CONTAINERS.SIM

first version: June 1995
current version: 6.4 (Jan. 1997)
documentation updated: Feb 5, 1998

Jean G. Vaucher
Département d'informatique et recherche opérationnelle
Université de Montréal
E-Mail : vaucher@iro.umontreal.ca

OVERVIEW

CONTAINERS is a SIMULA package which implements the following traditional data structures:

- Queue
  - sequence
- Priority_queue
  - heap
- Stack
- Dictionary
  - simple_dictionary
- Table

In this package, we use the term *CONTAINER* to denote objects which behave according to an abstract interface; the term, *data structure* is reserved for concrete implementation.

The package was inspired by Bjorn Kirkerud's SETTOOLS but has been redesigned and rewritten several times to serve not only as a tool but as an example of object programming.

The basic CLASS defined in the package is ELEMENT. All objects to be put into our containers must be sub-classes of ELEMENT (much as LIST elements in SIMSET must be prefixed by LINK). ELEMENTs should include a definition for a **KEY** function which returns a TEXT **identifier** representative of the element. The KEY is used by many operations in CONTAINERS, for example: to print out the contents of a data structure, or to order elements in a priority queue.

Note 1:
Elements can be in several containers at once (unlike in SIMSET where elements can only be in one list at a time). Note also that CONTAINER is a sub-class of ELEMENT and thus containers can contain other containers. Thus we can have a table of priority queues.

Note 2:
All provided containers are **dynamically sized**. This means that wherever an array is used and the number of elements grows too large, the array will be automatically extended. In particular, with dictionaries, it is not necessary to estimate the number of entries when the dictionary is created. Only in the case of TABLES must a size be specified at creation; but even there, it is only an initial size. The table will be extended as required.

Pointers

- Source Code: containers.sim

- Return to Simula

Common operations

Although one thinks of various types of containers such as queues or stacks in terms of the operations that are characteristic to each (enqueue or push) there are in fact many operations which are common to all types of containers. These fall in the following categories:

- **Display** (i.e. `display`)
- **Enquiry** (i.e. `size`)
- **Iteration** over all member elements (i.e. `for_each`)
- **Assignment** whereby all elements in one container are inserted into another (i.e. `copies`)
- **Generic insertion and extraction** (i.e. `insert` and `pop`)
- **Other Operations defined for ELEMENTS** such as `into`

Containers also have two boolean attributes, NUMERIC & ASCENDING, which can be set by a user to implement various ordering disciplines in containers where order is important, for example: priority_queue and tables.

Index:

Class ELEMENT

This is the common root for all the classes. User classes for objects that will be put into containers must be defined as sub-classes of **Element**. Here is the list of the methods defined for Elements.

key:
    Returns a representative and unique value for the element
type:
    Returns the class of the element as a text
display:
    Prints a representative text for the element, usually the KEY
toString
    Returns a representative text for the element, usually the KEY
into(C):
    Inserts the element into the container C

KEY is the most important attribute. It serves as a representative label for output operations like DISPLAY. It also serves to distinguish entries into a dictionary. **Key, type, display** and **toString** are all defined to be virtual procedures; that is to say, they can be redefined in sub-classes.

Actually, KEY is the only procedure which needs to be redefined in a sub-class; all other procedures have reasonable default implementations which are based on the KEY.

Classes for Simple types

To allow simple values to be put into containers, the following wrapper classes were added in February 1998. Note: once an object has been created, its value (I,R,T or C) can be accessed but cannot be changed

```
        element class Int_object  (I); integer I;...
        element class Real_object (R); integer R;...
        element class Text_object (T); integer T;...
        element class Char_object (C); integer C;...
```

```
Example of use:

    ref(queue) Q;
    for i := 1,2,5,77,13 do
            new Int_object(i). into(Q);
```

Element Class CONTAINER

This class is the common root for all containers and it contains the declarations of attributes and methods common to all containers.

In what follows: E denotes an ELEMENT and C denotes a CONTAINER.

Display Operations

Since Containers are also Elements, element operations are also available to containers. The most useful ones are:

C.display:
: Does a pretty print of all elements in container C, (recursively if some elements are containers)

C.toString
: Returns a text with the keys of the first (30) elements in C

Enquiry operations:

C.size:
: Gives the number of elements in C

C.empty:
: Tests if container C is empty

C.first:
: Returns the *first* element; the one that would be removed by POP

Adding, removing member elements:

C.insert(E):
: Adds E to C

E.into(C):
: Same as C.insert(E)

C.into(C2):
: Also works for containers... will insert the container C as an **element** of container C2. This is different from putting the elements of C into C2 (see the assignment operators below).

C.remove(E):
: Removes E from C

C.pop:
: Removes the *first* element from C and returns a pointer to it

C.clear:
: Removes all elements from the container C

Copy and assignment operators

C.copy:
: Gives copy of container C

C.copies(C2):
: ASSIGNMENT "operator". Makes C contain the same elements as C2

C.gains(C2):
: All elements of C2 are removed from C2 and added to C. By default, we use POP to remove the elements.

Iteration

C.for_each(P):
: For each element E in C, calls P(E)

C.elements:
: Returns an ITERATOR object much like a JAVA `enumeration`. An Iterator is a 'list' all the elements in the 'container' when `elements` was called. These elements can be processed via 3 operations: MORE, NEXTELEMENT and RESET.
  - IT.nextElement => next element in enumeration
  - IT.more => TRUE if a next element exists
  - IT.reset : repositions the iterator on the first element.

Here is an example of use of an iterator:

```
ref(iterator) Loop;
ref(container) C;
ref(element) E;

    Loop :- C.elements;
    while Loop.more do outtext(Loop.nextElement.key);
    Loop.reset;
    while Loop.more do
    begin
        E :- Loop.nextElement;
        if p(E) then C.remove(E);
    end;
```

Other Element Operations

Other Element operations are also available in addition to *display* and *toString*.

C.type:
: Returns the CLASS of the container as a text (Used in pretty-printing)

C.key:
: By default, this returns NOTEXT

Accessing member elements:

In containers where ORDER has a meaning, priority_queues and Tables, the following operation is provided.

C.get(nth):
: Returns a pointer to the Nth element in C. In tables, this operator is complemented by the operation C.set(nth,E) which replaces the Nth element in C by E

Order amongst elements:

The order of elements (where applicable) depends on a virtual function, PRECEDES, which calls upon another *Default_Precedence* function which compares elements according to their KEYS and two container attributes: NUMERIC & ASCENDING. These can be set by the user after creation (and before inserting elements) to implement simple variants. By default:

```
        ASCENDING = true

        NUMERIC = false for all containers EXCEPT priority_queues
                = true for Priority_queues
```

NUMERIC means that we use GETREAL to extract the numeric values of KEYS and determine order; otherwise the alphabetical order is used. Here is how, the following 3 keys: '100' '22' '33', would be ordered:

```
   Ascending Numeric
           T        T     :  '22',  '33', '100'
           T        F     :  '100', '22', '33' .... because '1' < '2'

           F        T     :  '100', '33', '22'
           F        F     :  '33',  '22', '100'
```

Other ordering disciplines

To implement ordering schemes based on attributes other than the KEY, PRECEDES can be redefined in the container. For example, given the following class:

```
    element class student( Nom, Prenom, NAS, year);
        text Nom, Prenom, NAS;
        integer year;
    begin
        Text procedure Key; Key :- NAS;
        Text procedure toString;
            toString :- "Student: " & Nom & ", " &
                         Prenom & " (" & NAS & ") " & int_as_text(year) ;
    end;
```

We could define the following structures (just for students) to order them according to year or by name;

```
    priority_queue class Year_Sorter;
    begin
        Boolean procedure Precedes(E1,E2);  Ref(Element) E1,E2;
            Precedes := E1 qua student.year < E2 qua student .year;
        end;

        priority_queue class Name_Sorter;
    begin
        Boolean procedure Precedes(E1,E2);  Ref(Element) E1,E2;
            Precedes := E1 qua student.Nom < E2 qua student. Nom;
    end;
```

Container Class QUEUE

Queues operate in a FIFO fashion: the elements are removed in the same order as they were put in. This is the way that the generic operators, **insert** and **pop**, operate. QUEUE also defines some synonyms with traditional names:

Q.length = Q.size

Q.enqueue(E) = Q.insert(E)
Puts E as last element of Queue Q
Q.dequeue = Q.pop
Removes and returns first element of Q

By default, a queue is implemented as a one way list with a pointer on the last element to speed-up insertion.

Creation

```
        ref(queue) Q;
        Q :- new Queue;
```

Container Class SEQUENCE

Sequence is a container that was kept for compatibility with Kirkerud's package as used in WindowTools. A sequence operates as a Queue but some synonyms have been added for the operations.

S.append(E) = S.insert(E)
: Puts E as last element of Sequence S

S.put(E) = S.insert(E)
: Puts E as last element of Sequence S

Creation

```
        ref(queue) S;
        Q :- new Sequence;
```

Queue Class PRIORITY_QUEUE

A Priority_queue has the same operations as a queue but the element removed via POP (or dequeue) depends on the ordering relationship maintained by the container. POP removes the FIRST element. By default this is the one with the smallest numeric value of Key.

The default implementation is based on an ordered one-way list. To find where to insert new elements, the list is scanned from the front.

Note:
: The operation *Get(nth)* is provided for objects of the class priority_queue (but not necessarily for sub_classes like the HEAP). Get(nth) accesses the elements in the order that they would be removed by successive calls of POP.

Creation

The code below creates a priority_queue where the elements are kept in descending alphabetical order.

```
        ref(priority_queue) Q;
        Q :- new Priority_Queue;
        Q.ascending := false;
        Q.numeric   := false;
```

Priority_queue Class HEAP

The Heap is a efficient, O(log N), kind of priority_queue based on a partially ordered array of elements. The array is dynamic and there is no need to specify a size.

Note:

Because a HEAP is only *partially* ordered, the operation **Get(nth)** is **not** provided.

Creation

The code below creates a efficient priority_queue where the elements are kept in ascending alphabetical order.

```
ref(priority_queue) Q;
Q :- new HEAP;
Q.numeric   := false;
```

---

container Class STACK

Stacks operate in a LIFO fashion: the elements are removed in the **reverse** order as they were put in. The traditional operations are provided:

S.push(E) = S.insert(E)
: Adds an element to the front/top of the stack

S.pop
: Removes and returns first element of S

S.top = S.first
: Returns the same element that POP would but does not remove it

Note:
: The operation *Get(nth)* is provided for STACKS. Get(nth) accesses the elements in the order that they would be removed by successive calls of POP.

Creation

```
ref(stack) S;
Q :- new stack;
```

By default, a stack is implemented as a one way list and elements are pushed onto the front.

---

container Class DICTIONARY

A **Dictionary** provides fast look-up of elements according to their Keys. Furthermore, only one entry is kept for each Key. Insertion of an element with the same KEY as one already in the dictionary replaces the old element by the new one. Typical operations include adding new elements and finding (and accessing) an element with a given KEY.

Operations

D.insert(E)
: Element E is added to the dictionary or if an element with an equal KEY exists, E replaces the old element.

D.contains(E) ==> Boolean
: Returns TRUE if E is in the dictionary

D.find(Key) ==> Element
: Returns a reference to a dictionary entry with the given Key (if it exists), or NONE otherwise.

D.find_or_insert(Key, NewElement)
: Like FIND but if there is no entry with the given Key, then 'NewElement' is evaluated (called by NAME !) and inserted... and its value returned.

D.remove_Key(Key)

Removes and returns the dictionary entry with the given Key (if it exists); NOP otherwise. This operation is in addition to the standard container operation, "D.remove(Element)".

D.pop

Removes and returns an element of D (Not too useful !!!).

Note:

Certain operations use a KEY as a parameter whereas others require an ELEMENT.

Creation

```
        ref(Dictionary) Dico;
        Dico :- new Dictionary;
```

By default, a dictionary is implemented as a hash-table of one way lists.

---

dictionary Class SIMPLE_DICTIONARY

The Simple_dictionary is a dictionary implemented with an ordered list instead of a HASH-Table. It has exactly the same operations as a DICTIONARY.

Operations

- D.insert(E)
- D.contains(E) ==> Boolean
- D.find(Key) ==> Element
- D.find_or_insert(Key, NewElement)
- D.remove_Key(Key)
- D.pop

For more details, see Dictionary.

Creation

```
        ref(Dictionary) Dico;
        Dico :- new Simple_dictionary;
```

---

container Class TABLE

A **table** is basically an array where the components are retrieved via *indexing* according to their position in the table. This kind of access is provided by two procedures: GET & SET. In addition to this, we provide operations to FIND the position of an element, SORT the elements according to their KEYS or ORDER the elements according to a user provided *Precedence* function. Operations are also provided to remove all elements in a given range of positions or to make space for new elements in the middle of the table by shifting the others.

Note that if adding new elements or inserting space would cause elements to overflow the table, a new bigger array is allocated and the elements copied to it.

Operations

T.get(pos)

Returns the element at position POS in the Table if it exists; and NONE otherwise.

T.set(pos,Element)

Inserts a reference to the element at the specified POSition in the Table. Allowable values of POS are between 1 and T.size+1; that is, SET can be used to append a new element at the end of the table. This allows a table to be filled sequentially:

```
for i := 1 step 1 until <max> do
     T.put(i, <some element> )
```

T.insert(E) = T.set(T.size+1, Element)
: Element E is appended to the end of the table; .

T.insert_places(pos,n) ==> self
: Shifts elements[pos..size] up by N places

T.remove_entries(pos,n) ==> self
: Removes N consecutive entries starting at POS.

T.Sort
: Sorts the elements according to increasing Key values.

T.Order(Precedence_function)
: Sorts the elements according to a function provided by the user. This function must have the following specification:

```
Boolean procedure <proc_name> (E1,E2); ref(Element) E1,E2;
```

and return **true** if the first Element comes *before* the second.

T.search(Key) ==> integer *position*
: Returns the index of a dictionary entry with the given Key (if it exists), or **Zero** otherwise.

T.BinarySearch(Key) ==> integer *position*
: Like Search, this returns the index of the desired entry (if it exists). The procedure assumes the elements have been sorted and uses the O(log n) binary search. The algorithm assumes that the element have been **sorted** are in order of increasing KEY values. If the elements are not sorted or have been ORDERED according a user defined precedence function, then BinarySearch will return a random element.

T.pop
: Returns the **first** element (at position 1) and removes it via remove_entries(1,1). { Our Tables are not meant to be used as Stacks ! }.

Creation

Tables are the only containers where you must provide an initial estimate for the size of the implementation array. Note that this is different from the number of elements in the table (the **table** size) which is used for displaying, sorting and search. Remenber also that tables grow according to need.

```
ref(Table) T;
T :- new Table(10);
outint(T.size, 5);  ! would print 0 (zero) because T.size refers to the
                      number of elements in T; not the size of the
                      implementation array  ;
```

By default, a table is implemented as a pointer to an object containing an array of references to Elements.

Design Objectives

This package follows design objectives implicit in Kirkerud's design:

- An element may be in any number of containers simultaneously. This is not the case in the SIMSET lists of common base SIMULA.
- All containers have iteration/mapping functions to go through each element in the container.

Further design objectives were:

- To design the hierarchy of classes according to the functionality and not according to the implementation: TYPE inheritance versus CODE sharing. This lead to having parallel hierarchies: one for the containers and the other for implementation data structures.
- All containers "methods" return a VALUE. Operations (like INSERT) which have no obvious value, return a reference to the container and thus may be cascaded.
- Try to use a uniform naming convention for all containers. Inspiration from Standish, "Data structures, Algorithms and Data Structures" and Aho et al., "Data structures and algorithms"
- To ease interchange with SETTOOLS, CONTAINERS contains synonym procedure declarations (as far as applicable).

Adding new containers

CONTAINERS provides the interfaces for commonly used data abstractions. The package also provides concrete implementations based mainly on one way lists and hash tables. Initially, to illustrate design possibilites, we had provided multiple implementations: for example, a STACK based on a LIST (as it is now) and one based on an array. Since this multiplicity made maintenance more difficult, we dropped most of the extra implementations.

A user can extend the package by defining containers with new interfaces. He could also implement existing container classes in different ways. However, we expect that the most common reason to define new containers will be to implement a variety of ordering disciplines.

To facilitate this, ordering within containers is based on a `virtual` function: **precedes** which calls upon the *Default_Precedence*. A new order can be obtained by defining a sub-class which redefines **precedes** (See section on ordering).

Implementation details

Containers makes heavy use of a simple one-way list structure based on that of Lisp. Following LISP tradition, list members are linked by CONS cells with 2 pointers: one towards the element and the other pointing the next CONS. This allows members to be in several lists at once.

```
class CONS ( elem, next);
        ref(element) elem;
        ref(cons)    next;
begin ....end;
```

We complement these lists with HEAD_CONS objects which operate as list HEADs. These List_heads maintain pointers to both ends to allow fast insertion at either ends. We also keep track of the number of elements in the list.

```
class HEAD_CONS;
begin
    integer numElements;
    ref(cons) head,tail;
end;
```

These one-way lists are used directly for the default Queue, Stack and Simple_dictionary containers. The Dictionary uses hashing with an array of these lists.

Pointers

- Source Code: containers.sim
- Return to Simula