	[005B 91	\ 005C 92	] 005D 93	^ 005E 94	_ 005F 95
6_	` 0060 96	a 0061 97	b 0062 98	c 0063 99	d 0064 100	e 0065 101	f 0066 102	g 0067 103	h 0068 104	i 0069 105	j 006A 106	k 006B 107	l 006C 108	m 006D 109	n 006E 110	o 006F 111
7_	p 0070 112	q 0071 113	r 0072 114	s 0073 115	t 0074 116	u 0075 117	v 0076 118	w 0077 119	x 0078 120	y 0079 121	z 007A 122	{ 007B 123	| 007C 124	} 007D 125	~ 007E 126	DEL 007F 127

```
printf("%c %c %c\n",65,48,97);
```

Figure 18: **Código e execução:** imprimindo caracteres pelo código ASCII em hex

```
A 0 a
```

```
int i;
for(i = 32; i < 127; i++) {
  printf("%4c : %3hhu %c", i, i,
  ((i%5 == 0)? '\n' : ' ')); }
```

Figure 19: **Código e execução:** impressão da tabela ASCII 7bits

```
:  32     ! :  33     " :  34     # :  35
$ :  36     % :  37     & :  38     ' :  39     ( :  40
) :  41     * :  42     + :  43     , :  44     - :  45
. :  46     / :  47     0 :  48     1 :  49     2 :  50
```

```
3 :  51     4 :  52     5 :  53     6 :  54     7 :  55
8 :  56     9 :  57     : :  58     ; :  59     < :  60
= :  61     > :  62     ? :  63     @ :  64     A :  65
B :  66     C :  67     D :  68     E :  69     F :  70
G :  71     H :  72     I :  73     J :  74     K :  75
L :  76     M :  77     N :  78     O :  79     P :  80
Q :  81     R :  82     S :  83     T :  84     U :  85
V :  86     W :  87     X :  88     Y :  89     Z :  90
[ :  91     \ :  92     ] :  93     ^ :  94     _ :  95
‘ :  96     a :  97     b :  98     c :  99     d : 100
e : 101     f : 102     g : 103     h : 104     i : 105
j : 106     k : 107     l : 108     m : 109     n : 110
o : 111     p : 112     q : 113     r : 114     s : 115
t : 116     u : 117     v : 118     w : 119     x : 120
y : 121     z : 122     { : 123     | : 124     } : 125
~ : 126
```

O termo ASCII extendido foi criado para se referir ao código ASCII de 8 bits (ou mais). Novos 128 caracteres foram acrescentados, muitos para suportar desenhos na tela (interface textual) e outros para suportar caracteres em outros idiomas (como o "ç" ou o "ñ"). Diversas versões surgiram (IBM code page 37, PESCII, ISO 8859), cada qual atendendo um subgrupo de interesse. As codificações mais utilizadas no Brasil foram a Windows-1252 e a ISO-8859-1 (encontradas até hoje em diversos arquivos). Normalmente, o C não suporta a visualização de tais caracteres, já que realiza a promoção automática de toda constante `char` com valor acima de 127 para o tipo `int`.

Nos dias de hoje, quando falamos de codificação de caracteres, uma das mais versáteis é a **UTF-8**, uma codificação de tamanho de bits variável projetada para representar qualquer caractere de qualquer idioma (detalhes: `https://pt.wikipedia.org/wiki/UTF-8`).

Mas a norma ANSI-C não ficou isenta nessa discussão e foi criada a ideia de "locales" para gerenciar a codificação de caracteres. Hoje podemos acentuar mensagens e receber *strings* com caracteres diversos com a inclusão de uma biblioteca `<locale.h>` e a configuração da codificação desejada. A mais versátil hoje em dia é a

```
printf("A ’cigüeña’ é um pássaro %.2f%% para aclamação!  \n", 94.5f);
```

Figure 20: **Código e execução:** diacríticos e outros caracteres são permitidos em *strings* no compilador C da GNU (gcc) na versão 4.8 e acima, código

compilado no Linux

```
A 'cigüeña' é um pássaro 94.50% para aclamação!
```

```
setlocale(LC_ALL,"Portuguese");
printf("A 'cigüeña' é um pássaro %.2f%% para aclamação!\n", 94.5f);
```

Figure 21: **Código e execução:** estratégia possível para acentuação (não ideal); usar `#include <locale.h>`

```
A 'cigüeña' é um pássaro 94.50% para aclamação!
```

A biblioteca `<wchar.h>` seria uma opção, com as *strings* definidas com o auxílio do `L` e funções específicas (ex. `wprintf(L"exemplo");`), mas todo o paradigma de visualização de dados teria que ser alterado.

3.4 String (definição simples)

De maneira simplificada, uma **string** é uma coleção de `char` a partir de um certo endereço de memória (reconhecível pelo identificador da variável) que encerra quando localiza o caractere `'\0'`. A **string** pode ser definida de maneira simples com a informação do número máximo de caracteres que ela aceita.

```
char str[16] = "Exemplo";
int i, len = 0;
for(i = 0; i < 16; i++)
    printf("char_%02d: %c (ascii %hhd)\n", i, str[i], str[i]);
while (str[len])
    len++;
printf("Tamanho efetivo: %d caracteres.\n", len);
```

Figure 22: **Código e execução:** string, definição e uso efetivo

```
char_00: E (ascii 69)
char_01: x (ascii 120)
char_02: e (ascii 101)
char_03: m (ascii 109)
char_04: p (ascii 112)
char_05: l (ascii 108)
```

```
char_06: o (ascii 111)
char_07:  (ascii 0)
char_08:  (ascii 0)
char_09:  (ascii 0)
char_10:  (ascii 0)
char_11:  (ascii 0)
char_12:  (ascii 0)
char_13:  (ascii 0)
char_14:  (ascii 0)
char_15:  (ascii 0)
Tamanho efetivo: 7 caracteres.
```