

Containers V 1

Table of Contents

1 Containers – Container items for Tcl programs.....	1
1.1 SYNOPSIS.....	1
1.2 DESCRIPTION.....	1
1.3 CONTAINER TYPES.....	2
1.4 ARRAY VARIABLES.....	2
2 Bag – Bag container items for Tcl programs.....	3
2.1 SYNOPSIS.....	3
2.2 DESCRIPTION.....	3
3 Queue – Sequential Queue Container for Tcl programs.....	5
3.1 SYNOPSIS.....	5
3.2 DESCRIPTION.....	5
4 Stack – Stack Container for Tcl programs.....	6
4.1 SYNOPSIS.....	6
4.2 DESCRIPTION.....	6
5 Hash – Hash table container items for Tcl programs.....	7
5.1 SYNOPSIS.....	7
5.2 DESCRIPTION.....	7
6 Tree – Tree Container for Tcl programs.....	9
6.1 SYNOPSIS.....	9
6.2 DESCRIPTION.....	9
7 PQueue – Priority Queue Container for Tcl programs.....	12
7.1 SYNOPSIS.....	12
7.2 DESCRIPTION.....	12
8 RQueue – Random Queue Container for Tcl programs.....	14
8.1 SYNOPSIS.....	14
8.2 DESCRIPTION.....	14
9 Struct – Structure Container for Tcl programs.....	15
9.1 SYNOPSIS.....	15
9.2 DESCRIPTION.....	15
9.3 TEMPLATES	15
9.4 INSTANCES.....	16
9.5 DATA ACCESS.....	16
9.6 DATA MODIFICATION.....	17
9.7 INHERITANCE.....	17
9.8 EMBEDDED STRUCTURES.....	18
9.9 ASSIGNMENT.....	19

1 Containers – Container items for Tcl programs

1.1 SYNOPSIS

`bag stack queue tree pqueue rqueue struct`

1.2 DESCRIPTION

This package is a dynamically loaded extension to the Tcl language that implements a few container objects that can be useful for processing lists of things. The containers are implemented as containers of Tcl objects, and therefore can hold any type of item that can be represented as a native Tcl object. This includes lists, lists of lists, arrays, blocks of binary data, and strings.

When a container is created using this package, it creates an associated Tcl array that is maintained consistent as to contents with the contents of the container. The usual Tcl array syntax can also be used to access the contents of the container. The array will have indices that match those of the relevant container. For example, a *hash* will have an associated variable with string indices that match the hash keys, while a *bag* container will have indices that are numerical keys that represent the order in which items entered the container.

When an instance of a container is created, it is represented by a Tcl command that is used to access the functions of the container. Each of the containers provides a set of functions that are appropriate to the container. For example, the *stack* container will process the functions *push* and *pop*. Similarly the *queue* container will process the functions *insert* and *remove*. Every container supports a set of common functions and whatever container specific functions that are meaningful for the specific container type.

All containers will process the following functions:

<code>empty</code>	Empty the container
<code>destroy</code>	Destroy the container
<code>count</code>	Get the number of items in the container
<code>elements</code>	Make a list of the element values in the container
<code>keys</code>	Make a list of the element keys in the container
<code>list</code>	Make a list of the elements and their keys.
<code>status</code>	Get or set the container status
<code>trace</code>	Get or set the variable trace state

The *count* function will return the number of items currently in the container.

The *destroy* function will delete the container command from the interpreter and remove the equivalent Tcl array variable, if any.

The *empty* function will remove all items from the container.

The *elements* function will return a Tcl list that contains all of the element values of the items in the container.

The *keys* function will return a Tcl list that contains all of the indices of the items in the container. Container

indices can be either numerical values or string keys, depending on the type of container.

The *list* function always returns a proper Tcl list that includes all of the items in the container. The format of the list is a list of lists, one sub-list for each of the items in the container. The first element of the sub-list is the key to the item, and the second element in the list is the item value.

The *status* function returns the value *clean* or *dirty* depending on the update status of the container. Whenever the container is modified, the status is set to *dirty*. The status function can be used to reset the container status to *clean*.

1.3 CONTAINER TYPES

The current package implements the following containers:

bag	A simple unordered indexed list of items
stack	A stack container
queue	A simple FIFO queue
tree	A sorted binary tree container
pqueue	A priority queue container
rqueue	A random queue container
hash	A hash table
struct	A structure

Each container command accepts options that allow for the configuration of features of the particular container. The result of the container command, if no error has been detected, is a new Tcl command that represents the container itself.

Each container object command implements the list of common and object specific functions appropriate to the container. The documentation entries for the specific containers describe the available functions.

1.4 ARRAY VARIABLES

When a container command is used to create an instance of a container, the resulting return value will be a token that is the name of the Tcl command that was created to represent the container. This name is also the name of a Tcl array variable, created in the global scope, that can be used to access the contents of the container using the usual Tcl array syntax.

The array variable is automatically created and initialized whenever an array reference is made using the container token as the array name. The package initialization creates the Tcl command *Containers::UpdateArray* that is called to recreate the array each time the container contents are modified. By using the *status* function of the container, you can determine if the containers has been modified since the last array update, and then call *Containers::UpdateArray* as appropriate.

2 Bag – Bag container items for Tcl programs

2.1 SYNOPSIS

```
bag ?-size? ?size? ?-unique? ?boolean? ?-circular? ?boolean?
```

2.2 DESCRIPTION

The *bag* command is part of the *Containers* extension to the Tcl programming language. The *bag* container implements a simple unordered indexed list of items. The *bag* can be created with a command of the form:

```
set l [bag ?-size? ?size? ?-unique? ?boolean? ?-circular? ?boolean?]
```

where the optional value *size* is the number of initial elements in the list to allocate. By default, *size* is *100*. Regardless of the initial size, the list will grow to handle the number of items that are added. The amount of memory used by the list is not reduced by the simple deletion of items from the list.

The *-unique* option can be used to indicate that only items whose string representation are unique can be added to the bag. By default, the items need not be unique.

The *-circular* option can be used to fix the number of items in the *bag* to the value specified for *-size*. Adding more than this number of items will cause the *bag* to act like a circular buffer, with the oldest item in the list deleted to make room for the newest item.

The *bag* command will create a container command that represents the *bag* object. The general format of the container command is:

```
$l function ?-option? ?option? ... ?item? ...
```

where the *function* must be one of the following:

add	Add one or more items to the list
delete	Delete one or more items from the list
get	Get an item from the list
destroy	Destroy the list object
empty	Empty the list object
count	Get the number of items in the list
list	Make a Tcl list of the items in the list

The following examples show how to use the command functions on a list:

```
$l add string1 string2 { string3 string4 } string 5
```

will add 4 items to the list, the third item being a list that contains 2 strings. In the list, the items are identified by ordinal, so the command

```
$l get 2
```

would return the result { string3 string4 } which is the third element of the list. Similarly, the command:

```
$l delete 2
```

would remove the third element from the list, and the resulting contents of the container could then be retrieved using the command:

```
$l list
```

which would return the following Tcl list:

```
{ string1 string2 string5 }
```

Bag elements are numbered from 0 through [expr [\$l count] - 1]. The *destroy* function will delete the container object command from the Tcl interpreter and release any space being used by the container.

3 Queue – Sequential Queue Container for Tcl programs

3.1 SYNOPSIS

```
queue ?-size? ?size? ?-unique? ?boolean? ?-fixed? ?boolean?
```

3.2 DESCRIPTION

The *queue* command will create a container that behaves as a FIFO type queue. The *queue* container can be created with a command of the form:

```
queue ?-size? ?size? ?-unique? ?boolean? ?-fixed? ?boolean?
```

where the value specified for the *-size* option is the initial number of entries in the queue to allocate, the value for the *-unique* option specifies whether the string representation of items in the queue must be unique, and the value specified for the *-fixed* option specifies whether the queue is limited to a fixed size.

Regardless of the value specified for the *-size* option, for queues that are not of fixed length, the size of the queue will be expanded to accommodate new entries.

By default, the value of the *-unique* option is *false*.

By default, the value of the *-fixed* option is *false*. If the value is set to true, then once the number of entries in the queue reaches the value specified for the *-size* option, all new additions to the queue will cause the new arrival to replace the oldest item in the queue. This makes the *queue* container act like a pipeline.

In addition to the common commands of *count*, *destroy*, *empty* and *list*, the *queue* container command will also process the following functions:

insert	Add one or more items to the queue
remove	Remove an item from the queue

The following example creates a *queue*, puts some items into the *queue*, then removes the items from the *queue*:

```
set q [queue -size 5]
$q insert string1 string2 string3 string4 string5
while { [$q count] != 0 } {
    puts [$q remove]
}
```

The result of this script is the printing of the elements of the *queue* in the order in which they entered the queue.

4 Stack – Stack Container for Tcl programs

4.1 SYNOPSIS

```
stack ?-size? ?size? ?-unique? ?boolean?
```

4.2 DESCRIPTION

The *stack* command will create a container object that implements a simple stack. The *stack* container can be created with a command of the form:

```
stack ?-size? ?size? ?-unique? ?boolean?
```

where the value specified for the *-size* option is the initial number of entries to allocate for the stack and the value specified for the *-unique* determines whether the string representations of the items placed on the stack must be unique.

Regardless of the value specified for the *-size* option, the stack will grow to accommodate new entries. By default, the value of the *-unique* option is *false*.

In addition to the common commands of *count*, *destroy*, *empty* and *list*, the *stack* container will process the following functions:

push	Push one or more items into the stack
pop	Pop one item from the stack
peek	Peek at the next item to be popped

The following example creates a *stack* and inserts a number of items into the stack:

```
set s [stack -size 20]
$s push string1 string2 string3 string4
while { [$s count] != 0 } {
    puts [$s pop]
}
```

The result of this little script is the printing of the list of *strings* in reverse order, as the *stack* is a LIFO implementation of a [queue](#).

5

Hash – Hash table container items for Tcl programs

5.1 SYNOPSIS

```
hash ?key1? ?value1? ... ?keyn? ?valuen?
```

5.2 DESCRIPTION

The *hash* command is part of the Containers extension to the Tcl programming language. The *hash* container implements a simple hash table container whose elements are identified by strings that are keys into the table. The hash table can be created with a command of the form:

```
set l [hash ?key1? ?value1? ... ?keyn? ?valuen?]
```

The *hash* command will create a container command that represents the hash table object. The key and value pairs, if specified, are used to initialize the contents of the container. An empty container is created if there are no parameters specified.

The general format of the container command is:

```
$l function ?-option? ?option? ... ?item? ...
```

where the function must be one of the following:

add	Add one or more items to the list
delete	Delete one or more items from the list
get	Get an item from the list
destroy	Destroy the list object
empty	Empty the list object
count	Get the number of items in the list
list	Make a Tcl list of the items in the list
keys	Make a Tcl list of the keys in the list
values	Make a Tcl list of the values in the list

The following examples show how to use the command functions on a list:

```
$l add key1 string1 key2 string2
```

will add 2 items to the *hash* list with keys *key1* and *key2*. To recover the items, a command of the form:

```
$l get key1 key2
```

could be used. The result of this command would be a Tcl list with the elements *string1* and *string2* as its members.

To delete items from the *hash* table, the command:

```
$l delete key1 ... keyn
```

could be used, where the *key* parameters are strings that match the keys of elements in the list.

6 Tree – Tree Container for Tcl programs

6.1 SYNOPSIS

```
tree ?-size? ?size? ?-reverse? ?boolean?  
      ?-unique? ?boolean? ?-nocase? ?boolean?  
      ?-alphabet? ?string? ?-first? ?value?  
      ?-range? ?count?
```

6.2 DESCRIPTION

The `tree` command will create a container that implements a binary tree data structure that will store items according to their sorting order. The `tree` command has the following format:

```
tree ?-size? ?size? ?-reverse? ?boolean?  
      ?-unique? ?boolean? ?-nocase? ?boolean?  
      ?-alphabet? ?string? ?-first? ?value?  
      ?-range? ?count?
```

where *size* sets the initial number of item entries to allocate, the value of the *-reverse* option establishes the sorting order to use, and the value of the *-unique* option determines whether duplicate items can be entered into the tree. The *-nocase* option is used to specify whether case is to be ignored when comparing items for sorting purposes. The *-alphabet* option can be used to specify the order of characters in the alphabetic order used to sort the items. The *-first* and *-range* options can be used to select the characters in the input strings that are used for sorting the items.

By default, the value of *-reverse* is *false*, and items are stored in ascending order according to their collating sequence value of the string representation of the Tcl object being stored. If *-reverse* is set to *true*, then the reverse collating sequence order is used.

By default, the value of the *-unique* option is *false* and the tree can accept multiple copies of items whose string representation is the same. By setting *-unique* to *true*, then duplicate entries will not be added to the tree.

By default, the value of the *-nocase* option is *false*. If this option is set to *true*, then the items in the containers are sorted without regard to the case of the characters in the item strings.

By default, the value of the *-alphabet* option is NULL. This causes the items in the container to be sorted according to the collating sequence for the character set being used that is the default for the computer being used. This would mean the ASCII collating sequence on many machines.

If a string of characters is supplied with the *-alphabet* option, then the order of the characters in the string is used to sort the items in the container. The supplied string need not contain the entire list of possible characters. If the supplied string is incomplete, then for items that contain characters in string, the sorting order will be based on the supplied list of characters in string, otherwise, the sorting order is based on the default collating sequence.

The string supplied with the *-alphabet* option has a syntax for the specification of ranges of characters based

on their numeric value in the default collating sequence. The general form of the specification is:

```
-alphabet cc(c-c,c,nnn,nnn)(nnn-nnn)
```

where the *c* are single characters and the *nnn* are numbers of more than 1 digit that represent a character by its collating sequence ordinal. The *–* symbols between two characters or numbers indicate a range of characters in the collating sequence from the character on the left hand side of the *–* up to and including the character on the right hand side of the *–* symbol. The *,* is used to separate single characters or ranges in a sequence of specifications that is contained within the parentheses. The backslash character can be used to escape characters such as (and), or the backslash itself. Here is an example that creates an alphabet that consists of the characters @+* and the numbers from 0 through 9:

```
-alphabet "@+*(0-9)"
```

This example would sort all items using the precedence @+*0123456789 and then, for strings not containing any of these characters, the default collating sequence.

The *–first* and *–range* options can be used to define which characters in the item strings are used for sorting. By default, the value of *–first* is 0, and the value of *–range* is the lesser of the lengths of the two items being compared during a sorting operation. By adjusting the value of *–first* and setting a value for *–range*, the comparison operation can be applied to an part of the item strings.

In addition to the common functions of *count*, *destroy*, *empty*, and *list*, the tree container will process the following functions:

add	Add one or more items to the tree
compact	Compact a tree
delete	Delete one or more items from the tree
get	Get one or more items from the tree
next	Get the first item from the tree

The *next* function returns the first item in the tree and marks it as deleted.

The *get* function can accept a list of indices of the items currently in the tree and return the items that correspond to those indices. Similarly the *delete* command will mark as deleted all of the items specified in the list of indices in the command.

The *compact* command can be used to recover the memory occupied by deleted tree nodes. When items are deleted from a tree the memory space occupied by the actual item is released, but the tree node itself remains in existence but marked deleted. The *compact* command will clean up the tree by reorganizing the nodes and removing the deleted ones.

The following demonstrates how to make use of the *tree* container:

```
set t [tree -reverse true -unique true]
$t add string3 string1 string4 string2 string2
puts [$t list]
```

which will result in a listing of the items in the following order:

```
string4 string3 string2 string1
```

The *tree* container is handy for creating and maintaining a sorted list of items without having to re-sort the list each time a new item is added.

The following *tree* shows how to sort a collection of strings that have some additional attributes that indicate that promotion in list order is wanted.

```
set t [tree -nocase true -unique true -alphabet @+]
$t add +Sally @George @+Mary Peter +Irene @Eric +Diane Willy
puts [$t list]
```

will result in the list:

```
@+Mary @Eric @George +Diane +Irene +Sally Peter Willy
```

7 PQueue – Priority Queue Container for Tcl programs

7.1 SYNOPSIS

```
pqueue ?-size? ?size? ?-reverse? ?boolean?  
        ?-unique? ?boolean? ?-priority? ?type?  
        ?-nocase? ?boolean?
```

7.2 DESCRIPTION

The *pqueue* command creates a priority queue data structure based on the *tree* container described above. *PQueue* containers support the same command set as does the *tree* container. A priority queue is a waiting queue which contains items that can have a priority associated with them. The interesting characteristic of the priority queue is that the items are always maintained in order of their priority, with the highest or lowest priority item at the head of the queue.

A priority queue container can be created with a command of the form:

```
pqueue ?-size? ?size? ?-reverse? ?boolean?  
        ?-unique? ?boolean? ?-priority? ?type?  
        ?-nocase? ?boolean?
```

where *-size* specifies the initial number of slots to be allocated for the queue, *-reverse* specifies the sorting order of the priority values, the *-unique* value specifies whether the items must have unique string representations, and the *-priority* option indicates the type of key that is being used. The *-nocase* option can be used to indicate that case should not be used when sorting key values of the items. By default, for non-numeric keys, case is used for sorting.

Regardless of the value specified for the *-size* option, the queue will expand to accommodate additional entries.

By default, the queue entries are maintained in order of decreasing priority, so the highest priority item is always at the front of the queue. If the *-reverse* option is *true*, then the queue will be maintained in the order of increasing priority, with the highest priority item being at the back of the queue.

By default, the value of the *-unique* item is *false*. If the value of the *-unique* item is set to *true*, then the string representation of items added to the queue must be unique.

The *-priority* option can be used to specify the type of the key that is to be used. By default, the key type is *string*, and the keys are sorted by collating sequence. If the value of the *-priority* option is *numeric*, then the keys must be integer values, and the item priorities are sorted by numeric value. This distinction can be important, as for *string* keys, the standard sorting sequence of the keys:

1 2 10 4

is

1 10 2 4

which may not be the desired result.

The *pqueue* command will create a container command that represents the container that has the following format:

```
$pq function ...
```

where the functions supported include the common container functions of *count*, *destroy*, *empty* and *list*, as well as the following:

<code>add</code>	Add a new entry to the queue
<code>next</code>	Get the item at the head of the queue

The *add* command has the following format:

```
$pq add key1 item1 .... keyn itemn
```

where there may be any number of *key* and *item* pairs. The *items* themselves could be any type of Tcl object, while the *keys* are things that can be viewed as either strings or integer numbers.

The *next* command will retrieve the current head of the queue and remove it from the queue. It requires no parameters.

8 RQueue – Random Queue Container for Tcl programs

8.1 SYNOPSIS

```
rqueue ?-size? ?size? ?-unique? ?boolean? ?-reverse? ?boolean?
```

8.2 DESCRIPTION

The *rqueue* command creates a *tree* based container that inserts new items into the queue in a random order. The *rqueue* container supports the same function set as the *tree* and the *pqueue* container commands.

A random queue can be created with a command of the form:

```
rqueue ?-size? ?size? ?-reverse? ?boolean? ?-unique? ?boolean?
```

where *-size* specifies the initial number of slots to be allocated for the queue, *-reverse* specifies the sorting order of the priority values, the *-unique* value specifies whether the items must have unique string representations. By default, the value of the *-reverse* option is *false* and the value of the *-unique* option is *false*. The value of the *-size* option specifies the initial allocation size for the random queue. Regardless of the value specified, the random queue will automatically grow to accommodate new entries.

The format of the command to *add* items to a random queue is:

```
$r add item1 ... itemn
```

where the *items* are Tcl objects of some kind. The *items* are inserted into an ordered list in random order. The *items* can then be removed from the random queue using the *next* function. For example, the command:

```
while {[set item [$r next]] != "" } { puts $item }
```

will empty a random queue.

9 Struct – Structure Container for Tcl programs

9.1 SYNOPSIS

```
struct ?-function? ?name? ... ?name?
```

9.2 DESCRIPTION

The *struct* command creates a data structure container that holds named elements. The named elements can hold any type of Tcl object.

The format of the *struct* command is:

```
struct ?-function? ?name1? ?name2? ... ?namen?
```

where *function* may be either *-list* to generate a list of structures currently known or *-templates* to generate a list of templates currently known. The list of *name* parameters are the names of the elements in the structure or the names of existing templates that are to be used to initialize the names of the structure.

In the simple case where the *name* parameters are names of the structure elements, then the result of the command will be a template whose element names match the list of strings given as parameters to the *struct* command. If any of the *names* is the name of an existing template, then the element names of the template are added to the new structure.

The *struct* command returns a token that represents the structure template. The template itself does not hold any data, but provides a pattern for use in initializing the instances of the structure. The returned token is a template command.

9.3 TEMPLATES

Instances of structures are initialized by using the template command. Template commands have the following form:

```
template ?-function? ?value1? ... ?valuen?
```

where *template* is the name of a template command that was created using the *struct* command, *function* is one of the functions supported by template commands, and the *value* parameters are the Tcl objects that are to be used to initialize the instance of the structure.

All template commands support the following functions:

```
-elements    List the element names of the template
-destroy     Destroy the template
```

The *-destroy* function is the equivalent of renaming the template command to an empty string. The command is deleted from the Tcl interpreter and the template is no longer available for the creation of new instances.

The *-elements* function returns a Tcl list of the element names of the elements in the structure.

There must be either a *function* parameter or a list of *value* parameters that includes the same number of values as there are elements in the structure. *Value* items are assigned to structure elements in order of the element names as specified on the *struct* command that created the template.

For example, suppose the following command is used to create a structure template that is to hold data on individuals:

```
set person [struct Name Address Phone]
```

then an instance of the template for the structure could be initialized with the following command:

```
set george [$person "George Peabody" "2035 Wonder Way" "202-555-6053"]
```

Here the resulting token in *george* would be a Tcl command that can then be used to manipulate the data in this instance of the template that is represented by the command token in *person*.

9.4 INSTANCES

An instance of a structure is represented by a token that is a Tcl command of the form:

```
name ?-function? ?element? ?value?
```

where *name* is the name of the instance command as returned by a previous template command, *function* is one of the functions that are supported by instance commands, *element* is an element name of an element in the structure, and *value* is a new value for the named element.

The list of functions supported by the instance commands is:

<code>-elements</code>	List the elements of the structure
<code>-values</code>	List the values of the elements of the structure
<code>-list</code>	List the elements and their values for the structure
<code>-destroy</code>	Destroy this instance of the structure
<code>-assign</code>	Assign the values of one structure to another
<code>-append</code>	Append the values of one structure to another
<code>-isequal</code>	Check if 2 structures contain the same data

The *-destroy* function is the equivalent of renaming the instance command to an empty string. The instance is deleted and is no longer available in the interpreter.

The *-elements*, *-values* and *-list* functions all return Tcl lists containing information about the structure instance. Because of the implementation of the structure there is not necessarily a relationship between the order of the elements in the returned lists and the order of the elements in the structure template.

9.5 DATA ACCESS

Once an instance of a *struct* is initialized, the values of the elements in the structure can be accessed using the element names and the instance command of the structure. The instance command of the structure is the token returned by the template command that was used to initialize the structure.

Using the above example, the instance command for the structure instance that contains the information about George Peabody is in the Tcl variable *george*. To access the *Address* element, use the following command:

```
$george Address
```

The value returned from this command will be the string 2035 Wonder Way. Similarly, the other elements of the structure can be accessed by using their names.

9.6 DATA MODIFICATION

Data elements can be changed using the instance command in the following form:

```
token element value
```

where *token* is the instance command for the relevant instance of the structure, *element* is the name of the element to be modified, and *value* is any Tcl object that is to be used as the element value.

For example, suppose George Peabody has a second residence. The construct:

```
$george Address { [$george Address] "300 SummerDays Lane" }
```

would replace the current value of *Address* with a list of 2 addresses, the original one and the new address of 300 SummerDays Lane.

9.7 INHERITANCE

Existing *struct* templates can be used to construct new templates. If any of the parameters to the *struct* command are themselves tokens that identify an existing template, all of the element names of that existing template are added to the definition of the new structure.

For example, assume that the token stored in *person* is that of the template that was as defined above with the elements *Name*, *Address*, and *Phone*. A new template might be defined as follows:

```
set business [struct $person Business]
```

which would result in a new template that contains the 3 elements that were defined for the *person*, and a new element called *Business*. To initialize an instance of the new structure the command:

```
set george [$business "George Peabody" "2035 Wonder Way" "202-555-6023"
"202-555-6197"]
```

might be used. Now the token *george* will be able to deliver the additional value by using the element name *Business* as well as the

original 3 values.

Because the new template *business* is derived from the template in *person*, then an instance of the new structure could also be initialized as follows:

```
set george [$business $george "202-555-6197"]
```

This last command will take the current values of the elements of *george* and assign them to the corresponding elements in the new structure instance, and then add in the final element.

9.8 EMBEDDED STRUCTURES

Existing structure templates can be used to create structures of structures. The format of the command used to create a structure that contains embedded, named structures as its elements is:

```
set list [struct [list name1 template1] [list name2 template2] ... ]
```

where the *name* and *template* values are respectively the element name and the defining template respectively. Note that the parameters to the *struct* command are proper Tcl lists of exactly 2 elements, the first of which is the element name, and the second of which is an existing template.

As an example, the following command creates a structure whose elements are structures that describe 3 individuals:

```
set l [struct [list George $person] [list Mary $person] [list alice $person]]
```

This structure will have elements that are of the form:

```
George.Name, George.Address George.Phone
Mary.Name, Mary.Address, Mary.Phone
Alice.Name, Alice.Address Alice.Phone
```

Initialization of the compound structure can be carried out using a syntax that is of the form:

```
$l -assign [list George $george] ...
```

where, as with the template definition, the use of proper Tcl lists as the items in the initialization list is required. The elements of the Tcl lists that are initialization items are the element name of the sub-structure, and the token that represents an instance of a structure that has elements whose names match those of the substructure. In the above example, *george* contains a token that represents an instance of the template described by the token in *person*.

Where the elements of the initialization list are not proper Tcl lists, they are assumed to be values. The list of values is used to initialize the elements of the compound structure exactly as they occur. For the example structure being used here, the first 3 initialization elements would be used to initialize the 3 elements of the substructure named *George*, the second 3 initialization elements would apply to the elements of *Mary* and the

last 3 would apply to *Alice*.

9.9 ASSIGNMENT

Given 2 structures that contain values identified by element names that are common to both, a command of the form:

```
$george –assign $home $office
```

will assign to *george* all of the values in *home* and *office* that are identified in either of these structures by element names that are also in *george*.

The assignment operator will replace any values in *george* by those in the other structures. If you want to append common element values, use a command of the form:

```
$george –append $home $office
```

which will cause the elements in common with *george* found in *home* and *office* to be concatenated into a list and used to replace the original value in *george*. Note that the final list of common element values will also contain the original value in *george*.

A final operator is the equivalence operator, which returns the value of 1 if the string representation of all of the elements in one structure match the string representation of all of the elements in another structure. For example, the command:

```
$george –isequal $george
```

will always return 1. If there is a mismatch, the value returned is 0.