Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

12 de fevereiro de 2023

Introdução a Estrutura de Dados

Memórias: Bits e Bytes. Como os dados são representados no computador:

- O armazenamento é em um espaço limitado de memória, notação binária.
- O tamanho da estrutura representa o número de elementos representados.
- Um espaço de memória de 1 Byte (8 bits) permite 2^8(256) valores representados.
  - Inteiro sem sinal: 0 a 255
  - Inteiro com sinal (primeiro bit é sinal): -128 a 127
  - Símbolos: '*A*' ... '*Z*', '*a*' ... '*z*', '0' ... '9', ...
  - Real (ponto flutuante): 1 bit para sinal da mantissa, 1 bit para sinal do expoente, alguns bits para mantissa, alguns bits para expoente. 8 bits fica pouco, melhor 16!
- O tamanho básico da memória depende da arquitetura do processador: 8 bits, 16 bits, ... 64 bits.

Inteiros:

Sem sinal: $(10110101)_2 =$
$(1.2^7 + 0.2^6 + 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0)_{10} = 181_{10}$
Com sinal: $(10110101)_2 =$
$(-1.2^7 + 0.2^6 + 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0)_{10} = -75_{10}$

Com sinal: Complemento de 2:

- Se o primeiro bit é 0, a notação é igual ao exemplo "sem sinal".
- Se o primeiro bit é 1, para identificar o valor, inverte-se os bits, soma-se 1 e aplica a regra acima:
  $10110101 =$ (Complemento de 2) $(-)(01001010 + 1) =$
  $(-)(01001011)_2 = (-)(2^6 + 2^3 + 2^1 + 2^0)_{10} = -75_{10}$

Reais: (ponto flutuante)

Vamos considerar 32 bits:
24 bits para a mantissa em uma representação com sinal.
8 bits para o expoente em uma representação com sinal.
Escolha da mantissa: 100 pode ser representado como 1×10^2 ou 10×10^1. Vamos escolher de forma que a mantissa não tenha (na base 10) zeros à direita, ou seja, 1×10^2.
Alguns exemplos:

0:	00000000000000000000000000000000
100:	00000000000000000000000100000010
0.03:	00000000000000000000000011111110
123.400:	00000000000000010011010010000010

Quais os limites?
Quantos números significativos o número possui?

Caracteres e Símbolos:

Padrão ASCII: 8 bits (1 Byte)

Definição de Byte é o número de bits usados para representar um caracter

0	NULL	16	DLE	32	␣	48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	”	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	TAB	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	–	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

UTF8-Unicode é uma representação de 2 Bytes para caracteres, permite a maioria dos caracteres acentuados e símbolos em outras línguas como cirílico, grego. Também caracteres especiais.
UTF8-Unicode não é uma representação de tipo primitivo.

Operações em Tipos Primitivos

O processador (e linguagens de programação) já oferecem operações básicas sobre tipos primitivos.

```
int i, j, k;
double a, b, c;
char s, t;
unsigned int u, v;
long double x, y, z;
j = 1;
k = j + 2;
s = 'Z';
t = s - k;
```

Seriam todas as operações acima válida? **char - int** ???
É possível misturar tipos: double, int, unsigned int, char???

Operações em Tipos Primitivos

O processador (e linguagens de programação) já oferecem operações básicas sobre tipos primitivos.

```
int i, j, k;
double a, b, c;
char s, t;
unsigned int u, v;
long double x, y, z;
j = 1;
k = j + 2;
s = 'Z';
t = s - k;
```

Seriam todas as operações acima válida? **char** - **int** ???
É possível misturar tipos: double, int, unsigned int, char???

Alocação da memória: Variável

- Vamos supor, que temos um espaço de memória de 32 bit com o seguinte valor: 00100010010010110100011011010110
- O que isto representa? Depende do tipo de dado a que vamos atribuir este valor armazenado.
  - *double* (32bits): 2.247494×10^{-36}
  - *long int* (32bits): 575358678
  - *int* (2x16bits): 8779 e 18134
  - *char* (4x8bits): ", K, F, $\div$
- Uma variável representa, então um endereço na memória que possui um conjunto de bits
- Dependendo do tipo o espaço na memória é contado como 8 bits, 16 bits, 32 bits, ...

Tipos abstratos ou não primitivos

- Uma variável do tipo primitivo pode ser representada por um único endereço de memória.
- Dependendo do tipo, este endereço representa um espaço específico (8, 16, ...)
- Um tipo não primitivo, ou abstrato não consegue ser representado desta forma.
- Para um tipo abstrato é comum agruparmos um conjunto de tipo primitivos para representá-lo.
- Vamos tomar como exemplo um tipo abstrato que represente uma coordenada GPS (latitude e longitude)
  - Poderia ser representada por dois valores em pontos flutuantes: -14.646929, -39.069176
  - Ou usando GMS, 4 inteiros (GM), 2 pontos flutuantes (S) e 2 símbolos de caracter (hemisfério e lado): $14^o\ 38'\ 48.944''\ S$, $39^o\ 4'\ 9.034''\ W$.

Endereço e deslocamento

- Um tipo abstrato passa a ser representado, então, por um conjunto de tipos primitivos.
- Dizemos que este tipo é representado na memória por um endereço (onde aponta para todo o conjunto)
- Mas também para acessar cada parte do tipo, usamos um deslocamento.
- Este deslocamento pode ser, por posição, ou por elemento.
- No GPS: declaramos a variável (endereço) `GPS gps`. Isto, considerando `GPS` o tipo abstrato e `gps` a variável
  - Se tomamos a representação de 2 pontos flutuantes: podemos acessar por posição: `gps[0]` e `gps[1]`
  - Se tomamos a representação por GMS: podemos acessar por elementos: `gps.latitude.g` ou `gps.longitude.m`...
  - Ou híbrido: `gps[0].m`, `gps[1].s`

Tipos não primitivos ou abstratos → estruturas

- Como indicado, usamos um conjunto ou estrutura para representar um tipo não primitivo.
- Esta estrutura pode ser uma estrutura sequencial simples, que chamamos de vetor.
- Pode ser também uma estrutura de elementos.
- Vamos apresentar alguns exemplos de estruturas, mas nesta disciplina, em especial, o foco inicial está nos vetores.

Como representar um número complexo: $c = x + y\mathbf{i}$?

Podemos criar uma estrutura que contenha dois valores reais representando a parte real e imaginária do número

```
structure complexo {
    double x, y;
} c;
```

Na manipulação precisamos indicar cada elemento da estrutura:

```
c.x = 3;
c.y = 5;
```
▷ representa $c = 3 + 5\mathbf{i}$

A estrutura em si não é um tipo, mas usando o `typedef` podemos definir um tipo com base na estrutura.
Em outras linguagens este recurso pode aparecer com outras definições, como o `record` no pascal/delphi.

Classes

Em linguagem orientadas a objetos, o tipo Classe representa um dado semelhante à estrutura, com o detalhe de que além dos dados internos da estrutura, também associamos funções aos elementos da estrutura.

```
class complexo {
    \\Elementos
    double x, y;
    \\Funções
    complexo modulo(complexo c) {...}
    \\Operações
    complexo operator=(complexo c) {...}
    complexo operator+(complexo c) {...}
}
```

A implementação pode variar de linguagem para linguagem.

```
complexo a, b, c;
a.x = 3;
a.y = 5;
b = a;
c = a + b;
```

Nem todas linguagens criam a sobreposições de operadores, ficando estes destinados apenas a tipos primitivos.

Unions

- Unions representam um tipo especial de estrutura que agrupa em um mesmo espaço de memória dois tipos diferentes de estruturas.
- Cabe ao programador interpretar como serão acessados os dados nesta estrutura.
- pode ser importante caso uma única representação interna pode ser interpretada de duas ou mais formas distintas.

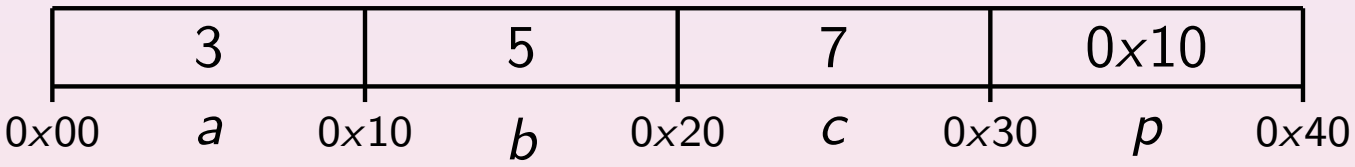
Ponteiros

- Ponteiro é um tipo especial primitivo que armazena um endereço de memória.
- Normalmente quando fazemos uma operação com ponteiros, nós manipulamos a memória cujo endereço está armazenado no ponteiro.
- Dizemos que ponteiros representam uma atribuição indireta.
- A linguagem C/C++ é extremamente poderosa pois permite manipular diretamente os ponteiros.
- A linguagem Java, como contrapartida, por ser Orientada a Objetos e todos os tipos serem praticamente abstratos (ainda existem os primitivos), toda e qualquer representação de tipo (exceto os primitivos) é feita indiretamente através de ponteiros. De forma implícita, sem que o programador possa interagir com os ponteiros.

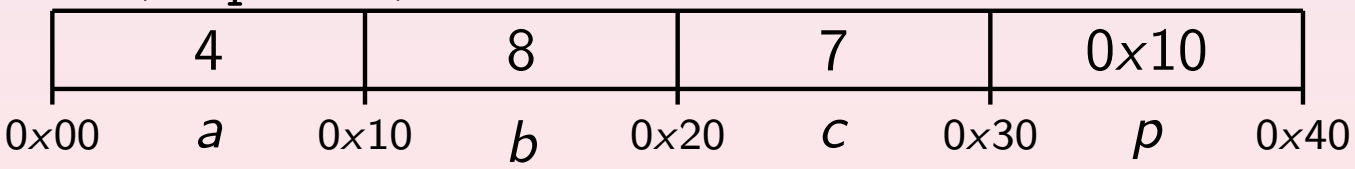
Ponteiros e memória

- Vamos considerar a seguinte representação de inteiros na memória:

```
int a, b, c;
int *p;
a = 3; b = 5; c = 7;
p = &b;
```




```
a = 4; *p = 8;
```



Ponteiros e Memória

- Quando usamos ponteiros, precisamos tratar de "endereços" e "conteúdos":
- `p = &b;` → representa que é atribuído a `p`, o "endereço" (`&`) da variável `b`.
- `*p = 8;` → o conteúdo (`*`) apontado (por isto ponteiro) por `p`, recebe o valor 8.
- Como o ponteiro trabalha com representação indireta, o nível de indireção pode ser maior que 1:

```
int a, *b, **p;
a = 3; b = &a;
p = &b;
*b = 5;
**p = 7;
```

Ponteiros e Estruturas

```
class complexo {
public:
 double real, imag;
};

int main() {
 complexo c, *p;
 c.real = 10; c.imag = 20;
 p = &c;
 (*p).real = 15;
 p->imag = 25;
 return 0;
}
```

Ponteiros e Estruturas

- Trabalhando com estruturas, o acesso aos elementos se dá pelo operador "."
- A estrutura c possui os elementos `real` e `imag`. Logo o acesso se dá como: `c.real` e `c.imag`
- Como `p` é um ponteiro para a estrutura `c`, ao usarmos `*p`, estamos fazendo referência a `c`, logo `(*p).real` e `(*p).imag` acessam os elementos de `c`.
- Para facilitar o conjunto "`(*).`" pode ser substituido por "`->`". Assim: `p->real` e `p->imag`.
- Nos próximos temas trabalharemos com tipos lineares (vetores), mas estruturas de elementos e ponteiros serão retomados com temas envolvendo tipos não lineares (árvores, por exemplo).

Vetor é uma estrutura de dados básica:

- Representa uma sequência de elementos de forma linear, mapeada diretamente na memória, estática.
- Possui operações e manipulações implementadas em hardware e software, inclusive otimizações.
- Todos elementos são do mesmo tipo.
- Se os elementos do vetor forem do tipo primitivo, o conteúdo do vetor é o valor do elemento.
- Se os elementos forem de um tipo não primitivo, o conteúdo pode ser um endereço da memória onde o elemento se encontra.
- Os elementos são indexados por números inteiros, normalmente a partir do 0.

Vetores e ponteiros

- Quando declaramos um vetor, indicamos uma variável que o representa e a dimensão do vetor (quantos elementos).
- O tamanho de cada elemento depende do tipo representado pelo vetor.
- A variável que representa o vetor na realidade guarda o endereço aonde está alocado o primeiro elemento.
- Os elementos seguintes seguem uma ordem sequencial de memória por conta do tamanho do tipo do elemento.
- De fato, a variável representada pelo vetor é um ponteiro para o primeiro elemento:
  `x[0]` $\equiv$ `*x`

Vetores e ponteiros

- Podemos trabalhar no vetor como uma indexação da variável:
  `int x[5];`
  `x[0] = 1; x[1] = 2; x[2] = 3; x[3] = 4; x[4] = 5;`
- Como `x[0]` $\equiv$ `*x`, e o próximo elemento está na sequência, então `x[1]` $\equiv$ `*(x+1)`
- Como o tamanho do elemento é conhecido pelo compilador, ao somar 1, estamos na realidade somando o tamanho de um elemento ao endereço apontado por `x`, logo `*(x+1)` é o conteúdo do próximo elemento a partir do endereço `x`
- Por curiosidade `x[k]` é na realidade a representação do conteúdo `*(x+k)`. Mas a operação soma não importa a ordem das parcelas, poderia ser o mesmo resultado de `*(k+x)` $\equiv$ `k[x]`
  `x[2]` $\equiv$ `2[x]`

Vetores e ponteiros

- Como a variável é um ponteiro em si, podemos trabalhar então com ponteiros para acessar os elementos do vetor

```
int x[5];
int *px;
int i;
x[0] = 1; x[1] = 2; x[2] = 3; x[3] = 4; x[4] = 5;
px = x;
px+ = 2;
*px = x[4];
1[x] = 7;
```

- Como fica o vetor no final desta operação?
- Podemos fazer `px[2] = 2`?

Vetor de caracteres ou *strings*

- Se o tipo de dado no vetor é um caracter, teremos uma sequência de letras que podem representar uma frase ou uma sentença.
- Como o espaço do vetor é maior que algumas palavras ou sentença, teremos caracteres que não são significativos no final do vetor, para não referenciá-los costuma-se colocar um terminador.
- Na linguagem C e outras, o terminador é o caracter '\0' ou simplesmente (int) 0. No pascal a estrutura é mais complexa.
- É oferecido em software uma implementação de bibliotecas especiais para tratar *strings*
- No C++ existe um tipo nato `string` para manipulação de strings.

Matrizes e além

Como o vetor pode ser uma coleção de tipos abstratos, os elementos de um vetor pode ser, também, um vetor.
Neste caso falamos de Matrizes. Para acessar um elemento precisamos indexar em qual vetor ele se encontra, no vetor de vetores, e dentro deste vetor qual sua posição, de fato.
Este conceito pode se estender a mais dimensões.

```
int m[3][3];
double x[2][2][2][2][2][2];
m[0][2] = 3;
x[0][0][1][1][0][1] = 2;
```

Vetores - (Geometria Analítica) Definição

- Vamos definir um vetor bidimensional como um elemento que liga um ponto de origem a um ponto de destino no plano cartesiano bidimensional $\overrightarrow{(a,b)(c,d)}$.
- Estamos aqui fazendo uma interpretação bem restrita do espaço vetorial em geometria analítica.
- O vetor passa a ser representado, então, por um comprimento e direção (inclinação e sentido).
- Um vetor passa a ser bem definido por um único ponto de destino, considerando a origem o ponto (0,0) $\rightarrow \overrightarrow{(a,b)}$.
- Para tanto, o vetor pode ser representado na computação por dois valores de ponto flutuante: a e b, indicando o ponto de destino.

Vetores - Manipulação

- Para a manipulação dos vetores, devemos ser capazes de:
  - Criar um vetor a partir de um ponto (origem é $(0, 0)$).
  - Criar um vetor a partir de dois pontos $\overrightarrow{(a, b)(c, d)}$
  - Obter a abscissa de um vetor (a projeção horizontal de um vetor com origem em $(0, 0)$)
  - Obter a ordenada de um vetor (a projeção vertical de um vetor com origem em $(0, 0)$)
  - Obter o módulo de um vetor (seu comprimento)
  - Obter a direção de um vetor (um ângulo de direção em relação ao eixo horizontal direito)

Vetores - Operações

- Operações definidas para vetores no plano bidimensional:
  - Atribuição $\vec{v} = \vec{w}$:
    $\vec{v}$ é uma cópia de $\vec{w}$.
  - Simétrica $\vec{v} = -\vec{w}$:
    $\vec{w} = \overrightarrow{(a, b)} \rightarrow \vec{v} = \overrightarrow{(-a, -b)}$
  - Soma $\vec{v} = \vec{w} + \vec{x}$:
    Sendo $\vec{w} = \overrightarrow{(a, b)}$ e $\vec{x} = \overrightarrow{(c, d)}$ então deslocamos o vetor $\vec{x}$ para origem em (a, b), fica assim: $\vec{x} = \overrightarrow{(a, b)(a + c, b + d)}$ o vetor resultante tem origem em $(0, 0)$ e destino no mesmo destino do vetor $\vec{x}$ deslocado, ou seja, $\vec{v} = \overrightarrow{(a + c, b + d)}$
  - Subtração $\vec{v} = \vec{w} - \vec{x}$:
    É o mesmo que a soma de $\vec{w}$ com o simétrico de $\vec{x}$
  - Produto escalar $p = \vec{v} \cdot \vec{w}$:
    Sendo $\vec{v} = \overrightarrow{(a, b)}$ e $\vec{w} = \overrightarrow{(c, d)}$, define-se $p = (a \times c) + (b \times d)$. Isto é importante por exemplo em física, quando calculamos o trabalho $\mathcal{T} = \vec{F} \cdot \vec{s}$.

Aplicação em C: vetores

- Vamos aqui construir toda uma solução para implementar vetores na linguagem.
- Vamos precisar criar a estrutura que representa o vetor.
- Também vamos precisar criar todas as operações que desejamos fazer com vetores.
- Na realidade estamos construindo uma biblioteca científica para manipulação de vetores na linguagem.
- Para esta implementação, vamos definir um arquivo de cabeçalho "`.h`" que conterá todas as definições.
- Também faremos um arquivo de implementação em "`.c`" com as operações.

Vetores - Estrutura de dados

- Primeiro a definição da estrutura, isto irá no arquivo "`vetor.h`"

```
typedef struct vetor_s {
  double a, b; // Vetor com origem em (0,0) e destino em (a,b)
} vetor;
```

- Observe que ao declarar valores em ponto flutuante, não há perda de performance em fazê-lo como `double`, o uso de `float` era importante quanto havia severas restrições de memória, este tipo existe para manter compatibilidade, tal como `short`.

Vetores - protótipos

```
void criar(vetor *v, double a, double b);
void criard(vetor *v, double a, double b, double c, double d);
double abscissa(vetor v);
double ordenada(vetor v);
double modulo(vetor v);
double direcao(vetor v);

void atribui(vetor *v, vetor w);
void simetrica(vetor *v, vetor w);
void soma(vetor *v, vetor w, vetor x);
void subtrai(vetor *v, vetor w, vetor x);
double escalar(vetor v, vetor w);
```

Também no "`.h`"

Vetores - implementação

- Vamos tomar como exemplo a `soma`: v = w + x.
- Da forma que fizemos, a soma se dá como: `soma(&v,w,x)`.
- Ou seja, como precisamos alterar o valor de a, precisamos passar seu endereço. Os demais só precisamos do valor logo passamos uma cópia e não o endereço.

```
void soma(vetor *v, vetor w, vetor x)
{
 v->a = w.a + x.a;
 v->b = w.b + x.b;
}
```

- as demais implementações seguem nos arquivos de códigos desta aula.

Vetores: Classe no C++

- Semelhante acontece na implementação em C++
- A declaração e protótipos acontecem no arquivo `vetor.hpp`
- A implementação no arquivo `vetor.cpp`
- Porém diferente da linguagem C, nós podemos sobrepor os operadores aritméticos. Ou seja, ao invés de criar o método "`soma`", vamos sobrepor o operador "+".

Vetores: Classe no C++

```
class vetor {
  private:
    double a, b;

  public:
    vetor();
    vetor(double a, double b);
    vetor(double a, double b, double c, double d);

    double abscissa();
    double ordenada();
    double modulo();
    double direcao();

    vetor operator=(vetor v); // atribuição
    vetor operator-(); // simétrica
    vetor operator+(vetor v); // soma
    vetor operator-(vetor v); // subtração
    double escalar(vetor v);
};
```

1. Resolver a 1ª Lista de Exercícios.