Capítulo III – Valores e Tipos de Dados

"Nem todos os bits tem o mesmo valor."

Carl Sagan

Juntamente com os programas, dados são a matéria prima da computação. De fato, para que haja computação é necessário que os programas manipulem dados. A importância dos dados é tão fundamental na computação que, durante muito tempo, essa área foi popularmente conhecida como a área de processamento de dados.

Grandes massas de dados, tais como, dados geográficos, dados de catálogos telefônicos de cidades, dados de censo demográfico, imagens de satélite e cadastro de consumidores são de grande interesse econômico, chegando a custar várias vezes mais que o preço dos programas que os manipulam.

Linguagens de Programação utilizam os conceitos de tipos de dados para permitir a representação de valores em programas. O termo valor é utilizado aqui como sinônimo de dado. Cada linguagem adota um conjunto próprio de valores e tipos para permitir a representação de dados.

Um valor é qualquer entidade que existe durante uma computação, isto é, tudo que pode ser avaliado, armazenado, incorporado numa estrutura de dados, passado como argumento para um procedimento ou função, retornado como resultado de funções, etc. São exemplos de valores em C:

3 2.5 'a' "Paulo" 0x1F 026

Um tipo de dado é um conjunto cujos valores exibem comportamento uniforme nas operações associadas com o tipo. Por exemplo:

{ true, 25, ' b', "azul" } não corresponde a um tipo
{ true, false } corresponde a um tipo

Dizer que um valor é de um determinado tipo significa dizer que esse valor pertence ao conjunto de valores definido por aquele tipo. De maneira similar, dizer que uma expressão é de um determinado tipo significa dizer que o resultado dessa expressão é um valor pertencente ao conjunto definido por aquele tipo.

Um importante conceito para o entendimento de tipos de dados é a sua cardinalidade (#), isto é, o número de valores distintos que fazem parte do tipo. Por exemplo, a cardinalidade do tipo *boolean* de JAVA é 2. Ao lon-

go desse capítulo, o conceito de cardinalidade será usado na apresentação dos tipos de dados.

Nesse capítulo são discutidos os diversos tipos de dados que costumam ser adotados em linguagens de programação. Esses tipos são divididos em duas categorias principais (primitivos e compostos). Por sua vez, os tipos compostos são subdivididos em outras seis categorias. Além dos aspectos sintáticos e semânticos de cada tipo de dados, são apresentados modelos de implementação desses tipos.

3.1 Tipos Primitivos

Tipos primitivos (ou atômicos) são aqueles cujos valores não podem ser decompostos em outros valores de tipos mais simples. Os tipos primitivos são a base de todo sistema de tipos de uma linguagem, pois a partir deles é que todos os demais tipos podem ser construídos.

Embora linguagens de propósito geral devam oferecer tipos que permitam lidar com qualquer tipo de aplicação, a escolha dos tipos primitivos da LP ajuda a revelar a área de aplicação pretendida para a LP. Por exemplo, COBOL, destinada para o processamento comercial, possuía como tipos primitivos strings de comprimento fixo e números de ponto fixo. FORTRAN, destinada para a computação numérica, possui como tipos primitivos números reais com precisão variada.

Tipos Primitivos costumam ser definidos na implementação da LP. Logo, limitações e variações de hardware podem fazer com que diferentes implementações possuam conjunto de valores diferentes para um mesmo tipo de uma LP. Isto prejudica a portabilidade dos programas nestas LPs. Por exemplo, em C, o conjunto de valores do tipo *int* normalmente corresponde ao intervalo que pode ser representado com o número de bits da palavra do computador. Numa máquina de 16 bits, o tipo *int* varia de –32.768 a +32.767. Numa máquina de 32 bits, o tipo *int* varia de –2.147.483.648 a +2.147.483.647.

3.1.1 Tipo Inteiro

Um tipo inteiro corresponde a um intervalo do conjunto dos números inteiros. Em geral, existem vários tipos inteiros numa mesma LP. Normalmente, existe pelo menos um dos tipos inteiros que reflete exatamente as operações de inteiros fornecidas por hardware. Por exemplo, C pode possuir até 8 tipos inteiros resultantes da combinação dos tipos básicos *char* e *int* com os modificadores *signed*, *unsigned*, *short* e *long*. Geralmente, o tipo *int* ocupará o tamanho da palavra do computador e refletirá as operações aritméticas embutidas em hardware. Um compilador C típico para

uma máquina de 16 bits possuirá os tipos de dados inteiros apresentados na tabela 3.1:

Tipo	Tamanho (bits)	Intervalo	
		Início	Fim
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32768	32767
unsigned int	16	0	65535
signed int	16	-32768	32767
short int	16	-32768	32767
unsigned short int	16	0	65535
signed short int	16	-32768	32767
long int	32	-2147483648	2147483647
unsigned long int	32	0	4294967295
signed long int	32	-2147483648	2147483647

Tabela 3. 1 - Tipos Inteiros de um Compilador C

Ao analisar a tabela 3.1 pode se constatar que os tipos definidos com o modificador *short* estabelecem intervalos de valores iguais aos definidos pelo tipo de dado equivalente sem o uso desse modificador. De fato, a existência dos modificadores *short* e *long* revela a intenção de prover intervalos diferentes de inteiro, onde isso for prático; contudo, a especificação de C [KERNIGHAN & RITCHIE, 1989] não obriga que seja sempre assim. Normalmente, *int* terá o tamanho da palavra de uma determinada máquina. Em geral, *short* ocupa 16 bits, *long* ocupa 32 bits e *int*, 16 ou 32 bits. Cada compilador é livre para escolher tamanhos adequados ao próprio hardware, com as únicas restrições de que *short*s e *int*s devem ocupar pelo menos 16 bits, *long*s pelo menos 32 bits, e *short* não pode definir um intervalo maior que *int*, que não pode ser maior do que *long*.

A mesma constatação pode ser feita em relação ao modificador *signed*. Os tipos definidos na tabela 3.1 usando esse modificador também estabelecem intervalos de valores iguais aos definidos pelo tipo de dado equivalente sem o uso desse modificador. Isso também ocorre porque a especificação de C [KERNIGHAN & RITCHIE, 1989] não exige que o intervalo do tipo básico (por exemplo, o tipo *char*) englobe valores positivos e negativos. Essa decisão normalmente será dependente da máquina e do compilador. Assim, se o tipo *char* definir um intervalo com valores positivos e negativos, tal como na tabela 3.1, o modificador *signed* será redundante quando aplicado ao tipo *char*. Por outro lado, se o tipo *char* definir um intervalo de valores não negativos, o modificador *unsigned* é que será redundante.

É interessante compreender porque os criadores de C adotaram a postura de deixar para os implementadores dos compiladores a definição dos intervalos dos tipos, uma vez que isso claramente traz problemas para a portabilidade dos programas. A razão dessa escolha é enfatizar a eficiência. Tendo essa liberdade, os implementadores dos compiladores podem selecionar os intervalos dos tipos de modo a utilizar da melhor maneira possível os recursos de hardware disponíveis.

JAVA rompe com a tradição de deixar a definição do intervalo de inteiros para a fase de implementação dos compiladores. Como JAVA prioriza a portabilidade de programas, ela já define na própria LP os intervalos de valores que cada tipo inteiro deve representar. A tabela 3.2 mostra os tipos de dados inteiros de JAVA:

Tipo	**Tamanho (bits)**	**Intervalo**	
		Início	**Fim**
byte	8	-128	127
short	16	-32768	32767
int	32	-2.147.483.648	2.147.483.647
long	64	-9223372036854775808	9223372036854775807

Tabela 3. 2- Tipos Inteiros de JAVA

Normalmente, o modelo de implementação dos tipos inteiros adota a notação de complemento a dois quando o intervalo de valores inteiros inclui números positivos e negativos. Já quando o intervalo inclui apenas números não negativos, a notação binária normal é adotada. A tabela 3.3 mostra exemplos da correspondência entre representações binárias de números e o seu valor decimal correspondente ao se adotar a notação de complemento a dois e a própria notação binária.

Representação Binária	**Inteiros**	
	Notação de Complemento a Dois	**Notação Binária**
0000 0101	5	5
0000 0100	4	4
0000 0011	3	3
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	-1	255
1111 1110	-2	254
1111 1101	-3	253
1111 1100	-4	252
1111 1011	-5	251

Tabela 3. 3 - Notação de Complemento a Dois e Binária

Para converter a representação binária de um número em seu decimal correspondente segundo a notação binária, basta utilizar a regra de conversão de números na base binária para a base decimal. Por exemplo:

$$11111101 = 1x2^7+1x2^6+1x2^5+1x2^4+1x2^3+1x2^2+0x2^1+1x2^1 = 253$$

Para converter a representação binária de um número em seu decimal correspondente segundo a notação de complemento a dois é um pouco mais trabalhoso. Primeiramente, é necessário verificar se o dígito mais à esquerda do número é um ou zero. Se for zero, o número será zero ou positivo. Se for um, o número decimal será negativo. Nesse caso, deve-se inverter a representação binária do número e incrementá-lo de um. Assim, obtém-se a representação do número na base binária. Basta, então convertê-la para a base decimal levando em conta o seu sinal. Por exemplo:

11111101 (representação na notação de complemento a dois)
1 (dígito mais a esquerda –> número é negativo)
00000010 (representação com inversão binária)
00000001 (representação binária do número um)
00000011 (adição binária do número invertido e um)
$00000011 = 0x2^7+0x2^6+0x2^5+0x2^4+0x2^3+0x2^2+1x2^1+1x2^1 = 3$
11111101 = -3 (considerando o sinal)

As principais vantagens da notação de complemento a dois são ter uma representação única para o número zero e também poder utilizar os operadores aritméticos binários para implementar suas próprias operações.

A cardinalidade dos tipos inteiros corresponde ao número total de valores que podem ser representados no intervalo definido pelo tipo. Esse número é limitado pelo total de combinações de bits que podem ser formadas com o número de bits utilizados para representar os números do tipo. Assim, se são usados **n** bits para a representação dos números de um determinado tipo inteiro, a cardinalidade desse tipo será $\mathbf{2^n}$.

3.1.2 Tipo Caracter

Valores caracteres são armazenados em computadores como códigos numéricos. Para permitir o processamento de caracteres, algumas LPs fornecem um tipo primitivo cujos valores correspondem aos símbolos de uma tabela padrão de caracteres. Por exemplo, PASCAL e MODULA 2 oferecem o tipo *char*.

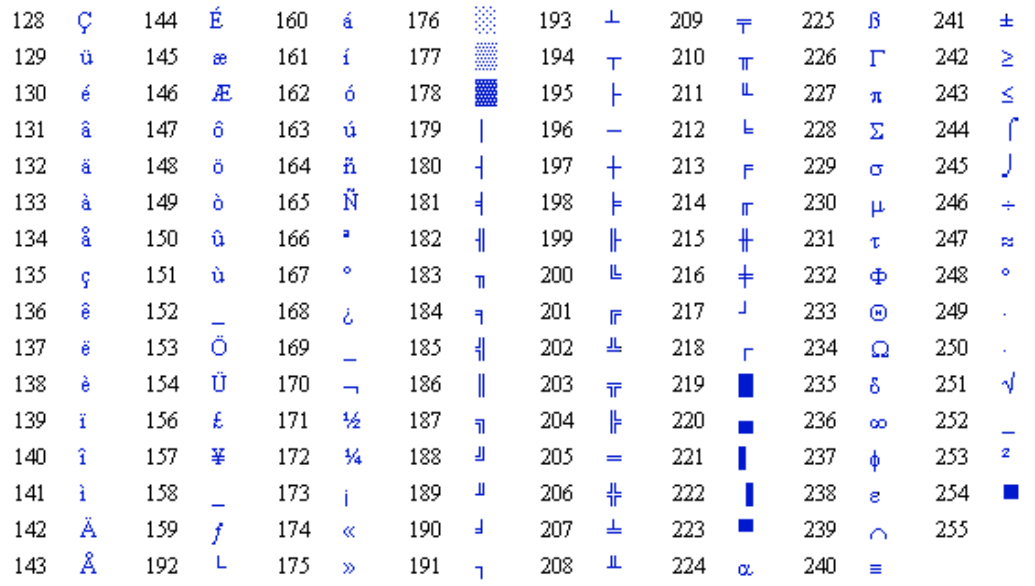
Existem várias tabelas padrão com códigos numéricos para caracteres, tais como, EBCDIC, ASCII e UNICODE. A tabela mais utilizada para codificação de caracteres é a ASCII ("American Standard Code for Information Interchange"), que usa os valores de 0..127, armazenados em 8 bits, para codificar 128 caracteres diferentes. A figura 3.1 mostra a tabela

ASCII padrão. Observe que os 32 primeiros códigos representam caracteres de controle. Note também que essa tabela não inclui caracteres acentuados.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Figura 3. 1 - Tabela ASCII Padrão

Por causa da globalização da economia e da necessidade de computadores se comunicarem com outros computadores ao redor do mundo, a tabela ASCII tem se tornado inadequada. Muitas vezes, para capturar caracteres internacionais não presentes na tabela ASCII, cria-se uma extensão específica da tabela ASCII, chamada ASCII estendida, através da inclusão de mais 128 caracteres à tabela padrão. Contudo, essa solução não é geral. A figura 3.2 apresenta uma extensão muito comum da tabela ASCII. Observe nessa tabela a inclusão de caracteres latinos acentuados, caracteres gregos e outros caracteres especiais.



128	Ç	144	É	160	á	176	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▒	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	|	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	168	¿	184	╕	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	169	_	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	▒	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	_
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

Figura 3. 2 - Extensão de Tabela ASCII

Uma nova tabela padrão, chamada UNICODE, foi desenvolvida recentemente como uma alternativa. Esta tabela utiliza 16 bits para armazenar caracteres da maioria das linguagens naturais existentes no mundo. Por exemplo, ela inclui caracteres usados na linguagem de países como a Sérvia, Japão, Tailândia e outros. Para manter a compatibilidade de programas baseados em ASCII, a tabela UNICODE engloba a tabela ASCII, isto é, ela utiliza os mesmos códigos numéricos para representar os caracteres da tabela ASCII padrão. JAVA adota a tabela padrão UNICODE.

Finalmente, é importante destacar que, embora C tenha um tipo primitivo *char*, esse tipo é classificado como um tipo inteiro, pois as operações realizadas sobre este tipo são as mesmas que podem ser realizadas sobre os demais tipos inteiros. De fato, os caracteres usados num programa C são na realidade sinônimos usados para representar os números correspondentes ao seu código na tabela padrão (tipicamente, ASCII). Isso é motivo de frequente confusão para os programadores iniciantes. Por exemplo, o seguinte trecho de código C atribui o valor inteiro *100* a variável *d* (o valor *100* corresponde a soma de *3* com o código numérico ASCII da letra minúscula *'a'*).

```
char d;
d = 3 + 'a';
```

Por outro lado, a opção adotada por C permite criar código mais facilmente redigível e eficiente em algumas situações. Em PASCAL e MODULA-II, por exemplo, para obter o código ASCII de um caracter, é necessário chamar uma função. Isso não é necessário em C.

A cardinalidade do tipo caracter é igual ao número de entradas na tabela de caracteres adotada.

3.1.3 Tipo Booleano

O tipo primitivo booleano é o tipo de dados mais simples que pode existir numa LP. Ele possui apenas dois valores, um correspondente a verdadeiro e outro a falso. Tipos booleanos são tipicamente usados como resultados de expressões condicionais ou como variáveis identificadoras de estado, popularmente conhecidas como "flags".

Com exceção de C, o tipo booleano tem sido incluído na maioria das LPs. Em C, as expressões numéricas podem ser usadas como condicionais. Neste caso, todos os operandos com valores **diferentes de zero** são considerados **verdadeiro** e todos os operandos com **valor zero** são considerados **falso**.

Embora outros tipos, tais como os inteiros de C, possam ser usados para atingir os mesmos propósitos para os quais o tipo booleano é indicado, o uso de tipos booleanos torna o programa mais legível e impede a ocorrência de erros, tais como:

if (c += 1) x = 10;

Esse trecho de código é um comando legal em C, mas pode ser decorrente de um erro de digitação (o programador poderia ter teclado + ao invés de =). Muito embora JAVA seja uma linguagem fortemente baseada em C, por causa do tipo de erro acima e da questão de legibilidade, ela inclui o tipo de dado *boolean*.

Um valor booleano pode ser representado por um único bit, contudo, como um único bit de memória é difícil de acessar eficientemente em muitas máquinas, eles são frequentemente armazenados na menor célula eficientemente endereçável de memória, tipicamente um byte.

Obviamente, a cardinalidade do tipo booleano é dois.

3.1.4 Tipo Decimal

O tipo primitivo decimal armazena um número fixo de dígitos decimais. A localização do ponto decimal é estabelecida arbitrariamente, pela LP, pelo próprio hardware ou pelo programador, em alguma posição das células de memória que armazenam o seu valor.

O tipo decimal é essencial para aplicações comerciais por ser capaz de armazenar precisamente valores decimais. Por conta disso, o tipo decimal é um tipo fundamental em COBOL e também é oferecido por ADA. Em-

bora limitado a um intervalo restrito, o tipo decimal garante que as operações sobre seus valores são sempre precisas. Isto não pode ser feito com números reais armazenados usando a notação de ponto flutuante.

Tipos decimais são representados como cadeia de caracteres, usando-se códigos binários para os dígitos decimais. Estas representações são chamadas BCD (`binary coded decimal`). Em alguns casos, elas são armazenadas usando um byte por dígito, mas em outros podem empacotar dois dígitos em cada byte, visto que para representar binariamente qualquer dígito decimal são necessários apenas 4 bits. A figura 3.3 mostra um exemplo de modelo de implementação para um tipo decimal:

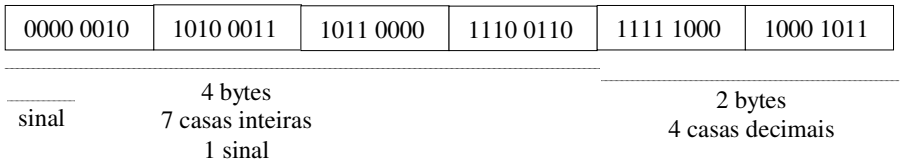


Figura 3. 3 - Representação Binária de Número Decimal

A representação BCD de números reais sempre ocupa mais memória do que a representação binária correspondente. Ela ocupa no mínimo 4 bits para codificar um dígito decimal. Por exemplo, para armazenar o número 18 na representação BCD é necessário utilizar 8 bits. Contudo, na representação binária, só são necessários 5 bits para armazenar o mesmo número.

Assim, as desvantagens do tipo decimal são o reduzido intervalo de valores decimais que pode ser representado (pois expoentes não podem ser utilizados) e o desperdício da memória usada pela representação.

As operações sobre valores decimais são realizadas por hardware, quando a máquina já as tem embutidas, ou emuladas por software.

A cardinalidade do tipo decimal é função do número total **n** de dígitos inteiros e fracionários do tipo. Como cada casa decimal pode ser um dos dez algarismos e como os números decimais podem ser positivos ou negativos, a cardinalidade corresponde a $\mathbf{2 \times 10^n}$ valores. Por exemplo, no caso do tipo decimal da figura 3.3, a cardinalidade seria de 2×10^{11}.

3.1.5 Tipo Ponto Flutuante

O tipo primitivo ponto flutuante modela os números reais. Como a representação em ponto flutuante é finita, números reais como π só podem ser representados aproximadamente.

Como números reais não possuem correspondência numa representação binária direta, valores de ponto flutuante necessitam ser representados

através de uma notação que combina representações binárias de frações e expoentes.

Operações de ponto flutuante são fornecidas pelo hardware. Máquinas atuais usam o padrão IEEE 754 para ponto flutuante. Como pode ser visto na figura 3.4, esse padrão define dois tipos de ponto flutuante: precisão simples e precisão dupla. O tipo de precisão simples é o de uso mais comum e ocupa 32 bits. O tipo de precisão dupla só é usado em situações onde maiores expoentes e partes fracionárias são necessários e ocupa 64 bits.

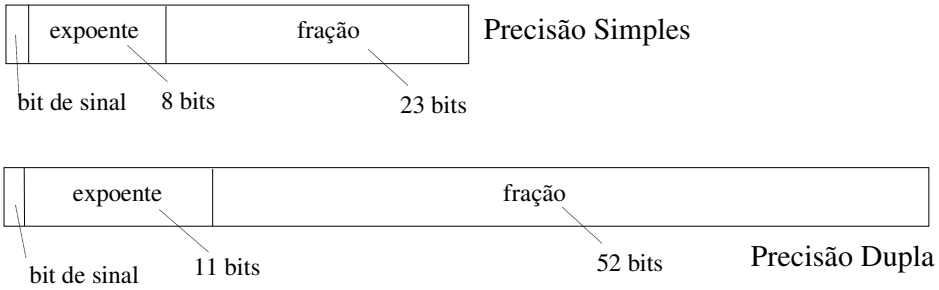


Figura 3. 4 - Padrão IEEE 754

Os implementadores de LPs normalmente usam a representação fornecida pelo hardware. A maioria das LPs inclui dois tipos de ponto flutuante, frequentemente chamados de *float* e *double*. JAVA possui esses dois tipos e adota o padrão IEEE para cada um deles. A tabela 3.4 ilustra as características dos tipos de dados *float* e *double* de JAVA.

Tipo	**Número de Bits**	**Intervalo**	
		Início	**Fim**
float	32	±3.40282347 E +38	±1.40239846 E –45
double	64	±1.79769313486231570 E +308	±4.94065645841246544 E –324

Tabela 3. 4 - Tipos de Dados Ponto Flutuante de JAVA (extraída de [FLANAGAN, 1997])

A cardinalidade dos tipos *float* e *double* é limitada pelo número de bits usado em cada uma das representações. Para evitar que a cardinalidade do tipo de ponto flutuante seja reduzida pelo fato de um mesmo número poder ser representado de formas diferentes (por exemplo, 0.01×10^{1} e 1.0×10^{-1}), a representação em ponto flutuante é sempre normalizada, isto é, o dígito mais significativo do número nunca deve ser zero. Por outro lado, algumas representações binárias específicas são utilizadas para a representação de valores especiais, tais como, zero, infinito e não números (por exemplo, a raiz quadrada de números negativos). Assim, o número de valores distintos dos tipos de ponto flutuante é sempre inferior ao total de

configurações binárias distintas disponíveis. Por exemplo, a cardinalidade de *float* é inferior a 2^{32}.

Uma explicação detalhada a respeito da representação em ponto flutuante e de como são implementadas suas operações aritméticas se encontra além do escopo desse livro. Maiores informações podem ser obtidas em [GOLDBERG, 1991].

3.1.6 Tipo Enumerado

Em algumas LPs, tais como PASCAL, ADA, C e C++, é permitido que o programador defina novos tipos primitivos através da enumeração dos identificadores que denotarão os valores do novo tipo. Por exemplo, em C é possível definir o seguinte tipo:

enum mes_letivo {mar, abr, mai, jun, ago, set, out, nov};

Tipos enumerados possuem correspondência direta (uma relação de 1 para 1) com intervalos de tipos inteiros e podem ser usados para indexar vetores e para contadores de repetições. De fato, em C e C++, os valores enumerados são convertidos implicitamente para números inteiros, sendo aplicados sobre eles as mesmas operações que regem os tipos inteiros.

Tipos enumerados são utilizados basicamente para aumentar a legibilidade e confiabilidade do código. A legibilidade é aumentada porque identificadores de valores são mais facilmente reconhecidos do que códigos numéricos. Por exemplo, num programa de processamento bancário é mais fácil identificar os bancos específicos se os descrevemos por seu nome ao invés de usar seus códigos numéricos.

A confiabilidade é aumentada porque valores fora da enumeração não são válidos. Além disso, operações aritméticas comuns não podem ser realizadas sobre estes tipos, permitindo que o compilador identifique possíveis erros lógicos e tipográficos. Como C e C++ tratam enumerações como inteiros, estas duas vantagens de confiabilidade não estão presentes.

Os projetistas de JAVA fizeram uma opção por simplificar a linguagem e não incluíram o tipo enumerado de C e C++.

A cardinalidade de um tipo enumerado corresponde ao número de identificadores usados na enumeração do tipo.

3.1.7 Tipo Intervalo de Inteiros

Em algumas LPs, tais como PASCAL e ADA, também é possível definir tipos intervalo de inteiros. Por exemplo, em PASCAL, é possível definir:

type meses = 1 .. 12;

Tipos intervalos herdam as operações dos inteiros. De fato, como as variáveis intervalo podem ser atribuídas por variáveis inteiras, e vice-versa, analisando estritamente, elas podem ser consideradas como um subtipo do tipo de dados inteiro.

As vantagens do tipo intervalo são praticamente as mesmas dos tipos enumerados. A legibilidade é aumentada a partir do momento que fica mais claro qual intervalo de valores que o tipo pode assumir. A confiabilidade também é aumentada porque a atribuição de valores fora do intervalo pode ser verificada estática e dinamicamente.

3.2 Tipos Compostos

Tipos compostos são aqueles que podem ser criados a partir de tipos mais simples. São exemplos os registros, os vetores, as listas, os arquivos, etc. David Watt [WATT, 1990] utiliza os conceitos de produto cartesiano, uniões, mapeamentos, conjuntos potência e tipos recursivos para classificar e explicar os tipos compostos. Essa também é a abordagem seguida aqui.

3.2.1 Produto Cartesiano

Consiste na combinação de valores de tipos diferentes em tuplas. São produtos cartesianos os registros de PASCAL, MODULA 2, ADA e COBOL e as estruturas de C. O exemplo 3.1 ilustra produtos cartesianos através do uso de estruturas em C:

```
struct nome {                       struct empregado {
        char primeiro [20];                 struct nome nfunc;
        char meio [10];                     float salario;
        char sobrenome [20];        } emp;
};
```

Exemplo 3. 1 - Produtos Cartesianos em C

Na *struct empregado*, os identificadores *nfunc* e *salario* são seletores utilizados para acessar os componentes da tupla. Eles liberam o programador de ter que lembrar a ordem dos componentes.

Para referenciar diretamente os componentes da tupla, utilizam-se normalmente referências completas. Por exemplo, para acessar o campo *meio* em C, é necessário usar:

```
emp.nfunc.meio
```

Algumas LPs possuem mecanismos para abreviar referências. Por exemplo, em PASCAL, pode-se utilizar o comando *with*:

```
WITH emp.nfunc DO BEGIN
  WRITE (primeiro, meio, sobrenome);
END;
```

A operação de atribuição entre registros é frequentemente permitida em LPs, tais como, PASCAL, ADA e C, bem como as comparações de igualdade e desigualdade. Muitas vezes, também é permitida a inicialização do registro com um agregado de valores. Em C, por exemplo:

```
struct data { int d, m, a; };
struct data d = { 7, 9, 1999 };
```

A figura 3.5 ilustra os conjuntos de valores dos tipos S, T e do produto cartesiano S x T:

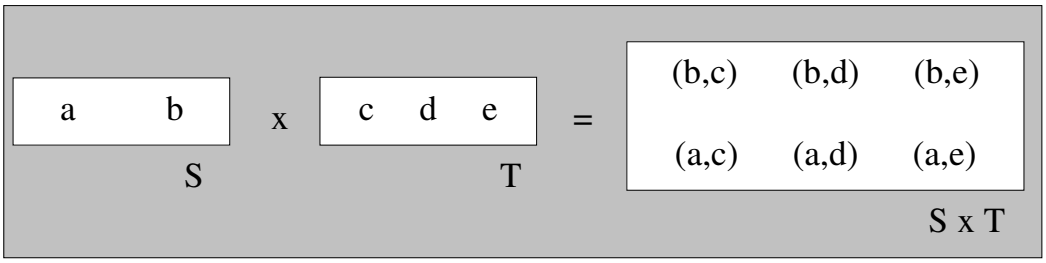


Figura 3. 5 - Produto Cartesiano de Dois Tipos (extraída de [WATT, 1990])

A cardinalidade do produto cartesiano é dada pelo produto da cardinalidade de seus componentes:

$$\#(S \times T) = \#S \times \#T$$

Generalizando, a cardinalidade de um produto cartesiano de **n** componentes é dada por:

$$\#(S_1 \times S_2 \times \ldots \times S_n) = \#S_1 \times \#S_2 \times \ldots \times \#S_n$$

No caso particular onde todos os componentes são do mesmo tipo (tupla homogênea), a cardinalidade é dada por:

$$\#(S^n) = (\#S)^n$$

Em LPs orientadas a objetos, produtos cartesianos são definidos a partir do conceito de classe. Embora C++ contenha classes, ela mantém as *struct* em decorrência da necessidade de manter compatibilidade com C. Contudo, isso introduz uma redundância na LP. JAVA, por sua vez, só possui o conceito de classe.

O modelo de implementação de produtos cartesianos consiste normalmente do armazenamento dos valores do produto em campos adjacentes de memória. O acesso é realizado através do par (endereço, deslocamento).

3.2.2 Uniões

Consiste na união de valores de tipos distintos para formar um novo tipo de dados. Um exemplo de uso de uniões pode ocorrer na definição de uma tabela de constantes de um compilador. Esta tabela pode ser composta com os diversos identificadores de constantes e os seus respectivos valores. Como constantes podem ser de tipos diferenciados, o campo valor da tabela pode ser implementado como uma união dos possíveis tipos de constantes. Uniões podem ser livres ou disjuntas.

3.2.2.1 Uniões Livres

Nas uniões livres pode haver interseção entre o conjunto de valores dos tipos que formam a união. No tipo união resultante haverá um único valor correspondente aos valores que são comuns aos diversos tipos. São exemplos de uniões livres os tipos resultantes do comando *EQUIVALENCE* de FORTRAN e *union* de C e C++.

Um problema existente com o uso de uniões livres é a possibilidade de violação do sistema de tipos da LP. O exemplo 3.2 ilustra uma união livre em C:

```
union medida {
    int centimetros;
    float metros;
};
union medida medicao;
float altura;
medicao.centimetros=180;
altura = medicao.metros;          // erro
printf("\n altura : %f metros\n", f);
```

Exemplo 3. 2 - União em C

No exemplo 3.2, o programador atribuiu um valor de 180 centímetros a variável *medicao* e posteriormente utiliza o valor de *medicao* em metros, o qual não havia sido atribuído, com consequências imprevisíveis para o resultado do programa. Este tipo de uso de uniões livres não pode ser verificado pelo compilador C. Por causa deste tipo de insegurança, e por poder utilizar herança para agrupar valores de tipos distintos, JAVA não possui uniões.

O conjunto de valores de uma união livre é determinado pela composição dos valores distintos de cada componente da união. Portanto, quando os componentes são disjuntos, a cardinalidade de uma união livre é dada pela soma das cardinalidades de seus componentes. Por outro lado, quando existe interseção entre o conjunto de valores dos componentes, a cardina-

lidade de uma união livre é dependente das possíveis interseções existentes entre os conjuntos de valores dos componentes. A figura 3.6 mostra a união livre de dois tipos de dados disjuntos.

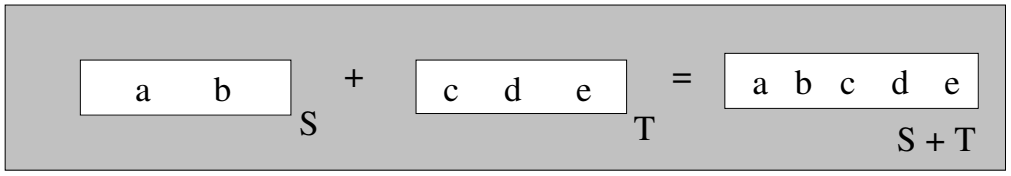


Figura 3. 6 - Uniões Livres sem Interseção

A cardinalidade da união apresentada na figura 3.6 é dada por:

$$\#(S + T) = \#S + \#T$$

Generalizando, a cardinalidade de uma união livre de **n** componentes é dada por:

$$\#(S_1 + S_2 + ... + S_n) = \#S_1 + \#S_2 ... + \#S_n$$

Essas fórmulas não se aplicam no caso de haver interseção, como na figura 3.7:

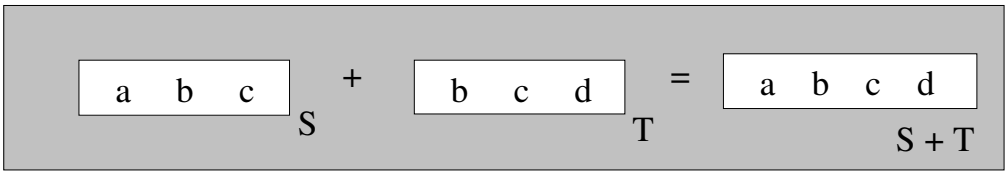


Figura 3. 7 - Uniões Livres com Interseção

O modelo de implementação de uniões livres frequentemente consiste em reservar espaço suficiente em memória para abrigar o componente da união que requer mais espaço e compartilhá-lo com os demais componentes. A figura 3.8 ilustra como a união livre *medicao* do exemplo 3.2 é armazenada na memória. Enquanto todos os 32 bits são usados para armazenar o componente *metros*, somente os 16 primeiros bits são usados para armazenar o componente *centimetros*.

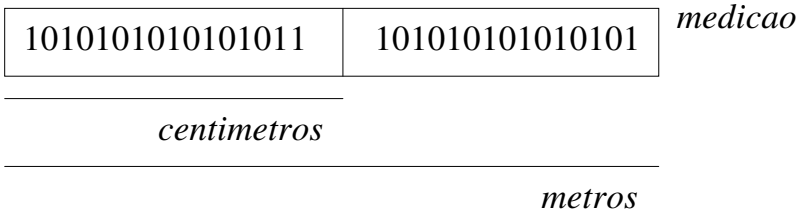


Figura 3. 8 - Implementação de União em C

3.2.2.2 Uniões disjuntas

Nas uniões disjuntas não há possibilidade de haver interseção entre o conjunto de valores dos tipos que formam a união. Tipicamente se utiliza um campo marcador, chamado de `tag`, para identificar qual o tipo originário do valor da união disjunta. São exemplos de uniões disjuntas os tipos associados aos registros variantes de PASCAL, MODULA 2 e ADA, à cláusula *REDEFINES* de COBOL e às *union* de ALGOL 68.

A figura 3.9 mostra a forma sintática dos registros variantes de PASCAL e MODULA 2.

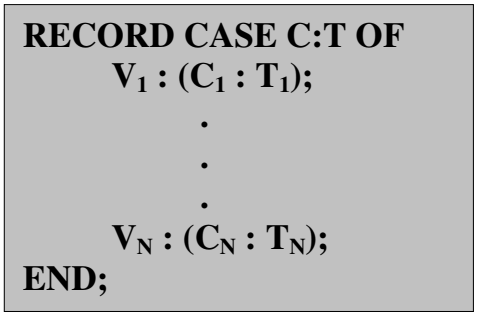


Figura 3. 9 - Sintaxe de Registros Variantes de PASCAL e MODULA-2

Na figura 3.9, os valores V_1, ..., V_n cobrem todos os possíveis valores do tipo primitivo discreto T do `tag`. O conjunto de valores desse tipo de registro variante é dado pela união dos valores pertencentes a T_1, T_2, ... , T_n. Considere o exemplo 3.3 em PASCAL:

```
TYPE Representacao = (decimal, fracionaria);
     Numero = RECORD CASE Tag: Representacao OF
          decimal: (val: REAL);
          fracionaria: (numerador, denominador: INTEGER);
     END;
```

Exemplo 3. 3 - União Disjunta em PASCAL

O conjunto de valores do tipo *Numero* é formado por:

{..., *decimal (-0.33), ... , decimal (-1.5), ... , decimal (0.6),* ...}

$\cup$

{..., *fracionaria (-1,3), ... , fracionaria (-3,2), ... , fracionaria (3,5),* ...}

Os valores *decimal* e *fracionaria* são as `tags` dos tipos da união.

Em PASCAL, a `tag` do registro variante e os componentes da variante podem ser acessados do mesmo modo como componentes de registros, provocando uma notória insegurança na programação.

Considere que uma variável *Num* do tipo *Numero* tem o valor *decimal 1.5*. Logo, *Num.Tag* possui o valor *decimal* e *Num.val* o valor *1.5*. O programa pode tentar acessar *Num.numerador*, o qual não existe correntemente. Isto provoca um tipo de erro de execução muito desagradável, quando a verificação dinâmica é implementada pelo compilador. Outro problema é que uma atribuição de *fracionaria* para *Num.Tag* provoca o efeito colateral de destruir *Num.val* e criar *Num.numerador* e *Num.denominador* com um valor indefinido. Portanto, o valor de *Num* pode ser mudado num único passo de *decimal (1.5)* para *fracionaria (?,?)*.

ADA torna o uso de uniões disjuntas mais seguro que em PASCAL e MODULA-2. ADA exige que todos os registros variantes tenham `tags`. Além disso, o programador não pode criar uniões inconsistentes porque o `tag` não pode ser atribuído separadamente. Mais ainda, ADA requer que qualquer referência ao campo variante seja verificada com relação à consistência do `tag`.

A cardinalidade de uniões disjuntas é obtida tal como no caso das uniões livres com inexistência de interseção entre os conjuntos de valores dos componentes, ou seja, através da soma da cardinalidade dos componentes. A figura 3.10 ilustra a união disjunta de dois tipos. Os sinais + e & indicam os valores do campo `tag`.

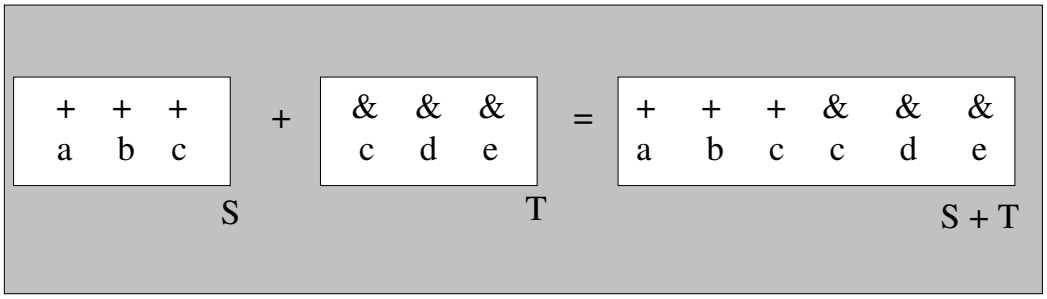


Figura 3. 10 - União Disjunta (adaptada de [WATT, 1990])

O modelo de implementação das uniões disjuntas é o mesmo das uniões livres.

Em PASCAL, MODULA-2 E ADA, registros misturam os conceitos distintos de Produto Cartesiano e Uniões. Considere o exemplo 3.4 em PASCAL:

```
TYPE TipoProduto = (musica, livro, video);
     Compra = RECORD
        valor: REAL;
       CASE produto: TipoProduto OF
          musica: (numeromusicas: INTEGER );
          livro: (numeropaginas: INTEGER);
```

```
        video: (duracao: INTEGER, colorido: BOOLEAN);
    END;
```

Exemplo 3. 4 – Produto Cartesiano com União Disjunta em PASCAL

O cardinalidade do tipo *Compra* é dado por:

#REAL x (#INTEGER + #INTEGER + (#INTEGER x #BOOLEAN))

São exemplos de valores do tipo *Compra*:

(25.00, musica (16))
(35.00, livro (257))
(40.00, video (121, TRUE))

3.2.3 Mapeamentos

Mapeamentos são tipos de dados cujo conjunto de valores corresponde a todos os possíveis mapeamentos de um tipo de dados S em outro T (que pode ser do mesmo tipo).

A notação S → T simboliza o conjunto de todos os possíveis mapeamentos distintos de S para T. A figura 3.11 mostra dois mapeamentos distintos de S para T.

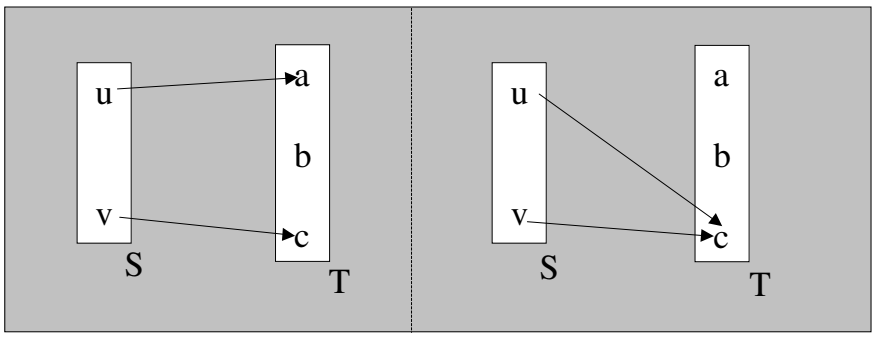


Figura 3. 11 - Mapeamentos (extraída de [WATT, 1990])

O conjunto de valores do mapeamento S → T é

{ ((u,a),(v,a)), ((u,b), (v,b)), ((u,c),(v,c)), ((u,a),(v,b)), ((u,a),(v,c)), ((u,b),(v,a)), ((u,b), (v,c)), ((u,c),(v,a)), ((u,c),(v,b)) }

É possível entender um mapeamento como sendo um produto cartesiano de #S elementos, no qual cada elemento pode assumir qualquer valor do tipo T. Logo, a cardinalidade de S → T é expressa por:

$$\#(S \rightarrow T) = \underbrace{\#T \times \#T \times \ldots \times \#T}_{\#S \text{ vezes}} = (\#T)^{\#S}$$

3.2.3.1 Mapeamentos Finitos

São mapeamentos no qual o conjunto domínio é finito. Os vetores podem ser vistos como sendo um mapeamento finito do conjunto índice para o conjunto dos elementos do vetor. Considere as linhas de código seguintes em PASCAL:

ARRAY S OF T; $[S \rightarrow T]$
A: ARRAY [1..50] OF CHAR; $A: ([1,50] \rightarrow CHAR)$

Nesse trecho de código qualquer valor possível do vetor corresponde a um mapeamento do conjunto índice (intervalo inteiro de 1 a 50) para o conjunto dos caracteres. A figura 3.12 ilustra um desses possíveis mapeamentos.

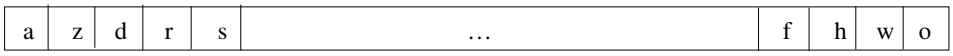


Figura 3. 12 - Vetor como Mapeamento Finito

Os elementos dos vetores são acessados através de indexação. As notações mais comuns utilizadas para indexação são parênteses ( ) e colchetes [ ]. O uso de parênteses pode provocar confusão porque normalmente chamadas de funções também os usam.

O conjunto índice deve ser finito (valores não podem ser strings de tamanho variável) e discreto (não podem ser números reais). A maior parte das LPs restringe o conjunto índice a um intervalo dos inteiros. Algumas fixam o limite inferior. C, C++ e JAVA, por exemplo, estabelecem 0 (zero) como o índice inicial. Outras permitem a definição dos limites pelo programador. PASCAL E ADA permitem que o conjunto índice seja de qualquer tipo primitivo discreto (inteiro, enumerado ou intervalo).

Em geral, o erro devido ao uso de índice inexistente só pode ser verificado em tempo de execução. Algumas LPs, como PASCAL, MODULA-2, ADA e JAVA fazem a verificação dinâmica do índice. Enquanto essa opção aumenta a confiabilidade dos programas escritos na LP, ela reduz a eficiência de execução porque qualquer acesso ao vetor requer o teste do índice. Isso se agrava bastante quando o acesso ocorre dentro de uma repetição. Por conta dessa perda de eficiência, LPs como C, C++ e FORTRAN não fazem a verificação dos índices e acabam comprometendo a confiabilidade. Portanto, é importante ter cuidado redobrado ao programar nessas LPs porque erros como o apresentado no código C a seguir não serão detectados.

```
int v[7];        // cria vetor de 7 posicoes com indices de 0 a 6
v[13] = 198;     // atribuicao valida c/ consequencias imprevisiveis
```

Quatro categorias de vetores podem ser definidas de acordo com o tamanho e tempo de definição do vetor e do momento e local de sua alocação na memória. A tabela 3.5 apresenta as características de cada uma dessas categorias.

Categoria de Vetor	Tamanho	Tempo de Definição	Alocação	Local de Alocação	Exemplos
Estáticos	Fixo	Compilação	Estática	Base	FORTRAN 77
Semi-Estáticos	Fixo	Compilação	Dinâmica	Pilha	PASCAL, C, MODULA 2
Semi-Dinâmicos	Fixo	Execução	Dinâmica	Pilha	ALGOL 68, ADA
Dinâmicos	Variável	Execução	Dinâmica	Monte	APL, PERL

Tabela 3. 5 - Categorias de Vetores

A coluna Exemplos da tabela 3.5 indica que a LP citada possui essa categoria de vetor. Embora se tenha classificado a LP naquela categoria que lhe é mais característica, isso não significa que ela só possua essa categoria de vetores. Por exemplo, C também permite a implementação de vetores estáticos.

Vetores estáticos são alocados no início do programa numa posição fixa da memória, chamada de base, e permanecem ali durante toda a execução. Eles apresentam como principal vantagem a eficiência de execução, visto que não requerem a alocação e desalocação de memória. Por outro lado, LPs que só possuem vetores estáticos, como FORTRAN 77, consomem mais memória do que é necessário, visto que vetores usados apenas em regiões do programa têm de ficar alocados durante toda a execução. Um exemplo de vetor estático em C é:

```
void f () {
  static int x[10];
```

Vetores semi-estáticos são alocados na pilha sempre que o bloco onde estão declarados começa a ser executado. Eles são mais econômicos com o uso da memória pois só alocam o vetor na região do programa onde são necessários. Contudo, essa política implica numa redução da eficiência de execução. Além disso, programas recursivos podem multiplicar o consumo de memória. Um exemplo de vetor semi-estático em C é:

```
void f () {
    int x[10];
```

Vetores semi-dinâmicos também são alocados na pilha sempre que o bloco onde estão declarados começa a ser executado. No entanto, o tamanho do vetor só é conhecido no momento da alocação. A grande vantagem do vetor semi-dinâmico sobre o vetor semi-estático é a flexibilidade que ele

proporciona. Nesta categoria, o programador não necessita definir um tamanho máximo para o vetor. A seguir, um exemplo de vetor semi-dinâmico em ADA é apresentado. Nele, a variável inteira *tam* é lida e, em seguida, o vetor *lista* é criado com *tam* elementos inteiros.

```
get (tam);
declare
   lista: array (1..tam) of integer;
   begin
```

Vetores dinâmicos são alocados em qualquer ponto da execução do programa na região de monte, também conhecida como `heap`. Além disso, o tamanho do vetor pode ser modificado durante a execução, em qualquer ponto onde uma atribuição é feita ao vetor. A redução e aumento do vetor é implementada através da alocação de um novo espaço de memória para o vetor, da cópia do conteúdo (se for o caso) e da desalocação da memória anteriormente reservada para ele. Um exemplo de vetores dinâmicos em APL é dado a seguir. Nesse exemplo, um vetor é criado inicialmente com três elementos e posteriormente tem seu tamanho aumentado para cinco elementos através de uma nova atribuição.

```
A ← (2  3  4)
A ← (2  3  4  15  20)
```

Se flexibilizarmos a exigência que vetores semi-dinâmicos sejam alocados na pilha, também é possível implementá-los em C, C++ e JAVA. Em C, usa-se ponteiros e as funções padrão *malloc* e *free* para gerenciar a alocação de memória. Em C++, usa-se ponteiros e os operadores *new* e *delete* para esta tarefa. Em JAVA, basta criar um objeto do tipo vetor. O exemplo 3.5 mostra como vetores semi-dinâmicos podem ser implementados em C, C++ e JAVA.

Em C:

```
void f (int a) {
   int *p;
   p = (int *)
     malloc (a * sizeof(int));
   p[0] = 10;
   free (p);
}
```

Em C++:

```
 void f (int a) {
    int *p;
    p =
       new int[a];
    p[0] = 10;
    delete[ ] p;
}
```

Em JAVA:

```
 void f (int a){
    int p[];
    p =
       new int[a];
    p[0] = 10;
 }
```

Exemplo 3. 5 –Vetores Semi-Dinâmicos em C, C++ e JAVA

Esse mesmo mecanismo pode ser usado para a implementação de vetores dinâmicos.

O uso desse mecanismo em C e C++, para a implementação de vetores semi-dinâmicos e dinâmicos, é algo tortuoso. Essas LPs deixam ao encargo do programador a tarefa de gerenciar a alocação e desalocação de memória, o que torna a programação desses vetores muito mais complexa e suscetível a erros.

Observe que em JAVA isso não ocorre. O programador não necessita usar explicitamente o conceito de ponteiros para criar os vetores. Além disso, JAVA possui coletor de lixo, o que exime o programador de ter de desalocar explicitamente a memória usada. Dessa forma, a abordagem de JAVA é mais redigível e confiável.

A maioria das LPs também possui vetores multidimensionais, conhecidos como matrizes. Um componente de uma matriz **n**-dimensional é acessado usando **n** valores do conjunto índice. De fato, podem-se encarar matrizes como tendo um único índice que é uma tupla. Assim, matrizes continuam sendo um mapeamento de um conjunto índice para o conjunto dos elementos. Note que o conjunto índice é formado pelo produto cartesiano dos conjuntos de valores de cada componente da tupla. Considere o seguinte exemplo em C:

int mat [5][4];

O conjunto de valores desse tipo é:

$\{0, \ldots, 4\} \times \{0, \ldots, 3\} \rightarrow \text{int}$

E sua cardinalidade é:

$$(\#\text{int})^{\#(\{0, \ldots, 4\} \times \{0, .., 3\})} = (\#\text{int})^{(\#\{0, \ldots, 4\} \times \#\{0, .., 3\})} = (\#\text{int})^{5 \times 4} = (\#\text{int})^{20}$$

Os elementos de vetores multidimensionais são acessados através da especificação de seus índices. Por exemplo, vetores bidimensionais são acessados através da especificação da linha e coluna do elemento. Uma representação gráfica para o vetor bidimensional *mat* é dada na figura 3.13.

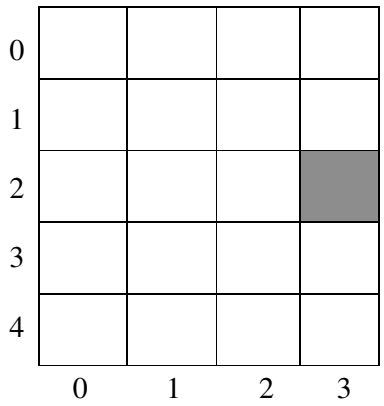


Figura 3. 13 - Representação Gráfica de Vetor Bidimensional

Note que o primeiro elemento da tupla especifica a linha do elemento que será acessado. O segundo elemento especifica a coluna do elemento. Assim, o elemento *mat [2] [3]* é indicado na figura 3.13 pela célula preenchida com cinza.

Os elementos de vetores unidimensionais são armazenados em posições contíguas de memória. Vetores multidimensionais são normalmente armazenados como um vetor unidimensional de tamanho igual ao número de células do vetor multidimensional. Tipicamente, vetores bidimensionais são armazenados por linha, isto é, armazena-se primeiramente os elementos da primeira linha, depois os da segunda, e assim por diante. Logo, o elemento a ser acessado pode ser obtido através da seguinte fórmula:

posição mat [i] [j] = endereço de mat [0][0] + i ×tamanho da linha + j ×tamanho do elemento = endereço de mat [0][0] + (i ×número de colunas + j) ×tamanho do elemento

É importante notar que o acesso a elemento via vetor multidimensional sempre implica no cálculo dinâmico da posição do elemento na memória. Isso é menos eficiente do que acessar o elemento de um vetor unidimensional. Em certas situações, onde se quer ter o máximo de eficiência computacional, pode-se considerar substituir a representação multidimensional por uma unidimensional.

Na maioria das LPs um vetor multidimensional é regular, isto é, o número de elementos de cada dimensão é fixo. Por exemplo, numa matriz bidimensional, cada linha possui o mesmo número de colunas. Em JAVA, além dos vetores multidimensionais regulares, é possível criar vetores onde uma ou mais dimensões podem ter número de elementos variado. Isto se torna possível porque em JAVA vetores multidimensionais são na realidade vetores unidimensionais cujos elementos são outros vetores. Veja o exemplo 3.6 em JAVA:

```
int [] [] a = new int [5] [];
for (int i = 0; i < a.length; i++) {
    a [i] = new int [i + 1];
}
```

Exemplo 3. 6 - Vetores Multidimensionais em Escada em JAVA

O exemplo 3.6 cria um vetor bidimensional em forma de escada (uma matriz triangular inferior esparsa). A primeira linha tem um elemento, a segunda tem dois elementos e assim por diante. Observe que LPs como PASCAL e MODULA-2 não permitem esse tipo de construção, uma vez que vetores devem possuir elementos de mesmo tipo e tamanho. C possi-

bilita a construção desse tipo de estrutura de dados através do uso de ponteiros.

Cada LP oferece um conjunto particular de operações que podem ser realizadas sobre vetores. Operações comuns são indexação, inicialização, atribuição, comparação de igualdade e desigualdade. Além da indexação, C apenas fornece a operação de inicialização:

```
int lista [] = {4, 5, 7, 83};
char name [] = "frederico";
char * names [] = {"Leo", "Marcos", "Isa"};
```

Além das operações mencionadas acima, ADA oferece também a concatenação de vetores.

Algumas LPs, como FORTRAN 90, permitem a manipulação de parte do vetor (um subvetor) Este tipo de operação é chamada de `slicing`. Nos trechos de código seguintes são criados em FORTRAN 90 um vetor *VET* de 8 elementos inteiros e uma matriz *MAT* de 4 linhas e 4 colunas com elementos inteiros. Mostra-se também como pode-se obter partes dessas estruturas de dados.

```
INTEGER VET (1:8), MAT (1:4, 1:4)
VET (2:5)          VET(2:10:2)          VET((/5, 3, 2, 7/))
MAT (3, 1:3)
```

VET(2:5) produz um vetor composto do segundo, terceiro, quarto e quinto elementos de *VET*. *VET(2:8:2)* produz um vetor com o segundo, quarto, sexto e oitavo elemento de *VET*. *VET((/5, 3, 2, 7/))* produz um vetor com o quinto, terceiro, segundo e sétimo elementos de *VET*, nessa ordem. *MAT(3, 1:3)* produz um vetor com os elementos das três primeiras colunas da terceira linha de *MAT*.

APL é a linguagem que fornece o conjunto mais amplo de operações sobre vetores. Dentre as operações pré-definidas estão a soma, subtração, multiplicação, transposição e inversão de matrizes.

3.2.3.2 Mapeamentos através Funções

Outra forma de mapeamento em LPs é através de funções. Uma função implementa um mapeamento $S \rightarrow T$, através de um algoritmo, o qual toma qualquer valor em S e computa sua imagem em T. O conjunto S não necessita ser finito.

O conjunto de valores do tipo mapeamento $S \rightarrow T$ são todas as funções que mapeiam o conjunto S no conjunto T. Considere o exemplo 3.7 em JAVA:

```
boolean positivo (int n) {
   return n > 0;
}
```

Exemplo 3. 7 - Mapeamento por Funções em JAVA

A função do exemplo 3.7 é um valor do tipo mapeamento do conjunto dos inteiros para o conjunto dos valores booleanos [int→boolean]. Outros valores desse tipo poderiam ser funções que implementem as abstrações par, ímpar, primo, palíndromo, múltiplo de 3, etc. O descritor de tipo do mapeamento (como, por exemplo, [int→boolean]), é normalmente conhecido pelo termo de assinatura da função.

Numa LP, funções podem ser valores de primeira ou segunda classe. Elas são valores de primeira classe quando se pode fazer com elas as mesmas coisas que se pode fazer com valores de outros tipos, por exemplo, criar variáveis e estruturas de dados daquele tipo, atribuir valores, passar como parâmetro, retornar como resultado de uma função, etc. Elas são valores de segunda classe quando existem restrições arbitrárias na LP que impedem algum desses usos. Por exemplo, em PASCAL é possível passar funções como parâmetros, mas não se pode criar variáveis do tipo função.

C utiliza o conceito de ponteiros para manipular endereços de funções como valores. O exemplo 3.8 utiliza uma função *conta*, a qual possui um parâmetro função, para calcular o número de ímpares, negativos e múltiplos de 7 num vetor *vet* de 10 elementos:

```
int impar (int n){ return n%2; }
int negativo (int n) { return n < 0; }
int multiplo7 (int n) { return !(n%7); }
int conta (int x[], int n, int (*p) (int) ) {
    int j, s = 0;
    for (j = 0; j < n; j++)
        if ( (*p) (x[j]) ) s++;
    return s;
}
main() {
    int vet [10];
    printf ("%d\n", conta (vet, 10, impar));
    printf ("%d\n", conta (vet, 10, negativo));
    printf ("%d\n", conta (vet, 10, multiplo7));
}
```

Exemplo 3. 8 - Uso de Funções como Valores em C

Embora use um mecanismo complicado, C trata funções como valores de primeira classe, pois é tanto possível criar estruturas de dados cujos ele-

mentos são do tipo ponteiro para função, quanto passar parâmetros para subprogramas do tipo ponteiro para função, tal como ilustrado no exemplo 3.8.

JAVA não trata funções (métodos) como sendo valores. Métodos não são dados e não podem ser manipulados por programas. Contudo, métodos podem ser passados para outros métodos através da passagem de uma instância de uma classe que defina o método.

Na maioria das LPs uma função pode ter **n** parâmetros. Quando chamada, devem ser passados **n** valores. Pode-se entender esse caso como se a função recebesse um único valor que é uma tupla. Por exemplo, a função C que possui o seguinte protótipo

float potencia (float b, int n);

define um mapeamento de [float x int $\rightarrow$ float].

Observe que a função *potencia* não pode ser passada como parâmetro para a função *conta* do exemplo 3.8 porque o seu tipo é diferente do tipo do parâmetro.

É importante lembrar ainda que se pode empregar algoritmos diferentes para implementar um mesmo mapeamento. Além disso, algumas vezes, vetores e funções podem ser usados para implementar o mesmo mapeamento finito. A escolha de um algoritmo específico ou a decisão sobre o uso de vetores ou funções dependerá de considerações sobre o tempo de resposta desejado e sobre o espaço de memória requerido.

3.2.4 Conjuntos Potência

Conjuntos potência são tipos de dados cujo conjunto de valores corresponde a todos os possíveis subconjuntos que podem ser definidos a partir de um tipo base S. A figura 3.14 mostra todos os valores de um conjunto potência (φS):

$$\varphi S = \{s \mid s \subseteq S\}$$

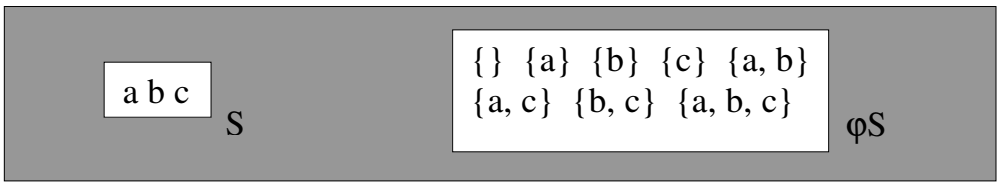


Figura 3. 14 - Conjunto Potência

Como cada valor em S pode ser membro ou não de um conjunto particular, a cardinalidade do conjunto potência de S é dada por

$$\#\varphi S = 2^{\#S}$$

As operações básicas que normalmente podem ser feitas sobre um tipo conjunto potência são as mesmas que podem ser feitas usando a teoria dos conjuntos: pertinência, contém, está contido, união, diferença, diferença simétrica e interseção. Além dessas, as operações de igualdade, desigualdade e atribuição também são válidas.

Nem todas as LPs oferecem tipos conjunto potência como um tipo predefinido. Mesmo quando fornecem esse tipo, eles são oferecidos de maneira restrita. Em PASCAL, por exemplo, só é permitido construir conjunto de valores de tipos discretos primitivos pequenos. Considere o exemplo 3.9 em PASCAL:

```
TYPE
   Carros = (corsa, palio, gol);
   ConjuntoCarros = SET OF Carros;
VAR
  Carro: Carros;
  CarrosPequenos: ConjuntoCarros;
BEGIN
  Carro:= corsa;
  CarrosPequenos:= [palio, gol];                 /*atribuicao*/
  CarrosPequenos:= CarrosPequenos + [corsa];     /*uniao*/
  CarrosPequenos:= CarrosPequenos * [gol];       /*intersecao*/
  if Carro in CarrosPequenos THEN                /*pertinencia*/
  if  CarrosPequenos >= [verde, azul] THEN       /*contem*/
```

Exemplo 3. 9 - Uso de Conjuntos em PASCAL

As restrições de PASCAL visam permitir uma implementação eficiente de conjuntos. As variáveis conjunto são armazenadas em uma palavra de memória e as operações são implementadas sobre cadeias de bits usando as instruções da máquina. Por exemplo, a figura 3.15 mostra como a união do código do exemplo 3.10 pode ser implementada através da operação de OR.

```
VAR S: SET OF [ 'a .. 'h' ];
BEGIN
    S := ['a', 'c', 'h'] + ['d'];
END;
```

Exemplo 3.10 - Operação de União em PASCAL

Observe na figura 3.15 que a representação binária indica quais elementos estão presentes no conjunto. Observe ainda que a operação ou binária produz em S o resultado esperado da união.

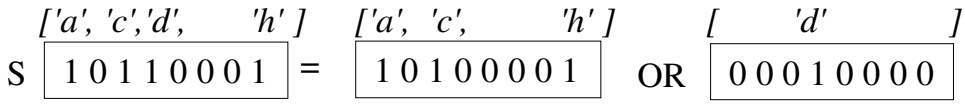


Figura 3. 15 - Implementação de Operação de União

LPs que não possuem tipos conjunto necessitam criar abstrações de dados que os implemente. Isto torna a programação mais trabalhosa e frequentemente menos eficiente. Embora C não possua tipos conjunto, ela fornece acesso a operações bit a bit que facilitam a sua implementação através de vetores.

Existe um grande número de problemas que poderiam ser resolvidos caso fosse permitido definir conjuntos de valores compostos (conjuntos de strings ou registros). Um exemplo de problema que seria facilmente resolvido é o de identificar os vocábulos comuns presentes em dois arquivos contendo texto. Bastaria inserir os vocábulos presentes em cada arquivo em dois conjuntos distintos e realizar a sua intersecção.

Contudo, a implementação de tipos conjunto com valores compostos não pode ser feita tão facilmente quanto em PASCAL porque esses valores não são enumeráveis, sendo difícil estabelecer uma relação um a um com vetores de bits. Normalmente, a implementação dos tipos conjunto tem de envolver listas ou tabelas `hash`, o que torna a implementação das operações de conjunto muito menos eficientes. Talvez seja por isso que a maioria das LPs opta por não incluir o tipo conjunto de valores compostos como tipo pré-definido da LP.

Uma abordagem que tem sido comum em LPs orientadas a objetos consiste em definir na biblioteca padrão uma classe (normalmente chamada de "set") que se comporte aproximadamente como um tipo conjunto potência. Nestas LPs os conjuntos podem envolver qualquer classe de objetos. Esta é a abordagem de SMALLTALK e JAVA. Contudo, nem todas as operações de conjunto mencionadas anteriormente costumam ser disponibilizadas.

3.2.5 Tipos Recursivos

Tipos recursivos são tipos de dados cujos valores são compostos por valores do mesmo tipo. Um tipo recursivo é definido em termos de si mesmo. De modo geral, o formato de definição de um tipo recursivo R é:

R ::= < parte inicial > R < parte final >

Por exemplo, listas são seqüências que podem ter zero ou mais valores. Formalmente, o tipo lista pode ser definido como sendo a união disjunta

do tipo lista vazia com os valores do produto cartesiano do tipo do elemento da lista com o tipo lista:

Tipo Lista ::= Tipo Lista Vazia | (Tipo Elemento x Tipo Lista)

Note que a definição do *Tipo Lista* usa o próprio *Tipo Lista*.

A cardinalidade de um tipo recursivo é infinita. Isto é verdade mesmo que o tipo do elemento da lista seja finito. O conjunto de valores do tipo listas é infinitamente grande (não podendo ser enumerado) embora toda lista individual seja finita.

Em PASCAL, C, C++ e ADA tipos recursivos devem ser definidos a partir de ponteiros. O exemplo 3.11 mostra a definição do tipo nó de uma lista de inteiros em C e C++:

```
struct no {                    class no {
    int  elem;                     int elem;
    struct no* prox;               no* prox;
};                             };
```

Exemplo 3. 11 - Definição do Tipo Recursivo Nó de Lista em C e C++

LPs puramente orientadas a objetos, tal como JAVA, permitem que tipos recursivos sejam definidos diretamente. Veja o exemplo 3.12 em JAVA:

```
class no {
    int  elem;
    no  prox;
};
```

Exemplo 3. 12 - Definição do Tipo Recursivo Nó de Lista em JAVA

3.2.5.1 Tipos Ponteiros

O uso de tipos ponteiros (também chamados de apontadores) não se restringe a estruturas de dados recursivas. Por exemplo, eles podem ser utilizados para simular vetores dinâmicos em C. Mesmo assim, eles são incluídos nesta subseção porque ocorrem mais frequentemente dentro dessas estruturas.

Ponteiro é um conceito de baixo nível relacionado com a arquitetura dos computadores. O conceito de ponteiros surgiu em decorrência da necessidade de se alocar memória de acordo com as demandas dinâmicas do programa e, desta forma, evitar o sub ou superdimensionamento do espaço de memória alocado.

É bem verdade que LPs funcionais, lógicas e orientadas a objetos contornam este problema sem que se tenha de incluir ponteiros como conceito da LP. Contudo, como LPs imperativas são fortemente influenciadas pelo

arquitetura de computadores, o conceito de ponteiros foi naturalmente herdado do conceito equivalente de endereçamento indireto dos `assemblers`.

Outro fator determinante na inclusão de ponteiros em LPs tem sido a eficiência da execução de programas. Implementações de LPs que não possuem ponteiros devem necessariamente incluir um sistema de gerenciamento de memória de modo a controlar o espaço utilizado e liberar o espaço que não esteja mais sendo utilizado. Esse sistema tende a tornar o processamento computacional do programa um pouco mais lento.

O conjunto de valores de um tipo ponteiro são os endereços de memória e o valor **nil**. O valor **nil** indica que o ponteiro não está referenciando qualquer célula de memória.

Ponteiros são valores de primeira classe em muitas LPs, tais como PASCAL, MODULA-2, C, ADA e C++. Nessas LPs pode-se criar variáveis do tipo ponteiro, passar ponteiros como parâmetros, incluí-los em estruturas de dados, etc. Com o uso de ponteiros é possível criar-se estruturas de dados complexas (um grafo, por exemplo) onde as conexões são representadas por ponteiros armazenados nos nós.

Tais estruturas complexas podem ser atualizadas para adicionar, remover um nó ou modificar a ligação entre nós através da manipulação de ponteiros. Isto é mais radical que a atualização em registros e vetores, pois nesses a atualização afeta os conteúdos das variáveis mas não a sua estrutura. Considere o programa C do exemplo 3.13. Nesse exemplo são criadas uma lista com número palíndromos e outra com números cuja soma dos dígitos que os formam é igual a dez. Observe que todos os números palíndromos fazem parte da outra lista. Note também que a atualização do ponteiro do nó de valor 343 para apontar para o nó de valor 181 provoca a remoção do nó 262 de ambas as listas.

```
#define nil 0
#include <stdlib.h>
#include <stdio.h>
typedef struct no* listaint;
struct no {
  int cabeca;
  listaint cauda;
};
listaint anexa (int cb, listaint cd) {
  listaint l;
  l = (listaint) malloc (sizeof (struct no));
  l->cabeca = cb;
  l->cauda = cd;
```

```
  return l;
}
void imprime (listaint l) {
   printf("\nlista: ");
  while (l) {
     printf("%d ",l->cabeca);
     l = l->cauda;
  }
}
main() {
   listaint palindromos, soma10, aux;
   palindromos = anexa(343, anexa(262, anexa(181, nil)));
  soma10 = anexa(1234, palindromos);
  imprime (palindromos);
  imprime (soma10);
  aux = palindromos ->cauda;
  palindromos ->cauda = palindromos ->cauda->cauda;
  free(aux);
  imprime (palindromos);
  imprime (soma10);
}
```

Exemplo 3. 13 - Uso de Ponteiros

As operações mais comuns sobre ponteiros são atribuição, alocação, desalocação e derreferenciamento. A atribuição pode ser feita entre variáveis do tipo ponteiro ou entre uma variável ponteiro e um endereço de memória. O exemplo 3.14 ilustra atribuições a ponteiros em C:

```
int *p, *q, r;    // dois ponteiros para int e um int
q = &r;           // atribui endereco onde variavel r esta alocada a q
p = q;            // atribui endereco armazenado em q a p
```

Exemplo 3. 14 - Atribuição em Ponteiros

Uma operação de alocação armazena dinamicamente um espaço de memória que será acessado via ponteiros e retorna o endereço da célula inicial alocada.

```
int* p = (int*) malloc (sizeof(int));
```

Na maioria das LPs existe uma operação de desalocação que força a liberação de áreas de memória que estão alocadas e apontadas pelo ponteiro. Em C, chama-se a função *free* da biblioteca padrão passando o ponteiro para a célula inicial da área de memória a ser desalocada.

```
free (p);
```

A operação de derreferenciamento retorna o conteúdo do que é apontado pelo ponteiro. A operação de derreferenciamento pode ser implícita (tal como em ALGOL 68 e FORTRAN 90) ou explícita (como em C ou PASCAL). Compare os códigos em FORTRAN 90 e C do exemplo 3.15:

```
INTEGER, POINTER :: PTR          int *p;
PTR = 10                         *p = 10;
PTR = PTR + 10                   *p = *p + 10;
```

Exemplo 3. 15 - Derreferenciamento Implícito em FORTRAN 90 e Explícito em C

Uma operação característica de C sobre ponteiros é a aritmética. O exemplo 3.16 mostra várias operações aritméticas sobre ponteiros.

```
p++;
++p;
p = p + 1;
p--;
--p;
p = p - 3;
```

Exemplo 3. 16 - Aritmética de Ponteiros em C

Essas operações deslocam o ponteiro de sua posição original. Por exemplo, se *p* aponta para uma variável que ocupa dois bytes, a instrução *p++* faz com que o ponteiro passe a apontar para uma posição dois bytes adiante na memória. Outra operação típica de C é o uso de indexação em ponteiros:

```
p[3];              // equivale a *(p + 3)
```

É possível entender a operação de indexação como um atalho sintático para a combinação da operação de adição a um ponteiro com a operação de derreferenciamento subsequente.

3.2.5.1.1 Ponteiros Genéricos

C inclui ainda uma categoria especial de ponteiros: aqueles que apontam para *void* (o tipo nulo). O tipo *void* é usado em C para indicar que uma função não retorna valor e para permitir a existência de ponteiros genéricos, isto é, ponteiros que podem apontar para valores de qualquer tipo. É importante destacar aqui que não é permitido criar-se variáveis do tipo *void*, apenas ponteiros para *void*:

```
void* p;                         // ponteiro generico
```

Ponteiros genéricos podem provocar problemas de erros de tipos se usados de maneira indiscriminada. Por exemplo, se um ponteiro genérico estiver apontando para uma célula que armazena um *float* e tentássemos acessá-la como um *int*, não haveria como verificar (ou avisar, no caso de

C) que estamos violando o sistema de tipos da linguagem. C evita este problema impedindo que os valores derreferenciados de ponteiros para *void* possam ser utilizados. O exemplo 3.17 mostra um erro dessa natureza identificado em tempo de compilação em C:

```
int f, g;
void* p;
f = 10;
p = &f;
g = *p;     // erro: ilegal derreferenciar ponteiro p/ void
```

Exemplo 3. 17 - Tentativa de Derreferenciamento de Ponteiro para Void em C

Ponteiros genéricos servem para criar funções genéricas que gerenciam a memória (por exemplo, *malloc*). Outra aplicação interessante é a criação de listas com elementos de tipos heterogêneos.

3.2.5.1.2 Problemas com Ponteiros

A programação com ponteiros requer atenção dos programadores para não provocar erros na sua manipulação. Esses erros podem ser bastante perigosos em determinadas situações. Os problemas mais comuns relacionados com a manipulação desse tipo de dados são:

a) Baixa Legibilidade: Ponteiros são conhecidos como o *GOTO* das estruturas de dados. A menos que se tenha muita disciplina e cuidado, a manipulação de ponteiros provoca muitos erros e é obscura quanto a seus efeitos. Por exemplo, numa atribuição de ponteiros

```
p->cauda = q;
```

não podemos saber, através de uma simples inspeção qual estrutura de dados está sendo atualizada. Seu efeito pode ser até radical, mudando toda a estrutura de dados através da introdução de um ciclo. Como vimos no exemplo das listas de palíndromos e de números cujos dígitos somam dez (exemplo 3.13), ponteiros permitem que duas ou mais estruturas de dados compartilhem os mesmos valores. Qualquer atualização de uma estrutura pode afetar diretamente as outras. Tais atualizações não ficam explícitas no código do programa, dificultando a legibilidade.

b) Erro de violação do sistema de tipos: Em LPs nas quais o tipo apontado pelo ponteiro não é restrito, expressões contendo ponteiros podem ser avaliadas com valores de tipos diferentes do esperado originalmente, provocando erros na avaliação em tempo de execução. Além de obscurecer a programação, isso também inviabiliza a checagem estática de tipos. O exemplo 3.18 em C:

```
int  i, j = 10;
p = &j;        // p aponta para a variavel inteira j
p++;           // p nao necessariamente aponta mais para um inteiro
i = *p + 5;  // valor imprevisivel atribuido a i
```

Exemplo 3.18 - Violação de Sistemas de Tipo pelo Uso de Ponteiros

c) Objetos Pendentes: Nesse tipo de problema, células de memória alocadas dinamicamente se tornam inacessíveis. O exemplo 3.19 mostra como objetos pendentes podem ser criados em C.

```
int* p = (int*) malloc (10*sizeof(int));
int *q = (int*) malloc (5*sizeof(int));
p = q;         // celula que era apontada por p torna-se inacessivel
```

Exemplo 3.19 - Objetos Pendentes em C

A ocorrência frequente de objetos pendentes provoca o problema de vazamento de memória, isto é, o espaço disponível para alocação vai se reduzindo.

d) Referências Pendentes: Nesse tipo de problema, um ponteiro possui como valor um endereço de uma variável dinâmica desalocada. Os exemplos a seguir mostram situações em C onde ocorrem referências pendentes. No caso mais frequente, ilustrado no exemplo 3.20, uma atribuição de ponteiros seguida de uma operação de desalocação deixa um ponteiro com referência pendente.

```
int* p = (int*) malloc(10*sizeof(int));
int* q = p;
free(p);       // q aponta agora para area de memoria desalocada
```

Exemplo 3.20 - Referência Pendente em C por Desalocação Explícita

Em outra situação, ilustrada no exemplo 3.21, uma referência a uma variável local é atribuída a uma variável com tempo de vida mais longo:

```
main() {
    int *p, x;
    x = 10;
    if (x) {
        int i;
        p = &i;
    }
    // i nao existe mais, mas p continua apontado para onde i estava
    // alocado
}
```

Exemplo 3.21 – Referência Pendente por Fim de Tempo de Vida

Outra causa potencial das referências pendentes é a falta de inicialização automática de ponteiros estabelecida na definição da LP. Neste caso, o programador pode esquecer de inicializar e acessar qualquer posição de memória, tal como no exemplo 3.22.

```
int* p;
*p = 0;          //  p aponta para um lugar desconhecido da memoria
```

Exemplo 3.22 - Referência Pendente por Falta de Alocação

3.2.5.2 Tipo Referência

Em C++ é possível criar variáveis do tipo referência. O conjunto de valores desse tipo é o conjunto de endereços das células de memória. O exemplo 3.23 mostra o uso de uma variável do tipo referência em C++:

```
int res = 0;
int& ref = res;        // ref passa a referenciar res
ref = 100;             // res passa a valer 100
```

Exemplo 3.23 - Tipo Referência em C++

Todas as variáveis que não são de tipos primitivos em JAVA são do tipo referência. Enquanto as variáveis do tipo referência em C++ não podem ter seu valor alterado após a inicialização, as variáveis de JAVA podem ser atribuídas a diferentes instâncias de objetos.

3.2.6 Strings

Strings correspondem a uma seqüência de caracteres. São tipicamente usadas para a realização de entrada e saída de dados e para armazenar dados não numéricos.

Strings são consideradas aqui como uma categoria a parte de tipos porque não existe consenso sobre como devem ser tratadas. Algumas LPs (como PERL, SNOBOL e ML) as tratam como tipos primitivos, fornecendo um conjunto específico e pré-definido de operações que podem ser realizadas sobre elas.

Outras LPs (tais como C e PASCAL) as tratam como mapeamentos finitos (tipicamente vetores de caracteres). As operações sobre strings são as mesmas fornecidas para o tipo vetor. Normalmente, estas LPs fornecem um conjunto de funções de biblioteca padrão para a manipulação de string. C, por exemplo, fornece funções para cópia de strings (*strcpy*), comparação (*strcmp*), etc.

JAVA considera strings como sendo um novo tipo, na verdade, uma classe da biblioteca padrão com as suas próprias operações.

Uma outra abordagem consiste em considerar uma string como o tipo recursivo lista. Esta abordagem é normalmente adotada por LPs (MIRANDA, PROLOG e LISP) que embutem o tipo lista como pré-definido na própria LP.

Operações comuns sobre strings são atribuição e comparação (PASCAL), concatenação e seleção de caracter ou substring (ADA). PERL oferece um conjunto poderoso de operações sobre string, como por exemplo, casamento de padrões através de expressão regular.

Existem três formas de implementação de strings:

- Estática: o tamanho da string é pré-definido ou definido em compilação e não é modificado durante a execução. Tipicamente, o resto da string é preenchido com brancos. Exemplos são as strings de COBOL.
- Dinâmica Limitada: o tamanho máximo da string é pré-definido ou definido em compilação, mas pode variar durante a execução até o máximo especificado. Em C, o final da string é indicado pelo caracter nulo *\0*. Em alguns dialetos de PASCAL, as strings possuem no índice 0, um caracter cujo código ASCII corresponde ao tamanho corrente da string.
- Dinâmica: o tamanho da string pode variar livremente durante a execução. Exemplos são as strings de PERL, SNOBOL e APL. Neste tipo de implementação sempre ocorre alocação e desalocação de memória quando há necessidade de aumentar uma string.

3.3 Considerações Finais

O passo inicial para aprender uma LP é estudar os tipos de dados oferecidos por ela. Esse estudo envolve saber qual o conjunto de valores de cada tipo, como esses tipos são armazenados na memória, quais operações podem ser feitas sobre cada tipo, como elas funcionam e como são implementadas, bem como identificar as limitações de cada tipo. Também é importante conhecer como os tipos podem ser combinados entre si para formar novos tipos de dados.

Nesse capítulo foi discutido em detalhe como os tipos de dados mais comuns se apresentam em LPs, como são implementados e quais operações são oferecidas sobre eles. Em especial, foi apresentada uma classificação abstrata que engloba a maior parte dos tipos de dados oferecidos pelas linguagens de programação. Essa classificação é reapresentada na figura 3.16.

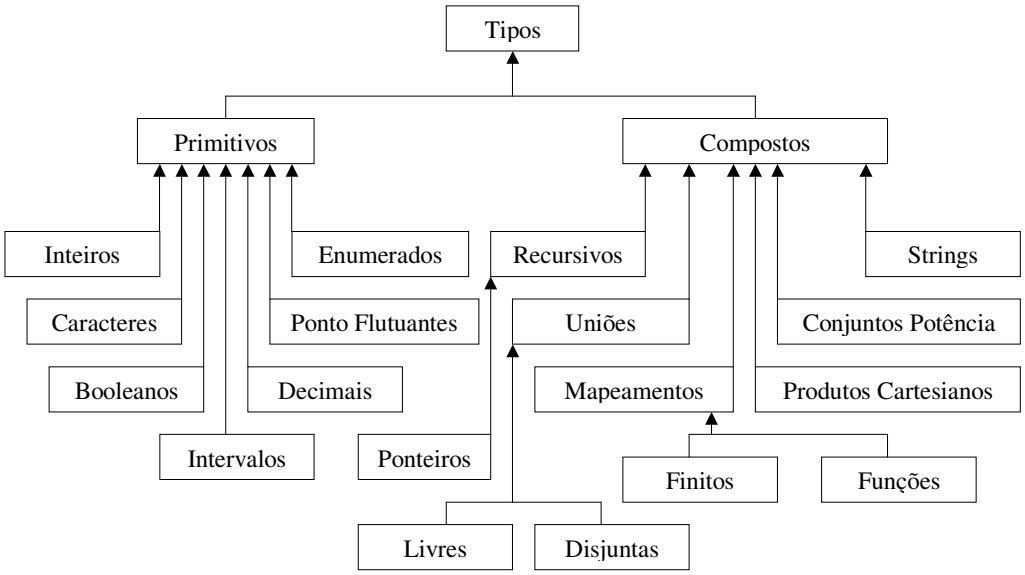


Figura 3. 16 - Classificação de Tipos de Dados

3.4 Exercícios

1. Ponteiros são causadores potenciais de erros em programação. Dê exemplos, com trechos de código em C, de erros causados por ponteiros que provocam violação dos sistemas de tipos da linguagem, ocorrência de objetos pendentes e ocorrência de referências pendentes.

2. Uma diferença significativa entre a definição de tipos primitivos em C++ e JAVA se refere ao intervalo de valores de cada tipo. Enquanto em JAVA os intervalos foram fixados na definição da LP, em C++ é a implementação do compilador que define esses intervalos. Compare estas duas abordagens, justificando a opção de cada uma dessas linguagens.

3. Em geral, a verificação de uso de índice fora dos limites do vetor só pode ser verificado em tempo de execução. Algumas LPs, como JAVA, PASCAL e MODULA-2 fazem a verificação dinâmica dos índices. Outras, como C, C++ e FORTRAN não fazem essa verificação. Justifique porque algumas LPs adotaram uma postura e outras adotaram uma postura oposta. Uma terceira postura, intermediária, seria gerar código com verificação dinâmica na fase de desenvolvimento e sem verificação dinâmica para a fase de uso. Discuta essa opção em termos dos conceitos usados para justificar as opções das LPs mencionadas acima.

4. *Arrays* podem ser estáticos, semi-estáticos, semi-dinâmicos e dinâmicos. Enquanto a criação de *arrays* estáticos e semi-estáticos pode ser feita facilmente em C, a construção de *arrays* semi-dinâmicos e dinâmicos envolve um maior esforço de programação. Responda como os mecanismos de C permitem a criação desses tipos de *arrays.* Ilustre com exemplos.

5. Produtos cartesianos, uniões, mapeamentos e tipos recursivos são categorias de tipos compostos de dados. Ilustre, com exemplos em C, cada um desses conceitos. Crie ainda um novo tipo de dados que combine três desses conceitos e diga qual a sua cardinalidade.

6. Determine a cardinalidade de cada um dos tipos abaixo, usando os conceitos de produto cartesiano, uniões e mapeamentos para explicar a cardinalidade dos tipos compostos:

```
enum sexo {masculino, feminino};
enum estado_civil {solteiro, casado, divorciado};
enum classe {baixa, media, alta};
enum instrucao {primario, secundario, superior};
union cidadania {
  enum classe c;
  enum instrucao i;
}
struct pessoa {
  enum sexo s;
  enum estado_civil e;
  union cidadania c;
};
struct amostra {
  int n;
  struct pessoa p[10];
}
```

7. Considere o seguinte programa escrito em C++:

```
#include <iostream>
int& xpto (int sinal) {
    int p = 4;
    if (!sinal) {
            p*=sinal;
    } else {
            p++;
    }
    return p;
```

```
}

void ypto () {
        int c[1000];
        int aux;
        for (aux = 0; aux < 1000; aux++) {
            c[aux] = aux;
        }
}

main() {
        int a = 1;
        int& b = xpto(a);
        ypto();
        cout << b;
}
```

Determine quais serão as saídas possíveis do programa acima. Explique sua resposta.

8. Considere o seguinte programa escrito em C:

```
#include <stdio.h>
int* calcula(int a){
   int p;
   p = a;
   if (a) {
          p*=3;
   } else {
           p++;
   };
   return &p;
}
main() {
   int x = 1;
   int* b = calcula(x);
   int* c = calcula (0);
   printf("%d\n", *b);
}
```

Descreva o que ocorre nesse programa. Justifique sua resposta.

9. Listas heterogêneas são estruturas de dados capazes de armazenar no seu campo de informação valores de tipos distintos. Uma forma de implementar listas heterogêneas em C é através do uso de uniões. Ou-

tra forma é através do uso de ponteiros para *void*. Mostre, através de exemplos de código em C, como se pode fazer para definir listas heterogêneas usando essas duas abordagens (não é preciso implementar as operações de lista, apenas a definição da estrutura de dados). Compare e discuta essas soluções em termos de redigibilidade (das operações da lista) e flexibilidade (em termos de necessidade de recompilação do módulo lista quando for necessário alterar ou incluir um novo tipo de dado no campo informação).

10. Em C é possível criar estruturas de dados heterogêneas, isto é, com elementos de tipos diferentes. Contudo, ao se retirar um elemento da estrutura é necessário identificar quais operações são válidas sobre o elemento sendo removido de modo a evitar erros no sistema de tipos da LP. De quais maneiras isso pode ser feito? Compare as soluções propostas em termos de redigibilidade e em termos da necessidade de alteração do código usuário quando da alteração ou inclusão de um novo tipo de elemento na lista.

11. Caracterize a diferença entre uniões livres e uniões disjuntas em termos de cardinalidade e segurança quanto ao sistema de tipos da linguagem. Discuta e exemplifique como as uniões de C podem ser utilizadas para criar estruturas de dados heterogêneas (isto é, que abrigam tipos de informação distintos), destacando como o programador (ou a linguagem, se for o caso) deve proceder para garantir o uso desse tipo de dado sem que haja violações do sistema de tipos. Discuta ainda quão genérica pode ser uma estrutura de dados heterogênea que se baseia no mecanismo de uniões.

12. Muito embora JAVA seja fortemente influenciada por C, os projetistas dessa LP resolveram incluir o tipo boolean, o qual não existe em C. Explique porque essa decisão foi tomada. Dê exemplo de situação na qual a postura de C traz alguma vantagem. Faça o mesmo em relação a postura de JAVA. Justifique suas respostas.