

# PONTEIROS PARTE II

**Merge Sort, Ponteiros e const, Operador sizeof, Vetor de ponteiros, Ponteiros para Ponteiros, Ponteiros para funções**

# MERGE SORT

20 12

13 11

7 9

2

1

- Ordenação por mistura.
- Ordena por comparação.
- Aplica uma estratégia do tipo divisão e conquista.

# MERGE SORT

20 12

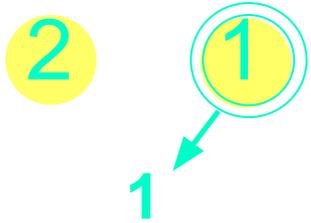
13 11

7 9

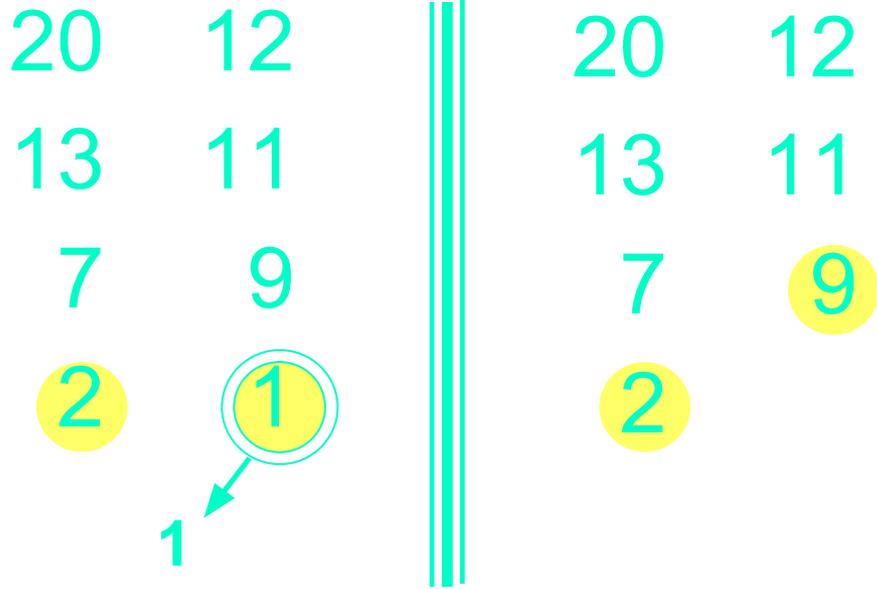
2

1

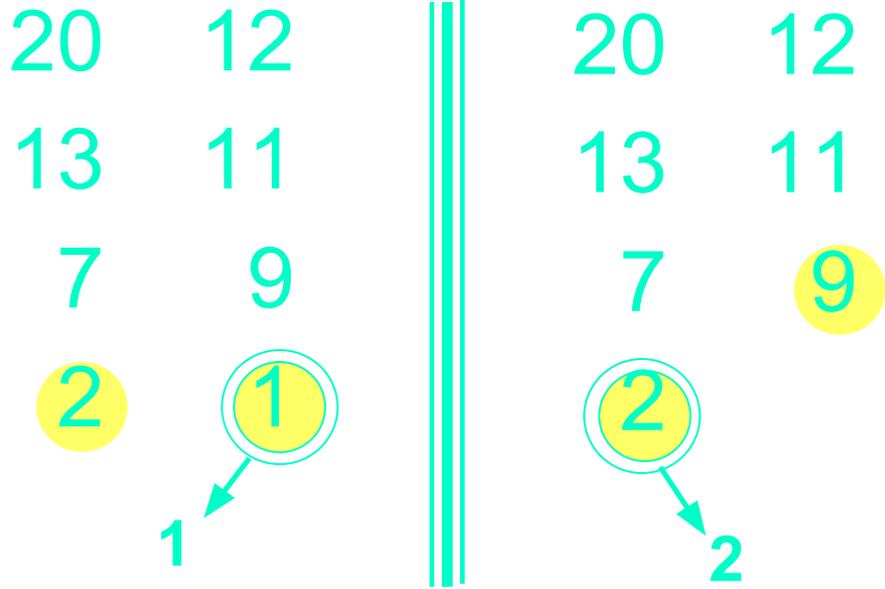
1



# MERGE SORT

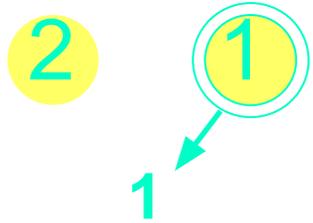


# MERGE SORT



# MERGE SORT

20 12  
13 11  
7 9



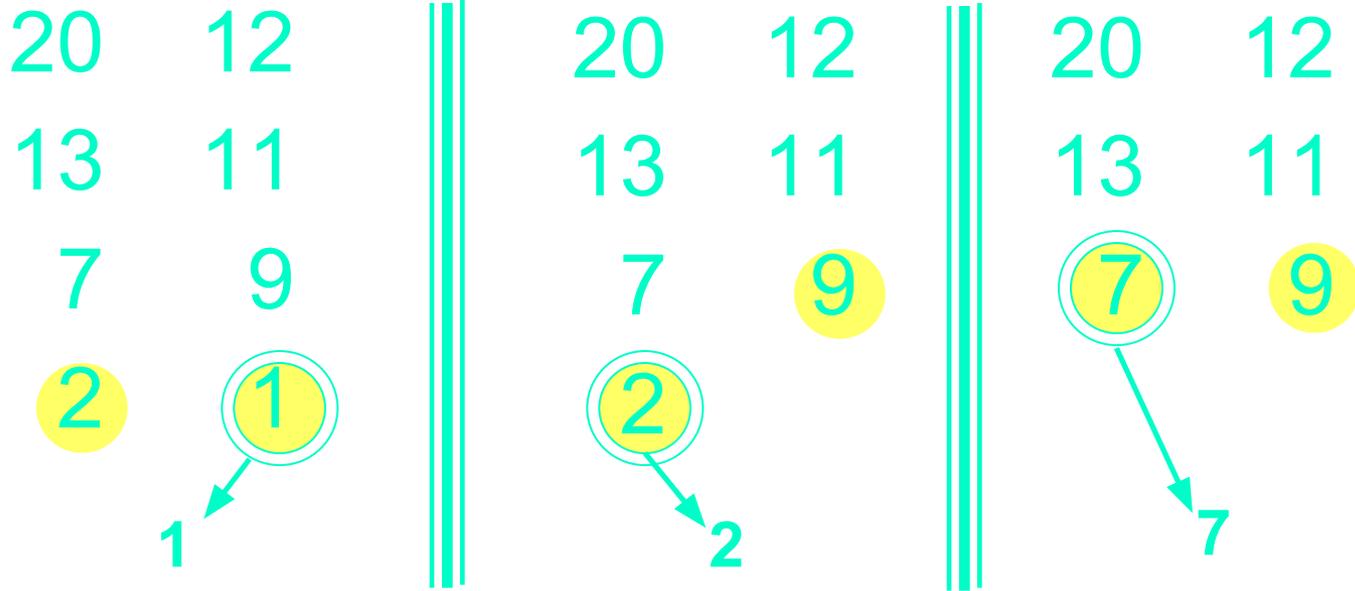
20 12  
13 11  
7 9



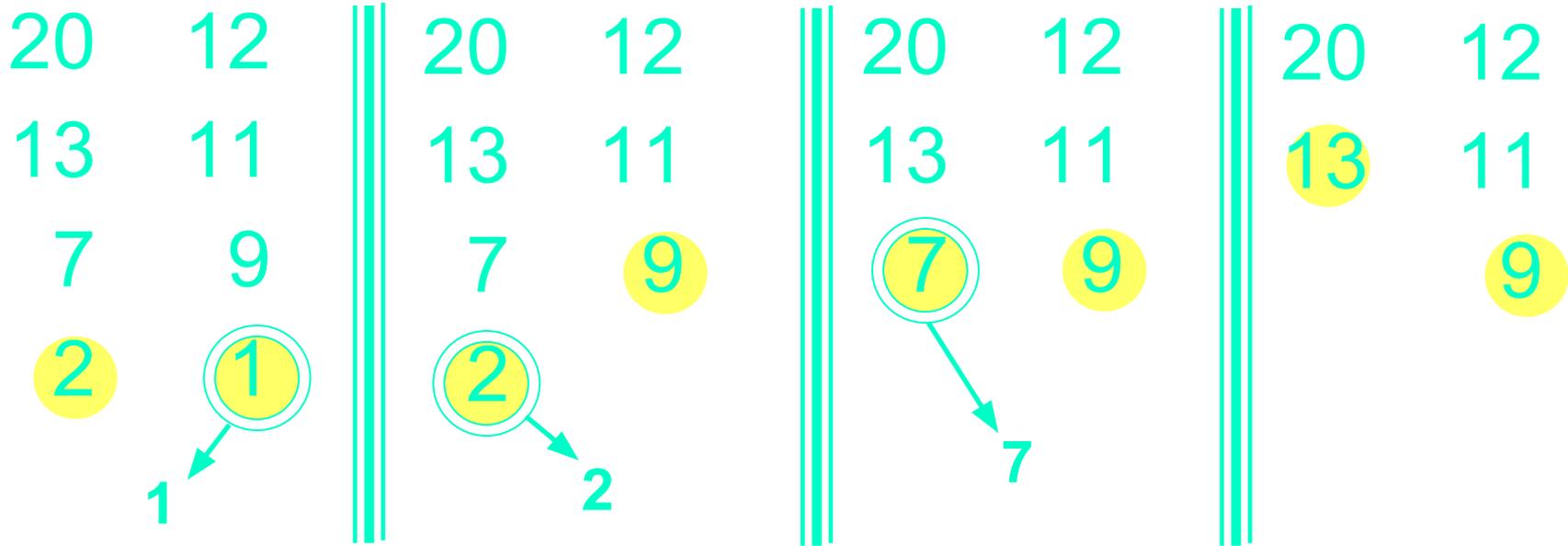
20 12  
13 11  
7 9



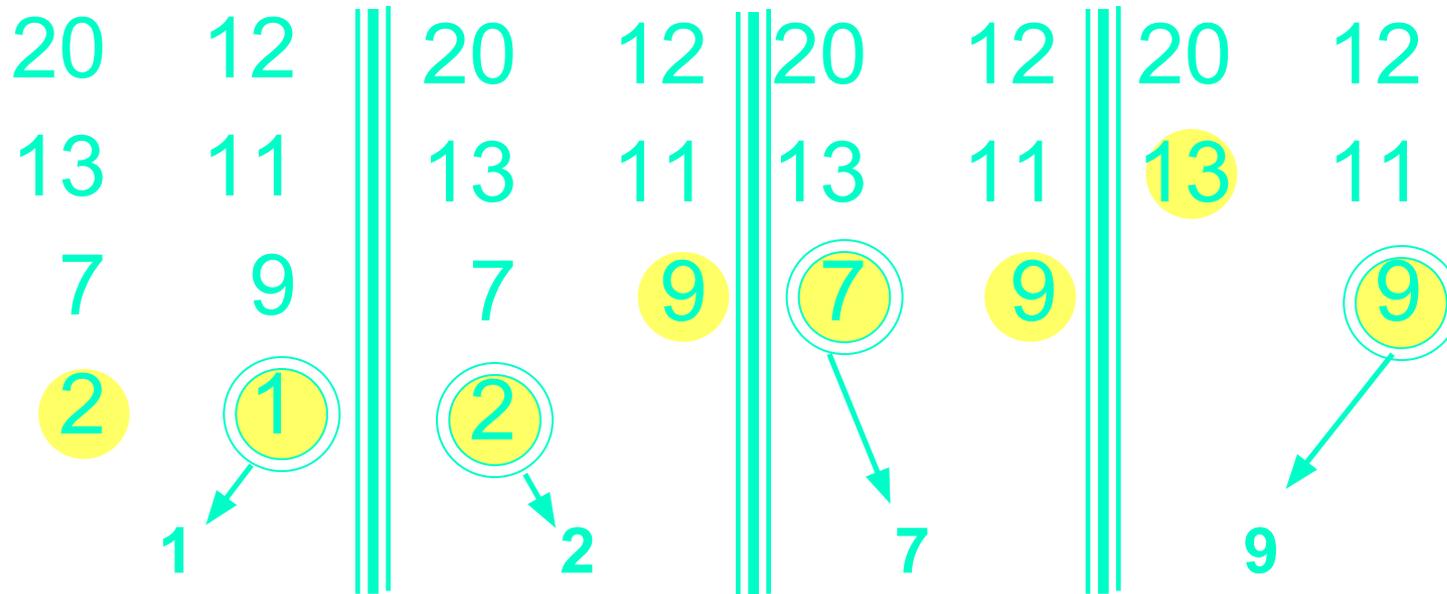
# MERGE SORT



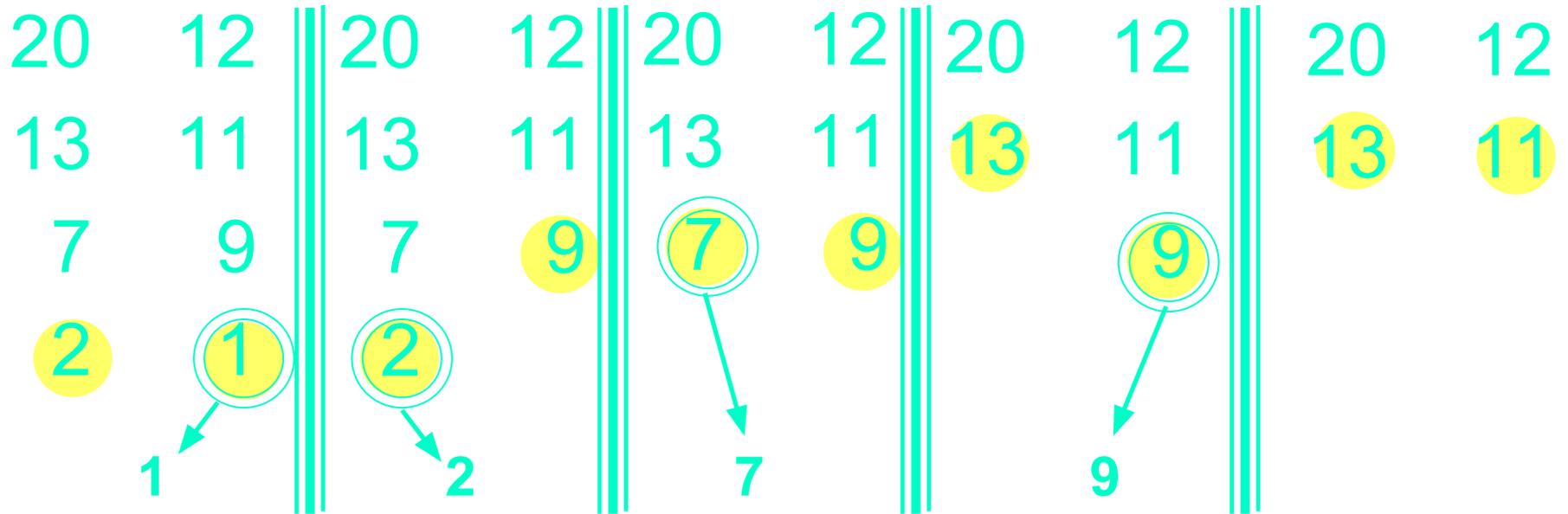
# MERGE SORT



# MERGE SORT



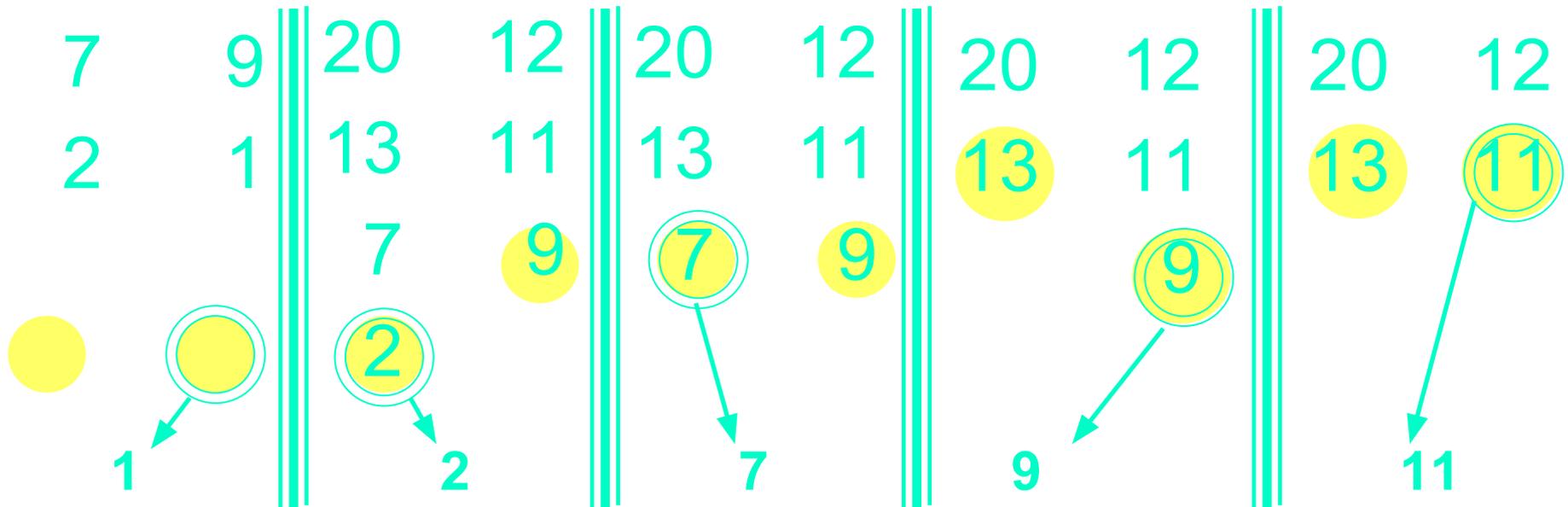
# MERGE SORT



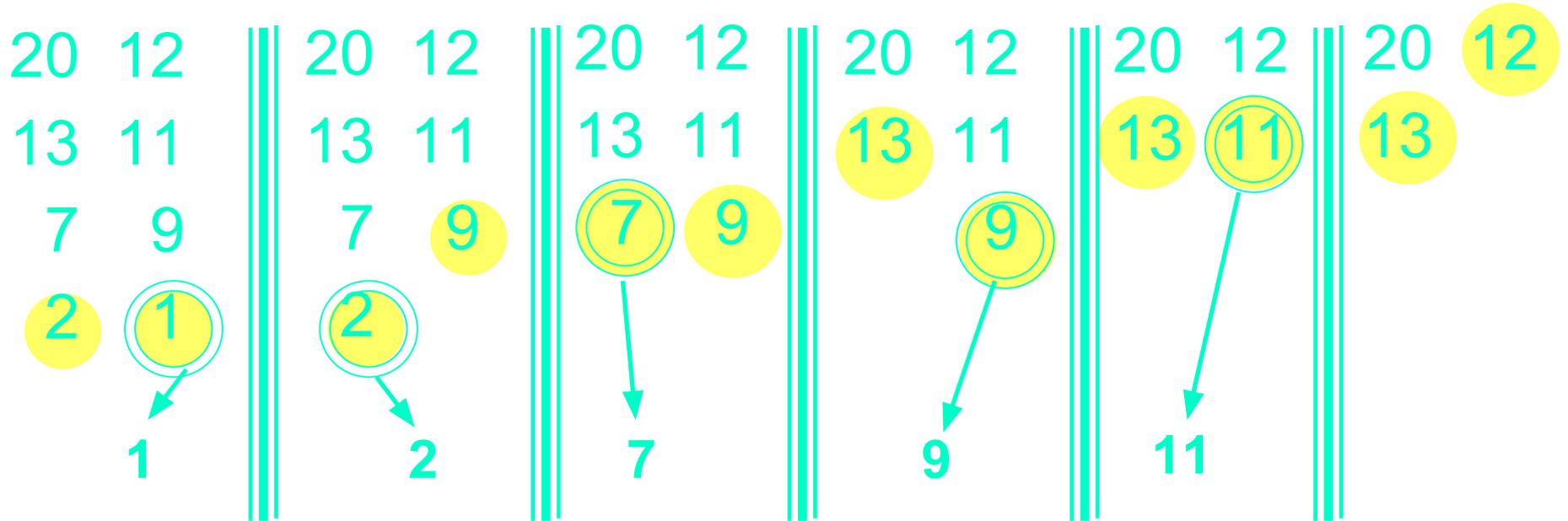
# MERGE SORT

20 12

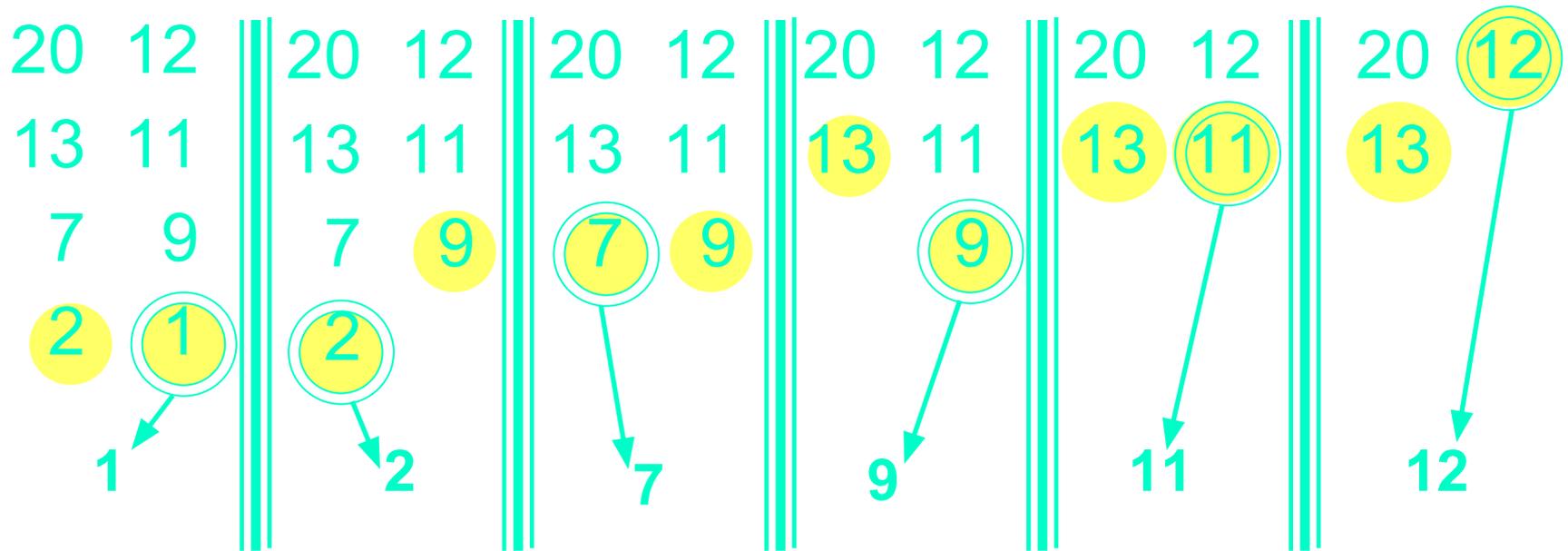
13 11



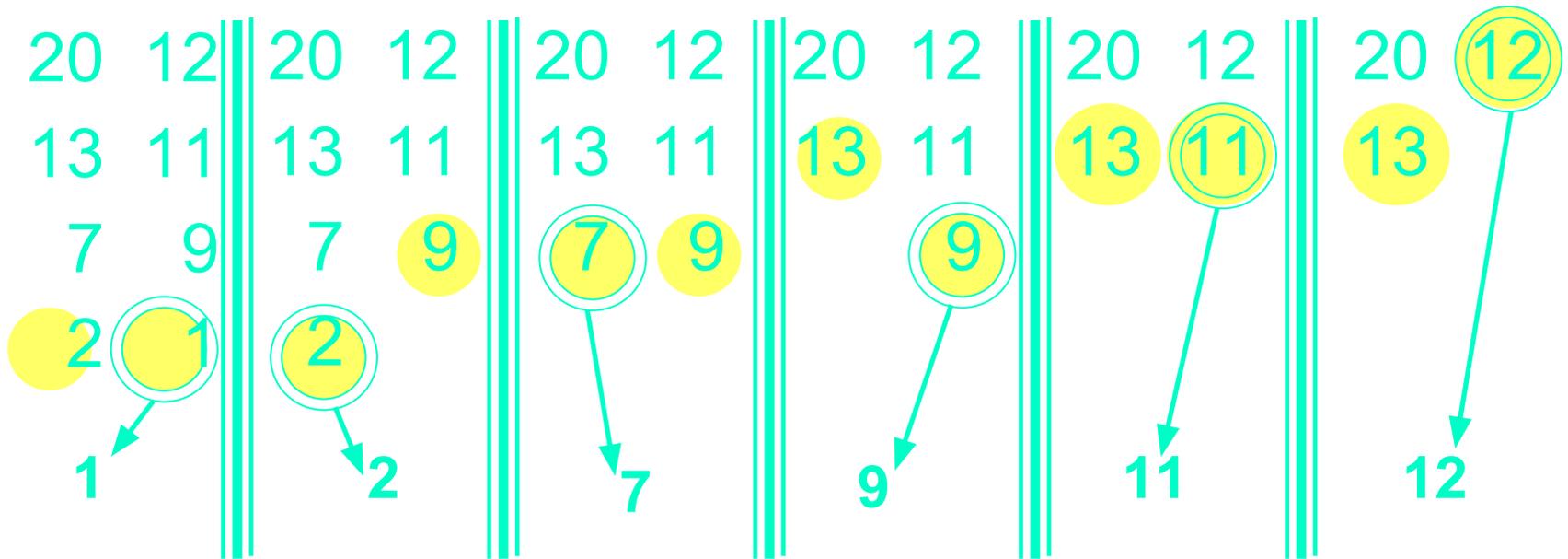
# MERGE SORT



# MERGE SORT



# MERGE SORT



# MERGE SORT

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4
5  void merge(int *a, int *b, int *c, int m, int n);
6  void mergesort(int *key, int tam);
7  void imprime(int *key, int tam);
8  int main()
9  {
10     int sz, key[]={4,3,1,67,55,8,0,4,
11                  -5,37,7,4,2,9,1,-1};
12     sz = sizeof(key)/sizeof(int); /*tamanho de key*/
13     printf("Antes do megesort:\n");
14     imprime(key, sz);
15     mergesort(key, sz);
16     imprime(key, sz);
17     return 0;
18 }
19
```

# MERGE SORT

```
20 void merge(int *a, int *b, int *c, int m, int n)
21 {
22     int i=0, j=0, k=0;
23     while(i<m && j<n){
24         if(a[i]<b[j]){
25             c[k++]=a[i++];
26         }
27         else{
28             c[k++]=b[j++];
29         }
30     }
31     while (i<m)//Valores
32         c[k++]=a[i++];
33     while (j<n)
34         c[k++]=b[j++];
35 }
36
```

```
37 void mergesort(int *key, int tam)
38 {
39     int j, k, m, *w;
40
41     for(m=1; m<tam; m*=2);/*m é potência de 2*/
42     if(n<m){
43         printf("Erro: Tamanho do vetor != 2^n - %d %d",n,m);
44         exit(1);
45     }
46     w = calloc(n, sizeof(int));
47     assert(w!=NULL);
48     for(k=1; k<n; k*=2){
49         for(j=0; j<n-k; j+=2*k){
50             merge(key+j, key+j+k, w+j, k, k);
51         }
52         for(j=0; j<n; ++j){
53             key[j]=w[j];
54         }
55     }
56     free(w);
57 }
```

# PONTEIROS E CONST

- **const** é um qualificador que informa ao compilador para não permitir alterações em determinada variável.
  - O compilador pode acusar erro ou apenas alertar o usuário.
- Princípio do menor privilégio
  - Conceda a uma função acesso suficiente aos dados para que realize a tarefa especificada e não mais que isso (Deitel, 2011).
- Ponteiro não constante para dados constantes:

```
1  #include<stdio.h>
2
3  void imprimeString(const char *s)
4  {
5      for(;*s!='\0';s++){
6          printf("%c",*s);
7      }
8      printf("\n");
9  }
10
11 int main(){
12     char string[100];
13     printf("Digite um nome:");
14     scanf("%s",string);
15     imprimeString(string);
16     return 0;
17 }
18
```

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  void umaFuncao(const int *a){
5      *a *= 2;
6  }
7
8  int main(){
9      int x;
10     umaFuncao(&x);
11     return 0;
12 }
13
```



# PONTEIROS E CONST

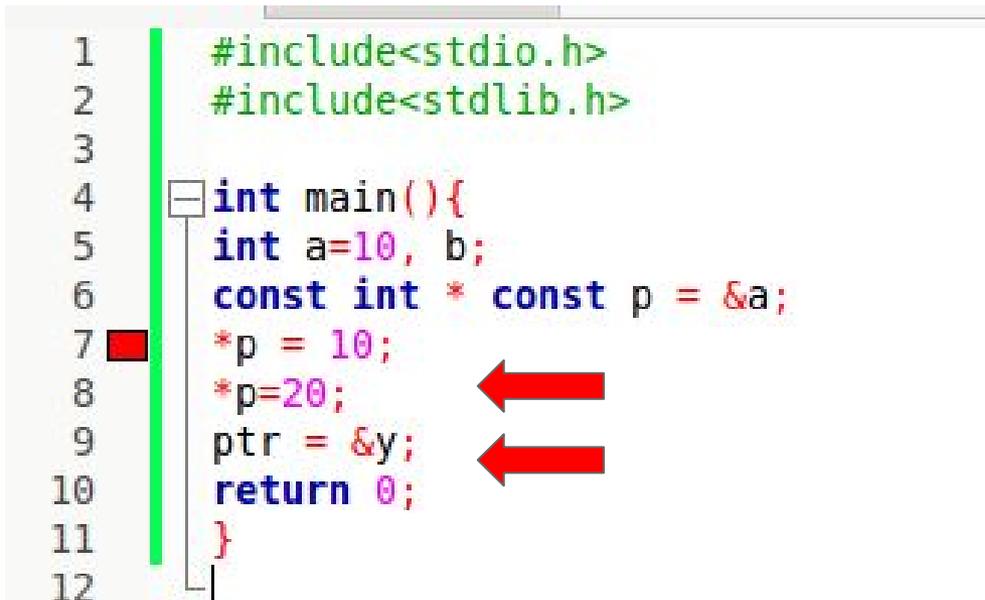
- Ponteiro constante para dados não constantes:
  - Ponteiro constante sempre aponta para o mesmo local de memória, ainda que os dados ali contidos sejam modificados.
  - Recurso adotado para vetor, onde o nome dos vetor representa um ponteiro constante.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(){
5      int a, b;
6      int * const p = &a;
7      *p = 10;
8      *p=20;
9      ptr = &y; ←
10     return 0;
11 }
12
```

# PONTEIROS E CONST

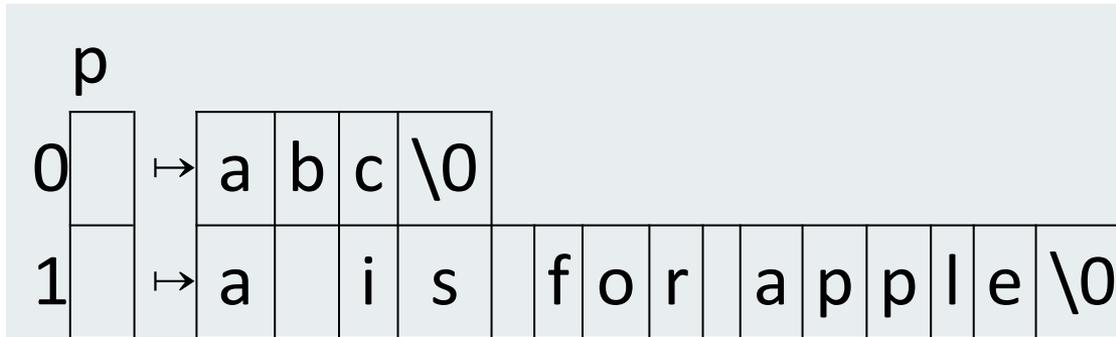
- Ponteiro constante para dados constantes:
  - Privilégio de mínimo acesso.
  - O ponteiro sempre aponta para o mesmo local de memória e os dados não podem ser modificados.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(){
5  int a=10, b;
6  const int * const p = &a;
7  *p = 10;
8  *p=20;
9  ptr = &y;
10 return 0;
11 }
12
```



# VETOR DE PONTEIROS

```
char *p[2]={"abc","a is for apple"}
```



- `char *p[2];`
- O identificador `p` representa um vetor de ponteiros.
- A declaração de `p` aloca espaço para dois ponteiros.
- O ponteiro `p[0]` é iniciado apontando para a cadeia de caracteres "abc:" que requer 5 espaços do tipo `char`.
- `printf("%s",p[0]);`  $\Rightarrow$  `abc`

# VETOR DE PONTEIROS

A

char A[2][15]={"abc", "a is for apple"}

a	b	c	\0											
a		i	s		f	o	r		a	p	p	l	e	\0

- p trabalha com menos espaço do que char A[2][15].
- Observe A[0][14] sendo posição válida, mas p[0][14] não.
- As cadeias de caracteres apontadas por p[0] e p[1] não podem ser modificadas, pois p[0] e p[1] são cadeias de caracteres constantes.
- As cadeias de caracteres de A[0] e A[1] podem ser alteradas.

p

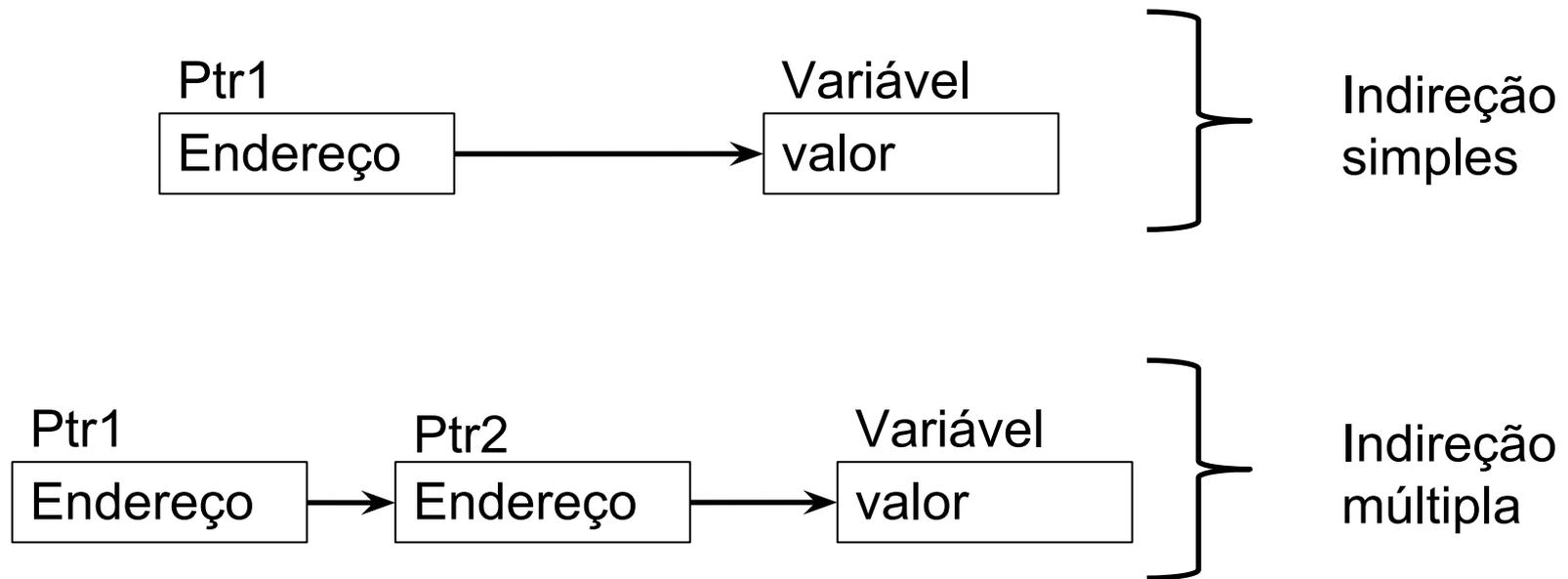
0	→	a	b	c	\0											
1	→	a		i	s		f	o	r		a	p	p	l	e	\0

# VETOR E PONTEIROS

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6  char *p[2]={"abc","a is for apple"};
7  char A[2][15]={"abc","a is for apple"};
8
9  printf("p=%s\n%s\n",p[0],p[1]);
10 printf("A=%s\n%s\n",A[0],A[1]);
11
12 return 0;
13 }
14
```

# PONTEIROS PARA PONTEIROS

- Indireção múltipla ou ponteiros para ponteiros ocorre quando temos ponteiro apontando para outro ponteiro que aponta para um determinado valor.



# PONTEIROS PARA PONTEIROS

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

```
int main(void){  
    int x, *p, **q;  
    x=10;  
    p=&x;  
    q=&p;  
    printf("%d",**q);  
    return 0;  
}
```

- Um ponteiro para um ponteiro deve ser declarado com a adição de mais um \*.
- `int **q;`
  - Indica que q é um ponteiro para um ponteiro do tipo int.
- `**q`
  - acessa o valor apontado pelos ponteiros.

# PONTEIROS PARA FUNÇÕES

- O ponteiro para função armazenará o endereço da função na memória.
- Da mesma forma que uma variável, a função tem posição física na memória capaz de ser armazenada por um ponteiro.
- No processo de compilação, o código-fonte da função é transformado em código-objeto que possui um ponto de entrada.
- Uma chamada à função durante a execução do programa dispara uma chamada em linguagem de máquina para o ponto de entrada da função.
- Logo, o endereço de uma função é o ponto de entrada da função.
- O nome da função sem parênteses fornece o endereço da função.

# PONTEIROS PARA FUNÇÕES

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <time.h>
5  #define Aleatorio(Min,Max) Min + rand()%(Max-Min+1)
6  int *criaVetor();
7  void bubbleSort(int *, int, int (*)(int , int ));
8  void inverte(int *v1, int *v2);
9  int crescente(int , int );
10 int decrescente(int , int );
11 void imprime(int *, int);
12 int main()
13 {
14     int op,tam,*seq;
15
16     srand(time(NULL));
17     printf("Digite 1 para ordem crescente\nDigite 2 para ordem decresce
18     scanf("%d",&op);
19     seq = criaVetor(&tam);
20     imprime(seq,tam);
21     if(op==1){
22         bubbleSort(seq,tam,crescente);
23     }
24     else{
25         bubbleSort(seq, tam, decrescente);
26     }
27     imprime(seq,tam);
28     return 0;
29 }
30
```

# PONTEIROS PARA FUNÇÕES

```
31  int *criaVetor(int *n){
32      int i, *vet;
33      *n = Aleatorio(5,20);
34      vet = (int*) calloc(*n, sizeof(int));
35      assert(vet!=NULL);
36      for(i=0; i<*n; i++){
37          vet[i]=Aleatorio(-10,10);
38      }
39      return vet;
40  }
```

# PONTEIROS PARA FUNÇÕES

```
43 void bubbleSort(int *v, int tam, int (*avalia)(int x, int y)){
44     int i,j;
45
46     for(i=1; i<tam;i++){
47         for(j=0; j<tam-1;j++){
48             if((*avalia)(v[j],v[j+1])){
49                 inverte(&v[j],&v[j+1]);
50             }
51         }
52     }
53 }
54
55
56 void inverte(int *valor1, int *valor2){
57     int aux;
58     aux = *valor1;
59     *valor1=*valor2;
60     *valor2=aux;
61 }
62
63 int crescente(int v1, int v2){
64     return v2<v1;
65 }
66
67 int decrescente(int v1, int v2){
68     return v2>v1;
69 }
```