Part III:

Writing Tcl Applications in C

Chapter 27
Philosophy

This part of the book describes how to write C applications based on Tcl. Since the Tcl interpreter is implemented as a C library package, it can be linked into any C or C++ program. The enclosing application invokes procedures in the Tcl library to create interpreters, evaluate Tcl scripts, and extend the built-in command set with new application-specific commands. Tcl also provides a number of utility procedures for use in implementing new commands; these procedures can be used to access Tcl variables, parse arguments, manipulate Tcl lists, evaluate Tcl expressions, and so on. This chapter discusses several high-level issues to consider when designing a Tcl application, such as what new Tcl commands to implement, how to name objects, and what form to use for command results. The following chapters present the specific C interfaces provided by the Tcl library.

Note: *The interfaces described in Part III are those that will be available in Tcl 7.0, which had not been released at the timex this draft was prepared. Thus there may some differences between what you read here and what you can do with your current version of Tcl. There are almost no differences in functionality; the differences mostly have to do with the interfaces. Be sure to consult your manual entries when you actually write C code.*

27.1 C vs. Tcl: primitives

In order to make a Tcl application as flexible and powerful as possible, you should organize its C code as a set of new Tcl commands that provide a clean set of *primitive operations*. You need not implement every imaginable feature in C, since new features can always be implemented later as Tcl scripts. The purpose of the C code is to provide basic operations that make it easy to implement a wide variety of useful scripts. If your C code lumps several functions together into a single command then it won't be possible to write scripts that use the functions separately and your application won't be very flexible or extensible. Instead, each command should provide a single function, and you should combine them together with Tcl scripts. You'll probably find that many of your application's essential features are implemented as scripts.

Given a choice between implementing a particular piece of functionality as a Tcl script or as C code, it's generally better to implement it as a script. Scripts are usually easier to write, they can be modified dynamically, and you can debug them more quickly because you don't have to recompile after each bug fix. However, there are three reasons why it is sometimes better to implement a new function in C. First, you may need to access low-level machine facilities that aren't accessible in Tcl scripts. For example, the Tcl built-in commands don't provide access to network sockets, so if you want to use the network you'll have to write C code to do it. Second, you may be concerned about efficiency. For example, if you need to carry out intensive numerical calculations, or if you need to operate on large arrays of data, you'll be able to do it more efficiently in C than in Tcl. The third reason for implementing in C is complexity. If you are manipulating complex data structures, or if you're writing a large amount of code, the task will probably be more manageable in C than in Tcl. Tcl provides very little structure; this makes it easy to connect different things together but hard to manage large complex scripts. C provides more structure, which is cumbersome when you're implementing small things but indispensable when you're implementing big complicated things.

As an example, consider a program to manipulate weather reports. Suppose that information about current weather is available for a large number of measurement stations from one or more network sites using a well-defined network protocol, and you want to write a Tcl application to manipulate this data. Users of your application might wish to answer questions like:

- What is the complete weather situation at station X?
- What is the current temperature at station X?
- Which station in the country has the highest current temperature?
- At which stations is it currently raining?

You'll need to write some C code for this application in order to retrieve weather reports over the network. What form should these new commands take?

One approach is to implement each of the above functions in C as a separate Tcl command. For example, you might provide a command that retrieves the weather report from a station, formats it into prose, and prints it on standard output. Unfortunately this command can only be used for one purpose; you'd have to provide a second command for situations where you want to retrieve a report without printing it out (e.g. to find all the station where it is raining).

Instead, I'd suggest providing just two commands in C: a `wthr_stations` command that returns a list of all the stations for which weather reports are available, and a `wthr_report` command that returns a complete weather report for a particular station. These commands don't implement any of the above features directly, but they make it easy to implement all of the features. For example, Tcl already has a `puts` command that can be used to print information on standard output, so the first feature (printing a weather report for a station) can be implemented with a script that calls `wthr_report`, formats the report, and prints it with `puts`. The second feature (printing just the temperature) can be implemented by extracing the temperature from the result of `wthr_report` and then printing it alone. The third and fourth features (finding the hottest station and finding all stations where it is raining) can be implemented with scripts that invoke `wthr_report` for each station and extract and print relevant information. Many other features could also be implemented, such as printing a sorted list of the ten stations with the highest temperatures.

The preceding paragraph suggests that lower-level commands are better than higher-level ones. However, if you make the commands too low level then Tcl scripts will become unnecessarily complicated and you may lose opportunities for efficient implementation. For example, instead of providing a single command that retrieves a weather report, you might provide separate Tcl commands for each step of the protocol that retrieves a report: one command to connect to a server, one command to select a particular station, one command to request a report for the selected station, and so on. Although this results in more primitive commands, it is probably a mistake. The extra commands don't provide any additional functionality and they make it more tedious to write Tcl scripts. Furthermore, suppose that network communication delays are high, so that it takes a long time to get a response from a weather server, but the server allows you to request reports for several stations at once and get them all back in about the same time as a single report. In this situation you might want an even higher level interface, perhaps a Tcl command that takes any number of stations as arguments and retrieves reports for all of them at once. This would allow the C code to amortize the communication delays across several report retrievals and it might permit a much more efficient implementation of operations such as finding the station with the highest temperature.

To summarize, you should pick commands that are primitive enough so that all of the application's key functions are available individually through Tcl commands. On the other hand, you should pick commands that are high-level enough to hide unimportant details and capitalize on opportunities for efficient implementation.

27.2 Object names

The easiest way to think about your C code is in terms of *objects*. The C code in a Tcl application typically implements a few new kinds of objects, which are manipulated by the application's new Tcl commands. In the C code of your application you'll probably refer to the objects using pointers to the C structures that represent the objects, but you can't use pointers in Tcl scripts. Strings of some sort will have to be used in the Tcl scripts, and the C code that implements your commands will have to translate from those strings to internal pointers. For example, the objects in the weather application are weather stations; the `wthr_stations` command returns a list of station names, and the `wthr_report` command takes a station name as an argument.

A simple but dangerous way to name objects is to use their internal addresses. For example, in the weather application you could name each station with a hexadecimal string giving the internal address of the C structure for that station: the command that returns a list of stations would return a list of hexadecimal strings, and the command to retrieve a weather report would take one of these hexadecimal strings as argument. When the C code receives one of these strings, it could produce a pointer by converting the string to a binary number. I don't recommend using this approach in practice because it is hard to verify that a hexadecimal string refers to a valid object. If a user specifies a bad address it might cause the C code to make wild memory accesses, which could cause the application to crash. In addition, hexadecimal strings don't convey any meaningful information to the user.

Instead, I recommend using names that can be verified and that convey meaningful information. One simple approach is to keep a hash table in your C code that maps from a string name to the internal pointer for the object; a name is only valid if it appears in the hash table. The Tcl library implements flexible hash tables to make it easy for you to use this approach (see Chapter 33). If you use a hash table then you can use any strings whatsoever for names, so you might as well pick ones that convey information. For example, Tk uses hierarchical path names like `.menu.help` for windows in order to indicate the window's position in the window hierarchy. Tcl uses names like `file3` or `file4` for open files; these names don't convey a lot of information, but they at least include the letters "`file`" to suggest that they're used for file access, and the number is the POSIX file descriptor number for the open file. For the weather application I'd recommend using station names such as the city where the station is located. Or, if the U.S. Weather Service has well-defined names for its stations then I'd suggest using those names.

27.3 Commands: action-oriented vs. object-oriented

There are two approaches you can use when defining commands in your application, which I call *action-oriented* and *object-oriented*. In the action-oriented approach there is one command for each action that can be taken on an object, and the command takes an object name as an argument. The weather application is action-oriented: the `wthr_report` command corresponds to an action (retrieve weather report) and it takes a weather station name as an argument. Tcl's file commands are also action-oriented: there are separate commands for opening files, reading, writing, closing, etc.

In the object-oriented approach there is one command for each object, and the name of the command is the name of the object. When the command is invoked its first argument specifies the operation to perform on the object. Tk's widgets work this way: if there is a button widget `.b` then there is also a command named `.b`; you can invoke "`.b flash`" to flash the widget or "`.b invoke`" to invoke its action.

The action-oriented approach is best when there are a great many objects or the objects are unpredictable or short-lived. For example, it wouldn't make sense to implement string operations using an object-oriented approach because there would have to be one command for each string, and in practice Tcl applications have large numbers of strings that are created and deleted on a command-by-command basis. The weather application uses the action-oriented approach because there are only a few actions and and potentially a large number of stations. In addition, the application probably doesn't need to keep around state for each station all the time; it just uses the station name to look up weather information when requested.

The object-oriented approach works well when the number of objects isn't too great (e.g. a few tens or hundreds) and the objects are well-defined and exist for at least moderate amounts of time. Tk's widgets fit this description. The object-oriented approach has the advantage that it doesn't pollute the command name space with lots of commands for individual actions. For example in the action-oriented approach the command "delete" might be defined for one kind of object, thereby preventing its use for any other kind of object. In the object-oriented approach you only have to make sure that your object names don't conflict with existing commands or other object names. For example, Tk claims all command names starting with "." for its widget commands. The object-oriented approach also makes it possible for different objects to implement the same action in different ways. For example, if `.t` is a text widget and `.l` is a listbox widget in Tk, the `commands` "`.t yview 0`" and "`.l yview 0`" are implemented in very different ways even though they produce the same logical effect (adjust the view to make the topmost line visible at the top of the window).

Note: *Although Tk's file commands are implemented using the action-oriented approach, in retrospect I wish that I had used the object-oriented fashion, since open files fit the object-oriented model nicely.*

27.4 Application prefixes

If you use the action-oriented approach, I strongly recommend that you add a unique prefix to each of your command names. For example, I used the prefix "`wthr_`" for the weather commands. This guarantees that your commands won't conflict with other commands as long as your prefix is unique, and it makes it possible to merge different applications together without name conflicts. I also recommend using prefixes for Tcl procedures that you define and for global variables, again so that multiple packages can be used together.

27.5 Representing information

The information passed into and out of your Tcl commands should be formatted for easy processing by Tcl scripts, not necessarily for maximum human readability. For example, the command that retrieves a weather report shouldn't return English prose describing the weather. Instead, it should return the information in a structured form that makes it easy to extract the different components under the control of a Tcl script. You might return the report as a list consisting of pairs of elements, where the first element of each pair is a keyword and the second element is a value associated with that keyword, such as:

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

This indicates that the current temperature at the station is 53 degrees, the high and low for the last 24 hours were 68 and 37 degrees, .02 inches of rain has fallen in the last 24 hours, and the sky is partly cloudy. Or, the command might store the report in an associative array where each keyword is used as the name of an array element and the corresponding value is stored in that element. Either of these approaches would make it easy to extract components of the report. You can always reformat the information to make it more readable just before displaying it to the user.

Although machine readability is more important than human readability, you need not gratuitously sacrifice readability. For example, the above list could have been encoded as

```
18 53 7 68 9 37 5 .02 17 4
```

where `18` is a keyword for current temperature, `7` for 24-hour high, and so on. This is unnecessarily confusing and will not make your scripts any more efficient, since Tcl handles strings at least as efficiently as numbers.

Chapter 28
Interpreters and Script Evaluation

This chapter describes how to create and delete interpreters and how to use them to evaluate Tcl scripts. Table 28.1 summarizes the library procedures that are discussed in the chapter.

28.1 Interpreters

The central data structure manipulated by the Tcl library is a C structure of type `Tcl_Interp`. I'll refer to these structures (or pointers to them) as *interpreters*. Almost all of the Tcl library procedures take a pointer to a `Tcl_Interp` structure as an argument. An interpreter embodies the execution state of a Tcl script, including commands implemented in C, Tcl procedures, variables, and an execution stack that reflects partially-evaluated commands and Tcl procedures. Most Tcl applications use only a single interpreter but it is possible for a single process to manage several independent interpreters.

28.2 A simple Tcl application

The program below illustrates how to create and use an interpreter. It is a simple but complete Tcl application that evaluates a Tcl script stored in a file and prints the result or error message, if any.

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

`Tcl_Interp *Tcl_CreateInterp(void)`	Create a new interpreter and return a token for it.
`Tcl_DeleteInterp(Tcl_Interp *interp)`	Delete an interpreter.
`int Tcl_Eval(Tcl_Interp *interp, char *script)`	Evaluate `script` in `interp` and return its completion code. The result or error string will be in `interp->result`.
`int Tcl_EvalFile(Tcl_Interp *interp, char *fileName)`	Evaluate the contents of file `fileName` in `interp` and return its completion code. The result or error string will be in `interp->result`.
`int Tcl_GlobalEval(Tcl_Interp *interp, char *script)`	Evaluate `script` in `interp` at global level and return its completion code. The result or error string will be in `interp->result`.
`int Tcl_VarEval(Tcl_Interp *interp, char *string, char *string, ... (char *) NULL)`	Concatenate all of the `string` arguments into a single string, evaluate the resulting script in `interp`, and return its completion code. The result or error string will be in `interp->result`.
`int Tcl_RecordAndEval(Tcl_Interp *interp, char *script, int flags)`	Records `script` as an event in `interp`'s history list and evaluates it if `eval` is non-zero (`TCL_NO_EVAL` means don't evaluate the script). Returns a completion code such as `TCL_OK` and leaves result or error message in `interp->result`.

Table 28.1. Tcl library procedures for creating and deleting interpreters and for evaluating Tcl

```
main(int argc, char *argv[]) {
    Tcl_Interp *interp;
    int code;

    if (argc != 2) {
        fprintf(stderr, "Wrong # arguments: ");
        fprintf("should be \"%s fileName\"\n",
                argv[0]);
        exit(1);
    }

    interp = Tcl_CreateInterp();
    code = Tcl_EvalFile(interp, argv[1]);
    if (*interp->result != 0) {
        printf("%s\n", interp->result);
    }
    if (code != TCL_OK) {
```

```
        exit(1);
    }
    exit(0);
}
```

If Tcl has been installed properly at your site you can copy the C code into a file named `simple.c` and compile it with the following shell command:

```
cc simple.c -ltcl -lm
```

Once you've compiled the program you can evaluate a script file `test.tcl` by typing the following command to your shell:

```
a.out test.tcl
```

The code for `simple.c` starts out with `#include` statements for `stdio.h` and `tcl.h`. You'll need to include `tcl.h` in every file that uses Tcl structures or procedures, since it defines structures like `Tcl_Interp` and declares the Tcl library procedures.

After checking to be sure that a file name was specified on the command line, the program invokes `Tcl_CreateInterp` to create a new interpreter. The new interpreter will contain all of the built-in commands described in Part I but no Tcl procedures or variables. It will have an empty execution stack. `Tcl_CreateInterp` returns a pointer to the `Tcl_Interp` structure for the interpreter, which is used as a token for the interpreter when calling other Tcl procedures. Most of the fields of the `Tcl_Interp` structure are hidden so that they cannot be accessed outside the Tcl library. The only accessible fields are those that describe the result of the last script evaluation; they'll be discussed later.

Next `simple.c` calls `Tcl_EvalFile` with the interpreter and the name of the script file as arguments. `Tcl_EvalFile` reads the file and evaluates its contents as a Tcl script, just as if you had invoked the Tcl `source` command with the file name as an argument. When `Tcl_EvalFile` returns the execution stack for the interpreter will once again be empty.

`Tcl_EvalFile` returns two pieces of information: an integer *completion code* and a string. The completion code is returned as the result of the procedure. It will be either `TCL_OK`, which means that the script completed normally, or `TCL_ERROR`, which means that an error of some sort occurred (e.g. the script file couldn't be read or the script aborted with an error). The second piece of information returned by `Tcl_EvalFile` is a string, a pointer to which is returned in `interp->result`. If the completion code is `TCL_OK` then `interp->result` points to the script's result; if the completion code is `TCL_ERROR` then `interp->result` points to a message describing the error.

Note: The result string belongs to Tcl. It may or may not be dynamically allocated. You can read it and copy it, but you should not modify it and you should not save pointers to it. Tcl may overwrite the string or reallocate its memory during the next call to `Tcl_EvalFile` *or any of the other procedures that evaluate scripts. Chapter 29 discusses the result string in more detail.*

If the result string is non-empty then `simple.c` prints it, regardless of whether it is an error message or a normal result. Then the program exits. It follows the UNIX style of exiting with a status of 1 if an error occurred and 0 if it completed successfully.

When the script file is evaluated only the built-in Tcl commands are available: no Tk commands will be available in this application and no application-specific commands have been defined.

28.3 Other evaluation procedures

Tcl provides three other procedures besides `Tcl_EvalFile` for evaluating scripts. Each of these procedures takes an interpreter as its first argument and each returns a completion code and string, just like `Tcl_EvalFile`. `Tcl_Eval` is similar to `Tcl_EvalFile` except that its second argument is a Tcl script rather than a file name:

```
code = Tcl_Eval(interp, "set a 44");
```

`Tcl_VarEval` takes a variable number of string arguments terminated with a `NULL` argument. It concatenates the strings and evaluates the result as a Tcl script. For example, the statement below has the same effect as the one above:

```
code = Tcl_VarEval(interp, "set a ", "44",
        (char *) NULL);
```

`Tcl_GlobalEval` is similar to `Tcl_Eval` except that it evaluates the script at global variable context (as if the execution stack were empty) even when procedures are active. It is used in special cases such as the `uplevel` command and Tk's event bindings.

If you want a script to be recorded on the Tcl history list, call `Tcl_RecordAndEval` instead of `Tcl_Eval`:

```
char *script;
int code;
...
code = Tcl_RecordAndEval(interp, script, 0);
```

`Tcl_RecordAndEval` is identical to `Tcl_Eval` except that it records the script as a new entry on the history list before invoking it. Tcl only records the scripts passed to `Tcl_RecordAndEval`, so you can select which ones to record. Typically you'll record only commands that were typed interactively. The last argument to `Tcl_RecordAndEval` is normally `0`; if you specify `TCL_NO_EVAL` instead, then Tcl will record the script without actually evaluating it.

28.4 Deleting interpreters

The procedure `Tcl_DeleteInterp` may be called to destroy an interpreter and all its associated state. It is invoked with an interpreter as argument:

```
Tcl_DeleteInterp(interp);
```

Once `Tcl_DeleteInterp` returns you should never use the interpreter again. In applications like `simple.c`, which use a single interpreter throughout their lifetime, there's no need to delete the interpreter.

Chapter 29
Creating New Tcl Commands

Each Tcl command is represented by a *command procedure* written in C. When the command is invoked during script evaluation, Tcl calls its command procedure to carry out the command. This chapter provides basic information on how to write command procedures, how to register command procedures in an interpreter , and how to manage the interpreter's result string. Table 29.1 summarizes the Tcl library procedures that are discussed in the chapter.

29.1 Command procedures

The interface to a command procedure is defined by the `Tcl_CmdProc` procedure prototype:

```
typedef int Tcl_CmdProc(ClientData clientData,
        Tcl_Interp *interp, int argc,
        char *argv[]);
```

Each command procedure takes four arguments. The first, `clientData`, will be discussed in Section 29.5 below. The second, `interp`, is the interpreter in which the command was invoked. The third and fourth arguments have the same meaning as the `argc` and `argv` arguments to a C main program: `argc` specifies the total number of words in the Tcl command and `argv` is an array of pointers to the values of the words. Tcl processes all the special characters such as `$` and `[]` before invoking command procedures, so the values in `argc` reflect any substitutions that were specified for the command. The command name is included in `argc` and `argv`, and `argv[argc]` is `NULL`. A command

`Tcl_CreateCommand(Tcl_Interp *interp, char *cmdName,`
`    Tcl_CmdProc *cmdProc, ClientData clientData,`
`    Tcl_CmdDeleteProc *deleteProc)`	

	Defines a new command in `interp` with name `cmdName`. When the command is invoked `cmdProc` will be called; if the command is ever deleted then `deleteProc` will be called.
`int Tcl_DeleteCommand(Tcl_Interp *interp, char *cmdName)`	
	If `cmdName` is a command or procedure in `interp` then deletes it and returns 0. Otherwise returns -1.
`Tcl_SetResult(Tcl_Interp *interp, char *string,Tcl_FreeProc *freeProc)`	
	Arrange for `string` (or a copy of it) to become the result for `interp`. `FreeProc` identifies a procedure to call to eventually free the result, or it may be `TCL_STATIC`, `TCL_DYNAMIC`, or `TCL_VOLATILE`.
`Tcl_AppendResult(Tcl_Interp *interp, char *string,` `    char *string, ... (char *) NULL)`	
	Appends each of the `string` arguments to the result string in `interp`.
`Tcl_AppendElement(Tcl_Interp *interp, char *string)`	
	Formats `string` as a Tcl list element and appends it to the result string in `interp`, with a preceding separator space if needed.
`Tcl_ResetResult(Tcl_Interp *interp)`	
	Resets `interp`'s result to the default empty state, freeing up any dynamically-allocated memory associated with it.

Table 29.1. Tcl library procedures for creating and deleting commands and for manipulating the

procedure returns two values just like `Tcl_Eval` and `Tcl_EvalFile`. It returns an integer completion code as its result (e.g. `TCL_OK` or `TCL_ERROR`) and it leaves a result string or error message in `interp->result`.

Here is the command procedure for a new command called `eq` that compares its two arguments for equality:

```
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;
}
```

EqCmd checks to see that was called with exactly two arguments (three words, including the command name), and if not it stores an error message in interp->result and returns TCL_ERROR. Otherwise it compares its two argument strings and stores a string in interp->result to indicate whether or not they were equal; then it returns TCL_OK to indicate that the command completed normally.

29.2 Registering commands

In order for a command procedure to be invoked by Tcl, you must register it by calling Tcl_CreateCommand. For example, EqCmd could be registered with the following statement:

```
Tcl_CreateCommand(interp, "eq", EqCmd,
        (ClientData *) NULL,
        (Tcl_CmdDeleteProc *) NULL);
```

The first argument to Tcl_CreateCommand identifies the interpreter in which the command will be used. The second argument specifies the name for the command and the third argument specifies its command procedure. The fourth and fifth arguments are discussed in Section 29.5 below; they can be specified as NULL for simple commands like this one. Tcl_CreateCommand will create a new command for interp named eq; if there already existed a command by that name then it is deleted. Whenever eq is invoked in interp Tcl will call EqCmd to carry out its function.

After the above call to Tcl_CreateCommand, eq can be used in scripts just like any other command:

```
eq abc def
0
eq 1 1
1
set w .dlg
set w2 .dlg.ok
eq $w.ok $w2
1
```

When processing scripts, Tcl carries out all of the command-line substitutions before calling the command procedure, so when EqCmd is called for the last eq command above both argv[1] and argv[2] are ".dlg.ok".

Tcl_CreateCommand is usually called by applications during initialization to register application-specific commands. However, new commands can also be created at any time while an application is running. For example, the proc command creates a new command for each Tcl procedure that is defined, and Tk creates a widget command for each new widget. In Section 29.5 you'll see an example where the command procedure for one command creates a new command.

Commands created by `Tcl_CreateCommand` are indistinguishable from Tcl's built-in commands. Each built-in command has a command procedure with the same form as EqCmd, and you can redefine a built-in command by calling `Tcl_CreateCommand` with the name of the command and a new command procedure.

29.3 The result protocol

The `EqCmd` procedure returns a result by setting `interp->result` to point to one of several static strings. However, the result string can also be managed in several other ways. Tcl defines a protocol for setting and using the result, which allows for dynamically-allocated results and provides a small static area to avoid memory-allocation overheads in simple cases.

The full definition of the `Tcl_Interp` structure, as visible outside the Tcl library, is as follows:

```
typedef struct Tcl_Interp {
    char *result;
    Tcl_FreeProc *freeProc;
    int errorLine;
} Tcl_Interp;
```

The first field, `result`, points to the interpreter's current result. The second field, `freeProc`, is used when freeing dynamically-allocated results; it will be discussed below. The third field, `errorLine`, is related to error handling and is described in Section XXX.

When Tcl invokes a command procedure the `result` and `freeProc` fields always have well-defined values. `Interp->result` points to a small character array that is part of the interpreter structure and the array has been initialized to hold an empty string (the first character of the array is zero). `Interp->freeProc` is always zero. This state is referred to as the *initialized state* for the result. Not only is this the state of the result when command procedures are invoked, but many Tcl library procedures also expect the interpreter's result to be in the initialized state when they are invoked. If a command procedure wishes to return an empty string as its result, it simply returns without modifying `interp->result` or `interp->freeProc`.

There are three ways that a command procedure can specify a non-empty result. First, it can modify `interp->result` to point to a static string as in `EqCmd`. A string can be considered to be static as long as its value will not change before the next Tcl command procedure is invoked. For example, Tk stores the name of each widget in a dynamically-allocated record associated with the widget, and it returns widget names by setting `interp->result` to the name string in the widget record. This string is dynamically allocated, but widgets are deleted by Tcl commands so the string is guaranteed not to be recycled before the next Tcl command executes. If a string is stored in automatic storage associated with a procedure it cannot be treated as static, since its value will change as soon as some other procedure re-uses the stack space.

The second way to set a result is to use the pre-allocated space in the `Tcl_Interp` structure. In its initialized state `interp->result` points to this space. If a command procedure wishes to return a small result it can copy it to the location pointed to by `interp->result`. For example, the procedure below implements a command `numwords` that returns a decimal string giving a count of its arguments:

```
int NumwordsCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    sprintf(interp->result, "%d", argc);
    return TCL_OK;
}
```

The size of the pre-allocated space is guaranteed to be at least 200 bytes; you can retrieve the exact size with the symbol `TCL_RESULT_SIZE` defined by `tcl.h`. It's generally safe to use this area for printing a few numbers and/or short strings, but it is *not* safe to copy strings of unbounded length to the pre-allocated space.

The third way to set a result is to allocate memory with a storage allocator such as `malloc`, store the result string there, and set `interp->result` to the address of the memory. In order to ensure that the memory is eventually freed, you must also set `interp->freeProc` to the address of a procedure that Tcl can call to free the memory, such as `free`. In this case the dynamically-allocated memory becomes the property of Tcl. Once Tcl has finished using the result it will free it by invoking the procedure specified by `interp->freeProc`. This procedure must match the following procedure prototype:

```
typedef void Tcl_FreeProc(char *blockPtr);
```

The procedure will be invoked with a single argument containing the address that you stored in `interp->result`. In most cases you'll use `malloc` for dynamic allocation and thus set `interp->freeProc` to `free`, but the mechanism is general enough to support other storage allocators too.

29.4 Procedures for managing the result

Tcl provides several library procedures for manipulating the result. These procedures all obey the protocol described in the previous section, and you may find them more convenient than setting `interp->result` and `interp->freeProc` directly. The first procedure is `Tcl_SetResult`, which simply implements the protocol described above. For example, `EqCmd` could have replaced the statement

```
interp->result = "wrong # args";
```

with a call to Tcl_SetResult as follows:

```
Tcl_SetResult(interp, "wrong # args", TCL_STATIC);
```

The first argument to Tcl_SetResult is an interpreter. The second argument is a string to use as result, and the third argument gives additional information about the string. TCL_STATIC means that the string is static, so Tcl_SetResult just stores its address into interp->result. A value of TCL_VOLATILE for the third argument means that the string is about to change (e.g. it's stored in the procedure's stack frame) so a copy must be made for the result. Tcl_SetResult will copy the string into the pre-allocated space if it fits, otherwise it will allocate new memory to use for the result and copy the string there (setting interp->freeProc appropriately). If the third argument is TCL_DYNAMIC it means that the string was allocated with malloc and is now the property of Tcl: Tcl_SetResult will set interp->freeProc to free as described above. Finally, the third argument may be the address of a procedure suitable for use in interp->freeProc; in this case the string is dynamically-allocated and Tcl will eventually call the specified procedure to free it.

Tcl_AppendResult makes it easy to build up results in pieces. It takes any number of strings as arguments and appends them to the interpreter's result in order. As the result grows in length Tcl_AppendResult allocates new memory for it. Tcl_AppendResult may be called repeatedly to build up long results incrementally, and it does this efficiently even if the result becomes very large (e.g. it allocates extra memory so that it doesn't have to copy the existing result into a larger area on each call). Here is an implementation of the concat command that uses Tcl_AppendResult:

```
int ConcatCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    int i;
    if (argc == 1) {
        return TCL_OK;
    }
    Tcl_AppendResult(interp, argv[1], (char *) NULL);
    for (i = 2; i < argc; i++) {
        Tcl_AppendResult(interp, " ", argv[i],
            (char *) NULL);
    }
    return TCL_OK;
}
```

The NULL argument in each call to Tcl_AppendResult marks the end of the strings to append. Since the result is initially empty, the first call to Tcl_AppendResult just sets the result to argv[1]; each additional call appends one more argument preceded by a separator space.

Tcl_AppendElement is similar to Tcl_AppendResult except that it only adds one string to the result at a time and it appends it as a list element instead of a raw string. It's useful for creating lists. For example, here is a simple implementation of the `list` command:

```
int ListCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char **argv) {
    int i;
    for (i = 1; i < argc; i++) {
        Tcl_AppendElement(interp, argv[i]);
    }
    return TCL_OK;
}
```

Each call to `Tcl_AppendElement` adds one argument to the result. The argument is converted to a proper list element before appending it to the result (e.g. it is enclosed in braces if it contains space characters). `Tcl_AppendElement` also adds a separator space if it's needed before the new element (no space is added if the result is currently empty or if its characters are " {", which means that the new element will be the first element of a sub-list). For example, if `ListCmd` is invoked with four arguments, "`list`", "`abc`", "`x y`", and "`}`", it produces the following result:

```
abc {x y} \}
```

Like `Tcl_AppendResult`, `Tcl_AppendElement` grows the result space if needed and does it in a way that is efficient even for large results and repeated calls.

If you set the result for an interpreter and then decide that you want to discard it (e.g. because an error has occurred and you want to replace the current result with an error message), you should call the procedure `Tcl_ResetResult`. It will invoke `interp->freeProc` if needed and then restore the interpreter's result to its initialized state. You can then store a new value in the result in any of the usual ways. You need not call `Tcl_ResetResult` if you're going to use `Tcl_SetResult` to store the new result, since `Tcl_SetResult` takes care of freeing any existing result.

29.5 ClientData and deletion callbacks

The fourth and fifth arguments to `Tcl_CreateCommand`, `clientData` and `deleteProc`, were not discussed in Section 29.2 but they are useful when commands are associated with objects. The `clientData` argument is used to pass a one-word value to a command procedure. Tcl saves the `clientData` value that is passed to `Tcl_CreateCommand` and uses it as the first argument to the command procedure. The type `ClientData` is large enough to hold either an integer or a pointer value. It is usually the address of a C data structure for the command to manipulate.

Tcl and Tk use *callback procedures* in many places. A callback is a procedure whose address is passed to a library procedure and saved in a data structure. Later, at some significant time, the address is used to invoke the procedure ("call it back"). A command procedure is an example of a callback: Tcl associates the procedure address with a Tcl command name and calls the procedure whenever the command is invoked. When a callback is specified in Tcl or Tk a `ClientData` argument is usually provided along with the procedure address and the `ClientData` value is passed to the callback as its first argument.

The `deleteProc` argument to `Tcl_CreateCommand` specifies a deletion callback. If its value isn't `NULL` then it is the address of a procedure for Tcl to invoke when the command is deleted. The procedure must match the following prototype:

```
typedef void Tcl_CmdDeleteProc(ClientData clientData);
```

The deletion callback takes a single argument, which is the ClientData value specified when the command was created. Deletion callbacks are used for purposes such as freeing the object associated with a command.

Figure 29.1 shows how `clientData` and `deleteProc` can be used to implement counter objects. The application containing this code must register `CounterCmd` as a Tcl command using the following call:

```
Tcl_CreateCommand(interp, "counter", CounterCmd,
        (ClientData) NULL, (Tcl_CmdDeleteProc) NULL);
```

New counters can then be created by invoking the `counter` Tcl command; each invocation creates a new object and returns a name for that object:

```
counter
ctr0
counter
ctr1
```

`CounterCmd` is the command procedure for `counter`. It allocates a structure for the new counter and initializes its value to zero. Then it creates a name for the counter using the static variable `id`, arranges for that name to be returned as the command's result, and increments `id` so that the next new counter will get a different name.

This example uses the object-oriented style described in Section 27.3, where there is one command for each counter object. As part of creating a new counter `CounterCmd` creates a new Tcl command named after the counter. It uses the address of the `Counter` structure as the `ClientData` for the command and specifies `DeleteCounter` as the deletion callback for the new command.

Counters can be manipulated by invoking the commands named after them. Each counter supports two options to its command: `get`, which returns the current value of the counter, and `next`, which increments the counter's value. Once `ctr0` and `ctr1` were created above, the following Tcl commands could be invoked:

```
ctr0 next; ctr0 next; ctr0 get
2
ctr1 get
0
```

```
typedef struct {
    int value;
} Counter;

int CounterCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    Counter *counterPtr;
    static int id = 0;
    if (argc != 1) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    counterPtr = (Counter *) malloc(sizeof(Counter));
    counterPtr->value = 0;
    sprintf(interp->result, "ctr%d", id);
    id++;
    Tcl_CreateCommand(interp, interp->result, ObjectCmd,
            (ClientData) counterPtr, DeleteCounter);
    return TCL_OK;
}

int ObjectCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    CounterPtr *counterPtr = (Counter *) clientData;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (strcmp(argv[1], "get") == 0) {
        sprintf(interp->result, "%d", counterPtr->value);
    } else if (strcmp(argv[1], "next") == 0) {
        counterPtr->value++;
    } else {
        Tcl_AppendResult(interp, "bad counter command \"",
                argv[1], "\": should be get or next",
                (char *) NULL);
        return TCL_ERROR;
    }
    return TCL_OK;
}

void DeleteCounter(ClientData clientData) {
    free((char *) clientData);
}
```

Figure 29.1. An implementation of counter objects.

```
ctr0 clear
bad counter command "clear": should be get or next
```

The procedure `ObjectCmd` implements the Tcl commands for all existing counters. It is passed a different `ClientData` argument for each counter, which it casts back to a value of type `Counter *`. `ObjectCmd` then checks `argv[1]` to see which command option was invoked. If it was `get` then it returns the counter's value as a decimal string; if it was `next` then it increments the counter's value and leaves `interp->result` untouched so that the result is an empty string. If an unknown command was invoked then `ObjectCmd` calls `Tcl_AppendResult` to create a useful error message.

Note: It is not safe to create the error message with a statement like

```
sprintf(interp->result, "bad counter command \"%s\": "
    "should be get or next", argv[1]);
```

This is unsafe because `argv[1]` *has unknown length. It could be so long that* `sprintf` *overflows the space allocated in the interpreter and corrupts memory .* `Tcl_AppendResult` *is safe because it checks the lengths of its arguments and allocates as much space as needed for the result.*

To destroy a counter you can delete its Tcl command, for example:

```
rename ctr0 {}
```

As part of deleting the command Tcl will invoke `DeleteProc`, which frees up the memory associated with the counter.

This object-oriented implementation of counter objects is similar to Tk's implementation of widgets: there is one Tcl command to create new instances of each counter or widget, and one Tcl command for each existing counter or widget. A single command procedure implements all of the counter or widget commands for a particular type of object, receiving a ClientData argument that identifies a specific counter or widget. A different mechanism is used to delete Tk widgets than for counters above, but in both cases the command corresponding to the object is deleted at the same time as the object.

29.6 Deleting commands

Tcl commands can be removed from an interpreter by calling `Tcl_DeleteCommand`. For example, the statement below will delete the `ctr0` command in the same way as the `rename` command above:

```
Tcl_DeleteCommand(interp, "ctr0");
```

If the command has a deletion callback then it will be invoked before the command is removed. Any command may be deleted, including built-in commands, application-specific commands, and Tcl procedures.

Chapter 30
Parsing

This chapter describes Tcl library procedures for parsing and evaluating strings in various forms such as integers, expressions and lists. These procedures are typically used by command procedures to process the words of Tcl commands. See Table 30.1 for a summary of the procedures.

30.1 Numbers and booleans

Tcl provides three procedures for parsing numbers and boolean values: `Tcl_GetInt`, `Tcl_GetDouble`, and `Tcl_GetBoolean`. Each of these procedures takes three arguments: an interpreter, a string, and a pointer to a place to store the value of the string. Each of the procedures returns `TCL_OK` or `TCL_ERROR` to indicate whether the string was parsed successfully. For example, the command procedure below uses `Tcl_GetInt` to implement a `sum` command:

```
int SumCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    int num1, num2;
    if (argc != 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, argv[1], &num1) != TCL_OK) {
        return TCL_ERROR;
    }
```

`int Tcl_GetInt(Tcl_Interp *interp, char *string, int *intPtr)` Parses `string` as an integer, stores value at `*intPtr`, and returns `TCL_OK`. If an error occurs while parsing, returns `TCL_ERROR` and stores an error message in `interp->result`. `int Tcl_GetDouble(Tcl_Interp *interp, char *string, double *doublePtr)` Same as `Tcl_GetInt` except parses `string` as a floating-point value and stores value at `*doublePtr`. `int Tcl_GetBoolean(Tcl_Interp *interp, char *string, int *intPtr)` Same as `Tcl_GetInt` except parses `string` as a boolean and stores 0/1 value at `*intPtr`. See Table 30.2 for legal values for `string`.	
`int Tcl_ExprString(Tcl_Interp *interp, char *string)` Evaluates `string` as an expression, stores value as string in `interp->result`, and returns `TCL_OK`. If an error occurs during evaluation, returns `TCL_ERROR` and stores an error message in `interp->result`. `int Tcl_ExprLong(Tcl_Interp *interp, char *string, long *longPtr)` Same as `Tcl_ExprString` except stores value as a long integer at `*longPtr`. An error occurs if the value can't be converted to an integer. `int Tcl_ExprDouble(Tcl_Interp *interp, char *string, double *doublePtr)` Same as `Tcl_ExprString` except stores value as double-precision floating-point value at `*doublePtr`. An error occurs if the value can't be converted to a floating-point number. `int Tcl_ExprBoolean(Tcl_Interp *interp, char *string, int *intPtr)` Same as `Tcl_ExprString` except stores value as 0/1 integer at `*intPtr`. An error occurs if the value can't be converted to a boolean value.	
`int Tcl_SplitList(Tcl_Interp *interp, char *list, int *argcPtr, char ***argvPtr)` Parses `list` as a Tcl list and creates an array of strings whose values are the elements of list. Stores count of number of list elements at `*argcPtr` and pointer to array at `*argvPtr`. Returns `TCL_OK`. If an error occurs while parsing `list`, returns `TCL_ERROR` and stores an error message in `interp->result`. Space for string array is dynamically allocated; caller must eventually pass `*argvPtr` to `free`. `char *Tcl_Merge(int argc, char **argv)` Inverse of `Tcl_SplitList`. Returns pointer to Tcl list whose elements are the members of `argv`. Result is dynamically-allocated; caller must eventually pass it to `free`.	

```
if (Tcl_GetInt(interp, argv[2], &num2) != TCL_OK) {
    return TCL_ERROR;
}
sprintf(interp->result, "%d", num1+num2);
```

```
    return TCL_OK;
}
```

SumCmd expects each of the command's two arguments to be an integer. It calls Tcl_GetInt to convert them from strings to integers, then it sums the values and converts the result back to a decimal string in interp->result. Tcl_GetInt accepts strings in decimal (e.g. "492"), hexadecimal (e.g. "0x1ae") or octal (e.g. "017"), and allows them to be signed and preceded by white space. If the string is in one of these formats then Tcl_GetInt returns TCL_OK and stores the value of the string in the location pointed to by its last argument. If the string can't be parsed correctly then Tcl_GetInt stores an error message in interp->result and returns TCL_ERROR; SumCmd then returns TCL_ERROR to its caller with interp->result still pointing to the error message from Tcl_GetInt.

Here are some examples of invoking the sum command in Tcl scripts:

```
sum 2 3
5
sum 011 0x14
29
sum 3 6z
expected integer but got "6z"
```

Tcl_GetDouble is similar to Tcl_GetInt except that it expects the string to consist of a floating-point number such as "-2.2" or "3.0e-6" or "7". It stores the double-precision value of the number at the location given by its last argument or returns an error in the same way as Tcl_GetInt. Tcl_GetBoolean is similar except that it converts the string to a 0 or 1 integer value, which it stores at the location given by its last argument. Any of the true values listed in Table 30.2 converts to 1 and any of the false values converts to 0.

True Values	False Values
1	0
true	false
on	off
yes	no

Table 30.2. Legal values for boolean strings parsed by Tcl_GetBoolean. Any of the values may be abbreviated or capitalized.

Many other Tcl and Tk library procedures are similar to Tcl_GetInt in the way they use an interp argument for error reporting. These procedures all expect the interpreter's result to be in its initialized state when they are called. If they complete successfully then they usually leave the result in that state; if an error occurs then they put an error message in the result. The procedures' return values indicate whether they succeeded, usually as a TCL_OK or TCL_ERROR completion code but sometimes in other forms such as a NULL string pointer. When an error occurs, all the caller needs to do is to return a failure itself, leaving the error message in the interpreter's result.

30.2 Expression evaluation

Tcl provides four library procedures that evaluate expressions of the form described in Chapter XXX: Tcl_ExprString, Tcl_ExprLong, Tcl_ExprDouble, and Tcl_ExprBoolean. These procedures are similar except that they return the result of the expression in different forms as indicated by their names. Here is a slightly simplified implementation of the expr command, which uses Tcl_ExprString:

```
int ExprCmd(ClientData clientData, TclInterp *interp,
        int argc, char *argv[]) {
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    return Tcl_ExprString(interp, argv[1]);
}
```

All ExprCmd does is to check its argument count and then call Tcl_ExprString. Tcl_ExprString evaluates its second argument as a Tcl expression and returns the value as a string in interp->result. Like Tcl_GetInt, it returns TCL_OK if it evaluated the expression successfully; if an error occurs it leaves an error message in interp->result and returns TCL_ERROR.

Tcl_ExprLong, Tcl_ExprDouble, and Tcl_ExprBoolean are similar to Tcl_ExprString except that they return the expression's result as a long integer, double-precision floating-point number, or 0/1 integer, respectively. Each of the procedures takes an additional argument that points to a place to store the result. For these procedures the result must be convertible to the requested type. For example, if "abc" is passed to Tcl_ExprLong then it will return an error because "abc" has no integer value. If the string "40" is passed to Tcl_ExprBoolean it will succeed and store 1 in the value word (any non-zero integer is considered to be true).

30.3 Manipulating lists

Tcl provides several procedures for manipulating lists, of which the most useful are `Tcl_SplitList` and `Tcl_Merge`. Given a string in the form of a Tcl list, `Tcl_SplitList` extracts the elements and returns them as an array of string pointers. For example, here is an implementation of Tcl's `lindex` command that uses `Tcl_SplitList`:

```
int LindexCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    int index, listArgc;
    char **listArgv;
    if (argc != 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, argv[2], &index) != TCL_OK) {
        return TCL_ERROR;
    }
    if (Tcl_SplitList(interp, argv[1], &listArgc,
            &listArgv) != TCL_OK) {
        return TCL_ERROR;
    }
    if ((index >= 0) && (index < listArgc)) {
        Tcl_SetResult(interp, listArgv[index],
                TCL_VOLATILE);
    }
    free((char *) listArgv);
    return TCL_OK;
}
```

`LindexCmd` checks its argument count, calls `Tcl_GetInt` to convert `argv[2]` (the index) into an integer, then calls `Tcl_SplitList` to parse the list. `Tcl_SplitList` returns a count of the number of elements in the list to `listArgc`. It also creates an array of pointers to the values of the elements and stores a pointer to that array in `listArgv`. If `Tcl_SplitList` encounters an error in parsing the list (e.g. unmatched braces) then it returns `TCL_ERROR` and leaves an error message in `interp->result`; otherwise it returns `TCL_OK`.

`Tcl_SplitList` calls `malloc` to allocate space for the array of pointers and for the string values of the elements; the caller must free up this space by passing `listArgv` to `free`. The space for both pointers and strings is allocated in a single block of memory so only a single call to `free` is needed. `LindexCmd` calls `Tcl_SetResult` to copy the desired element into the interpreter's result. It specifies `TCL_VOLATILE` to indicate that the string value is about to be destroyed (its memory will be freed); `Tcl_SetResult` will make a copy of the `listArgv[index]` for `interp`'s result. If the specified index is outside the range of elements in the list then LindexCmd leaves interp->result in its initialized state, which returns an empty string.

Tcl_Merge is the inverse of Tcl_SplitList. Given argc and argv information describing the elements of a list, it returns a malloc'ed string containing the list. Tcl_Merge always succeeds so it doesn't need an interp argument for error reporting. Here's another implementation of the list command, which uses Tcl_Merge:

```
int ListCmd2(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    interp->result = Tcl_Merge(argc-1, argv+1);
    interp->freeProc = (Tcl_FreeProc *) free;
    return TCL_OK;
}
```

ListCmd2 takes the result from Tcl_Merge and stores it in the interpreter's result. Since the list string is dynamically allocated ListCmd2 sets interp->freeProc to free so that Tcl will call free to release the storage for the list when it is no longer needed.

Chapter 31
Exceptions

Many Tcl commands, such as `if` and `while`, have arguments that are Tcl scripts. The command procedures for these commands invoke `Tcl_Eval` recursively to evaluate the scripts. If `Tcl_Eval` returns a completion code other than `TCL_OK` then an *exception* is said to have occurred. Exceptions include `TCL_ERROR`, which was described in Chapter 31, plus several others that have not been mentioned before. This chapter introduces the full set of exceptions and describes how to unwind nested evaluations and leave useful information in the `errorInfo` and `errorCode` variables. See Table 31.1 for a summary of procedures related to exception handling.

31.1 Completion codes.

Table 31.2 lists the full set of Tcl completion codes that may be returned by command procedures. If a command procedure returns anything other than `TCL_OK` then Tcl aborts the evaluation of the script containing the command and returns the same completion code as the result of `Tcl_Eval` (or `Tcl_EvalFile`, etc). `TCL_OK` and `TCL_ERROR` have already been discussed; they are used for normal returns and errors, respectively. The completion codes `TCL_BREAK` or `TCL_CONTINUE` occur if `break` or `continue` commands are invoked by a script; in both of these cases the interpreter's result will be an empty string. The `TCL_RETURN` completion code occurs if `return` is invoked; in this case the interpreter's result will be the intended result of the enclosing procedure.

As an example of how to generate a `TCL_BREAK` completion code, here is the command procedure for the `break` command:

```
Tcl_AddErrorInfo(Tcl_Interp *interp, char *message)
```
Adds `message` to stack trace being formed in the `errorInfo` variable.

```
Tcl_SetErrorCode(Tcl_Interp *interp, char *field, char *field,
    ... (char *) NULL)
```
Creates a list whose elements are the `field` arguments, and sets the `errorCode` variable to the contents of the list.

Table 31.1. A summary of Tcl library procedures for setting `errorInfo` and `errorCode`.

Completion Code	Meaning
TCL_OK	Command completed normally.
TCL_ERROR	Unrecoverable error occurred.
TCL_BREAK	`Break` command was invoked.
TCL_CONTINUE	`Continue` command was invoked.
TCL_RETURN	`Return` command was invoked.

Table 31.2. Completion codes that may be returned by command procedures and procedures that evaluate scripts, such as `Tcl_Eval`.

```
int BreakCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    return TCL_BREAK;
}
```

`TCL_BREAK`, `TCL_CONTINUE`, and `TCL_RETURN` are used to unwind nested script evaluations back to an enclosing looping command or procedure invocation. Under most circumstances, any procedure that receives a completion code other than `TCL_OK` from `Tcl_Eval` should immediately return that same completion code to its caller without modifying the interpreter's result. However, a few commands process some of the special completion codes without returning them upward. For example, here is an implementation of the `while` command:

```
int WhileCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    int bool;
    int code;
    if (argc != 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    while (1) {
        Tcl_ResetResult(interp);
        if (Tcl_ExprBoolean(interp, argv[1], &bool)
                != TCL_OK) {
            return TCL_ERROR;
        }
        if (bool == 0) {
            return TCL_OK;
        }
        code = Tcl_Eval(interp, argv[2]);
        if (code == TCL_CONTINUE) {
            continue;
        } else if (code == TCL_BREAK) {
            return TCL_OK;
        } else if (code != TCL_OK) {
            return code;
        }
    }
}
```

After checking its argument count, `WhileCmd` enters a loop where each iteration evaluates the command's first argument as an expression and its second argument as a script. If an error occurs while evaluating the expression then `WhileCmd` returns the error. If the expression evaluates successfully but its value is 0, then the command terminates with a normal return. Otherwise it evaluates the script argument. If the completion code is `TCL_CONTINUE` then `WhileCmd` goes on to the next loop iteration. If the code is `TCL_BREAK` then `WhileCmd` ends the execution of the command and returns `TCL_OK` to its caller. If `Tcl_Eval` returns any other completion code besides `TCL_OK` then `WhileCmd` simply reflects that code upwards. This causes the proper unwinding to occur on `TCL_ERROR` or `TCL_RETURN` codes, and it will also unwind if any new completion codes are added in the future.

If an exceptional return unwinds all the way through the outermost script being evaluated then Tcl checks the completion code to be sure it is either `TCL_OK` or `TCL_ERROR`. If not then Tcl turns the return into an error with an appropriate error message. Furthermore, if a `TCL_BREAK` or `TCL_CONTINUE` exception unwinds all the way out of a procedure then Tcl also turns it into an error. For example:

```
break
invoked "break" outside of a loop
proc badbreak {} {break}
badbreak
invoked "break" outside of a loop
```

Thus applications need not worry about completion codes other then TCL_OK and TCL_ERROR when they evaluate scripts from the outermost level.

31.2 Augmenting the stack trace in errorInfo

When an error occurs, Tcl modifies the errorInfo global variable to hold a stack trace of the commands that were being evaluated at the time of the error. It does this by calling the procedure Tcl_AddErrorInfo, which has the following prototype:

```
void Tcl_AddErrorInfo(Tcl_Interp *interp,
        char *message)
```

The first call to Tcl_AddErrorInfo after an error sets errorInfo to the error message stored in interp->result and then appends message. Each subsubsequent call for the same error appends message to errorInfo's current value. Whenever a command procedure returns TCL_ERROR Tcl_Eval calls Tcl_AddErrorInfo to log information about the command that was being executed. If there are nested calls to Tcl_Eval then each one adds information about its command as it unwinds, so that a stack trace forms in errorInfo.

Command procedures can call Tcl_AddErrorInfo themselves to provide additional information about the context of the error. This is particularly useful for command procedures tha invoke Tcl_Eval recursively. For example, consider the following Tcl procedure, which is a buggy attempt to find the length of the longest element in a list:

```
proc longest list {
    set i [llength $list]
    while {$i >= 0} {
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i
    }
    return $max
}
```

This procedure is buggy because it never initializes the variable max, so an error will occur when the if command attempts to read it. If the procedure is invoked with the command “`longest {a 12345 xyz}`”, then the following stack trace will be stored in `errorInfo` after the error:

```
can't read "max": no such variable
    while executing
"if {$length > $max} {
            set max $length
        }"
    ("while" body line 3)
    invoked from within
"while {$i >= 0} {
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i
    }"
    (procedure "longest" line 3)
    invoked from within
"longest {a 12345 xyz}"
```

All of the information is provided by `Tcl_Eval` except for the two lines with comments in parentheses. The first of these lines was generated by the command procedure for `while`, and the second was generated by the Tcl code that evaluates procedure bodies. If you used the implementation of `while` on page 287 instead of the built-in Tcl implementation then the first parenthesized message would be missing. The C code below is a replacement for the last `else` clause in `WhileCmd`; it uses `Tcl_AppendResult` to add the parenthetical remark.

```
...
} else if (code != TCL_OK) {
    if (code == TCL_ERROR) {
        char msg[50];
        sprintf(msg, "\n    (\"while\" body line %d)",
                interp->errorLine);
        Tcl_AddErrorInfo(interp, msg);
    }
    return code;
}
...
```

The `errorLine` field of `interp` is set by `Tcl_Eval` whenever a command procedure returns an error; it gives the line number of the command that produced the error, within the script being executed. A line number of 1 corresponds to the first line, which is the line containing the open brace in this example; the `if` command that generated the error is on line 3.

For simple Tcl commands you shouldn't need to invoke `Tcl_AddErrorInfo`: the information provided by `Tcl_Eval` will be sufficient. However, if you write code that calls `Tcl_Eval` then I recommend calling `Tcl_AddErrorInfo` whenever `Tcl_Eval` returns an error, to provide information about why `Tcl_Eval` was invoked and also to include the line number of the error.

Note: *You must call* `Tcl_AddErrorInfo` *rather than trying to set the* `errorInfo` *variable directly, because* `Tcl_AddErrorInfo` *contains special code to detect the first call after an error and clear out the old contents of* `errorInfo`.

31.3 Setting errorCode

The last piece of information set after an error is the `errorCode` variable, which provides information about the error in a form that's easy to process with Tcl scripts. It's intended for use in situations where a script is likely to catch the error, determine exactly what went wrong, and attempt to recover from it if possible. If a command procedure returns an error to Tcl without setting `errorCode` then Tcl sets it to `NONE`. If a command procedure wishes to provide information in `errorCode` then it should invoke `Tcl_SetErrorCode` before returning `TCL_ERROR`.

`Tcl_SetErrorCode` takes as arguments an interpreter and any number of string arguments ending with a null pointer. It forms the strings into a list and stores the list as the value of `errorCode`. For example, suppose that you have written several commands to implement gizmo objects, and that there are several errors that could occur in commands that manipulate the objects, such as an attempt to use a non-existent object. If one of your command procedures detects a non-existent object error, it might set `errorCode` as follows:

```
Tcl_SetErrorCode(interp, "GIZMO", "EXIST",
        "no object by that name", (char *) NULL);
```

This will leave the value "`GIZMO EXIST {no object by that name}`" in `errorCode`. `GIZMO` identifies a general class of errors (those associated with gizmo objects), `EXIST` is the symbolic name for the particular error that occurred, and the last element of the list is a human-readable error message. You can store whatever you want in `errorCode` as long as the first list element doesn't conflict with other values already in use, but the overall idea is to provide symbolic information that can easily be processed by a Tcl script. For example, a script that accesses gizmos might catch errors and if the error is a non-existent gizmo it might automatically create a new gizmo.

Note: *It's important to call* `Tcl_SetErrorCode` *rather than setting* `errorCode` *directly with* `Tcl_SetVar`. *This is because* `Tcl_SetErrrorCode` *also sets other information in the interpreter so that* `errorCode` *isn't later set to its default value; if you set* `errorCode` *directly, then Tcl will override your value with the default value* `NONE`.

Chapter 32
Accessing Tcl Variables

This chapter describes how you can access Tcl variables from C code. Tcl provides library procedures to set variables, read their values, and unset them. It also provides a tracing mechanism that you can use to monitor and restrict variable accesses. Table 32.1 summarizes the library procedures that are discussed in the chapter.

32.1 Naming variables

The procedures related to variables come in pairs such as `Tcl_SetVar` and `Tcl_SetVar2`. The two procedures in each pair differ only in the way they name a Tcl variable. In the first procedure of each pair, such as `Tcl_SetVar`, the variable is named with a single string argument, `varName`. This form is typically used when a variable name has been specified as an argument to a Tcl command. The string can name a scalar variable, e.g. "`x`" or "`fieldName`", or it can name an element of an array, e.g. "`a(42)`" or "`area(South America)`". No substitutions or modifications are performed on the name. For example, if `varName` is "`a($i)`" Tcl will not use the value of variable `i` as the element name within array `a`; it will use the string "`$i`" literally as the element name.

The second procedure of each pair has a name ending in "`2`", e.g. `Tcl_SetVar2`. In these procedures the variable name is separated into two arguments: `name1` and `name2`. If the variable is a scalar then `name1` is the name of the variable and `name2` is `NULL`. If the variable is an array element then `name1` is the name of the array and `name2` is the name of the element within the array. This form of procedure is less commonly used but it is slightly faster than the first form (procedures like `Tcl_SetVar` are implemented by calling procedures like `Tcl_SetVar2`).

`char *Tcl_SetVar(Tcl_Interp *interp, char *varName, char *newValue, int flags)` `char *Tcl_SetVar2(Tcl_Interp *interp, char *name1, char *name2, char *newValue, int flags)`	Sets the value of the variable to `newValue`, creating the variable if it didn't already exist. Returns the new value of the variable or `NULL` in case of error.
`char *Tcl_GetVar(Tcl_Interp *interp, char *varName, int flags)` `char *Tcl_GetVar2(Tcl_Interp *interp, char *name1, char *name2, int flags)`	Returns the current value of the variable, or `NULL` in case of error.
`int Tcl_UnsetVar(Tcl_Interp *interp, char *varName, int flags)` `int Tcl_UnsetVar2(Tcl_Interp *interp, char *name1, char *name2, int flags)`	Removes the variable from `interp` and returns `TCL_OK`. If the variable doesn't exist or has an active trace then it can't be removed and `TCL_ERROR` is returned.
`int Tcl_TraceVar(Tcl_Interp *interp, char *varName, int flags, Tcl_VarTraceProc *proc, ClientData clientData)` `int Tcl_TraceVar2(Tcl_Interp *interp, char *name1, char *name2, int flags, Tcl_VarTraceProc *proc, ClientData clientData)`	Arrange for `proc` to be invoked whenever one of the operations specified by `flags` is performed on the variable. Returns `TCL_OK` or `TCL_ERROR`.
`Tcl_UntraceVar(Tcl_Interp *interp, char *varName, int flags, Tcl_VarTraceProc *proc, ClientData clientData)` `Tcl_UntraceVar2(Tcl_Interp *interp, char *name1, char *name2, int flags, Tcl_VarTraceProc *proc, ClientData clientData)`	Removes the trace on the variable that matches `proc`, `clientData`, and `flags`, if there is one.
`ClientData Tcl_VarTraceInfo(Tcl_Interp *interp, char *varName, int flags, Tcl_VarTraceProc *proc, ClientData prevClientData)` `ClientData Tcl_VarTraceInfo2(Tcl_Interp *interp, char *name1, char *name2, int flags, Tcl_VarTraceProc *proc, ClientData prevclientData)`	If `prevClientData` is `NULL`, returns the ClientData associated with the first trace on the variable that matches `flags` and `proc` (only the `TCL_GLOBAL_ONLY` bit of `flags` is used); otherwise returns the `ClientData` for the next trace matching `flags` and `proc` after the one whose `ClientData` is `prevClientData`. Returns `NULL` if there are no (more) matching traces.

Table 32.1. Tcl library procedures for manipulating variables. The procedures come in pairs; in one procedure the variable is named with a single string (which may specify either a scalar or an array element) and in the other procedure the variable is named with separate array and element names (`name1` and `name2`, respectively). If `name2` is `NULL` then the variable must be a scalar.

Flag Name	Meaning
TCL_GLOBAL_ONLY	Reference global variable, regardless of current execution context.
TCL_LEAVE_ERR_MSG	If operation fails, leave error message in `interp->result`.
TCL_APPEND_VALUE	Append new value to existing value instead of overwriting.
TCL_LIST_ELEMENT	Convert new value to a list element before setting or appending.

Table 32.2. Values that may be OR'ed together in the flags arguments to `Tcl_SetVar` and `Tcl_SetVar2`. Other procedures use a subset of these flags.

32.2 Setting variable values

`Tcl_SetVar` and `Tcl_SetVar2` are used to set the value of a variable. For example,

```
Tcl_SetVar(interp, "a", "44", 0);
```

will set the value of variable `a` in `interp` to the string "`44`". If there does not yet exist a variable named `a` then a new one will be created. The variable is set in the current execution context: if a Tcl procedure is currently being executed, the variable will be a local one for that procedure; if no procedure is currently being executed then the variable will be a global variable. If the operation completed successfully then the return value from `Tcl_SetVar` is a pointer to the variable's new value as stored in the variable table (this value is static enough to be used as an interpreter's result). If an error occurred, such as specifying the name of an array without also specifying an element name, then `NULL` is returned.

The last argument to `Tcl_SetVar` or `Tcl_SetVar2` consists of an OR'ed combination of flag bits. Table 32.2 lists the symbolic values for the flags. If the `TCL_GLOBAL_ONLY` flag is specified then the operation always applies to a global variable, even if a Tcl procedure is currently being executed. `TCL_LEAVE_ERR_MSG` controls how errors are reported. Normally, `Tcl_SetVar` and `Tcl_SetVar2` just return `NULL` if an error occurs. However, if `TCL_LEAVE_ERR_MSG` has been specified then the procedures will also store an error message in the interpreter's result. This last form is useful when the procedure is invoked from a command procedure that plans to abort if the variable access fails.

The flag `TCL_APPEND_VALUE` means that the new value should be appended to the variable's current value instead of replacing it. Tcl implements the append operation in a way that is relatively efficient, even in the face of repeated appends to the same variable. If the variable doesn't yet exist then `TCL_APPEND_VALUE` has no effect.

The last flag, `TCL_LIST_ELEMENT`, means that the new value should be converted to a proper list element (e.g. by enclosing in braces if necessary) before setting or appending. If both `TCL_LIST_ELEMENT` and `TCL_APPEND_VALUE` are specified then a separator space is also added before the new element if it's needed.

Here is an implementation of the `lappend` command that uses `Tcl_SetVar`:

```
int LappendCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    int i;
    char *newValue;
    if (argc < 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    for (i = 2; i < argc; i++) {
        newValue = Tcl_SetVar(interp, argv[1], argv[i],
                TCL_LIST_ELEMENT|TCL_APPEND_VALUE
                |TCL_LEAVE_ERR_MSG);
        if (newValue == NULL) {
            return TCL_ERROR;
        }
    }
    interp->result = newValue;
    return TCL_OK;
}
```

It simply calls `Tcl_SetVar` once for each argument and lets `Tcl_SetVar` do all the work of converting the argument to a list value and appending it to the variable. If an error occurs then `Tcl_SetVar` leaves an error message in `interp->result` and `LappendCmd` returns the message back to Tcl. If the command completes successfully then it returns the variable's final value as its result. For example, suppose the following Tcl command is invoked:

```
set a 44
lappend a x {b c}
44 x {b c}
```

When `LappendCmd` is invoked `argc` will be 4. `Argv[2]` will be "`x`" and `argv[3]` will be "`b c`" (the braces are removed by the Tcl parser). `LappendCmd` makes two calls to `Tcl_SetVar`; during the first call no conversion is necessary to produce a proper list element, but during the second call `Tcl_SetVar` adds braces back around "`b c`" before appending it the variable.

32.3 Reading variables

The procedures Tcl_GetVar and Tcl_GetVar2 may be used to retrieve variable values. For example,

```
char *value;
...
value = Tcl_GetVar(interp, "a", 0);
```

will store in value a pointer to the current value of variable a. If the variable doesn't exist or some other error occurs then NULL is returned. Tcl_GetVar and Tcl_GetVar2 support the TCL_GLOBAL_ONLY and TCL_LEAVE_ERR_MSG flags in the same way as Tcl_SetVar. The following command procedure uses Tcl_GetVar and Tcl_SetVar to implement the incr command:

```
int IncrCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    int value, inc;
    char *string;
    if ((argc != 2) && (argc != 3)) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (argc == 2) {
        inc = 1;
    } else if (Tcl_GetInt(interp, argv[2], &inc)
            != TCL_OK) {
        return TCL_ERROR;
    }
    string = Tcl_GetVar(interp, argv[1],
            TCL_LEAVE_ERR_MSG);
    if (string == NULL) {
        return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, string, &value) != TCL_OK) {
        return TCL_ERROR;
    }
    sprintf(interp->result, "%d", value + inc);
    if (Tcl_SetVar(interp, argv[1], interp->result,
            TCL_LEAVE_ERR_MSG) == NULL) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

IncrCmd does very little work itself. It just calls library procedures and aborts if errors occur. The first call to Tcl_GetInt converts the increment from text to binary.

Tcl_GetVar retrieves the original value of the variable, and another call to Tcl_GetInt converts that value to binary. IncrCmd then adds the increment to the variable's value and calls sprintf to convert the result back to text. Tcl_SetVar stores this value in the variable, and IncrCmd then returns the new value as its result.

32.4 Unsetting variables

To remove a variable, call Tcl_UnsetVar or Tcl_UnsetVar2. For example,

```
Tcl_UnsetVar2(interp, "population", "Michigan", 0);
```

will remove the element Michigan from the array population. This statement has the same effect as the Tcl command

```
unset population(Michigan)
```

Tcl_UnsetVar and Tcl_UnsetVar2 return TCL_OK if the variable was successfully removed and TCL_ERROR if the variable didn't exist or couldn't be removed for some other reason. TCL_GLOBAL_ONLY and TCL_LEAVE_ERR_MSG may be specified as flags to these procedures. If an array name is given without an element name then the entire array is removed.

32.5 Setting and unsetting variable traces

Variable traces allow you to specify a C procedure to be invoked whenever a variable is read, written, or unset. Traces can be used for many purposes. For example, in Tk you can configure a button widget so that it displays the value of a variable and updates itself automatically when the variable is modified. This feature is implemented with variable traces. You can also use traces for debugging, to create read-only variables, and for many other purposes.

The procedures Tcl_TraceVar and Tcl_TraceVar2 create variable traces, as in the following example:

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES, WriteProc,
        (ClientData) NULL);
```

This creates a write trace on variable x in interp: WriteProc will be invoked whenever x is modified. The third argument to Tcl_TraceVar is an OR'ed combination of flag bits that select the operations to trace: TCL_TRACE_READS for reads, TCL_TRACE_WRITES for writes, and TCL_TRACE_UNSETS for unsets. In addition, the flag TCL_GLOBAL_ONLY may be specified to force the variable name to be interpreted as global. Tcl_TraceVar and Tcl_TraceVar2 normally return TCL_OK; if an error occurs then they leave an error message in interp->result and return TCL_ERROR.

The library procedures `Tcl_UntraceVar` and `Tcl_UntraceVar2` remove variable traces. For example, the following call will remove the trace set above:

```
Tcl_UntraceVar(interp, "x", TCL_TRACE_WRITES,
        WriteProc, (ClientData) NULL);
```

`Tcl_UntraceVar` finds the specified variable, looks for a trace that matches the flags, trace procedure, and ClientData specified by its arguments, and removes the trace if it exists. If no matching trace exists then `Tcl_UntraceVar` does nothing. `Tcl_UntraceVar` and `Tcl_UntraceVar2` accept the same flag bits as `Tcl_TraceVar`.

32.6 Trace callbacks

Trace callback procedures such as `WriteProc` in the previous section must match the following prototype:

```
typedef char *Tcl_VarTraceProc(ClientData clientData,
        Tcl_Interp *interp, char *name1, char *name2,
        int flags);
```

The `clientData` and `interp` arguments will be the same as the corresponding arguments passed to `Tcl_TraceVar` or `Tcl_TraceVar2`. `ClientData` typically points to a structure containing information needed by the trace callback. `Name1` and `name2` give the name of the variable in the same form as the arguments to `Tcl_SetVar2`. `Flags` consists of an OR'ed combination of bits. One of `TCL_TRACE_READS`, `TCL_TRACE_WRITES`, or `TCL_TRACE_UNSETS` is set to indicate which operation triggered the trace, and `TCL_GLOBAL_ONLY` is set if the variable is a global variable that isn't accessible from the current execution context; the trace callback must pass this flag back into procedures like `Tcl_GetVar2` if it wishes to access the variable. The bits `TCL_TRACE_DESTROYED` and `TCL_INTERP_DESTROYED` are set in special circumstances described below.

For read traces, the callback is invoked just before `Tcl_GetVar` or `Tcl_GetVar2` returns the variable's value to whomever requested it; if the callback modifies the value of the variable then the modified value will be returned. For write traces the callback is invoked after the variable's value has been changed. The callback can modify the variable to override the change, and this modified value will be returned as the result of `Tcl_SetVar` or `Tcl_SetVar2`. For unset traces the callback is invoked after the variable has been unset, so the callback cannot access the variable. Unset callbacks can occur when a variable is explicitly unset, when a procedure returns (thereby deleting all of its local variables) or when an interpreter is destroyed (thereby deleting all of the variables in the interpreter).

A trace callback procedure can invoke `Tcl_GetVar2` and `Tcl_SetVar2` to read and write the value of the traced variable. All traces on the variable are temporarily disabled while the callback executes so calls to `Tcl_GetVar2` and `Tcl_SetVar2` will not trigger additional trace callbacks. As mentioned above, unset traces aren't invoked until after the variable has been deleted, so attempts to read the variable during unset callbacks will fail. However, it is possible for an unset callback procedure to write the variable, in which case a new variable will be created.

The code below sets a write trace that prints out the new value of variable `x` each time it is modified:

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES, Print,
        (ClientData) NULL);
...
char *Print(ClientData clientData,
        Tcl_Interp *interp, char *name1, char *name2,
        int flags) {
    char *value;
    value = Tcl_GetVar2(interp, name1, name2,
            flags & TCL_GLOBAL_ONLY);
    if (value != NULL) {
        printf("new value is %s\n", value);
    }
    return NULL;
}
```

`PrintProc` must pass the `TCL_GLOBAL_ONLY` bit of its `flags` argument on to `Tcl_GetVar2` in order to make sure that the variable can be accessed properly. `Tcl_GetVar2` should never return an error, but `PrintProc` checks for one anyway and doesn't try to print the variable's value if an error occurs.

Trace callbacks normally return `NULL` values; a non-`NULL` value signals an error. In this case the return value must be a pointer to a static string containing an error message. The traced access will abort and the error message will be returned to whomever initiated that access. For example, if the access was invoked by a `set` command or `$`-substitution then a Tcl error will result; if the access was invoked via `Tcl_GetVar`, `Tcl_GetVar` will return `NULL` and also leave the error message in `interp->result` if the `TCL_LEAVE_ERR_MSG` flag was specified.

The code below uses a trace to make variable `x` read-only with value `192`:

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES, Reject,
        (ClientData) "192");
char *Reject(ClientData clientData, Tcl_Interp *interp,
        char *name1, char *name2, int flags) {
    char *correct = (char *) ClientData;
    Tcl_SetVar2(interp, name1, name2, correct,
                flags & TCL_GLOBAL_ONLY);
    return "variable is read-only";
};
```

`Reject` is a trace callback that's invoked whenever `x` is written. It returns an error message to abort the write access. Since `x` has already been modified before `Reject` is invoked, `Reject` must undo the write by restoring the variable's correct value. The correct value is passed to the trace callback using its `clientData` argument. This implementation allows the same procedure to be used as the write callback for many different read-only variables; a different correct value can be passed to `Reject` for each variable.

32.7 Whole-array traces

You can create a trace on an entire array by specifying an array name to `Tcl_TraceVar` or `Tcl_TraceVar2` without an element name. This creates a whole-array trace: the callback procedure will be invoked whenever any of the specified operations is invoked on any element of the array. If the entire array is unset then the callback will be invoked just once, with `name1` containing the array name and `name2` `NULL`.

32.8 Multiple traces

Multiple traces can exist for the same variable. When this happens, each of the relevant callbacks is invoked on each variable access. The callbacks are invoked in order from most-recently-created to oldest. If there are both whole-array traces and individual element traces, then the whole-array callbacks are invoked before element callbacks. If an error is returned by one of the callbacks then no subsequent callbacks are invoked.

32.9 Unset callbacks

Unset callbacks are different from read and write callbacks in several ways. First of all, unset callbacks cannot return an error condition; they must always succeed. Second, two extra flags are defined for unset callbacks: `TCL_TRACE_DELETED` and `TCL_INTERP_DESTROYED`. When a variable is unset all of its traces are deleted; unset traces on the variable will still be invoked, but they will be passed the `TCL_TRACE_DELETED` flag to indicate that the trace has now been deleted and won't be invoked anymore. If an array element is unset and there is a whole-array unset trace for the element's array, then the unset trace is not deleted and the callback will be invoked without the `TCL_TRACE_DELETED` flag set.

If the `TCL_INTERP_DESTROYED` flag is set during an unset callback it means that the interpreter containing the variable has been destroyed. In this case the callback must be careful not to use the interpreter at all, since the interpreter's state is in the process of being deleted. All that the callback should do is to clean up its own internal data structures.

32.10 Non-existent variables

It is legal to set a trace on a variable that does not yet exist. The variable will continue to appear not to exist (e.g. attempts to read it will fail), but the trace's callback will be invoked during operations on the variable. For example, you can set a read trace on an undefined variable and then, on the first access to the variable, assign it a default value.

32.11 Querying trace information

The procedures `Tcl_VarTraceInfo` and `Tcl_VarTraceInfo2` can be used to find out if a particular kind of trace has been set on a variable and if so to retrieve its ClientData value. For example, consider the following code:

```
ClientData clientData;
...
clientData = Tcl_VarTraceInfo(interp, "x", 0, Reject,
        (ClientData) NULL);
```

`Tcl_VarTraceInfo` will see if there is a trace on variable `x` that has `Reject` as its trace callback. If so, it will return the ClientData value associated with the first (most recently created) such trace; if not it will return `NULL`. Given the code in Section 32.6 above, this call will tell whether `x` is read-only; if so, it will return the variable's read-only value. If there are multiple traces on a variable with the same callback, you can step through them all in order by making multiple calls to `Tcl_VarTraceInfo`, as in the following code:

```
ClientData clientData;
...
clientData = NULL;
while (1) {
    clientData = Tcl_VarTraceInfo(interp, "x", 0,
            Reject, clientData);
    if (clientData == NULL) {
        break;
    }
    ... process trace ...
}
```

In each call to `Tcl_VarTraceInfo` after the first, the previous ClientData value is passed in as the last argument. `Tcl_VarTraceInfo` finds the trace with this value, then returns the ClientData for the next trace. When it reaches the last trace it returns `NULL`.

Chapter 33
Hash Tables

A *hash table* is a collection of *entries*, where each entry consists of a *key* and a *value*. No two entries have the same key. Given a key, a hash table can very quickly locate its entry and hence the associated value. Tcl contains a general-purpose hash table package that it uses in several places internally. For example, all of the commands in an interpreter are stored in a hash table where the key for each entry is a command name and the value is a pointer to information about the command. All of the global variables are stored in another hash table where the key for each entry is the name of a variable and the value is a pointer to information about the variable.

Tcl exports its hash table facilities through a set of library procedures so that applications can use them too (see Table 33.1 for a summary). The most common use for hash tables is to associate names with objects. In order for an application to implement a new kind of object it must give the objects textual names for use in Tcl commands. When a command procedure receives an object name as an argument it must locate the C data structure for the object. Typically there will be one hash table for each type of object, where the key for an entry is an object name and the value is a pointer to the C data structure that represents the object. When a command procedure needs to find an object it looks up its name in the hash table. If there is no entry for the name then the command procedure returns an error.

For the examples in this chapter I'll use a hypothetical application that implements objects called "gizmos". Each gizmo is represented internally with a structure declared like this:

```
typedef struct Gizmo {
    ... fields of gizmo object ...
} Gizmo;
```

`Tcl_InitHashTable(Tcl_HashTable *tablePtr, int keyType)`
Creates a new hash table and stores information about the table at `*tablePtr`. `KeyType` is either `TCL_STRING_KEYS`, `TCL_ONE_WORD_KEYS`, or an integer greater than 1.
`Tcl_DeleteHashTable(Tcl_HashTable *tablePtr)`
Deletes all the entries in the hash table and frees up related storage.

`Tcl_HashEntry *Tcl_CreateHashEntry(Tcl_HashTable *tablePtr, char *key, int *newPtr)`
Returns a pointer to the entry in `tablePtr` whose key is `key`, creating a new entry if needed. `*NewPtr` is set to 1 if a new entry was created or 0 if the entry already existed.
`Tcl_HashEntry *Tcl_FindHashEntry(Tcl_HashTable *tablePtr, char *key)`
Returns a pointer to the entry in `tablePtr` whose key is `key`, or `NULL` if no such entry exists.
`Tcl_DeleteHashEntry(Tcl_HashEntry *entryPtr)`
Deletes an entry from its hash table.

`ClientData Tcl_GetHashValue(Tcl_HashEntry *entryPtr)`
Returns the value associated with a hash table entry.
`Tcl_SetHashValue(Tcl_HashEntry *entryPtr, ClientData value)`
Sets the value associated with a hash table entry.
`char *Tcl_GetHashKey(Tcl_HashEntry *entryPtr)`
Returns the key associated with a hash table entry.

`Tcl_HashEntry *Tcl_FirstHashEntry(Tcl_HashTable *tablePtr, Tcl_HashSearch *searchPtr)`
Starts a search through all the elements of a hash table. Stores information about the search at `*searchPtr` and returns the hash table's first entry or `NULL` if it has no entries.
`Tcl_HashEntry *Tcl_NextHashEntry(Tcl_HashSearch *searchPtr)`
Returns the next entry in the search identified by `searchPtr` or `NULL` if all entries in the table have been returned.

`char *Tcl_HashStats(Tcl_HashTable *tablePtr)` Returns a string giving usage statistics for `tablePtr`. The string is dynamically allocated and must be freed by the caller.

The application uses names like “`gizmo42`” to refer to gizmos in Tcl commands, where each gizmo has a different number at the end of its name. The application follows the action-oriented approach described in Section 27.3 by providing a collection of Tcl commands to manipulate the objects, such as `gcreate` to create a new gizmo, `gdelete` to delete an existing gizmo, `gsearch` to find gizmos with certain characteristics, and so on.

33.1 Keys and values

Tcl hash tables support three different kinds of keys. All of the entries in a single hash table must use the same kind of key, but different tables may use different kinds. The most common form of key is a string. In this case each key is a `NULL`-terminated string of arbitrary length, such as "`gizmo18`" or "`Waste not want not`". Different entries in a table may have keys of different length. The gizmo implementation uses strings as keys.

The second form of key is a one-word value. In this case each key may be any value that fits in a single word, such as an integer. One-word keys are passed into Tcl using values of type "`char *`" so the keys are limited to the size of a character pointer.

The last form of key is an array. In this case each key is an array of integers (C `int` type). All keys in the table must be the same size.

The values for hash table entries are items of type `ClientData`, which are large enough to hold either an integer or a pointer. In most applications, such as the gizmo example, hash table values are pointers to records for objects. These pointers are cast into `ClientData` items when storing them in hash table entries, and they are cast back from `ClientData` to object pointers when retrieved from the hash table.

33.2 Creating and deleting hash tables

Each hash table is represented by a C structure of type `Tcl_HashTable`. Space for this structure is allocated by the client, not by Tcl; typically these structures are global variables or elements of other structures. When calling hash table procedures you pass in a pointer to a `Tcl_HashTable` structure as a token for the hash table. You should never use or modify any of the fields of a `Tcl_HashTable` directly. Use the Tcl library procedures and macros for this.

Here is how a hash table might be created for the gizmo application:

```
Tcl_HashTable gizmoTable;
...
Tcl_InitHashTable(&gizmoTable, TCL_STRING_KEYS);
```

The first argument to `Tcl_InitHashTable` is a `Tcl_HashTable` pointer and the second argument is an integer that specifies the sort of keys that will be used for the table. `TCL_STRING_KEYS` means that strings will be used in the table; `TCL_ONE_WORD_VALUES` specifies one-word keys; and an integer value greater than one means that keys are arrays with the given number of int's in each array. `Tcl_InitHashTable` ignores the current contents of the table it is passed and re-initializes the structure to refer to an empty hash table with keys as specified.

`Tcl_DeleteHashTable` removes all the entries from a hash table and frees up any memory that was allocated for the table (except space for the `Tcl_HashTable` structure itself, which is the property of the client). For example, the following statement could be used to delete the hash table initialized above:

```
Tcl_DeleteHashTable(&gizmoTable);
```

33.3 Creating entries

The procedure `Tcl_CreateHashEntry` creates an entry with a given key and `Tcl_SetHashValue` sets the value associated with the entry. For example, the code below might be used to implement the `gcreate` command, which makes a new gizmo object:

```
int GcreateCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    static int id = 1;
    int new;
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    ... check argc, etc ...
    do {
        sprintf(interp->result, "gizmo%d", id);
        id++;
        entryPtr = Tcl_CreateHashEntry(&gizmoTable,
                interp->result, &new);
    } while (!new);
    gizmoPtr = (Gizmo *) malloc(sizeof(Gizmo));
    Tcl_SetHashValue(entryPtr, gizmoPtr);
    ... initialize *gizmoPtr, etc ...
    return TCL_OK;
}
```

This code creates a name for the object by concatenating "`gizmo`" with the value of the static variable `id`. It stores the name in `interp->result` so that the command's result will be the name of the new object. `GcreateCmd` then increments `id` so that each new object will have a unique name. `Tcl_CreateHashEntry` is called to create a new entry with a key equal to the object's name; it returns a token for the entry. Under normal conditions there will not already exist an entry with the given key, in which case `Tcl_CreateHashEntry` sets `new` to 1 to indicate that it created a new entry. However, it is possible for `Tcl_CreateHashEntry` to be called with a key that already exists in the table. In `GcreateCmd` this can only happen if a very large number of objects are created, so that `id` wraps around to zero again. If this happens then `Tcl_CreateHashEntry` sets new to 0; `GcreateCmd` will try again with the next larger `id` until it eventually finds a name that isn't already in use.

After creating the hash table entry GcreateCmd allocates memory for the object's record and invokes Tcl_SetHashValue to store the record address as the value of the hash table entry. Tcl_SetHashValue is actually a macro, not a procedure; its first argument is a token for a hash table entry and its second argument, the new value for the entry, can be anything that fits in the space of a ClientData value. After setting the value of the hash table entry GcreateCmd initializes the new object's record.

Note: *Tcl's hash tables restructure themselves as you add entries. A table won't use much memory for the hash buckets when it has only a small number of entries, but it will increase the size of the bucket array as the number of entries increases. Tcl's hash tables should operate efficiently even with very large numbers of entries.*

33.4 Finding existing entries

The procedure Tcl_FindHashEntry locates an existing entry in a hash table. It is similar to Tcl_CreateHashEntry except that it won't create a new entry if the key doesn't already exist in the hash table. Tcl_FindHashEntry is typically used to find an object given its name. For example, the gizmo implementation might contain a utility procedure called GetGizmo, which is something like Tcl_GetInt except that it translates its string argument to a Gizmo pointer instead of an integer:

```
Gizmo *GetGizmo(Tcl_Interp *interp, char *string) {
    Tcl_HashEntry *entryPtr;
    entryPtr = Tcl_FindHashEntry(&gizmoTable, string);
    if (entryPtr == NULL) {
        Tcl_AppendResult(interp, "no gizmo named \"",
                string, "\", (char *) NULL);
        return TCL_ERROR;
    }
    return (Gizmo *) Tcl_GetHashValue(entryPtr);
}
```

GetGizmo looks up a gizmo name in the gizmo hash table. If the name exists then GetGizmo extracts the value from the entry using the macro Tcl_GetHashValue, converts it to a Gizmo pointer, and returns it. If the name doesn't exist then GetGizmo stores an error message in interp->result and returns NULL.

GetGizmo can be invoked from any command procedure that needs to look up a gizmo object. For example, suppose there is a command gtwist that performs a "twist" operation on gizmos, and that it takes a gizmo name as its first argument. The command might be implemented like this:

```
int GtwistCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Gizmo *gizmoPtr;
    ... check argc, etc ...
```

```
    gizmoPtr = GetGizmo(interp, argv[1]);
    if (gizmoPtr == NULL) {
        return TCL_ERROR;
    }
    ... perform twist operation ...
}
```

33.5 Searching

Tcl provides two procedures that you can use to search through all of the entries in a hash table. `Tcl_FirstHashEntry` starts a search and returns the first entry, and `Tcl_NextHashEntry` returns successive entries until the search is complete. For example, suppose that there is a `gsearch` command that searches through all existing gizmos and returns a list of the names of the gizmos that meet a certain set of criteria. This command might be implemented as follows:

```
int GsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Tcl_HashSearch search;
    Gizmo *gizmoPtr;
    ... process arguments to choose search criteria ...
    for (entryPtr = Tcl_FirstHashEntry(&gizmoTable,
            &search); entryPtr != NULL;
            entryPtr = Tcl_NextHashEntry(&search)) {
        gizmoPtr = (Gizmo *) Tcl_GetHashValue(entryPtr);
        if (...object satisfies search criteria...) {
            Tcl_AppendElement(interp,
                    Tcl_GetHashKey(entryPtr));
        }
    }
    return TCL_OK;
}
```

A structure of type `Tcl_HashSearch` is used to keep track of the search. `Tcl_FirstHashEntry` initializes this structure and `Tcl_NextHashEntry` uses the information in the structure to step through successive entries in the table. It's possible to have multiple searches underway simultaneously on the same hash table by using a different `Tcl_HashSearch` structure for each search. `Tcl_FirstHashEntry` returns a token for the first entry in the table (or `NULL` if the table is empty) and `Tcl_NextHashEntry` returns pointers to successive entries, eventually returning `NULL` when the end of the table is reached. For each entry `GsearchCmd` extracts the value from the entry, converts it to a `Gizmo` pointer, and sees if that object meets the criteria specified in the command's arguments. If so, then `GsearchCmd` uses the `Tcl_GetHashKey` macro to get the name of the object (i.e. the entry's key) and invokes Tcl_AppendElement to append the name to the interpreter's result as a list element.

Note: It is not safe to modify the structure of a hash table during a search. If you create or delete entries then you should terminate any searches in progress.

33.6 Deleting entries

The procedure Tcl_DeleteHashEntry will delete an entry from a hash table. For example, the following procedure uses Tcl_DeleteHashEntry to implement a gdelete command, which takes any number of arguments and deletes the gizmo objects they name:

```
int GdeleteCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    int i;
    for (i = 1; i < argc; i++) {
        entryPtr = Tcl_FindHashEntry(&gizmoTable,
                argv[i]);
        if (entryPtr == NULL) {
            continue;
        }
        gizmoPtr = (Gizmo *) Tcl_HashGetValue(entryPtr);
        Tcl_DeleteHashEntry(entryPtr);
        ... clean up *gizmoPtr ...
        free((char *) gizmoPtr);
    }
    return TCL_OK;
}
```

GdeleteCmd checks each of its arguments to see if it is the name of a gizmo object. If not, then the argument is ignored. Otherwise GdeleteCmd extracts a gizmo pointer from the hash table entry and then calls Tcl_DeleteHashEntry to remove the entry from the hash table. Then it performs internal cleanup on the gizmo object if needed and frees the object's record.

33.7 Statistics

The procedure Tcl_HashStats returns a string containing various statistics about the structure of a hash table. For example, it might be used to implement a gstat command for gizmos:

```
int GstatCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    if (argc != 1) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    interp->result = Tcl_HashStats(&gizmoTable);
    interp->freeProc = free;
    return TCL_OK;
}
```

The string returned by `Tcl_HashStats` is dynamically allocated and must be passed to free; `GstatCmd` uses this string as the command's result, and then sets `interp->freeProc` so that Tcl will free the string.

The string returned by `Tcl_HashStats` contains information like the following:

```
1416 entries in table, 1024 buckets
number of buckets with 0 entries: 60
number of buckets with 1 entries: 591
number of buckets with 2 entries: 302
number of buckets with 3 entries: 67
number of buckets with 4 entries: 5
number of buckets with 5 entries: 0
number of buckets with 6 entries: 0
number of buckets with 7 entries: 0
number of buckets with 8 entries: 0
number of buckets with 9 entries: 0
number of buckets with more than 10 entries: 0
average search distance for entry: 1.4
```

You can use this information to see how efficiently the entries are stored in the hash table. For example, the last line indicates the average number of entries that Tcl will have to check during hash table lookups, assuming that all entries are accessed with equal probability.

Chapter 34
String Utilities

This chapter describes Tcl's library procedures for manipulating strings, including a dynamic string mechanism that allows you to build up arbitrarily long strings, a procedure for testing whether a command is complete, and a procedure for doing simple string matching. Table 34.1 summarizes these procedures.

Note: *None of the dynamic string facilities are available in versions of Tcl earlier than 7.0.*

34.1 Dynamic strings

A *dynamic string* is a string that can be appended to without bound. As you append information to a dynamic string, Tcl automatically grows the memory area allocated for it. If the string is short then Tcl avoids dynamic memory allocation altogether by using a small static buffer to hold the string. Tcl provides five procedures for manipulating dynamic strings:

`Tcl_DStringInit` creates a new empty string;

`Tcl_DStringAppend` adds characters to a dynamic string;

`Tcl_DStringAppendElement` adds a new list element to a dynamic string;

`Tcl_DStringFree` releases any storage allocated for a dynamic string and reinitializes the string;

and `Tcl_DStringResult` moves the value of a dynamic string to the result string for an interpreter and reinitializes the dynamic string.

`Tcl_DStringInit(Tcl_DString *dsPtr)`	
	Initializes `*dsPtr` to an empty string (previous contents of `*dsPtr` are discarded without cleanup).
`char *Tcl_DStringAppend(Tcl_DString *dsPtr, char *string, int length)`	
	Appends `length` bytes from `string` to `dsPtr`'s value and returns new value of `dsPtr`. If `length` is less than zero, appends all of `string` up to terminating `NULL` character.
`char *Tcl_DStringAppendElement(Tcl_DString *dsPtr, char *string)`	
	Converts `string` to proper list element and appends to `dsPtr`'s value (with separator space if needed). Returns new value of `dsPtr`.
`Tcl_DStringFree(Tcl_DString *dsPtr)`	
	Frees up any memory allocated for `dsPtr` and reinitializes `*dsPtr` to an empty string.
`Tcl_DStringResult(Tcl_Interp *interp, Tcl_DString *dsPtr)`	
	Moves the value of `dsPtr` to `interp->result` and reinitializes `dsPtr`'s value to an empty string.
`int Tcl_CommandComplete(char *cmd)`	
	Returns `1` if `cmd` holds one or more complete commands, `0` if the last command in `cmd` is incomplete due to open braces etc.
`int Tcl_StringMatch(char *string, char *pattern)`	
	Returns `1` if `string` matches `pattern` using glob-style rules for pattern matching, `0` otherwise.

The code below uses all of these procedures to implement a `map` command, which takes a list and generates a new list by applying some operation to each element of the original list. `Map` takes two arguments: a list and a Tcl command. For each element in the list, it executes the given command with the list element appended as an additional argument. It takes the results of all the commands and generates a new list out of them, and then returns this list as its result. Here are some exmples of how you might use the `map` command:

```
proc inc x {expr $x+1}
map {4 18 16 19 -7} inc
5 19 17 20 -6
proc addz x {return "$x z"}
map {a b {a b c}} addz
{a z} {b z} {a b c z}
```

Here is the command procedure that implements `map`:

```
int MapCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
```

```
    Tcl_DString command, newList;
    int listArgc, i, result;
    char **listArgv;
    if (argc != 3) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (Tcl_SplitList(interp, argv[1], &listArgc,
            &listArgv) != TCL_OK) {
        return TCL_ERROR;
    }
    Tcl_DStringInit(&newList);
    Tcl_DStringInit(&command);
    for (i = 0; i < listArgc; i++) {
        Tcl_DStringAppend(&command, argv[2], -1);
        Tcl_DStringAppendElement(&command,
                listArgv[i]);
        result = Tcl_Eval(interp, command.string);
        Tcl_DStringFree(&command);
        if (result != TCL_OK) {
            Tcl_DStringFree(&newList);
            free((char *) listArgv);
            return result;
        }
        Tcl_DStringAppendElement(&newList,
                interp->result);
    }
    Tcl_DStringResult(interp, &newList);
    free((char *) listArgv);
    return TCL_OK;
}
```

MapCmd uses two dynamic strings. One holds the result list and the other holds the command to execute in each step. The first dynamic string is needed because the length of the command is unpredictable, and the second one is needed to store the result list as it builds up (this information can't be placed immediately in interp->result because the interpreter's result will be overwritten by the command that's evaluated to process the next list element). Each dynamic string is represented by a structure of type Tcl_DString. The structure holds information about the string such as a pointer to its current value, a small array to use for small strings, and a length. The only field that you should ever access is the string field, which is a pointer to the current value. Tcl doesn't allocate Tcl_DString structures; it's up to you to allocate the structure (e.g. as a local variable) and pass its address to the dynamic string library procedures.

After checking its argument count, extracting all of the elements from the initial list, and initializing its dynamic strings, MapCmd enters a loop to process the elements of the list. For each element it first creates the command to execute for that element. It does this by calling `Tcl_DStringAppend` to append the part of the command provided in `argv[2]`, then it calls `Tcl_DStringAppendElement` to append the list element as an additional argument. These procedures are similar in that both add new information to the dynamic string. However, `Tcl_DStringAppend` adds the information as raw text whereas `Tcl_DStringAppendElement` converts its string argument to a proper list element and adds that list element to the dynamic string (with a separator space, if needed). It's important to use `Tcl_DStringAppendElement` for the list element so that it becomes a single word of the Tcl command being formed. If `Tcl_DStringAppend` were used instead and the element were "`a b c`" as in the example on page 310, then the command passed to `Tcl_Eval` would be "`addz a b c`" , which would result in an error (too many arguments to the `addz` procedure). When `Tcl_DStringAppendElement` is used the command is "`addz {a b c}`", which parses correctly.

Once `MapCmd` has created the command to execute for an element, it invokes `Tcl_Eval` to evaluate the command. The `Tcl_DStringFree` call frees up any memory that was allocated for the command string and resets the dynamic string to an empty value for use in the next command. If the command returned an error then `MapCmd` returns that same error; otherwise it uses `Tcl_DStringAppendElement` to add the result of the command to the result list as a new list element.

`MapCmd` calls `Tcl_DStringResult` after all of the list elements have been processed. This transfers the value of the string to the interpreter's result in an efficient way (e.g. if the dynamic string uses dynamically allocated memory then `Tcl_DStringResult` just copies a pointer to the result to `interp->result` rather than allocating new memory and copying the string).

Before returning, `MapCmd` must be sure to free up any memory allocated for the dynamic strings. It turns out that this has already been done by `Tcl_DStringFree` for `command` and by `Tcl_DStringResult` for `newList`.

34.2 Command completeness

When an application is reading commands typed interactively, it's important to wait until a complete command has been entered before evaluating it. For example, suppose an application is reading commands from standard input and the user types the following three lines:

```
foreach i {1 2 3 4 5} {
    puts "$i*$i is [expr $i*$i]"
}
```

If the application reads each line separately and passes it to `Tcl_Eval`, a "`missing close-brace`" error will be generated by the first line. Instead, the application should collect input until all the commands read are complete (e.g. there are no unmatched braces or quotes) then execute all of the input as a single script. The procedure Tcl_CommandComplete makes this possible. It takes a string as argument and returns 1 if the string contains syntactically complete commands, 0 if the last command isn't yet complete.

The C procedure below uses dynamic strings and Tcl_CommandComplete to read and evaluate a command typed on standard input. It collects input until all the commands read are complete, then it evaluates the command(s) and returns the completion code from the evaluation. It uses Tcl_RecordAndEval to evaluate the command so that the command is recorded on the history list.

```
int DoOneCmd(Tcl_Interp *interp) {
    char line[200];
    Tcl_DString cmd;
    int result;
    Tcl_DStringInit(&cmd);
    while (1) {
        if (fgets(line, 200, stdin) == NULL) {
            break;
        }
        Tcl_DStringAppend(&cmd, line, -1);
        if (Tcl_CommandComplete(cmd.string)) {
            break;
        }
    }
    result = Tcl_RecordAndEval(interp, cmd.string, 0);
    Tcl_DStringFree(&cmd);
    return result;
}
```

In the example of the previous page DoOneCmd will collect all three lines before evaluating them. If an end-of-file occurs fgets will return NULL and DoOneCmd will evaluate the command even if it isn't complete yet.

34.3 String matching

The procedure Tcl_StringMatch provides the same functionality as the "string match" Tcl command. Given a string and a pattern, it returns 1 if the string matches the pattern using glob-style matching and 0 otherwise. For example, here is a command procedure that uses Tcl_StringMatch to implement lsearch. It returns the index of the first element in a list that matches a pattern, or -1 if no element matches:

```
int LsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    int listArgc, i, result;
    char **listArgv;
    if (argc != 3) {
```

```
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    if (Tcl_SplitList(interp, argv[1], &listArgc,
            &listArgv) != TCL_OK) {
        return TCL_ERROR;
    }
    result = -1;
    for (i = 0; i < listArgc; i++) {
        if (Tcl_StringMatch(listArgv[i], argv[2])) {
            result = i;
            break;
        }
    }
    sprintf(interp->result, "%d", result);
    free((char *) listArgv);
    return TCL_OK;
}
```

Chapter 35
POSIX Utilities

This chapter describes several utilities that you may find useful if you use POSIX system calls in your C code. The procedures can be used to expand “~” notation in file names, to generate messages for POSIX errors and signals, and to manage sub-processes. See Table 35.1 for a summary of the procedure.

35.1 Tilde expansion

Tcl and Tk allow you to use ~ notation when specifying file names, and if you write new commands that manipulate files then you should support tildes also. For example, the command

```
open ~ouster/.login
```

opens the file named `.login` in the home directory of user `ouster`, and

```
open ~/.login
```

opens a file named `.login` in the home directory of the current user (as given by the `HOME` environment variable). Unfortunately, tildes are not supported by the POSIX system calls that actually open files. For example, in the first `open` command above the name actually presented to the `open` system call must be something like

```
/users/ouster/.login
```

where `~ouster` has been replaced by the home directory for `ouster`. `Tcl_TildeSubst` is the procedure that carries out this substitution. It is used internally by Tcl and Tk

`char *Tcl_TildeSubst(Tcl_Interp *interp, char *name, Tcl_DString *resultPtr)`	If `name` starts with ~, returns a new name with the ~ and following characters replaced with the corresponding home directory name. If `name` doesn't start with ~, returns `name`. Uses `*resultPtr` if needed to hold new name (caller need not initialize `*resultPtr`, but must free it by calling `Tcl_DStringFree`). If an error occurs, returns `NULL` and leaves an error message in `interp->result`.
`char *Tcl_PosixError(Tcl_Interp *interp)`	Sets the `errorCode` variable in `interp` based on the current value of `errno`, and returns a string identifying the error.
`char *Tcl_ErrnoId(void)`	Returns a symbolic name corresponding to the current value of `errno`, such as `ENOENT`.
`char *Tcl_SignalId(int sig)`	Returns the symbolic name for `sig`, such as `SIGINT`.
`char *Tcl_SignalMsg(int sig)`	Returns a human-readable message describing signal `sig`.
`int Tcl_CreatePipeline(Tcl_Interp *interp, int argc, char *argv[], int **pidPtr, int *inPipePtr, int *outPipePtr, int *errFilePtr)`	Creates a process pipeline, returns a count of the number of processes created, and stores at `*pidPtr` the address of a `malloc`-ed array of process identifiers. If an error occurs, returns `-1` and leaves an error message in `interp->result`. `InPipePtr`, `outPipePtr`, and `errFilePtr` are used to control default I/O redirection (see text for details).
`Tcl_DetachPids(int numPids, int *pidPtr)`	Passes responsibility for `numPids` at `*pidPtr` to Tcl: Tcl will allow them to run in backround and reap them in some future call to `Tcl_ReapDetachedProcs`.
`Tcl_ReapDetachedProcs(void)`	Checks to see if any detached processes have exited; if so, cleans up their state.

to process file names before using them in system calls, and you may find it useful if you write C code that deals with POSIX files.

For example, the implementation of the `open` command contains code something like the following:

```
int fd;
Tcl_DString buffer;
char *fullName;
...
```

```
fullName = Tcl_TildeSubst(interp, argv[1], &buffer);
if (fullName == NULL) {
    return TCL_ERROR;
}
fd = open(fullName, ...);
Tcl_DStringFree(fullName);
...
```

`Tcl_TildeSubst` takes as arguments an interpreter, a file name that may start with a tilde, and a dynamic string. It returns a new file name, which is either the original name (if it didn't start with ~), a new tilde-expanded name, or `NULL` if an error occurred; in the last case an error message is left in the interpreter's result.

If `Tcl_TildeSubst` has to generate a new name, it uses the dynamic string given by its final argument to store the name. When `Tcl_TildeSubst` is called the dynamic string should either be uninitialized or empty. `Tcl_TildeSubst` initializes it and then uses it for the new name if needed. Once the caller has finished using the new file name it must invoke `Tcl_DStringFree` to release any memory that was allocated for the dynamic string.

35.2 Generating messages

When an error or signal occurs in the C code of a Tcl application, the application should report the error or signal back to the Tcl script that triggered it, usually as a Tcl error. To do this, information about the error or signal must be converted from the binary form used in C to a string form for use in Tcl scripts. Tcl provides four procedures to do this: `Tcl_PosixError`, `Tcl_ErronId`, `Tcl_SignalId`, and `Tcl_SignalMsg`.

`Tcl_PosixError` provides a simple "all in one" mechanism for reporting errors in system calls. `Tcl_PosixError` examines the C variable `errno` to determine what kind of error occurred, then it calls `Tcl_SetErrorCode` to set the `errorCode` variable appropriately and it returns a human-readable string suitable for use in an error message. For example, consider the following fragment of code, which might be part of a command procedure:

```
FILE *f;
...
f = fopen("prolog.ps", "r");
if (f == NULL) {
    char *msg = Tcl_PosixError(interp);
    Tcl_AppendResult(interp,
            "couldn't open prolog.ps: ", msg,
            (char *) NULL);
    return TCL_ERROR;
}
```

If the file doesn't exist or isn't readable then an error will occur when `fopen` invokes a system call to open the file. An integer code will be stored in the `errno` variable to identify the error and `fopen` will return a null pointer. The above code detects such errors and invokes `Tcl_PosixError`. If the file didn't exist then `Tcl_PosixError` will set `errorCode` to

```
POSIX ENOENT {no such file or directory}
```

and return the string "`no such file or directory`". The code above incorporates `Tcl_PosixError`'s return value into its own error message, which it stores in `interp->result`. In the case of an non-existent file, the code above will return "`couldn't open prolog.ps: no such file or directory`" as its error message.

`Tcl_ErrnoId` takes no arguments and returns the official POSIX name for the error indicated by `errno`. The names are the symbolic ones defined in the header file `errno.h`. For example, if `errno`'s value is `ENOENT` then `Tcl_ErrnoId` will return the string "`ENOENT`". The return value from `Tcl_ErrnoId` is the same as the value that `Tcl_PosixError` will store in the second element of `errorCode`.

`Tcl_SignalId` and `Tcl_SignalMsg` each take a POSIX signal number as argument, and each returns a string describing the signal. `Tcl_SignalId` returns the official POSIX name for the signal as defined in `signal.h`, and `Tcl_SignalMsg` returns a human-readable message describing the signal. For example,

```
Tcl_SignalId(SIGILL)
```

returns the string "`SIGILL`", and

```
Tcl_SignalMsg(SIGILL)
```

returns "`illegal instruction`".

35.3 Creating subprocesses

`Tcl_CreatePipeline` is the procedure that does most of the work of creating subprocesses for `exec` and `open`. It creates one or more subprocesses in a pipeline configuration. It has the following arguments and result:

```
int Tcl_CreatePipeline(Tcl_Interp *interp, int argc,
        char *argv[], int **pidPtr, int *inPipePtr,
        int *outPipePtr, int *errFilePtr)
```

The `argc` and `argv` arguments describe the commands for the subprocesses in the same form they would be specified to `exec`. Each string in `argv` becomes one word of one command, except for special strings like ">" and "|" that are used for I/O redirection and separators between commands. `Tcl_CreatePipeline` normally returns a count of the number of subprocesses created, and it stores at `*pidPtr` a pointer to an array containing the process identifiers for the new processes. The array is dynamically allocated and must be freed by the caller by passing it to free. If an error occurred while spawning the subprocesses (e.g. argc and argv specified that output should be redirected to a file but the file couldn't be opened) then Tcl_CreatePipeline returns -1 and leaves an error message in interp->result.

The last three arguments to Tcl_CreatePipeline are used to control I/O to and from the pipeline if argv and argc don't specify I/O redirection. If these arguments are NULL then the first process in the pipeline will takes its standard input from the standard input of the parent, the last process will write its standard output to the standard output of the parent, and all of the processes will use the parent's standard error channel for their error message. If inPipePtr is not NULL then it points to an integer; Tcl_CreatePipeline will create a pipe, connect its output to the standard input of the first subprocess, and store a writable file descriptor for its input at *inPipePtr. If outPipePtr is not NULL then standard output goes to a pipe and a read descriptor for the pipe is stored at *outPipePtr. If errFilePtr is not NULL then Tcl_CreatePipeline creates a temporary file and connects the standard error files for all of the subprocesses to that file; a readable descriptor for the file will be stored at *errFilePtr. Tcl_CreatePipeline removes the file before it returns, so the file will only exist as long as it is open.

If argv specifies input or output redirection then this overrides the requests made in the arguments to Tcl_CreatePipeline. For example, if argv redirects standard input then no pipe is created for standard input; if inPipePtr is not NULL then -1 is stored at *inPipePtr to indicate that standard input was redirected. If argv redirects standard output then no pipe is created for it; if outPipePtr is not NULL then -1 is stored at *outPipePtr. If argv redirects some or all of the standard error output and errFilePtr is not NULL, the file will still be created and a descriptor will be returned, even though it's possible that no messages will actually appear in the file.

35.4 Background processes

Tcl_DetachPids and Tcl_ReapDetachedProcs are used to keep track of processes executing in the background. If an application creates a subprocess and abandons it (i.e. the parent never invokes a system call to wait for the child to exit), then the child executes in background and when it exits it becomes a "zombie". It remains a zombie until its parent officially waits for it or until the parent exits. Zombie processes occupy space in the system's process table, so if you create enough of them you will overflow the process table and make it impossible for anyone to create more processes. To keep this from happening, you must invoke a system call such as waitpid, which will return the exit status of the zombie process. Once the status has been returned the zombie relinquishes its slot in the process table.

In order to prevent zombies from overflowing the process table you should pass the process identifiers for background processes to Tcl_DetachPids:

```
Tcl_DetachPids(int numPids, int *pidPtr)
```

The `pidPtr` argument points to an array of process identifiers and `numPids` gives the size of the array. Each of these processes now becomes the property of Tcl and the caller should not refer to them again. Tcl will assume responsibility for waiting for the processes after they exit.

In order for Tcl to clean up background processes you may need to call `Tcl_ReapDetachedProcs` from time to time. `Tcl_ReapDetachedProcs` invokes the `waitpid` kernel call on each detached process so that its state can be cleaned up if it has exited. If some of the detached processes are still executing then `Tcl_ReapDetachedProcs` doesn't actually wait for them to exit; it only cleans up the processes that have already exited. Tcl automatically invokes `Tcl_ReapDetachedProcs` each time `Tcl_CreatePipeline` is invoked, so under normal circumstances you won't ever need to invoke it. However, if you create processes without calling `Tcl_CreatePipeline` (e.g. by invoking the `fork` system call) and subsequently pass the processes to `Tcl_DetachPids`, then you should also invoke `Tcl_ReapDetachedProcs` from time to time. For example, a good place to call `Tcl_ReapDetachedProcs` is in the code that creates new subprocesses.