Object Lifetimes in Tcl - Uses, Misuses, Limitations

By

Phil Brooks - Mentor – A Siemens Corporation

Presented at the 25nd annual Tcl/Tk conference, Houston Texas, November 2018

Mentor – A Siemens Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

Abstract: The management of the lifetime of an object in Tcl presents a number of unique challenges. In the C++ world, the technique of Resource Allocation is Initialization (RAII) is commonly used to control the lifetime of certain objects. With this technique, an object on the call stack is used to insure proper cleanup of a resource. When the object goes out of scope, the resource is released. It presents a low overhead mechanism for a garbage collection like facility without the complication of more complete garbage collection systems. Tcl doesn't have a similar direct capability, and the need for it has resulted in two techniques that are commonly used to substitute for the capability. The techniques are:

1) **Use of the trace command to catch unset of a variable.**
2) **Mis-use of the lifetime of a Tcl_Obj created with Tcl_NewObj.**

Each of these techniques has drawbacks. The primary issue with the trace technique is that it requires the user of an object to arrange for explicit destruction of the object. This makes the interface more error prone. The mis-use of lifetime of a Tcl_Obj is subject to premature cleanup of the resource if the object is shimmered to a string for any reason.
This paper surveys these lifetime management patterns and demonstrates the use of a new user level technique for lifetime management. A new Tcl feature is advocated to make this cleaner and more efficient.

C++ Resource Allocation Is Initialization (RAII)

Used as a common pattern in C++ design, the RAII technique makes use of a class destructor and automatic destruction of stack objects to handle required release of resources needed by a class. Consider this C++ example:

```
// A simple demonstration of RAII (Resource Allocation Is Initialization)
//
//

#include <iostream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>

std::string lsof_command;

class RAII_file {
    public:
        FILE* fh;
        RAII_file( const char* filename ) : fh(NULL) {
            if ( filename ) {
                fh = fopen( filename, "w" );
            }
        }
        ~RAII_file() {
            if ( fh ) {
                fclose( fh );
            }
        }
};

void do_file_stuff() {
    RAII_file F( "foo.txt" );
    fprintf( F.fh, "hello world\n" );

    std::cout << "\nopen files during call to do_file_stuff:" << std::endl;
    system( lsof_command.c_str() );

}


int main() {
    // Set up to run Unix lsof (list open files) for this process
    pid_t mypid = getpid();
    std::stringstream lsof;
    lsof << "/usr/sbin/lsof +d . -p " << mypid << " | grep foo.txt";
    lsof_command = lsof.str();
    std::cout << lsof_command << std::endl;

    std::cout << "\nopen files to start with:" << std::endl;
    system( lsof_command.c_str() );

    do_file_stuff();

    std::cout << "\nopen files at the end:" << std::endl;
    system( lsof_command.c_str() );
}
```

The purpose of the RAII_file class is to insure that the file opened in the constructor is closed in the destructor. In the use of the class, in the function do_file_stuff(), the user doesn't need to do anything specific to insure that the file is closed when the do_file_stuff() function returns. It

is simply handled by the destructor of the class as the object F goes out of scope. This pattern is frequently used in C++ to make up for the fact that C++ does not have built-in garbage collection.

Tcl File Example

The standard mechanism for doing something similar in Tcl is to use the trace command to close the file when a particular variable is unset. So, for example, consider the following Tcl code:

```
proc do_file_stuff { } {
    set fh [ open "foo.txt" "w" ]
    puts $fh "hello world"

    puts "open files during call to do_file_stuff:"
    puts [ chan names ]

    close $fh
}

proc run_example { } {

    puts "open files to start with:"
    puts [ chan names ]
    do_file_stuff

    puts "open files at the end:"
    puts [ chan names ]
}

run_example
```

Here the file is opened and then explicitly closed after its use. Failure to close the file will likely result in bugs.

Tcl Trace Example

In this example, we make the first attempt to use an RAII type behavior to the open file handle by adding a trace to the newly created file handle variable so that it can be closed when it is unset. The do_file_stuff proc opens a file, writes to it and closes it at the end of the proc. In order to automatically close the file at the end of the proc, we add the trace immediately after opening the file:

```
proc cleanup_file { objname args } {
    puts "cleanup_file is closing $objname"
    close $objname
}

proc do_file_stuff { } {
    set fh [ open "foo.txt" "w" ]
    trace add variable fh unset "cleanup_file $fh"

    puts $fh "hello world"

    puts "open files during call to do_file_stuff:"
    puts [ chan names ]

}

proc run_example { } {

    puts "open files to start with:"
    puts [ chan names ]

    do_file_stuff

    puts "open files at the end:"
    puts [ chan names ]
}

run_example
```

Since unset is called on local variables when the proc returns, the file is closed appropriately. While this achieves the desired result of closing the file at the end of the proc, it requires the user to add the trace manually after the file is opened. It is not as simple to use as the C++ example where details of the file handling are all hidden inside of the class.

Tcl Object Example using Itcl

We can get rid of the need to create a specific cleanup proc by encapsulating the file object creation and deletion in a Tcl object system like TclOO or IncrTcl:

```
package require Itcl

# File cleanup using an Itcl class
itcl::class file_handler {
  variable m_data
  constructor { filename } {
    set m_data [ open $filename "w" ]
  }
  destructor {
    puts "closing file $m_data"
    close $m_data
  }
  method get_file {} {
      return $m_data
  }
}

proc do_file_stuff {} {
    file_handler f1 "foo.txt"
    puts [f1 get_file] "hello_world"
    puts "open files during call to do_file_stuff:"
    puts [ chan names ]
    itcl::delete object f1
}

proc run_example {} {
    puts "open files to start with:"
    puts [ chan names ]

    do_file_stuff

    puts "open files at the end:"
    puts [ chan names ]
}

run_example
```

However, the file_handler object still requires specific deletion of the object. Object, unlike variables, do not have lifetimes limited by the scope of the proc they are created in. Ultimately, this can again be handled by the trace command – or using the built-in encapsulation of the trace command provided by the Itcl local object declaration:

```
itcl::local file_handler f1 "foo.txt"
```

It is interesting to note what is going on behind the scenes with itcl::local. As described in the Itcl documentation and as the following info commands illustrate, Itcl is setting an unset trace on a special variable that was created behind the scenes:

```
    set local_vars [ info vars "itcl-local*" ]
    puts "itcl-local variables $local_vars"
    foreach var $local_vars {
        puts "Trace info for $var: [ trace info variable $var ]"
```

```
    }
```

produces the output:

```
itcl-local variables: itcl-local-f1
Trace info for itcl-local-f1: {unset {::itcl::delete_helper ::f1}}
```

The problem with this mechanism is, again, the requirement for a special syntax in order to get the destructor called. It is much simpler than setting up the trace explicitly. In addition, it doesn't work for global object. Though unset traces are supposed to be called when the interpreter is destroyed and global objects are said to be unset during deletion of an interpreter, they are apparently not called when the exit command is called (as it is when you use tclsh directly. It also seems a bit hokey that we can't get the behavior we want from an Itcl or other Tcl objects directly, so we approximate it by creating a regular Tcl variable and tracing that instead. Why can't we get the desired behavior from the Itcl object itself?

The Tcl_Obj Hack

A common mis-use of the Tcl_ObjType structure and its free function pointer is to try to subvert this into an object system (As an aside, Don Porter says Tcl_Obj would better be called Tcl_Value instead of Tcl_Obj, because it is not designed as an object, but, more accurately, as a value). The mis-use comes about when the Tcl_ObjType's string value is not sufficient to resurrect it after it has been converted to a pure string in the side effect known as "shimmering". Since the lifetime of the Tcl_Obj follows exactly what we need in creating a stack based object lifetime, this misuse is a tempting and frequent blunder for those trying to extend Tcl. This lifetime is precisely why the Itcl local mechanism works so well – it relies on handling of the Tcl_Obj lifetime to manage the lifetime of an object.

Frequently, the motivation to do this comes from the need to create an object that contains more state than can easily be contained or encoded in a string – say a reference to a location in a large data-structure with a lifetime that we have to manage. The temptation is to tie the lifetime of the large data-structure directly to the Tcl_Obj. The problem with that is that if or when the Tcl_Obj undergoes shimmering, the Tcl_ObjType's free function is called and the large data-structure is deleted and that state can't easily be replicated using the string representation of the Tcl_Obj since the large data-structure is now gone.

Using the secret variable mechanism with Tcl_ObjType

It is possible to use a variation on the same mechanism that Itcl uses to maintain lifetime of an "Object" created using the Tcl_ObjType interface. That is, to create a separate variable in the same scope as the object who's lifetime we want to manage and then tracing that variable to manage the lifetime of the large data-structure. Consider the following C++ code that is used to create a Tcl_Obj of type Foo and to track the destruction of Foo on the basis of a local variable created

```
Tcl_Obj * Create_Tcl_Foo (Tcl_Interp* interp )
{
    size_t tracker_index = 0;
    FooKeeper* the_keeper = NULL;
    Tcl_Obj* local_foo_tracker = Tcl_GetVar2Ex( interp, FooTrackerVarname, NULL, 0);
    if ( !local_foo_tracker ) {
        // First time one is created at this scope
        tracker_index = FooKeeper::last_index++;
        AllFoos[tracker_index] = FooKeeper(tracker_index);
        the_keeper = &AllFoos[tracker_index];
        Tcl_Obj *new_obj = Tcl_NewObj();
        new_obj->typePtr = &Tcl_Foo_tracker;
        new_obj->internalRep.otherValuePtr = the_keeper;
        UpdateStringOf_Tcl_FooTracker(new_obj);

        Tcl_SetVar2Ex( interp, FooTrackerVarname, NULL, new_obj, 0 );
        Tcl_TraceVar2( interp, FooTrackerVarname, NULL, TCL_TRACE_UNSETS, trace_unset_collect
    } else if ( local_foo_tracker->typePtr == &Tcl_Foo_tracker ) {
        // Use the previously created tracker.
        the_keeper = (FooKeeper*)local_foo_tracker->internalRep.otherValuePtr;
        tracker_index = the_keeper->index;
    }
    size_t f_index = the_keeper->Foos.size();
    Foo* f = new Foo(tracker_index, f_index);
    f->ref_count = 1;
    Tcl_Obj *objPtr = Tcl_NewObj();

    objPtr->bytes = NULL;

    objPtr->internalRep.otherValuePtr = (void*)f;
    objPtr->typePtr = &Tcl_Foo;

    if ( the_keeper ) {
        the_keeper->Foos.push_back(f);
        f->ref_count++;
    }
    cout << "Created Foo: " << f->name << endl;
    return objPtr;
}
```

The above code is managing a collection of Foo objects in a global collection. It will delete all of the objects from a particular scope when that scope's tracking variable unset trace function is called. That way, the string representation of a particular object can be used to retrieve the full access to the object and its large data-structure without reconstruction and the large data-structure can be cleaned up at the appropriate time.

The following Tcl code illustrates use of the FooTracker collection manager:

```
proc try_it { } {
    # the do_foo command is creating a local variable that will manage the
    # lifetime of the foo object.
    set foo [ do_foo make ]
    puts [ do_foo name $foo “Harry” ]
    puts [ do_foo name $foo “Jane” ]
    # foo destructor is called automatically due to trace set up when it was created
    # by do_foo make.
}
```

Why the need for two variables?

In this case, both the user level object and the hidden object used to maintain the large data-structure's lifetime could really be handled at the same time if there were a mechanism available through the set command if there were an easy way for a particular Tcl_Obj to indicate to the set command that it requires tracing as well. Proposals have been floated for changing the Tcl_ObjType and Tcl_Obj structures for Tcl 9 – perhaps something could allow for automatic tracing of variables on creation? It would be preferable not to have to change the arguments to set as this, again, pushes the requirement to do something special onto the user of an object rather than allowing the object implementation to manage its own cleanup.

Proposed Tcl 9.0 Enhancement Discussion

Make it easier to place an unset trace (or equivalent) on a variable during assignment. So the code:

```
set x [ get_a_foo ]
```

would allow the underlying "C" function or Tcl proc for get_a_foo to set a trace on x. Similarly lappend, array set, etc. would need similar support. From C/C++, this might be accomplished through extensions on the Tcl_Obj and Tcl_ObjType data structures. For Tcl, it might be accomplished by an option to the return statement.