Fall 2004 Handout 9

A C program that executes a Tcl script

The following C program executes a file of Tcl commands, like the example in Ousterhout, pp. 288−289. This is easier than executing individual commands as they are typed interactively, because we don't have to worry about continuation lines and recognizing the end of each command. See ''Command completeness'' in Ousterhout, pp. 349−350.

The program also creates two new commands, **`push`** and **`pop`**, for storing and retrieving up to **`N stack_t`**'s. See the example in Ousterhout, pp. 294−295, and

```
1$ man -M /usr/local/man 3 CrtInterp | grops | lpr
2$ man -M /usr/local/man 3 CrtCommand | grops | lpr
```

A program with only one interpreter doesn't need to delete it (Ousterhout, p. 291), since it is automatically deleted at the end of the program. In C++, however, you could make a **`class interpreter`** whose constructor calls **`Tcl_CreateInterp`** and whose destructor calls **`Tcl_DeleteInterp`**.

Similarly, there is no need to delete the two new commands **`push`** and **`pop`** (Ousterhout, p. 303), since they are automatically deleted when their interpreter is deleted. In C++, however, you could make a **`class command`** whose constructor calls **`Tcl_CreateCommand`** and whose destructor calls **`Tcl_DeleteCommand`**.

There's no way to tell if **`Tcl_CreateInterp`**, **`Tcl_CreateCommand`**, etc., completed successfully—they just dump core if something goes wrong.

Line 18 deposits the result of the script's last command (which may be the null string) or an error message into **`interp->result`**. Before each command is executed, **`interp->result`** is reinitialized to point to a reasonably big buffer whose first byte is **`'\0'`** (Ousterhout, p. 297):

```
/* excerpt from tcl.h */
#define TCL_RESULT_SIZE 200
```

Therefore **`pop`** can go ahead and write a string there in line 75, and **`push`** returns the null string. You can also **`malloc`** a buffer to hold the result, but then you have to **`free`** it later (Ousterhout, p. 298).

`Tcl_ExprLong` (Ousterhout pp. 314, 316) in line 54 needs a pointer to an interpreter to know where to deposit its error message if its second argument does not evaluate to a legal **`long`**.

If the script file terminated with an **`exit`** command, the variable **`code`** in line 18 will hold the argument of the **`exit`**. Otherwise, **`code`** will hold one of the completion codes **`TCL_OK`** or **`TCL_ERROR`** in Ousterhout, p. 320.

The C functions implementing the two new commands **`push`** and **`pop`** return either **`TCL_OK`** or **`TCL_ERROR`**. But there are other possible return values. For example, if you implement a new command with a C function that returns **`TCL_BREAK`** (Ousterhout, pp. 319−322), and you write the new command inside of a Tcl loop (**`for`**, **`foreach`**, **`while`**, etc.) then the new command will break you out of the loop.


```
#This file is named scriptfile1.

set a 20

push 10
push $a
push 10+$a           ;#no space around the plus: push takes only 1 arg

for {set i 1} {$i <= 3} {incr i} {
    puts [pop]
}

exit 0
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <tcl.h>
 4
 5 int pushCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv);
 6 int  popCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv);
 7
 8 main (int argc, char **argv)
 9 {
10     Tcl_Interp *interp = Tcl_CreateInterp();
11     int code;
12
13     Tcl_CreateCommand (interp,
14         "push", pushCmd, (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
15     Tcl_CreateCommand (interp,
16         "pop", popCmd, (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
17
18     code = Tcl_EvalFile(interp, "scriptfile1");
19
20     if (interp->result[0] != '\0') {
21         if (code == TCL_OK) {
22             printf ("%s\n", interp->result);
23         } else {
24             fprintf (stderr, "%s: %s\n", argv[0], interp->result);
25         }
26     }
27
28     Tcl_DeleteCommand(interp, "pop");
29     Tcl_DeleteCommand(interp, "push");
30     Tcl_DeleteInterp(interp);
31     exit (code == TCL_OK ? EXIT_SUCCESS : EXIT_FAILURE);
32 }
33
34 #define N 10            /* maximum stack size */
35 typedef long stack_t;
36 #define FORMAT "%ld"
37 stack_t stack[N];
38 int stackp = -1;        /* stack is initially empty */
39
40 int pushCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
```

```
41 {
42     stack_t i;
43
44     if (argc != 2) {
45         interp->result = "wrong # args";
46         return TCL_ERROR;
47     }
48
49     if (stackp >= N - 1) {
50         interp->result = "stack is already full";
51         return TCL_ERROR;
52     }
53
54     if (Tcl_ExprLong(interp, argv[1], &i) != TCL_OK) {
55         /* Tcl_ExprLong has already put error message into interp->result. */
56         return TCL_ERROR;
57     }
58
59     stack[++stackp] = i;
60     return TCL_OK;
61 }
62
63 int popCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
64 {
65     if (argc != 1) {
66         interp->result = "wrong # args";
67         return TCL_ERROR;
68     }
69
70     if (stackp < 0) {
71         interp->result = "stack is already empty";
72         return TCL_ERROR;
73     }
74
75     sprintf (interp->result, FORMAT, stack[stackp--]);
76     return TCL_OK;
77 }
```

I compiled this program with

```
gcc -I/usr/local/include prog.c -ltcl -lm
```
minus lowercase L

because my file **tcl.h** is in the directory **/usr/local/include** instead of **/usr/include**. The standard output of the program is

```
30
20
10
```

and the exit status is zero (**EXIT_SUCCESS**).

Object-oriented commands: Ousterhout, pp. 300−3

```
#This file is named scriptfile2.

#Create a stack named stack1.  Also create a command named stack1.
#The first argument of the stack1 command must be either push or pop.
stack stack1

stack1 push 10
stack1 push 20
stack1 push 30

stack stack2
stack2 push 40

puts [stack1 pop]
puts [stack2 pop]
puts [stack1 pop]
puts [stack1 pop]

#Delete the stack1 and stack2 commands (Ousterhout, pp. 135-6, 302-3)
#and call the DeleteStack function.
rename stack1 ""
rename stack2 ""
exit 0
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <tcl.h>
 5
 6 int  StackCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv);
 7 int ObjectCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv);
 8 void DeleteStack(ClientData clientData);
 9
10 main (int argc, char **argv)
11 {
12     Tcl_Interp *interp = Tcl_CreateInterp();
13     int code;
14
15     Tcl_CreateCommand (interp,
16         "stack", StackCmd, (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
17
18     code = Tcl_EvalFile(interp, "scriptfile2");
19
20     if (interp->result[0] != '\0') {
21         if (code == TCL_OK) {
22             printf ("%s\n", interp->result);
23         } else {
24             fprintf (stderr, "%s: %s\n", argv[0], interp->result);
25         }
26     }
27
```

```
28      Tcl_DeleteCommand(interp, "StackCmd");
29      Tcl_DeleteInterp(interp);
30      exit (code == TCL_OK ? EXIT_SUCCESS : EXIT_FAILURE);
31  }
32
33  #define N 10            /* maximum stack size */
34  typedef long stack_t;
35  #define FORMAT "%ld"
36
37  typedef struct {
38      stack_t stack[N];
39      int pointer;
40  } Stack;
41
42  /* Create a new stack and return its name. */
43
44  int StackCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
45  {
46      Stack *stackPtr;
47
48      if (argc != 2) {
49          interp->result = "wrong # args";
50          return TCL_ERROR;
51      }
52
53      stackPtr = malloc(sizeof(Stack));
54      if (stackPtr == NULL) {
55          interp->result = "not enough memory";
56          return TCL_ERROR;
57      }
58
59      stackPtr->pointer = -1;  /* stack is initially empty */
60      strcpy(interp->result, argv[1]);
61      Tcl_CreateCommand(interp, interp->result,
62          ObjectCmd, (ClientData)stackPtr, DeleteStack);
63      return TCL_OK;
64  }
65
66  void DeleteStack(ClientData clientData)
67  {
68      Stack *stackPtr = (Stack *)clientData;
69
70      if (stackPtr->pointer != -1) {
71          fprintf (stderr, "warning: deleting non-empty stack\n");
72      }
73
74      free (clientData);
75  }
76
77  int ObjectCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
78  {
79      Stack *stackPtr = (Stack *)clientData;
80
81      if (argc < 2) {
```

```
 82          interp->result = "wrong # args";
 83          return TCL_ERROR;
 84      }
 85
 86      if (strcmp(argv[1], "push") == 0) {
 87          stack_t i;
 88
 89          if (argc != 3) {
 90              interp->result = "wrong # args";
 91              return TCL_ERROR;
 92          }
 93
 94          if (stackPtr->pointer >= N - 1) {
 95              interp->result = "stack is already full";
 96              return TCL_ERROR;
 97          }
 98
 99          if (Tcl_ExprLong(interp, argv[2], &i) != TCL_OK) {
100              /* Tcl_ExprLong has already put error message into interp->result. */
101              return TCL_ERROR;
102          }
103
104          stackPtr->stack[++stackPtr->pointer] = i;
105          return TCL_OK;
106      }
107
108      else if (strcmp(argv[1], "pop") == 0) {
109          if (argc != 2) {
110              interp->result = "wrong # args";
111              return TCL_ERROR;
112          }
113
114          if (stackPtr->pointer < 0) {
115              interp->result = "stack is already empty";
116              return TCL_ERROR;
117          }
118
119          sprintf (interp->result, FORMAT, stackPtr->stack[stackPtr->pointer--]);
120          return TCL_OK;
121      }
122
123      interp->result = "first argument must be push or pop";
124      return TCL_ERROR;
125  }
```

The standard output is

```
30
40
20
10
```

and the exit status is zero (**EXIT_SUCCESS**).

Allocate your own space for a command's result string: Ousterhout pp. 296−298

Let's implement a command named **firstarg** whose value is a copy of its first argument:

```
#This file is scriptfile3.
puts [firstarg "hello"]
```

The output is

```
hello
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <tcl.h>
 4
 5 int firstargCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv);
 6
 7 main (int argc, char **argv)
 8 {
 9     Tcl_Interp *interp = Tcl_CreateInterp();
10     int code;
11
12     Tcl_CreateCommand (interp,
13         "firstarg", firstargCmd, (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
14
15     code = Tcl_EvalFile(interp, "scriptfile3");
16     if (interp->result[0] != '\0') {
17         printf ("%s\n", interp->result);
18     }
19
20     Tcl_DeleteCommand(interp, "firstarg");
21     Tcl_DeleteInterp(interp);
22     exit (code == TCL_OK ? EXIT_SUCCESS : EXIT_FAILURE);
23 }
24
25 int firstargCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
26 {
27     if (argc != 2) {
28         interp->result = "wrong # args";
29         return TCL_ERROR;
30     }
31
32     strcpy (interp->result, argv[1]);
33     return TCL_OK;
34 }
```

If **argv[1]** might be longer than the buffer pointed to by **interp->result**, you can use your own buffer. Your buffer will be used for only this one command: **interp->result** will automatically be set back to its old value before the next command:

```
1 #include <string.h>
2
3 int firstargCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
4 {
5     size_t length;
6     static char buffer[1024];
7
```



```
 8     if (argc != 2) {
 9         interp->result = "wrong # args";
10         return TCL_ERROR;
11     }
12
13     length = strlen(argv[1]);
14     if (length >= sizeof buffer)
15         interp->result = "buffer too small"
16         return TCL_ERROR;
17     }
18
19     if (length >= TCL_RESULT_SIZE) {
20         interp->result = buffer;
21     }
22
23     strcpy (interp->result, argv[1]);
24     return TCL_OK;
25 }
```

Use **malloc** and **free** if you can't predict the maximum length of **argv[1]**. **interp->freeProc** is automatically initialized to **NULL** before each Tcl command. If you put the address of a function in **interp->freeProc** (line 21 below), the Tcl interpreter will pass **interp->result** to that function:

```
(*interp->freeProc)(interp->result);
```

```
 1 #include <string.h>
 2
 3 int firstargCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
 4 {
 5     size_t length;
 6     char *p;
 7
 8     if (argc != 2) {
 9         interp->result = "wrong # args";
10         return TCL_ERROR;
11     }
12
13     length = strlen(argv[1]);
14     if (length >= TCL_RESULT_SIZE) {
15         p = malloc(length + 1);
16         if (p == NULL) {
17             interp->result = "allocation failed";
18             return TCL_ERROR;
19         }
20         interp->result = p;
21         interp->freeProc = free;
22     }
23
24     strcpy (interp->result, argv[1]);
25     return TCL_OK;
26 }
```

What the Tcl interpreter does before and after it executes each Tcl command

```
 1      char buffer[TCL_RESULT_SIZE];
 2
 3      for (;;) {
 4          interp->freeProc = NULL;
 5          buffer[0] = '\0';
 6          interp->result = buffer;
 7
 8          Read and execute the next Tcl command;
 9          if (the command did not return TCL_OK) {
10              break or go to the enclosing "catch" command;
11          }
12
13          if (there are more Tcl commands coming && interp->freeProc != NULL) {
14              (*interp->freeProc)(interp->result);
15          }
16      }
```

Don't call malloc, realloc, and free yourself Ousterhout pp. 298−300

You can build the result in stages. The following is an excerpt from a command whose value is the string **`George Herbert Walker Bush`**.

```
 1      /* Error checking removed for brevity. */
 2
 3      interp->result = malloc(7);
 4      strcpy(interp->result, "George");
 5
 6      interp->result = realloc(interp->result, 22);
 7      strcat(interp->result, " Herber Walker");
 8
 9      interp->result = realloc(interp->result, 27);
10      strcat(interp->result, " Bush");
```

It's simpler to say

```
 1      /* The result is automatically initialized to the null string:
 2      you don't need to start with Tcl_SetResult. */
 3
 4      Tcl_AppendResult(interp, "George", (char *)NULL);
 5      Tcl_AppendResult(interp, " Herbert", " Walker", (char *)NULL);
 6      Tcl_AppendResult(interp, " Bush",   (char *)NULL);
```

`Tcl_AppendElement` puts curly braces around the appended element if it contains embedded white space. More precisely, it puts curly braces around an appended element whose **`length`** is greater than 1. For example, the value of a command containing

```
 1      /* The result is automatically initialized to the null string:
 2      you don't need to start with Tcl_SetResult. */
 3
 4      Tcl_AppendElement(interp, "January 31");
 5      Tcl_AppendElement(interp, "February 28");
 6      Tcl_AppendElement(interp, "March 31");
```

would be the list


```
    {January 31} {February 28} {March 31}
```

Where does the interpreter get the Tcl code from? (Ousterhout pp. 287−291)

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <tcl.h>
 4
 5 main()
 6 {
 7     Tcl_Interp *interp = Tcl_CreateInterp();
 8     int code;
 9
10     code = Tcl_EvalFile(interp, "scriptfile3");
11     code = Tcl_Eval(interp, "puts two\nputs three; puts four");
12     code = Tcl_VarEval(interp, "puts five\n", "puts si",
13         "x; puts seven", (char *)NULL);
14
15     Tcl_DeleteInterp(interp);
16     exit (code == TCL_OK ? EXIT_SUCCESS : EXIT_FAILURE);
17 }
```

```
#This file is scriptfile3.
puts one
```

The output is:

```
one
two
three
four
five
six
```

Command completeness: Ousterhout, pp. 349−350

A Tcl command can span more than one line:

```
puts \
    supercalifragilisticexpialidocious

for {set i 1} {$i <= 10} {incr i} {
    puts $i
}
```

The following program reads Tcl commands from the standard input and executes them. We dynamically allocate a string with **`malloc`** and **`realloc`** to hold the concatenation of all the lines of a command. When the command is syntactically complete, we execute the command, **`free`** the string, and start to build up another string.

The program prints **`prompt1:`** before the user types the first (or only) line of a command, and **`prompt2:`** before the user types the remaining line(s) of a command. See the variables **`$tcl_prompt1`** and **`$tcl_prompt2`**.

The C function **`gets`** discards newlines; **`fgets`** doesn't.

```
1 #include <stdio.h>
```



```
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <tcl.h>
 5
 6 main()
 7 {
 8     Tcl_Interp *interp = Tcl_CreateInterp();
 9     char line[200];  /* buffer to hold each line of input */
10     char *p;         /* pointer to dynamically allocated string */
11     int code;
12
13     /* Each iteration of this loop executes one Tcl command. */
14     for (;;) {
15         printf ("\nprompt1: ");
16         p = malloc(1);
17         p[0] = '\0';
18
19         /* Loop to read all the lines of the command. */
20         for (;;) {
21             if (fgets(line, sizeof line, stdin) == NULL) {
22                 goto done;
23             }
24             p = realloc(p, strlen(p) + strlen(line) + 1);
25             strcat(p, line);
26             if (Tcl_CommandComplete(p)) {
27                 break;
28             }
29             printf ("prompt2: ");
30         }
31
32         /* Arrive here when all the lines of a command have been read. */
33         code = Tcl_Eval(interp, p);
34         if (interp->result[0] != '\0') {
35             if (code == TCL_OK) {
36                 printf ("%s\n", interp->result);
37             } else {
38                 fprintf (stderr, "%s: %s\n", argv[0], interp->result);
39             }
40         }
41         free (p);
42     }
43
44     done:;
45     free (p);
46     Tcl_DeleteInterp(interp);
47     exit (code == TCL_OK ? EXIT_SUCCESS : EXIT_FAILURE);
48 }
```

```
1$ a.out

prompt1: puts hello
hello
```

```
prompt1: puts \
prompt2: supercalifragilisticexpialidocious
supercalifragilisticexpialidocious

prompt1: puts
wrong # args: should be "puts" ?-nonewline? ?fileId? string

prompt1: expr 1 + 2
3

prompt1: for {set i 1} {$i <= 3} {incr i} {
prompt2:     puts $i
prompt2: }
1
2
3

prompt1: puts \
prompt2: control-d
2$
```

Dynamically allocated strings: Ousterhout, pp. 345−348

If the functions **malloc**, **realloc**, and **free** give you the creeps, you can use the following utility functions instead. I underlined the new statements.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <tcl.h>
 5
 6 main()
 7 {
 8     Tcl_Interp *interp = Tcl_CreateInterp();
 9     char line[200];  /* buffer to hold each line of input */
10     Tcl_DString s;
11 int     code;
12
13 /*     Execute one Tcl command during each iteration of this loop. */
14     for (;;) {
15         printf ("\nprompt1: ");
16         Tcl_DStringInit (&s);
17
18 /*         Loop to read all the lines of the command. */
19         for (;;) {
20             if (fgets(line, sizeof line, stdin) == NULL) {
21                 goto done;
22             }
23             Tcl_DStringAppend (&s, line, -1);
24
25             if (Tcl_CommandComplete(s)) {
26                 break;
27             }
28             printf ("prompt2: ");
29         }
```



```
30
31          /* Arrive here when all the lines of command have been read. */
32          code = Tcl_Eval(interp, Tcl_DStringValue(&s));
33          if (interp->result[0] != '\0') {
34              if (code == TCL_OK) {
35                  printf ("%s\n", interp->result);
36              } else {
37                  fprintf (stderr, "%s: %s\n", argv[0], interp->result);
38              }
39          }
40          Tcl_DStringFree (&s);
41      }
42
43      done:;
44      Tcl_DeleteInterp(interp);
45      exit (0);
46  }
```

Access Tcl variables and arrays in C: Ousterhout pp. 325−336

To set the Tcl variable **$x** and the array element **$a(3)**,

```
1      Tcl_SetVar (interp, "x", "10", 0);
2      Tcl_SetVar2(interp, "a", "3", "20", 0);
3
4      Tcl_UnsetVar (interp, "x", 0);
5      Tcl_UnsetVar2(interp, "a", "3", 0);
6      Tcl_UnsetVar2(interp, "a", NULL, 0);  /* the whole array */
```

To get the value of a Tcl variable or array element,

```
1      printf ("$x == \"%s\"\n", Tcl_GetVar(interp, "x", "10", 0));
2      printf ("$a(3) == \"%s\"\n", Tcl_GetVar2(interp, "a", "3", "20", 0));
```

Let's make the above program initialize the Tcl variables **$argv0**, **$argc**, and **$argv** (Ousterhout, p. 48):

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <tcl.h>
 5
 6  main (int argc, char **argv)
 7  {
 8      Tcl_Interp *interp = Tcl_CreateInterp();
 9      char line[200];  /* buffer to hold each line of input */
10      int i;
11      Tcl_DString p;
12      int code;
13
14      Tcl_SetVar(interp, "argv0", argv[0], 0);
15
16      sprintf (line, "%d", argc - 1);
17      Tcl_SetVar(interp, "argc", line, 0);
18
19      Tcl_SetVar(interp, "argv", "", 0);
20      for (i = 1; i < argc; ++i) {
```



```
21          Tcl_SetVar(interp, "argv", argv[i], TCL_APPEND_VALUE | TCL_LIST_ELEMENT);
22      }
23
24      /* Execute one Tcl command during each iteration of this loop. */
25      for (;;) {
26          printf ("\nprompt1: ");
27          /* The rest is the same as above. */
```

```
1$ a.out moe 'larry curly'

prompt1: set argv0
a.out

prompt1: set argc
2

prompt1: set argv
moe {larry curly}
```

A Tcl interface to the Unix mail command

The Unix **`mail`** command has a non-programmable interface consisting of the commands **h**, **p** *n* , **s** *n filename* , **d** *n* , and **q**. To make it programmable, implement the following Tcl commands in C:

`headers`: return a list of the headers
`body` *n* : return the body of letter number *n* .
d *n* : delete letter number *n* .
q: quit from **`mail`**.

Then implement the following additional commands in Tcl. (Error checking removed for brevity).

```
 1 #Print the headers of all the letters on the screen, one per line.
 2 proc h {} {
 3     foreach header [headers] {
 4         puts $headers
 5     }
 6 }
 7
 8 #Print a letter on the screen.
 9 proc p {n} {
10     puts [body $n]
11 }
12
13 #Save a letter in a file.
14 proc s {n filename} {
15     set id [open $filename w]
16     puts -nonewline $id [body $n]
17     close $id
18 }
19
20 #Return the number of letters.
21 proc number {} {
22     return [llength [headers]]
23 }
```

You have now provided the user with the familiar interface to **`mail`** consisting of the commands **h**, **p** *n* , **s** *n filename* , **d** *n* , and **q**. But the user will now also be able to use the variables and control structure of

Tcl to program **`mail`** to do things it couldn't have done before:

```
 1 #Automatically delete every letter from that idiot.
 2 proc delete_idiot {} {
 3     set n 1
 4     foreach header [headers] {
 5         if {[string compare [lindex $header 2] "idiot"] == 0} {
 6             d $n
 7         }
 8     }
 9 }
10
11 #If I have received two or more copies of the same letter,
12 #delete all but the first.
13 proc delete_dupes {} {
14     set n [number]
15
16     for {set i 1} {$i <= $n} {incr i} {
17         for {set j [expr $i + 1]} {$j <= $n} {incr j} {
18             if {[string compare [body $i] [body $j]] == 0} {
19                 d $j
20             }
21         }
22     }
23 }
```

□