Strings

Video desta aula

Uma string é uma sequência de caracteres que permite representar nomes, endereços e outras informações textuais.

Declaração

Em C, strings são implementadas como vetores de caracteres (tipo `char`) terminados pelo caractere especial `'\0'` (caractere cujo código numérico é zero). Esse caractere terminal é considerado no tamanho do vetor. As aspas duplas ("...") são usadas para declarar strings constantes.

Exemplos:

```
// declara string variável com até 99 caracteres (mais o \0)
char nome[100] ;

// declara string constante (com o \0 no final)
char *profissao = "estudante" ;

// declara ponteiro para uma string (não aloca espaço para ela)
char *endereco ;

// ou
char endereco[] ;

// ponteiro aponta para string constante
endereco = "Rua da Batata, 1000" ;
```

Deve-se observar que uma string é um **vetor de caracteres**, portanto as duas declarações abaixo são equivalentes:

```
char codigo[]  = "XT07A" ;
char codigo[]  = { 'X', 'T', '0', '7', 'A', '\0' } ;
```

A visão da string como vetor permite o acesso aos seus caracteres individuais. O código abaixo converte uma string em maiúsculas, usando os códigos ASCII dos caracteres:

```
i = 0 ;
while (str[i] != '\0')                // varre todos os caracteres
{
  if (str[i] >= 'a' && str[i] <= 'z') // se for letra minúscula
    str[i] -= 32 ;                    // converte em letra maiúscula
  i++ ;
}
```

```
// ou, usando strlen()
for (i=0; i < strlen (str); i++)      // varre todos os caracteres
  if (str[i] >= 'a' && str[i] <= 'z') // se for letra minúscula
    str[i] -= 32 ;                    // converte em letra maiúscula
```

Leitura e escrita

A escrita de strings pode ser feita com `printf` (usando o formato `%s` ou `%NNNs`), `puts` ou ainda `putchar` (para escrever caractere por caractere):

```
char nome[] = "Homer Simpson" ;

printf ("Nome: %s\n", nome) ;         // escrita com printf

puts (nome) ;                         // escrita com puts

for (i = 0; i < strlen(nome); i++)    // escrita com putchar
  putchar (nome[i]) ;
```

Por sua vez, a leitura de strings pode ser feita usando a função `scanf`:

```
#define SIZE 100

char nome[SIZE+1] ;  // não esquecer do '\0' no final da string

printf ("Digite seu nome: ") ;

// lê até encontrar espaço, tabulação, nova linha ou fim de arquivo
scanf ("%s", nome) ;

// idem, no máximo 20 caracteres
scanf ("%20s", nome) ;

// lê somente letras e dígitos (até encontrar outro caractere)
scanf("%[A-Za-z0-9]", nome);

// lê até encontrar um fim de linha (\n), ou seja
// lê enquanto não encontrar um caractere '\n'
scanf("%[^\n]", nome);
getchar() ;              // para ler o "\n" no fim da linha
```

Observe que a leitura de uma string deve ser feita em uma variável com **espaço suficiente** para recebê-la (incluindo o '\0'), para não gerar um estouro de buffer (buffer overflow).

Pode-se também usar a função `fgets`:

```
// lê da entrada padrão até encontrar \n ou SIZE caracteres
fgets (nome, SIZE, stdin) ;

// a string lida por fgets pode incluir o \n do fim de linha,
// se ele foi encontrado; ele pode ser retirado assim:
nome[strcspn (nome, "\n")] = '\0' ;
```

Para mais informações sobre as funções acima, deve ser consultada a respectiva página de manual Unix.

Existe uma função de leitura `gets()` que não limita o número de bytes lidos e pode provocar **estouro de buffer**, por isso **não deve ser usada**! Use a função `fgets()` em seu lugar.

Manipulação

A manipulação de strings é geralmente efetuada através de funções disponíveis na biblioteca padrão C, que podem ser acessadas através dos arquivos de cabeçalho `string.h` e `strings.h`.

Algumas das funções mais usuais são:

função	**operação realizada**
`int strlen (s)`	informa o número de caracteres da string s (sem considerar o '\0' no final)
`char * strcpy (b, a)`	copia a string a no local indicado por b; a área de memória de destino deve ter sido previamente alocada (como variável normal ou dinâmica)
`char * strdup (s)`	Aloca uma área de memória dinâmica, copia a string s nela e devolve um ponteiro para a mesma
`char * strncpy (b, a, n)`	Copia n caracteres da string a no local indicado por b
`int strcmp (a, b)`	Compara as duas strings indicadas, retornando 0 se forem iguais, um valor negativo se a<b e um valor positivo se a>b, considerando a ordem alfabética
`int strncmp (a, b ,n)`	Idem, mas só considera os n primeiros caracteres
`char * strcat (a, b)`	Concatena a string b ao final da string a (deve haver espaço disponível previamente alocado)
`char * strncat (a, b, n)`	Idem, mas só concatena os n primeiros caracteres
`char * strchr (s, c)`	Retorna um ponteiro para a primeira ocorrência do caractere c na string s, ou NULL se não encontrar
`char * strrchr (s, c)`	Idem, mas retorna um ponteiro para a **última** ocorrência do caractere

Várias outras funções para manipulação de strings estão disponíveis na página de manual (comando `man string`);

Exercícios

Escrever programas em C para:

1. Ler uma string da entrada padrão e escrevê-la na saída padrão ao contrário (do final para o início), de forma similar ao comando `rev` do *shell* UNIX.
2. Calcular o tamanho de uma string (sem usar `strlen`).
3. Converter as letras de uma string em minúsculas (dica: estude a estrutura da tabela ASCII antes de implementar).
4. Ler linhas da entrada padrão e escrevê-las na saída padrão em ordem alfabética crescente, de forma similar ao comando `sort` do *shell* UNIX.
5. Remover de uma string os caracteres que não sejam letras, números ou espaço, sem usar string auxiliar.
6. Remover de uma string caracteres repetidos em sequência (rr, ss, ee, etc), sem usar string auxiliar.
7. Colocar entre colchetes ([ ]) os caracteres de uma string que não sejam letras, números ou espaço; as alterações devem ser feitas na própria string, sem usar string auxiliar.
8. Escrever uma função `int busca(agulha, palheiro)`, que busca a string `agulha` dentro da string `palheiro`, sem usar funções prontas da biblioteca C. A função deve retornar o índice onde `agulha` começa em `palheiro`, -1 se não for encontrada ou -2 em caso de erro (uma ou ambas as strings são nulas).
9. Escrever sua própria versão das funções de manipulação de strings `strlen`, `strcpy` e `strcat`. Depois, comparar o desempenho de sua implementação em relação às funções originais da LibC (sugestão: meça o tempo necessário para ativar cada função um milhão de vezes).
10. Escrever uma função `palindromo(s)` que testa palíndromos: ela recebe uma string `s` de caracteres sem acentos e retorna 1 se a string é um palíndromo ou 0 senão. Acentos, espaços em branco e maiúsculas/minúsculas devem ser ignorados. Exemplos de palíndromos:
    - A cara rajada da jararaca
    - O poeta ama até o pó
    - Socorram-me, subi no ônibus em Marrocos!

Codificação de caracteres

Video desta aula

Internamente, um computador só armazena e processa bytes, números inteiros entre 0 e 255. Não é possível armazenar diretamente textos, imagens, sons ou qualquer outra informação que não sejam bytes.

Para armazenar informações mais complexas que os bytes, é necessário **codificar** as mesmas, ou seja, transformá-las em sequências de bytes. Esta página discute as técnicas usadas para transformar as **letras e símbolos de um texto** em bytes, para poder armazená-lo e tratá-lo em um computador.

A figura a seguir mostra as etapas do tratamento de uma letra A pelo computador: o hardware do teclado é responsável por converter a letra digitada em byte(s) na memória do computador; em seguida, o hardware do terminal converte esses bytes em uma representação gráfica na tela.

Algumas definições

- **Caractere**: é um símbolo da linguagem (letra, dígito ou sinal). Exemplos: `A i ç ë ¥ β ɣ 诶 ✂`
- **Conjunto de caracteres** (*charset*): é o conjunto de todos os caracteres suportados por um sistema ou por um padrão de codificação. Exemplo: `A-Z, a-z, 0-9, ! @ # $ % * ( ) ~ _ - + = { } [ ] | \ / < . ,`
- **Codificação** (*encoding*): é a tradução entre os caracteres e seus respectivos valores numéricos em bytes. Exemplo, A → 65.

Existem diversas codificações de caracteres; a seguir serão apresentadas as mais usuais.

A codificação ASCII

A codificação de caracteres mais antiga ainda em amplo uso é a ASCII (*American Standard Code for Information Interchange*), criada nos anos 1960 a partir de códigos de telegrafia. Sua última atualização ocorreu em 1986, mesmo assim é considerada uma codificação universal.

Praticamente **TODOS** os sistemas computacionais suportam ASCII !

A codificação ASCII abrange o conjunto de caracteres da língua inglesa, sinais gráficos e alguns caracteres de controle (nova linha, tabulação, etc), num total de 128 caracteres. Cada caractere é codificado em um byte, mas ocupa somente 7 bits; o oitavo bit de cada byte era antigamente usado para verificação de paridade.

A codificação ASCII é definida através da famosa **Tabela ASCII**, que é dividida em duas partes:

- 0 - 31 e 127: caracteres de controle (*newline*, *form feed*, *tab*, etc), que dependem do terminal utilizado.
- 32 - 126: caracteres imprimíveis (A, B, C, ...), independentes de terminal.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

A codificação ASCII ainda é amplamente usada para codificação de textos puros em inglês, como códigos-fonte de programas, páginas HTML, arquivos de configuração, etc.

Code pages

A codificação ASCII não suporta caracteres acentuados (á é ñ ë) ou caracteres específicos de outras línguas, como ç ¥ β ɣ 诶 etc. Pode-se usar o oitavo bit de cada byte para associar caracteres aos valores acima de 127. Isso levou à criação de diversas **tabelas ASCII estendidas** para definir os símbolos de 128 a 255. Cada codificação é denominada uma **code page**; algumas das mais conhecidas são:

- CP-437: (code page 437), codificação usada nos primeiros PCs, com caracteres acentuados e gráficos simples (▒ ╝ ╦ ┼).
- Windows-1252: codificação usada em sistemas Windows mais antigos; é parte de um conjunto de codificações para diversas linguagens chamado Windows code pages.
- KOI8-R: cirílico russo (Код Обмена Информацией, 8 бит).
- BraSCII: português brasileiro, usada nos anos 1980-90.
- ISO-8859 : codificações da ISO para diversas línguas.

Codificações ISO-8859

Nos anos 1980, para para tentar organizar a profusão de codepages ASCII estendidas, a ISO propôs o conjunto

de padrões ISO-8859, que define codificações ASCII estendidas para diversas linguagens, como por exemplo:

- ISO-8859-1: Europa ocidental (francês, espanhol, italiano, alemão, etc)
- ISO-8859-15: revisão do ISO-8859-1, contendo o € e outros símbolos
- ISO-8859-2: Europa central (Bósnio, Polonês, Croata, etc)
- ISO-8859-6: árabe simplificado
- ISO-8859-7: grego

As codificações ISO-8859 se tornaram um padrão mundial e ainda são amplamente usadas em muitos sistemas, sendo gradualmente substituída pela codificação Unicode em sistemas mais recentes. Elas são compatíveis com a codificação ASCII, pois representam cada caractere com **somente um byte** e respeitam as definições ASCII dos caracteres de 0 a 127.

Programas que manipulem caracteres ISO devem usar variáveis `unsigned char`, para poder representar valores de o a 255.

Caracteres multibyte

O maior problema das codificações ISO-8859 é o uso de somente **um byte por caractere**, o que limita cada *code page* a 256 caracteres. Essa limitação impede a representação completa de línguas asiáticas e do árabe, por exemplo.

Para representar conjuntos com mais de 256 caracteres é necessário usar **caracteres multibyte**, ou seja, com mais de um byte. Por exemplo, se usarmos 2 bytes por caractere é possível representar até 2^{16} = 65.536 caracteres distintos na mesma tabela, sem precisar trocar de *code page*.

Vários padrões de codificação multibyte foram propostos, como:

- ISO-2022-CJK: chinês, japonês, coreano
- Shift-JIS: japonês (Windows)
- GB 18030: padrão oficial chinês
- Big5: chinês tradicional (Taiwan)
- **Unicode**

Alguns destes padrões definem todos os caracteres com uma quantidade fixa de bits (16 ou 32), enquanto outros definem caracteres com tamanho variável (8, 16 ou 32 bits).

Unicode

O padrão Unicode define um imenso conjunto de caracteres e os modos de codificação dos mesmos. Atualmente, existem cerca de 140.000 caracteres definidos em Unicode, para todas as línguas conhecidas (inclusive Klingon!), além de símbolos e emojis. Eles ocupam pouco mais de 10% da capacidade total desse padrão.

Em Unicode, cada caractere possui um código numérico único, chamado *code point*, que pode ser representado de diversas formas. Por exemplo, o *code point* do emoji vale 128540 ($1F61C_h$) e pode ser representado como:

- `U+1f61c` : em hexadecimal
- `&#x1f61c;` ou `&#128540;` : em páginas Web (decimal ou hexadecimal)
- `\u1f61c` : em algumas linguagens de programação

Caracteres em Unicode podem ser codificados (representados em bytes) de diversas formas:

- UTF-8: *8-bit Unicode Transformation Format*, usa de 1 a 4 bytes por caractere. É usado no Linux, Windows 10 e outros sistemas recentes.
- UTF-16: usa 2 ou 4 bytes por caractere; muito usado nas APIs dos sistemas Windows, em Java, Python e PHP.
- UTF-32: usa sempre 4 bytes por caractere. É pouco usado na prática.

A codificação UTF-8

UTF-8 é certamente a codificação multibyte mais utilizada hoje em dia, por ser plenamente compatível com a codificação ASCII e por ser econômica em espaço.

Em UTF-8, cada caractere Unicode é codificado usando de 1 a 4 bytes, conforme o número de bits de seu *code point*:

Caractere	Code point	Em binário	bits
A	41_h (65)	100 0001	7
ç	$E7_h$	1110 0111	8
©	$C2A9_h$	1100 0010 1010 1001	16
🙀	$1F600_h$	1 1111 0110 0000 0000	17

A regra de codificação de cada caractere é escolhida conforme o número de bits usados pelo seu *code point*:

# de bits do caractere	formato codificado	bytes	Uso
até 7 bits	`0xxx-xxxx`	1	tabela ASCII
8-11 bits	`110x-xxxx 10xx-xxxx`	2	caracteres estendidos
12-16 bits	`1110-xxxx 10xx-xxxx 10xx-xxxx`	3	caracteres estendidos
17-21 bits	`1111-0xxx 10xx-xxxx 10xx-xxxx 10xx-xxxx`	4	caracteres estendidos

Pode-se observar que bytes os bytes iniciando em `0…` sempre representam caracteres ASCII. Então, um texto codificado em UTF-8 contendo somente caracteres com códigos entre 0 e 127 equivale a um texto codificado em ASCII padrão.

Além disso, todos os bytes iniciando em `10…` são bytes de continuação da codificação de um caractere multibyte. Isso significa que é fácil localizar o início de cada caractere no texto, mesmo na presença de erros.

Dica: pode-se visualizar o conteúdo de um arquivo em hexadecimal ou binário usando o comando `xxd`.

O mecanismo de codificação de *code points* Unicode em UTF-8 funciona da seguinte forma:

1. Dado um caractere, verifica-se quantos bits são necessários para armazenar seu código em UTF-8. Por exemplo, o caractere 🙀 (*code point* `U+1f600`) precisa de 17 bits: 1F600 → 1 1111 0110 0000 0000.
2. Para 17 bits é necessário codificar usando 4 bytes (faixa 17-21 bits)
3. Distribui-se os bits do código numérico do caractere nos espaços disponíveis:

```
Code point (hex)                    1    f    6    0    0
Code point (bin)                    0001 1111 0110 0000 0000
Encoding format       1111-0xxx 10xx-xxxx 10xx-xxxx 10xx-xxxx
Code point (bin)            000   01 1111   01 1000   00 0000
Encoded character     1111 0000 1001 1111 1001 1000 1000 0000
                      f    0    9    f    9    8    8    0
```

4. Com isso, a codificação de U+1f600 em UTF-8 resulta nos 4 bytes `f0 9f 98 80`.

5. A decodificação (de UTF-8 para o *code point*) se efetua fazendo o caminho inverso.

Alguns arquivos codificados em UTF-* podem apresentar em seus dois primeiros bytes um valor chamado BOM (*Byte Order Mark*), que define em que ordem os bytes de cada caractere devem ser considerados: *big endian* ou *little endian*. o campo BOM não é necessário em UTF-8, mas pode estar presente às vezes:

Bytes	**Encoding Form**
`00 00 FE FF`	UTF-32, big-endian
`FF FE 00 00`	UTF-32, little-endian
`FE FF`	UTF-16, big-endian
`FF FE`	UTF-16, little-endian
`EF BB BF`	UTF-8

Dica: no Linux, pode-se digitar caracteres Unicode usando as seguintes teclas: `ctrl` + `shift` + u, código hexadecimal, `enter`

Comparando as codificações

O quadro a seguir compara a representação da string “equação” usando algumas das codificações estudadas. No caso da codificação ASCII, considera-se a letra sem acento ou cedilha; a representação UTF-16 usa dois bytes de cabeçalho BOM (*Byte Order Mark*).

Codificação	**BOM**	**e**	**q**	**u**	**a**	**ç**	**ã**	**o**	
ASCII		65	71	75	61	63	61	6f	00
ISO-8859-1		65	71	75	61	e7	e3	6f	00
UTF-8		65	71	75	61	c3 a7	c3 a3	6f	00
UTF-16 (be)	fe ff	00 65	00 71	00 75	00 61	00 e7	00 e3	00 6f	00 00

Conversão de codificações

O comando `file` do UNIX informa o tipo de codificação usado em um arquivo de texto:

```
$ file exemplo.*
exemplo.c:    C source, ISO-8859 text
exemplo.html: HTML document, ASCII text
exemplo.txt:  UTF-8 Unicode text
```

A conversão de codificação de um arquivo de texto pode ser feita com utilitários específicos, como o `iconv` no Linux:

```
iconv -f ISO-8859-15 -t UTF-8 < input.txt > output.txt
```

Além disso, os editores de texto geralmente permitem escolher a codificação ao salvar o arquivo. Por exemplo, no VI:

```
:set fileencoding=utf8
:w myfilename
```

Mais informações

Sobre codificações e Unicode:

- https://www.cl.cam.ac.uk/~mgk25/unicode.html
- https://www.ime.usp.br/~pf/algoritmos/apend/unicode.html
- https://www.cprogramming.com/tutorial/unicode.html
- https://begriffs.com/posts/2019-05-23-unicode-icu.html
- http://kunststube.net/encoding/

Exercícios

1. Use o programas `file` e `iconv` para fazer as seguintes conversões:
    1. o arquivo exemplo.c para UTF-8
    2. o arquivo exemplo.c para ASCII

Strings multibyte

Video desta aula

Locale

Em computação, o termo locale designa um conjunto de parâmetros que definem a “localização”, ou seja as preferências de linguagem de um sistema, como a língua usada, a codificação de caracteres e formatos de informações usuais (números, data/hora, moeda, etc).

No sistema Linux, por exemplo, o comando `locale` informa os parâmetros locais em uso. No exemplo abaixo o sistema usa como língua o Português brasileiro e como codificação padrão o UTF-8:

```
$ locale
LANG=pt_BR.utf8
LANGUAGE=pt_BR.utf8
LC_CTYPE="pt_BR.utf8"
LC_NUMERIC="pt_BR.utf8"
LC_TIME="pt_BR.utf8"
...
```

O parâmetro mais importante para um programa em C é LC_CTYPE (*character type*), pois ele define o conjunto de caracteres e afeta o comportamento de funções como `printf` e `scanf`.

Um programa em C pode consultar ou modificar os parâmetros de *locale* do SO através da função `setlocale()`:

locale.c

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

int main()
{
  char *locale ;

  // obtém o LC_CTYPE atual do programa
  locale = setlocale (LC_CTYPE, NULL) ;
  printf ("Current locale is %s\n", locale) ;

  // ajusta o LC_TYPE do programa para o default do SO
  locale = setlocale (LC_CTYPE, "") ;
  if (locale)
    printf ("Current locale is %s\n", locale) ;
  else
    fprintf(stderr, "Can't set the specified locale\n") ;

  // ajusta o LC_TYPE do programa para "pt_BR.iso88591"
  locale = setlocale (LC_CTYPE, "pt_BR.iso88591") ;
  if (locale)
    printf ("Current locale is %s\n", locale) ;
  else
```

```
    fprintf(stderr, "Can't set the specified locale\n") ;
}
```

Se a função `setlocale()` for chamada com uma string vazia (“”), a configuração de localização do programa é feita com base nas variáveis de ambiente providas pelo sistema operacional. Então é recomendável **sempre chamar essa função** no início de programas que manipulam caracteres não-ASCII.

Caracteres e strings em C

Caracteres ASCII

A linguagem C manipula caracteres codificados em ASCII sem dificuldade, usando variáveis do tipo `char`. Em ASCII, strings são meros vetores de caracteres terminados com um caractere nulo (`\0`).

Caracteres ISO-8859

Como as codificações ISO-8859-* usam apenas um byte por caractere, programas em C podem manipular caracteres em ISO sem dificuldade, usando variáveis do tipo `unsigned char` (para representar caracteres de 0 a 255).

Além disso, deve-se definir o *locale* do programa para garantir o funcionamento correto de funções como `toupper()`, `isalpha()`, etc. com os caracteres estendidos:

```
char *locale ;

locale = setlocale (LC_CTYPE, "pt_BR.ISO-8859-1") ;
```

Obviamente, o locale ISO-8859-1 deve estar disponível no sistema operacional (essa informação pode ser consultada com o comando `locale -a`). Além disso, se houver escrita na tela, o **terminal deve estar configurado** para usar caracteres ISO.

Caracteres UTF-8

As coisas mudam para as codificações multibyte, pois tipo `char` é insuficiente para armazenar caracteres em codificações multibyte. Por isso, alguns cuidados devem ser tomados ao definir e usar strings em UTF-8, por exemplo:

- Ao alocar memória para as strings, lembre-se que alguns caracteres podem ocupar mais de um byte.
- As funções de entrada/saída formatadas, como `printf`, `scanf` suas variantes, suportam UTF-8 sem modificações, basta executar `setlocale()` no início.
- O índice não corresponde mais necessariamente à posição de cada caractere na string. Por exemplo, `nome[3]` não corresponde necessariamente ao quarto caractere da string `nome`, caso ela esteja codificada em UTF-8.
- A função `strlen` sempre informa o **número de bytes** da string; para obter o número de caracteres, deve-se usar a função `mbstowcs` (*multi-byte-string-to-wide-character-string*), que retorna o número de caracteres da string.

Como regra geral, deve-se sempre consultar o manual para verificar se a função desejada funciona com strings multibyte.

O código abaixo apresenta um exemplo de programa que manipula strings em UTF-8:

char-utf8.c

```c
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stdlib.h>

int main()
{
  char *frase = "Olá ɣ 诶  ˝ ;

  // ajusta a localização de acordo com o SO
  setlocale (LC_ALL, "") ;

  // conteúdos da string
  printf ("Frase           : %s\n",  frase) ;

  // número de caracteres usando strlen()
  printf ("strlen (frase)  : %ld\n", strlen(frase)) ;

  // número de caracteres usando mbstowcs()
  printf ("mbstowcs (frase): %ld\n", mbstowcs(NULL, frase, 0)) ;
}
```

Caracteres largos

O padrão C 90 introduziu o conceito de caracteres “largos”, ou seja, com mais de um byte. Ao contrário dos caracteres *multibyte*, os caracteres largos têm **sempre o mesmo tamanho**, geralmente 2 ou 4 bytes (depende da plataforma). Em Linux, um caractere largo ocupa 4 bytes e pode representar qualquer *code point* do padrão Unicode.

Caracteres largos e strings largas são definidos pelo tipo `wchar_t`:

char-wide.c

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main ()
{
  wchar_t c ;         // um caractere largo
  wchar_t *s ;        // ponteiro para uma string larga

  c = L'a' ;          // caractere constante largo
  s = L"equação" ;    // string constante larga

  // ajusta a localização de acordo com o SO
  setlocale(LC_ALL,"");
```

```
  // escrita de caracteres largos
  printf ("O caractere [%lc] tem %ld bytes\n", c, sizeof (c)) ;

  // escrita de strings largas
  printf ("A string [%ls] tem %ld caracteres\n", s, wcslen (s)) ;
}
```

Várias funções são definidas pelo padrão POSIX para manipular caracteres e strings largas. Elas geralmente estão presentes na LibC.

Algumas diferenças entre strings largas e strings multibyte UTF-8:

- Uma string larga é terminada pelo caractere largo nulo `L'\0'`, enquanto string comuns e UTF-8 são terminadas por um caractere nulo com um byte `'\0'`.
- Em uma string larga, o número de campos equivale ao número de caracteres, por isso `s[10]` sempre é o 11º caractere da string, independente do conteúdo, o que não ocorre em UTF-8.
- Uma string larga ocupa mais memória que uma string multibyte, pois todos os seus caracteres ocupam o mesmo número de bytes independente de seu *code point*.

Caracteres largos são empregados na implementação de aplicações que manipulam muitas strings, como editores de texto. O ambiente Python usa caracteres largos para armazenar strings.

O código abaixo exemplifica compara algumas operações usando strings largas e strings UTF-8. Ele gera diversos avisos (*warnings*) ao compilar, devidos às chamadas de funções inadequadas:

wide-utf8.c

```
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stdlib.h>
#include <wchar.h>

int main()
{
  int i ;
  char    *frase1 =  "Olá ɣ 诶  ″ ;
  wchar_t *frase2 = L"Olá ɣ 诶  ″ ;

  // ajusta a localização de acordo com o SO
  setlocale(LC_ALL,"");

  // conteúdos das strings
  printf ("Frase 1          : %s\n",  frase1) ;
  printf ("Frase 2          : %ls\n", frase2) ;

  // tamanho em bytes
  printf ("sizeof (char)    : %ld\n", sizeof(char)) ;
  printf ("sizeof (wchar_t) : %ld\n", sizeof(wchar_t)) ;

  // número de caracteres usando strlen()
  printf ("strlen (frase1)  : %ld\n", strlen(frase1)) ;
  printf ("strlen (frase2)  : %ld\n", strlen(frase2)) ;  // incorreto

  // número de caracteres usando wcslen()
```

```
  printf ("wcslen (frase1)  : %ld\n", wcslen(frase1)) ;  // incorreto
  printf ("wcslen (frase2)  : %ld\n", wcslen(frase2)) ;

  // número de caracteres usando mbstowcs()
  printf ("mbstowcs (frase1): %ld\n", mbstowcs(NULL, frase1, 0)) ;
  printf ("mbstowcs (frase2): %ld\n", mbstowcs(NULL, frase2, 0)) ; //
incorreto

  // percurso por índice, string estreita (narrow)
  printf ("Frase1: ") ;
  for (i=0; i<strlen(frase1); i++)
    printf ("[%c] ", frase1[i]) ;
  printf ("\n") ;

  // percurso por índice, string larga (wide)
  printf ("Frase2: ") ;
  for (i=0; i<wcslen(frase2); i++)
    printf ("[%lc] ", frase2[i]) ;
  printf ("\n") ;
}
```

Ao executar, este programa gera:

```
Current locale is pt_BR.UTF-8

Frase 1          : Olá ɣ 诶
Frase 2          : Olá ɣ 诶

sizeof (char)    : 1
sizeof (wchar_t) : 4

strlen (frase1)  : 16
strlen (frase2)  : 1     // incorreto

wcslen (frase1)  : 4     // incorreto
wcslen (frase2)  : 9

mbstowcs (frase1): 9
mbstowcs (frase2): 1     // incorreto

Frase1: [O] [l] [�] [�] [ ] [�] [�] [ ] [�] [�] [�] [ ] [�] [�] [�] [�]
Frase2: [O] [l] [á] [ ] [ɣ] [ ] [诶] [ ] [ ]
```

Mais informações

Bibliotecas para UTF-8:

- https://en.wikipedia.org/wiki/International_Components_for_Unicode
- https://developer.gnome.org/glib/2.62/glib-Unicode-Manipulation.html
- https://juliastrings.github.io/utf8proc/

Exercícios

1. escreva um programa em C para converter um texto em ISO-8859-1 para ASCII, substituindo as letras acentuadas e cedilha por seus equivalentes sem acento.
2. escreva um programa em C para converter um texto em ISO-8859-1 para UTF-8.
3. escreva uma função `char* utf8strn(char* s, int n)` que devolva um ponteiro para a posição do n-ésimo **caractere** da string `s`, que está codificada em UTF-8.
4. escreva um programa C que imprima as tabelas ASCII e ISO-8859-1.

Acesso a arquivos

Video desta aula

Aqui serão descritas algumas das funções mais usuais para operações de entrada/saída em arquivos na linguagem C. A maioria das funções aqui descritas está declarada no arquivo de cabeçalho `stdio.h`:

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

Conceitos básicos

Tipos de arquivos

Um arquivo armazena uma sequência de bytes, cuja interpretação fica a cargo da aplicação. Contudo, para facilitar a manipulação de arquivos, a linguagem C considera dois tipos de arquivos:

- **arquivos de texto**: contém sequências de bytes representando caracteres de texto, separadas por caracteres de controle como `\n` ou `\r` e usando uma codificação como ASCII, ISO-8859-1 ou UTF-8. São usados para armazenar informação textual, como código-fonte, páginas Web, arquivos de configuração, etc.
- **arquivos binários**: contém sequências de bytes, cuja interpretação fica totalmente a cargo da aplicação. São usualmente empregados para armazenar imagens, vídeos, músicas, dados compactados, etc.

Streams e descritores

As operações sobre arquivos em C podem ser feitas de duas formas:

- por **streams**: acesso em um nível mais elevado, independente de sistema operacional e portanto portável. Permite entrada/saída formatada, mas pode ter um desempenho inferior ao acesso de baixo nível. *Streams* são acessadas através de variáveis do tipo `FILE*`.
- por **descritores**: acesso através dos descritores de arquivo fornecidos pelo sistema operacional, pouco portável mas com melhor desempenho.

Na sequência deste texto serão apresentadas as funções de acesso usando *streams*, que são o padrão da linguagem C e valem para qualquer sistema operacional. Explicações sobre entrada/saída usando descritores UNIX podem ser encontradas nesta página.

Entradas e saídas padrão

Cada programa em execução tem acesso a três arquivos padrão definidos no arquivo de cabeçalho `stdio.h`, que são:

- `FILE* stdin`: a entrada padrão, normalmente associada ao teclado do terminal, usada para a entrada de dados do programa (`scanf`, por exemplo).
- `FILE* stdout`: a saída padrão, normalmente associada à tela do terminal, usada para as saídas normais do programa (`printf`, por exemplo).
- `FILE* stderr`: a saída de erro, normalmente associada à tela do terminal, usada para mensagens de erro.

Esses três arquivos não precisam ser abertos, eles estão prontos para uso quando o programa inicia. Geralmente eles estão associados ao terminal onde o programa foi lançado, mas podem ser redirecionados pelo *shell* (mais detalhes aqui).

Abrindo e fechando arquivos

Antes de ser usado, um arquivo precisa ser "aberto" pela aplicação (com exceção dos arquivos padrão descritos acima, que são abertos automaticamente). Isso é realizado através da chamada `fopen`:

```
FILE* fopen (const char *filename, const char *mode)
```

Abre um arquivo indicado por `filename` e retorna um ponteiro para o *stream*. A *string* `mode` define o modo de abertura do arquivo:

- `r` : abre um arquivo existente para leitura (*read*).
- `w` : abre um arquivo para escrita (*write*). Se o arquivo já existe, seu conteúdo é descartado. Senão, um novo arquivo vazio é criado.
- `a` : abre um arquivo para concatenação (*append*). Se o arquivo já existe, seu conteúdo é preservado e as escritas serão concatenadas no final do arquivo. Senão, um novo arquivo vazio é criado.
- `r+` : abre um arquivo existente para leitura e escrita. O conteúdo anterior do arquivo é preservado e o ponteiro é posicionado no início do arquivo.
- `w+` : abre um arquivo para leitura e escrita. Se o arquivo já existe, seu conteúdo é descartado. Senão, um novo arquivo vazio é criado.
- `a+` : abre um arquivo para escrita e concatenação. Se o arquivo já existe, seu conteúdo é preservado e as escritas serão concatenadas no final do arquivo. Senão, um novo arquivo vazio é criado. O ponteiro de leitura é posicionado no início do arquivo, enquanto as escritas são efetuadas no seu final.

Os modos a e a+ **sempre** escreverão no final do arquivo, mesmo se o cursor do mesmo for movido para outra posição.

Fecha um *stream*. Os dados de saída em *buffer* são escritos, enquanto dados de entrada são descartados:

```
int fclose (FILE* stream)
```

Fecha e abre novamente um *stream*, permitindo alterar o arquivo e/ou modo de abertura:

```
FILE* freopen (const char *filename, const char *mode, FILE *stream)
```

Exemplo: abrindo o arquivo x em leitura:

fopen-read.c

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

int main ()
{
   FILE* arq ;

   arq = fopen ("x", "r") ;

   if ( ! arq )
```

```
   {
      perror ("Erro ao abrir arquivo x") ;
      exit (1) ; // encerra o programa com status 1
   }

   fclose (arq) ;
   exit (0) ;
}
```

Exemplo: abre o arquivo x em leitura/escrita, criando-o se não existir:

fopen-write.c

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

int main (int argc, char *argv[])
{
   FILE* arq ;

   arq = fopen ("x", "w+") ;

   if ( ! arq )
   {
      perror ("Erro ao abrir/criar arquivo x") ;
      exit (1) ; // encerra o programa com status 1
   }

   fclose (arq) ;
   exit (0) ;
}
```

Arquivos-texto

Arquivos-texto contêm sequências de bytes representando um texto simples (sem formatações especiais, como negrito, itálico, etc), como código-fonte ou uma página em HTML, por exemplo.

Em um arquivo-texto, os caracteres do texto são representados usando uma codificação padronizada, para que possam ser abertos por diferentes aplicações em vários sistemas operacionais. As codificações de caracteres mais usuais hoje são:

- ASCII: criada em 1963 para representar os caracteres comuns da língua inglesa, usando 7 bits (valores entre 0 e 127).
- ISO-8859: conjunto de extensões da codificação ASCII para suportar outras línguas com alfabeto latino, como o Português. Os caracteres entre 0 e 127 os mesmos da tabela ASCII, enquanto os caracteres entre 128 e 255 são específicos. Por exemplo, a extensão ISO-8859-1 contém os caracteres acentuados e cedilhas da maior parte das linguagens ocidentais (Português, Espanhol, Francês etc).
- UTF-8: codificação baseada no padrão Unicode, capaz de representar mais de um milhão de caracteres em todas as línguas conhecidas, além de sinais gráficos (como *emojis*). Os caracteres em UTF-8 podem usar entre 1 e 4 bytes para sua representação, o que torna sua manipulação mais complexa em programas.

Além dos caracteres em si, as codificações geralmente suportam **caracteres de controle**, que permitem representar algumas estruturas básicas de formatação do texto, como quebras de linha. Alguns caracteres de controle presentes nas codificações acima são:

nome	valor	representação
null	0	NUL \0 ^@
bell	7	BEL \a ^G
backspace	8	BS \b ^H
tab	9	HT \t ^I
line feed	10	LF \n ^J
form feed	12	FF \f ^L
carriage return	13	CR \r ^M
escape	27	ESC ^[

Escrita de arquivos

Escrita simples

Estas funções permitem gravar caracteres ou strings simples em *streams*.

```
int fputc   (int c, FILE* stream)   // escreve o caractere c no stream
int putc    (int c, FILE* stream)   // idem, implementada como macro
int putchar (int c)                 // idem, em "stdout"

int fputs (const char *s, FILE* stream)  // escreve a string s no stream
int puts  (const char *s)                // idem, em "stdout"
```

Um exemplo de uso de operações de escrita simples em arquivo:

escreve-ascii.c

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

int main ()
{
  FILE *arq ;
  unsigned char c ;

  // abre o arquivo em escrita
  arq = fopen ("ascii.txt", "w+") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir/criar arquivo") ;
     exit (1) ;
  }

  // escreve os caracteres ascii
  fputs ("caracteres ASCII:", arq) ;
  for (c=32; c<128; c++)
  {
    fputc (c, arq) ;
    fputc (' ', arq) ;
```

```
  }
  fputc ('\n', arq) ;

  // fecha o arquivo
  fclose (arq) ;
}
```

Escrita formatada

As operações de entrada e saída formatada usam padrões para formatação dos diversos tipos de dados descritos em livros de programação em C e no manual da GLibC.

Escreve dados usando a formatação definida em `format` no *stream* de saída padrão `stdout`:

```
int printf (const char* format, ...)
```

Idêntico a `printf`, usando o stream indicado:

```
int fprintf (FILE* stream, const char* format, ...)
```

Similar a `printf`, mas a saída é depositada na string `str`:

```
int sprintf (char* str, const char* format, ...)
```

Atenção: o programador deve garantir que `str` tenha tamanho suficiente para receber a saída; caso contrário, pode ocorrer um *buffer overflow* com consequências imprevisíveis. As funções `snprintf` e `asprintf` são mais seguras e evitam esse problema.

escreve-tabuada.c

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

int main ()
{
  FILE *arq ;
  int i, j ;

  // abre o arquivo em escrita
  arq = fopen ("tabuada.txt", "w+") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir/criar arquivo") ;
     exit (1) ;
  }

  // escreve o cabeçalho
  fprintf (arq, "Tabuada:\n") ;

  fprintf (arq, "       ") ;
  for (j=0; j<= 10; j++)
```

```
      fprintf (arq, "%4d", j) ;
  fprintf (arq, "\n") ;

  fprintf (arq, "       ") ;
  for (j=0; j<= 10; j++)
      fprintf (arq, "----") ;
  fprintf (arq, "\n") ;

  // escreve as linhas de valores
  for (i=0; i<= 10; i++)
  {
    fprintf (arq, "%4i | ", i) ;
    for (j=0; j<= 10; j++)
      fprintf (arq, "%4d", i*j) ;
    fprintf (arq, "\n") ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Leitura de arquivos

Leitura simples

Estas funções permitem ler caracteres isolados de um *stream*. O valor lido é um `int` indicando o caractere lido ou então o valor especial EOF (*End-Of-File*):

```
int fgetc (FILE* stream)  // Lê o próximo caractere do stream
int getc  (FILE* stream)  // Idem, como macro (mais rápida)
int getchar ()            // Idem, sobre stdin
```

Para a leitura de strings  :

```
char* gets (char *s)
```

Lê caracteres de `stdin` até encontrar um *newline* e os armazena na string `s`. O caractere *newline* é descartado.

Atenção: a função `gets` é **perigosa**, pois não provê segurança contra *overflow* na string `s`. Sempre que possível, deve ser usada a função `fgets` ou `getline`.

Lê uma linha de caracteres do *stream* e a deposita na string `s`. O tamanho da linha é limitado em `count-1` caracteres, aos quais é adicionado o `'\0'` que marca o fim da string. O *newline* é incluso.

```
char* fgets (char *s, int count, FILE *stream)
```

O exemplo a seguir lê e numera as 10 primeiras linhas de um arquivo:

numera-linhas.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  int i ;
  char line[LINESIZE+1] ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê as 10 primeiras linhas do arquivo
  for (i=0; i<10; i++)
  {
    fgets (line, LINESIZE, arq) ;
    printf ("%d: %s", i, line) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Leitura formatada

Lê dados do *stream* `stdin` de acordo com a formatação definida na string `format`. Os demais argumentos são ponteiros para as variáveis onde os dados lidos são depositados. Retorna o número de dados lidos ou EOF:

```c
int scanf (const char* format, ...)
```

Similar a `scanf`, mas usando como entrada o *stream* indicado:

```c
int fscanf (FILE* stream, const char* format, ...)
```

Similar a `scanf`, mas usando como entrada a string `s`:

```c
int sscanf (const char* s, const char* format, ...)
```

O exemplo a seguir lê 10 valores reais de um arquivo:

le-valores.c

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

int main ()
```

```
{
  FILE *arq ;
  int i ;
  float value ;

  // abre o arquivo em leitura
  arq = fopen ("numeros.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê os 10 primeiros valores do arquivo
  for (i=0; i<10; i++)
  {
    fscanf (arq, "%f", &value) ;
    printf ("%d: %f\n", i, value) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Experimente executá-lo com os dados de entrada abaixo. Pode explicar o que acontece?

numeros.txt

```
10 21 4
  23.7  55 -0.7
6 5723.8, 455
1, 2, 3, 4
```

A função `scanf` considera espaços (espaços em branco, tabulações e novas linhas) como separadores *default* dos campos de entrada. O arquivo `numeros.txt` contém uma vírgula, que não é um separador, então a leitura não pode prosseguir até que a vírgula seja lida.

Um bloco de leitura mais robusto, imune a esse problema, seria:

```
// lê os 10 primeiros valores do arquivo
i = 0 ;
while (i < 10)
{
  ret = fscanf (arq, "%f", &value) ;

  // fim de arquivo ou erro?
  if (ret == EOF)
    break ;

  // houve leitura?
  if (ret > 0)
  {
```

```
    printf ("%d: %f\n", i, value) ;
    i++ ;
  }
  // não houve, tira um caractere e tenta novamente
  else
    fgetc (arq) ;
}
```

Fim de arquivo

Muitas vezes deseja-se ler os dados de um arquivo até o fim, mas não se conhece seu tamanho a priori. Para isso existem funções e macros que indicam se o final de um arquivo foi atingido.

A função recomendada para testar o final de um arquivo é `feof(stream)`. Ela retorna 0 (zero) se o final do arquivo **não foi** atingido. Eis um exemplo de uso:

numera-todas.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  int i ;
  char line[LINESIZE+1] ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê TODAS as linhas do arquivo
  i = 1 ;
  fgets (line, LINESIZE, arq) ;     // tenta ler uma linha
  while (! feof (arq))              // testa depois de tentar ler!
  {
    printf ("%d: %s", i, line) ;
    fgets (line, LINESIZE, arq) ;   // tenta ler a próxima linha
    i++ ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Observe que a função `feof()` indica TRUE somente **após** o final do arquivo ter sido atingido, ou seja, após uma leitura falhar. Por isso, `feof()` deve ser testada **depois** da leitura e não antes.

A macro EOF representa o valor devolvido por funções de leitura de caracteres como `getchar` e `fgetc` quando o final do arquivo é atingido:

le-eof.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  char c ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê os caracteres até o fim do arquivo
  c = fgetc (arq) ;       // tenta ler um caractere
  while (c != EOF)      // não é o fim do arquivo
  {
    printf ("%c ", c) ;       // tenta ler o próximo
    c = fgetc (arq) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Por fim, esta função retorna um valor não nulo se ocorreu um erro no último acesso ao *stream*:

```
int ferror (FILE* stream)
```

Além de ajustar o indicador de erro do *stream*, as funções de acesso a *streams* também ajustam a variável `errno`.

Exercícios

1. Escreva um programa em C para informar o número de caracteres presentes em um arquivo de texto.
2. Escreva um programa em C que leia um arquivo de texto com números reais (um número por linha) e informe a média dos valores lidos.
3. Escreva um programa em C para ler um arquivo `minusc.txt` e escrever um arquivo `maiusc.txt` contendo o mesmo texto em maiúsculas.
4. O arquivo mapa.txt contém o mapa de um nível do jogo Boulder Dash. Escreva um programa em C que

carregue esse mapa em uma matriz de caracteres.

5. Escreva um programa `mycp` para fazer a cópia de um arquivo em outro: `mycp arq1 arq2`. Antes da cópia, `arq1` deve existir e `arq2` não deve existir. Mensagens de erro devem ser geradas caso essas condições não sejam atendidas ou o nome dado a `arq2` seja inválido. Para acessar os nomes dos arquivos na linha de comando use os parâmetros `argc` e `argv` (veja aqui).
6. o comando `grep` do UNIX imprime na saída padrão (*stdout*) as linhas de um arquivo de entrada que contenham uma determinada string informada como parâmetro. Escreva esse programa em C (dica: use a função `strstr`).

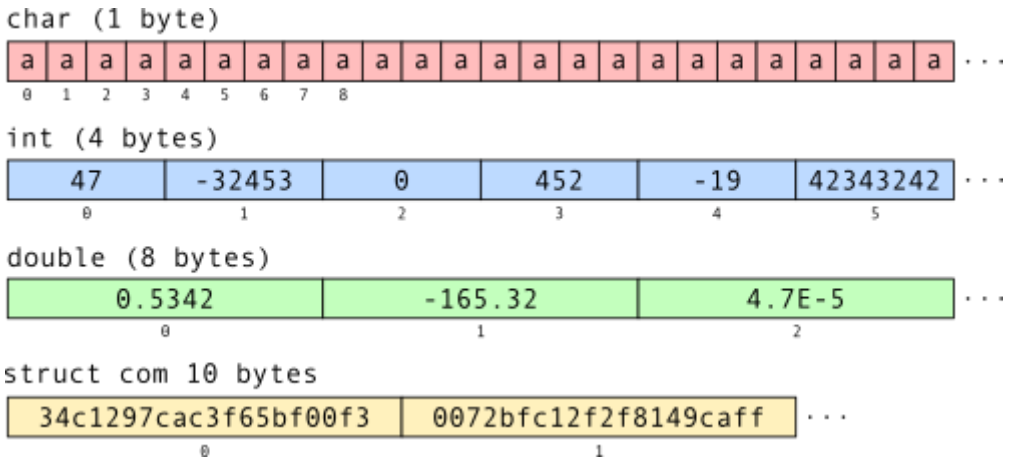
Mais exercícios no capítulo 11 da apostila do NCE/UFRJ.

Arquivos binários

Video desta aula

Todos os arquivos contêm sequências de bytes, mas costuma-se dizer que um arquivo é “binário” quando seu conteúdo não é uma informação textual (ou seja, não representa um texto usando codificações como ASCII, UTF-8 ou outras). Arquivos binários são usados para armazenar informações mais complexas como imagens, música, código executável, etc.

Em C, um arquivo binário é visto como uma sequência de blocos de mesmo tamanho. O tamanho dos blocos depende do tipo de informação armazenada no arquivo. Por exemplo, um arquivo de números reais `double` terá blocos de 8 bytes, enquanto um arquivo de caracteres (`char`) terá blocos de 1 byte, como mostra a figura:



Lembre-se que o SO só armazena a sequência de bytes, sem considerar nem registrar o tamanho dos blocos. **Cabe à aplicação** definir o tamanho de bloco que deseja usar em cada arquivo.

Leitura/escrita de blocos

A linguagem C oferece funções para ler e escrever blocos de bytes em arquivos, que efetuam a cópia desses bytes da memória para o arquivo ou vice-versa.

A funções a seguir permitem ler/escrever blocos de bytes em arquivos binários. Todas essas funções estão definidas no arquivo `stdio.h`.

Lê até `count` blocos de tamanho `size` bytes cada um e os deposita no vetor `data`, a partir do *stream* indicado. Retorna o número de blocos lidos:

```
size_t fread (void* data, size_t size, size_t count, FILE* stream)
```

Escreve até `count` blocos de tamanho `size` bytes do vetor `data` no *stream* indicado. Retorna o número de blocos escritos:

```
size_t fwrite (const void* data, size_t size, size_t count, FILE* stream)
```

Essas funções também podem ser usadas para ler/escrever em arquivos-texto, pois textos são sequências de blocos de 1 byte.

Exemplo de uso

Este exemplo manipula um arquivo binário `numeros.dat` contendo números reais. São implementadas (em arquivos separados) as operações de escrita de números no arquivo, listagem e ordenação do conteúdo:

escreve.c

```c
// Escreve N valores reais aleatórios em um arquivo, em formato binário

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

#define ARQUIVO "numeros.dat"
#define NUMVAL 10

int main (int argc, char *argv[])
{
  FILE* arq ;
  int i, ret ;
  float value[NUMVAL] ;

  // abre o arquivo em modo "append"
  arq = fopen (ARQUIVO, "a") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  // inicia gerador de aleatórios
  srand (clock()) ;

  // gera NUMVAL valores aleatórios reais
  for (i = 0; i< NUMVAL; i++)
    value[i] = random() / 100000.0 ;

  // escreve os valores gerados no final do arquivo
  ret = fwrite (value, sizeof(float), NUMVAL, arq) ;
  if (ret)
    printf ("Gravou %d valores com sucesso!\n", ret) ;
  else
    printf ("Erro ao gravar...\n") ;

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

lista.c

```c
// Lista o conteúdo de um arquivo que contém números reais em formato binário

#include <stdio.h>
```

```c
#include <stdlib.h>

#define ARQUIVO "numeros.dat"

int main (int argc, char *argv[])
{
  FILE* arq ;
  float value ;

  // abre o arquivo em modo leitura
  arq = fopen (ARQUIVO, "r") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  // lê e imprime os valores contidos no arquivo
  fread (&value, sizeof(float), 1, arq) ;
  while (! feof(arq))
  {
    printf ("%f\n", value) ;
    fread (&value, sizeof(float), 1, arq) ;
  }

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

ordena.c

```c
// Lê os números reais de um arquivo binário, os ordena e os escreve de volta
no arquivo

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

#define ARQUIVO "numeros.dat"
#define MAXVAL 100000

float value[MAXVAL] ;
int num_values ;

int main (int argc, char *argv[])
{
  FILE* arq ;
  int i, j, menor ;
  float aux ;

  // abre o arquivo em leitura/escrita, preservando o conteúdo
  arq = fopen (ARQUIVO, "r+") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
```

```
    exit (1) ;
  }

  // lê números do arquivo no vetor
  num_values = fread (value, sizeof(float), MAXVAL, arq) ;
  printf ("Encontrei %d números no arquivo\n", num_values) ;

  // ordena os números (por seleção)
  for (i=0; i < num_values-1; i++)
  {
    // encontra o menor elemento no restante do vetor
    menor = i ;
    for (j=i+1; j < num_values; j++)
      if (value[j] < value[menor])
        menor = j ;

    // se existe menor != i, os troca entre si
    if (menor != i)
    {
      aux = value[i] ;
      value[i] = value[menor] ;
      value[menor] = aux ;
    }
  }

  // retorna o ponteiro ao início do arquivo
  rewind (arq) ;

  // escreve números do vetor no arquivo
  fwrite (value, sizeof(float), num_values, arq) ;

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

No arquivo `ordena.c`, o conteúdo inteiro do arquivo é lido com **apenas uma** chamada `fread` e escrito com apenas uma chamada `fwrite`. Isso é perfeitamente possível, desde que a estrutura usada para receber os dados na memória coincida byte a byte com a forma como eles estão dispostos no arquivo.

Posicionamento no arquivo

Para cada arquivo aberto em uma aplicação, o sistema operacional mantém um contador interno indicando a posição da próxima operação de leitura ou escrita. Esse contador é conhecido como **ponteiro de posição** (embora não seja realmente um ponteiro).

Por default, as operações em um arquivo em C ocorrem em posições sucessivas dentro do arquivo: cada leitura (ou escrita) corre **após** a leitura (ou escrita) precedente, até atingir o final do arquivo. Essa forma de acesso ao arquivo é chamada de **acesso sequencial**.

Por vezes uma aplicação precisa ler ou escrever em posições específicas de um arquivo, ou precisa voltar a ler uma posição do arquivo que já percorreu anteriormente. Isso ocorre frequentemente em aplicações que manipulam arquivos muito grandes, como vídeos ou bases de dados. Para isso é necessária uma forma de

acesso direto a posições específicas do arquivo.

Em C, o acesso direto a posições específicas de um arquivo é feita através de funções de **posicionamento de ponteiro**, que permitem alterar o valor do ponteiro de posição do arquivo, antes da operação de leitura/escrita desejada.

Todas as funções de manipulação do ponteiro de arquivo consideram as **posições em bytes** a partir do início do arquivo, nunca em número de blocos.

As funções mais usuais para acessar o ponteiro de posição de um arquivo em C são indicadas a seguir.

Ajusta posição do ponteiro no *stream* indicado:

```
int fseek (FILE* stream, long int offset, int whence)
```

O ajuste é definido por `offset`, enquanto o valor de `whence` indica se o ajuste é relativo ao início do arquivo (SEEK_SET), à posição corrente (SEEK_CUR) ou ao final do arquivo (SEEK_END). Ver também `fseeko` e `fseeko64`. Exemplos:

```
                              // posiciona o ponteiro de "arq":
fseek (arq, 1000, SEEK_SET) ; // 1000 bytes após o início
fseek (arq, -300, SEEK_END) ; // 300 bytes antes do fim
fseek (arq, -500, SEEK_CUR) ; // 500 bytes antes da posição atual
```

Reposiciona o ponteiro no início (posição 0) do *stream* indicado:

```
void rewind (FILE* stream)
```

Informa a posição corrente de leitura/escrita em `stream` (ver também `ftello` e `ftello64` no manual).

```
long int ftell (FILE* stream)
```

Outras funções

Para truncar (“encurtar”) um arquivo, deixando somente os primeiros `length` bytes:

```
#include <unistd.h>
#include <sys/types.h>

// usando o nome do arquivo, sem abri-lo
int truncate  (const char *path, off_t length);

// usando um descritor UNIX (fd)
int ftruncate (int fd, off_t length);
```

Para obter as propriedades (metadados) de um arquivo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

// usando o nome do arquivo, sem abri-lo
```

```
int stat  (const char *pathname, struct stat *statbuf);

// usando um descritor UNIX (fd)
int fstat (int fd, struct stat *statbuf);
```

As informações sobre o arquivo serão depositadas pelo núcleo na estrutura `statbuf`, cujo conteúdo está descrito abaixo. Os campos da estrutura são detalhados na página de manual da função `fstat()`.

```
struct stat
{
  dev_t     st_dev;         /* ID of device containing file */
  ino_t     st_ino;         /* Inode number */
  mode_t    st_mode;        /* File type and mode */
  nlink_t   st_nlink;       /* Number of hard links */
  uid_t     st_uid;         /* User ID of owner */
  gid_t     st_gid;         /* Group ID of owner */
  dev_t     st_rdev;        /* Device ID (if special file) */
  off_t     st_size;        /* Total size, in bytes */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */
  #define st_atime st_atim.tv_sec  /* For backward compatibility */
  #define st_mtime st_mtim.tv_sec
  #define st_ctime st_ctim.tv_sec
};
```

Exercícios

1. Escreva três programas C separados para:
    1. escrever um arquivo com 10 milhões de inteiros `long` aleatórios, armazenados em modo binário;
    2. ler o arquivo de inteiros em um vetor, ordenar o vetor e reescrever o arquivo;
    3. escrever na tela os primeiros 10 números e os últimos 10 números contidos no arquivo.

Mais exercícios no capítulo 11 da apostila do NCE/UFRJ.

Organização do código

Video desta aula

À medida em que um software cresce em tamanho e funcionalidades, seu código-fonte deve ser organizado corretamente para facilitar sua compreensão, manutenção e evolução. É importante quebrar o código-fonte em arquivos separados, dividindo-o de acordo com os módulos e/ou funcionalidades do sistema.

O método usual de organização de código em C consiste em dividir o programa em módulos que são vistos como "bibliotecas", provendo funcionalidades para a construção do programa principal. O programa principal deve usar as funcionalidades desses módulos e permanecer o mais compacto e abstrato possível.

Um programa em C é estruturado em arquivos de código (extensão `.c`) e arquivos de cabeçalho (*header*, extensão `.h`). Os arquivos de código contêm implementações concretas, enquanto os de cabeçalho contêm protótipos de funções e os tipos de dados necessários a esses protótipos.

Dado um arquivo `cpx.c` contendo funções e tipos de dados para manipular números complexos, o arquivo `cpx.h` deve ser visto como a definição da **interface** para outros arquivos C usarem as funcionalidades implementadas por `cpx.c`.

Com isso, funções e tipos que são usados **somente** dentro de `cpx.c` não precisam (nem devem) aparecer no arquivo de interface `cpx.h`.

Ao dividir o código-fonte em arquivos separados, alguns cuidados devem ser tomados:

- Agrupe as funções e definições de dados associados ao mesmo tópico ou assunto em um mesmo arquivo `.c`;
- Coloque os protótipos das funções públicas e as definições de dados necessárias a esses protótipos em um arquivo de cabeçalho `.h` com o mesmo nome do arquivo `.c` correspondente;
- Somente faça inclusões (`#include`) de arquivos de cabeçalho (`.h`).

Maldições imperdoáveis :

- fazer inclusão de arquivos "`.c`" (`#include "arquivo.c"`)
- colocar código real (for, if, while, ...) em arquivos de cabeçalho [1)]

Exemplo

Este exemplo implementa uma mini-biblioteca de números complexos, ou seja, um conjunto de funções para definir e manipular números complexos[2)].

O arquivo principal (neste exemplo, `exemplo.c`) usa funções dessa biblioteca. Para isso, ele deve incluir todos os arquivos de cabeçalho necessários para sua compilação e também deve definir a função `main`:

exemplo.c

```
// Demonstraçao da biblioteca simples de números complexos :-)
// Carlos Maziero, DINF/UFPR 2020
```

```c
#include <stdio.h>
#include "cpx.h"

int main ()
{
  cpx_t a, b, c, d ;

  // (10 + 7i) + (-2 + 4i) = (8 + 11i)
  a = cpx (10, 7) ;
  b = cpx (-2, 4) ;
  c = cpx_sum (a, b);
  printf ("c vale %s\n", cpx_str (c)) ;

  // (3 + 2i) * (1 + 4i) = –5 + 14i
  d = cpx_mult (cpx (3, 2), cpx (1, 4));
  printf ("d vale %s\n", cpx_str (d)) ;
}
```

Como o arquivo `exemplo.c` não define funções (ou estruturas, tipos, etc) que serão usadas em outros arquivos do programa, não se deve criar um arquivo `exemplo.h`.

Nossa "biblioteca" de números complexos é implementada pelos arquivos `cpx.c` e `cpx.h`.

O arquivo de cabeçalho `cpx.h` deve declarar somente informações públicas: os tipos de dados e protótipos das funções que devem ser conhecidas por quem irá utilizar as funcionalidades da biblioteca:

cpx.h

```c
// Biblioteca simples de números complexos :-)
// Carlos Maziero, DINF/UFPR 2020

#ifndef __CPX__
#define __CPX__

// estrutura de um número complexo
typedef struct {
  float r, i;   // partes real e imaginária
} cpx_t ;

// define o valor de um complexo
cpx_t cpx (float r, float i) ;

// gera uma string a partir de um número complexo
char* cpx_str (cpx_t c) ;

// operações aritméticas entre dois complexos
cpx_t cpx_sum  (cpx_t a, cpx_t b) ;
cpx_t cpx_sub  (cpx_t a, cpx_t b) ;
cpx_t cpx_mult (cpx_t a, cpx_t b) ;
cpx_t cpx_div  (cpx_t a, cpx_t b) ;

// outras operações
// ...
```

```
#endif
```

Deve-se observar o uso das macros de pré-compilação `#ifndef` e `#define`. Elas constituem uma *include guard*, usada para evitar a repetição das definições, caso o mesmo arquivo de cabeçalho seja incluído múltiplas vezes em diferentes locais do código.

Por sua vez, o arquivo `cpx.c` contém as informações privadas da biblioteca (estruturas de dados internas, variáveis globais) e as implementações das funções definidas em `cpx.h`. Esse arquivo deve incluir todos os cabeçalhos necessários à implementação das funções.

cpx.c

```
// Biblioteca simples de números complexos :-)
// Carlos Maziero, DINF/UFPR 2020

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cpx.h"

// função interna, usada somente neste arquivo
// com o "static", esta função só é visível neste arquivo
static void polar_to_rect (float r, float a, float *x, float *y)
{
  // implementação da função
  // ...
}

// define o valor de um complexo
cpx_t cpx (float r, float i)
{
  cpx_t new = {r, i} ;
  return (new) ;
}

// gera uma string representando um número complexo
char* cpx_str (cpx_t c)
{
  char *str ;

  // um dia, consertar este "memory leak"...
  str = malloc (128) ;  // suficiente para dois floats
  if (!str)
  {
    perror ("malloc") ;
    return (NULL) ;
  }
  sprintf (str, "%f%+fi", c.r, c.i) ;
  return (str) ;
}

// soma de dois complexos
cpx_t cpx_sum (cpx_t a, cpx_t b)
{
```

```
  cpx_t sum ;
  sum.r = a.r + b.r ;
  sum.i = a.i + b.i ;
  return (sum) ;
}

// diferença de dois complexos
cpx_t cpx_sub (cpx_t a, cpx_t b)
{
  cpx_t sum ;
  sum.r = a.r - b.r ;
  sum.i = a.i - b.i ;
  return (sum) ;
}

// produto de dois complexos
cpx_t cpx_mult (cpx_t a, cpx_t b)
{
  cpx_t prod ;
  prod.r = a.r * b.r - a.i * b.i ;
  prod.i = a.r * b.i + a.i * b.r ;
  return (prod) ;
}

// ... (demais funções)
```

Em resumo:

- `cpx.c`: implementação das funções de manipulação de números complexos.
- `cpx.h`: interface (protótipos) das funções **públicas** definidas em `cpx.c`.
- `exemplo.c`: programa principal, que usa as funções descritas em `cpx.h` e implementadas em `cpx.c`.

Para compilar:

```
cc -Wall exemplo.c cpx.c -o exemplo
```

O arquivo `cpx.c` também pode ser compilado separadamente, gerando um arquivo-objeto `cpx.o` que poderá ser ligado ao arquivo `exemplo.o` posteriormente:

```
cc -Wall -c cpx.c

cc -Wall exemplo.c cpx.o -o exemplo
```

Essa organização torna mais simples a construção de bibliotecas e a distribuição de código binário para incorporação em outros projetos (reuso de código). Além disso, essa estruturação agiliza a compilação de grandes projetos, através do sistema Make.

Um outro aspecto importante da organização do código é o uso de declarações `extern` para variáveis globais usadas em vários arquivos de código-fonte. Esta página contém uma excelente explicação sobre o uso correto da declaração `extern`.

1)
Exceto quando se tratar de funções ''inline''

2)
Esta biblioteca é inútil, pois o padrão C99 inclui o suporte a números complexos ...

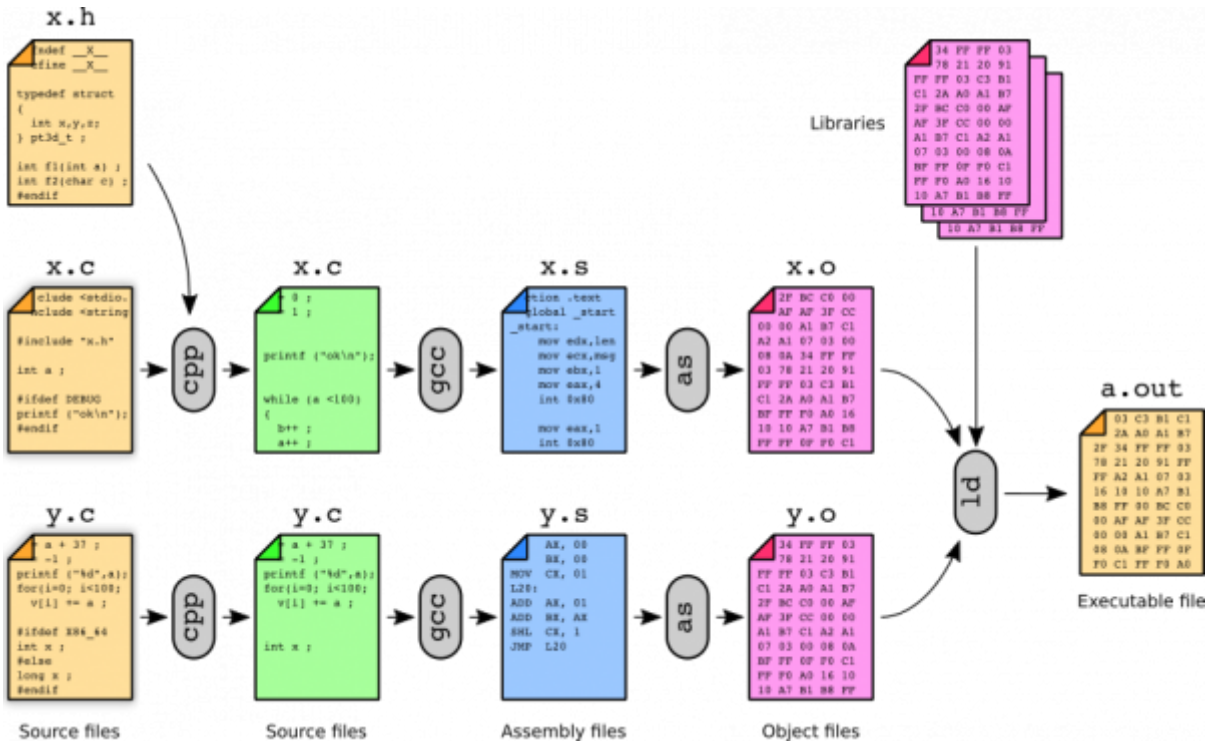
O preprocessador C

Video desta aula

A transformação de um programa C em um arquivo executável é um processo complexo, com várias etapas, sendo as mais importantes:

- **preprocessamento**: tratamento das diretivas do preprocessador (`#include`, etc)
- **compilação**: conversão de C para *assembly*
- **tradução**: conversão de *assembly* para código de máquina (binário)
- **ligação**: junção dos arquivos-objeto e bibliotecas em código de máquina para formar o arquivo executável

Ele está ilustrado na figura a seguir, onde `cpp` corresponde ao preprocessamento, `gcc` à compilação, `as` à tradução e `ld` à ligação.



Preprocessamento

O preprocessador C (CPP - *C PreProcessor*) é uma ferramenta de **substituição de texto** invocada automaticamente pelo compilador C/C++ no início do processo de compilação. Apesar de não fazer parte formal da sintaxe da linguagem C, seu uso é praticamente indispensável para a estruturação de programas C, mesmo os mais simples.

Todos os comandos do preprocessador começam com o símbolo # no início de uma linha (podem haver espaços e tabs antes).

A saída do preprocessador C, que será enviada ao compilador propriamente dito, pode ser obtida através do flags `-E`:

```
gcc -E arquivo.c
```

Inclusão de arquivos

O preprocessador é frequentemente usado para incluir arquivos externos em um código-fonte.

Considere este exemplo:

escreva.h

```
#ifndef __ESCREVA__
#define __ESCREVA__

void escreva (char *msg) ;

#endif
```

escreva.c

```
#include <stdio.h>

void escreva (char *msg)
{
  printf ("%s", msg) ;
}
```

hello.c

```
#include "escreva.h"

int main ()
{
  escreva ("Hello, world!\n") ;
  return (0) ;
}
```

Neste exemplo, o preprocessador irá **substituir** cada linha `#include …` pelo conteúdo do respectivo arquivo, gerando um único arquivo temporário que será entregue ao compilador C.

Há uma diferença importante entre as duas formas de inclusão:

- `#include <…>`: o arquivo indicado será buscado nos diretórios default do compilador, geralmente `/usr/include/*` nos sistemas Unix.
- `#include "…"`: o arquivo indicado será buscado primeiro no diretório corrente (onde está o arquivo que está sendo compilado), e depois nos diretórios default do compilador.

Definição e uso de constantes

O preprocessador é frequentemente usado para a definição de constantes, através do comando `#define`:

vetor.c

```c
#define VETSIZE 64

int main ()
{
  int vetor[VETSIZE] ;

  for (i=0; i<VETSIZE; i++)
    vetor[i] = i ;
}
```

Após a definição, todas as ocorrências da string VETSIZE no arquivo serão substituídas pelo valor 64, **antes da compilação**, resultando no seguinte código-fonte:

```c
int vetor[64] ;

for (i=0; i<64; i++)
  vetor[i] = 0 ;
```

Para evitar confusões entre variáveis da linguagem C e constantes do preprocessador, convencionou-se definir as constantes em MAIÚSCULAS.

Constantes predefinidas

Algumas constantes são definidas previamente pelo sistema:

- __DATE__: data atual (formato “MMM DD YYYY”)
- __TIME__: horário atual (formato “HH:MM:SS”)
- __FILE__: nome do arquivo corrente.
- __LINE__: número da linha corrente do código-fonte.
- __func__: nome da função corrente.

Um exemplo de uso:

data.c

```c
#include <stdio.h>

int main ()
{
  printf ("Este código foi compilado em %s\n", __DATE__) ;
}
```

Além destas, muitas outras constantes podem estar disponíveis, dependendo da plataforma:

- Macros predefinidas no preprocessador GNU
- Pre-defined C/C++ Compiler Macros

Compilação condicional

Uma constante pode ser definida sem um valor específico, Neste caso ela funciona como como um *flag* verdadeiro/falso, que pode ser testado pelo preprocessador através de comandos específicos:

debug.c

```
#include <stdio.h>

#define VETSIZE 64

int main ()
{
  int i, vetor[VETSIZE] ;

  for (i=0; i<VETSIZE; i++)
  {
    vetor[i] = i ;
    #ifdef DEBUG
    printf ("Valor de i: %d\n", i) ;
    #endif
  }
}
```

No exemplo acima, a linha do `printf` só estará presente no código enviado ao compilador se a constante DEBUG estiver definida.

Constantes podem ser definidas no código-fonte usando o comando `#define` (como nos exemplos acima), mas também podem ser definidas na linha de comando, ao invocar o compilador:

```
gcc -DDEBUG debug.c
```

Um uso frequente da compilação condicional é a construção de *include guards*, ou seja código para evitar múltiplas inclusões do mesmo arquivo:

headers.h

```
#ifndef _THIS_HEADER_FILE_
#define _THIS_HEADER_FILE_
...
#endif
```

Da mesma forma, pode-se evitar redefinir constantes que já estejam definidas:

```
#ifndef NULL
#define NULL (void *) 0
#endif
```

Além do `ifdef`, existem outros operadores condicionais, como o `if` - `elif` - `else`:

```
#if DEBUG_LEVEL > 5
  // print all debug messages
  ...
#elif DEBUG_LEVEL > 3
  // print relevant debug messages
```

```
  ...
#elif DEBUG_LEVEL > 1
  // print prioritary debug messages
  ...
#else
  // print no debug messages
#endif
```

O operador `defined` permite testar se uma macro está definida:

```
#if defined (__arm__)         // macro definida em sistemas ARM
  #warning "Generating code for ARM processor."
  // code for ARM processors
  ...
#elif defined (__i386__)      // idem, x86
  #warning "Generating code for x86 processor."
  // code for Intel 32-bit processors
  ...
#else
  // abort compilation
  #error "Unknown architecture, aborting."
#endif
```

Observe o uso das diretivas `#warning` e `#error` no programa acima.

Macros com parâmetros

O preprocessador é usado com frequência para construir macros, que são funções simples com parâmetros:

macro1.c

```
#include <stdio.h>

#define SQUARE(x) x*x

int main ()
{
  printf ("O quadrado de 5 é %d\n", SQUARE(5)) ;
}
```

O código acima, ao ser tratado pelo preprocessador, será transformado em:

```
printf ("O quadrado de 5 é %d\n", 5*5) ;
```

Observe que a macro SQUARE não computou o resultado de 5*5, apenas fez a substituição do parâmetro 5 pela expressão que ela define (5*5).

Tome **muito** cuidado ao definir macros com parâmetros, pois a substituição de texto pode levar a expressões erradas!

O exemplo abaixo apresenta um erro dessa natureza:

```
#define SQUARE(x) x*x
...
printf ("O quadrado de 2+3 é %d\n", SQUARE(2+3)) ;
```

O preprocessador transformará essa expressão em:

```
printf ("O quadrado de 2+3 é %d\n", 2+3*2+3) ;
```

O resultado da expressão deveria ser 25 (o quadrado de 2+3) mas será 11, por causa da precedência entre os operadores aritméticos (que o preprocessador não trata).

Para evitar esse erro, a macro deve ser declarada usando **parênteses**:

macro2.c

```
#include <stdio.h>

#define SQUARE(x) (x)*(x)

int main ()
{
  printf ("O quadrado de 5 é %d\n", SQUARE(2+3)) ;
}
```

Que resulta na expressão correta:

```
printf ("O quadrado de 2+3 é %d\n", (2+3)*(2+3)) ;
```

Operações avançadas

O preprocessador C tem operações avançadas que vão muito além do escopo desta breve introdução. Para uma visão mais completa e profunda consulte este documento: The C Preprocessor.

O sistema Make

Video desta aula

O make é um programa utilitário Unix (criado em 1976) responsável por organizar o processo de compilação e ligação do código-fonte de um projeto, para a geração do programa executável.

O Make é uma ferramenta essencial em grandes projetos, quando há muitos arquivos envolvidos, ou quando a compilação usa muitos *flags* e opções.

O makefile

Para compilar um projeto, o programa make lê um arquivo `Makefile` ou `makefile`, que contém as definições de comandos de compilação/ligação e as regras de dependência entre os diversos arquivos do projeto.

Um arquivo `Makefile` é composto de uma ou mais regras, cuja estrutura básica é a seguinte:

```
# comentário
alvo: dependência dependência dependência ...
    receita
    receita
    ...
```

Onde:

- **alvo**: nome do objeto a ser construído (geralmente um nome de arquivo ou uma ação sobre os arquivos)
- **dependência**: alvos (ou arquivos) dos quais este alvo depende para ser construído
- **receita**: comandos necessários para compilar/construir/gerar o arquivo alvo

A endentação das linhas de comando abaixo de cada regra deve ser feita com **tabulação** (TAB) e nunca com espaços em branco.

Exemplo

Considere um programa C composto pelos seguintes arquivos:

escreva.h

```
#ifndef __ESCREVA__
#define __ESCREVA__

void escreva (char *msg) ;

#endif
```

escreva.c

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

```
void escreva (char *msg)
{
  printf ("%s", msg) ;
}
```

hello.c

```
#include "escreva.h"

int main ()
{
  escreva ("Hello, world!\n") ;
  return (0) ;
}
```

O arquivo a seguir define uma regra mínima para compilar o programa e gerar o executável `hello`:

Makefile

```
hello: hello.c escreva.c escreva.h
    gcc -Wall hello.c escreva.c -o hello
```

A regra acima significa, literalmente:

- `hello` depende de `hello.c`, `escreva.c` e `escreva.h`: se algum destes tiver sido modificado, o alvo `hello` deve ser recompilado;
- para (re)compilar o alvo `hello`, deve-se executar o comando `gcc -Wall hello.c escreva.c -o hello`.

Ao executar o comando `make hello` será gerado o executável `hello`:

```
$ ls
escreva.c  escreva.h  hello.c  Makefile

$ make
gcc hello.c escreva.c -o hello -Wall

$ ls
escreva.c  escreva.h  hello  hello.c  Makefile
```

Caso nenhum alvo seja indicado na linha de comando do `make`, por default o **primeiro alvo** definido no arquivo `Makefile` é ativado. Assim, `make` e `make hello` irão gerar o mesmo resultado, no exemplo anterior.

Uma característica essencial do sistema `make` é a verificação automática da necessidade de reconstruir os alvos, através da análise das datas dos arquivos declarados como dependências: um alvo só é reconstruído se alguma de suas dependências tiver sido modificada (ou seja, se a data de algum dos arquivos correspondentes for mais recente que a data do arquivo-alvo).

Regras de compilação e de ligação

Podem ser definidas regras separadas para compilação e ligação, agilizando a construção de sistemas com muitos arquivos:

Makefile

```
# regras de ligação
hello: hello.o escreva.o
    gcc hello.o escreva.o -o hello

# regras de compilação
hello.o: hello.c escreva.h
    gcc -c hello.c -Wall

escreva.o: escreva.c escreva.h
    gcc -c escreva.c -Wall
```

Assim, como as regras do arquivo `Makefile` acima definem que `hello` depende de `hello.o` e `escreva.o`; assim a receita da regra `hello` só será ativada caso os arquivos `hello.o` ou `escreva.o` sejam **mais recentes** que o arquivo `hello` (ou não existam). O mesmo ocorre entre `hello.o` e `hello.c`, e assim por diante.

Exemplo (o comando `touch` atualiza a data de um arquivo):

```
$ make
make: Nada a ser feito para `hello'.

$ touch hello.c
$ make
gcc -c hello.c -Wall
gcc hello.o escreva.o -o hello -Wall

$ touch escreva.c
$ make
gcc -c escreva.c -Wall
gcc hello.o escreva.o -o hello -Wall

$ touch escreva.h
$ make
gcc -c hello.c -Wall
gcc -c escreva.c -Wall
gcc hello.o escreva.o -o hello -Wall
```

Regras usuais

As seguintes regras são usualmente encontradas em arquivos `Makefile`:

- `all`: constrói todos os alvos definidos no `Makefile`; costuma ser a primeira regra, ativada por default.
- `clean`: remove arquivos temporários como os arquivos objeto (`.o`), backups de editor, etc.
- `purge`: remove os arquivos temporários e os alvos construídos, preservando somente os arquivos-fonte do projeto (também são usados os nomes `mrproper` e `distclean`).

- debug: mesmo que `all`, mas incluindo *flags* de depuração, como `-g` e `-DDEBUG`, por exemplo.
- `test`: compila e executa procedimentos de teste.

```
# regra default (1a)
all: hello

...

# remove arquivos temporários
clean:
    -rm -f *~ *.o

# remove tudo o que não for o código-fonte original
purge: clean
    -rm -f hello
```

Por default, o `make` interrompe a execução quando uma receita resulta em erro; para evitar esse comportamento, pode-se adicionar o caractere "-" no início da receita cujo erro se deseja ignorar (como nas receitas das regras `clean` e `purge` acima).

Variáveis

Variáveis podem ser criadas dentro do `Makefile` para facilitar a escrita de regras envolvendo muitos arquivos. O exemplo a seguir ilustra a sintaxe de criação e uso de variáveis:

Makefile

```
# Makefile de exemplo (Manual do GNU Make)

CFLAGS = -Wall     # flags de compilacao
LDLIBS = -lm       # bibliotecas a ligar

# arquivos-objeto
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects) $(LDLIBS)
main.o : main.c defs.h
    cc -c main.c $(CFLAGS)
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c $(CFLAGS)
command.o : command.c defs.h command.h
    cc -c command.c $(CFLAGS)
display.o : display.c defs.h buffer.h
    cc -c display.c $(CFLAGS)
insert.o : insert.c defs.h buffer.h
    cc -c insert.c $(CFLAGS)
search.o : search.c defs.h buffer.h
    cc -c search.c $(CFLAGS)
files.o : files.c defs.h buffer.h command.h
    cc -c files.c $(CFLAGS)
utils.o : utils.c defs.h
```

```
    cc -c utils.c $(CFLAGS)
clean :
    -rm -f edit $(objects)
```

Na declaração da variável `objects`, observe como quebrar linhas muito longas!

Regras podem ser usadas para alterar o valor de variáveis. No exemplo abaixo, a regra debug adiciona alguns flags à variável CFLAGS e em seguida chama a regra `all`:

```
# compila com flags de depuração
debug: CFLAGS += -DDEBUG -g
debug: all
```

Regras implícitas

O sistema Make possui um conjunto de receitas e regras implícitas para as operações mais usuais. Receitas e regras implícitas não precisam ser declaradas, o que simplifica o arquivo `Makefile`.

A geração de arquivo-objeto (`.o`) a partir do código-fonte C correspondente (`.c`) usa esta receita implícita:

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c fonte.c
```

A ligação de arquivos-objeto (`.o`) para a geração do executável usa esta receita implícita:

```
$(CC) $(LDFLAGS) arq1.o arq2.o ... $(LDLIBS) -o executavel
```

As variáveis usadas nessas regras implícitas têm os seguintes significados:

variável	**significado**	**exemplo**
CC	compilador a ser usado (por default cc)	`CC = clang`
CPPFLAGS	opções para o preprocessador	`CPPFLAGS = -DDEBUG -I/opt/opencv/include`
CFLAGS	opções para o compilador	`CFLAGS = -std=c99`
LDFLAGS	opções para o ligador	`LDFLAGS = -static -L/opt/opencv/lib`
LDLIBS	bibliotecas a ligar ao código gerado	`LDLIBS = -lpthreads -lm`

Eis como ficaria nosso `Makefile` com variáveis e receitas implícitas:

Makefile

```
CFLAGS = -Wall -g  # gerar "warnings" detalhados e infos de depuração

objs = hello.o escreva.o

# regra default (primeira regra)
all: hello

# regras de ligacao
hello: $(objs)

# regras de compilação
hello.o:   hello.c escreva.h
```

```
escreva.o: escreva.c escreva.h

# remove arquivos temporários
clean:
    -rm -f $(objs) *~

# remove tudo o que não for o código-fonte
purge: clean
    -rm -f hello
```

Uma lista de receitas e regras implícitas para várias linguagens pode ser encontrada nesta página.

Tópicos avançados

Este pequeno tutorial apenas "arranha a superfície" do sistema Make, mas é suficiente para o uso em projetos mais simples. Contudo, existem muitos tópicos avançados não tratados aqui, como:

- Variáveis automáticas
- Regras com padrões
- Condicionais
- *Wildcards*
- Makefiles recursivos
- Avaliação de regras em paralelo
- Geração automática de dependências
- ...

Esses e outros tópicos podem ser estudados em maior profundidade no Manual do GNU Make.

Depuração

Videos desta aula:

Parte 1: GDB

,

Parte 2: memória

e

Parte 3: tracing e outros

.

Depurar significa “purificar” ou “limpar”. Ao programar em C, frequentemente temos necessidade de depurar a lógica de um programa, para resolver erros de execução, de alocação/liberação de memória ou de desempenho. Esta página descreve algumas ferramentas disponíveis para tal propósito.

Preparação

O primeiro passo para a depuração de um programa executável é compilá-lo de forma a **incluir no executável os símbolos** necessários ao processo de depuração (como os nomes das variáveis e funções, referências às linhas do código fonte, etc). Isso é feito adicionando a opção `-g` ao comando de compilação.

O comando de compilação do arquivo

fatorial.c

, por exemplo, seria:

```
$ gcc -g -o fatorial fatorial.c
```

Feito isso, o programa está pronto para ser depurado com as ferramentas apresentadas a seguir.

Depuração de execução: GDB

O depurador padrão para a linguagem C no Linux é o GDB (GNU Debugger). O GDB é um depurador em modo texto com muitas funcionalidades, mas relativamente complexo de usar para os iniciantes.

Para iniciar uma depuração, basta invocar o GDB com o executável (compilado com a opção `-g`):

```
$ gdb fatorial
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
... (msgs diversas)
Lendo símbolos de fatorial...concluído.
(gdb) run
Starting program: /home/prof/maziero/fatorial
O fatorial de 4 é 24
O fatorial de 10 é 3628800
```

```
[Inferior 1 (process 24652) exited normally]
(gdb) quit
```

O *prompt* do GDB aceita diversos comandos, entre eles o *run* e o *quit*, ilustrados acima. Os comandos mais básicos disponíveis são:

comando	**ação**	**exemplos**

`run`
`r`
	inicia a execução (*run*)	`run`
		`r < dados.dat`
		`r > saida.txt`

`list` `l`	lista linhas do código-fonte	`l 23`
`b`	cria um ponto de parada (*breakpoint*)	`b 17` `b main`
`c`	continua após o ponto de parada	`c`
`s`	avança para a próxima linha de código (*step*)	`s`
`n`	avança para a próxima linha de código (*next*), sem parar dentro das funções	`n`
`p`	imprime o valor de uma variável ou expressão	`p soma` `p /x soma`
`watch`	avisa quando uma variável muda de valor	`watch i`
`disp`	mostra o valor de uma variável ou expressão a cada pausa (*display*)	`disp soma`
`set variable`	Ajusta o valor de uma variável do programa em análise	`set variable soma = 100`
`bt`	Mostra a posição atual do programa (*backtrace*), incluindo as funções ativas no momento	`bt`
`frame`	Seleciona o frame de execução a analisar (o nível de chamada de função)	`frame 3`
`^x^a`	alterna entre as interfaces padrão e NCurses	

Uma relação mais extensa de comandos pode ser encontrada neste GDB Reference Card.

O GDB proporciona uma forma fácil de localizar erros fatais de memória, como *Segmentation Fault* e outros. Basta executar o programa no GDB (usando *run*) até ocorrer o erro. Quando este ocorrer, o número da linha será informado e os valores das variáveis envolvidas podem ser inspecionados para encontrar o erro.

Para testar, use o GDB para encontrar os erros de acesso à memória neste programa: memerror.c.

Vários guias de uso do GDB podem ser encontrados nos links abaixo:

- Beej's Quick Guide to GDB
- A GDB tutorial
- Debugging with GDB
- Guide to faster, less frustrating debugging
- Using GNU's GDB debugger
- Tutorial de GDB (em português)

Como interfaces alternativas para o GDB, existe o modo *NCurses*, que pode ser ativado invocando o GDB com a opção `-tui` ou pelo comando `^x^a` (*ctrl-x ctrl-a*). Outras interfaces disponíveis em modo texto são o `cgdb`, que usa comandos similares aos do VI, e o modo GDB do EMACS.

Também existem diversas interfaces gráficas para o GDB, como o Data Display Debugger (ddd), Nemiver e Gede. O GDB também pode ser usado através de IDEs (*Integrated Development Environments*) gráficos como *Code::Blocks*, *Codelite*, *Eclipse*, *KDevelop*, *NetBeans*, etc.

Despejo de memória

A maioria dos sistemas UNIX permite salvar em um arquivo o conteúdo da memória de um programa em execução (processo), quando este é interrompido por um erro. Esse arquivo se chama `core file` (despejo de memória) e pode ser aberto por depuradores, para auxiliar na compreensão da causa do erro.

O despejo de memória é útil para identificar a causa de erros fatais em programas, sobretudo para **erros esporádicos ou difíceis de reproduzir**.

Para usar esse recurso deve-se seguir os seguintes passos (usando como exemplo o arquivo memerror.c):

1. Habilitar a geração de arquivo *core* no terminal atual:

```
$ ulimit -c unlimited
```

2. Compilar o programa com o flag de depuração (`-g`):

```
$ cc -g memerror.c -o memerror
```

3. Lançar o programa:

```
$ memerror
```

4. Quando o programa abortar por erro, será gerado um arquivo *core* no diretório corrente:

5.
```
$ ls -l
-rw-------  1 maziero maziero 262144 Nov 30 14:57 core
-rwxrwxr-x  1 maziero maziero  11104 Nov 30 14:56 memerror
-rw-rw-r--  1 maziero maziero    839 Nov 30 14:56 memerror.c
```

6. Alternativamente, pode-se **forçar o encerramento** (e a consequente geração do arquivo *core*) através de um sinal `SIGQUIT` (sinal n° 3). Esse sinal deve ser enviado ao processo através do comando `kill` (a partir de outro terminal, se necessário):

```
$ kill -3 PID
```

   onde PID é o identificador numérico do processo a ser encerrado.

7. Uma vez obtido o arquivo *core*, basta abri-lo através do depurador (GDB) e analisar seu estado:

```
$ gdb memerror core
```

Em algumas versões de Linux (Ubuntu, etc) o serviço *Apport* trata os erros fatais e impede a geração de arquivos *core*. Pode-se desligar temporariamente esse serviço através do comando `sudo service apport stop`, para obter os arquivos *core*.

Depuração de memória

Um dos problemas mais frequentes (e de depuração mais difícil) na programação em C é o uso incorreto da memória. Situações como acesso a posições inválidas de vetores ou matrizes (*buffer overflow*), uso de ponteiros não inicializados, não-liberação de memória dinâmica (*memory leaks*) podem gerar comportamentos erráticos difíceis de depurar.

Há várias ferramentas para auxiliar na depuração de problemas de memória, vistas na sequência.

Análise estática

Ferramentas de análise estática examinam o código-fonte de uma forma mais detalhada que o compilador, permitindo encontrar diversos erros que podem passar despercebidos, como índices de vetores fora da faixa válida.

Algumas ferramentas disponíveis para análise estática de código:

- `cppcheck`: verificador estático de código C/C++ (recomenda-se usar flag `--enable=all`)
- `splint`: idem

Flags do compilador

Opções do GCC para depuração de memória:

- `-fsanitize=address`: ativa o AddressSanitizer, um detector de erros de memória em tempo de execução. O código do executável é instrumentado (são adicionadas instruções) para verificar erros de acesso a posições inválidas de memória.
- `-fcheck-pointer-bounds`: ativa a verificação de limites de ponteiros.
- `-fstack-protector`: gera código adicional para verificar a integridade da pilha (*flag* habilitado por default).

As verificações adicionadas por esses *flags* são efetuadas a cada acesso à memória, por isso têm um forte impacto no desempenho e no uso de memória do executável. Então, só devem ser usadas durante o processo de desenvolvimento e nunca no produto final.

Bibliotecas de depuração

São bibliotecas que instrumentam as rotinas de alocação/liberação de memória, permitindo depurar erros relacionados ao uso de memória dinâmica, como *memory leaks*, *double free* e *use after free*.

- DMalloc
- Electric Fence

Depuradores de memória

Ferramentas como Mtrace e Valgrind realizam a análise dinâmica do programa (ou seja, durante sua execução) para encontrar erros relacionados à memória.

A ferramenta mais usada e popular é sem dúvida o **Valgrind**. Ele é particularmente útil para encontrar problemas de vazamento de memória (*memory leaks*), mas pode realizar diversas análises envolvendo memória, caches e *profiling* da execução.

Algumas opções usuais do Valgrind são:

- `--tool=toolname` : escolhe a ferramenta de análise, que pode ser:
  - `memcheck` : análise de acesso à memória (default)
  - `cachegrind` : análise de uso dos caches
  - `exp-sgcheck` : análise mais detalhada de variáveis globais e do stack
  - ...

- `--leak-check=full` : relatório detalhado sobre *memory leaks*

Eis abaixo um programa com alguns erros de memória frequentes:

errors.c

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

#define VETSIZE 100

int main()
{
  int *vet1, *vet2 ;
  int x ;

  vet1 = malloc(VETSIZE * sizeof(int)) ;
  vet2 = malloc(VETSIZE * sizeof(int)) ;

  // erro 1: acesso a uma posição fora do vetor (buffer overflow)
  vet1[VETSIZE] = 0 ;

  // erro 2: leitura de uma variável não inicializada
  if (x == 0)
    printf ("x vale zero\n") ;

  free (vet2) ;

  // erro 3: liberar duas vezes a mesma área (double free)
  free (vet2) ;

  // erro 4: usar uma área após tê-la liberado (use after free)
  vet2[0] = 0 ;

  // erro 5: a área de vet1 não foi liberada (memory leak)
}
```

Primeiro, deve-se compilar o código com a opção `-g`. Em seguida, executar o Valgrind com as opções de depuração de memória:

```
$ gcc -Wall -g errors.c -o errors
$ valgrind --leak-check=full ./errors
```

O relatório gerado pelo Valgrind pode ser extenso e deve ser analisado com atenção. A seguir destacamos os trechos do relatório relativos aos quatro erros de memória do código acima.

O trecho abaixo diz respeito ao **erro 1**:

```
==29519== Invalid write of size 4
==29519==    at 0x1086F8: main (errors.c:15)
==29519==  Address 0x522d1d0 is 0 bytes after a block of size 400 alloc'd
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086DB: main (errors.c:11)
```

- escrita de 4 bytes inválida, na função `main`, na linha 15 de `errors.c`
- essa escrita ocorreu depois do bloco de 400 bytes que foi alocado na linha 11

O trecho abaixo diz respeito ao **erro 2**:

```
==29519== Conditional jump or move depends on uninitialised value(s)
==29519==    at 0x108702: main (errors.c:18)
```

- a condicional na linha 18 depende de um valor não inicializado (pode conter lixo)

O trecho abaixo diz respeito ao **erro 3**:

```
==29519== Invalid free() / delete / delete[] / realloc()
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x108727: main (errors.c:24)
==29519==  Address 0x522d210 is 0 bytes inside a block of size 400 free'd
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x10871B: main (errors.c:21)
==29519==  Block was alloc'd at
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086E9: main (errors.c:12)
```

- `free()` inválido na linha 24
- a área de 400 bytes em questão já foi liberada na linha 21
- essa área havia sido alocada na linha 12.

O trecho abaixo diz respeito ao **erro 4**:

```
==29519== Invalid write of size 4
==29519==    at 0x10872C: main (errors.c:27)
==29519==  Address 0x522d210 is 0 bytes inside a block of size 400 free'd
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x10871B: main (errors.c:21)
==29519==  Block was alloc'd at
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086E9: main (errors.c:12)
```

- escrita de 4 bytes inválida na linha 27
- essa área de memória de 400 bytes já foi liberada na linha 21
- essa área havia sido alocada na linha 12.

O trecho abaixo diz respeito ao **erro 5**:

```
==29519== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086DB: main (errors.c:11)
```

- *memory leak*: a área de memória alocada na linha 11 não foi liberada ao encerrar o programa.

Como exercício, analise o programa memerror.c usando o Valgrind.

Tracing

Uma ferramenta útil para auxiliar na depuração de programas é o utilitário `strace`, que permite listar as chamadas de sistema efetuadas por um executável qualquer. Por não precisar de opções especiais de compilação, nem do código-fonte do executável, é uma ferramenta útil para investigar o comportamento de executáveis de terceiros. Além disso, o `strace` pode analisar processos já em execução (através da opção `-p`).

Eis um exemplo (abreviado) da execução de `strace` sobre o programa fatorial.c:

```
$ strace ./fatorial
execve("./fatorial", ["./fatorial"], 0x7fff761a4a10 /* 63 vars */) = 0
brk(NULL)                               = 0x5628852a9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=141702, ...}) = 0
mmap(NULL, 141702, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f39dc9c3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
...
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0x5628852a9000
brk(0x5628852ca000)                     = 0x5628852ca000
write(1, "O fatorial de 4 \303\251 24\n", 22O fatorial de 4 é 24
) = 22
write(1, "O fatorial de 10 \303\251 3628800\n", 28O fatorial de 10 é 3628800
) = 28
exit_group(0)                           = ?
+++ exited with 0 +++
```

De forma similar, o comando `ltrace` gera uma listagem sequencial de todas as chamadas de biblioteca geradas durante a execução de um programa:

```
$ ltrace ./fatorial
printf("O fatorial de %d \303\251 %ld\n", 4, 24O fatorial de 4 é 24
) = 22
printf("O fatorial de %d \303\251 %ld\n", 10, 36288000 fatorial de 10 é 3628800
) = 28
+++ exited (status 0) +++
```

Performance profiling

Além da depuração propriamente dita, em busca de erros, por vezes torna-se necessário analisar o comportamento temporal do programa. Para realizar essa análise pode-se utilizar o *GNU Profiler* (`gprof`). O `gprof` permite verificar:

- o tempo gasto em cada função
- o grafo de chamadas (call graph)
  - que funções são chamadas por que funções
  - que funções chamam outras funções
- quantas vezes cada função é chamada
- ...

Para realizar o *profiling* de um executável, é necessário inicialmente compilá-lo com o flag adequado (`-pg`) e em seguida executá-lo:

```
$ gcc -pg -g -o fatorial.c
$ ./fatorial
```

A execução irá gerar um arquivo binário `gmon.out`, que contém os dados de *profiling*. Esse arquivo é usado pelo utilitário `gprof` para gerar as estatísticas desejadas:

```
$ gprof fatorial gmon.out
```

O relatório de *profilling* do programa

demo-profile.c

obtido com o `gprof` pode ser encontrado

neste arquivo

. O grafo de chamadas (*call graph*) pode ser visualizado de forma gráfica através da ferramenta Gprof2dot.

Mais detalhes e opções de relatório podem ser obtidas no manual GNU gprof.

O Valgrind também permite realizar *profiling*, através de sua ferramenta interna `callgrind` e do visualizador externo KCachegrind.

Tratamento de erros

A maior parte das chamadas de sistema e funções UNIX retorna erros na forma de códigos numéricos, que são descritos nas páginas de manual das chamadas e funções (vide `man fopen` para um bom exemplo). Normalmente, uma chamada com erro retorna o valor -1 e ajusta a variável global inteira `errno` para o código do erro. Além disso, as seguintes funções podem ser úteis na interpretação dos erros:

- `assert (int expression)` : se a expressão indicada for falsa, encerra a execução com uma mensagem de erro da forma:

  ```
  assertion failed in file xxx.c, function yyy(), line zzz
  ```

- `perror (char * msg)` : imprime na saída de erro (`stderr`) a mensagem `msg` seguida de uma descrição do erro encontrado. O programa não é encerrado.
- `strerror(int errnum)` : retorna a descrição do erro indicado por `errnum`.

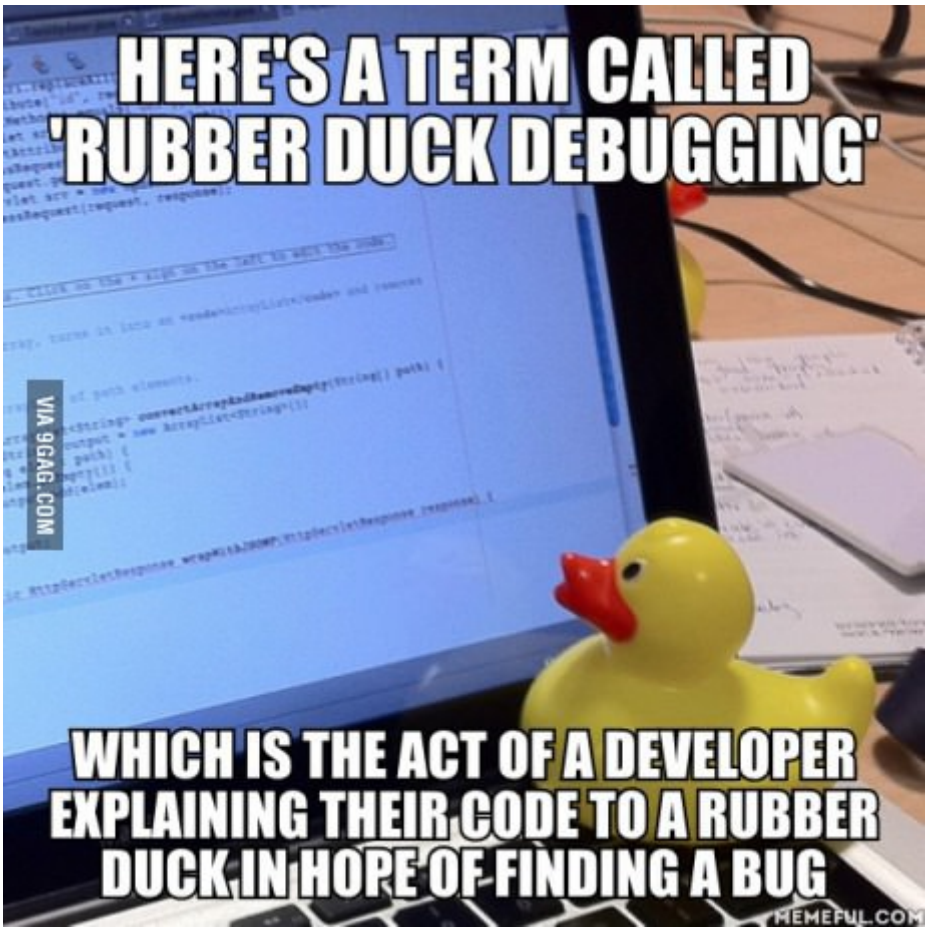
Além disso, alguns erros de execução, como operações matemáticas inválidas (divisão por zero, etc) ou violações de acesso à memória (ponteiros inválidos, etc) pode ser interceptados e tratados pelo programa, sem que seja necessária sua finalização. Esses erros geram sinais que são enviados ao processo em execução, que pode interceptá-los e tratá-los de forma a contornar o erro e continuar funcionando.

Rubber duck debugging

Se nada mais funcionar, **explique seu código a alguém** !

> *Outra técnica efetiva [de depuração] é explicar seu código a alguém. Isto vai geralmente fazer você explicar o bug para si mesmo. Às vezes bastam algumas frases, seguidas de um envergonhado "Esqueça, eu estou vendo o que está errado. Desculpe incomodá-lo". Isto funciona notavelmente bem; você pode até mesmo usar não-programadores como ouvintes. O centro de computação de uma universidade mantinha um ursinho de pelúcia perto do "help desk". Alunos com bugs misteriosos deviam explicá-los ao ursinho antes de poder*

> *falar com um assistente humano.*
>
> *The Practice of Programming*, Brian Kernighan and Rob Pike, 1999.



Outras ferramentas

- `strip`: permite remover os símbolos e código não usado de um executável ou arquivo-objeto, diminuindo consideravelmente seu tamanho (mas impedindo futuras depurações no mesmo).
- `diff`: permite comparar dois arquivos ou diretórios (recursivamente), apontando as diferenças entre seus conteúdos. Muito útil para comparar diferentes versões de árvores de código fonte.
- `patch` : permite aplicar um arquivo de diferenças (gerado pelo comando `diff`) sobre uma árvore de arquivos, modificando os arquivos originais de forma a obter uma nova árvore. Muito usado para divulgar novas versões de códigos-fonte muito grandes.
- `grep`: permite encontrar linhas em arquivos contendo uma determinada string ou expressão regular. Pode ser muito útil para encontrar trechos de código específicos em grandes volumes de código.
- `indent`: reformatador de código-fonte, aceita diversas opções de endentação automática.

A função main

Video desta aula

A função `main` é o local de início (*entry point*) da execução de um código em C. Apesar de termos usado até o momento essa função sem parâmetros, ela possui alguns parâmetros que permitem a comunicação entre o programa em C e o *shell* do sistema operacional.

Protótipos

O protótipo da função `main` depende do sistema operacional subjacente:

```
// padrão C ANSI
int main (void) ;
int main (int argc, char **argv) ;
int main (int argc, char *argv[]) ;

// sistemas UNIX-like (Linux, FreeBSD, Solaris, ...) e Windows
int main (int argc, char **argv, char **envp) ;

// sistemas Apple (MacOS, iOS)
int main (int argc, char **argv, char **envp, char **apple) ;
```

Argumentos da linha de comando

Significado dos parâmetros usuais:

- `argc`: número de argumentos na linha de comando que lançou a execução;
- `argv`: vetor de strings (`char *`) contendo os argumentos da linha de comando, finalizado por um ponteiro nulo;
- `envp`: vetor de strings (`char *`) na forma "nome=valor" contendo as variáveis de ambiente exportadas pelo *shell* que lançou a execução do programa (também finalizado por um ponteiro nulo);

O código a seguir imprime na tela os argumentos usados no lançamento do programa:

argv.c

```
#include <stdio.h>

int main (int argc, char **argv, char **envp)
{
  int i ;

  printf ("Numero de argumentos: %d\n", argc) ;
```

```
  for (i=0; i<argc; i++)
    printf ("argv[%d]: %s\n", i, argv[i]) ;

  return (0) ;
}
```

Um exemplo de compilação e execução do código acima:

```
$ gcc argv.c -o argv -Wall

$ ./argv teste 1 2 3 --help
Numero de argumentos: 6
argv[0]: ./argv
argv[1]: teste
argv[2]: 1
argv[3]: 2
argv[4]: 3
argv[5]: --help
```

Funções específicas

Para ler e tratar mais facilmente as opções da linha de comando informadas por *argc/argv*, sugere-se usar funções já prontas para isso, como `getopt` ou `arg_parse` (link)

Eis um exemplo de uso da função `getopt` para a leitura de opções da linha de comando, adaptado do manual da GNU-LibC:

options.c

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

int main (int argc, char **argv)
{
  int flag_a = 0;
  int flag_b = 0;
  char *value_c = NULL;
  int option;

  opterr = 0;

  // options:  -a, -b, -c value (defined by "abc:")
  while ((option = getopt (argc, argv, "abc:")) != -1)
    switch (option)
      {
      case 'a':      // option -a was set
        flag_a = 1;
        break;
      case 'b':      // option -b was set
        flag_b = 1;
        break;
```

```
      case 'c':      // option -c was set with value
        value_c = optarg;
        break;
     default:
        fprintf (stderr, "Usage: %s -a -b -c value\n", argv[0]);
    exit (1) ;
      }

  printf ("flag_a = %d, flag_b = %d, value_c = %s\n",
          flag_a, flag_b, value_c);

  return 0;
}
```

Status de saída

Outro canal de interação importante entre o programa C e o sistema operacional é o valor de retorno da função `main`, que é devolvido ao SO após a execução na forma de um *status* de saída (*exit status*).

retval.c

```
#include <stdio.h>

int main (int argc, char **argv, char **envp)
{
  return (14) ;
}
```

O *status* de saída de um processo pode ser consultado no terminal UNIX (shell Bash) através da variavel `$?` disponível no shell:

```
$ gcc retval.c -o retval -Wall
$ ./retval
$ echo $?
14
```

O status de saída também pode ser usado em scripts do shell:

testa-retorno.sh

```
#!/bin/sh

if retval
then
  # exit status is zero
  echo "true"
else
  # exit status is NOT zero
  echo "false"
fi
```

Exercícios

1. Escrever um programa que recebe uma lista de parâmetros na linha de comando. Ele não escreve nada na saída, mas devolve como *status* de saída o número de parâmetros que iniciam com o caractere "`-`".
2. Escrever um programa `exists`, que recebe um nome (ou caminho) de arquivo na linha de comando e devolve, no *status* de saída, 0 se o arquivo existe ou 1 se ele não existe.
3. Escrever um programa para listar as variáveis de ambiente recebidas pelo programa (parâmetro `envp` da função `main`); essas variáveis podem ser consultadas no terminal (*shell*) através do comando `env`.

Tipos enumerados

Video desta aula

Enumerações, ou tipos enumerados, são tipos de dados definidos pelo programador. Uma variável de tipo enumerado pode assumir um valor dentre um conjunto fixo de valores possíveis previamente definido.

Um tipo enumerado é visto como um caso especial de tipo inteiro. Variáveis de tipos enumerados são implementadas pelo compilador usando valores inteiros, mas isso é transparente para o programador.

Declaração e uso

Um exemplo simples de declaração de tipo enumerado:

```
// definição do tipo enumerado
enum Color_t { BLACK, BLUE, GREEN, RED, YELLOW, WHITE } ;

// declaração de variável do tipo enum Color_t
enum Color_t c1;
```

A definição do tipo enumerado também pode ser feita usando `typedef`:

```
typedef enum { BLACK, BLUE, GREEN, RED, YELLOW, WHITE } Color_t ;

Color_t c1;
```

Variáveis enumeradas podem ser usadas como variáveis inteiras:

```
// atribuição
c1 = GREEN ;

// uso
if (c1 == RED) { ... }

// os valores enumerados têm uma ordem definida
if (c1 >= BLUE && c1 <= YELLOW)  { ... }

// variáveis enumeradas podem ser incrementadas
c1 = BLACK ;
while (c1 <= WHITE)
{
  ...
  c1++ ;
}

// um laço for enumerado
for (c1 = BLACK; c1 <= WHITE; c1++)
```

```
{
  ...
}
```

Por default, o primeiro valor de um tipo enumerado vale 0 (zero), o segundo valor vale 1 e assim por diante. Esses valores podem ser alterados se isso for conveniente:

```
typedef enum {
  BLACK = 1,
  BLUE,
  GREEN,
  RED = 10,
  YELLOW,
  WHITE
} Color_t ;
```

Os valores inteiros associados à enumeração são:

valor simbólico	valor original	valor com atribuição
BLACK	0	1
BLUE	1	2
GREEN	2	3
RED	3	10
YELLOW	4	11
WHITE	5	12

Tipos enumerados podem ser usados para tornar o código mais legível e também diminuir erros devidos a valores imprevistos em variáveis. Um exemplo clássico de uso de tipos enumerados é a definição de um tipo booleano em C:

```
typedef enum
{
  FALSE = 0,
  False = 0,
  false = 0,
  TRUE  = 1,
  True  = 1,
  true  = 1
} bool_t ;

bool_t flag ;
```

Imprimindo tipos enumerados

Internamente, um tipo enumerado é representado por um inteiro, os valores simbólicos (WHITE, RED, ...) são descartados durante a compilação. Por isso, **não é possível imprimir o valor simbólico** diretamente, apenas seus valores internos:

print_enum.c

```
#include <stdio.h>

// tipo "Cor"
```

```c
typedef enum { BLACK, BLUE, GREEN, RED, YELLOW, WHITE } Color_t ;

int main ()
{
  Color_t c1;

  // ...
  c1 = BLUE ;

  printf ("Cor %s\n", c1) ;    // errado!
  printf ("Cor %d\n", c1) ;    // correto!
}
```

Uma forma simples de contornar esse problema consiste em usar um vetor de nomes:

color-array.c

```c
#include <stdio.h>

// tipo "Cor"
typedef enum { BLACK, BLUE, GREEN, RED, YELLOW, WHITE } Color_t ;

// nome das cores
char *color_name[] = {"black", "blue", "green", "red", "yellow", "white" } ;

int main ()
{
  Color_t c1;

  // ...
  c1 = BLUE ;

  printf ("Cor %s\n", color_name[c1]) ;
}
```

A solução acima só funciona se a enumeração usar os valores numéricos *default* (0, 1, 2, 3, ...).

Uma solução mais robusta consiste em usar uma estrutura `switch`:

color-switch.c

```c
#include <stdio.h>

// tipo "Cor"
typedef enum { BLACK, BLUE, GREEN, RED, YELLOW, WHITE } Color_t ;

// define o nome da cor
char* color_name (Color_t c)
{
  switch (c)
  {
    case BLACK   : return ("black")  ;
```

```
    case BLUE    : return ("blue")   ;
    case GREEN   : return ("green")  ;
    case RED     : return ("red")    ;
    case YELLOW  : return ("yellow") ;
    case WHITE   : return ("white")  ;
    default      : return ("") ;
  }
}

int main ()
{
  Color_t c1;

  // ...
  c1 = BLUE ;

  printf ("Cor %s\n", color_name(c1)) ;
}
```

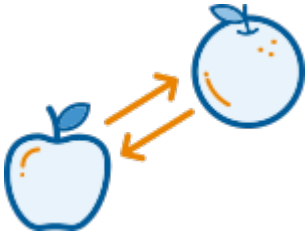
Exercícios

1. Escreva um programa em C capaz de classificar os alunos de uma determinada matéria de acordo com sua situação. O programa deve ser capaz de:
    1. Ler os dados do aluno referentes a sua nota na disciplina e sua frequência.
    2. Classificar o aluno pelas enumerações `Aprovado`, `Exame Final` ou `Reprovado` de acordo com os dados obtidos.
2. Escreva um programa em C capaz de receber um número e classificá-lo, por meio de tipos enumerados, entre número par ou impar; positivo, negativo ou zero; palíndromo ou não-palíndromo; múltiplo de 10 ou não múltiplo;
3. Escreva um programa em C que leia uma determinada sequência de caracteres, que seja capaz de classificar cada um deles entre `letra`, `pontuação` e `caractere especial` e guarde os dados em uma estrutura. Para o caso de letras, o programa deve ainda classificá-las, por meio de enumerações, entre `maiúsculas` e `minúsculas`; `vogais` e `consoantes`.

Conversão de tipos

Video desta aula

Muitas vezes, expressões lógicas/aritméticas envolvem valores de tipos distintos, como `char`, `int`, `float` e `double`, que devem ser convertidos durante o cálculo.



Algumas conversões são efetuadas implicitamente pelo compilador, sem riscos para a integridade dos dados manipulados. Todavia, certas conversões podem levar a resultados errados ou à perda de informação e precisam ser "forçadas" pelo programador, através de conversões explícitas.

Conversão implícita

Quando uma operação lógica/aritmética envolve dois operandos de tipos distintos, o compilador primeiro os converte para um único tipo, antes de avaliar a operação. Nos caso de tipos numéricos, essa conversão é denominada **promoção automática**,

Na promoção automática, o tipo de menor tamanho (com menos bytes) é automaticamente convertido ("promovido") para o tipo de maior tamanho (com mais bytes).

A hierarquia de tipos considerada para a promoção automática é a seguinte (os tipos `unsigned` estão no mesmo nível de hierarquia que seus equivalentes `signed`):

```
char < short < int < long < long long < float < double < long double
```

A regra de promoção automática é aplicada separadamente **A CADA OPERAÇÃO** da expressão lógica/aritmética.

O código a seguir mostra alguns exemplos de promoção automática:

```
char   c = 'A' ;
int    i = 3 ;
short  s = 31 ;
float  f = 31.8 ;
double d ;

i = c + 30 ;    //  o valor de c (65) é convertido de char para int

d = i * f + s ; // ao avaliar i*x,       i é promovido de int a float
                // ao avaliar float + k, k é promovido de short a float
```

A conversão implícita (ou automática) é realizada para as operações aritméticas (`+ - * / %`), relacionais (`< <= > >= == !=`) e com bits (`& | ^`).

Os operadores de atribuição (=, +=, ...) também recebem uma conversão implícita de tipo, para compatibilizar o valor calculado (lado direito da atribuição) com a variável que irá recebê-lo (lado esquerdo).

Erros de conversão

A regra de promoção automática é aplicada a cada operação da expressão lógica/aritmética. Por isso, deve ser tomado cuidado em expressões mais complexas. O código abaixo traz um exemplo dos riscos envolvidos:

erro-conversao.c

```c
#include <stdio.h>

int main ()
{
  int i   = 100 ;
  int j   = 6 ;
  float x = 6 ;
  float y ;

  y = i / j * x ;              // (int / int) * float  -> int * float
  printf ("y vale %f\n", y) ;  //  ^^^^^^^^^ divisão inteira!

  y = x * i / j ;              // (float * int) / int
  printf ("y vale %f\n", y) ;  // resultado correto (vide abaixo)
}
```

Ao avaliar primeiro a operação `i/j`, ocorre um erro de arredondamento:

```
$ ./a.out
y vale 96.000000
y vale 100.000000
```

As expressões são usualmente avaliadas da esquerda para a direita, mas isso depende da precedência dos operadores, além do compilador e do nível de otimização usado. Por isso, a ordem de avaliação das operações em uma expressão envolvendo tipos distintos deve ser explicitada usando **parênteses**:

```c
y = (x * i) / j ;            // (float * int) / int
printf ("y vale %f\n", y) ;  // resultado correto sempre
```

A conversão implícita na atribuição também pode provocar perda de informação. O exemplo abaixo mostra algumas dessas situações:

perdas.c

```c
#include <stdio.h>

int main ()
{
  char   c ;
  int    i ;
  long   l ;
  float  f ;
  double d ;

  c = 'A' ;
  i = c ;     // ok, pois int > char
```

```
  printf ("c = %d, i = %d\n", c, i) ;

  i = 34 ;
  f = i ;     // ok, pois float > int
  printf ("i = %d, f = %f\n", i, f) ;

  l = 214748364347243 ;
  f = l ;     // perda de precisão
  printf ("l = %ld, f = %f\n", l, f) ;

  l = 214748364347243 ;
  d = l ;     // ok, precisão suficiente
  printf ("l = %ld, d = %lf\n", l, d) ;

  f = 451.28 ;
  i = f ;     // parte fracionária é truncada
  printf ("f = %f, i = %d\n", f, i) ;

  d = 3.141592653589793264 ;
  f = d ;     // perda de precisão
  printf ("d = %.15f, f = %.15f\n", d, f) ;

  l = 12345677890 ;
  i = l ;     // perda dos bits mais significativos
  printf ("l = %ld, i = %d\n", l, i) ;
}
```

Observe que a precisão de um `float` é menor que a de um `int` a partir de $2^{23}+1$, pois a mantissa do `float` tem apenas 23 bits (padrão IEEE 754). Assim, no caso de um valor inteiro muito grande ($> 2^{23}$), a conversão do mesmo em `float` implica em perda de precisão.

Conversão explícita

Em alguns casos, é necessário forçar a conversão de tipos de dados, para que uma expressão seja avaliada da forma correta.

Por exemplo:

media-erro.c

```
#include <stdio.h>

int main ()
{
  int soma, num ;
  float media ;

  soma = 10 ;
  num  = 4 ;

  media = soma / num ;
```

```
  printf ("media = %f\n", media) ;
}
```

No caso acima, a expressão `soma / num` será avaliada como `int / int`, resultando em uma divisão inteira e consequente perda de precisão. Para gerar o resultado correto, a expressão deve ser avaliada como `float`. Isso pode ser obtido de duas formas:

- adicionando um elemento neutro de tipo `float` à expressão;
- forçando a avaliação de `soma` ou `num` como float (*type casting*).

media.c

```
#include <stdio.h>

int main ()
{
  int soma, num ;
  float media ;

  soma = 10 ;
  num  = 4 ;

  media = soma / num ;           // errado, perda de precisão
  printf ("media = %f\n", media) ;

  media = 1.0 * soma / num ;      // soma é "promovida" a float
  printf ("media = %f\n", media) ;

  media = (float) soma / num ;        // soma é avaliada como float (casting)
  printf ("media = %f\n", media) ;

  media = soma / (float) num ;        // num é avaliado como float (casting)
  printf ("media = %f\n", media) ;
}
```

A avaliação forçada de um valor ou variável para um tipo específico usando o prefixo `(type)`, como no exemplo acima, é chamada *type casting*. Essa operação é muito usada, sobretudo na avaliação de ponteiros.

Conversão de ponteiros

Uma operação frequente em C é a conversão de tipos de ponteiros. Muitas funções importantes, como `qsort` e `bsearch`, usam ponteiros genéricos `void*` como parâmetros de entrada, para poder receber dados de diversos tipos.

Como ponteiros para `void` não apontam para nenhum tipo válido de dado, eles não podem ser desreferenciados (ou seja, não é possível acessar diretamente os dados que eles apontam). Por isso, ponteiros para `void` precisam ser convertidos em ponteiros para algum tipo válido antes de serem desreferenciados.

O exemplo a seguir mostra como ponteiros `void*` são convertidos em `int*`, dentro da função `compara_int(a,b)`:

qsort.c

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

#define VETSIZE 10

int vetor[VETSIZE] ;

// compara dois inteiros apontados por "a" e "b"
int compara_int (const void* a, const void* b)
{
  int *pa, *pb ;

  pa = (int*) a ;  // "vê" a como int*
  pb = (int*) b ;  // idem, b

  if (*pa > *pb) return  1 ;
  if (*pa < *pb) return -1 ;
  return 0 ;
}

int main ()
{
  int i ;

  // preenche o vetor de inteiros com aleatórios
  for (i = 0; i < VETSIZE; i++)
    vetor[i] = random() % 1000 ;

  // escreve o vetor
  for (i = 0; i < VETSIZE; i++)
    printf ("%d ", vetor[i]) ;
  printf ("\n") ;

  // ordena o vetor (man qsort)
  // Protótipo: int (*compara_int) (const void *, const void *)
  qsort (vetor, VETSIZE, sizeof (int), compara_int) ;

  // escreve o vetor
  for (i = 0; i < VETSIZE; i++)
    printf ("%d ", vetor[i]) ;
  printf ("\n") ;
}
```

Conversão de/para strings

No caso específico de strings, a conversão destas para outros tipos é efetuada através de funções específicas:

```c
#include <stdlib.h>

// string to float, int, long, long long
double    atof  (const char *str) ;
```

```
int       atoi  (const char *str) ;
long      atol  (const char *str) ;
long long atoll (const char *str) ;

// string to double, float, long double, com teste de erro
double strtod (const char *nptr, char **endptr) ;
float  strtof (const char *nptr, char **endptr) ;
long   double strtold (const char *nptr, char **endptr) ;
```

No sentido *número → string*, a forma mais simples de converter um dado de qualquer tipo para *string* é usando a função `sprintf`, que formata e "imprime" o dado em uma *string*, de forma similar ao que a função `printf` realiza na saída padrão:

float2string.c

```
#include <stdio.h>

int main ()
{
  char buffer[256] ;
  float x ;

  x = 32.4 / 7 ;

  sprintf (buffer, "%5.4f", x) ; // "imprime" x na string buffer

  printf ("%s\n", buffer) ;

  return 0 ;
}
```

Leitura complementar

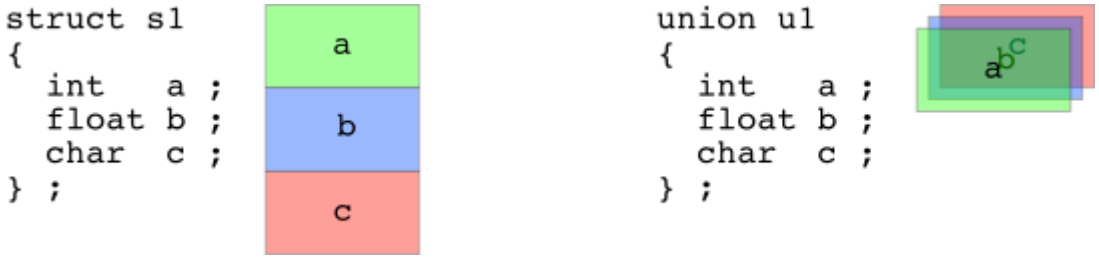
- Type Conversions, C in a Nutshell.

Uniões

Video desta aula

Uma **união** (`union`) é um tipo especial de estrutura (`struct`) no qual todos os campos internos são sobrepostos e ocupam a mesma posição na memória. Com isso, uma mesma posição na memória pode ser "vista" de diversas formas, conforme o campo da união que for acessado.

A figura a seguir ilustra a diferença entre `struct` e `union`:



Uniões são uma forma eficiente de armazenar valores de diferentes tipos em uma mesma posição de memória. Obviamente, somente um valor pode ser armazenado a cada instante. A quantidade de memória ocupada por uma união corresponde ao tamanho de seu **maior campo**.

No exemplo abaixo, uma união permite "enxergar" os bytes individuais de um inteiro sem necessidade de operações de bits ou aritmética de ponteiros:

union.c

```c
#include <stdio.h>

// guarda um inteiro OU um vetor de 4 bytes
typedef union
{
  int value ;
  unsigned char byte[sizeof(int)] ;
} intParts ;

int main ()
{
  intParts a ;
  int i ;

  a.value = 653459 ;

  printf ("%d: ", a.value) ;

  for (i=0; i< sizeof(int); i++)
    printf ("%02x ", a.byte[i]) ;
  printf ("\n") ;
}
```

Os campos `a.value` e `a.byte[]` estão alocados da seguinte forma na memória:

addr	***addr+1***	***addr+2***	***addr+3***
a.value (4 bytes)			
a.byte[0]	a.byte[1]	a.byte[2]	a.byte[3]

Uniões podem ser usadas para armazenar valores de diversos tipos em uma única locação de memória. Por exemplo, a união a seguir pode ser usada para armazenar números de diversos tipos:

```
typedef union
{
  short  shortVal ;
  int    intVal ;
  long   longVal ;
  float  floatVal ;
  double doubleVal ;
} numeric_t ;
```

Entretanto, como saber qual o tipo do último valor armazenado, ou seja, qual o valor corrente?

Se armazenarmos um `int` e tentarmos ler um `float` teremos um valor errado, pois os valores são lidos byte a byte diretamente da área de memória da união, sem conversões.

union-error.c

```
#include <stdio.h>

typedef union
{
  short  shortVal ;
  int    intVal ;
  long   longVal ;
  float  floatVal ;
  double doubleVal ;
} numeric_t ;

int main ()
{
  numeric_t a ;

  a.shortVal = 741 ;
  printf ("short : %d\n", a.shortVal) ;
  printf ("float : %f\n\n", a.floatVal) ;

  a.floatVal = 327.5432 ;
  printf ("float : %f\n", a.floatVal) ;
  printf ("short : %d\n\n", a.shortVal) ;

  a.doubleVal = 327.5432 ;
  printf ("double: %lf\n", a.doubleVal) ;
  printf ("float : %f\n", a.floatVal) ;
  printf ("short : %d\n", a.shortVal) ;
  return 0 ;
}
```

Para resolver esse problema pode ser usada uma estrutura contendo a união e uma variável que indique o tipo do último valor armazenado:

union-type.c

```c
#include <stdio.h>

typedef struct
{
  union        // ATTENTION: "anonymous" union
  {
    short  shortVal ;
    int    intVal ;
    long   longVal ;
    float  floatVal ;
    double doubleVal ;
  } ;
  enum { SHORT, INT, LONG, FLOAT, DOUBLE } type ;
} numeric_t ;

// imprime tipo numérico
void print_num (numeric_t n)
{
  switch (n.type)
  {
    case SHORT  : printf ("%d", n.shortVal)   ; break ;
    case INT    : printf ("%d", n.intVal)     ; break ;
    case LONG   : printf ("%ld", n.longVal)   ; break ;
    case FLOAT  : printf ("%f", n.floatVal)   ; break ;
    case DOUBLE : printf ("%lf", n.doubleVal) ; break ;
    default     : printf ("NaN") ;
  }
}

int main ()
{
  numeric_t a ;

  a.shortVal = 119 ;
  a.type = SHORT ;
  print_num (a) ;
  printf ("\n") ;

  a.longVal = 3451212796756 ;
  a.type = LONG ;
  print_num (a) ;
  printf ("\n") ;

  a.doubleVal = 3.141592653589793 ;
  a.type = DOUBLE ;
  print_num (a) ;
  printf ("\n") ;
}
```

O exemplo acima traz um exemplo de **união anônima**, ou seja, sem nome. Nesse caso, os membros da união são considerados como membros da estrutura externa que contém a união. Estruturas também podem ser anônimas.

A linguagem C respeita a ordem de declaração dos campos de uma estrutura, ou seja, eles são colocados na memória na mesma ordem em que são declarados.

Isso permite outro exemplo interessante do uso de uma união, no qual as moedas podem ser acessadas com nomes individuais ou como elementos de um vetor:

```
typedef union {
  struct
  {
    int quarter;
    int dime;
    int nickel;
    int penny;
  };
  int coins[4];
} Coins_t ;

Coins_t a ;

a.dime = 34 ;          // são operações equivalentes
a.coins[1] = 34 ;
```

Outro exemplo interessante é a definição de números reais segundo o padrão IEEE 754. Ele permite acessar o valor real ou suas partes (mantissa, expoente e sinal) separadamente:

```
// extraído (e simplificado) de /usr/include/x86_64-linux-gnu/ieee754.h

union ieee754_float
{
  float f;

  /* This is the IEEE 754 single-precision format.  */
  struct
  {
    #if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int negative:1;
    unsigned int exponent:8;
    unsigned int mantissa:23;
    #endif                          /* Big endian.  */
    #if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int mantissa:23;
    unsigned int exponent:8;
    unsigned int negative:1;
    #endif                          /* Little endian.  */
  } ieee;
};
```

Exercícios

1. Utilizando uma única variável `union`, crie uma função que receba um inteiro e calcule seu quadrado, em seguida, receba um caractere e, caso maiúsculo, imprima minúsculo, caso minúsculo, imprima

maiúsculo, e por último, receba uma string de no máximo 8 caracteres e imprima seu inverso.
2. Variáveis de 32 bits do tipo `int` podem representar valores entre −2,147,483,647 e +2,147,483,647, enquanto variáveis de 32 bits do tipo `unsigned int` podem representar valores entre 0 e +4,294,967,295. Crie um programa que receba um valor negativo do tipo `int` e mostre qual o valor resultante da conversão para o tipo `unsigned int`.
3. Utilizando unions, crie um programa capaz de receber um determinado valor e calcular o módulo de 256 desse valor (dica: utilize `char[sizeof(int)]`).

Operações com bits

Video desta aula

Neste módulo são abordadas técnicas para acessar e manipular bits. Elas são úteis para armazenar grandes quantidades de informação simples em pouco espaço, como vetores de flags ou inteiros com faixas de valores pequenas. O acesso a bits individuais também é útil em operações de baixo nível, envolvendo registradores, pacotes de rede ou portas de entrada/saída.

Bit fields

Ao construir programas que manipulem muitos dados, é importante escolher o tipo mais adequado para cada variável. Por exemplo, podemos definir uma estrutura de dados para armazenar datas/horas da seguinte forma:

```
typedef struct
{
  unsigned int year, month, day, hour, min, sec ;
} date_t ;
```

Usando essa estrutura, cada variável do tipo `date_t` ocupa 24 bytes (6 inteiros de 4 bytes).

No entanto, usar `int` para os campos da estrutura é um desperdício, pois os valores armazenados são pequenos. Podemos usar tipos inteiros menores, como `char` e `short`:

```
typedef struct
{
  unsigned short year ;
  unsigned char  month, day ;
  unsigned char  hour, min, sec ;
} date_t ;
```

Em máquinas com arquitetura de 32 ou 64 bits, cada variável to tipo `date_t` ocupará 8 bytes de memória:

- 5 bytes para os 5 `unsigned char`
- 2 bytes para o `unsigned short`
- 1 byte de *padding* (“enchimento”)

O *padding* é necessário porque as variáveis inteiras deve estar “alinhadas” na memória, ou seja, seu primeiro byte deve estar em um endereço par (*word-size* = 2 bytes), para ser lido corretamente pelo processador. O alinhamento se aplica às variáveis e aos campos internos das estruturas.

Entretanto, há espaço para economizar mais memória, pois os campos da estrutura não precisam usar toda a faixa de valores oferecida por seus tipos:

campo	faixa	bits necessários
year	0 ... 4000[1)]	12
mon	0 ... 11	4
day	0 ... 31	5
hour	0 ... 23	5
min	0 ... 59	6

campo	faixa	bits necessários
sec	0 ... 59	6
TOTAL		38

A linguagem C permite definir variáveis inteiras com um número específico de bits **dentro de structs**, através de uma funcionalidade chamada *bitfield*:

```
typedef struct
{
  unsigned short year:12 ;
  unsigned char  month:4 ;
  unsigned char  day:5 ;
  unsigned char  hour:5 ;
  unsigned char  min:6 ;
  unsigned char  sec:6 ;
} date_t ;
```

Essa nova estrutura demanda 38 bits, ou 4,75 bytes. Devido à necessidade de usar bytes inteiros e do alinhamento, na realidade a estrutura ocupa 6 bytes de memória, ou seja, 2 bytes a menos que no caso anterior.

O código abaixo apresenta os tamanhos das três estruturas:

struct-size.c

```
#include <stdio.h>

typedef struct
{
  unsigned int year, month, day, hour, min, sec ;
} date_t1 ;

typedef struct
{
  unsigned short year ;
  unsigned char  month, day ;
  unsigned char  hour, min, sec ;
} date_t2 ;

typedef struct
{
  unsigned short year:12 ;
  unsigned char  month:4 ;
  unsigned char  day:5 ;
  unsigned char  hour:5 ;
  unsigned char  min:6 ;
  unsigned char  sec:6 ;
} date_t3 ;

int main ()
{
  printf ("date_t1 ocupa %ld bytes\n", sizeof (date_t1)) ;
  printf ("date_t2 ocupa %ld bytes\n", sizeof (date_t2)) ;
  printf ("date_t3 ocupa %ld bytes\n", sizeof (date_t3)) ;
}
```

Bitfields são muito úteis quando é necessário ler ou manipular bits individuais na memória. Uma aplicação frequente é no acesso a estruturas de dados de baixo nível, em drivers de acesso ao hardware. Por exemplo, o *struct* abaixo representa um registrador de 32 bits da interface de um controlador de disco rígido:

```
struct DISK_REGISTER  {
     unsigned ready:1 ;
     unsigned error_occured:1 ;
     unsigned disk_spinning:1 ;
     unsigned write_protect:1 ;
     unsigned head_loaded:1 ;
     unsigned error_code:8 ;
     unsigned track:9 ;
     unsigned sector:5 ;
     unsigned command:5 ;
} ;
```

Outro exemplo muito interessante de uso de *bitfields* pode ser encontrado no arquivo `ieee754.h` do código-fonte do Linux. Esse arquivo define a estrutura em memória dos números de ponto flutuante conforme o padrão IEEE 754.

Formato do `float`:

- sinal (1 bit)
- expoente (8 bits)
- mantissa (23 bits)

```
// extraído (e simplificado) de /usr/include/x86_64-linux-gnu/ieee754.h

union ieee754_float
{
  float f;

  /* This is the IEEE 754 single-precision format.  */
  struct
  {
    #if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int negative:1;
    unsigned int exponent:8;
    unsigned int mantissa:23;
    #endif                          /* Big endian.  */
    #if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int mantissa:23;
    unsigned int exponent:8;
    unsigned int negative:1;
    #endif                          /* Little endian.  */
  } ieee;
};
```

Cuidados a tomar no uso de *bitfields*:

- elementos de *bitfield* não podem ser endereçados por ponteiros, pois podem não começar no início de um byte de memória;
- muitos compiladores limitam o tamanho de um bitfield ao tamanho máximo de um inteiro (16, 32 ou 64 bits);
- vetores de *bitfields* não são permitidos.
- o uso de *bitfields* pode tornar o código não-portável entre máquinas com configuração little/big endian distintas.

Acesso a bits individuais

Para testar um bit específico em uma variável inteira, podemos efetuar um AND bit a bit entre essa variável e uma máscara de bits (*bitmask*), na qual somente o bit a ser testado é verdadeiro.

Por exemplo: para verificar se o 4° bit de `value` está ativo, usa-se a máscara `0x8`:

```
if ( value & 0x8 )        // 0x8 = 0000 1000 em binário
{
   ...
}
```

Para ativar um bit específico, efetua-se um OR bit-a-bit entre a variável e a máscara de bits correspondente.

Por exemplo: para ativar o 3° bit de `value`, usa-se a máscara `0x4`:

```
value = value | 0x4 ;      // 0x4 = 0000 0100 em binário
value |= 0x4 ;           // versão compacta
```

Similarmente, para desativar o 3° bit da variável `value`:

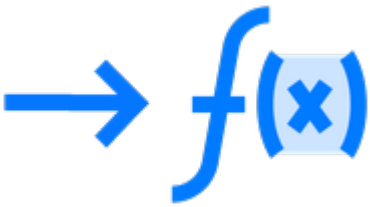
```
value = value & ~ 0x4 ;     // explique!
```

A máscara de bits também pode ser gerada por deslocamentos:

Ponteiros para funções

Video desta aula

Em C, uma função é vista como uma referência (ou endereço) para uma área de memória onde se encontra seu código. Por isso, o identificador de uma função pode ser visto como um ponteiro. Funções podem ser acessadas usando ponteiros, de forma similar às variáveis.



Declaração e uso

Declarar um ponteiro para uma função é relativamente simples, apesar da sintaxe assustar à primeira vista. O código abaixo declara um ponteiro `fp` para uma função que tem um parâmetro inteiro e não retorna nada:

```
// uma função com protótipo "void name (int)"
void f (int)
{
  ...
}

// um ponteiro para funções com protótipo "void name (int)"
void (*fp) (int) ;
```

Observe que os parênteses envolvendo `*fp` são necessários. Caso sejam omitidos, a declaração acima muda completamente de sentido:

```
void * fp (int) ; // protótipo de função retornando ''void*''
```

O uso de ponteiros para funções é simples:

funcptr.c

```
#include <stdio.h>

void inc (int *n)
{
  (*n)++ ;
}

int main ()
{
  void (*fp) (int *) ;     // function pointer

  fp = inc ;          // fp points to inc

  int a = 0 ;
  printf ("a vale %d\n", a) ;

  inc(&a) ;         // normal call
  printf ("a vale %d\n", a) ;
```

```
  fp(&a) ;          // call using the function pointer
  printf ("a vale %d\n", a) ;
}
```

Sua execução resulta em:

```
a vale 0
a vale 1
a vale 2
```

Obviamente, um ponteiro de função só pode apontar para funções que tenham o mesmo protótipo (assinatura) com o qual o ponteiro foi declarado.

Caso necessário, pode-se fazer *type casting* de ponteiros para funções.

Funções como parâmetros

Como uma variável pode apontar para uma função, então funções podem ser usadas como parâmetros de outras funções. O código a seguir ilustra como fazer isso:

funcparam.c

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

// aplica a função "func" aos caracteres de str
void aplica (int (func) (int), char *str)
{
  for (int i=0; str[i]; i++)
    str[i] = func (str[i]) ;
}

// se c for uma vogal, devolve '-'
int tira_vogal (int c)
{
  switch (c)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U': return ('-') ;
    default : return (c) ;
   }
```

```
}

int main ()
{
  char frase[128] ;

  strcpy (frase, "Uma frase com MAIUSCULAS e minusculas") ;
  printf ("Frase: %s\n", frase) ;

  aplica (toupper, frase) ;
  printf ("Frase: %s\n", frase) ;

  aplica (tolower, frase) ;
  printf ("Frase: %s\n", frase) ;

  aplica (tira_vogal, frase) ;
  printf ("Frase: %s\n", frase) ;
}
```

Resultado da execução:

```
Frase: Uma frase com MAIUSCULAS e minusculas
Frase: UMA FRASE COM MAIUSCULAS E MINUSCULAS
Frase: uma frase com maiusculas e minusculas
Frase: -m- fr-s- c-m m---sc-l-s - m-n-sc-l-s
```

Uma função também pode retornar um ponteiro de função. Neste caso, a sintaxe da declaração da função pode ficar complexa, sobretudo se a função tiver parâmetros e a função retornada também.

Exercícios

1. A função `qsort` (*man 3 qsort*) aplica o algoritmo *QuickSort* a um vetor de dados de um tipo definido pelo usuário (int, float, struct, ...). Para ser genérica, essa função depende de uma função externa para comparar os elementos do vetor. Escreva um programa que a) crie um vetor de 100 inteiros aleatórios e ordene esse vetor usando a função `qsort`.
2. Idem, para um vetor de tipo `double`.
3. Idem, para um vetor de *structs* (ordenar por um dos campos do *struct*).

Bibliotecas para C

Video desta aula

Existe uma grande quantidade de bibliotecas disponíveis para a linguagem C, algumas delas mais genéricas e muitas outras construídas para áreas específicas, como o processamento de imagens, serviços de rede, etc.

Esta página visa apresentar algumas bibliotecas para C que podem ser úteis para o desenvolvimento de programas mais complexos.

Uso geral

LibC

A biblioteca padrão C (C Standard Library) ou simplesmente LibC contém a maioria das funções básicas da linguagem C como `printf`, `scanf`, `fopen` e muitas outras.

A LibC oferece funções para:

- interface com o sistema operacional
  - alocação de memória (`malloc`, ...)
  - acesso a arquivos (`fopen`, ...)
  - acesso a *streams* (`printf`, `scanf`, `getchar`, ...)
- manipulação de caracteres e strings (normais ou multi-bytes)
- ordenação e busca (`qsort`, `bsearch`)
- operações matemáticas (exponenciação, raiz, trigonometria, logaritmos)
- conversões numéricas
- geração de números aleatórios
- operações com números complexos
- manipulação de datas e horas
- ...

Uma extensão da biblioteca padrão C específica para sistemas operacionais no padrão POSIX for definido como POSIX C Library. Além das funções da LibC padrão, ela também implementa operações específicas de sistema operacional, como:

- operações de rede
- tratamento de sinais e eventos do SO
- operações de entrada/saída assíncronas
- filas de mensagens
- semáforos
- *threads*
- ...

Como o próprio nome diz, a biblioteca **padrão** está disponível por default na grande maioria dos sistemas operacionais que suportam a linguagem C. Sistemas UNIX como o Linux e o FreeBSD usam geralmente a implementação da LibC construída pelo projeto GNU, chamada GNU C Library, ou simplesmente GLibC.

A GLibC implementa as funcionalidades da LibC padrão e da extensão POSIX, mas traz também um grande conjunto de extensões que não estão disponíveis em outras implementações. Por isso, ao desenvolver programas que devem funcionar em mais de uma plataforma, deve-se verificar se as funções utilizadas são suportadas em todas elas.

Estruturas de dados

GLib

A GLib é uma biblioteca de uso geral com um grande conjunto de funcionalidades para a construção de estruturas de dados (listas, árvores, tabelas *hash*, etc.). A biblioteca GLib é muito usada para a construção de aplicações em Linux, sobretudo no ambiente gráfico Gnome. Todavia, pode ser compilada e usada em outras plataformas, pois não tem relação com a interface gráfica.

Páginas com mais informações sobre a GLib:

- GLib na Wikipedia
- tutorial da IBM

SGLib

Outra opção para a construção de estrutura de dados usuais, como listas e árvores, é a biblioteca SGLib - A Simple Generic Library for C.

Interface gráfica

SDL

A biblioteca SDL (Simple DirectMedia Layer) oferece acesso à interface gráfica do computador. Estas funcionalidades são oferecidas:

- abertura de janelas
- operações de desenho (linhas, áreas, etc)
- operações de áudio
- leitura de posição e eventos do mouse
- leitura do teclado
- Uso de fontes de caracteres

Esta biblioteca está disponível para C e C++ em várias plataformas, como Linux, Windows, Android e iOS. Mais informações sobre SDL podem ser obtidas em:

- https://www.libsdl.org
- https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer
- http://lazyfoo.net/SDL_tutorials/index.php

Allegro

A biblioteca Allegro permite a manipulação de gráficos simples e áudio, sendo bem adaptada para a construção de jogos 2D. É uma biblioteca mais simples (mais limitada) que SDL, mas boa para projetos menores.

Algumas de suas características:

- multiplataforma: Linux, Windows, MacOS, Android, iOS
- pode ser usada em C e outras linguagens
- usa aceleração gráfica (através de OpenGL ou DirectX)
- manipulação de áudio e vídeo

Interface de usuário

NCurses

A biblioteca NCurses permite a manipulação do terminal de texto, oferecendo as seguintes funcionalidades:

- posicionamento do cursor
- leitura não-bloqueante do teclado
- leitura de teclas especiais (setas, teclas de função)
- manipulação de cores
- criação de janelas, menus e forms em modo texto
- leitura de eventos do mouse

Estas páginas oferecem informações adicionais sobre a biblioteca NCurses:

- Ncurses Programming Howto
- Writing Programs with Ncurses
- Ncurses Programming Guide
- Exemplos do ncurses no Linux, em `/usr/lib/ncurses/examples/` (instalar o pacote `ncurses-examples`)
- Código-fonte do pacote `ncurses-examples` disponível no Debian e derivados (Ubuntu, Mint, etc).

GTK

A biblioteca GIMP ToolKit foi desenvolvida para o ambiente de desktop Gnome (Linux). Ela permite a construção de janelas gráficas com elementos de interface do usuário (janelas, menus, botões, etc).

- https://www.gtk.org
- https://en.wikipedia.org/wiki/GTK
- Tutorial GTK: https://developer.gnome.org/gtk-tutorial/stable
- Construtor de interfaces: https://glade.gnome.org

Armazenamento

GDBM

A biblioteca GDBM (GNU dbm) permite criar bases de dados simples em disco, estruturadas na forma de pares *chave/valor*. São oferecidas funções para criar/destruir bases e criar/remover/buscar registros em uma base.

A estrutura interna da base permite operações de busca/inserção muito rápidas.

SQLite

Quando um programa precisa armazenar e manipular um grande volume de dados estruturados, pode fazer uso da biblioteca SQLite. Esta biblioteca constrói a abstração de uma base de dados relacional no padrão SQL em um arquivo em disco. A maioria das operações típicas de DBMS relacionais, como criação de tabelas, buscas, fusões, etc são suportadas pela biblioteca.

SQLite é a biblioteca usada para armazenamento de dados do usuário no ambiente Android e nos navegadores web Firefox e Chrome.

Ciência

GSL

A biblioteca GSL (GNU Scientific Library) oferece mais de 1000 funções para operações matemáticas de alto desempenho, como:

- números complexos
- raízes de polinômios
- álgebra linear
- equações diferenciais
- transformadas
- estatística
- integração (Monte Carlo, etc)
- otimização (mínimos quadrados, etc)

OpenCV

A biblioteca OpenCV (Open Computer Vision) foi desenvolvida pela Intel para o processamento de imagens e vídeos, trazendo uma grande quantidade de funções com desempenho otimizado para o tratamento de fotografias, vídeos capturados por câmeras, etc.

Construção de bibliotecas

Video desta aula

Bibliotecas são amplamente utilizadas na linguagem C. Além da biblioteca padrão (LibC) e das bibliotecas disponibilizadas pelo sistema operacional, o programador pode desenvolver suas próprias bibliotecas, para usar em seus projetos ou disponibilizá-las a terceiros.

As bibliotecas podem ser construídas para ligação estática ou dinâmica (DLL) com o código executável que as utiliza. Este texto explica as duas técnicas em ambiente Linux.

Estrutura geral

Vamos usar como exemplo uma biblioteca simples chamada *Hello*, que oferece funções para escrever na tela mensagens de "olá" em diversas linguagens. O código-fonte dessa biblioteca é composto pelos arquivos abaixo.

O arquivo de cabeçalho define a interface da biblioteca *Hello*:

hello.h

```
#ifndef __HELLO__
#define __HELLO__

void hello_pt () ;
void hello_en () ;
void hello_fr () ;

#endif
```

Os demais arquivos definem a implementação dessas funções:

hello_pt.c

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

void hello_pt ()
{
  printf ("Ola, mundo!\n") ;
}
```

hello_en.c

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

```
void hello_en ()
{
  printf ("Hello, world!\n") ;
}
```

hello_fr.c

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

void hello_fr ()
{
  printf ("Salut, le monde !\n") ;
}
```

Um programa que utilize a biblioteca *Hello* pode ser escrito desta forma:

main.c

```
#include "hello.h"

int main ()
{
  hello_pt () ;
  hello_en () ;
  hello_fr () ;

  return 0 ;
}
```

Bibliotecas estáticas

Bibliotecas estáticas são ligadas ao programa durante o processo de compilação, resultando em um executável maior, mas menos dependentes das bibliotecas instaladas no sistema. Para construir uma biblioteca de ligação estática são necessários vários passos, descritos a seguir.

1) Inicialmente, todos os arquivos-fonte que irão compor a biblioteca devem ser compilados, para gerar seus arquivos-objeto correspondentes:

```
$ gcc -Wall -c hello_pt.c
$ gcc -Wall -c hello_en.c
$ gcc -Wall -c hello_fr.c
```

2) A seguir, deve ser usado o utilitário `ar` (*archiver*) para juntar todos os arquivos-objeto em uma biblioteca estática chamada `libhello.a`:

```
$ ar rvs libhello.a hello_pt.o hello_en.o hello_fr.o
```

Os flags `rvs` indicam:

- r (*replace*): substituir versões anteriores dos arquivos na biblioteca, caso existam
- v (*verbose*): mostrar na tela as inclusões que estão sendo realizadas
- s (*symbols*): criar uma tabela dos símbolos[1)] que estão sendo agregados à biblioteca

O utilitário `ar` possui diversos outros *flags*. Por exemplo, pode-se consultar o conteúdo de uma biblioteca estática:

```
$ ar t libhello.a
hello_en.o
hello_fr.o
hello_pt.o
```

Pode-se consultar todos os símbolos definidos em uma biblioteca estática (ou em qualquer arquivo objeto) através do utilitário `nm`:

```
$ nm libhello.a

hello-en.o:
0000000000000000 T hello_en
                 U puts

hello-fr.o:
0000000000000000 T hello_fr
                 U puts

hello-pt.o:
0000000000000000 T hello_pt
                 U puts
```

Para atualizar/incluir qualquer arquivo da biblioteca, basta executar `ar` novamente, indicando o(s) arquivo(s) a atualizar/incluir:

```
$ ar rvs libhello.a hello_it.o hello_es.o hello_jp.o
```

3) A forma mais simples de usar a biblioteca é indicá-la ao compilador no momento da compilação ou ligação:

```
$ gcc -Wall main.c -o main libhello.a
```

Uma opção abreviada de ligação pode ser utilizada. Nela, não é necessário indicar o nome completo da biblioteca:

```
$ gcc -Wall main.c -o main -L. -lhello
```

Esta abordagem é melhor que a anterior, pois neste caso o ligador somente irá incluir no executável final os objetos que forem efetivamente necessários.

A opção `-L.` é necessária para incluir o diretório corrente nos caminhos de busca de bibliotecas do ligador.

Observe que a biblioteca foi informada ao ligador na opção `-lhello`. Por default, ao encontrar uma opção `-labc`, o ligador irá procurar pela biblioteca `libabc.a` nos diretórios default de bibliotecas (`/lib`, `/usr/lib`, `/usr/local/lib`, ...) e depois disso nos diretórios informados pela opção `-L`.

Bibliotecas dinâmicas

Bibliotecas dinâmicas (DLLs) são ligadas ao programa durante a carga do executável na memória, resultando em um executável menores, mas que dependem das bibliotecas necessárias estarem instaladas no sistema operacional.

A construção de uma biblioteca de ligação dinâmica é um pouco mais complexa:

1) Primeiro, é necessário compilar os arquivos-fonte que irão compor a biblioteca usando a opção `-fPIC`, que irá gerar código binário independente de posição (PIC - Position Independent Code)[2]:

```
$ gcc -Wall -fPIC -c hello_pt.c
$ gcc -Wall -fPIC -c hello_en.c
$ gcc -Wall -fPIC -c hello_fr.c
```

2) A seguir, pode-se criar a biblioteca dinâmica, a partir dos arquivos-objeto:

```
$ gcc -Wall -g -shared -Wl,-soname,libhello.so.0 -o libhello.so.0.0 hello_pt.o
hello_en.o hello_fr.o
```

Observe que a opção `-Wl` transfere a opção `-soname=libhello.so.0` ao ligador. Essa opção permite definir o nome e versão da biblioteca.

3) Finalmente, para instalar a biblioteca, deve-se movê-la para o diretório adequado (geralmente `/usr/lib` ou `/usr/local/lib`)[3] e gerar os atalhos necessários para indicar os números de versão (0) e revisão (0):

```
# mv libhello.so.0.0 /usr/local/lib
# cd /usr/local/lib
# ln -s libhello.so.0.0 libhello.so.0
# ln -s libhello.so.0 libhello.so

$ ls -l
lrwxrwxrwx  1  prof      12   Out 2 18:20  libhello.so -> libhello.so.0
lrwxrwxrwx  1  prof      14   Out 2 18:06  libhello.so.0 -> libhello.so.0.0
-rwxr-xr-x  1  prof    6914   Out 2 18:06  libhello.so.0.0
```

4) A compilação usando a biblioteca ocorre da mesma forma que no caso estático:

```
$ gcc -Wall main.c -o main -L. -lhello
```

Ao carregar o executável, o sistema operacional irá localizar as bibliotecas dinâmicas necessárias, carregá-las e mapeá-las na área de memória do novo processo:

```
$ ./main
```

Caso a biblioteca esteja em um diretório não listado em `/etc/ld.so.conf` (arquivo de configuração do carregador e ligador dinâmico), ocorrerá um erro. Nesse caso, deve-se incluir o diretório nesse arquivo e a seguir executar `ldconfig`, ou informar o carregador dinâmico do SO através da variável de ambiente `LD_LIBRARY_PATH`:

```
$ export LD_LIBRARY_PATH=.
$ ./main
```

[1)]

nomes de funções e de variáveis globais
2)

Como a ligação da biblioteca ocorre durante a carga/execução, a posição de seu código na memória dos processos que irão utilizá-la não pode ser determinada previamente.
3)

se for um diretório público, isso deve ser feito pelo administrador.