Algoritmos e Estruturas de Dados I

Prof. Lúcio Mauro Pereira

26-27/09/2024

Aula anterior

Ao final, tudo se resume em *bits.*

Bit: Contração de *Binary Digit*

Assume um entre dois estados, tais como:

Ligado ou desligado

O ou 1

Alto ou baixo

Aula anterior

O número de *bits* combinados determina a quantidade de representações.

Por exemplo, uma combinação de 8 *bits* permite 256 representações.

00000001

00000010

00000011

00000100

00000101

...

11111111

$2^8 = 256$

Byte: Contração de *Binary Term*

Aula anterior

RAM: *Random Access Memory*

Uma coleção de *bytes* disponibilizados para armazenar dados e programas

Cada *byte* possui um endereço único que o identifica

A posição da memória é tipicamente expressa em hexadecimal

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14

Aula anterior

O Sistema Operacional (Windows, Linux *etc.*) se encarrega do controle

Uma tabela permite gerenciar os *bytes* livres e ocupados da memória

Uma variável é apenas uma forma de nomear uma posição da memória

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Os bastidores da declaração de uma variável

int x;

A linguagem C irá definir o número de *bytes* requerido para representar um inteiro

Em sua máquina, serão reservados 4 *bytes.*

O Sistema Operacional irá identificar 4 *bytes* contíguos que estejam livres

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Os bastidores da declaração de uma variável

int x;

Por exemplo, na imagem abaixo, poderiam ser: 1E, 1F, 20, 21

Em uma tabela interna, o SO irá associar a variável **x** ao endereço **1E**

Por serem contíguos, basta o endereço do **primeiro byte**. O tipo determinará a quantidade

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Tente:

```
int main() {
    int x;
    printf("\nA variável x foi declarada na posição %p", &x);
    return 0;
}
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

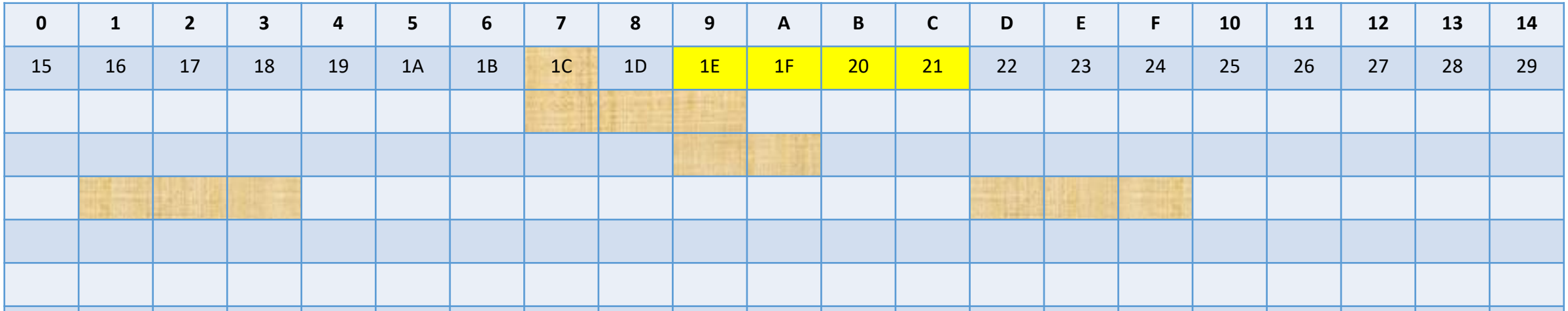
Tente:

```
int main() {
    int x;
    printf("\nA variável x foi declarada na posição %p", &x);
    printf("\nOcupando %i bytes", sizeof(int));
    return 0;
}
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Tente:

```
int main() {
    int x;
    printf("\nA variável x foi declarada na posição %p", &x);
    printf("\nOcupando %i bytes", sizeof(int));
    x = 7;
    printf("\nE guarda o valor %i", x);
    return 0;
}
```



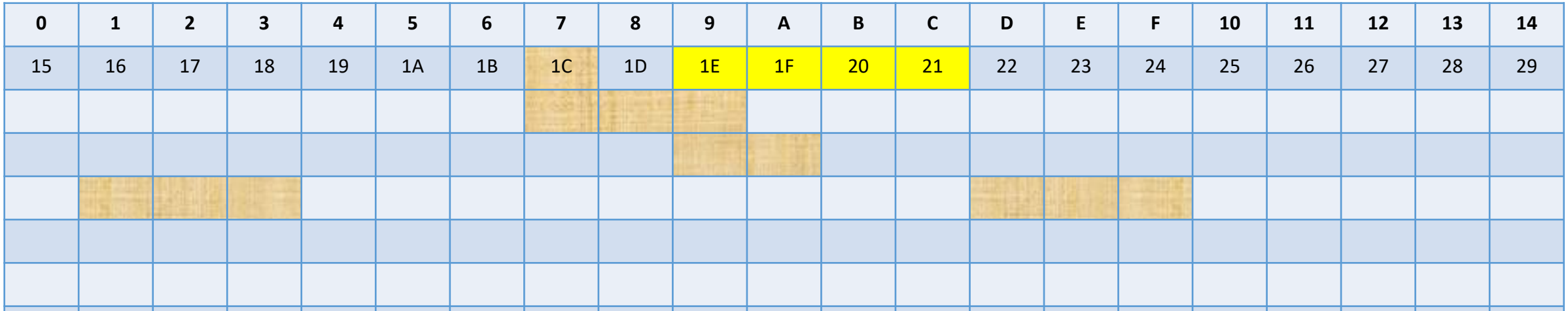
Alocando memória de forma dinâmica

```
int main() {
    int *x;
    printf("\nO ponteiro x foi declarado na posição %p", &x);
    x = malloc( 4 );
    printf("\nOcupando %i bytes", sizeof(int));
    printf("\nE aponta para %p posição", x);
    return 0;
}
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Alocando memória de forma dinâmica

```
int main() {
      int *x;
      printf("\nO ponteiro x foi declarado na posição %p", &x);
      x = malloc( sizeof(int) );
       printf("\nOcupando %i bytes", sizeof(int));
      printf("\nE aponta para %p posição", x);
      return 0;
}
```



Alocando memória de forma dinâmica

```
int main() {
      int *x;
      printf(“\nO ponteiro x foi declarado na posição %p”, &x);
      x = malloc( sizeof(int) );
       printf(“\nOcupando %i bytes”, sizeof(int));
      printf(“\nE aponta para %p posição”, x);

      *x = 7;
      printf(“\nE guarda o valor %i”, *x);
      return 0;
}
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29

Experimente

```
int main() {
	int A = 7;
	int *p;

	p = &A;

	// Escreva a posição de A
	// Escreva a posição do ponteiro p
	// Escreva para onde aponta o ponteiro p
	// Escreva o valor de A
	// Escreva o valor guardado no local para onde aponta p

	return 0;
}
```