ADT OBJECT: A TUTORIAL

Prologue: this tutorial is independent from a specific typed imperative programming language. Some illustrations are given for the C programming language, especially in chap. 4 and Annex B.

1. STRUCTURED SETS

Definition (structured set). A structured set is a set in the mathematical sense, where the elements satisfy some relations between them: $S_{struct} = S + R$

Classification of structured sets

1. Set : no relation at all between the elements.
2. Linear structure : each element is related to 2 elements, except for the first and last one which are only related to one element.
3. Hierarchy or tree structure: each element is related to a parent and some children, except for one called "root" that has no parent and some of them, called "leaves", that have no children.
4. Graph structure : each element e is related to e(n) elements, possibly even to itself, e(n) ≥ 0.

A graphical representation is illustrated in Fig. 1.

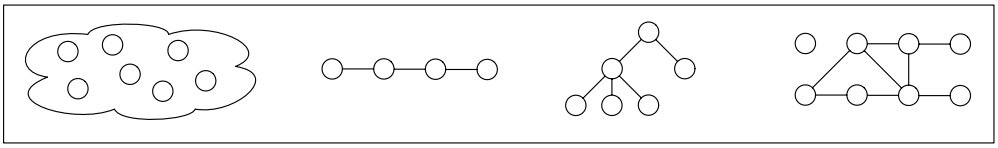

Fig. 1. Graphical representation of a set, and linear, tree and graph structures

Examples

1. Sets : set, multiset, bitset, …
2. Linear structures : sequence, also called list. There are many further attributes that can be added to specify them to more specialized linear structures:
    - stack (lifo), queue (fifo),
    - absolute or relative position,
    - recursive, circular, ordered, priority,
    - token, …
3. Trees: tree, binary tree, n-tree, search tree, heap, B tree, AVL tree, …
4. Graph structures: graph with many possible attributes: none, (non-)oriented, (k-)connected, labeled, colored, …

Typical operations on structured sets

- add, remove, belong_to
- intersection, union, complement
- exists, size
- traverse, …

About specification and implementation in a typed programming language

How can we specify and implement these kind of objects in a systematic way, such that their usage and manipulation is as easy and as reliable as possible ? A short answer is:

1. Come up with a solution which hides the data structure and algorithmic details as much as possible (and in C also the famous and a bit annoying pointers).
2. Take great care by defining a clean data structure which is shared by the functions, and possibly don't reveal it to the application level (in C use the generic pointer void *).
3. Separate the specification of the functions (in C called "prototype" or "declaration", commonly put in a header file *.h) from their implementation (in C called "definition", commonly put in a separate *.c file)

2. GENERIC ADT OBJECT MODEL

Definition. We call **object** everything that exists in a programming language: files, variables, functions, …

Definition. We define an (**A)DT object** as a *structured set of elements* (i.e. a set S where the elements satisfy some relation R between them), supplied with *operations O* acting on these elements and which must ensure that the relation R is guaranteed: $O_{(A)DT} = S + R + O$.

By **abstract data type** we mean that only the name of the data type is known to the user application, in addition to the operations. No details about the internal implementation are revealed.

Very simple example in C: a set of variables of 'type int', with the operations '+, -, * and /', constitutes a trivial ADT object (and a DT object with the bit operations '|, &, ~').

There are mainly four ways to specify and implement a general (A)DT object (like a stack, a queue, a graph, …), depending on what is visible resp. hidden to the user application, cf. Fig. 2.

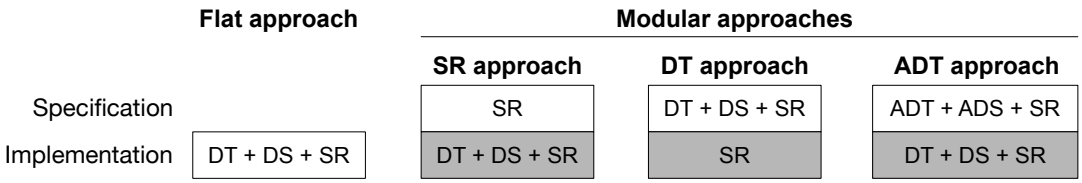

Fig. 2. Generic ADT Model: a schematic representation of flat, SR, DT and ADT approaches
White boxes are visible and accessible to the user application level, but not the gray ones.
(A)DT stands for (abstract) data type, **(A)DS** for (abstract) data structure and **SR** for subroutines.

1. Flat approach: Everything is visible to the user application: the data type, the data structure (e.g. array, linked list, …) and the implementation of the subroutines.

In C, this is reflected by having no user specification files *.h: the whole program resides in one or several *.c files (and possibly some system library files *.h, e.g. `stdio.h`).

2. SR (subroutines) approach: only the names of the subroutines are visible to the user application.

Advantage: the user application program is independent of any specific implementation of the DT object.

Disadvantage: as no (abstract) data type is revealed, only one instance is possible for users (there is no way for them to declared variables of this object type or to access its private data). This sounds to be a severe restriction, but is often sufficient.

In C, only the function prototypes acting on the elements are visible to the user application, and are declared in a header file .h. All the rest (data type, data structure, and function implementations are hidden in a separate .c file).

3. DT (data type) approach: The previous disadvantage is removed by making the data type and the data structure, used to implement the DT object, explicitly available to the user application.

Advantage: the user application can define as many variables of the DT object as needed.

Disadvantage 1: the user application can directly manipulate the data structure without using the available subroutines. This can cause serious *reliability* problems.

Disadvantage 2: for each implementation variant (e.g. using an array, a linked list, etc.), the user application need a new instance that differ only in the implementation of the data structure. This causes a serious problem of maintaining several similar programs in future inevitable code *refactoring*.

In C, this can be realized with a struct type that wraps up the type of the element.

4. ADT (abstract data type) approach: The advantages of the SR and DT approaches, without their disadvantages, can be reached by giving to the user of the ADT object just the name for the ADT object.

This approach is not only very reliable (access the object only via the specified operations), but also easy to maintain (refactoring the data structure and subroutines don't affect the user application programs).

In C, this can be realized via the generic pointer `void *`.

3. GENERIC FUNCTION CALLS FOR LINEAR (A)DT OBJECTS: Informal Description

Assumptions

- The relationship among the elements follows the protocol **P**, e.g. LIFO or FIFO or else.
- All functions, but create() and exists(), have the precondition that the ADT object *exists* or *had exist*.
- Remark: for SR approach all `obj` and `&obj` call parameters have to be removed. And for DT approach, `obj` call parameter must be changed to `&obj` for manipulator functions (see Annex B).

Constructor Functions

create(&obj,cap) : constructs a data structure of type obj_t, empty for the moment, and obj-post points to it. The created object has a capacity of cap elements (not used for UL implementations).

destroy(&obj) : frees all allocated memory to obj, and obj-post is set to a NULL pointer.

Manipulator Functions

put(obj,&e) : adds element e in P order to obj.

get(obj,&e) : copies the element in P order from obj to e, and removes it from obj.

new_capacity(obj,cap) : reallocates a new region of cap elements for obj. Does nothing for UL impl.

Access Functions

consult(obj,&e) : copies the element in P order from obj to e.

is_empty(obj) : returns true if obj has no elements, and false otherwise.

is_full(obj) : returns true if obj has reached its maximum allowed size and false otherwise.

is_mem_av(obj) :
- none semantics: returns always true
- strong semantics: returns true if obj *has enough memory* for another element and false otherwise.
- weak semantics: returns true if obj *possibly has enough memory* for another elt and false otherwise.

exists(obj) : returns true if obj exists and false otherwise.

size(obj) : returns the number of elements contained in obj.

capacity(obj) : returns the capacity of obj (-1 for UL implementations).

imp_type() : returns the implementation type: XYz
X : Bounded (B) or Unbounded (U)
Y : Array (A) or linked List (L)
z : none (n), strong (s) or weak (w) 'is_mem_av' semantics: *has or possibly has enough system memory for another elt*

imp_version() : returns the implementation version.

Traverse Function

traverse(obj,f,&buf) : calls the user supplied function f for all elements of obj, using buffer buf.

Error treatment

The error treatment can be done with a global string. If the string is empty then no error occurred. Otherwise it contains a short description of the error. Other error treatment mechanisms exist, like when a function returns a negative value, updates a global variable of type int, raises an exception, …

create : out_of_memory, negative_capacity_is_not_allowed
destroy : none

put : object_ is_full, out_of_memory
get : object_ is_empty
new_capacity : out_of_memory, new_capacity_is_too_small, negative_capacity_is_not_allowed

consult : object_is_empty
is_empty, is_full, is_mem_av : none
exists, size, capacity, imp_type, imp_version, traverse : none

4. GENERIC DATA TYPE AND DATA STRUCTURE FOR ADT OBJECTS

Figure 3 illustrates the main two possible implementations of a data type and data sructure for general ADT objects.

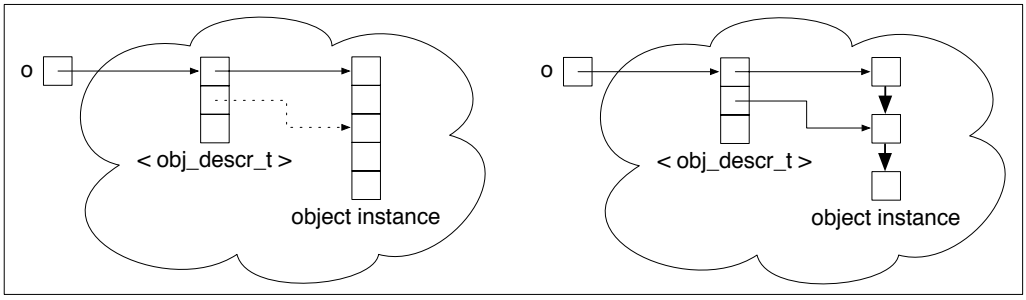

Fig. 3. A generic data type and data structure with array or linked list implementations
A dotted arrow represents a *pseudo-pointer*, a full arrow a *pointer* and a bold arrow a possible *multi-pointer.*

In C, the memory cells in the cloud are allocated by calls to `malloc()`, and are therefore stored in the heap segment. The variable `o` that points to the generic ADT object is most probably stored in the stack segment, or as global variable in the data segment. And the data types are typically realized as follows:

Header file *.h

```
typedef  void *obj_t;
```

Array implementation file *.c

```
typedef struct obj_descr_t  {
   elt_t *base; //points to elt base[0] of the ADT object array instance.
   ...    //All what is needed for the impl. of the ADT object functions.
} obj_descr_t;
```

Linked List implementation file *.c

```
typedef struct p_node_t {  // p stands for protocol P of the ADT oject
   elt_t: e; //Element to be stored in a node.
   ...    //All what is needed for the impl. of the ADT object protocol P.
} p_node_t;

typedef struct obj_descr_t {
   p_node_t *entry; //Points to the entry elt of the ADT object instance.
   ...       //All what is needed for the impl. of the ADT object functions.
} obj_descr_t;
```

Final remarks

In Annex A an example of a concise and nevertheless precise specification of a STACK is provided. Its design is valid for any imperative typed oriented programming languages.

In Annex B some possible variants for the function specification of an ADT object are described.

A full implementation of the ADT stack object in various flavors, as some others ADTs, like FIFO and token list, is available online Moodle -> Tutorials -> C01 -> adt > ADT_obj.zip.

Béat Hirsbrunner, University of Fribourg, © 2002 – 2016
Major rev. of Section 2, reviewed by Ammar Halabi, 25 April 2016

ANNEX A. STACK SPECIFICATION

Written by : Béat Hirsbrunner; Reviewed by : Georges Goy;

Elements. Although the elements can be of a variety of types, for concreteness we assume that they are of type *StdElement*.

Structure. The main feature of a stack is that it follows a last-in / first-out (LIFO) protocol. The relationship among elements is linear.

Operations. The operation *Create* brings an initially empty stack into existence and must be executed before any other operations are used. Thus, all but *Create* have the additional precondition that the stack exists.

Occasionally in the postcondition we must reference the value of the stack *s* immediately before execution of the operation. We use *s-pre* as notation for this value.

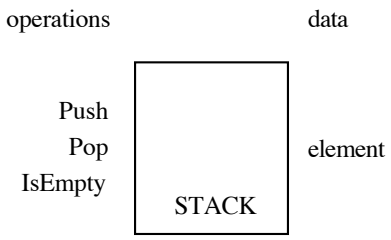

Figure 1. Stack Capsule with the basic data and operations.

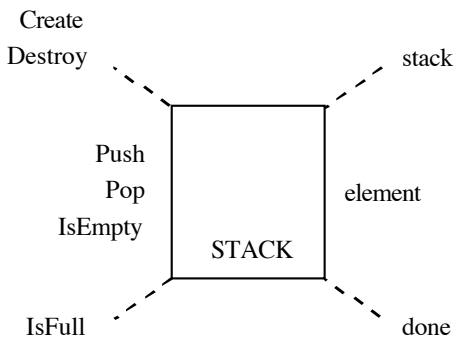

Figure 2. Stack Capsule with Modula-2 oriented «begin-end» and «error handling» mechanisms.

TYPES

type StdElement **is struct** (Key, Data);
type Stack **is private**;
type Done **is** Boolean;

BASIC OPERATIONS

Push (**in out** s; **in** e);

> **pre** — The stack *s* is not full.
> **post** — The stack includes *e* as its most recently arrived element.

Pop (**in out** s; **out** e);

> **pre** — The stack *s* is not empty.
> **post** — The element *e* is the most recently arrived element of *s-pre*. The stack *s* no longer contains *e*.

IsEmpty (**in** s) **return** Boolean;

> **post** — Returns true if *s* has no elements and false otherwise.

«BEGIN - END» OPERATIONS

Create (**out** s; **out** done);

> **post** — If a new stack can be created then *s* exists and is empty, and *done* is true. Otherwise *done* is false.

Destroy (**in out** s);

> **post** — The stack *s* no longer exists.

«ERROR HANDLING» OPERATIONS

IsFull (**in** s) **return** Boolean;

> **post** — Returns true if *s* has reached its maximum allowed size and false otherwise.

ANNEX B. SOME VARIANTS OF FUNCTION SPECIFICATIONS OF AN ADT OBJECT

In this annex, some function specification variants are illustrated in **C**. So let be:

```
typedef void        *obj_t;  // ADT
typedef obj_descr_t *obj_t;  // DT*
typedef obj_descr_t  obj_t;  // DT
```

Classical C prototypes

CONSTRUCTORS: create or destroy an object and return the handler of the object descriptor (ADT, DT*) or return/update the object descriptor itself (DT):

```
c)  obj_t create();             obj_t destroy(obj_t  obj);
c*) void  create(obj_t *obj);   void  destroy(obj_t *obj);
```

MANIPULATORS: update the object and/or its descriptor, but without changing the handler itself, e.g.:

```
m)  void put(obj_t  obj; elt_t e);  // or 'elt_t *e' for opt. reason
m*) void put(obj_t *obj; elt_t e);  // or 'elt_t *e' for opt. reason
```

ACCESS FUNCTIONS: the object and its descriptor are only read, e.g.:

```
a)  void size(obj_t  obj);
a*) void size(obj_t *obj);  for optimization reason
```

Classical usage

ADT: `c` or `c*`, `m` (`m*` is possible, but not natural), `a` (`a*` is possible, but not natural)
DT*: same as ADT
DT: `c` or `c*`, `m*` (`m` is not possible), `a` (`a*` is possible for optimization reasons)

Remark 1 : `c` strictly respects the semantics of 'out' for create and 'inout' for destroy. In this respect, `c*` is a bit fuzzy for create, but nevertheless often used.

Remark 2 : In C, a function can return a 'struct' that can be of any size (i.e. not limited to the maximum size of a pointer, as in many languages). This is why `c` is also possible for DT.

Remark 3 : if necessary following terminology could be adopted (and used for file naming) :

```
adt1   : c-m-a     : classical ADT and DT*
adt2   : c*-m-a    : classical ADT and DT* (we finally opted for this standard)

dt1    : c-m*-a    : classical DT
dt2    : c*-m*-a   : classical DT (we finally opted for this standard)
dt_opt : c*-m*-a*  : optimized DT if sizeof(obj_descr_t) >> sizeof(void *)
```