Aula 07:
- Mapa de memória de um processo
- Ponteiros (parte 1)

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

3Q-2017

Mapa de memória de um processo

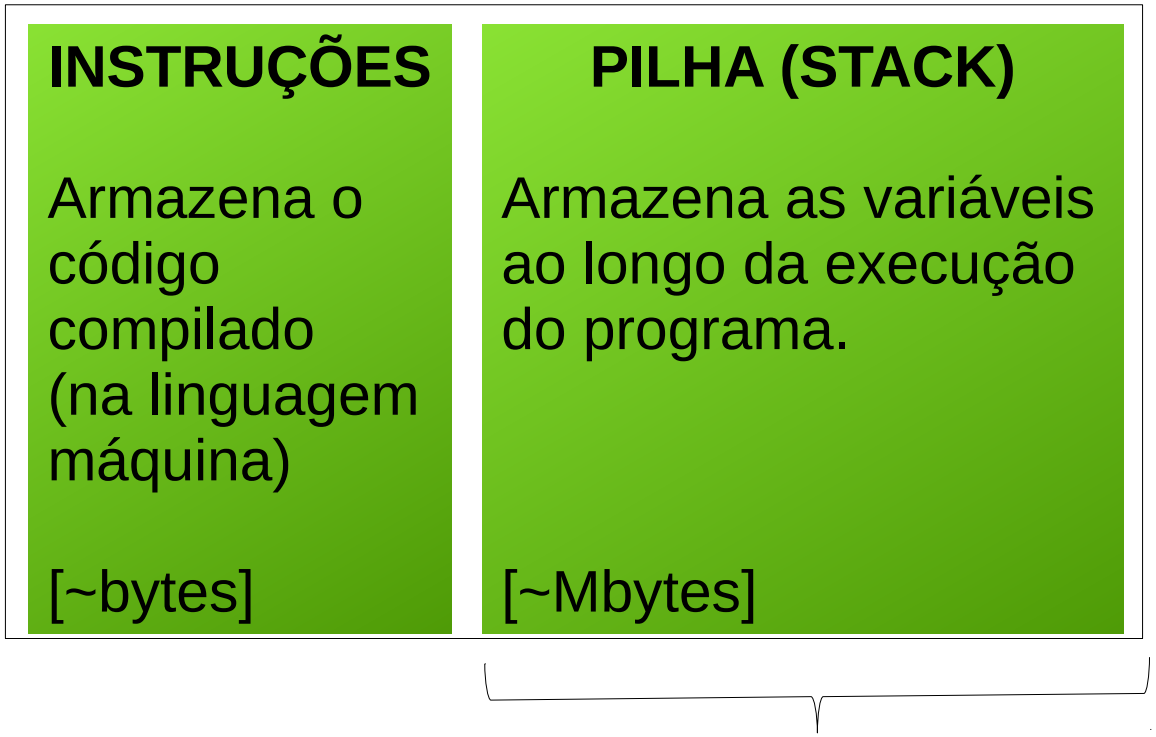
Alocação de memória: estática VS Dinâmica

Na execução, **um programa é um processo**.

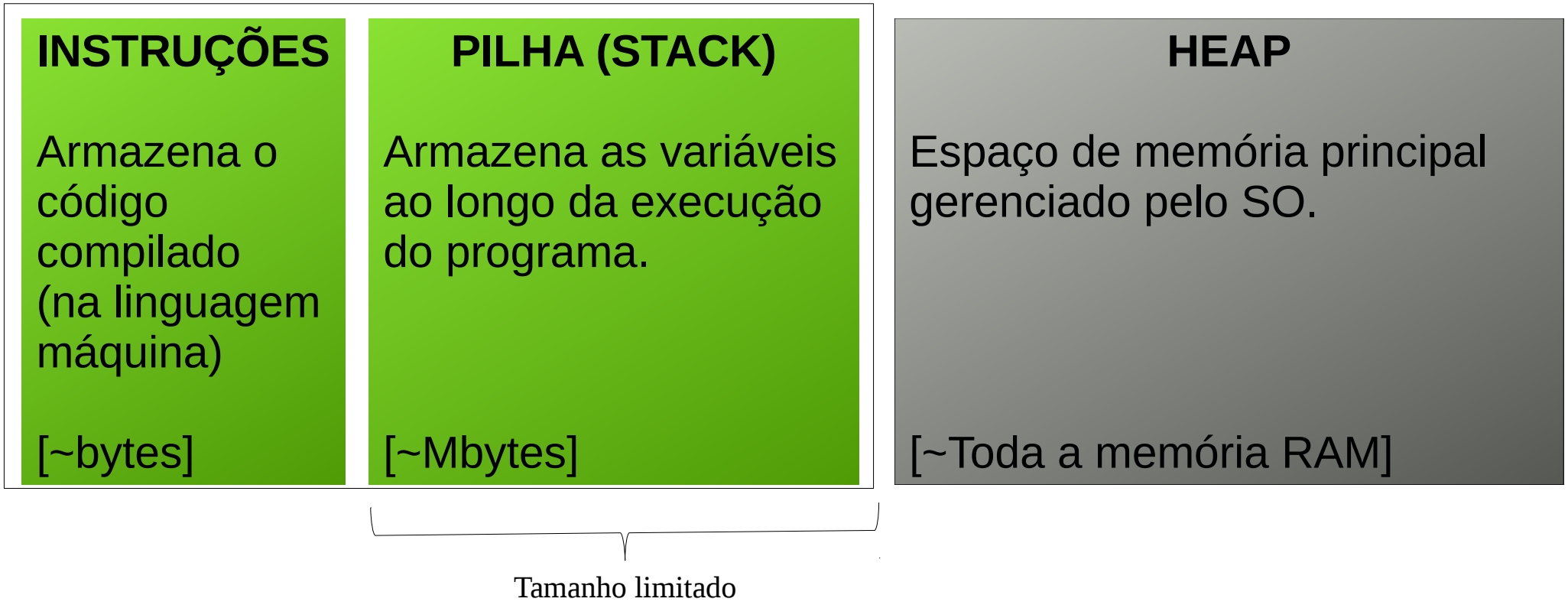
Um processo ocupa parte da memória principal, reservada para:

- As instruções, e
- A pilha

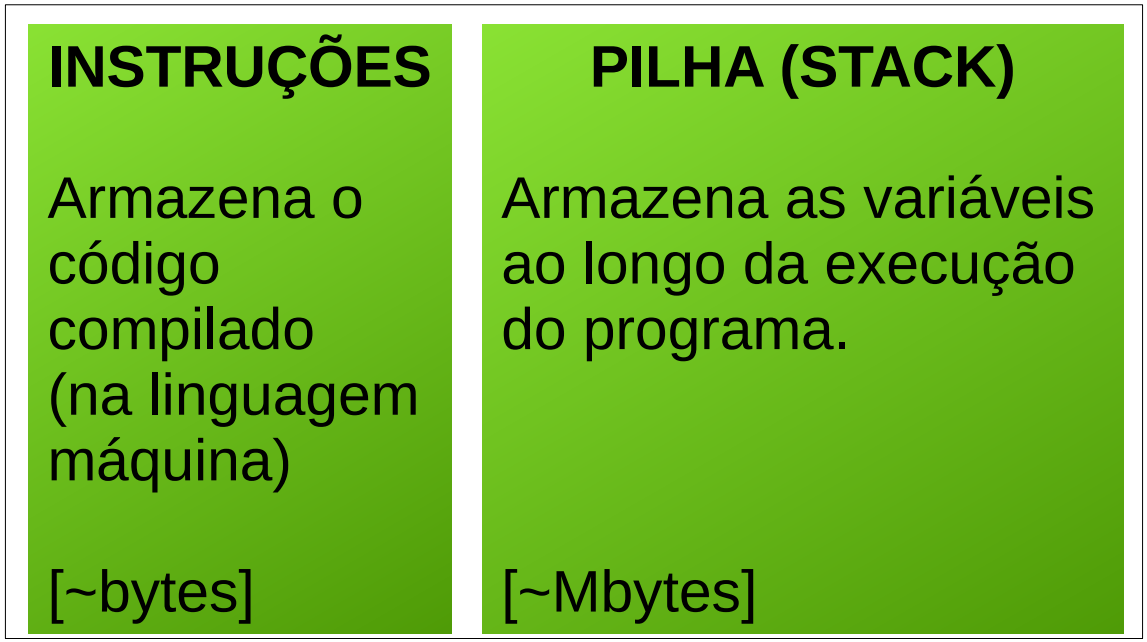
Processo na memória

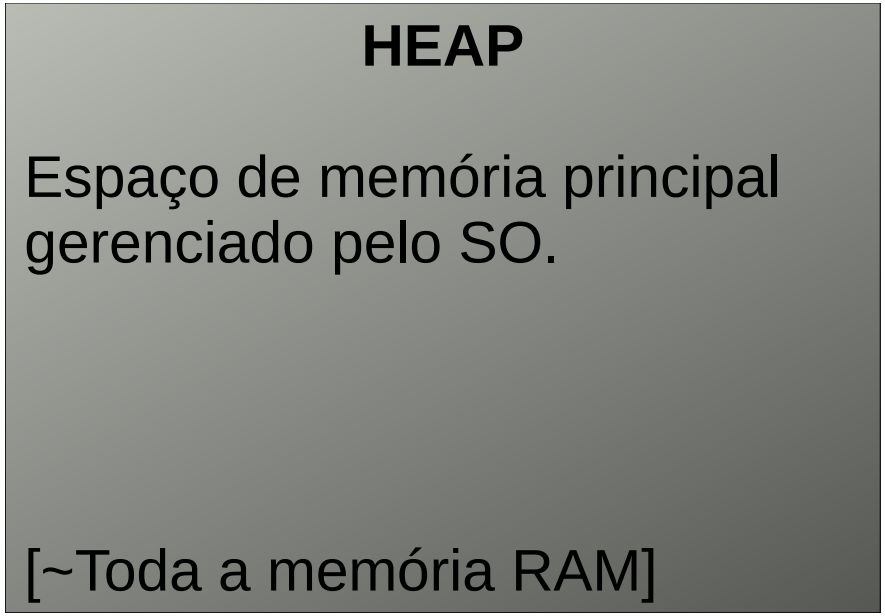


Processo na memória



Processo na memória





Alocação estática

Alocação dinâmica

```
int x;
double M[10][20];
char *c;
```

```
double M = malloc(...);
```

NAME

malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

The **malloc**() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by ptr, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **realloc**() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**() or **realloc**(). If the area pointed to was moved, a free(ptr) is done.

RETURN VALUE

The **malloc**() and **calloc**() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a size of zero, or by a successful call to **calloc**() with nmemb or size equal to zero.

The **free**() function returns no value.

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

Maior elemento em um vetor

```c
#include <stdio.h>

void main() {
    int i, n;

    scanf("%d", &n);

    long int vetor[n];

    for (i=0; i<n; i++)
        scanf("%ld", &vetor[i]);

    long int maior = vetor[0];

    for (i=1; i<n; i++)
        if (maior < vetor[i])
            maior = vetor[i];

    printf("O vetor de %d elementos contem como maior elemento: %ld\n", n, maior);
}
```

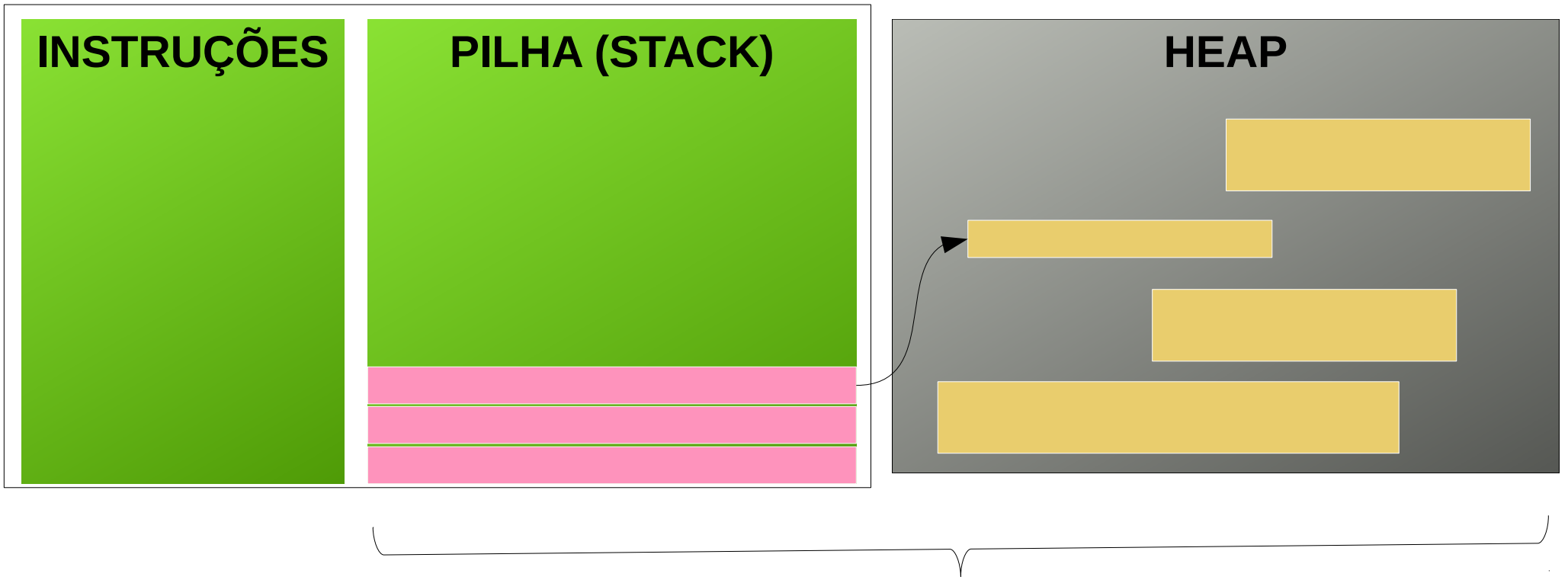
← *Alocação estática*

← *Alocação **dinâmica***

← *Alocação estática*

Ponteiros?

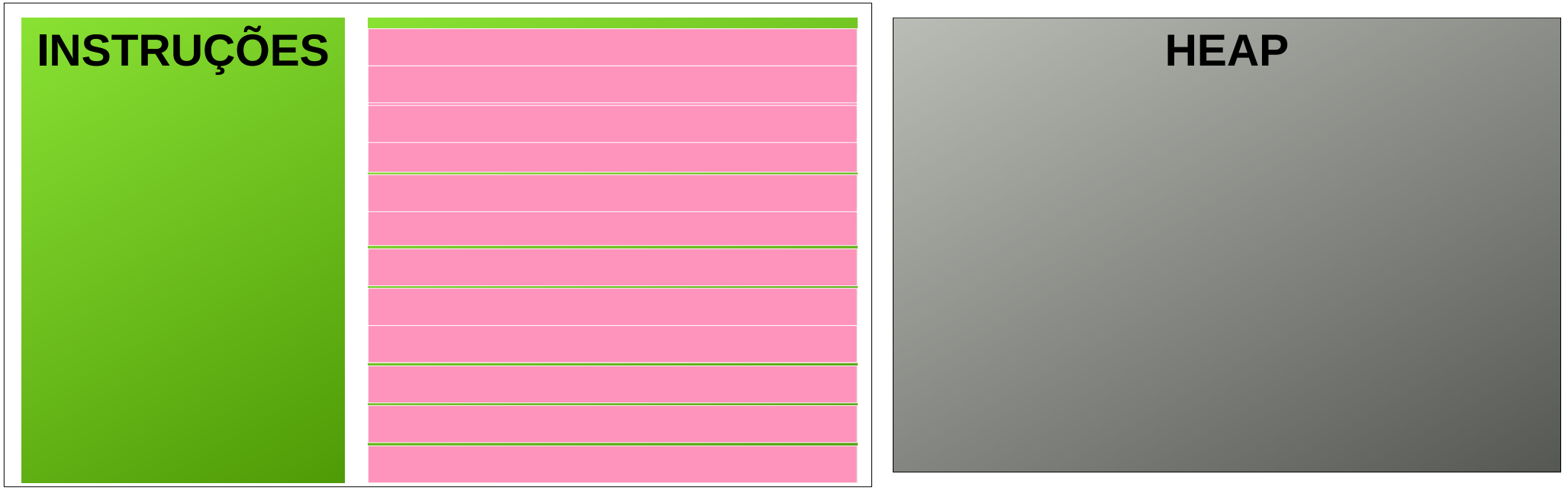
Processo na memória



Em Java e Python o uso é transparente.
Não precisa se preocupar de alocar e liberar memória

Fenômeno: Stack Overflow

Processo na memória



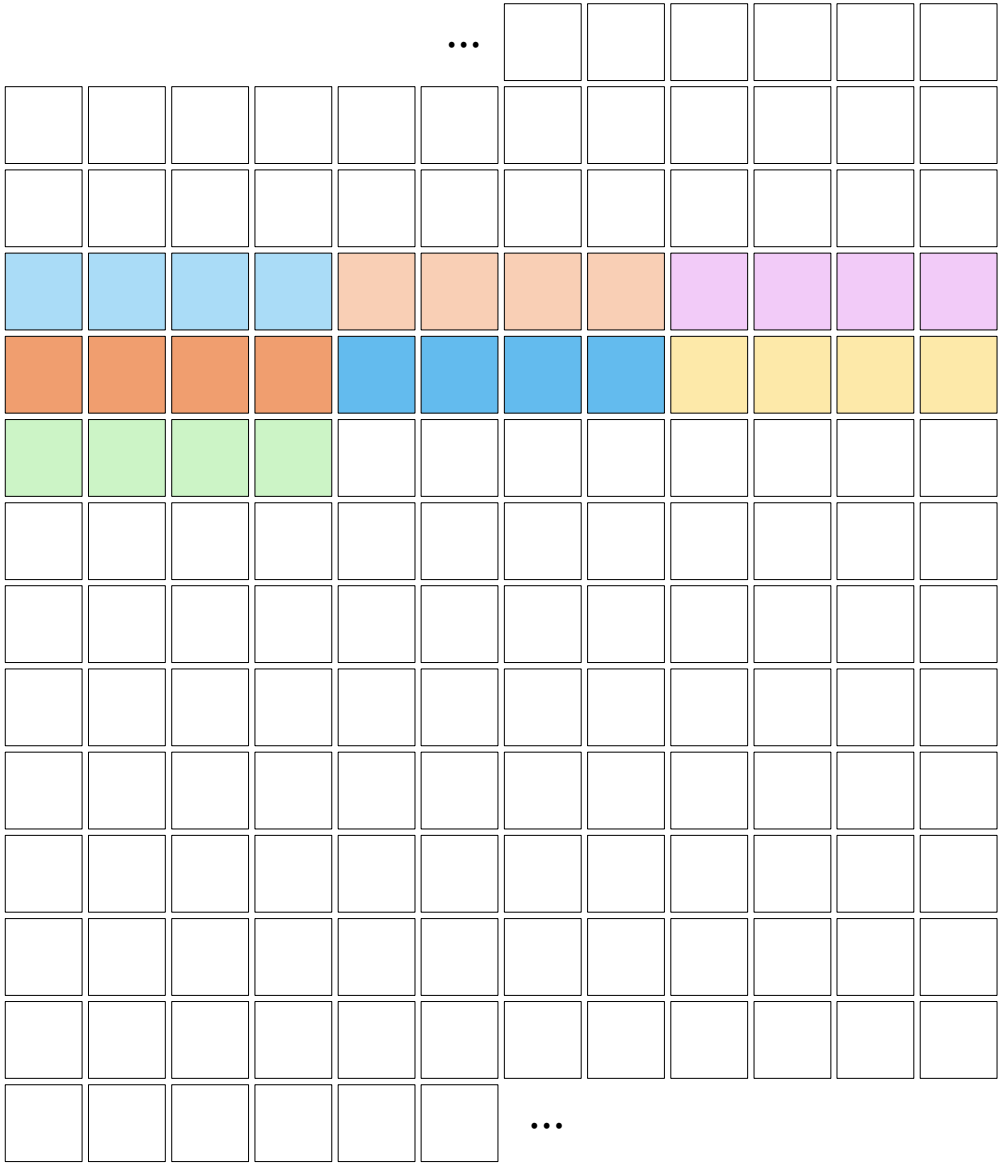
Ponteiros

Vetores e ponteiros

```
int vetor[7] = {1,2,3,4,3,2,1}
```

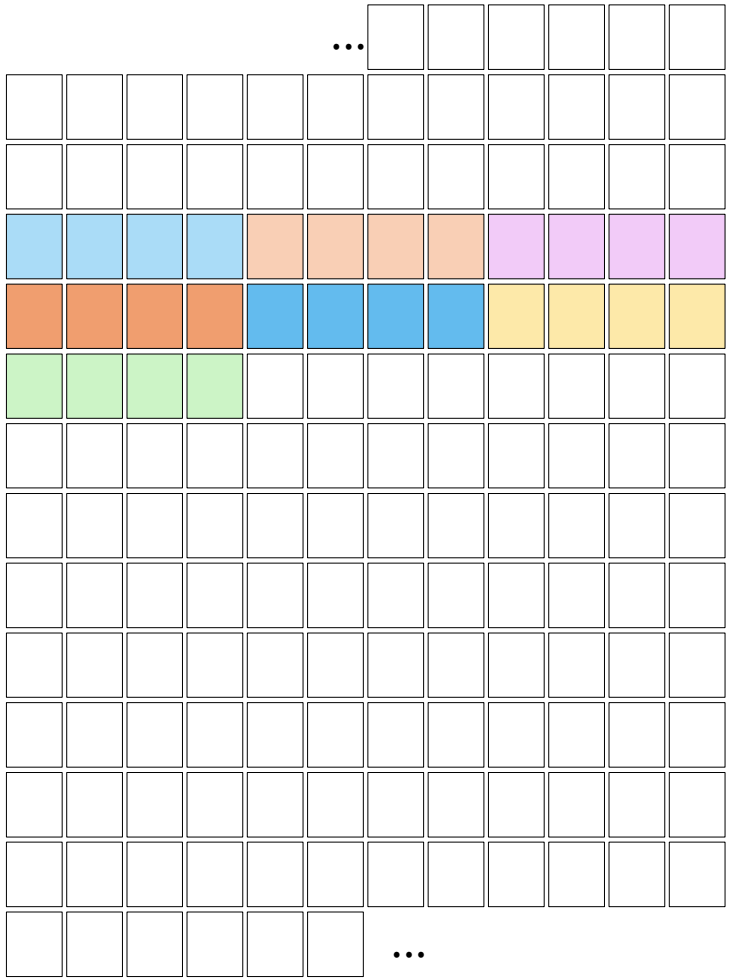
vetor

```
0 : 1
1 : 2
2 : 3
3 : 4
4 : 3
5 : 2
6 : 1
7 : 0
8 : 0
9 : 0
10 : -1190166843
11 : 32596
12 : 0
13 : 0
14 : -812538680
15 : 32764
16 : 0


Process exited with code: 0
```



Vetores e ponteiros

```c
#include <stdio.h>

int main()
{
    int vetor[7] = {1,2,3,4,3,2,1};
    int i;

    printf("%p : %d\n", vetor, *vetor);
    printf("%p : %d\n", vetor+0, *(vetor+0));
    printf("%p : %d\n", vetor+1, *(vetor+1));
    printf("%p : %d\n", vetor+2, *(vetor+2));
    printf("%p : %d\n", vetor+3, *(vetor+3));
    printf("%p : %d\n", vetor+4, *(vetor+4));
}
```

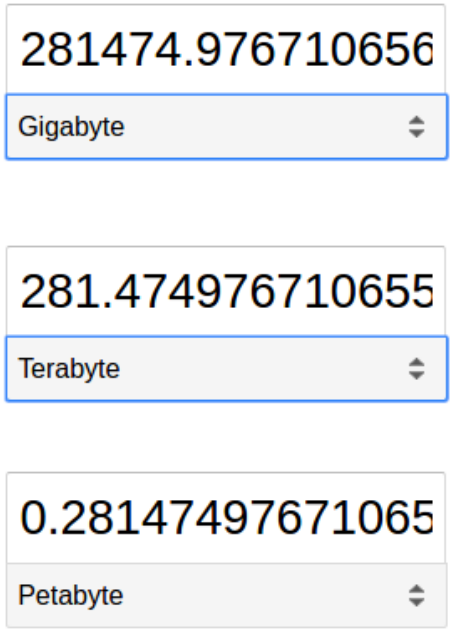


```
0x7ffd627c49b0 : 1
0x7ffd627c49b0 : 1
0x7ffd627c49b4 : 2
0x7ffd627c49b8 : 3
0x7ffd627c49bc : 4
0x7ffd627c49c0 : 3
```

Endereços de memória

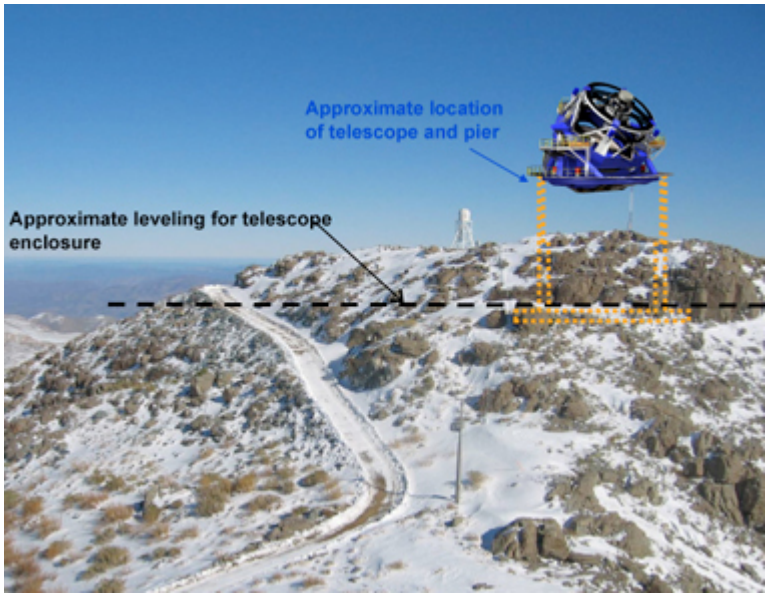
```
0x7ffd627c49b0 : 1
0x7ffd627c49b0 : 1
0x7ffd627c49b4 : 2
0x7ffd627c49b8 : 3
0x7ffd627c49bc : 4
0x7ffd627c49c0 : 3
```

16^{12}= 281 474 976 710 656



4 bits para representar cada número na base 16.
→ 12*4=48 bits → 2^48 (números diferentes)

Large Synoptic Survey Telescope, LSST



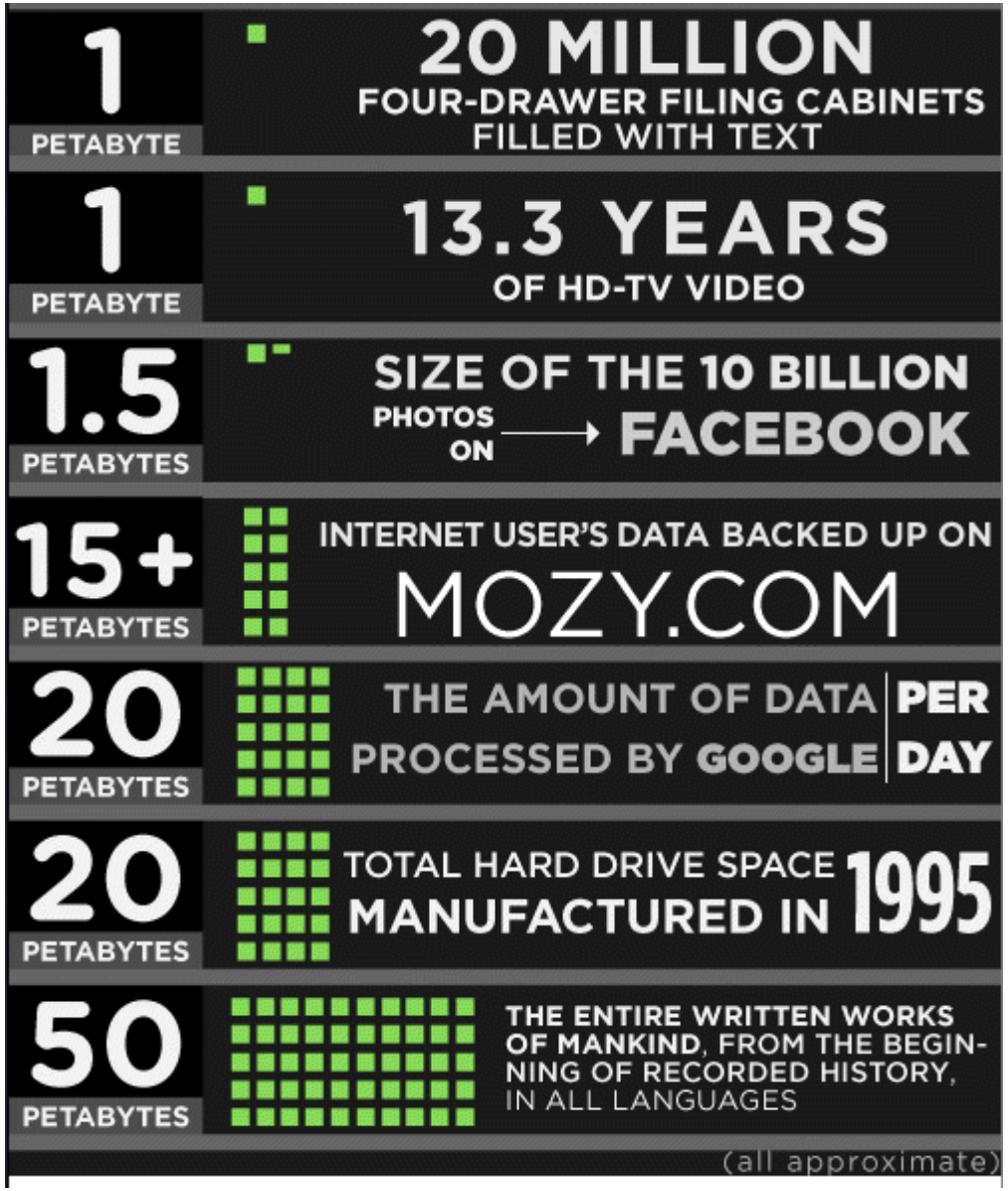
Telescopio Grande para Rastreos Sinópticos
(Large Synoptic Survey Telescope, LSST):
– 8,4 metros
– capaz de examinar a totalidade do ceu visible
– Norte do Chile (2016)
– Camera de 3200 megapixels (~3 Gigapixels)

Planejado para armazenar mais de 30 Terabytes de dados de imagens por noite, mantendo em ~10 anos um banco de dados de 15 Petabytes

http://www.lsst.org/lsst/

Grande escala?

1000	kB	kilobyte
1000^2	MB	megabyte
1000^3	GB	gigabyte
1000^4	**TB**	**terabyte**
1000^5	**PB**	**petabyte**
1000^6	**EB**	**exabyte**
1000^7	**ZB**	**zettabyte**
1000^8	**YB**	**yottabyte**



http://mozy.com/blog/misc/how-much-is-a-petabyte/

Endereços e ponteiros

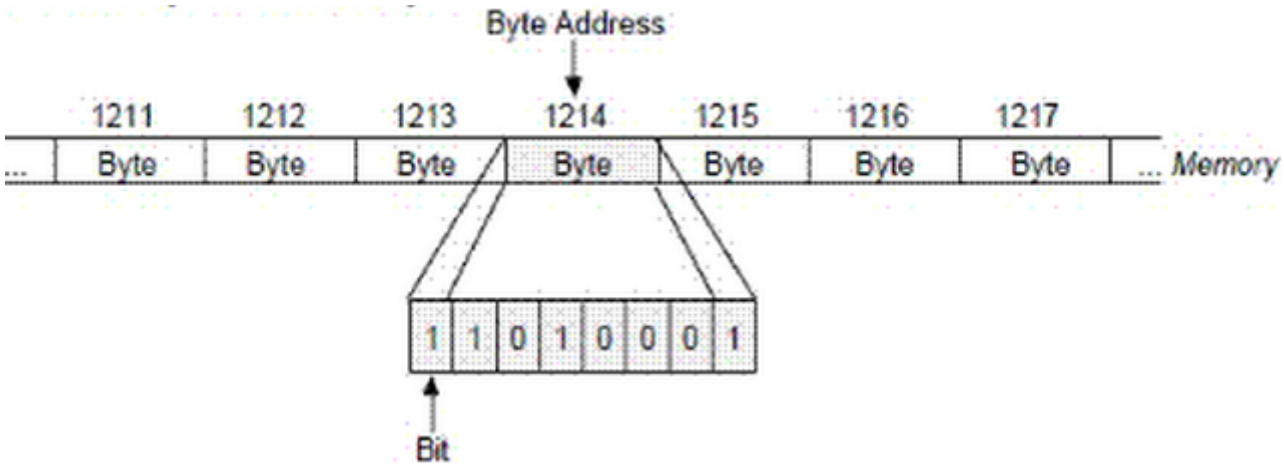
Endereços e ponteiros

- **Os conceitos de endereço e ponteiro são importantes em qualquer linguagem de programação.**
- Na linguagem C é mais **visível** este conceito.
- Requer um **esforço** para usar os ponteiros.

(*) Fonte: P. Feofiloff. Algoritmos em Linguagem C. 1ª Edição, Editora Campos, 2008.

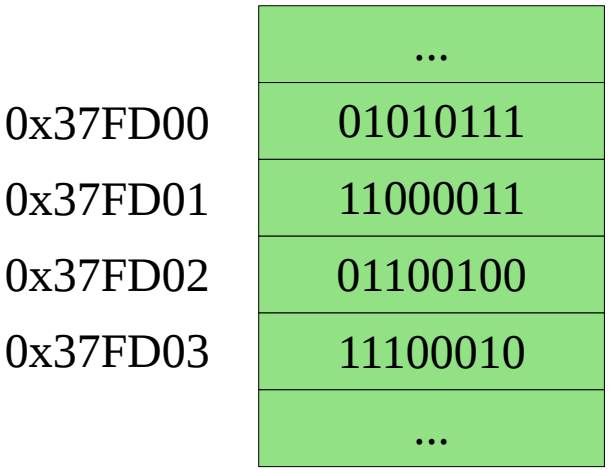
Endereços

- A memória de qualquer computador (arquitetura de Von Neumann) **é uma sequência de bytes**.
- Cada byte armazena um de 256 possíveis valores.
- Os bytes são numerados sequencialmente e o número de um byte é o seu endereço



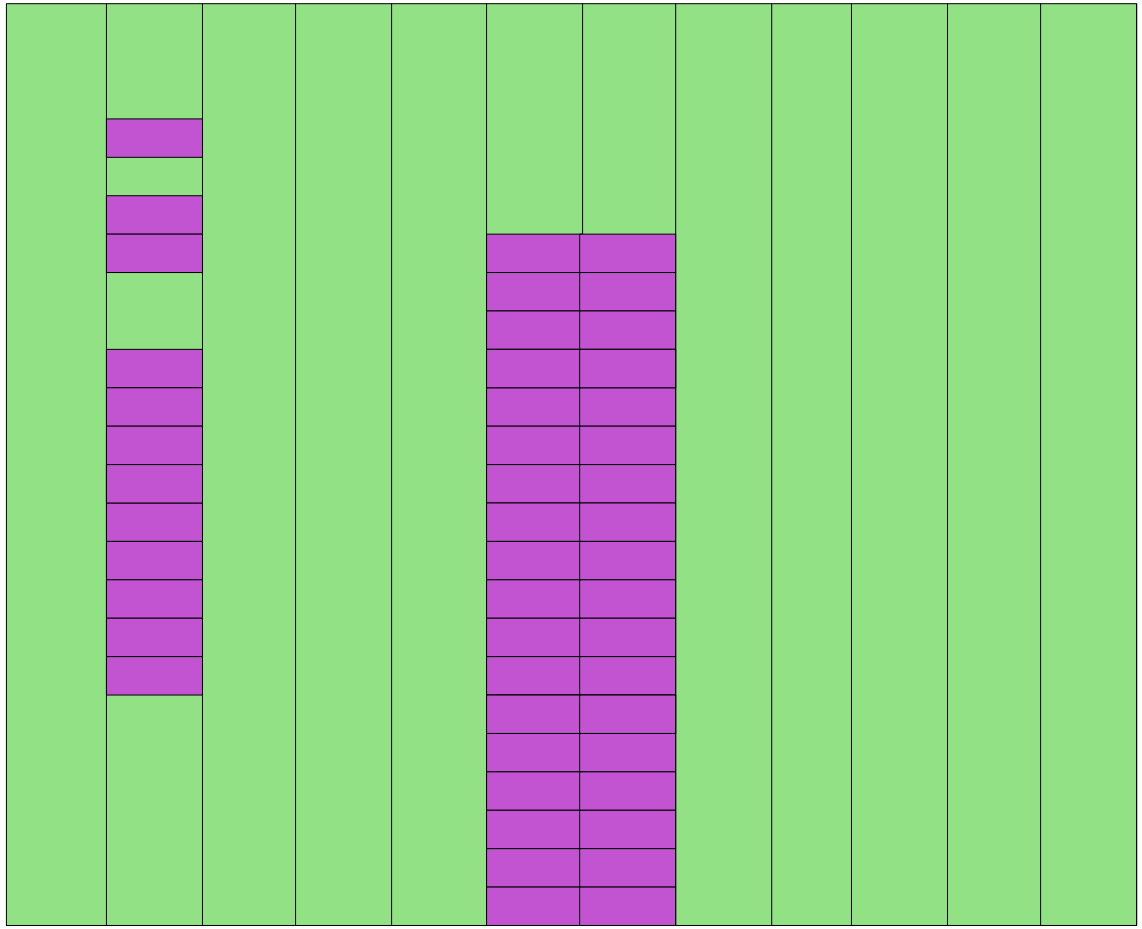
(*) Fonte: http://codingfox.com/3-1-computer-memory-and-data-representation/

Endereços


...
0x37FD00
01010111
0x37FD01
11000011
0x37FD02
01100100
0x37FD03
11100010
...

Endereços

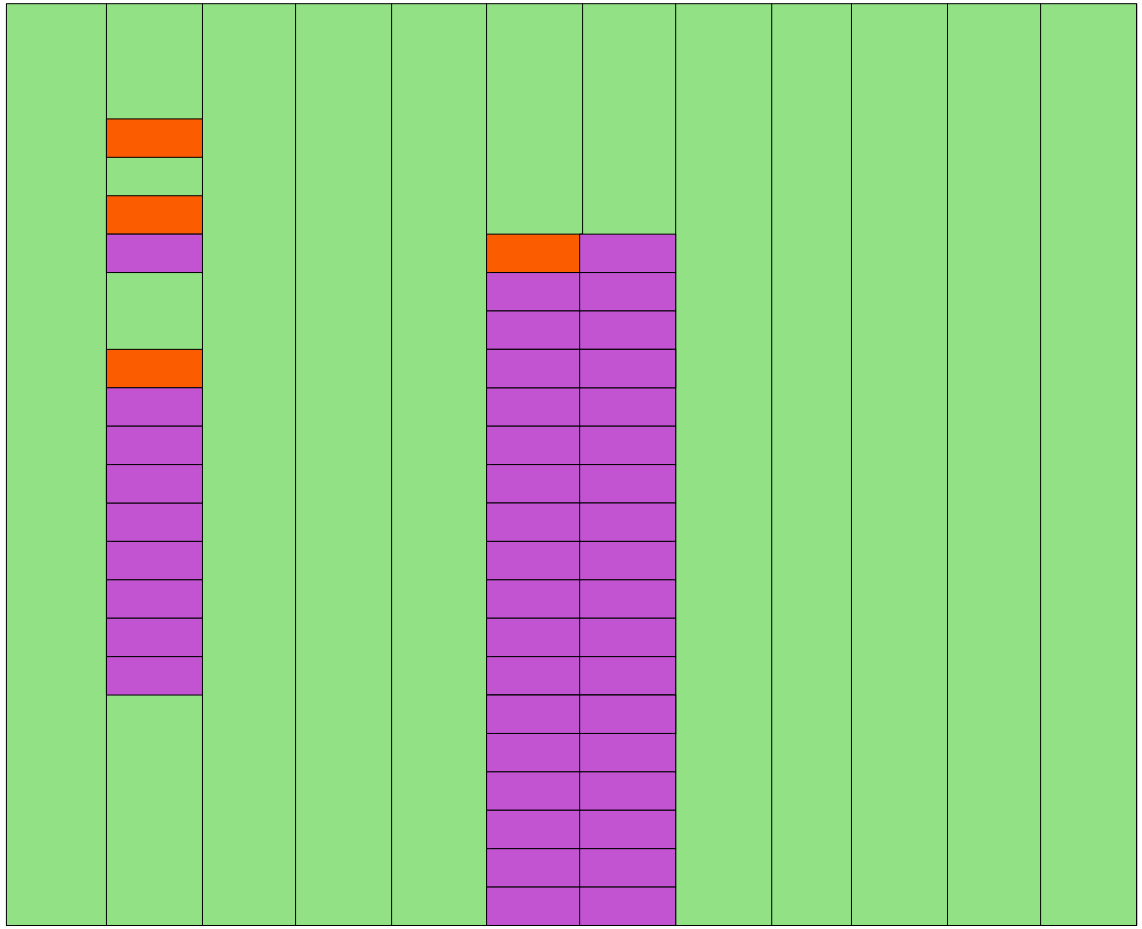
	...
0x37FD00	01010111
0x37FD01	11000011
0x37FD02	01100100
0x37FD03	11100010
	...



Cada objeto na memória do computador tem um endereço.

Endereços

	...
37FD00	01010111
37FD01	11000011
37FD02	01100100
37FD03	11100010
	...



Geralmente o endereço do objeto é o endereço do 1ro byte.

Endereços

```c
#include <stdio.h>

int main() {

    double raio, n=3.14159;
    scanf("%lf", &raio);

    printf("A=%.4lf\n", raio*raio*n);

    return 0;
}

```

Observe o símbolo &

The **scanf**() function reads input from the standard input stream stdin, **fscanf**() reads input from the stream pointer stream, and **sscanf**() reads its input from the character string pointed to by str.

The **scanf**() family of functions scans input according to format as described below. This format may contain conversion specifications; the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments that follow format. Each pointer argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.

Endereços

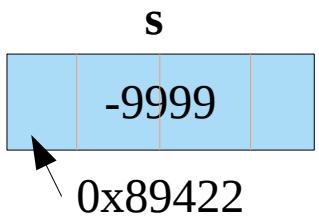
```c
#include <stdio.h>


int main() {
        int x;

        while (scanf("%d", &x) != EOF ) {
          if (0<x && x<10){
             printf("Num: %d\n", x);
          }
        }
        printf("%d", EOF);

}
```

Endereços

- Em **c** o endereço de um objeto é dado pelo operador **&**

- **Se x é uma variável, então &x é o seu endereço**

```
int s = -9999
```

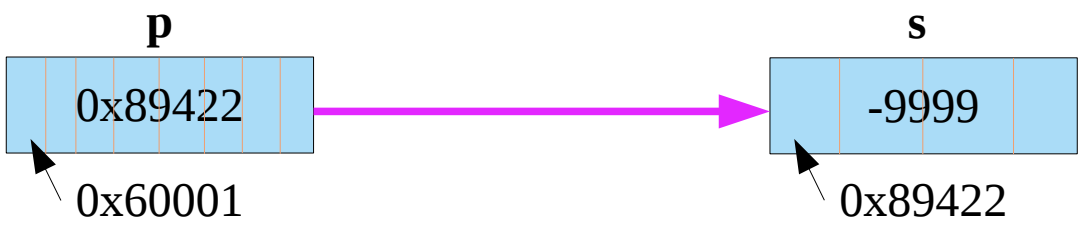


Endereços

- Em **c** o endereço de um objeto é dado pelo operador **&**

- **Se x é uma variável, então &x é o seu endereço**

```
int s = -9999
int *p = &s
```



"p aponta para a s"
"p é o endereço de s"
"p aponta a s"

p = 0x89422
&p = 0x60001

*p = -9999

*p é o mesmo que escrever "s"

Endereços

```c
#include <stdio.h>

void main() {
    int  s = -999;
    int *p = &s;

    printf("%ld\n", sizeof(s));
    printf("%ld\n", sizeof(p));
}
```

4
8

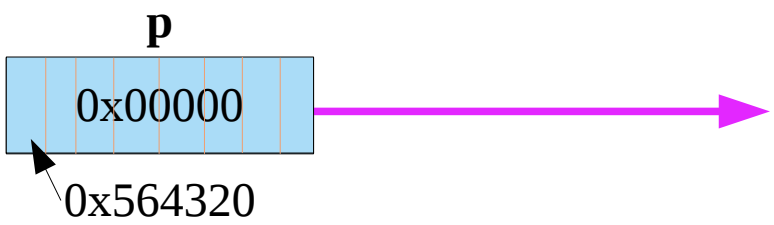
$$x^n = 2^{64}$$

$$= 1.844674407371E + 19$$

Endereços

- Todo ponteiro pode ter o valor NULL.
- NULL é uma constante, geralmente vale 0 (definida no arquivo interface stdlib)

```
int *p = NULL;
```



Endereços

Há vários tipos de ponteiros:

- P. para caracteres
- P. para inteiros
- P. para registros
- P. para ponteiros para inteiros
- P. para função

int* p ; ← Um tipo de dado novo int* (conceitualmente correto)

int *p; ← O "*" modifica a variável e não o int (mais aceito)

int * p;

O compilador C aceita qualquer das formas.

exemploPonteiro.c

```c
#include<stdio.h>

int main() {
    int x;
    int i = 100;

    int *p;          /* p é um ponteiro para um inteiro */
    p = &i;          /* p aponta para i*/

    x = *p+900;      /* o mesmo que x = i+900 */

    printf("O valor de i   : %d\n", i);
    printf("O endereco de i: %p\n", &i);
    printf("O valor de p   : %p\n", p);
    printf("O valor de x   : %d\n", x);

}
```

```
O valor de i   : 100
O endereco de i: 0x7ffd987e5930
O valor de p   : 0x7ffd987e5930
O valor de x   : 1000
```

Operadores unarios

`&` → Referência: na frente de **uma variável**:
Devolve o endereço de memória onde a variável está armazenada

`*` → Derreferência: na frente de **variável ou expressão:**
Devolve o valor ou conteúdo do endereço de memória apontada pela variável ou expressão

exemploPonteiro.c

```
%d %i      Decimal signed integer.
%o         Octal integer.
%x %X      Hex integer.
%u         Unsigned integer.
%c         Character.
%s         String.
%f         double
%e %E      double.
%g %G      double.
%p         pointer.
```

exemploPonteiro2.c

```c
#include<stdio.h>

int main() {
    int i;
    i = 100;

    printf("%d\n", *&i);

}
```

exemploPonteiro3.c

```c
#include<stdio.h>

void troca(int i, int j) {
    int temp;
    temp = i;
    i = j;
    j = temp;
}

int main() {
    int a=1;
    int b=10;

    troca(a,b);

    printf("\ta=%d\n\tb=%d", a, b);
}
```

exemploPonteiro3.c

```c
#include<stdio.h>

void troca(int *i, int *j) {
    int temp;
    temp = *i;
    *i = *j;
    *j = temp;
}

int main() {
    int a=1;
    int b=10;

    troca(&a,&b);

    printf("\ta=%d\n\tb=%d", a, b);
}
```

```
a=10
b=1
```

exemploPonteiro3.c

Por que o código abaixo está errado?

```
void troca(int *i, int *j) {
    int *temp;
    *temp = *i;
    *i = *j;
    *j = *temp;
}
```