First Class ADT Pattern

Encapsulation is a tool for **managing complexity** and to **separate aspects of a problem**.

Two views of a design: the **external behavior** (interface) and the **internal details** (implementation).

ADT: Abstract Data Type - a set of values and operations on these values [1]

FIRST-CLASS ADT : "a first-class citizen [..] is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable." [2]

Exercise

Write a program that manages Order and Customers

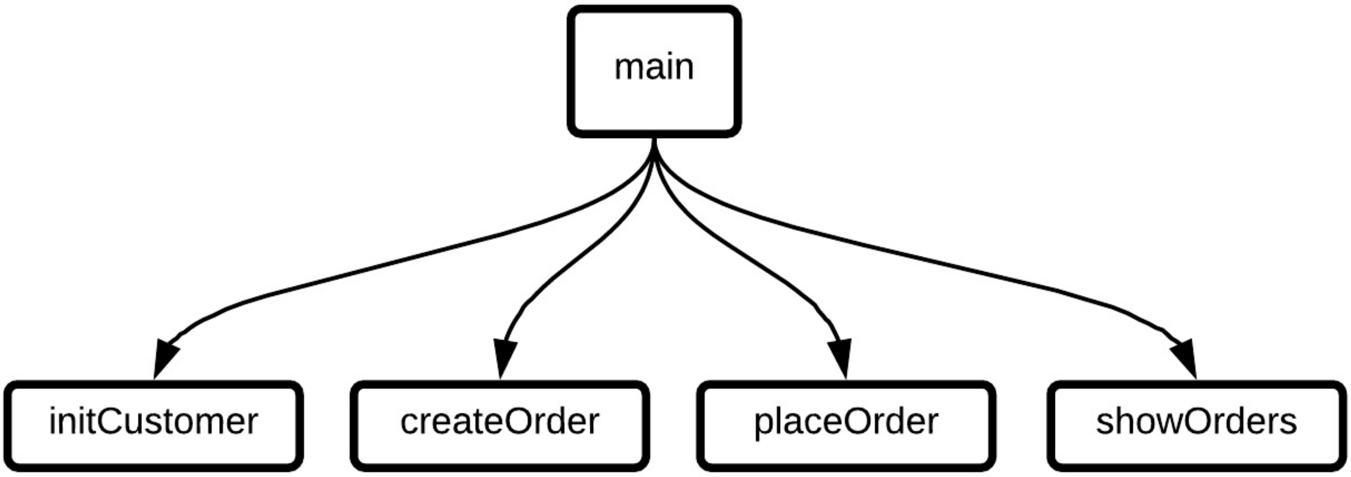
data:
- order: idOrder, price
- customer: name, number of orders, list of orders.

functions:
- `initCustomer()` initializes a Customer
- `createOrder()` creates an Order
- `placeOrder()` adds an *order* to the *list of orders* contained in a Customer
- `showOrders()` shows all the Orders in a Customer

---

functions:

- `initCustomer()` initializes a Customer
- `createOrder()` creates an Order
- `placeOrder()` adds an *order* to the *list of orders* contained in a Customer
- `showOrders()` shows all the Orders in a Customer



A typical `main` :

```
int main(){
    // Create 2 customers and 4 orders
    // Show that customers contain NO order, for now.
    // ...

    // add orders to customers
    // ...

    // Show that customers contain order, now!
    // ...

    getchar();
    return 0;
}
```

A typical `main` :

```c
// include...
int main(){
    Customer aCustomer_1, aCustomer_2;
    Order anOrder_1, anOrder_2, anOrder_3, anOrder_4;

    initCustomer(&aCustomer_1,"Luca");

    // attenzione: avrei potuto scrivere anche
    // aCustomer_1.name="Luca";
    // aCustomer_1.nOfOrders=0;
    // accedendo direttamente ai campi della struttura
    // i dati NON sono incapsulati

    initCustomer(&aCustomer_2,"Franco");
    printf("\n\n");
    showOrders(&aCustomer_1);
    showOrders(&aCustomer_2);

    anOrder_1=createOrder(1,100);
    anOrder_2=createOrder(2,200);
    anOrder_3=createOrder(3,300);
    anOrder_4=createOrder(4,400);

    placeOrder(&aCustomer_1, anOrder_1);
    placeOrder(&aCustomer_1, anOrder_2);
    placeOrder(&aCustomer_2, anOrder_3);
    placeOrder(&aCustomer_2, anOrder_4);

    printf("\n\n");
    showOrders(&aCustomer_1);
    showOrders(&aCustomer_2);

    getchar();
    return 0;
}
```

Exercise (hint)

3 files: main.c, customer.c, customer.h

The data definition (Order, Customer) are in the interface customer.h

Customer contains a list (a static array) of orders

Solve the exercise!

https://codeboard.io/projects/139782

(Solution in https://github.com/DIEE-PAIS-code/placeOrder_ADT_01_c)

PROBLEMS

Lack of information hiding:

- details of the module customer are unveiled to the client (see `customer.h`)
- if we change the implementation (i.e. linked list instead of array) we must inform the client
- the client has access to the data structure

SOLUTION

The C standard [C 1999] allows us to declare objects of incomplete types in a context where there sizes aren’t needed.
We can specify **a pointer to an incomplete type**.

See the code in
https://github.com/DIEE-PAIS-code/ADT_01_c

SOLUTION

- A pointer to an incomplete type

```
typedef struct Customer* CustomerPtr;
```

provides the possibility to handle the entity Customer, without being able to directly access to the data.

Try to solve the problem before looking at the solution in https://github.com/DIEE-PAIS-code/placeOrder_ADT_02_c

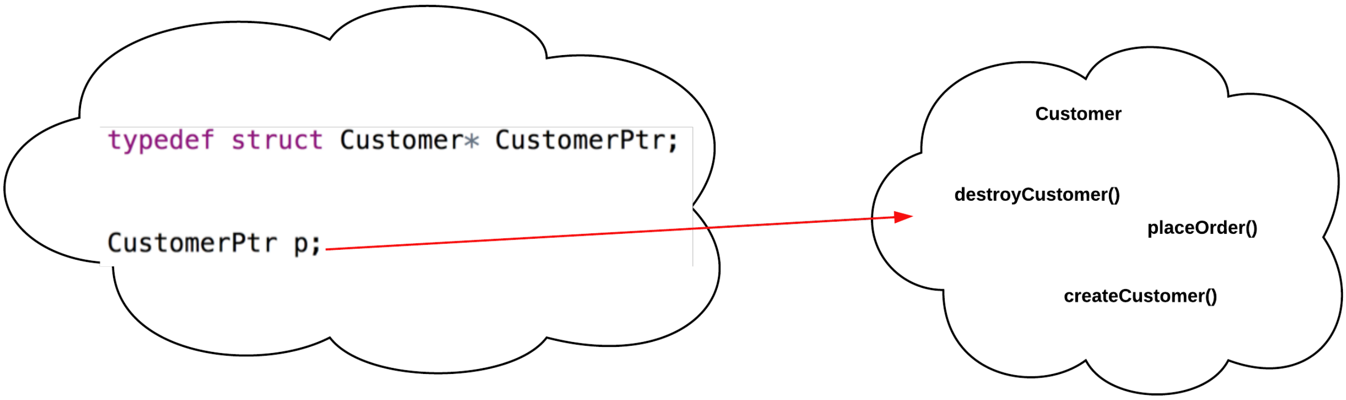
- A complete set of functions to manipulate the data

```
createCustomer(const char* name);

/* Destroy the given Customer.
All handles to it will be invalidated. */
void destroyCustomer(CustomerPtr p_customer);

// add an order
void placeOrder(CustomerPtr p_customer, Order anOrder);

// show orders
void showOrders(CustomerPtr p_customer);
```



```
typedef struct Customer* CustomerPtr;

CustomerPtr p;
```

Allocation/deallocation

```
struct Customer {
    const char* name;
    int nOfOrders;
    Order orders[MAX_NO_OF_ORDERS];
};
void destroyCustomer(CustomerPtr p_customer) {
    /* Perform clean-up of the customer internals, if necessary. */
    free(p_customer);
}
```

NB We must perform clean-up of the customer internals, if necessary.

i.e.

```
struct Customer {
    const char* name;
    int nOfOrders;
    Order * p_linkedOrders;
};
```

Other details (not related with ADT pattern)

```
void initCustomer(Customer * p_customer, const char* name){
```

It ensures that the function can not change the value pointed to by name

```
const char *name;
```

name is an ordinary, modifiable pointer, but the thing that it points to must not be modified.

```
char *const name = &c;
```

name is a pointer to a char. The pointer is constant, not the thing that it points to.

```
Customer * p_customer = malloc(sizeof Customer);
```

Problem: If we change `Customer` into `NewCustomer`, we need to change the code in 2 places!

```
NewCustomer * p_customer = malloc(sizeof NewCustomer);
```

A failure to update both places may have fatal consequences, potentially leaving the code with undefined behavior.

Solution:

```
Customer * p_customer = malloc(sizeof * p_customer);
```

or

```
CustomerPtr p_customer = malloc(sizeof * p_customer);
```

---

[1] https://en.wikipedia.org/wiki/Abstract_data_type ↩

[2] https://en.wikipedia.org/wiki/First-class_citizen ↩