Exploring Tcl Iteration Interfaces

By Phil Brooks

Presented at the 19th annual Tcl/Tk conference, Chicago Illinois November 2012

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

Abstract--- In Mentor Graphics' Calibre verification tool, Tcl is frequently used as a customer extension language - allowing customers to customize and drive the tool through various exposed interfaces. These interfaces are frequently used to access large collections of application data and provide a wide variety of mechanisms for iteration over that data. This paper will examine several interfaces that have been used for iteration over large C++ data structures along with the benefits and drawbacks of each method. Methods explored include Tcl lists, indexed array-like access, iterator object accessor (similar to C++ STL iterators), and specialized foreach style commands. Example stand alone implementations are provided and discussed from within the context of their original use in Calibre customer scripting interfaces. Ease of use and performance are considered.

1 Introduction

The simple task of iteration over each object in a container is one of the most common in programming. The task is so common that every programming language tends to develop common idioms for the form. Simple expression of the concept of iteration in a vernacular form aids readability and maintainability of code. In Tcl where the list is the most commonly used aggregate data structure, the foreach command is the standard for an iterative vernacular:

```
set test_list { a b c d }
foreach var $test_list {
    puts $var
}
```

When C based Tcl_Obj object interfaces that represent collections of underlying objects are being used, the foreach command itself is of little use since it works only with Tcl lists. So for iteration, either the Tcl object must convert its contents into Tcl list form, or another interface must be constructed for iteration over the object data sets. The remainder of this paper considers potential interfaces for this purpose.

2 Demo Environment

All of the demo interfaces used in the example program are providing access to the contents of a C++ array of doubles - or in C++ "std::vector<double>". A Tcl program is used to iterate over the contents and to accumulate a result which is returned to the C++ program. The context of the environment is that of a customized analysis routine that is called from the C++ application. The Tcl interface allows an end user to perform custom calculations without having access to the application C++ source code or having to manage a C compilation environment. The Tcl program has read only access to the C++ vector.

Since the Tcl routine is called directly from the C++ program, a record based user interface is provided so that the user can direct the application with the name of the Tcl script and the proc to call. In the Mentor Graphics Calibre environment, these Tcl calls are specified from the Standard Rulefile Verification Format (SVRF) language that makes up the bulk of the application's programming interface.

In this example program, a configuration file specifies the name of the Tcl script, a proc to call, and the iteration interface that is to be selected (from the 4 we are describing).

These fields are specified as simple text fields on a single line of the file:

```
<script_file> <called_proc> <interface>
```

For example:

```
list_user_script.tcl do_calculation list
```

describes list_user_script.tcl as the script file, calc_abmi as the Tcl proc, and the list generation interface as the interface to provide.

3 Loading the script file

After the config file is read, the script file itself is read and evaluated in the Tcl interpreter so that it can be called repeatedly as the application progresses through its data set. This is accomplished by first creating a Tcl_Obj that will contain the script:

```
Tcl_Obj* tcl_script = Tcl_NewObj();
Tcl_IncrRefCount( tcl_script );
```

(Note that code examples in the paper are sometimes slightly altered for brevity from the example program.) Then, the following code adds the script, line by line, to that object using Tcl_AppendStringsToObj:

```
std::ifstream file_loader( load_file.c_str() );
std::string load_line;
while( file_loader ) {
    std::getline(file_loader, load_line);
    Tcl_AppendStringsToObj( tcl_script,
        load_line.c_str(), "\n", NULL );
}
```

The script file itself is now loaded into the interpreter using Tcl_EvalObjEx:

```
rc = Tcl_EvalObjEx(interp,tcl_script,TCL_EVAL_GLOBAL);
```

Now the interpreter is ready to run the indicated proc for each vector in the analysis set.

4 Exploring the Iteration Interfaces

The main body of the paper explores several interfaces that an application can present to user through the Tcl C Tcl_Obj and Tcl_ObjType interfaces. The goal for these interfaces is to provide users simple and intuitive access to large native application datasets in an efficient manner that looks at least vaguely familiar and intuitive to Tcl users.

4.1 Accessing a Tcl List directly

The most natural and straight forward mechanism for iteration in Tcl is simple iteration through a Tcl list:

```
proc do_calculation input_list {
    # now iterate
    foreach var $input_list {
        puts $var
    }
}
```

The Tcl list is, then, a very straight forward mechanism to providing access to application data. The Tcl List interface is used in the Calibre product's LVS Device recognition application in order to provide access to a (usually) short set of numbers describing proximity of features near a transistor. Lists in the example program are constructed using the Tcl_ListObjAppendElement interface. The command is invoked by name (from the proc_name argument), with the list passed in the second field of the command:

```
//
// Tcl List construction from a C++ std::vector<double>
//
Tcl_Obj *command[2];
command[0] = Tcl_NewStringObj( proc_name.c_str(), -1 );
Tcl_IncrRefCount(command[0]);
command[1] = Tcl_NewObj(); Tcl_IncrRefCount(command[1]);
for( std::vector<double>::iterator i = data.begin();
     i != data.end(); ++i )
{
    Tcl_ListObjAppendElement(interp, command[1],
        Tcl_NewDoubleObj( *i ));
}
```

After construction of the list, the command text and the list are passed in to the calling script using Tcl_EvalObjv with the script text as the first argument and the list as the second argument.

```
int rc = Tcl_EvalObjv(interp,2,command,TCL_EVAL_GLOBAL);
```

Since the interface here is through a real Tcl list, this method presents the most natural interface to the Tcl programmer. Its main drawback is that the data structure must be fully copied from its native C++ into the Tcl list. For applications that have very large datasets, or high performance goals, the overhead required to form the Tcl list may be unacceptable. For those applications, the other access mechanisms may be more appropriate.

4.2 Access through an Index

The second interface demonstrated uses an index for random access into the contents of the container:

```
proc do_calculation my_arr {
    # returns an object count
    set entry_count [ $my_arr entry_count ]
    # iterate using an index
    for { set i 0 } { $i < $entry_count } { incr i } {
        puts "my_arr $i => [ $my_arr value $i ]
    }
}
```

The interface to the array is provided through the Tcl_CreateObjCommand interface.. In order to construct that interface, the example program uses Tcl's Tcl_CreateObjCommand interface.

```
Tcl_CreateObjCommand(interp,"arg1",
    vector_interface,data,NULL);
```

This call creates a command object named “arg1”, bound with data pointer data, and implemented through the some_stats_vector_interface function. The name “arg1” is arbitrary and it is only used when inside the application as seen below. Inside the called proc, this command is bound to a parameter of the called proc. This technique allows the end user to select meaningful names for what are potentially a large number of parameters that all have real names that aren't very meaningful to the end user.

Next, the index_interface function provides implementation for the required commands:

```
int index_interface(
       ClientData cd,
       struct Tcl_Interp *interp,
       int objc,
       Tcl_Obj *CONST objv[] )
{
    std::vector<double>* data =
         static_cast<std::vector<double>*>(cd);
    const char* command =
            Tcl_GetStringFromObj( objv[1], NULL );
    if ( strcmp( command, "size" ) == 0 ) {
    f  size_t sz = data->size();
      Tcl_Obj *result=Tcl_NewLongObj(sz);
      Tcl_IncrRefCount(result);
      Tcl_SetObjResult( interp, result );
   } else if ( strcmp( command, "value" ) == 0 ) {
      ...
```

The object command is passed along with the name of the proc as an argument to Tcl_EvalObjEx. This is where the name “arg1” is used. It is not visible to the user (unless the user knows to look for it).

```
std::string invoke_line = procname;
invoke_line.append( " arg1" );
int rc = Tcl_Eval(interp,invoke_line.c_str());
```

The array index interface is used in the Calibre product's LVS Device recognition application in order to provide access to a randomly accessible array of measurement numbers related to a transistor. The advantage of this method over the constructed List method is mainly efficiency. The contents of the C++ vector are accessed directly by methods implemented through the Tcl_CreateObjCommand interface. The interface is not nearly as elegant as the list interface for simple iteration over the contents of a container. It also isn't suitable for data that doesn't fit an index->value retrieval model. The next interface extends the index to a more fully fledged iterator accessor.

4.3 Using an iterator interface similar to C++ iterators

The C++ standard library provides a convenient common mechanism for iteration through containers. That mechanism is called the 'iterator'. The code looks like this if you want to iterate through all members of an array of doubles called 'data' printing each item on a separate line:

```
std::vector<double>::iterator i = data.begin();
while ( i != data.end() ) {
   std::cout << *i << std::endl
   ++i;
}
```

We might construct a similar interface in Tcl where code could look like this:

```
set my_iter [ $data get_iterator ]

while { ! [$data at_end $my_iter] } {

    puts "my_arr $i => [ $data value $my_iter ]

    $data incr $my_iter

}
```

In the example program, the iterator interface is constructed from two parts. The record is accessed via a Tcl command object that is similar to the one used in the indexed interface. In the place of the index, the iterator is a full fledged Tcl_ObjType object. It can retain state and independent settings from the container itself. It is also more vulnerable to going out of synch with the container, so may require mechanisms to void its validity if the container changes state while the iterator is still in existence. The initialization of the command object is pretty much the same, using Tcl_CreateCommandObj, as it is for the indexed access. The commands supported by the implementation command are:

- get_iterator - returns an iterator to the beginning of the data container
- at_end - indicates the iterator is past the last data item in the data container
- incr - moves the iterator to the next item in the container
- value - retrieves the value represented by the iterator

The iterator itself represents the std::vector<double>::iterator and that is its only data member in this implementation. That is actually quite inadevalue quate since the vector iterator is represented by a raw pointer into the memory of the data vector. As long as the data vector remains in its original state, the iterator is fine. If the data container is altered or goes away, the iterator should, in fact, be invalidated immediately. This would normally be done with some sort of Observer pattern where the interface retains a list of active iterators and can void them when ever any operation occurs that would invalidate an iterator.

The Tcl_ObjType interface that contains the iterator pointer is implemented using the standard name and set of type handling functions for free, duplicate, update_string and set_from_any. These functions manage the access to the C++ iterator.

The iterator style interface is used in the Calibre product's Yield Server application to access a wide variety of EDA design data like electrical nets, devices, design cells, and geometries etc.

5 Exploring more consistent interfaces

The implementations explored thus far have resulted in vastly different Tcl code because of the mechanics of the underlying iteration mechanism and the fact that the Tcl foreach command is strictly a list-based iteration mechanism. In the next section, two methods of providing a more generic interface are explored. While the foreach command is strictly list based, a specialized foreach-like command can be used to soften the differences between the custom interfaces and the Tcl list interface. Coroutines, new to tcl, are also referred to as generators. They provide a potentially much more powerful and consistent interface to the problem of iteration.

5.1 Using a specialized foreach command

It is possible to adapt the interfaces presented earlier to get closer to the syntactic simplicity of the original foreach loop around the list. A specialized foreach-like command can be implemented that allows use of syntax that is very similar to the original foreach implementation on the list. The specialized foreach command can hide the differences between the various access interfaces allowing the user routine to The example program implements such a foreach_instance command on top of the indexed access method presented earlier. It does this with a specialized command “foreach_instance” which allows the following interface:

```
proc do_calculation record {
    foreach_instance value $record {
        puts $value
    }
}
```

which is very close to a Tcl List interface:

```
proc do_calculation record {
    foreach value $input_list {
        puts $value
    }
}
```

This foreach_instance command is implemented entirely in Tcl - and it hides the complexity of the index access interface. The foreach_instance proc is implemented as:

```
proc foreach_instance { var1 record body } {
    set vlen [ $record size ]
    upvar 1 $var1 value# now iterate
    for { set i 0 } { $i < $vlen } { incr i } {
        set value [$record value $i]
        uplevel 1 $body
    }
}
```

The foreach style top level command is used by the Calibre LVS Comparison application's device reduction application to give iterative access to a potentially large singly linked list of devices. While the two scripts are quite similar, they vary on the name of the critical foreach command itself. In the next section, use of a coroutine allows the difference to be obscured using another mechanism.

5.2 Using a Tcl coroutine with the index interface

Implementing a coroutine interface further explores the iterative style command in the context of coroutines. Using the coroutine, like the specialized foreach command, requires a specialized adapter routine that traverses the data structure for another command that is doing the calculation. One simple way to traverse a coroutine until it is empty follows. This example uses a coroutine to traverse a Tcl list:

```
proc do_calculation { record } {
  coroutine data_fetcher get_from_record $record
  while 1 {
    puts [ data_fetcher ]
  }
}
```

In this proc, the coroutine data_fetcher is created from the proc get_from_record (not shown) and its argument $record, the list of data. It then goes into a while loop that prints the value of each item in the list. The loop is broken when data_fetcher returns with a -code break return code that indicates the end of the list. Next is the implementation of a list iteration form of get_from_record:

```
proc get_from_record record {
    yield [ info coroutine ]
    foreach value $record {
        yield $value
    }
    return -code break
}
```

This calculation proc can remain unchanged while the get_from_record proc changes to cover a different interface - in this instance, the index interface shown above:

```
proc get_from_record record {
    set vlen [ record size ]
    yield [ info coroutine ]
    for { set idx 0 } { $idx < $vlen } { incr idx } {
        yield [ $the_record value $idx ]
    }
    return -code break
}
```

Tcl Coroutines are not currently used in the Calibre application family which is still using Tcl 8.4.

6 Conclusion

While Tcl provides a number of high performance adaptable interfaces to a C application programmer, iteration over a data collection is still quite cumbersome due to the differences in handling C object type collections and Tcl lists. These differences in interface are overcome in certain situations through the use of a customized foreach-like command, but that approach has a drawback in that the foreach-like command itself is specific to its data container. Coroutines provide promise for providing a common iteration mechanism within Tcl, though the language feature is new and idioms are not yet well developed.

7 Acknowledgments

Special thanks go to Donal K. Fellows for his assistance in writing the specialized foreach command and coroutine adapter interfaces.