Writing Tcl-Based Applications In C

John Ousterhout

Computer Science Division
Department of EECS

University of California at Berkeley

Outline

1. Philosophy: focus on primitives.

2. Basics: interpreters, executing scripts.

3. Implementing new commands.

4. Managing the result string.

5. Useful library procedures: parsing, variables, list manipulation, hash tables.

Philosophy

- Take advantage of Tcl as scripting language.
- Application should:
  - (a) Implement new kinds of objects in C.
  - (b) Define textual names for objects (to use in Tcl commands).
  - (c) Implement primitive operations on objects as Tcl commands.
- Build complex features as Tcl scripts.
- For C code, focus on clean, orthogonal primitives.

Interpreters

- Tcl_Interp data 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_VarEval(interp, "set a",
   " 1", (char *) NULL);
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.

Where Do Scripts Come From?

- Read from standard input (see **`tclTest.c`**).
- Read from script file (see **`tclTest.c`**).
- Associate with X events, wait for events, invoke associated scripts (see **`main.c`** for **`wish`**).

Creating New Tcl Commands

- Write command procedure in C:

```
int cmdProc(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;
}
```

- Register with interpreter:

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

ClientData

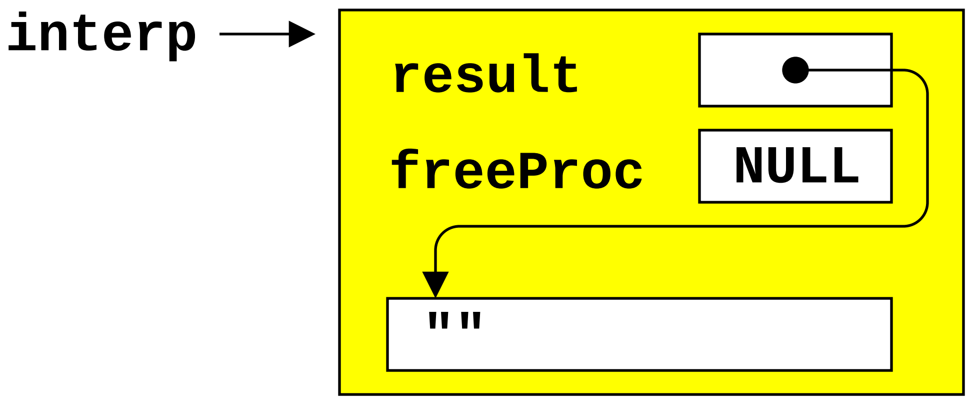
```
Tcl_CreateCommand(interp, "eq", cmdProc,
  clientData, ...);
int cmdProc(ClientData clientData, ...) {
  ...
}
```

- Used to pass any one-word value to command procedures and other callbacks.
- `clientData` is usually a pointer to data structure needed by procedure.
- Widget commands: `clientData` points to widget record.
- Similar in use to `client_data` in Xt.

Managing The Result String

- Need conventions for **`interp->result`**:
  - Permit results of any length.
  - Avoid **`malloc`** overheads if possible.
  - Avoid storage reclamation problems.
  - Simplify command procedures.

- Normal state of interpreter (e.g. whenever command procedure is invoked):

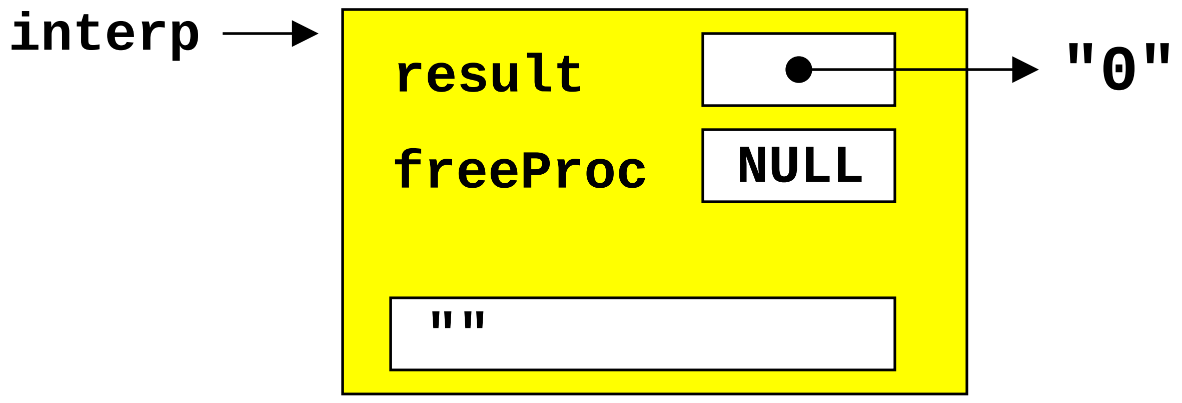


- Default: command returns empty string.

Result String, cont'd

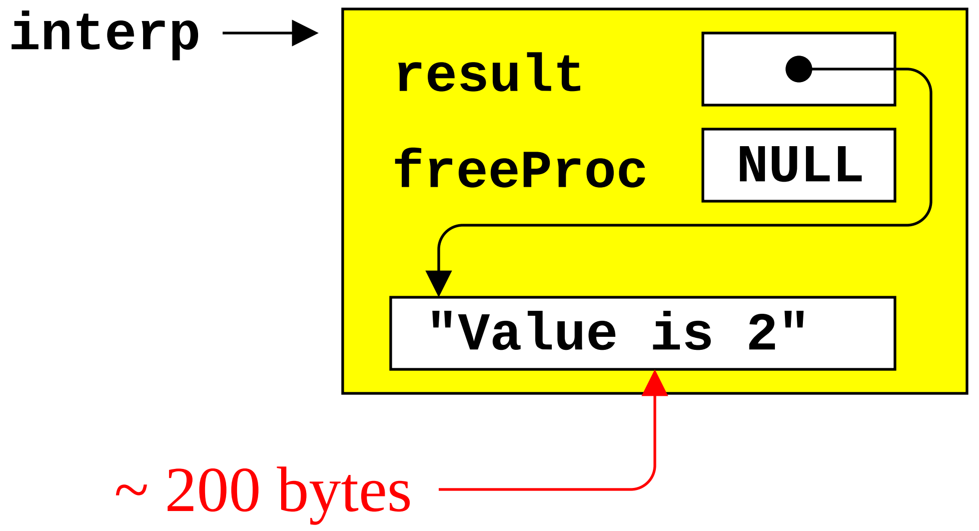
- Option 1: (semi-) static result.

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



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

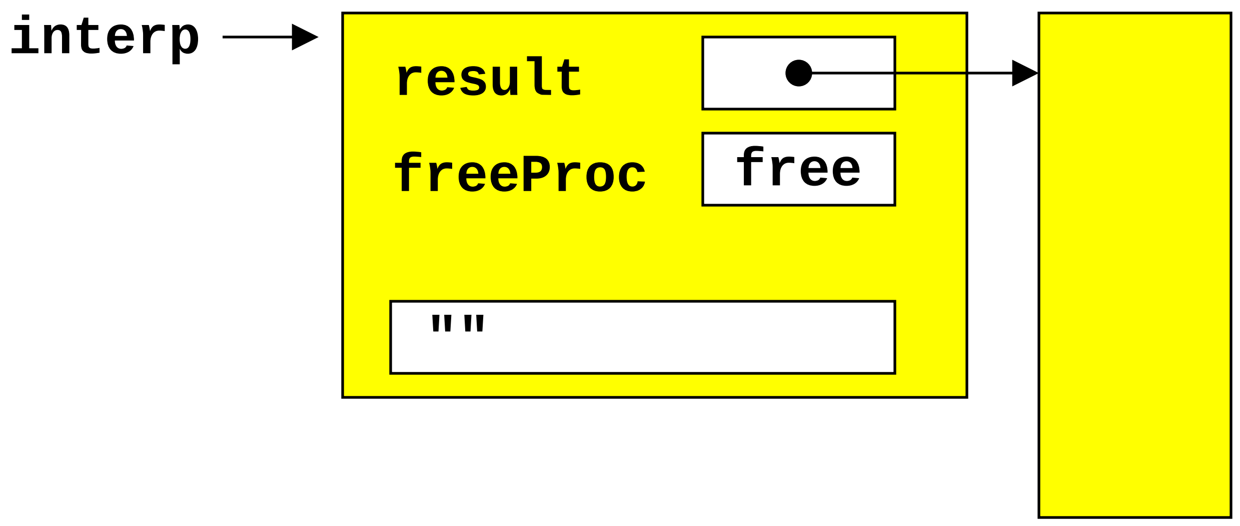
```
sprintf(interp->result, "Value is %d", i);
```



Result String, cont'd

- Option 3: allocate new space for result.

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



- Tcl will call **`freeProc`** (if non-**`NULL`**) to dispose of result.

- Mechanism supports storage allocators other than **`malloc/free`**.

Procedures For Managing Result

When in doubt, use library procedures: sometimes slower, always safe.

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

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

Tcl_AppendElement(interp, string, ...);

Tcl_ResetResult(interp);
```

Utility Procedures: Parsing

- Used by command procedures to parse arguments:

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

- Stores integer value 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
```

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

- Assembling multi-line commands from input:

```
Tcl_CreateCmdBuf(...)
Tcl_AssembleCmd(...)
Tcl_CommandComplete(...)
```

Summary

- Interfaces to C are simple: Tcl was designed to make this possible.

- Focus on primitives, use Tcl scripts to compose fancy features.