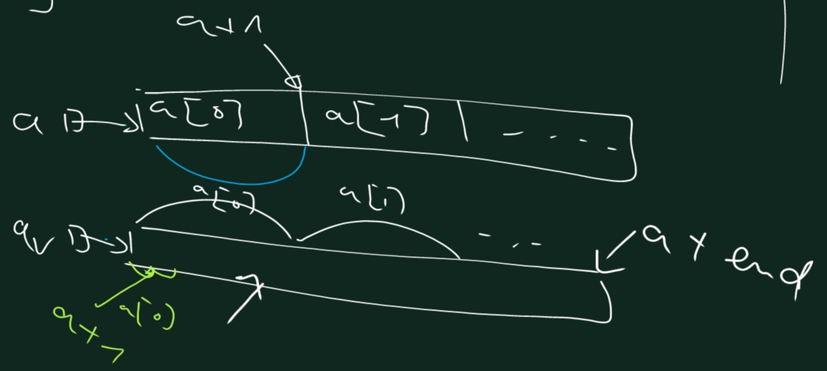
```
typedef int (*compfun)(int,int);
void sort(int* a, int size, compfun comp){
  int l,r;
  for(l=0; l<size-1; l++)
    for(r=l+1; r<size; r++)
      if(comp(a[l],a[r])>0){
        int tmp=a[l];
        a[l]=a[r];
        a[r]=tmp;
      }
}
```



sort.h (+ projet)

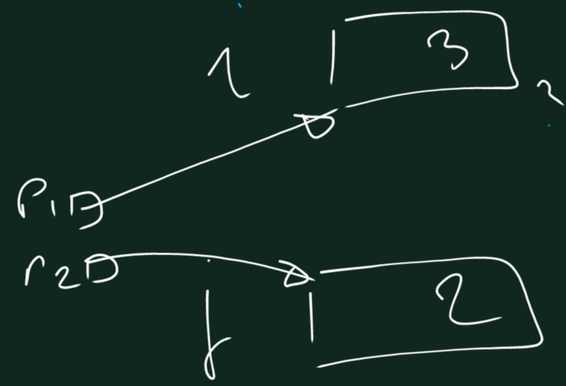
```
typedef int (*compfun)(void*, void*);
void sort(void* a, int size, int eltSize, compfun comp){
  int l,r; int end = eltSize * size;
  void* tmp = malloc(eltSize);
  for(l=0; l<end-eltSize; l+=eltSize)
    for(r=l+eltSize; r<end; r+=eltSize)
      if(comp(a+l,a+r)>0){
        memcpy(tmp, a+l, eltSize);
        memcpy(a+l, a+r, ——);
        memcpy(a+r, tmp, ——);
      }
  free(tmp);
}
```

sort.c

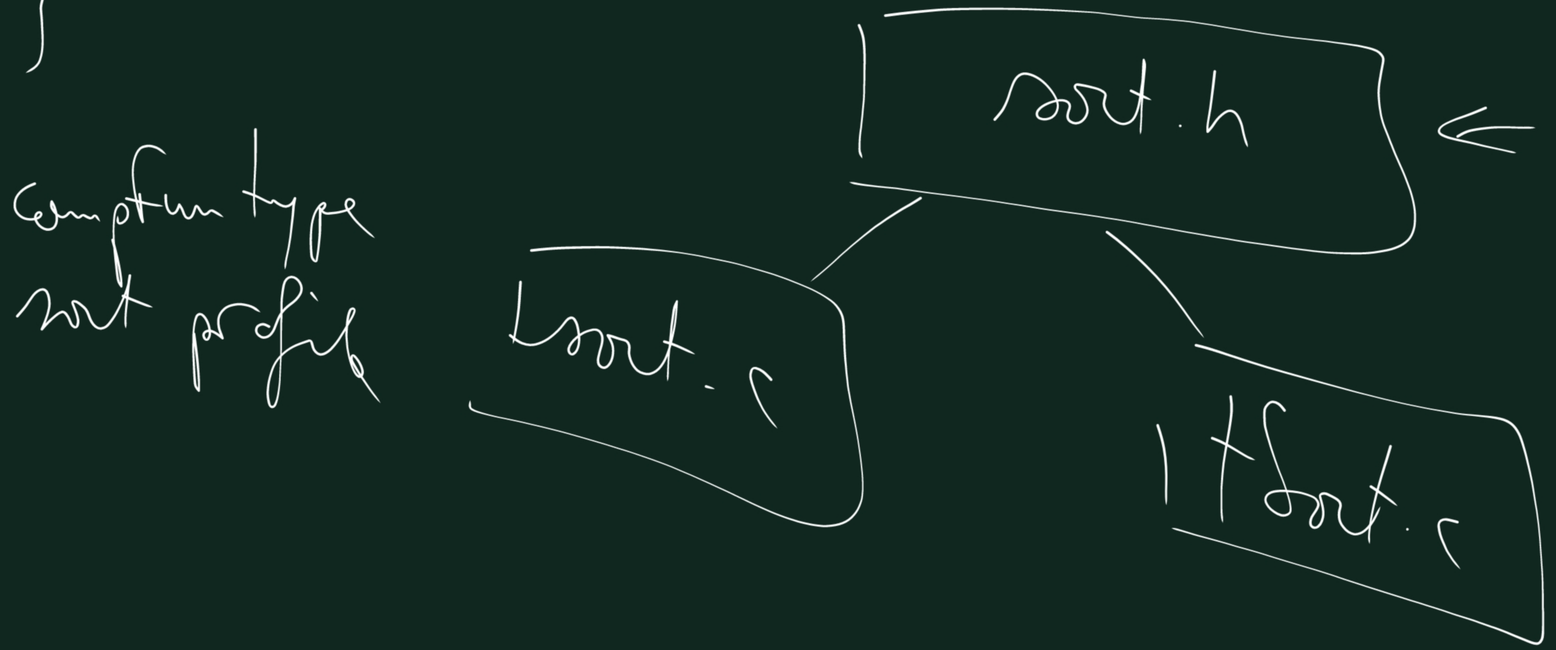
"comparator":

```
int compInt(void *p1, void *p2){
    return *p1 - *p2;
}
```

```
int compInt(void *p1, void *p2){
    return *(int*)p1 - *(int*)p2;
}
```




```
compInt(&i, &j);
```



comptype
sort profile

profile + comptype

```
compInt(+ other comptypes)
+
main
```

test program:

```
./test [-q] size
```

```
$ time ./test [-q] size
```

① create a random array of size "size"

② sort it using sort (by default) or qsort (-q option)

③ display the sorted array (if size <= 100)

④ measure time = f(size)

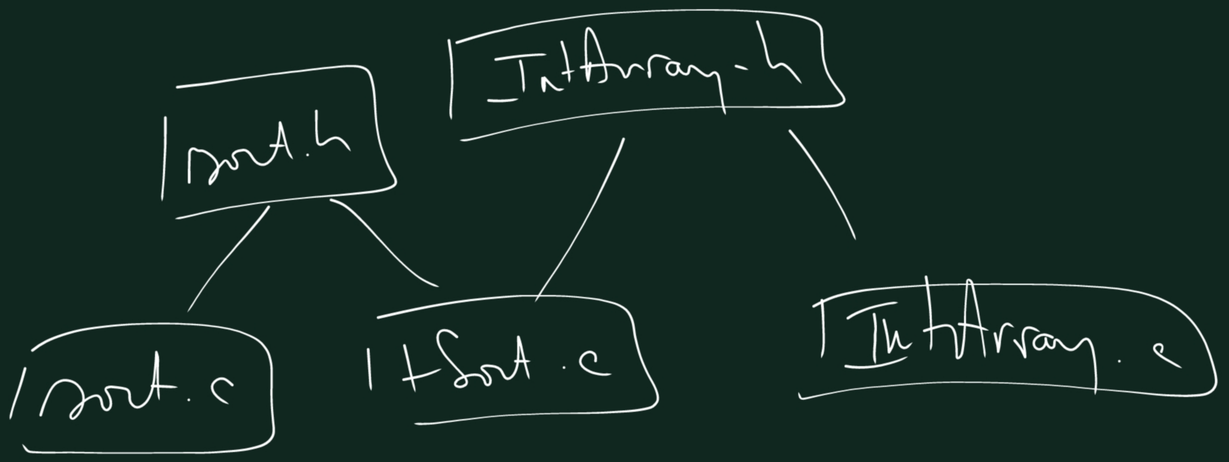
```
/* sort.h */
typedef int (*compfun)(void*, void*);
void sort(void* a, int size, int eltSize, compfun comp);
```

```
/* sort.c */
#include "sort.h"
#include <string.h>
void sort(void* a, int size, int eltSize, compfun comp) {
    ----
}
```

```
/* tSort.c */
#include "sort.h"
#include <stdio.h>
#include <stdlib.h>
#include "intArray.h"

int compInt(.....

int main (int nWords, char * words[]) {
    int size, quick ; int *a;
    if (nWords == 3) {
        quick = 1;
        size = atoi (words[2]);
    } else { quick = 0;
        size = atoi (words[1]);
    }
    a = createRandomArray (size);
    if (!a) return 1;
    if (quick)
        qsort (a, size, sizeof (int),
               compInt);
    else
        sort (a, size, sizeof (int),
              compInt);
    if (size < 100)
        displayArray (a, size);
    destroyArray (a, size);
    return 0;
}
```




```
# makefile
OBJ = sort.o tSort.o IntArray.o

tSort : $(OBJ)
	cc -o tSort $(OBJ)

tSort.o : sort.h IntArray.h
sort.o : sort.h
IntArray.o : IntArray.h
```

```
tSort 10
tSort -9 10
```

```
tmm tSort 4000
```

Size	t(mst)(s)	t(qsort)(s)
4000	0.069	0.004
8000	0.254	0.006
16000	1.019	0.007
32000	4.046	0.008
64000	16.136	0.012
	(stop)	
512000		0.07
1024000		0.141
2048000		0.286
4096000		0.591
8,92000	>3 days	0.218

basic tools to generalize code :

① pointers to functions
↳ algorithm (function : sort)

② untyped pointers (void*)
↳ type (ex : Stack)

in addition, minimum error-handling system

functions → cannot fail → unchanged (ex : nothing)

→ can fail → change (ex pop)

type status : an enumeration of all encountered types of errors + a function associating a message to each error defined

```
/* status.h */
#ifndef ...
typedef enum {
    OK,
    ERROPEN,
    ERRALLOC,
    ERRINDEX,
    ERRABSENT,
    ...
    ERRUNKNOWN
} status;

char* errorMessage(status s);

#endif
```

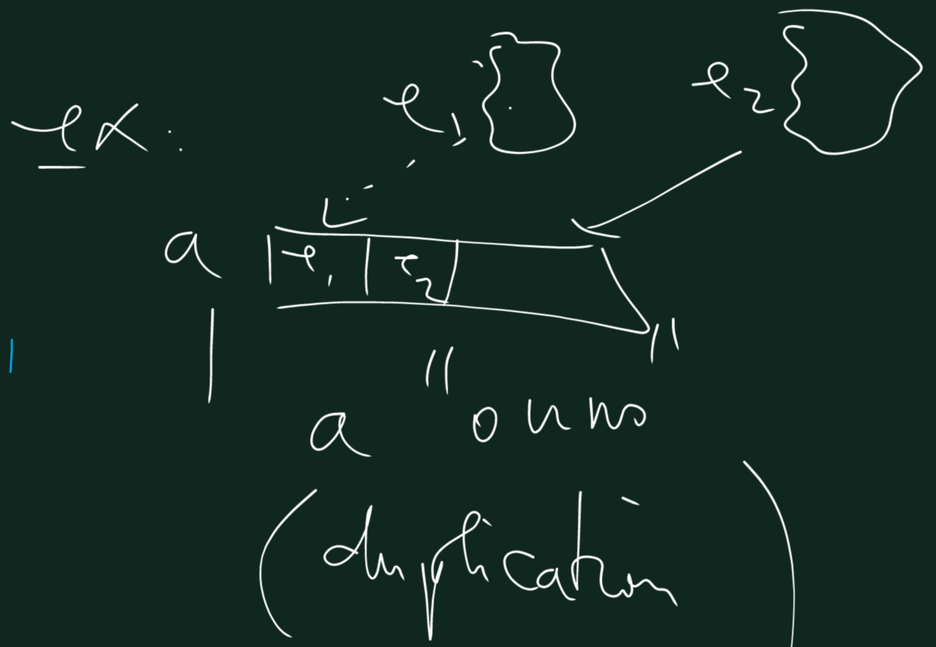
```
/* status.c */
#include "status.h"
static char* messages[] = {
    "Ok",
    "file open failed",
    "memory allocation error",
    ...
    "unknown error"
};
char* errorMessage(status s){
    return s < ERRUNKNOWN ? messages[s]
                          : messages[ERRUNKNOWN];
}
```

Generic collections and "ownership" of elements

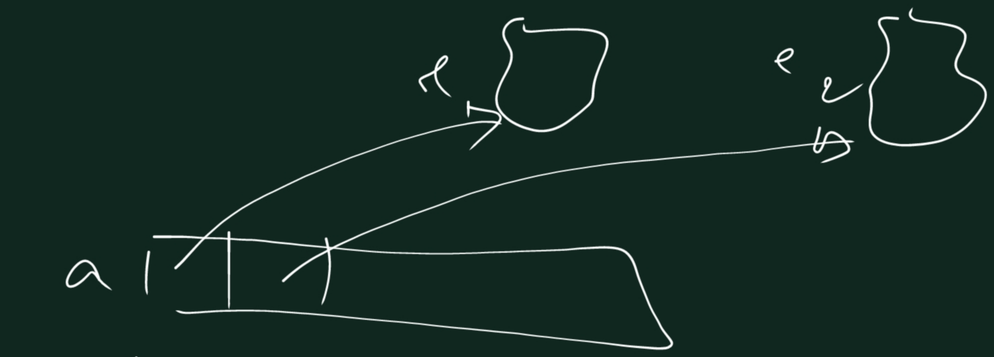
a collection can "own" the elements it contains

or "not own"

ex:



a "owns"
(duplication)



a "does not own"
(no duplication)

Generic stack (linked, not owner)


```
Stack* s = createStack();
push(s, &e1); push(s, &e2);
```

```
Stack* createStack();
int isEmpty(Stack* s);
Status push(Stack* s, void* e);
Status top(Stack* s, void* result);
```

```
/* Stack.h */
#include "status.h"
typedef struct Box {
    void* data;
    struct Box* link;
} Box, *Stack;
```

```
Status pop(Stack* s);
void destroyStack(Stack* s);
```

Process for using type status:

* if function cannot fail, no change
* _____ returns a ptr, _____
* if function returns void, just return status
* otherwise, { return a status, add the "normal" return as a new (output) param

test program:

- define 2 integers (1, 2) and a stack
- push the ints onto the stack
- while stack not empty,
  - print top
  - pop stack
- destroy stack

2

1

```
/* tStack.c */
#include <stdio.h>
#include "Stack.h"

int main() {
    int i=1, j=2;
    status st;
    Stack *s = createStack();
    if (!s) return 1;
    st = push(s, &i);
    if (st) { puts(errorMessage(st)); return 2; }
    st = _____ 3
    if _____
    {
        while (!isEmpty(s)) {
            int res;
            st = pop(s, &res);
            if (st) _____ 3
            printf("%d\n", res);
            st = pop(s);
            if (st) _____ 5
        }
        destroyStack(s);
        return 0;
    }
```