Tópico 12

Representação e Estruturação de Dados

Autor: Wu Shin-Ting

DCA - FEEC - Unicamp

Outubro de 2019

No Capítulo 4 vimos que um microprocessador é, em princípio, projetado para acessar um espaço de memória (principal) correspondente à quantidade de *bits* de endereços que ele dispõe. Ele considera 8 *bits* (1 *byte*) como a menor unidade de armazenamento. Vimos ainda que todas as palavras são definidas através de um alfabeto de dígitos binários

{0,1}. Mostramos, não só no Capítulo 4 como no Capítulo 10, como podemos construir instruções a uma máquina com este alfabeto. Nos capítulos 5 e 6 mostramos como podemos mapear neste espaço de memória os módulos de memória disponíveis comercialmente, a fim de que os dados de interesse sejam efetivamente armazenados. É bem natural limitar as instruções de máquinas digitais em códigos binários, já que os processamentos delas são baseados em dois níveis lógicos. Porém, e os dados do mundo ciber-físico que se comunica com estas máquinas? Uma alfabeto binário não é demasiadamente pobre para transmitir para uma máquina os dados que nos cercam? Lembre-se de que estamos familiarizados a nos comunicar por meio de um rico sistema de linguagens verbais e não-verbais. E, dentre as linguagens verbais temos alfabetos, ideogramas e diferentes sistemas de numeração.

Neste capítulo apresentaremos algumas técnicas e ferramentas existentes capazes de reduzir a distância entre uma máquina e um projetista, para que este consiga comunicar verbalmente as suas ideias e fazer que a máquina execute eficientemente as tarefas designadas a ela. O processo da "tradução" do rico vocabulário de uma linguagem natural em códigos binários de uma máquina se passa por vários estágios, muitos deles ocorrem de forma automática hoje em dia. Figura 12.1 ilustra os estágios que ocorrem automaticamente num ambiente de desenvolvimento integrado, em inglês *Integrated Development Environment* (IDE), como CodeWarrior de NXP [1]. Um programa em linguagem de alto nível é **compilado**/traduzido para a linguagem de *assembly* e **ligado** com os mesmos símbolos nos códigos de outros arquivos e/ou das bibliotecas disponíveis, quando necessário. Depois, o programa em linguagem *assembly* é traduzido por um **montador** para uma sequência de instruções capazes de serem processadas pela máquina, como vimos na Seção 4.1. Esta sequência de instruções é também conhecida como **código executável**, pois basta **carregá**-lo num espaço da memória principal e escrever o seu endereço inicial no contador de programa (PC), a máquina executará, instrução por instrução, a tarefa programada como vimos na Seção 4.2.2.

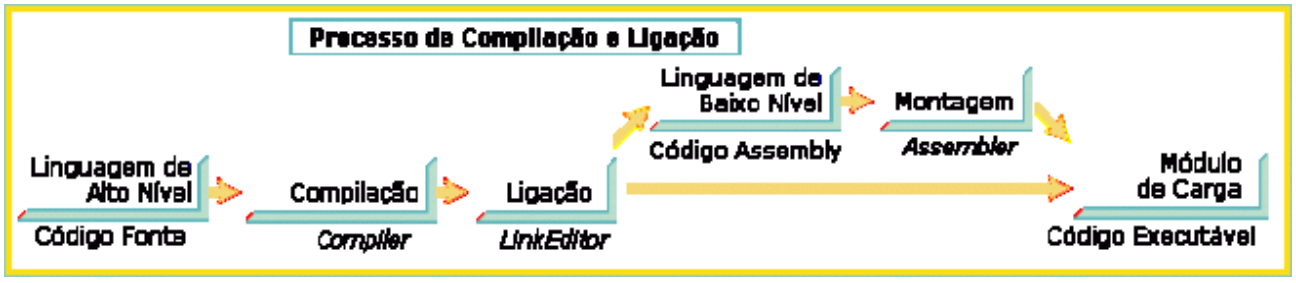

Figura 12.1: Tradução de uma linguagem de programação em alto nível para um código binário executável (Fonte: [16]).

Com essas ferramentas de suporte à tradução de uma linguagem mais próxima da linguagem natural (humana) a uma linguagem em códigos binários (máquina), o trabalho

de um projetista se reduz a escrever as suas ideias/comandos em linguagem de alto nível, como a linguagem de programação C.

Sob o ponto de vista computacional, os elementos da linguagem verbal são traduzidos basicamente em quatro classes de dados: números naturais (inteiros sem sinal), números inteiros (inteiros com sinal), números reais (pontos flutuantes) e caracteres alfanuméricos [2]. O tamanho dos números inteiros é tipicamente definido pelo tamanho dos registradores de trabalho na unidade lógico-aritmética do microprocessador, como comentamos na Seção 5.8. Na mesma Seção 5.8, vimos que os dados podem ser diferenciados através dos tipos de dados na **linguagem de alto nível C**. Através dos tipos de dados, um projetista consegue definir para o tamanho do espaço de memória que ele precisa alocar e como interpretar os códigos binários armazenados no espaço alocado.

A rápida evolução da interconexão dos objetos físicos dotados de tecnologia embarcada com uma rede tem impulsionado um crescimento explosivo de dados processados por um microcontrolador. Ao invés de declará-los individualmente como variáveis distintas, organizá-los e manipulá-los como um grupo de dados pode não só melhorar o uso do espaço da memória (*hardware*) como simplificar os processamentos (*software*). Isso é fundamental num projeto baseado em microcontroladores cujos recursos são escassos. Neste capítulo mostraremos as **estruturas de dados** que a linguagem C suporta e como podemos aplicá-las em projetos de sistemas embarcados para organizar os dados que fluem intra e inter-microcontroladores. Mostraremos também como podemos implementar algumas **estruturas de dados** mais complexas com uso da linguagem C. Vale ressaltar aqui que entendemos como uma estrutura de dados não só a forma como os dados são organizados como também um conjunto de operações definidas sobre estes dados organizados.

12.1 Representação de Dados

Os computadores digitais são essencialmente poderosas máquinas “trituradoras de números”. Todos os microprocessadores têm integrados nos seus circuitos as unidades lógico-aritméticas para processarem números inteiros (Seção 4.1) e muitos deles têm coprocessadores de pontos flutuantes integrados para realização de operações sobre o domínio real (Seção 4.4) .

Por isso, os primeiros códigos binários foram os de **números inteiros** sobre os quais os circuitos combinacionais conseguiam realizar operações aritméticas de forma análoga como fazemos conta com os números na base decimal. Para atender a demanda de cálculos científicos envolvendo valores reais, foi introduzida a representação em **ponto flutuante** baseada na notação científica[1]. E, para o restante dos tipos de dados, pode-se

[1] As notações científicas possuem a seguinte forma **N . 10^{n}**, onde **N** é a mantissa, ou um **coeficiente** real maior ou igual a 1 e menor que 10, também é chamado de **mantissa**, e **n** é o **expoente inteiro** ou **ordem de grandeza**, negativo para um número muito pequeno e positivo para um número muito grande.

sempre usar o código internacional de troca de informações textuais em que a cada caractere alfa-numérico é associado um código binário.

Nesta seção apresentaremos diferentes códigos binários usados para armazenar os dados na memória principal.

12.1.1 Números Naturais

O sistema de numeração decimal é o mais conhecido entre os seres humanos. Esse sistema é composto por dez diferentes dígitos (0-9). Adota-se a notação posicional para representar um valor, ou seja, o dígito mais à direita em um número é o menos significativo. Cada posição vale 10 vezes mais que a de posição à sua direita. No entanto, o sistema de numeração decimal não é compatível com a natureza binária dos computadores. Para os computadores, o sistema binário que representa os números utilizando apenas dois dígitos (0-1) é o mais apropriado. Este sistema também usa a notação posicional em que cada dígito vale 2 vezes mais que um dígito à sua direita.

O **sistema de numeração binária** é usado, de forma direta, para codificar os números naturais ou os números inteiros sem sinal. Com *n* bits alocados na memória, podemos representar, de forma não ambígua, até 2^n valores (0 até 2^n-1). Como a menor unidade endereçável é um *byte*, é típico especificar o tamanho de memória para representar um número inteiro sem sinal em termos de quantidade de *bytes*. Podemos ter um valor natural representado por 1 *byte* (tipo de dado em C, char), 2 *bytes* (tipo de dado em C, short int), 4 *bytes* (tipo de dado em C, unsigned int) e 8 *bytes* (tipo de dado em C, long long int). Vale ressaltar aqui que numa representação binária de máquina, todos os bits alocados ao valor, e não ocupados pelo valor, são automaticamente preenchidos de 0. Por exemplo, se declararmos uma variável do tipo de dado short int (16 *bits*) e atribuirmos à variável o valor 2, será armazenado no espaço de memória o valor 0b0000000000000010 e não simplesmente 0b10.

Formas mais compactas para representar códigos binários são agrupamentos dos *bits* em números que são potência de 2, como 4 (sistema quaternário), 8 (octal) e 16 (hexadecimal, em 1 *nibble*). Destes sistemas, a compactação mais utilizada em sistemas embarcados é o sistema de numeração **hexadecimal**. Este sistema tem um fator de multiplicação 16. Ele é composto pelos dígitos de 0-9 e letras A-F, que correspondem aos valores em decimal 10-15. Cada dígito hexadecimal é representado por 4 dígitos binários, o que torna direta a conversão de binário para hexadecimal.

12.1.2 Números Inteiros (com Sinal)

Números inteiros com sinal incluem tanto os números positivos quanto os negativos. Para representar esses números em binário destacam-se quatro diferentes formas.

A primeira delas é denominada **representação de sinal e magnitude**, em que um *bit*, usualmente o *bit* mais significativo, é reservado para representar o sinal do valor. Por convenção, 0 representa o sinal "+", correspondendo a um número positivo, e 1 representa "-", o que corresponde a um número negativo. Esse *bit* é chamado de *bit de sinal*. Cabe ressaltar que o restante da sequência de *bits*, tanto para um número positivo quanto para um negativo, é usado para o código binário do módulo do valor. Em termos do valor máximo representável em n *bits*, a representação sem sinal vai até 2^n-1, enquanto a representação de sinal e magnitude consegue representar até $2^{(n-1)}$-1.

A segunda representação é conhecida como **complemento de um**, em que todos os números negativos são gerados a partir da sua contrapartida positiva, complementando *bit* a *bit* todos os seus *bits* inclusive os *bits* 0 estendidos (Seção 12.1.1). Ou seja, troca-se os 0's por 1's, e vice-versa. Esta representação tem a desvantagem de que há dois códigos binários associados ao número 0. Por exemplo, os dois códigos binários 0000 e 1111 representam o valor zero para um conjunto de números representado por 4 *bits*. Note que, como há também um *bit* de sinal para distinguir valores positivos dos negativos, o valor máximo representável é a metade do valor máximo que a repesentação sem sinal consegue representar para uma mesma quantidade de *bits*.

A terceira alternativa é a representação por **complemento de 2**, que resolve não só o problema da representação em complemento de 1, como também facilita a implementação dos circuitos de operações aritméticas sobre os números inteiros. A notação de complemento 2 de um número negativo é obtida ao somar 1 à sua representação em complemento de 1. Com *n* bits nesta notação podemos representar os valores de -2^{n-1} até 2^{n-1}-1. O número 0 possui uma representação única e o *bit* mais significativo é o *bit* de sinal. É adotada a mesma convenção das outras duas representações para interpretar este *bit*: se 0 o número é positivo, senão o número é negativo. O interessante é que se quisermos converter um número negativo em complemento de 2 para a sua contrapartida positiva, também em complemento de 2, basta complementarmos *bit* a *bit* o número e somarmos 1 ao resultado. Um problema observado nesta representação é a ordenação dos números pelo valor representado.

A quarta representação é a representação de **excesso-N**. Esta representação usa o valor N como um valor de deslocamento (polarização), de forma que o código binário de N corresponde ao número 0 e o código binário com todos os *bits* zerados corresponde a -N. Desta forma, podemos ordenar os números negativos e positivos comparando diretamente os seus códigos binários. Tabela 12.1 ilustra os números representados em excesso-7 por 4 *bits*. Como a polarização é 7, qualquer valor inteiro i é representada por i-7 em código binário.

Tabela 12.1: Excesso-7 em 4 *bits*.

Valor binário	Valor representado	Valor binário	Valor representado

0000	-7	1000	1
0001	-6	1001	2
0010	-5	1010	3
0011	-4	1011	4
0100	-3	1100	5
0101	-2	1101	6
0110	-1	1110	7
0111	0	1111	8

12.1.3 Números Reais

Vale lembrar que a quantidade distinta de elementos, que um conjunto de *n* bits pode representar de forma biunívoca, é limitada em até 2^n. Como podemos representar um número infinito de valores contidos num intervalo real? Uma primeira ideia foi a **representação por ponto fixo**, alocando uma parte de *bits* para a parte inteira e o restante para a parte decimal. Percebeu-se logo que essa representação não se ajusta dinamicamente ao valor da parte inteira e da parte decimal. Daí, veio a inspiração de representar um número real N na forma normalizada, análoga à notação científica

$$N = M \times 10^E,$$

com $M \in (-10, 10)$. Os valores *M* e *E* são denominados, respectivamente, **mantissa** e **expoente**. Adequando à base binária e à necessidade de distinguir a parte inteira da parte decimal e o valor positivo do valor negativo, foi proposta uma **representação em ponto flutuante** em que

$$N = s\ (1.m \times 2^e),$$

com a mantissa representando somente a parte decimal na base binária, o expoente *e* na base binária e o sinal do valor N. O *bit* 1 na parte inteira da mantissa não é armazenado. Ele é implícito na representação. Portanto, uma palavra de *n* bits é dividida em 3 campos: 1 *bit* de sinal, x *bits* de mantissa e y *bits* de expoente. O expoente pode ser negativo ou positivo e a notação de excesso é usada para representá-lo. Os campos expoente e mantissa variam conforme a precisão da normalização.

O padrão **IEEE 754-1985** estabelece dois formatos: um de precisão simples (padrão 4 *bytes* ou 32 *bits*) e um de precisão dupla (padrão 8 *bytes* ou 64 *bits*). Para o primeiro, o expoente tem 8 *bits* e a mantissa 23 *bits*, e para o segundo, o expoente tem 11 *bits* e a mantissa 52 *bits*. Para a precisão simples é usada a notação de excesso-127 e para a precisão dupla, excesso-1023. Por exemplo, para o seguinte código binário

$$1\ 10000001\ 00100000000000000000000_2$$

temos

- S = 1
- E = 10000001 na base 2 = 129 → 129-127 =2
- $m = 2^{-1}*0+2^{-2}*0+2^{-3}*1=0.125$.

Juntando as peças, temos o valor na base decimal:

$$N =(-1)\ (1.125 \times 2^2) = -4.5_{10}.$$

Observe que a representação impícita de 1 na parte inteira inviabiliza a representação exata de zero e a quantidade de *bits* no campo expoente limita a representação de um valor "infinitamente" grande. O padrão IEEE 754 considerou em separado, a representação de 4 casos especiais:

- Representação de zero: assume-se que, quando todos os *bits* nos campos expoente e mantissa estiverem 0, o valor representado é zero.
- Valores denormalizados: se todos os *bits* no campo expoente estiverem em 0, então o valor representado no campo mantissa é diferente de zero, então o valor representado é $(0.m \times 2^{-126})$ (em precisão simples).
- Representação de "infinito": com todos os *bits* no campo expoente em 1 e todos os *bits* no campo mantissa em 0.
- Representação NaN (*Not a Number*): com todos os *bits* no campo expoente em 1 e o campo mantissa com um valor diferente de zero.

Vale ressaltar que, como a quantidade de *n* bits para representar a mantissa é limitada, pode existir uma pequena perda de acurácia em relação ao número original, por exemplo uma dízima periódica, na representação em ponto flutuante.

12.1.4 Codificação de Caracteres Alfanuméricos

Em sistemas digitais todos os dados são representados em códigos binários, 0s e 1s. Alfabetos ou símbolos especiais podem ser representados em códigos binários e armazenados digitalmente por meio dos códigos ASCII (*American Standard Code for Information Interchange*), ASCII estendido [28] ou EBCDIC (*Extended Binary Coded Decimal Interchange Code*), ISO-8859-1 [27], e UNICODE, incluindo UTF-8 (8-*bit Unicode Transformation Format*) [26] e UTF-16 (16-*bit Unicode Transformation Format*).

Tabela12.2 sintetiza os 255 códigos ASCII dos caracteres de controle e caracteres alfanuméricos. Sendo 255 caracteres, 8 *bits,* ou um *byte*, é suficiente para representar cada código [4].

Tabela 12.2: Códigos ASCII.

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

Note que, a partir dos códigos ASCII, podemos construir qualquer texto em qualquer língua que não tenha caracteres especiais e acentos. E, a partir de UNICODE, é possível montar um arquivo textual de uma grande variedade de línguas incluindo o chinês [18].

12.2 Estruturas Básicas em C

Na Seção 5.8 vimos formas de alocar espaços de memória de tamanhos diferentes para uma variável. Na Seção 12.1 mostramos como são representados em códigos binários os valores associados a essa variável, sejam eles em números naturais, inteiros, reais ou em caracteres alfanuméricos. Porém, como já mencionado antes, em decorrência do volume que os dados alcançaram, sem uma organização adequada torna quase inviável gerenciá-los e processá-los apropriadamente com os recursos disponíveis nos sistemas embarcados. Veremos nesta seção alguns comandos básicos em linguagem C que nos permitem fazer alguns agrupamentos dos dados.

12.2.1 Arranjo

Quando se faz amostragem periódica das entradas de um sensor, é interessante armazenar as amostras de mesmo tipo de dados num vetor. Através do seguinte comando da linguagem C podemos instruir um compilador a alocar um bloco contíguo de *bytes* na memória para armazenar uma sequência de dados de mesmo tipo, denominado

um **arranjo** (*array*) [13]

<tipo de dado> <nome do arranjo> [<número de elementos do arranjo>];

O tipo de dado de um arranjo é o tipo de dado dos seus elementos. O primeiro elemento de um arranjo se inicia sempre na posição 0 e seus elementos são endereçados por índices inteiros não negativos. Para um arranjo com N elementos, o índice do seu último elemento é N-1. Por exemplo, para armazenar uma página de até 32 *bytes* num acesso de uma memória externa I2C 24AA64 [], é interessante ter as amostras digitais em 16 *bits*, convertidas por um conversor AD de um microcontrolador, agrupadas de 16 em 16. Uma forma é declarar de forma estática um arranjo amostras_AD_e de 16 elementos de 16 *bits* com o seguinte comando:

```
uint16_t amostras_AD_e[16];
```

O tipo de dado uint16_t corresponde a números inteiros sem sinal (*unsigned*) representados por 16 *bits* (Seção 5.8). Podemos também declarar somente uma variável de endereço

```
uint16_t *amostras_AD_d;
```

e alocar dinamicamente no tempo de execução um espaço contíguo com a quantidade desejada de elementos com uso da função de C malloc (Seção 6.5.1.2)

```
amostra_AD_d = (uint16_t *)(malloc(sizeof(uint16_t)*16);
```

Esta instrução aloca da memória principal uma espaço contíguo de 16*(tamanho do tipo de dado uint16_t correspondente a 2 *bytes*). Como o tipo de dado retornado pela função malloc é o endereço do tipo de dado char (uint8_t), precisamos redefiní-lo para o tipo (uint16_t *) para que o microprocessador acesse os elementos em 2 *bytes* (uint16_t) e não em 1 *byte* (char).

Cabe aqui comentar que as funções de alocação dinâmica podem comprometer a velocidade de execução de um programa. Recomenda-se o uso de alocações estáticas se houver espaço suficiente na memória do microcontrolador. **Estimar o tamanho da memória requerido para as aplicações que deverão ser atendidas por um projeto é muitas vezes uma das tarefas mais difíceis para um projetista, principalmente quando o espaço de memória disponível é muito limitado**.

É possível declarar um arranjo com os seus elementos inicializados, como o comando abaixo:

```
uint8_t arranjo[]={0,0,0,0,0,0,0,0,0,0};
```

que equivale a

```
uint8_t arranjo[10]={0,0,0,0,0,0,0,0,0,0};
```

Pois, pela quantidade de inicializações, o compilador aloca automaticamente um espaço de 10 elementos para o arranjo mesmo que na declaração não tenha fornecida explicitamente a quantidade de elementos.

Organizando os dados como um arranjo, podemos não só fazer transmissões por um bloco de dados como acessar, em C, elementos diferentes i de um arranjo por uma única variável com o comando

amostras_AD_d[i]

ou pelo comando

*(amostras_AD_d+i).

12.2.2 *String*

Hoje em dia muitos comandos relacionados com a comunicação entre os dispositivos são em sequências de caracteres, como os comandos AT do dispositivo Bluetooth HC-05 [19]. Por exemplo, para resetar uma comunicação ou para restaurar a configuração do fabricante deste dispositivo, deve-se enviar, respectivamente, a sequência de caracteres alfanuméricos “AT+RESET” e “AT+ORGL” para ele. Em linguagem C, uma sequência de caracteres, ou uma *string*, pode ser tratado como um arranjo especial de caracteres alfanuméricos, do tipo *char*, terminado com o caractere nulo ('\0'), em inglês *null character* [14], cuja declaração assume o formato [20]:

char **<**nome_da_string>[<número de caracteres>].

Assim, para as sequências de caracteres mencionadas anteriormente, podemos declarar duas variáveis com inicializações:

char BLUETOOTH_RESET[9]=”AT+RESET”;
char BLUETOOTH_RESTORE[8]=”AT+ORGL”;

Deve-se lembrar que o caractere nulo (‘\0’), cujo valor em hexadecimal é 0x00, é contabilizado no tamanho da *string* ao declará-la e que a primeira posição do vetor começa em zero. Pois, são equivalentes às declarações acima os seguintes comandos de inicialização geral de arranjos:

char BLUETOOTH_RESET[]={'A','T’,’+’,’R’,’E’,’S’,’E’,’T’,’\0’};
char BLUETOOTH_RESTORE[]={‘A’,’T’,’+’,’O’,’R’,’G’,’L’,’\0’};

Se alocarmos mais espaços na memória do que a quantidade de caracteres na iunicialização, o compilador preenche o restante dos campos com o caractere nulo. Por exemplo,

char BLUETOOTH_RESET[12]=”AT+RESET”;

equivale a armazenar no espaço de 12 bytes alocados 'A', 'T’, ’+’, ’R’, ’E’, ’S’, ’E’, ’T’, ’\0’, ’\0’, ’\0’ e ‘\0’. A linguagem C suporta uma grande gama de funções que facilitam a manipulação de strings. Os protótipos dessas funções se encontram no arquivo <string.h>. Tabela 12.3 apresenta uma descrição sucinta das funções mais comumente usadas. Observe que o tipo de dado size_t é equivalente ao tipo de dado unsigned int.

Tabela 12.3: Funções em C que facilitam a manipulação de *strings*

Função	Finalidade
char *strcpy(str1, str2)	Copia str2 para str1.
char *strncpy(str1,str2,n)	Copia str2 para str1 até n caracteres.
char *strcat(str1, str2);	Concatena str2 ao final de str1.
size_t strlen(str1);	Retorna o tamanho de str1, a menos do caractere nulo no fim da *string*.
int strcmp(str1, str2);	Compara as *strings* – retorna 0 caso elas sejam iguais; retorna um inteiro negativo caso str1 seja menor que str2 (alfabeticamente); retorna um inteiro positivo caso str1 seja maior que str2. Essa função considera se os caracteres são minúsculos ou maiúsculos (*case sensitive*).
char *strrchr(s1, letra);	Retorna o endereço da última ocorrência do caracter letra na *string*.
char *strpbrk(s1, s2);	Retorna o endereço do primeiro caracter em s1 que casa com uma letra em s2.
char *strstr(s1, s2);	Localiza uma *substring* (s2) dentro de uma *string* (s1), retornando o endereço da primeira ocorrência da *substring*.
char *strtok(s1, delim);	Divide recorrentemente uma *string* em uma série de itens separados pelos delimitadores (símbolos) especificados em delim.

12.2.3 Union

Quando queremos armazenar num mesmo espaço de memória um dos dados de diferentes tipos de uma mesma fonte, podemos usar um tipo de dados especial denominado **union** [21]. Neste caso, o compilador aloca um espaço de memória do tamanho do membro que ocupa mais espaço de memória e só permite armazenar um membro em cada vez.

Por exemplo, podemos reservar um mesmo espaço de memória para armazenar uma amostra de temperatura que pode assumir diferentes formatos:

```
uint8_t amostra_8bits;
uint16_t amostra_16bits;
float temperatura;
```

com o seguinte comando:

```
union temperatura {
        uint8_t amostra_8bits;
        uint16_t amostra_16bits;
        float temperatura;
};
```

e a seguinte declaração

```
union temperatura variavel, *dado;
```

Para acessar um dos membros da estrutura **union**, usamos o operador de acesso ao membro (.). E para acessar um dos membros através de um endereço da estrutura union, usamos o operador de acesso ao membro (->). Por exemplo, variavel.amostra_8bits e dado->amostra_16bits.

12.2.4 Struct

Nas Seções 12.2.1 e 12.2.2 mostramos como usamos comandos da linguagem C para agrupar uma lista de dados e na Seção 12.2.3, como criar uma estrutura de dados multi-propósito. Em muitas situações, é interessante podermos ainda associá-los com outros tipos de dados formando um novo tipo de estrutura, como associar à lista amostra_AD_e[100] da Seção 12.2.1 a frequência de amostragem freq e o instante inicial de amostragem na resolução de minutos ou segundos:

```
float freq;
uint8_t min_seg;
union unidades_tempo {
        uint32_t minuto;
        uint32_t segundo;
        } inicio;
```

Para isso podemos usar uma variável especial da linguagem C: **struct**. Esta variável permite associar num único **registro** de informação diversas outras variáveis internas [12]. Essas variáveis internas podem ser de diferentes tipos de dados e são denominadas **membros** da struct. Para as 100 amostras do sensor, podemos por exemplo criar o seguinte tipo de (estrutura de) dado:

```
struct registro {
        uint16_t amostras_AD_e[100];
        float freq;
        uint8_t min_seg;
        union unidades_tempo {
                uint32_t minuto;
                uint32_t segundo;
        } inicio;
};
```

e declarar uma ou mais variáveis com este tipo de estrutura, como

```
struct registro amostras_AD;
```

ou uma variável que endereça a este tipo de estrutura, como:

```
struct registro *amostras_AD_ptr;
```

Para se referir a um membro de um registro, escreve-se o nome da variável e o nome do membro separados por um ponto, como amostras_AD.amostras_AD_e[1], amostras_AD.freq e amostras_AD.hora. E para acessar a um membro a partir do endereço de um registro, escreve-se o nome da variável e o nome do membro separados por uma seta, como amostras_AD_ptr->amostras_AD_e[1], amostras_AD_ptr->freq e amostras_AD_ptr->minuto.

O tipo de (estrutura) de dado criado com **struct** pode ser redefinido com um outro nome através da diretiva **typedef**, como vimos na Seção 5.8. Por exemplo,
#typedef struct registro dados_AD;

Por fim, vale destacar que a **struct** permite que sejam definidos explicitamente a quantidade de *bits* a serem alocados para cada um dos seus membros. A quantidade de *bytes* alocados é computada automaticamente com base na quantidade de *bits* atribuída a cada membro. Por exemplo, podemos definir um registro de controle em que os 8 *bits* controlem a direção dos 8 pinos de uma porta:

```
struct registro_direcao {
        uint8_t dir0:1;
        uint8_t dir1:1;
        uint8_t dir2:1;
        uint8_t dir3:1;
        uint8_t dir4:1;
        uint8_t dir5:1;
        uint8_t dir6:1;
        uint8_t dir7:1;
};
```

12.3 Estrutura de Dados

Na Seção 11.3.2 comentamos que, organizando os dados de forma adequada na memória, podemos mitigar os problemas relacionados com a concorrência do uso compartilhado de recursos. Além das estruturas básicas suportada pela linguagem C, mostraremos nesta seção algumas estruturas complexas que facilitariam manipulações dos dados e que podem ser facilmente construídas com uso dos comandos de C.

12.3.1 Pilhas

Quando falamos sobre o processamento de interrupções na Seção 10.4.1, mencionamos que processadores usam pilhas para salvar os dados do fluxo de controle em execução antes do desvio para a rotina de serviço no atendimento de uma exceção ou uma interrupção. Cada vez que se chaveia de um trecho de código para um outro trecho, é

empilhado automaticamente na pilha o conteúdo dos registradores de trabalho e as variáveis locais, como mostra a Figura 12.2. Quando ocorrem múltiplos chaveamentos, são empilhados múltiplos blocos de forma que o contexto mais recente fique no topo da pilha. Isso facilita a recuperação do contexto no retorno de cada exceção/interrupção, do mais recente para o mais antigo.

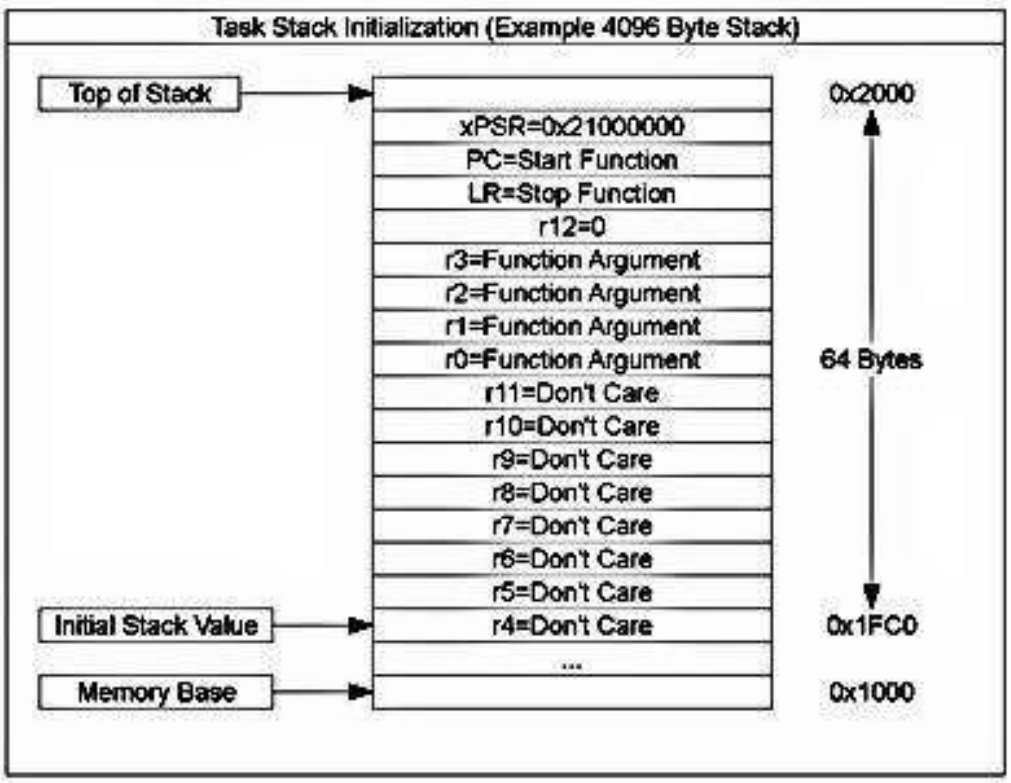

Fig. 12.2: Empilhando os registradores e variáveis locais no processamento de uma interrupção.

As **pilhas** (*stacks*) são estruturas de dados linear LIFO (*last-in*, *first out*), onde o último elemento inserido será o primeiro elemento a ser retirado. Tìpicamente, uma pilha é alocada a um espaço contíguo da memória principal cujo menor endereço corresponde à sua base e o maior endereço ao seu topo.

O endereço do **topo de uma pilha**, em inglês *stack pointer* (SP), é armazenado num registrador e é considerado o endereço-base para acessar todos os dados armazenados nela. Duas operações são associadas à manipulação de uma pilha (Figura 12.3): **empilhar** (*push*) e **desempilhar** (*pop*). Empilhar seria escrever novos dados a partir do endereço SP e atualizar o valor de SP incrementando-o com o tamanho dos dados em *bytes* adicionados. Desempilhar, por sua vez, seria remover os dados da pilha subtraindo o valor de SP do tamanho dos dados retirados em *bytes*.

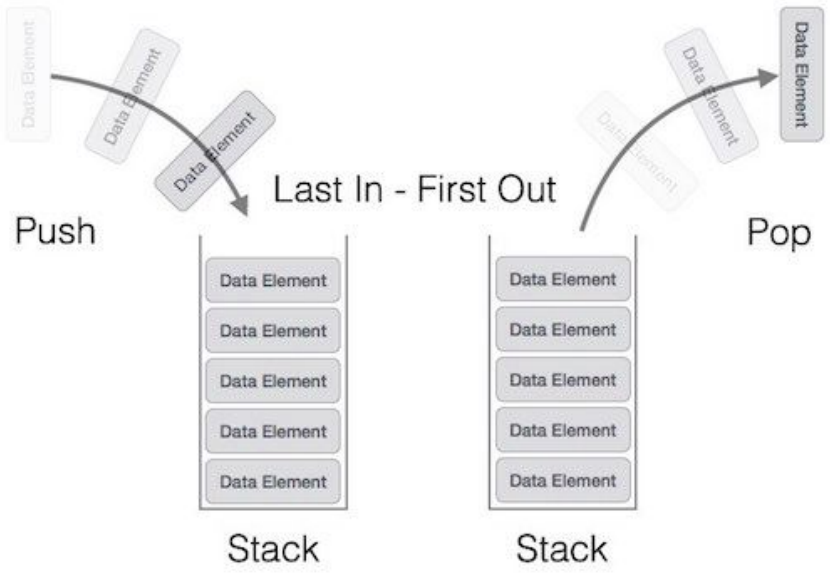

Figura 12.3: Pilha (Fonte: [22]).

Vale comentar que a estimativa do tamanho do espaço de uma pilha é fundamental para evitar **vazamento de memória**, em inglês *memory leakage* [23]. Figura 12.4 mostra um caso de invasão do espaço de memória tipicamente reservado para variáveis globais e estáticas na organização do espaço de memória de um microcontrolador. Isso pode destruir os dados armazenados e causar comportamentos inesperados do sistema.

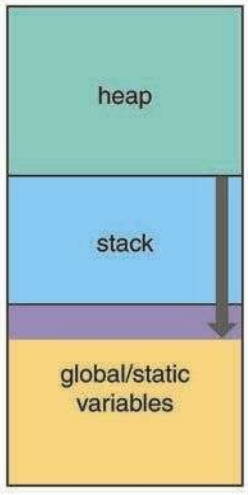

Figura 12.4: Invasão do espaço de memória por vazamento (Fonte: [23]).

O seguinte trecho de código em C demonstra uma implementação simples da estrutura de uma pilha com uso de *struct* e arranjo de tamanho fixo alocado dinamicamente.

```
#include <stdlib.h>

struct pilha {
	unsigned int topo;
	unsigned int top;
	int *elementos;
};

void cria_pilha (struct pilha *p, unsigned int n)
```

```
{
	p->topo = 0;
	p->top = n;
	if (n == 0)
		p->elementos = NULL;
	else
		p->elementos = (int *)malloc(sizeof(int)*n);
	return;
}

uint8_t empilha (struct pilha *p, int v)
{
	if (p->topo == p->top) return 0;
	p->elementos[p->topo++] = v;
	return 1;
}

int desempilha (struct pilha *p)
{
	int v;
	if (p->topo) {
		v = p->elementos[p->topo-1];
		p->topo--;
		return v;
	} else {
		return 0xffffffff;
	}
}

int topo (struct pilha *p)
{
    return(p->elementos[p->topo-1]);
}

uint8_t pilha_cheia (struct pilha *p)
{
	if (p->topo == p->top) return 1;
	return 0;
}

uint8_t pilha_vazia (struct pilha *p)
{
	if (p->topo == 0) return 1;
	return 0;
}

int tamanho (struct pilha *p)
{
     return (p->top);



}

void deleta_pilha (struct pilha *p)
{
     if (p->elementos)
       free((void *)p->elementos);
```

```
    p->top = p->topo = 0;
    p->elementos = NULL;
}
```

12.3.2 Filas

Fila de mensagem, em inglês *message queue*, é uma das formas mais simples para transferir os dados entre diferentes dispositivos interconectados. O envio de um comando AT do módulo HC-06 mostrado na Seção 12.2.2 através de uma sequência de caracteres é um exemplo de uma fila de mensagem. Tipicamente, uma mensagem é serializada em caracteres num *buffer* do transmissor e mantida temporariamente neste *buffer* até que o receptor endereçado fique pronto para recebê-la como ilustra a Figura 12.5. Podemos pensar que o transmissor seja um módulo de comunicação serial UART de um microcontrolador e o receptor, um outro UART integrado num outro microcontrolador, e que a comunicação se dá por interrupção pelo evento de "presença de dado para ser transmitido" no transmissor. Como resposta a esta interrupção, a rotina ISR do transmissor (esquerda) coloca os dados na fila de mensagem. A presença desses dados no pino de entrada do receptor ativa o evento de "presença de dado na entrada" no receptor e a rotina ISR associada (direita) é então executada para ler o *buffer*. O processamento dessas filas de mensagem se beneficiaria muito com uma estrutura de dados conhecida como **fila**.

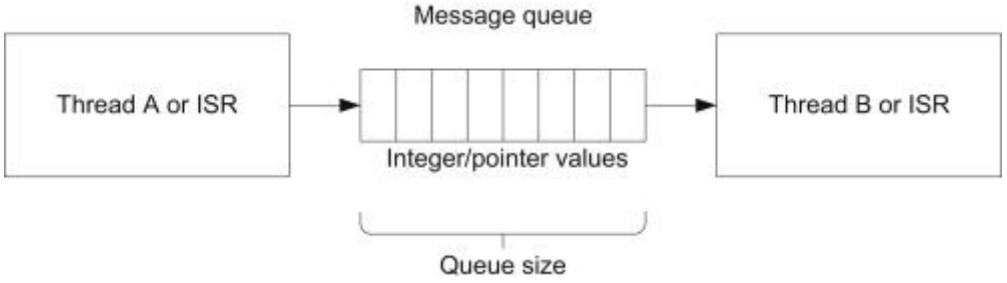

Figura 12.5: Fila de mensagem entre um transmissor e um receptor.

As **filas** (*queues*) são estruturas de dados lineares FIFO (*first-in*, *first-out*), onde o primeiro elemento inserido será o primeiro elemento a ser removido. Para poder inserir e remover de forma eficiente os elementos na fila, o endereço do **primeiro** e do **último** elemento da fila são armazenados junto com a fila. Como a estrutura de pilha, é alocada a um espaço contíguo da memória principal. Duas operações são associadas à manipulação dos dados de uma fila: **enfileirar** (*enqueue*) e **desenfileirar** (*dequeue*) (Figura 12.6). Enfileirar consiste essencialmente "adicionar um novo endereço" no fim da fila e desenfileirar, "remover um endereço" no início da fila.

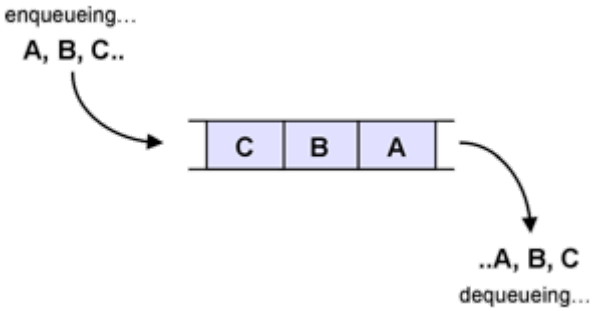

Figura 12.6: Fila (Fonte: [3])

O seguinte trecho de código em C implementa uma fila de “num” elementos com uso de *struct* e arranjo de tamanho fixo alocado dinamicamente.

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

struct fila {
	unsigned int ultimo;
	unsigned int num;
	int *elementos;
};

void cria_fila (struct fila *f, unsigned int n)
{
	f->ultimo = 0;
	f->num = n;
	if (n == 0)
		f->elementos = NULL;
	else
		f->elementos = (int *)malloc(sizeof(int)*n);
}

uint8_t enfileira (struct fila *f, int v)
{
	if (f->ultimo >= f->num) return 0;

    f->elementos[f->ultimo++] = v;
    return 1;
}

int desenfileira (struct fila *f)
{
	int v;

	if (f->ultimo > 0) {
	    v = f->elementos[0];
	    memcpy (&(f->elementos[0]), &(f->elementos[1]), (f->ultimo - 1)*sizeof(int));
	    f->ultimo--;
	    return v;
	}
	return 0xffffffff;	///< invalid value
}
```

```
int primeiro (struct fila *f)
{
    return(f->elementos[0]);
}

int ultimo (struct fila *f)
{
       if (f->ultimo)
              return(f->elementos[f->ultimo-1]);
       else
              return 0xffffffff;   ///< invalid value
}

uint8_t fila_cheia (struct fila *f)
{
       if (f->ultimo == f->num) return 1;
       return 0;
}

uint8_t fila_vazia (struct fila *f)
{
       if (f->ultimo == 0) return 1;
       return 0;
}

int tamanho (struct fila *f) {
       return (f->num);
}

void deleta_fila (struct fila *f)
{
     if (f->elementos)
       free((void *)f->elementos);
     f->ultimo = f->num = 0;
     f->elementos = NULL;
}
```

12.3.3 *Buffer* Circular

A implementação em *software* do conceito de fila, em que a vacância de um lugar faz com que todos os elementos sucessores desloquem automaticamente de uma casa para frente, é usualmente complexa em termos da quantidade de acessos à memória. Uma forma de contornar o custo de deslocamentos de todos os elementos quando se remove um elemento de uma fila de tamanho fixo *n* é utilizar *buffers* circulares. Um ***buffer* circular** (*circular buffer*) é uma fila de tamanho fixo que não tem fim nem início, como mostra Figura 12.7. Esta estrutura dispõe de dois indexadores, ou ponteiros, para manipular os dados armazenados no *buffer*. Eles são chamados ***head*** e ***tail***, e são ilustrados como duas setas na Figura 12.7. O produtor (de dados) **coloca** (*put*) um dado no *buffer*, incrementando o indexador de acesso *head*, e o consumidor (de dados) **consome** (*remove*) um dado do *buffer*, incrementando o indexador de acesso *tail*.

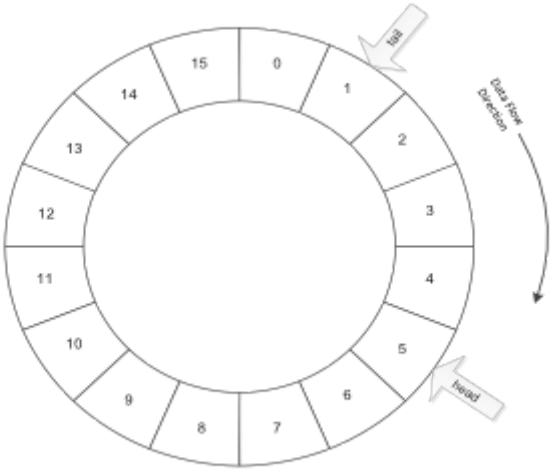

Figura 12.7: *Buffer* circular (Fonte: [4])

Podemos implementar esta estrutura com um arranjo de tamanho fixo, operando os seus índices com aritmética modular (%), de forma que o índice de acesso volta automaticamente para 0 quando se chega ao último elemento do arranjo. Uma implementação é mostrada no seguinte trecho de código em C:

```c
#include <stdlib.h>

struct bufCircular {
	unsigned int head;
	unsigned int tail;
	unsigned int num;
	int *elementos;
};

void cria_bufCircular (struct bufCircular *c, unsigned int n)
{
	c->head = 0;
	c->tail = 0;
	c->num = n;
	if (n == 0)
		c->elementos = NULL;
	else
		c->elementos = (int *)malloc(sizeof(int)*n);
	return;
}

uint8_t puxa (struct bufCircular *c, int v)
{
	unsigned int ind = (c->head+1)%c->num;
	if (ind == c->tail) return 0;
	c->elementos[c->head] = v;
	c->head = ind;
	return 1;
}
```

```
int remove (struct bufCircular *c)
{
	int v;
	if (c->head != c->tail) {
		v = c->elementos[c->tail];
		c->tail = (c->tail+1)%c->num;
		if (c->head == c->tail) c->head = c->tail = 0;
		return v;
	} else {
		return 0xffffffff;		///< invalid value
	}
}

int tail (struct bufCircular *c)
{
	if (c->head != c->tail) {
			return(c->elementos[c->tail]);
	} else {
		return 0xffffffff;		///< invalid value
	}
}

int head (struct bufCircular *c)
{
	if (c->head != c->tail) {
		if (c->head == 0) return (c->elementos[c->num-1]);
			return(c->elementos[c->head-1]);
	} else {
		return 0xffffffff;		///< invalid value
	}
}

uint8_t bufCircular_cheia (struct bufCircular *c)
{
	if ((c->head+1)%c->num == c->tail) return 1;
	return 0;
}

uint8_t bufCircular_vazia (struct bufCircular *c)
{
	if (c->head == c->tail) return 1;
	return 0;
}

int tamanho (struct bufCircular *c)
{
     return (c->num);
}

void deleta_bufCircular (struct bufCircular *c)
{
     if (c->elementos)
        free((void *)c->elementos);
     c->head = c->tail = c->num =0;
     c->elementos = NULL;
}
```

Observe que a simples troca do movimento de elementos de um *buffer* pelo movimento de ponteiros consegue reduzir drasticamente a quantidade de acessos à memória, principalmente quando *n* for muito grande. Esta vantagem tem levado à substituição da estrutura fila pelo *buffer* circular nos programas de sistemas embarcados, mesmo que a implementação da aritmética modular para manipular os índices seja um pouco mais trabalhosa. Hoje em dia, é uma estrutura muito utilizada em sistemas embarcados para transferência de dados entre processos de diferentes velocidades de processamento.

12.3.4 *Heap*

Vimos na Seção 6.5.1.2 as funções usadas para fazer alocação dinâmica de espaço de memória durante o tempo de execução de um programa. Para gerenciar de forma eficiente o dinamismo do processo sem impactar em demasia no desempenho do sistema, os dados são organizados numa estrutura denominada *heap*. ***Heap*** é uma **árvore binária especial armazenada numa estrutura linear de vetor.** Distinguem-se dois tipos de organização dos dados numa estrutura de árvore (Figura 12.8):

- **Max-heap**: a chave do nó-raíz de todas as suas sub-árvores deve ser maior do que as chaves de todos os seus nós-filho.
- **Min-heap**: a chave do nó-raíz de todas as suas sub-árvores deve ser menor do que as chaves de todos os seus nós-filho.

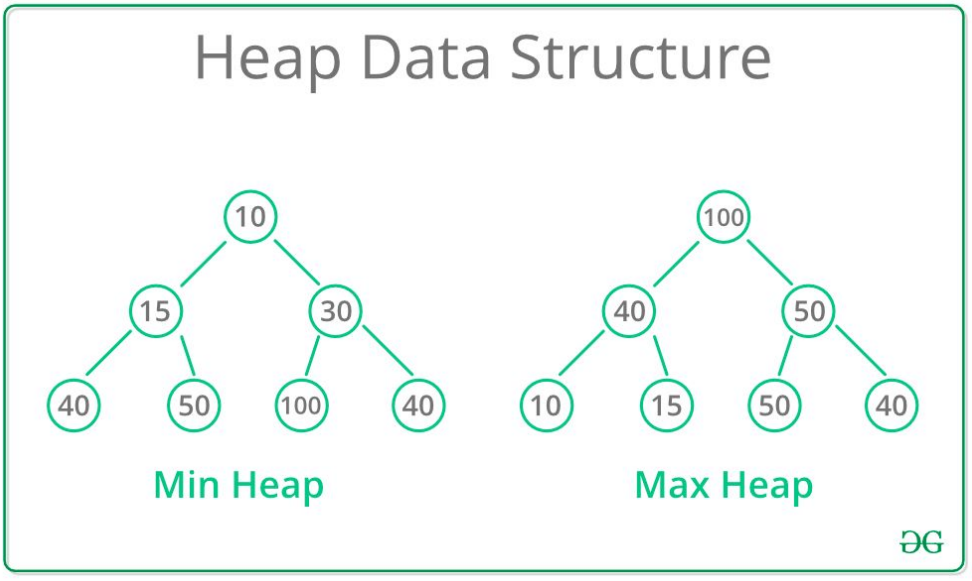

Figura 12.8: Duas formas de organização dos dados numa estrutura de árvore binária.

Para linearizar uma árvore binária, há essencialmente três formas, como ilustra a Figura 12.9: pré-ordenação (nó, sub-árvore esquerda, sub-árvore direita), in-ordenação (sub-árvore esquerda, nó, sub-árvore direita), pós-ordenação (sub-árvore esquerda, sub-árvore direita, nó).

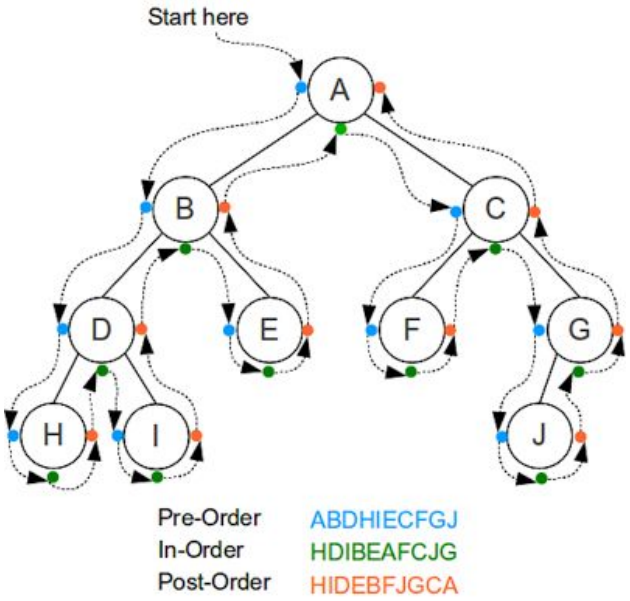

Figura 12.9: Linearização de uma estrutura binária.

Os espaços de memória demandados dinamicamente são alocados conforme o tamanho requisitado e ligados pelos ponteiros formando uma sequência de nós correspondente a uma árvore binária linearizada. Observe na Figura 12.10 uma implementação com uma lista duplamente ligada[2] [25], em que cada nó contém um ponteiro para o nó anterior (fwd_ptr) e o outro para o nó posterior (bwd_ptr) da sequência.

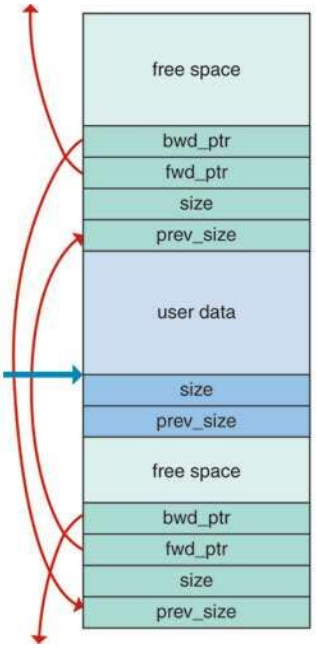

Figura 12.10: Implementação de uma estrutura *heap* com lista duplamente encadeada (Fonte: [24]).

[2] Lista ligada é um estrutura de dados em que um conjunto de registros são sequencialmente conectados pelos ponteiros/endereços. Quando todos os registros tem um campo de endereço onde é armazenado o endereço do próximo registro, não necessariamente contíguo, dizemos que é uma lista ligada simples. Quando todos os registros tem dois campos de endereço, um para o endereço do registro anterior e outro para o endereço do próximo registro, dizemos que é uma **lista duplamente ligada**.

Sendo uma alocação dinâmica sob demanda, é muito comum fragmentar o espaço de memória em blocos pequenos insuficientes para atender uma requisição, embora a soma desses blocos menores seja suficiente. Para recuperar os blocos “inutilizados”, reagrupando-os num único bloco maior, muitos dos sistemas lançam mão às técnicas de **defragmentador de memória** e de **coletor de lixo**, em inglês *garbage colector*. O defragmentador procura realocar os espaços de memória juntando os pequenos blocos fragmentados num bloco contíguo e o coletor de lixo procura remover os “lixos” (espaços) que foram alocados dinamicamente mas não devidamente removidos.

A seguir transcrevemos de [26] uma implementação do Min-heap em C utilizando somente um arranjo:

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

/*Declaring heap globally so that we do not need to pass it as an argument every time*/
/* Heap implemented  here is Min Heap */

int heap[1000000], heapSize;
/*Initialize Heap*/
void Init() {
   heapSize = 0;
   heap[0] = -INT_MAX;
}

/*Insert an element into the heap */
void Insert(int element) {
   heapSize++;
   heap[heapSize] = element; /*Insert in the last place*/
   /*Adjust its position*/
   int now = heapSize;
   while (heap[now / 2] > element) {
      heap[now] = heap[now / 2];
      now /= 2;
   }
   heap[now] = element;
}

int DeleteMin() {
   /* heap[1] is the minimum element. So we remove heap[1]. Size of the heap is
decreased.
```

```
    Now heap[1] has to be filled. We put the last element in its place and see if it fits.
    If it does not fit, take minimum element among both its children and replaces parent
with it.
    Again See if the last element fits in that place.*/
  int minElement, lastElement, child, now;
  minElement = heap[1];
  lastElement = heap[heapSize--];
  /* now refers to the index at which we are now */
  for (now = 1; now * 2 <= heapSize; now = child) {
     /* child is the index of the element which is minimum among both the children */
     /* Indexes of children are i*2 and i*2 + 1*/
     child = now * 2;
     /*child!=heapSize beacuse heap[heapSize+1] does not exist, which means it has only
one
       child */
     if (child != heapSize && heap[child + 1] < heap[child]) {
        child++;
     }
     /* To check if the last element fits ot not it suffices to check if the last element
       is less than the minimum element among both the children*/
     if (lastElement > heap[child]) {
        heap[now] = heap[child];
     } else /* It fits there */
     {
        break;
     }
  }
  heap[now] = lastElement;
  return minElement;
}
```

12.4 Exercícios

1. Represente os seguintes valores em complemento de 2 em 8 *bits* e em 32 *bits*: 0, 121, -56, 128.
2. Dada uma representação binária de um valor em ponto flutuante (64 bits). Como sabemos que o valor é NaN (infinito) e que o valor é zero?
3. Como seria a representação dos seguintes valores reais pelo padrão IEEE-754 32 *bits*: 12, 45,3, -4, -21.
4. Qual é o valor real codificado nos seguintes códigos do padrão IEEE-754:
    a. 00111111000000000000000000000000
    b. 11000010000000000110011001100110

5. Como seria, quando possível, a codificação da sequência de caracteres: “EA075 - Introdução ao Projeto de Sistemas Embarcados” nos sistemas de codificação de caracteres: ASCII, ASCII estendido, ISO-8859-1 e UTF-8?
6. Qual seria a saída do seguinte trecho de código em C? Justifique [29].

```
#include<stdio.h>
int main()
{
	struct simp
	{
		int i = 6;
		char city[] = "chennai";
	};
	struct simp s1;
	printf("%d",s1.city);
	printf("%d", s1.i);
	return 0;
}
```

7. Qual seria a saída do seguinte trecho de código em C? Justifique [29].

```
#include<stdio.h>
struct
{
	int i;
	float ft;
}decl;
int main()
{
	decl.i = 4;
	decl.ft = 7.96623;
	printf("%d %.2f", decl.i, decl.ft);
	return 0;
}
```

8. Qual é a saída do seguinte trecho de código em C? Justifique [29].

```
#include<stdio.h>
int main()
{
	struct bitfields
	{
		int bits_1: 2;
		int bits_2: 4;
		int bits_3: 4;
		int bits_4: 3;
	}bit = {2, 3, 8, 7};
	printf("%d %d %d %d", bit.bits_1, bits.bit_2, bit.bits_3, bits.bit_4);
}
```

9. Qual é a saída do seguinte trecho de código? Justifique [29].

```
void main()
 {
 struct bitfields {
 int bits_1: 2;
 int bits_2: 9;
 int bits_3: 6;
 int bits_4: 1;
 }bit;
 printf("%d", sizeof(bit));
 }
```

10. Qual é a saída do seguinte trecho de código? Justifique [29].

```
#include<stdio.h>
int main()
{
        struct leader
        {
        char *lead;
        int born;
        };
        struct leader l1 = {"AbdulKalam", 1931};
        struct leader l2 = l1;
        printf("%s %d", l2.lead, l1.born);
}
```

11. Em relação às estruturas de dados que apresentamos na Seção 12.3, qual(is) delas têm uma estrutura linear? E qual(is) delas têm uma estrutura dinâmica que facilita a inserção e a remoção de um elemento qualquer da estrutura?
12. Cite as aplicações das estruturas de dados pilhas, filas, *heaps* e *buffers* circulares em projetos de sistemas embarcados.
13. Em quais das estruturas de dados que apresentamos na Seção 12.3 não precisamos atualizar o conteúdo do espaço de memória quando um dos seus elementos for removido?
14. Analise o código de implementação da estrutura de dados heap na Seção 12.3.4. Qual é a lista ligada adotada para encadear dinamicamente os nós de dados?
15. O que você entende por defragmentador de memória e coletor de lixo no contexto de alocação dinâmica de espaços de memória.

12.5 Referências

[1] NXP. CodeWarrior Development Studio Common Features Guide
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/manuais/CWCFUG.pdf
[2] Chua Hock-Chuan. A Tutorial on Data Representation: Integers, Floating-point Numbers, and Characters.

https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html
[3] CS Education Research Group. Data Representation.
http://csfieldguide.org.nz/en/chapters/data-representation.html
[4] Wikipedia. ASCII.
http://en.wikipedia.org/wiki/ASCII
[5] Robson R. Linhares. Aspectos básicos de linguagem C.
http://www.dainf.cefetpr.br/~robson/prof/common/c/aspec.htm
[6] Técnicas de Linguagem de Programação. Conceitos Gerais - Tipos de dados.
http://www.prof2000.pt/users/famaral/ig/tlp/variaveis.htm
[7] Nigel Jones. Introduction to the volatile keyword.
http://www.embedded.com/electronics-blogs/beginner-s-corner/4023801/Introduction-to-the-Volatile-Keyword
[8] Programiz. Scope and Lifetime of a variable.
http://www.programiz.com/c-programming/c-storage-class
[9] C4Learn.com. C enum: Enumerated Types.
http://www.c4learn.com/c-programming/c-enum/
[10] Wikibooks. Programar em C/Ponteiros.
http://pt.wikibooks.org/wiki/Programar_em_C/Ponteiros
[11] Tep Dobry. Dynamic Allocation.
http://www-ee.eng.hawaii.edu/~tep/EE160/Book/chap14/section2.1.2.html
[12] Tutorialspoint. C – Structures.
http://www.tutorialspoint.com/cprogramming/c_structures.htm
[13] Fresh2fresh.com. C – Array.
http://fresh2refresh.com/c-programming/c-array/
[14] Tutorialspoint. C – Strings.
http://www.tutorialspoint.com/cprogramming/c_strings.htm
[15] Derrick Klotz. C for Embedded Systems Programming.
http://www.nxp.com/files/training/doc/dwf/AMF_ENT_T0001.pdf
[16] Diego Macedo. Conversões de Linguagens: Tradução, Montagem, Compilação, Ligação e Interpretação.
https://www.diegomacedo.com.br/conversoes-de-linguagens-traducao-montagem-compilacao-ligacao-e-interpretacao/
[17] Tutorialspoint. C-Structure. https://www.tutorialspoint.com/cprogramming/c_structures.htm
[18] Key shortcuts, Unicode and UTF-8 tables.
https://www.key-shortcut.com/en/writing-systems/%E6%96%87%E5%AD%97-chinese-cjk/cjk-characters-1
[19] ITead Studio. HC-05.
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea076/datasheet/Bluetooth_HC05.pdf
[20] Wikibooks. C Programming/String manipulation.
https://en.wikibooks.org/wiki/C_Programming/String_manipulation
[21] Studytonight. C Unions. https://www.studytonight.com/c/unions-in-c.php
[22] Data Structure and Algorithms - Stack.
https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
[23] Embedded Staff. Mastering stack and heap for system reliability: Part 1 - Calculating stack size.
https://www.embedded.com/mastering-stack-and-heap-for-system-reliability-part-1-calculating-stack-size/
[24] Embedded Staff. Mastering stack and heap for system reliability: Part 3 - Avoiding heap errors.

https://www.embedded.com/mastering-stack-and-heap-for-system-reliability-part-3-avoiding-heap-errors/
[25] Wikipedia. Doubly linked list. https://en.wikipedia.org/wiki/Doubly_linked_list
[26] FileFormat.Info. Complete Character List for UTF-8.
http://www.fileformat.info/info/charset/UTF-8/list.htm
[27] Jorge Stolfi. ISO-8859-1 (ISO Latin 1) Character Encoding.
https://www.ic.unicamp.br/~stolfi/EXPORT/www/ISO-8859-1-Encoding.html
[28] ASCII table , ascii codes. https://theasciicode.com.ar/.
[29] C Structures Questions. https://www.2braces.com/c-questions/structure-questions-c-1