# record(n) 1.2.1 record "Tcl Data Structures"

## NAME

record - Define and create records (similar to 'C' structures)

## SYNOPSIS

package require **Tcl 8.2**

package require **struct ?1.2.1?**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **record define** *recordName* *recordMembers* ?*instanceName1 instanceName2 ...*? | | **record show** *record* | | **record show** *instances* *recordName* | | **record show** *members* *recordName* | | **record show** *values* *instanceName* | | **record exists** *record* *recordName* | | **record exists** *instance* *instanceName* | | **record delete** *record* *recordName* | | **record delete** *instance* *instanceName* | | *recordName* ***instanceName|#auto*** ?*-member1 value1 -member2 value2 ...*? | | *instanceName* **cget** ?*-member1 -member2 ...*? | | *instanceName* **configure** ?*-member1 value1 -member2 value2 ...*? | |

## DESCRIPTION

The **::struct::record** package provides a mechanism to group variables together as one data structure, similar to a 'C' structure. The members of a record can be variables or other records. However, a record can not contain circular record, i.e. records that contain the same record as a member.

This package was structured so that it is very similar to how Tk objects work. Each record definition creates a record object that encompasses that definition. Subsequently, that record object can create instances of that record. These instances can then be manipulated with the **cget** and **configure** methods.

The package only contains one top level command, but several sub commands (see below). It also obeys the namespace in which the record was define, hence the objects returned are fully qualified.

**record define** *recordName* *recordMembers* ?*instanceName1 instanceName2 ...*? Defines a record. *recordName* is the name of the record, and is also used as an object command. This object command is used to create instances of the record definition. *recordMembers* are the members of the record that make up the record definition. These are variables and other record. If optional *instanceName* args are given, then an instance is generated after the definition is created for each *instanceName*.

**record show** *record* Returns a list of records that have been defined.

**record show** *instances* *recordName* Returns the instances that have been instantiated by *recordName*.

**record show** *members* *recordName* Returns the members that are defined for record *recordName*. It returns the same format as how the records were defined.

**record show** *values* *instanceName* Returns a list of values that are set for the instance *instanceName*. The output is a list of key/value pairs. If there are nested records, then the values of the nested records will itself be a list.

**record exists** *record* *recordName* Tests for the existence of a *record* with the name *recordName*.

**record exists** *instance* *instanceName* Tests for the existence of a *instance* with the name *instanceName*.

**record delete** *record* *recordName* Deletes *recordName*, and all instances of *recordName*. It will return an error if the record does not exist.

**record delete** *instance* *instanceName* Deletes *instance* with the name of *instanceName*. It will return an error if the instance does not exist.

## RECORD MEMBERS

Record members can either be variables, or other records, However, the same record can not be nested witin itself (circular). To define a nested record, you need to specify the **record** keyword, along the with name of the record, and the name of the instance of that nested record. For example, it would look like this:

|  |  |
| --- | --- |
|  | # this is the nested record record define mynestedrecord {  nest1  nest2 }  # This is the main record record define myrecord {  mem1  mem2  {record mynestedrecord mem3} } |

You can also assign default or initial values to the members of a record, by enclosing the member entry in braces:

|  |  |
| --- | --- |
|  | record define myrecord {  mem1  {mem2 5} } |

All instances created from this record definition, will initially have 5 as the value for *mem2*. If no default is given, then the value will be the empty string.

**Getting Values**

To get a value of a member, there are several ways to do this.

1. To get a member value, then use the instance built-in **cget** method:  
     
   *instanceName* **cget** -mem1
2. To get multiple member values, you can specify them all in one command:  
     
   *instanceName* **cget** -mem1 -mem2
3. To get a list of the key/value of all of the members, there are 3 ways:  
     
   - *instanceName* **cget**  
     
   - *instanceName* **configure**  
     
   - *instanceName*
4. To get a value of a nested member, then use the dot notation:  
     
   *instanceName* **cget** -mem3.nest1

**Setting Values**

To set a value of a member, there are several ways to do this.

1. To set a member value, then use the instance built-in **configure** method:  
     
   *instanceName* **configure** -mem1 val1
2. To set multiple member values, you can specify them all in one command:  
     
   *instanceName* **configure** -mem1 va1 -mem2 val2
3. To set a value of a nested member, then use the dot notation:  
     
   *instanceName* **configure** -mem3.nest1 value

**Alias access**

In the original implementation, access was done by using dot notation similar to how 'C' structures are accessed. However, there was a concensus to make the interface more Tcl like, which made sense. However, the original alias access still exists. It might prove to be helpful to some.

Basically, for every member of every instance, an alias is created. This alias is used to get and set values for that member. An example will illustrate the point, using the above defined records:

|  |  |
| --- | --- |
|  | # Create an instance first % myrecord inst1 ::inst1 % # To get a member of an instance, just use the  % # alias (it behaves like a Tcl command): % inst1.mem1 % % # To set a member via the alias, just include  % # a value (optionally the equal sign - syntactic sugar) % inst1.mem1 = 5 5 % inst1.mem1 5 % # For nested records, just continue with the  % # dot notation (note no equal sign) % inst1.mem3.nest1 10 10 % inst1.mem3.nest1 10 % # just the instance by itself gives all  % # member/values pairs for that instance % inst1 -mem1 5 -mem2 {} -mem3 {-nest1 10 -nest2 {}} % # and to get all members within the nested record % inst1.mem3 -nest1 10 -nest2 {} % |

## RECORD COMMAND

The following subcommands and corresponding arguments are available to any record command: *recordName* ***instanceName|#auto*** ?*-member1 value1 -member2 value2 ...*? Using the *recordName* object command that was created from the record definition, instances of the record definition can be created. Once a instance is created, then it inherits the members of the record definition, very similar to how objects work. During instance generation, an object command for the instance is created as well, using *instanceName*. This object command is used to access the data members of the instance. During the instantiation, values for that instance can be given, **but** all values must be given, and be given in key/value pairs. Nested records, need to be in list format.

Optionally, *#auto* can be used in place of *instanceName*. When #auto is used, then a instance name will automatically be generated, of the form recordName<integer>, where <integer> is a unique integer (starting at 0) that is generated.

## INSTANCE COMMAND

The following subcommands and corresponding arguments are available to any record instance command: *instanceName* **cget** ?*-member1 -member2 ...*? Each instance has the sub command **cget** associated with it. This is very similar to how Tk widget's cget command works. It queries the values of the member for that particular instance. If no arguments are given, then a key/value list is returned.

*instanceName* **configure** ?*-member1 value1 -member2 value2 ...*? Each instance has the sub command **configure** associated with it. This is very similar to how Tk widget's configure command works. It sets the values of the particular member for that particular instance. If no arguments are given, then a key/value list is returned.

## EXAMPLES

Two examples are provided to give an good illustration on how to use this package.

**Example 1**

Probably the most obvious example would be to hold contact information, such as addresses, phone numbers, comments, etc. Since a person can have multiple phone numbers, multiple email addresses, etc, we will use nested records to define these. So, the first thing we do is define the nested records:

|  |  |
| --- | --- |
|  | ## ## This is an interactive example, to see what is  ## returned by each command as well. ##  % namespace import ::struct::record::\*  % # define a nested record. Notice that country has default 'USA'. % record define locations {  street  street2  city  state  zipcode  {country USA}  phone } ::locations % # Define the main record. Notice that it uses the location record twice. % record define contacts {  first   middle   last   {record locations home}  {record locations work} } ::contacts % # Create an instance for the contacts record. % contacts cont1 ::cont1 % # Display some introspection values % record show records ::contacts ::locations % # % record show values cont1 -first {} -middle {} -last {} -home {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} -work {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} % # % record show instances contacts ::cont1 % # % cont1 config -first {} -middle {} -last {} -home {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} -work {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} % # % cont1 cget -first {} -middle {} -last {} -home {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} -work {-street {} -street2 {} -city {} -state {} -zipcode {} -country USA -phone {}} % # copy one record to another record % record define contacts2 [record show members contacts] ::contacts2 % record show members contacts2 first middle last {record locations home} {record locations work} % record show members contacts first middle last {record locations home} {record locations work} % |

**Example 1**

This next example just illustrates a simple linked list

|  |  |
| --- | --- |
|  | % # define a very simple record for linked list % record define llist {  value  next } ::llist % llist lstart ::lstart % lstart config -value 1 -next [llist #auto] % [lstart cget -next] config -value 2 -next [llist #auto] % [[lstart cget -next] cget -next] config -value 3 -next "end" % set next lstart lstart % while 1 { lappend values [$next cget -value] set next [$next cget -next] if {[string match "end" $next]} {break} } % puts "$values" 1 2 3 % # cleanup linked list % # We could just use delete record llist also % foreach I [record show instances llist] { record delete instance $I } % record show instances llist % |

## KEYWORDS

struct, record, data structures