CHAPTER 44

C Programming and Tcl

This chapter explains how to extend a Tcl application with new built-in commands. Tcl 8.0 replaces the original string-based command interface with a more efficient dual-ported object interface. This chapter describes both interfaces.

This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.

http://www.beedub.com/book/

Tcl is implemented in a C library that is easy to integrate into an existing application. By adding the Tcl interpreter to your application, you can configure and control it with Tcl scripts, and with Tk you can provide a nice graphical interface to it. This was the original model for Tcl. Applications would be largely application-specific C code and include a small amount of Tcl for configuration and the graphical interface. However, the basic Tcl shells proved so useful by themselves that relatively few Tcl programmers need to worry about programming in C or C++.

Tcl is designed to be easily extensible by writing new command implementations in C. A command implemented in C is more efficient than an equivalent Tcl procedure. A more pressing reason to write C code is that it may not be possible to provide the same functionality purely in Tcl. Suppose you have a new device, perhaps a color scanner or a unique input device. The programming interface to that device is through a set of C procedures that initialize and manipulate the state of the device. Without some work on your part, that interface is not accessible to your Tcl scripts. You are in the same situation if you have a C or C++ library that implements some specialized function such as a database. Fortunately, it is rather straightforward to provide a Tcl interface that corresponds to the C or C++ interface.

Note: Where this chapter says "C", you can always think "C or C++". There is also a package called *TclBlend* that lets you extend Tcl by writing Java instead of C, and to evaluate Tcl scripts from Java. Find out more about *TclBlend* at:

```
http://www.scriptics.com/java/
```

Basic Concepts

This chapter assumes that you know some C or C++. You do not have to be an expert programmer to use the Tcl APIs. Indeed, one of Tcl's strengths is the ease with which you can extend it by writing C code. This chapter provides a few working examples that explain how to initialize your application and create Tcl commands. It describes how to organize your code into packages. It concludes with notes about compiling Tcl under UNIX, Windows, and Macintosh.

Getting Started

There are two ways to get started writing C code for Tcl applications. The easiest way is to write an *extension* that just adds some new commands to a standard Tcl shell like *tclsh* or *wish*. With this approach the Tcl shell creates a basic framework for you, and your C code just extends this framework with new commands. Tcl supports dynamic loading, so you can compile your extension as a shared library (i.e., DLL) and load it into a running Tcl shell. This is the easiest approach because the Tcl shell handles the details of startup and shutdown, and it provides an interactive console to enter Tcl commands. In the case of *wish*, it also provides the framework for a graphical user interface. Finally, a loadable extension can be shared easily with other Tcl users.

The second way to use the Tcl library is to add it to an existing application. If your application is very simple, it may make sense to turn it into an extension for a standard Tcl shell, which brings you back to the first, simpler approach. However, if your application already has a complex framework (e.g., it is a long-running server process), then you can just add Tcl to it and export the functionality of your application as one or more Tcl commands. Once you do this, you will find that you can extend your application with all the features provided by Tcl.

C Command Procedures and Data Objects

The C or C++ code that implements a Tcl command is called a *command procedure*. The interface to a command procedure is much like the interface to a main program. The inputs are an array of values that correspond exactly to the arguments in the Tcl script command. The result of the command procedure becomes the result of the Tcl command.

There are two kinds of command procedures: string-based and "object-based." I've quoted "object" here because we are really talking about the data representation of the arguments and results. We are not talking about methods and inheritance and other things associated with object oriented programming. However, the Tcl C APIs use a structure called a `Tcl_Obj`, which is called a *dual ported object* in the reference material. I prefer the term "`Tcl_Obj` value".

The string interface is quite simple. A command procedure gets an array of strings as arguments, and it computes a string as the result. Tcl 8.0 generalized strings into the `Tcl_Obj` type, which can have two representations: both a string and another native representation like an integer, floating point number, list, or

bytecodes. An object-based command takes an array of Tcl_Obj pointers as arguments, and it computes a Tcl_Obj as its result. The goal of the Tcl_Obj type is to reduce the number of conversions between strings and native representations. Object-based commands will be more efficient than the equivalent string-based commands, but the APIs are a little more complex. For simple tasks, and for learning, you can use just the simpler string-based command interface.

SWIG

David Beasley created a nice tool called SWIG (Simple Wrapper Interface Generator) that generates the C code that implements command procedures that expose a C or C++ API as Tcl commands. This can be a great time saver if you need to export many calls to Tcl. The only drawback is that a C interface may not feel that comfortable to the script writer. Handcrafted Tcl interfaces can be much nicer, but automatically-generated interfaces are just fine for rapid prototyping and for software testing environments. You can learn more about SWIG at its web site:

```
http://www.swig.org/
```

Tcl Initialization

Before you can use your command procedures from Tcl scripts, you need to register them with Tcl. In some cases, you may also need to create the Tcl interpreter, although this is done for you by the standard Tcl shells.

If you are writing an extension, then you must provide an initialization procedure. The job of this procedure is to register Tcl commands with Tcl_CreateCommand or Tcl_CreateObjCommand. This is shown in Example 44–1 on page 608. The name of this procedure must end with _Init, as in Expect_Init, Blt_Init, or Foo_Init, if you plan to create your extension as a shared library. This procedure is called automatically when the Tcl script loads your library with the load command, which is described on page 607.

If you need to create the Tcl interpreter yourself, then there are two levels of APIs. At the most basic level there is Tcl_CreateInterp. This creates an interpreter that includes the standard commands listed in Table 1–4 on page 22. You still have to initialize all your custom commands (e.g., by calling Foo_Init) and arrange to run a script using Tcl_Eval or Tcl_EvalFile. However, there are a lot of details to get right, and Tcl provides a higher level interface in Tcl_Main and Tcl_AppInit. Tcl_Main creates the interpreter for you, processes command line arguments to get an initial script to run, and even provides an interactive command loop. It calls out to Tcl_AppInit, which you provide, to complete the initialization of the interpreter. The use of Tcl_Main is shown in Example 44–13 on page 629. There are even more details to get right with a Tk application because of the window system and the event loop. These details are hidden behind Tk_Main, which makes a similar call out to Tk_AppInit that you provide to complete initialization.

Calling Out to Tcl Scripts

An application can call out to the script layer at any point, even inside command procedures. `Tcl_Eval` is the basic API for this, and there are several variations depending on how you pass arguments to the script. When you look up `Tcl_Eval` in the reference material, you will get a description of the whole family of `Tcl_Eval` procedures.

You can also set and query Tcl variables from C using the `Tcl_SetVar` and `Tcl_GetVar` procedures. Again, there are several variations on these procedures that account for different types, like strings or `Tcl_Obj` values, and scalar or array variables. The `Tcl_LinkVar` procedure causes a Tcl variable to mirror a C variable. Modifications to the Tcl variable are reflected in the C variable, and reading the Tcl variable always returns the C variable's value. `Tcl_LinkVar` is built on a more general variable tracing facility, which is exposed to Tcl as the `trace` command, and available as the `Tcl_TraceVar` C API.

I think a well-behaved extension should provide both a C and Tcl API, but most of the core Tcl and Tk commands do not provide an exported C API. This forces you to eval Tcl scripts to get at their functionality. Example 44–15 on page 635 shows the `Tcl_Invoke` procedure that can help you work around this limitation. `Tcl_Invoke` is used to invoke a Tcl command without the parsing and substitution overhead of `Tcl_Eval`.

Using the Tcl C Library

Over the years the Tcl C Library has grown from a simple language interpreter into a full featured library. An important property of the Tcl API is that it is cross platform: its works equally well on UNIX, Windows, and Macintosh. One can argue that it is easier to write cross-platform applications in Tcl than in Java! Some of the useful features that you might not expect from a language interpreter include:

- A general hash table package that automatically adjusts itself as the hash table grows. It allows various types of keys, including strings and integers.
- A dynamic string (i.e., `DString`) package that provides an efficient way to construct strings.
- An I/O channel package that replaces the old "standard I/O library" found on UNIX with something that is cross-platform, does buffering, allows non-blocking I/O, and does character set translations. You can create new I/O channel types.
- Network sockets for TCP/IP communication.
- Character set translations between Unicode, UTF-8, and other encodings.
- An event loop manager that interfaces with network connections and window system events. You can create new "event sources" that work with the event loop manager.
- Multithreading support in the form of mutexes, condition variables, and thread-local storage.

- A registration system for exit handlers that are called when Tcl is shutting down.

This Chapter focuses just on the Tcl C API related to the Tcl interpreter. Chapter 47 gives a high-level overview of all the procedures in the Tcl and Tk C library, but this book does not provide a complete reference. Refer to the on-line manual pages for the specific details about each procedure; they are an excellent source of information. The manual pages should be part of every Tcl distribution. They are on the book's CD, and they can be found web at:

```
http://www.scriptics.com/man/
```

The Tcl source code is worth reading.

Finally, it is worth emphasizing that the source code of the Tcl C library is a great source of information. The code is well written and well commented. If you want to see how something really works, reading the code is worthwhile.

Creating a Loadable Package

You can organize your C code into a loadable package that can be dynamically linked into *tclsh*, *wish*, or your own Tcl application. The details about compiling the code into the shared library that contains the package are presented in Chapter 45. This section describes a package that implements the `random` Tcl command that returns random numbers.

The `load` Command

The Tcl `load` command is used to dynamically link in a compiled package:

```
load library package ?interp?
```

The *`library`* is the file name of the shared library file (i.e., the DLL), and *`package`* is the name of the package implemented by the library. This name corresponds to the *`package`*`_Init` procedure called to initialize the package (e.g., `Random_Init`) The optional *`interp`* argument lets you load the library into a slave interpreter. If the library is in `/usr/local/lib/random.so`, then a Tcl script can `load` the package like this:

```
load /usr/local/lib/random.so Random
```

On most UNIX systems, you can set the `LD_LIBRARY_PATH` environment variable to a colon-separated list of directories that contain shared libraries. If you do that, then you can use relative names for the libraries:

```
load librandom.so Random
```

On Macintosh, the `load` command looks for libraries in the same folder as the Tcl/Tk application (i.e., *Wish*) and in the `System:Extensions:Tool Command Language` folder:

```
load random.shlib Random
```

On Windows, `load` looks in the same directory as the Tcl/Tk application, the current directory, the `C:\Windows\System` directory (or `C:\Windows\System32` on

Windows NT), the C:\Windows directory, and then the directories listed in the PATH environment variable.

```
load random.dll Random
```

Fortunately, you usually do not have to worry about these details because the Tcl package facility can manage your libraries for you. Instead of invoking load directly, your scripts can use package require instead. The package facility keeps track of where your libraries are and knows how to call load for your platform. It is described in Chapter 12.

The Package Initialization Procedure

When a package is loaded, Tcl calls a C procedure named *package*_Init, where *package* is the name of your package. Example 44–1 defines Random_Init. It registers a command procedure, RandomCmd, that implements a new Tcl command, random. When the Tcl script uses the random command, the RandomCmd procedure will be invoked by the Tcl interpreter. Two styles of command registrations are made for comparison: the original Tcl_CreateCommand and the Tcl_CreateObjCommand added in Tcl 8.0. The command procedures are described in the next section:

Example 44–1 The initialization procedure for a loadable package.

```
/*
 * random.c
 */
#include <tcl.h>
/*
 * Declarations for application-specific command procedures
 */

int RandomCmd(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);
int RandomObjCmd(ClientData clientData,
              Tcl_Interp *interp,
              int objc, Tcl_Obj *CONST objv[]);

/*
 * Random_Init is called when the package is loaded.
 */

int Random_Init(Tcl_Interp *interp) {
   /*
    * Initialize the stub table interface, which is
    * described in Chapter 45.
    */

   if (Tcl_InitStubs(interp, "8.1", 0) == NULL) {
      return TCL_ERROR;
   }
```

```
    /*
     * Register two variations of random.
     * The orandom command uses the object interface.
     */

    Tcl_CreateCommand(interp, "random", RandomCmd,
           (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
    Tcl_CreateObjCommand(interp, "orandom", RandomObjCmd,
           (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);

    /*
     * Declare that we implement the random package
     * so scripts that do "package require random"
     * can load the library automatically.
     */
    Tcl_PkgProvide(interp, "random", "1.1");
    return TCL_OK;
}
```

Using `Tcl_PkgProvide`

`Random_Init` uses `Tcl_PkgProvide` to declare what package is provided by the C code. This call helps the `pkg_mkIndex` procedure learn what libraries provide which packages. `pkg_mkIndex` saves this information in a package database, which is a file named `pkgIndex.tcl`. The `package require` command looks for the package database files along your `auto_path` and automatically loads your package. The general process is:

- Create your shared library and put it into a directory listed on your `auto_path` variable, or a subdirectory of one of the directories on your `auto_path`.
- Run the `pkg_mkIndex` procedure in that directory, giving it the names of all the script files and shared libraries it should index. Now your shared library is ready for use by other scripts.
- A script uses `package require` to request a package. The correct `load` command for your system will be used the first time a command from your package is used. The `package` command is the same on all platforms:

  ```
  package require random
  => 1.1
  ```

  This process is explained in more detail on page 165.

A C Command Procedure

Tcl 8.0 introduced a new interface for Tcl commands that is designed to work efficiently with its internal on-the-fly byte code compiler. The original interface to commands was string oriented. This resulted in a lot of conversions between strings and internal formats such as integers, double-precision floating point numbers, and lists. The new interface is based on the `Tcl_Obj` type that can store

different types of values. Conversions between strings and other types are done in a lazy fashion, and the saved conversions help your scripts run more efficiently.

This section shows how to build a random number command using both interfaces. The string-based interface is simpler, and we start with that to illustrate the basic concepts. You can use it for your first experiments with command procedures. Once you gain some experience, you can start using the interfaces that use `Tcl_Obj` values instead of simple strings. If you have old command procedures from before Tcl 8.0, you need to update them only if you want extra efficiency. The string and `Tcl_Obj` interfaces are very similar, so you should find updating your command procedures straightforward.

The String Command Interface

The string-based interface to a C command procedure is much like the interface to the `main` program. You register the command procedure like this:

```
Tcl_CreateCommand(interp, data, "cmd", CmdProc, DeleteProc);
```

When the script invokes `cmd`, Tcl calls `CmdProc` like this:

```
CmdProc(data, interp, argc, argv);
```

The `interp` is type `Tcl_Interp *`, and it is a general handle on the state of the interpreter. Most Tcl C APIs take this parameter. The `data` is type `ClientData`, which is an opaque pointer. You can use this to associate state with your command. You register this state along with your command procedure, and then Tcl passes it back to you when the command is invoked. This is especially useful with Tk widgets, which are explained in more detail in Chapter 46. Our simple `RandomCmd` command procedure does not use this feature, so it passes `NULL` into `Tcl_CreateCommand`. The `DeleteProc` is called when the command is destroyed, which is typically when the whole Tcl interpreter is being deleted. If your state needs to be cleaned up, you can do it then. `RandomCmd` does not use this feature, either.

The arguments from the Tcl command are available as an array of strings defined by an `argv` parameter and counted by an `argc` parameter. This is the same interface that a main program has to its command line arguments. Example 44–2 shows the `RandomCmd` command procedure:

Example 44–2 The `RandomCmd` C command procedure.

```
/*
 * RandomCmd --
 * This implements the random Tcl command. With no arguments
 * the command returns a random integer.
 * With an integer valued argument "range",
 * it returns a random integer between 0 and range.
 */
int
RandomCmd(ClientData clientData, Tcl_Interp *interp,
       int argc, char *argv[])
```

```
{
    int rand, error;
    int range = 0;
    if (argc > 2) {
        interp->result = "Usage: random ?range?";
        return TCL_ERROR;
    }
    if (argc == 2) {
        if (Tcl_GetInt(interp, argv[1], &range) != TCL_OK) {
            return TCL_ERROR;
        }
    }
    rand = random();
    if (range != 0) {
        rand = rand % range;
    }
    sprintf(interp->result, "%d", rand);
    return TCL_OK;
}
```

The return value of a Tcl command is really two things: a result string and a status code. The result is a string that is either the return value of the command as seen by the Tcl script, or an error message that is reported upon error. For example, if extra arguments are passed to the command procedure, it raises a Tcl error by doing this:

```
interp->result = "Usage: random ?range?";
return TCL_ERROR;
```

The `random` implementation accepts an optional argument that is a range over which the random numbers should be returned. The `argc` parameter is tested to see if this argument has been given in the Tcl command. `argc` counts the command name as well as the arguments, so in our case `argc == 2` indicates that the command has been invoked something like:

```
random 25
```

The procedure `Tcl_GetInt` converts the string-valued argument to an integer. It does error checking and sets the interpreter's result field in the case of error, so we can just return if it fails to return `TCL_OK`.

```
if (Tcl_GetInt(interp, argv[1], &range) != TCL_OK) {
    return TCL_ERROR;
}
```

Finally, the real work of calling `random` is done, and the result is formatted directly into the result buffer using `sprintf`. A normal return looks like this:

```
sprintf(interp->result, "%d", rand);
return TCL_OK;
```

Result Codes from Command Procedures

The command procedure returns a status code that is either TCL_OK or TCL_ERROR to indicate success or failure. If the command procedure returns TCL_ERROR, then a Tcl error is raised, and the result value is used as the error message. The procedure can also return TCL_BREAK, TCL_CONTINUE, TCL_RETURN, which affects control structure commands like foreach and proc. You can even return an application-specific code (e.g., 5 or higher), which might be useful if you are implementing new kinds of control structures. The status code returned by the command procedure is the value returned by the Tcl catch command, which is discussed in more detail on page 77.

Managing the String Result

There is a simple protocol that manages the storage for a command procedure's result string. It involves interp->result, which holds the value, and interp->freeProc, which determines how the storage is cleaned up. When a command is called, the interpreter initializes interp->result to a static buffer of TCL_RESULT_SIZE, which is 200 bytes. The default cleanup action is to do nothing. These defaults support two simple ways to define the result of a command. One way is to use sprintf to format the result in place:

```
sprintf(interp->result, "%d", rand);
```

Using sprintf is suitable if you know your result string is short, which is often the case. The other way is to set interp->result to the address of a constant string. In this case, the original result buffer is not used, and there is no cleanup required because the string is compiled into the program:

```
interp->result = "Usage: random ?random?";
```

In more general cases, the following procedures should be used to manage the result and freeProc fields. These procedures automatically manage storage for the result:

```
Tcl_SetResult(interp, string, freeProc)
Tcl_AppendResult(interp, str1, str2, str3, (char *)NULL)
Tcl_AppendElement(interp, string)
```

Tcl_SetResult sets the return value to be *string*. The *freeProc* argument describes how the result should be disposed of: TCL_STATIC is used in the case where the result is a constant string allocated by the compiler, TCL_DYNAMIC is used if the result is allocated with Tcl_Alloc, which is a platform- and compiler-independent version of malloc, and TCL_VOLATILE is used if the result is in a stack variable. In the TCL_VOLATILE case, the Tcl interpreter makes a copy of the result before calling any other command procedures. Finally, if you have your own memory allocator, pass in the address of the procedure that should free the result.

Tcl_AppendResult copies its arguments into the result buffer, reallocating the buffer if necessary. The arguments are concatenated onto the end of the existing result, if any. Tcl_AppendResult can be called several times to build a

result. The result buffer is overallocated, so several appends are efficient.

`Tcl_AppendElement` adds the string to the result as a proper Tcl list element. It might add braces or backslashes to get the proper structure.

If you have built up a result and for some reason want to throw it away (e.g., an error occurs), then you can use `Tcl_ResetResult` to restore the result to its initial state. `Tcl_ResetResult` is always called before a command procedure is invoked.

The `Tcl_Obj` Command Interface

The `Tcl_Obj` command interface replaces strings with *dual-ported values*. The arguments to a command are an array of pointers to `Tcl_Obj` structures, and the result of a command is also of type `Tcl_Obj`. The replacement of strings by `Tcl_Obj` values extends throughout Tcl. The value of a Tcl variable is kept in a `Tcl_Obj`, and Tcl scripts are stored in a `Tcl_Obj`, too. You can continue to use the old string-based API, which converts strings to `Tcl_Obj` values, but this conversion adds overhead.

The `Tcl_Obj` structure stores both a string representation and a native representation. The native representation depends on the type of the value. Tcl lists are stored as an array of pointers to strings. Integers are stored as 32-bit integers. Floating point values are stored in double-precision. Tcl scripts are stored as sequences of byte codes. Conversion between the native representation and a string are done upon demand. There are APIs for accessing `Tcl_Obj` values, so you do not have to worry about type conversions unless you implement a new type. Example 44–3 shows the `random` command procedure using the `Tcl_Obj` interfaces:

Example 44–3 The `RandomObjCmd` C command procedure.

```
/*
 * RandomObjCmd --
 * This implements the random Tcl command from
 * Example 44-2 using the object interface.
 */
int
RandomObjCmd(ClientData clientData, Tcl_Interp *interp,
       int objc, Tcl_Obj *CONST objv[])
{
   Tcl_Obj *resultPtr;
   int rand, error;
   int range = 0;
   if (objc > 2) {
      Tcl_WrongNumArgs(interp, 1, objv, "?range?");
      return TCL_ERROR;
   }
   if (objc == 2) {
      if (Tcl_GetIntFromObj(interp, objv[1], &range) !=
            TCL_OK) {
         return TCL_ERROR;
```

```
        }
    }
    rand = random();
    if (range != 0) {
        rand = rand % range;
    }
    resultPtr = Tcl_GetObjResult(interp);
    Tcl_SetIntObj(resultPtr, rand);
    return TCL_OK;
}
```

Compare Example 44–2 with Example 44–3. You can see that the two versions of the C command procedures are similar. The `Tcl_GetInt` call is replaced with `Tcl_GetIntFromObj` call. This receives an integer value from the command argument. This call can avoid conversion from string to integer if the `Tcl_Obj` value is already an integer.

The result is set by getting a handle on the result object and setting its value. This is done instead of accessing the `interp->result` field directly:

```
resultPtr = Tcl_GetObjResult(interp);
Tcl_SetIntObj(resultPtr, rand);
```

The `Tcl_WrongNumArgs` procedure is a convenience procedure that formats an error message. You pass in `objv`, the number of arguments to use from it, and additional string. The example creates this message:

```
wrong # args: should be "random ?range?"
```

Example 44–3 does not do anything obvious about storage management. Tcl initializes the result object before calling your command procedure and takes care of cleaning it up later. It is sufficient to set a value and return `TCL_OK` or `TCL_ERROR`. In more complex cases, however, you have to worry about reference counts to `Tcl_Obj` values. This is described in more detail later.

If your command procedure returns a string, then you will use `Tcl_SetStringObj`. This command makes a copy of the string you pass it. The new Tcl interfaces that take strings also take length arguments so you can pass binary data in strings. If the length is minus 1, then the string is terminated by a NULL byte. A command that always returned "boring" would do this:

```
resultPtr = Tcl_GetObjResult(interp);
Tcl_SetStringObj(resultPtr, "boring", -1);
```

This is a bit too boring. In practice you may need to build up the result piecemeal. With the string-based API, you use `Tcl_AppendResult`. With the `Tcl_Obj` API you get a pointer to the result and use `Tcl_AppendToObj` or `Tcl_AppendStringsToObj`:

```
resultPtr = Tcl_GetObjResult(interp);
Tcl_AppendStringsToObj(resultPtr, "hello ", username, NULL);
```

Managing `Tcl_Obj` Reference Counts

The string-based interfaces copy strings when passing arguments and returning results, but the `Tcl_Obj` interfaces manipulate reference counts to avoid these copy operations. References come from Tcl variables, from the interpreter's result, and from sharing caused when a value is passed into a Tcl procedure. Constants are also shared. When a C command procedure is called, Tcl does not automatically increment the reference count on the arguments. However, each `Tcl_Obj` referenced by `objv` will have at least one reference, and it is quite common to have two or more references.

The C type definition for `Tcl_Obj` is shown below. There are APIs to access all aspects of an object, so you should refrain from manipulating a `Tcl_Obj` directly unless you are implementing a new type:

Example 44–4 The `Tcl_Obj` structure.

```
typedef struct Tcl_Obj {
    int refCount;       /* Counts number of shared references */
    char *bytes;        /* String representation */
    int length;         /* Number of bytes in the string */
    Tcl_ObjType *typePtr;/* Type implementation */
    union {
        long longValue;  /* Type data */
        double doubleValue;
        VOID *otherValuePtr;
        struct {
            VOID *ptr1;
            VOID *ptr2;
        } twoPtrValue;
    } internalRep;
} Tcl_Obj;
```

Each type implementation provides a few procedures like this:

```
Tcl_Get<Type>FromObj(interp, objPtr, valuePtr);
Tcl_Set<Type>Obj(resultPtr, value);
objPtr = Tcl_New<Type>Obj(value);
```

The initial reference count is zero.

The `Tcl_New`*`Type`*`Obj` allocates storage for a `Tcl_Obj` and sets its reference count to zero. `Tcl_IncrRefCount` and `Tcl_DecrRefCount` increment and decrement the reference count on an object. `Tcl_DecrRefCount` frees the storage for `Tcl_Obj` when it goes to zero. The initial reference count of zero was chosen because functions like `Tcl_SetObjResult` automatically increment the reference count on an object.

The `Tcl_Get`*`Type`*`FromObj` and `Tcl_Set`*`Type`*`Obj` procedures just get and set the value; the reference count does not change. Type conversions are automatic. You can set a `Tcl_Obj` value to an integer and get back a string or double precision number later. The type implementations automatically take care of the storage for the `Tcl_Obj` value as it changes. Of course, if a `Tcl_Obj` stays the same type, then no string conversions are necessary and accesses are more efficient.

Modifying Tcl_Obj Values

It is not safe to modify a shared `Tcl_Obj`. The sharing is only for efficiency: Logically, each reference is a copy, and you must honor this model when creating and modifying `Tcl_Obj` values. `Tcl_IsShared` returns 1 if there is more than one reference to an object. If a command procedure modifies a shared object, it must make a private copy with `Tcl_DuplicateObj`. The new copy starts with a reference count of zero. You either pass this to `Tcl_SetResultObj`, which adds a reference, or you have to explicitly add a reference to the copy with `Tcl_IncrRefCount`.

Example 44–5 implements a `plus1` command that adds one to its argument. If the argument is not shared, then `plus1` can be implemented efficiently by modifying the native representation of the integer. Otherwise, it has to make a copy of the object before modifying it:

Example 44–5 The `Plus1ObjCmd` procedure.

```
/*
 * Plus1ObjCmd --
 * This adds one to its input argument.
 */
int
Plus1ObjCmd(ClientData clientData, Tcl_Interp *interp,
       int objc, Tcl_Obj *CONST objv[])
{
   Tcl_Obj *objPtr;
   int i;
   if (objc != 2) {
      Tcl_WrongNumArgs(interp, 1, objv, "value");
      return TCL_ERROR;
   }
   objPtr = objv[1];
   if (Tcl_GetIntFromObj(interp, objPtr, &i) != TCL_OK) {
      return TCL_ERROR;
   }
   if (Tcl_IsShared(objPtr)) {
      objPtr = Tcl_DuplicateObj(objPtr);   /* refCount 0 */
      Tcl_IncrRefCount(objPtr);            /* refCount 1*/
   }
   /*
    * Assert objPtr has a refCount of one here.
    * OK to set the unshared value to something new.
    * Tcl_SetIntObj overwrites the old value.
    */
   Tcl_SetIntObj(objPtr, i+1);
   /*
    * Setting the result object adds a new reference,
    * so we decrement because we no longer care about
    * the integer object we modified.
    */
   Tcl_SetObjResult(interp, objPtr);       /* refCount 2*/
   Tcl_DecrRefCount(objPtr);               /* refCount 1*/
   /*
```

```
     * Now only the interpreter result has a reference to objPtr.
     */
    return TCL_OK;
}
```

Pitfalls of Shared `Tcl_Obj` Values

You have to be careful when using the values from a `Tcl_Obj` structure. The Tcl C library provides many procedures like `Tcl_GetStringFromObj`, `Tcl_GetIntFromObj`, `Tcl_GetListFromObj`, and so on. These all operate efficiently by returning a pointer to the native representation of the object. They will convert the object to the requested type, if necessary. The problem is that shared values can undergo type conversions that may invalidate your reference to a particular type of the value.

Value references are only safe until the next `Tcl_Get*FromObj` call.

Consider a command procedure that takes two arguments, an integer and a list. The command procedure has a sequence of code like this:

```
Tcl_GetListFromObj(interp, objv[1], &listPtr);
/* Manipulate listPtr */
Tcl_GetIntFromObj(interp, objv[2], &int);
/* listPtr may be invalid here */
```

If, by chance, both arguments have the same value, (e.g., `1` and `1`), which is possible for a Tcl list and an integer, then Tcl will automatically arrange to share these values between both arguments. The pointers in `objv[1]` and `objv[2]` will be the same, and the reference count on the `Tcl_Obj` they reference will be at least 2. The first `Tcl_GetListFromObj` call ensures the value is of type list, and it returns a direct pointer to the native list representation. However, `Tcl_GetInFromObj` then helpfully converts the `Tcl_Obj` value to an integer. This deallocates the memory for the list representation, and now `listPtr` is a dangling pointer! This particular example can be made safe by reversing the calls because `Tcl_GetIntFromObj` copies the integer value:

```
Tcl_GetIntFromObj(interp, objv[2], &int);
Tcl_GetListFromObj(interp, objv[1], &listPtr);
/* int is still a good copy of the value */
```

By the way, you should always test your `Tcl_Get*` calls in case the format of the value is incompatible with the requested type. If the object is not a valid list, the following command returns an error:

```
if (Tcl_GetListFromObj(interp, obj[1], &listPtr) != TCL_OK) {
    return TCL_ERROR;
}
```

The `blob` Command Example

This section illustrates some standard coding practices with a bigger example. The example is still artificial in that it doesn't actually do very much. However, it illustrates a few more common idioms you should know about when creating Tcl commands.

The `blob` command creates and manipulates blobs. Each blob has a name and some associated properties. The `blob` command uses a hash table to keep track of blobs by their name. The hash table is an example of state associated with a command that needs to be cleaned up when the Tcl interpreter is destroyed. The Tcl hash table implementation is nice and general, too, so you may find it helpful in a variety of situations.

You can associate a Tcl script with a blob. When you poke the blob, it invokes the script. This shows how easy it is to associate behaviors with your C extensions. Example 44–6 shows the data structures used to implement blobs.

Example 44–6 The `Blob` and `BlobState` data structures.

```
/*
 * The Blob structure is created for each blob.
 */
typedef struct Blob {
   int N;              /* Integer-valued property */
   Tcl_Obj *objPtr;    /* General property */
   Tcl_Obj *cmdPtr;    /* Callback script */
} Blob;
/*
 * The BlobState structure is created once for each interp.
 */
typedef struct BlobState {
   Tcl_HashTable hash;        /* List blobs by name */
   int uid;                   /* Used to generate names */
} BlobState;
```

Creating and Destroying Hash Tables

Example 44–7 shows the `Blob_Init` and `BlobCleanup` procedures. `Blob_Init` creates the command and initializes the hash table. It registers a delete procedure, `BlobCleanup`, that will clean up the hash table.

The `Blob_Init` procedure allocates and initializes a hash table as part of the `BlobState` structure. This structure is passed into `Tcl_CreateObjCommand` as the `ClientData`, and gets passed back to `BlobCmd` later. You might be tempted to have a single static hash table structure instead of allocating one. However, it is quite possible that a process has many Tcl interpreters, and each needs its own hash table to record its own blobs.

When the hash table is initialized, you specify what the keys are. In this case, the name of the blob is a key, so `TCL_STRING_KEYS` is used. If you use an integer key, or the address of a data structure, use `TCL_ONE_WORD_KEYS`. You can

also have an array of integers (i.e., a chunk of data) for the key. In this case, pass in an integer larger than 1 that represents the size of the integer array used as the key.

The BlobCleanup command cleans up the hash table. It iterates through all the elements of the hash table and gets the value associated with each key. This value is cast into a pointer to a Blob data structure. This iteration is a special case because each entry is deleted as we go by the BlobDelete procedure. If you do not modify the hash table, you continue the search with Tcl_NextHashEntry instead of calling Tcl_FirstHashEntry repeatedly.

Example 44–7 The Blob_Init and BlobCleanup procedures.

```
/*
 * Forward references.
 */

int BlobCmd(ClientData data, Tcl_Interp *interp,
       int objc, Tcl_Obj *CONST objv[]);
int BlobCreate(Tcl_Interp *interp, BlobState *statePtr);
void BlobCleanup(ClientData data);

/*
 * Blob_Init --
 *
 *     Initialize the blob module.
 *
 * Side Effects:
 *     This allocates the hash table used to keep track
 *     of blobs. It creates the blob command.
 */
int
Blob_Init(Tcl_Interp *interp)
{
   BlobState *statePtr;
   /*
    * Allocate and initialize the hash table. Associate the
    * BlobState with the command by using the ClientData.
    */
   statePtr = (BlobState *)Tcl_Alloc(sizeof(BlobState));
   Tcl_InitHashTable(&statePtr->hash, TCL_STRING_KEYS);
   statePtr->uid = 0;
   Tcl_CreateObjCommand(interp, "blob", BlobCmd,
          (ClientData)statePtr, BlobCleanup);
   return TCL_OK;
}
/*
 * BlobCleanup --
 *     This is called when the blob command is destroyed.
 *
 * Side Effects:
 *     This walks the hash table and deletes the blobs it
*      contains. Then it deallocates the hash table.
 */
```

```
void
BlobCleanup(ClientData data)
{
    BlobState *statePtr = (BlobState *)data;
    Blob *blobPtr;
    Tcl_HashEntry *entryPtr;
    Tcl_HashSearch search;

    entryPtr = Tcl_FirstHashEntry(&statePtr->hash, &search);
    while (entryPtr != NULL) {
        blobPtr = Tcl_GetHashValue(entryPtr);
        BlobDelete(blobPtr, entryPtr);
        /*
         * Get the first entry again, not the "next" one,
         * because we just modified the hash table.
         */
        entryPtr = Tcl_FirstHashEntry(&statePtr->hash, &search);
    }
    Tcl_Free((char *)statePtr);
}
```

`Tcl_Alloc` and `Tcl_Free`

Instead of using `malloc` and `free` directly, you should use `Tcl_Alloc` and `Tcl_Free` in your code. Depending on compilation options, these procedures may map directly to the system's `malloc` and `free`, or use alternate memory allocators. The allocators on Windows and Macintosh are notoriously poor, and Tcl ships with a nice efficient memory allocator that is used instead. In general, it is not safe to allocate memory with `Tcl_Alloc` and free it with `free`, or allocate memory with `malloc` and free it with `Tcl_Free`.

When you look at the Tcl source code you will see calls to `ckalloc` and `ckfree`. These are macros that either call `Tcl_Alloc` and `Tcl_Free` or `Tcl_DbgAlloc` and `Tcl_DbgFree` depending on a compile-time option. The second set of functions is used to debug memory leaks and other errors. You cannot mix these two allocators either, so your best bet is to stick with `Tcl_Alloc` and `Tcl_Free` everywhere.

Parsing Arguments and `Tcl_GetIndexFromObj`

Example 44–8 shows the `BlobCmd` command procedure. This illustrates a basic framework for parsing command arguments. The `Tcl_GetIndexFromObj` procedure is used to map from the first argument (e.g., "names") to an index (e.g., `NamesIx`). This does error checking and formats an error message if the first argument doesn't match. All of the subcommands except "create" and "names" use the second argument as the name of a blob. This name is looked up in the hash table with `Tcl_FindHashEntry`, and the corresponding `Blob` structure is fetched using `Tcl_GetHashValue`. After the argument checking is complete, `BlobCmd` dispatches to the helper procedures to do the actual work:

Example 44–8 The BlobCmd command procedure.

```
/*
 * BlobCmd --
 *
 *     This implements the blob command, which has these
 *     subcommands:
 *         create
 *         command name ?script?
 *         data name ?value?
 *         N name ?value?
 *         names ?pattern?
 *         poke name
 *         delete name
 *
 * Results:
 *     A standard Tcl command result.
 */
int
BlobCmd(ClientData data, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[])
{
    BlobState *statePtr = (BlobState *)data;
    Blob *blobPtr;
    Tcl_HashEntry *entryPtr;
    Tcl_Obj *valueObjPtr;

    /*
     * The subCmds array defines the allowed values for the
    * first argument. These are mapped to values in the
    * BlobIx enumeration by Tcl_GetIndexFromObj.
    */

    char *subCmds[] = {
        "create", "command", "data", "delete", "N", "names",
        "poke", NULL
    };
    enum BlobIx {
        CreateIx, CommandIx, DataIx, DeleteIx, NIx, NamesIx,
        PokeIx
    };
    int result, index;

    if (objc == 1 || objc > 4) {
        Tcl_WrongNumArgs(interp, 1, objv, "option ?arg ...?");
        return TCL_ERROR;
    }
    if (Tcl_GetIndexFromObj(interp, objv[1], subCmds,
            "option", 0, &index) != TCL_OK) {
        return TCL_ERROR;
    }
    if (((index == NamesIx || index == CreateIx) &&
            (objc > 2)) ||
        ((index == PokeIx || index == DeleteIx) &&
            (objc == 4))) {
```

```
        Tcl_WrongNumArgs(interp, 1, objv, "option ?arg ...?");
        return TCL_ERROR;
    }
    if (index == CreateIx) {
        return BlobCreate(interp, statePtr);
    }
    if (index == NamesIx) {
        return BlobNames(interp, statePtr);
    }
    if (objc < 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
            "option blob ?arg ...?");
        return TCL_ERROR;
    } else if (objc == 3) {
        valueObjPtr = NULL;
    } else {
        valueObjPtr = objv[3];
    }
    /*
     * The rest of the commands take a blob name as the third
     * argument. Hash from the name to the Blob structure.
     */
    entryPtr = Tcl_FindHashEntry(&statePtr->hash,
            Tcl_GetString(objv[2]));
    if (entryPtr == NULL) {
        Tcl_AppendResult(interp, "Unknown blob: ",
                Tcl_GetString(objv[2]), NULL);
        return TCL_ERROR;
    }
    blobPtr = (Blob *)Tcl_GetHashValue(entryPtr);
    switch (index) {
        case CommandIx: {
            return BlobCommand(interp, blobPtr, valueObjPtr);
        }
        case DataIx: {
            return BlobData(interp, blobPtr, valueObjPtr);
        }
        case NIx: {
            return BlobN(interp, blobPtr, valueObjPtr);
        }
        case PokeIx: {
            return BlobPoke(interp, blobPtr);
        }
        case DeleteIx: {
            return BlobDelete(blobPtr, entryPtr);
        }
    }
}
```

Creating and Removing Elements from a Hash Table

The real work of BlobCmd is done by several helper procedures. These form the basis of a C API to operate on blobs as well. Example 44–9 shows the BlobCreate and BlobDelete procedures. These procedures manage the hash table entry, and they allocate and free storage associated with the blob.

Example 44–9 BlobCreate and BlobDelete.

```
int
BlobCreate(Tcl_Interp *interp, BlobState *statePtr)
{
    Tcl_HashEntry *entryPtr;
    Blob *blobPtr;
    int new;
    char name[20];
    /*
     * Generate a blob name and put it in the hash table
     */
    statePtr->uid++;
    sprintf(name, "blob%d", statePtr->uid);
    entryPtr = Tcl_CreateHashEntry(&statePtr->hash, name, &new);
    /*
     * Assert new == 1
     */
    blobPtr = (Blob *)Tcl_Alloc(sizeof(Blob));
    blobPtr->N = 0;
    blobPtr->objPtr = NULL;
    blobPtr->cmdPtr = NULL;
    Tcl_SetHashValue(entryPtr, (ClientData)blobPtr);
    /*
     * Copy the name into the interpreter result.
     */
    Tcl_SetStringObj(Tcl_GetObjResult(interp), name, -1);
    return TCL_OK;
}
int
BlobDelete(Blob *blobPtr, Tcl_HashEntry *entryPtr)
{
    Tcl_DeleteHashEntry(entryPtr);
    if (blobPtr->cmdPtr != NULL) {
        Tcl_DecrRefCount(blobPtr->cmdPtr);
    }
    if (blobPtr->objPtr != NULL) {
        Tcl_DecrRefCount(blobPtr->objPtr);
    }
    /*
     * Use Tcl_EventuallyFree because of the Tcl_Preserve
     * done in BlobPoke. See page 626.
     */
    Tcl_EventuallyFree((char *)blobPtr, Tcl_Free);
    return TCL_OK;
}
```

Building a List

The BlobNames procedure iterates through the elements of the hash table using Tcl_FirstHashEntry and Tcl_NextHashEntry. It builds up a list of the names as it goes along. Note that the object reference counts are managed for us. The Tcl_NewStringObj returns a Tcl_Obj with reference count of zero. When that object is added to the list, the Tcl_ListObjAppendElement procedure increments the reference count. Similarly, the Tcl_NewListObj returns a Tcl_Obj with reference count zero, and its reference count is incremented by Tcl_SetObjResult:

Example 44–10 The BlobNames procedure.

```
int
BlobNames(Tcl_Interp *interp, BlobState *statePtr)
{
    Tcl_HashEntry *entryPtr;
    Tcl_HashSearch search;
    Tcl_Obj *listPtr;
    Tcl_Obj *objPtr;
    char *name;
    /*
     * Walk the hash table and build a list of names.
     */
    listPtr = Tcl_NewListObj(0, NULL);
    entryPtr = Tcl_FirstHashEntry(&statePtr->hash, &search);
    while (entryPtr != NULL) {
        name = Tcl_GetHashKey(&statePtr->hash, entryPtr);
        if (Tcl_ListObjAppendElement(interp, listPtr,
                Tcl_NewStringObj(name, -1)) != TCL_OK) {
            return TCL_ERROR;
        }
        entryPtr = Tcl_NextHashEntry(&search);
    }
    Tcl_SetObjResult(interp, listPtr);
    return TCL_OK;
}
```

Keeping References to Tcl_Obj Values

A blob has two simple properties: an integer N and a general Tcl_Obj value. You can query and set these properties with the BlobN and BlobData procedures. The BlobData procedure keeps a pointer to its Tcl_Obj argument, so it must increment the reference count on it:

Example 44–11 The BlobN and BlobData procedures.

```
int
BlobN(Tcl_Interp *interp, Blob *blobPtr, Tcl_Obj *objPtr)
{
    int N;
    if (objPtr != NULL) {
        if (Tcl_GetIntFromObj(interp, objPtr, &N) != TCL_OK) {
            return TCL_ERROR;
        }
        blobPtr->N = N;
    } else {
        N = blobPtr->N;
    }
    Tcl_SetObjResult(interp, Tcl_NewIntObj(N));
    return TCL_OK;
}
int
BlobData(Tcl_Interp *interp, Blob *blobPtr, Tcl_Obj *objPtr)
{
    if (objPtr != NULL) {
        if (blobPtr->objPtr != NULL) {
            Tcl_DecrRefCount(blobPtr->objPtr);
        }
        Tcl_IncrRefCount(objPtr);
        blobPtr->objPtr = objPtr;
    }
    if (blobPtr->objPtr != NULL) {
        Tcl_SetObjResult(interp, blobPtr->objPtr);
    }
    return TCL_OK;
}
```

Using Tcl_Preserve and Tcl_Release to Guard Data

The BlobCommand and BlobPoke operations let you register a Tcl command with a blob and invoke the command later. Whenever you evaluate a Tcl command like this, you must be prepared for the worst. It is quite possible for the command to turn around and delete the blob it is associated with! The Tcl_Preserve, Tcl_Release, and Tcl_EventuallyFree procedures are used to handle this situation. BlobPoke calls Tcl_Preserve on the blob before calling Tcl_Eval. BlobDelete calls Tcl_EventuallyFree instead of Tcl_Free. If the Tcl_Release call has not yet been made, then Tcl_EventuallyFree just marks the memory for deletion, but does not free it immediately. The memory is freed later by Tcl_Release. Otherwise, Tcl_EventuallyFree frees the memory directly and Tcl_Release does nothing. Example 44–12 shows BlobCommand and BlobPoke:

Example 44–12 The `BlobCommand` and `BlobPoke` procedures.

```
int
BlobCommand(Tcl_Interp *interp, Blob *blobPtr,
    Tcl_Obj *objPtr)
{
    if (objPtr != NULL) {
        if (blobPtr->cmdPtr != NULL) {
            Tcl_DecrRefCount(blobPtr->cmdPtr);
        }
        Tcl_IncrRefCount(objPtr);
        blobPtr->cmdPtr = objPtr;
    }
    if (blobPtr->cmdPtr != NULL) {
        Tcl_SetObjResult(interp, blobPtr->cmdPtr);
    }
    return TCL_OK;
}
int
BlobPoke(Tcl_Interp *interp, Blob *blobPtr)
{
    int result = TCL_OK;
    if (blobPtr->cmdPtr != NULL) {
        Tcl_Preserve(blobPtr);
        result = Tcl_EvalObj(interp, blobPtr->cmdPtr);
        /*
         * Safe to use blobPtr here
         */
        Tcl_Release(blobPtr);
        /*
         * blobPtr may not be valid here
         */
    }
    return result;
}
```

It turns out that `BlobCmd` does not actually use the `blobPtr` after calling `Tcl_EvalObj`, so it could get away without using `Tcl_Preserve` and `Tcl_Release`. These procedures do add some overhead: They put the pointer onto a list of preserved pointers and have to take it off again. If you are careful, you can omit these calls. However, it is worth noting the potential problems caused by evaluating arbitrary Tcl scripts!

Strings and Internationalization

There are two important topics related to string handling: creating strings dynamically and translating strings between character set encodings. These issues do not show up in the simple examples we have seen so far, but they will arise in more serious applications.

The DString Interface

It is often the case that you have to build up a string from pieces. The Tcl_DString data type and a related API are designed to make this efficient. The DString interface hides the memory management issues, and the Tcl_DString data type starts out with a small static buffer, so you can often avoid allocating memory if you put a Tcl_String type on the stack (i.e., as a local variable). The standard code sequence goes something like this:

```
Tcl_DString ds;
Tcl_DStringInit(&ds);
Tcl_DStringAppend(&ds, "some value", -1);
Tcl_DStringAppend(&ds, "something else", -1);
Tcl_DStringResult(interp, &ds);
```

The Tcl_DStringInit call initializes a string pointer inside the structure to point to a static buffer that is also inside the structure. The Tcl_DStringAppend call grows the string. If it would exceed the static buffer, then a new buffer is allocated dynamically and the string is copied into it. The last argument to Tcl_DStringAppend is a length, which can be minus 1 if you want to copy until the trailing NULL byte in your string. You can use the string value as the result of your Tcl command with Tcl_DStringResult. This passes ownership of the string to the interpreter and automatically cleans up the Tcl_DString structure.

If you do not use the string as the interpreter result, then you must call Tcl_DStringFree to ensure that any dynamically allocated memory is released:

```
Tcl_DStringFree(&ds);
```

You can get a direct pointer to the string you have created with Tcl_DStringValue:

```
name = Tcl_DStringValue(&ds);
```

There are a handful of additional procedures in the DString API that you can read about in the reference material. There are some that create lists, but this is better done with the Tcl_Obj interface (e.g., Tcl_NewListObj and friends).

To some degree, a Tcl_Obj can replace the use of a Tcl_DString. For example, the Tcl_NewStringObj and Tcl_AppendToObj allocate a Tcl_Obj and append strings to it. However, there are a number of Tcl API procedures that take Tcl_DString types as arguments instead of the Tcl_Obj type. Also, for small strings, the DString interface is still more efficient because it can do less dynamic memory allocation.

Character Set Conversions

As described in Chapter 15, Tcl uses UTF-8 strings internally. UTF-8 is a representation of Unicode that does not contain NULL bytes. It also represents 7-bit ASCII characters in one byte, so if you have old C code that only manipulates ASCII strings, it can coexist with Tcl without modification.

However, in more general cases, you may need to convert between UTF-8 strings you get from Tcl_Obj values to strings of a particular encoding. For

example, when you pass strings to the operating system, it expects them in its native encoding, which might be 16-bit Unicode, ISO-Latin-1 (i.e., iso-8859-1), or something else.

Tcl provides an encoding API that does translations for you. The simplest calls use a `Tcl_DString` to store the results because it is not possible to predict the size of the result in advance. For example, to convert from a UTF-8 `string` to a `Tcl_DString` in the system encoding, you use this call:

```
Tcl_UtfToExternalDString(NULL, string, -1, &ds);
```

You can then pass `Tcl_DStringValue(&ds)` to your system call that expects a native string. Afterwards you need to call `Tcl_DStringFree(&ds)` to free up any memory allocated by `Tcl_UtfToExternalDString`.

To translate strings the other way, use `Tcl_ExternalToUtfDString`:

```
Tcl_ExternalToUtfDString(NULL, string, -1, &ds);
```

The third argument to `these procedures` is the length of `string` in bytes (not characters), and minus 1 means that Tcl should calculate it by looking for a NULL byte. Tcl stores its UTF-8 strings with a NULL byte at the end so it can do this.

The first argument to `these procedures` is the encoding to translate to or from. NULL means the system encoding. If you have data in nonstandard encodings, or need to translate into something other than the system encoding, you need to get a handle on the encoding with `Tcl_GetEncoding`, and free that handle later with `Tcl_FreeEncoding`:

```
encoding = Tcl_GetEncoding(interp, name);
Tcl_FreeEncoding(encoding);
```

The names of the encodings are returned by the `encoding names` Tcl command, and you can query them with a C API, too.

Windows has a quirky string data type called `TCHAR`, which is an 8-bit byte on Windows 95/98, and a 16-bit Unicode character on Windows NT. If you use a C API that takes an array of `TCHAR`, then you have to know what kind of system you are running on to use it properly. Tcl provides two procedures that deal with this automatically. `Tcl_WinTCharToUf` works like `Tcl_ExternalToUtfDString`, and `Tcl_WinUtfToTChar` works like `Tcl_UtfToExternalDString`:

```
Tcl_WinUtfToTChar(string, -1, &ds);
Tcl_WinTCharToUtf(string, -1, &ds);
```

Finally, Tcl has several procedures to work with Unicode characters, which are type `Tcl_UniChar`, and UTF-8 encoded characters. Examples include `Tcl_UniCharToUtf`, `Tcl_NumUtfChars`, and `Tcl_UtfToUniCharDString`. Consult the reference materials for details about these procedures.

`Tcl_Main` and `Tcl_AppInit`

This section describes how to make a custom main program that includes Tcl. However, the need for custom main programs has been reduced by the use of loadable modules. If you create your commands as a loadable package, you can

just load them into *tclsh* or *wish*. Even if you do not need a custom main, this section will explain how all the pieces fit together.

The Tcl library supports the basic application structure through the Tcl_Main procedure that is designed to be called from your main program. Tcl_Main does three things:

- It calls Tcl_CreateInterp to create an interpreter that includes all the standard Tcl commands like set and proc. It also defines a few Tcl variables like argc and argv. These have the command-line arguments that were passed to your application.
- It calls Tcl_AppInit, which is not part of the Tcl library. Instead, your application provides this procedure. In Tcl_AppInit you can register additional application-specific Tcl commands.
- It reads a script or goes into an interactive loop.

You call Tcl_Main from your main program and provide an implementation of the Tcl_AppInit procedure:

Example 44–13 A canonical Tcl main program and Tcl_AppInit.

```
/* main.c */
#include <tcl.h>
int Tcl_AppInit(Tcl_Interp *interp);
/*
 * Declarations for application-specific command procedures
 */
int Plus1ObjCmd(ClientData clientData,
             Tcl_Interp *interp,
             int objc, Tcl_Obj *CONST objv[]);

main(int argc, char *argv[]) {
   /*
    * Initialize your application here.
    *
    * Then initialize and run Tcl.
    */
   Tcl_Main(argc, argv, Tcl_AppInit);
   exit(0);
}
/*
 * Tcl_AppInit is called from Tcl_Main
 * after the Tcl interpreter has been created,
 * and before the script file
 * or interactive command loop is entered.
 */
int
Tcl_AppInit(Tcl_Interp *interp) {
   /*
    * Tcl_Init reads init.tcl from the Tcl script library.
    */
   if (Tcl_Init(interp) == TCL_ERROR) {
      return TCL_ERROR;
```

```
    }
    /*
     * Register application-specific commands.
     */
    Tcl_CreateObjCommand(interp, "plus1", Plus1ObjCmd,
           (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
    Random_Init(interp);
    Blob_Init(interp);
    /*
     * Define the start-up filename. This file is read in
     * case the program is run interactively.
     */
    Tcl_SetVar(interp, "tcl_rcFileName", "~/.mytcl",
       TCL_GLOBAL_ONLY);
    /*
     * Test of Tcl_Invoke, which is defined on page 635.
     */
    Tcl_Invoke(interp, "set", "foo", "$xyz [foo] {", NULL);
    return TCL_OK;
}
```

The `main` program calls `Tcl_Main` with the `argc` and `argv` parameters passed into the program. These are the strings passed to the program on the command line, and `Tcl_Main` will store these values into Tcl variables by the same name. `Tcl_Main` is also given the address of the initialization procedure, which is `Tcl_AppInit` in our example. `Tcl_AppInit` is called by `Tcl_Main` with one argument, a handle on a newly created interpreter. There are three parts to the `Tcl_AppInit` procedure:

- The first part initializes the various packages the application uses. The example calls `Tcl_Init` to set up the script library facility described in Chapter 12. The core Tcl commands have already been defined by `Tcl_CreateInterp`, which is called by `Tcl_Main` before the call to `Tcl_AppInit`.
- The second part of `Tcl_AppInit` does application-specific initialization. The example registers the command procedures defined earlier in this Chapter.
- The third part defines a Tcl variable, `tcl_RcFileName`, which names an application startup script that executes if the program is used interactively.

You can use your custom program just like *tclsh*, except that it includes the additional commands you define in your `Tcl_AppInit` procedure. The sample makefile on the CD creates a program named *mytcl*. You can compile and run that program and test `random` and the other commands.

Tk_Main

The structure of Tk applications is similar. The `Tk_Main` procedure creates a Tcl interpreter and the main Tk window. It calls out to a procedure you provide to complete initialization. After your `Tk_AppInit` returns, `Tk_Main` goes into an

event loop until all the windows in your application have been destroyed.

Example 44–14 shows a Tk_AppInit used with Tk_Main. The main program processes its own command-line arguments using Tk_ParseArgv, which requires a Tcl interpreter for error reporting. The Tk_AppInit procedure initializes the clock widget example that is the topic of Chapter 46:

Example 44–14 A canonical Tk main program and Tk_AppInit.

```
/* main.c */
#include <tk.h>

int Tk_AppInit(Tcl_Interp *interp);

/*
 * A table for command line arguments.
 */
char *myoption1 = NULL;
int myint2 = 0;

static Tk_ArgvInfo argTable[] = {
   {"-myoption1", TK_ARGV_STRING, (char *) NULL,
      (char *) &myoption1, "Explain myoption1"},
   {"-myint2", TK_ARGV_CONSTANT, (char *) 1, (char *) &myint2,
      "Explain myint2"},
   {"", TK_ARGV_END, },
};

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

   /*
    * Create an interpreter for the error message from
    * Tk_ParseArgv. Another one is created by Tk_Main.
    * Parse our arguments and leave the rest to Tk_Main.
    */

   interp = Tcl_CreateInterp();
   if (Tk_ParseArgv(interp, (Tk_Window) NULL, &argc, argv,
          argTable, 0) != TCL_OK) {
      fprintf(stderr, "%s\n", interp->result);
      exit(1);
   }
   Tcl_DeleteInterp(interp);

   Tk_Main(argc, argv, Tk_AppInit);
   exit(0);
}
int ClockCmd(ClientData clientData,
             Tcl_Interp *interp,
             int argc, char *argv[]);
int ClockObjCmd(ClientData clientData,
             Tcl_Interp *interp,
             int objc, Tcl_Obj *CONST objv[]);
```

```
void ClockObjDestroy(ClientData clientData);

int
Tk_AppInit(Tcl_Interp *interp) {
    /*
     * Initialize packages
     */
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    /*
     * Define application-specific commands here.
     */
    Tcl_CreateCommand(interp, "wclock", ClockCmd,
        (ClientData)Tk_MainWindow(interp),
        (Tcl_CmdDeleteProc *)NULL);
    Tcl_CreateObjCommand(interp, "oclock", ClockObjCmd,
        (ClientData)NULL, ClockObjDestroy);

    /*
     * Define start-up filename. This file is read in
     * case the program is run interactively.
     */
    Tcl_SetVar(interp, "tcl_rcFileName", "~/.mytcl",
        TCL_GLOBAL_ONLY);
    return TCL_OK;
}
```

The Event Loop

An event loop is used to process window system events and other events like timers and network sockets. The different event types are described later. All Tk applications must have an event loop so that they function properly in the window system environment. Tk provides a standard event loop with the `Tk_MainLoop` procedure, which is called at the end of `Tk_Main`. The *wish* shell provides an event loop automatically. The *tclsh* shell does not, although you can add an event loop using pure Tcl as shown in Example 16–2 on page 220.

Some applications already have their own event loop. You have two choices if you want to add Tk to such an application. The first is to modify the existing event loop to call `Tcl_DoOneEvent` to process any outstanding Tcl events. The `unix` directory of the source distribution has a file called `XtTest.c` that adds Tcl to an Xt (i.e., Motif) application. The other way to customize the event loop is to make your existing events look like Tcl *event sources*, and register them with the event loop. Then you can just use `Tk_Main`. There are four event classes, and they are handled in the following order by `Tcl_DoOneEvent`:

- Window events. Use the `Tk_CreateEventHandler` procedure to register a handler for these events. Use the `TCL_WINDOW_EVENTS` flag to process these in `Tcl_DoOneEvent`.
- File events. Use these events to wait on slow devices and network connections. On UNIX you can register a handler for all files, sockets, and devices with `Tcl_CreateFileHandler`. On Windows and Macintosh, there are different APIs for registration because there are different system handles for files, sockets, and devices. On all platforms you use the `TCL_FILE_EVENTS` flag to process these handlers in `Tcl_DoOneEvent`.
- Timer events. You can set up events to occur after a specified time period. Use the `Tcl_CreateTimerHandler` procedure to register a handler for the event. Use the `TCL_TIMER_EVENTS` flag to process these in `Tcl_DoOneEvent`.
- Idle events. These events are processed when there is nothing else to do. Virtually all the Tk widgets use idle events to display themselves. Use the `Tcl_DoWhenIdle` procedure to register a procedure to call once at the next idle time. Use the `TCL_IDLE_EVENTS` flag to process these in `Tcl_DoOneEvent`.

Invoking Scripts from C

The main program is not the only place you can evaluate a Tcl script. You can use the `Tcl_Eval` procedure essentially at any time to evaluate a Tcl command:

```
Tcl_Eval(Tcl_Interp *interp, char *command);
```

The `command` is evaluated in the current Tcl procedure scope, which may be the global scope. This means that calls like `Tcl_GetVar` and `Tcl_SetVar` access variables in the current scope. If for some reason you want a new procedure scope, the easiest thing to do is to call your C code from a Tcl procedure used for this purpose. It is not easy to create a new procedure scope with the exported C API.

`Tcl_Eval` modifies its argument.

You should be aware that `Tcl_Eval` may modify the string that is passed into it as a side effect of the way substitutions are performed. If you pass a constant string to `Tcl_Eval`, make sure your compiler has not put the string constant into read-only memory. If you use the *gcc* compiler, you may need to use the `-fwritable-strings` option. Chapter 45 shows how to get the right compilation settings for your system.

Variations on Tcl_Eval

There are several variations on `Tcl_Eval`. The possibilities include strings or `Tcl_Obj` values, evaluation at the current or global scope, a single string (or `Tcl_Obj` value) or a variable number of arguments, and optional byte-code compilation. The most general string-based eval is `Tcl_EvalEx`, which takes a counted string and some flags:

```
Tcl_EvalEx(interp, string, count, flags);
```

The flags are TCL_GLOBAL_EVAL and TCL_EVAL_DIRECT, which bypasses the byte-code compiler. For code that is executed only one time, TCL_EVAL_DIRECT may be more efficient. Tcl_GlobalEval is equivalent to passing in the TCL_GLOBAL_EVAL flag. The Tcl_VarEval procedure takes a variable number of strings arguments and concatenates them before evaluation:

```
Tcl_VarEval(Tcl_Interp *interp, char *str, ..., NULL);
```

Tcl_EvalObj takes an object as an argument instead of a simple string. The string is compiled into byte codes the first time it is used. If you are going to execute the script many times, then the Tcl_Obj value caches the byte codes for you. The general Tcl_Obj value interface to Tcl_Eval is Tcl_EvalObjEx, which takes the same flags as Tcl_EvalEx:

```
Tcl_EvalObjEx(interp, objPtr, flags);
```

For variable numbers of arguments, use Tcl_EvalObjv, which takes an array of Tcl_Obj pointers. This routine concatenates the string values of the various Tcl_Obj values before parsing the resulting Tcl command:

```
Tcl_EvalObjv(interp, objc, objv);
```

Bypassing Tcl_Eval

In a performance-critical situation, you may want to avoid some of the overhead associated with Tcl_Eval. David Nichols showed me how to call the implementation of a C command procedure directly. The trick is facilitated by the Tcl_GetCommandInfo procedure that returns the address of the C command procedure for a Tcl command, plus its client data pointer. The Tcl_Invoke procedure is shown in Example 44–15. It is used much like Tcl_VarEval, except that each of its arguments becomes an argument to the Tcl command without any substitutions being performed.

For example, you might want to insert a large chunk of text into a text widget without worrying about the parsing done by Tcl_Eval. You could use Tcl_Invoke like this:

```
Tcl_Invoke(interp, ".t", "insert", "insert", buf, NULL);
```

Or:

```
Tcl_Invoke(interp, "set", "foo", "$xyz [blah] {", NULL);
```

No substitutions are performed on any of the arguments because Tcl_Eval is out of the picture. The variable foo gets the following literal value:

```
$xyz [blah] {
```

Example 44–15 shows Tcl_Invoke. The procedure is complicated for two reasons. First, it must handle a Tcl command that has either the object interface or the old string interface. Second, it has to build up an argument vector and may need to grow its storage in the middle of building it. It is a bit messy to deal with both at the same time, but it lets us compare the object and string interfaces. The string interfaces are simpler, but the object interfaces run more efficiently because they reduce copying and type conversions.

Example 44–15 Calling C command procedure directly with `Tcl_Invoke`.

```
#include <tcl.h>

#if defined(__STDC__) || defined(HAS_STDARG)
#   include <stdarg.h>
#else
#   include <varargs.h>
#endif

/*
 * Tcl_Invoke --
 *     Directly invoke a Tcl command or procedure
 *
 *     Call Tcl_Invoke somewhat like Tcl_VarEval
 *     Each arg becomes one argument to the command,
 *     with no further substitutions or parsing.
 */
   /* VARARGS2 */ /* ARGSUSED */

int
Tcl_Invoke TCL_VARARGS_DEF(Tcl_Interp *, arg1)
{
   va_list argList;
   Tcl_Interp *interp;
   char *cmd;          /* Command name */
   char *arg;          /* Command argument */
   char **argv;        /* String vector for arguments */
   int argc, i, max;   /* Number of arguments */
   Tcl_CmdInfo info;   /* Info about command procedures */
   int result;         /* TCL_OK or TCL_ERROR */

   interp = TCL_VARARGS_START(Tcl_Interp *, arg1, argList);
   Tcl_ResetResult(interp);

   /*
    * Map from the command name to a C procedure
    */
   cmd = va_arg(argList, char *);
   if (! Tcl_GetCommandInfo(interp, cmd, &info)) {
      Tcl_AppendResult(interp, "unknown command \"",
         cmd, "\"", NULL);
      va_end(argList);
      return TCL_ERROR;
   }

   max = 20;           /* Initial size of argument vector */

#if TCL_MAJOR_VERSION > 7
   /*
    * Check whether the object interface is preferred for
    * this command
    */

   if (info.isNativeObjectProc) {
```

```
        Tcl_Obj **objv;     /* Object vector for arguments */
        Tcl_Obj *resultPtr; /* The result object */
        int objc;

        objv = (Tcl_Obj **) Tcl_Alloc(max * sizeof(Tcl_Obj *));
        objv[0] = Tcl_NewStringObj(cmd, strlen(cmd));
        Tcl_IncrRefCount(objv[0]); /* ref count == 1*/
        objc = 1;

        /*
         * Build a vector out of the rest of the arguments
         */

        while (1) {
           arg = va_arg(argList, char *);
           if (arg == (char *)NULL) {
              objv[objc] = (Tcl_Obj *)NULL;
              break;
           }
           objv[objc] = Tcl_NewStringObj(arg, strlen(arg));
           Tcl_IncrRefCount(objv[objc]); /* ref count == 1*/
           objc++;
           if (objc >= max) {
              /* allocate a bigger vector and copy old one */
              Tcl_Obj **oldv = objv;
              max *= 2;
              objv = (Tcl_Obj **) Tcl_Alloc(max *
                     sizeof(Tcl_Obj *));
              for (i = 0 ; i < objc ; i++) {
                 objv[i] = oldv[i];
              }
              Tcl_Free((char *)oldv);
           }
        }
        va_end(argList);

        /*
         * Invoke the C procedure
         */
        result = (*info.objProc)(info.objClientData, interp,
               objc, objv);

        /*
         * Make sure the string value of the result is valid
         * and release our references to the arguments
         */
        (void) Tcl_GetStringResult(interp);
        for (i = 0 ; i < objc ; i++) {
           Tcl_DecrRefCount(objv[i]);
        }
        Tcl_Free((char *)objv);

        return result;
     }
#endif
```

```
        argv = (char **) Tcl_Alloc(max * sizeof(char *));
        argv[0] = cmd;
        argc = 1;

        /*
         * Build a vector out of the rest of the arguments
         */
        while (1) {
            arg = va_arg(argList, char *);
            argv[argc] = arg;
            if (arg == (char *)NULL) {
                break;
            }
            argc++;
            if (argc >= max) {
                /* allocate a bigger vector and copy old one */
                char **oldv = argv;
                max *= 2;
                argv = (char **) Tcl_Alloc(max * sizeof(char *));
                for (i = 0 ; i < argc ; i++) {
                    argv[i] = oldv[i];
                }
                Tcl_Free((char *) oldv);
            }
        }
        va_end(argList);

        /*
         * Invoke the C procedure
         */
        result = (*info.proc)(info.clientData, interp, argc, argv);

        /*
         * Release the arguments
         */
        Tcl_Free((char *) argv);
        return result;

    }
```

This version of `Tcl_Invoke` was contributed by Jean Brouwers. He uses `TCL_VARARGS_DEF` and `TCL_VARARGS_START` macros to define procedures that take a variable number of arguments. These standard Tcl macros hide the differences in the way you do this on different operating systems and different compilers. It turns out that there are numerous minor differences between compilers that can cause portability problems in a variety of situations. Happily, there is a nice scheme used to discover these differences and write code in a portable way. This is the topic of the next chapter.