Writing Tcl-Based Applications In C

John Ousterhout

Sun Microsystems Laboratories

`john.ousterhout@eng.sun.com`

Tcl/Tk Tutorial, Part IV

Outline

- **Philosophy: focus on primitives.**
- **Basics: interpreters, executing scripts.**
- **Implementing new commands.**
- **Managing packages; dynamic loading.**
- **Managing the result string.**
- **Useful library procedures: parsing, variables, lists, hash tables.**

Philosophy

- **Usually better to write Tcl scripts than C code:**
  - Faster development (higher level, no compilation).
  - More flexible.
- **Why write C?**
  - Need access to low-level facilities (sockets?).
  - Efficiency concerns (iterative calculations).
  - Need more structure (code is complex).
- **Implement new Tcl commands that provide a few simple orthogonal primitives:**
  - Low-level to provide independent access to all key features.
  - High-level to hide unimportant details, allow efficient implementation.

Example: Weather Reports

- **Goal: retrieve weather reports over network from servers.**
- **Tcl command set #1:**
  - Retrieve report, format, print on standard output.

  } Too high-level (inflexible)
- **Tcl command set #2:**
  - Open socket to weather server.
  - Select station.
  - Retrieve first line of report....

  } Too low-level (tedious)
- **Tcl command set #3:**
  - Return list of available stations.
  - Given station name, retrieve report.

  } Just right

Interpreters

- **`Tcl_Interp` structure encapsulates execution state:**
  - Variables.
  - Commands implemented in C.
  - Tcl procedures.
  - Execution stack.
- **Can have many interpreters in a single application (but usually just one).**
- **Creating and deleting interpreters:**

```
Tcl_Interp *interp;
interp = Tcl_CreateInterp();
Tcl_DeleteInterp(interp);
```


Executing Tcl Scripts

```
int code;
code = Tcl_Eval(interp, "set a 1");
code = Tcl_EvalFile(interp, "init.tcl");
```

- **`code` indicates success or failure:**
  - `TCL_OK`: normal completion.
  - `TCL_ERROR`: error occurred.
- **`interp->result` points to string: result or error message.**
- **Application should display result or message for user.**

Designing New Commands

- **Choose textual names for objects:**

```
.dlg.bottom.ok
file3 or stdin
```

  - Use hash tables to map to C structures.
- **Object-oriented commands:**

```
.dlg.bottom.ok configure -fg red
```

  - Good for small numbers of well-defined objects.
  - Doesn't pollute name space.
  - Allows similar commands for different objects.
- **Action-oriented commands:**

```
string compare $x $y
```

  - Good if many objects or short-lived objects.

Designing New Commands, cont'd

- **Formatting command results:**
  - Make them easy to parse with Tcl scripts:

```
tmp 53 hi 68 lo 37 precip .02 sky part
```

  - Make them symbolic wherever possible, e.g. not 53 68 37 .02 7
- **Use package prefixes in command names and global variables:**

```
wthr_stations
wthr_report
midi_play
```

  - Allows packages to coexist without name clashes.

Creating New Tcl Commands, cont'd

- **Register with interpreter:**

```
Tcl_CreateCommand(interp, "eq", EqCmd,
        (ClientData) NULL, ...);
Tcl_DeleteCommand(interp, "eq");
```

- **Once registered, `EqCmd` will be called whenever `eq` command is invoked in `interp`.**

Where Do Scripts Come From?

- **Read from standard input (see tclMain.c).**
- **Read from script file (see tclMain.c).**
- **Associate with X events, wait for events, invoke associated scripts (see tkMain.c).**

ClientData

```
Tcl_CreateCommand(interp, "eq", EqCmd,
        clientData, ...);
int EqCmd(ClientData clientData, ...) {...}
```

- **Used to pass any one-word value to command procedures and other callbacks.**
- **`clientData` is usually a pointer to data structure manipulated by procedure.**
- **Cast pointers in and out of `ClientData` type:**
  - `Tcl_CreateCommand(... (ClientData) gizmoPtr, ...);`
  - `gizmoPtr = (Gizmo *) clientData;`

Creating New Tcl Commands

- **Write command procedure in C:**

```
int EqCmd(ClientData clientData, Tcl_Interp
        *interp, int argc, char **argv) {
    if (argc != 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (strcmp(argv[1], argv[2]) == 0) {
        interp->result = "1";
    } else {
        interp->result = "0";
    }
    return TCL_OK;
}
```

Packages, cont'd

3. To use package:

- Compile as shared library, e.g. on Solaris:

```
cc -K pic -c rdb.c
ld -G -z text rdb.o -o rdb.so
```

- Dynamically load into **`tclsh`** or **`wish`**:

```
load rdb.so Rdb
```

- Tcl will call **`Rdb_Init`** to initialize the package.

Conventions For Packages

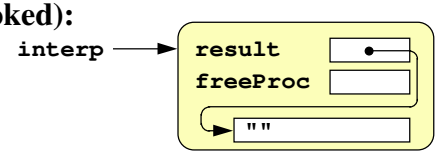
Goal: make it easy to develop and use Tcl extensions.

1. Use package prefixes to prevent name conflicts:

- Pick short prefix for package, e.g. **`rdb`**.
- Use in all global names:
  - » C procedure: **`Rdb_Open`**
  - » C variable: **`rdb_NumRecords`**
  - » Tcl command: **`rdb_query`**
- See Tcl book and Tcl/Tk Engineering Manual for more details.

Managing The Result String

- **Need conventions for `interp->result`:**
  - Permit results of any length.
  - Avoid **`malloc`** overheads if possible.
  - Avoid storage reclamation problems.
  - Keep as simple as possible.
- **Normal state of interpreter (e.g., whenever command procedure is invoked):**



- **Default: command returns empty string.**

Packages, cont'd

2. Create package initialization procedure:

- Named after package: **`Rdb_Init`**.
- Creates package's commands.
- Evaluates startup script, if any.

```
int Rdb_Init(Tcl_Interp *interp) {
    Tcl_CreateCommand(...);
    Tcl_CreateCommand(...);
    ...
    return Tcl_EvalFile(interp,
            "/usr/local/lib/rdb/init.tcl");
}
```

Procedures For Managing Result

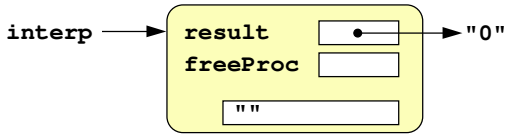
- **Option 4: use library procedures.**

```
Tcl_SetResult(interp, string, ...);

Tcl_AppendResult(interp, string,
    string, ... string, (char *) NULL);

Tcl_AppendElement(interp, string);

Tcl_ResetResult(interp);
```


Result String, cont'd

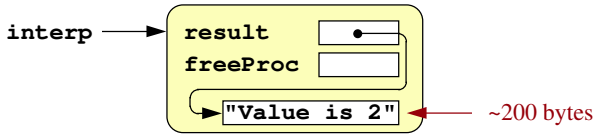
- **Option 1: (semi-) static result.**

```
interp->result = "0";
```



- **Option 2: use pre-allocated space in interp.**

```
sprintf(interp->result, "Value is %d", i);
```



Utility Procedures: Parsing

- **Used by command procedures to parse arguments:**

```
int value, code;
code = Tcl_GetInt(interp, argv[1], &value);
```

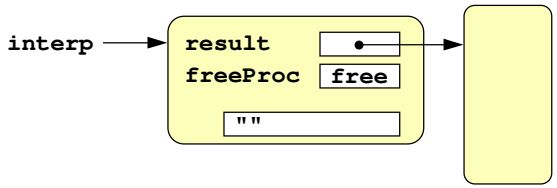
- **Stores integer in `value`.**
- **Returns `TCL_OK` or `TCL_ERROR`.**
- **If parse error, returns `TCL_ERROR` and leaves message in `interp->result`.**
- **Other procedures:**

`Tcl_GetDouble`	`Tcl_ExprDouble`
`Tcl_GetBoolean`	`Tcl_ExprBoolean`
`Tcl_ExprLong`	`Tcl_ExprString`

Result String, cont'd

- **Option 3: allocate new space for result.**

```
interp->result = malloc(2000);
...
interp->freeProc = free;
```



- **Tcl will call `freeProc` (if not NULL) to dispose of result.**
- **Mechanism supports storage allocators other than `malloc`/`free`.**

Utility Procedures: Variables

- **Read, write, and unset:**

```
char *value;
value = Tcl_GetVar(interp, "a", ...);
Tcl_SetVar(interp, "a", "new", ...);
Tcl_UnsetVar(interp, "a", ...);
```

- **Set traces:**

```
Tcl_TraceVar(interp, "a",
        TCL_TRACE_READS|TCL_TRACE_WRITES,
        traceProc, clientData);
```

- **`traceProc` will be called during each read or write of `a`:**
  - Can monitor accesses.
  - Can override value read or written.

Other Utility Procedures

- **Parsing, assembling proper lists:**

```
Tcl_SplitList(...)
Tcl_Merge(...)
```

- **Flexible hash tables:**

```
Tcl_InitHashTable(...)
Tcl_CreateHashEntry(...)
Tcl_FindHashEntry(...)
Tcl_DeleteHashEntry(...)
Tcl_DeleteHashTable(...)
```

- **Dynamic strings:**

```
Tcl_DStringInit(...)
Tcl_DStringAppend(...)
Tcl_DStringAppendElement(...)
Tcl_DStringValue(...)
Tcl_DStringFree(...)
```


Summary

- **Interfaces to C are simple: Tcl was designed to make this true.**
- **Focus on primitives, use Tcl scripts to compose fancy features.**