# Tcl/C++ Binding Made Easy*

John Menges, Brian Ladd
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
{menges,ladd}@cs.unc.edu

May 27, 1994

## Abstract

*This paper describes a type-safe binding mechanism for mapping Tcl commands to calls on member functions of particular C++ objects. Given an easy-to-write mapping class for each C++ type used in the signatures of the member functions to be called, our method makes it simple to implement a binding to any member function using any combination of these types.*

## 1 Introduction

The Artifact Based Collaboration System (ABC)[3] at the University of North Carolina provides computer support for synchronous and asynchronous collaboration among distributed users. ABC is not a single application but rather a cooperating collection of processes. The processes composing ABC have object-oriented designs and are implemented with C++.

Tcl/Tk has been adopted to provide control, monitoring, and communication among the various process-level and object-level components of ABC. A brief description of ABC and its use of Tk and Tcl is provided in a paper presented at the 1993 Tcl/Tk Workshop[1].

During system development it is often necessary to expose the interfaces of C++ objects at run-time. This is of particular interest when building scaffolding for exercising objects for debugging purposes but is also useful for building interfaces for users of the system. Such exposure requires an easy-to-use, type-safe binding mechanism for mapping Tcl commands to calls on member functions of particular C++ objects. The method described in this paper is an outgrowth of the one described in the appendix of our 1993 Tcl/Tk Workshop paper[1].

The method described in [1] was type-safe and effective, but cumbersome to use. Judicious use of C++ templates simplifies the specification of the Tcl/C++ bindings with no loss of safety or effectiveness.

The remainder of this paper presents the improved binding method and is laid out as follows: Section 2 lays out the problem in more detail and presents an example; Section 3 presents our solution using the same example to trace the programmer's changes to the code; Section 4 mentions additional areas we would like to explore with this technique.

*This work is supported in part by the National Science Foundation (Grant # IRI-9015443), and a Graduate Fellowship from the IBM Corporation.

# 2 The Problem

Suppose we wish to control and monitor a C++-based application process from a separate Tk-based front-end process. Our application process contains a Tcl interpreter object belonging to a subclass of Extended Tcl's **TclInterp_cl** class. Communication between the front-end and the application is performed using Tcl-DP[2]. The **TclInterp_cl** class enables the application to create new commands, implemented as C++ functions and accessible from the front-end via **dp_RPC** calls.

This works fine for controlling the application at the process level. We would like, however, to be able to interact with the individual objects comprising the application. For example, suppose we have a class **Employees** as defined in Figure 1.

```
class Employees
{
public:
  Employees();
  void insert(char *ssn,
              char *name,
              Date birthdate,
              int salary);
  Tuple get(char *ssn);
  int raise(char *ssn,
            int percent);
protected:
  Hashtable<Tuple> employees;
}
```

Figure 1: The Employees Class

Given a *handle*, **Employees#5**, on a particular object of type **Employees**, we would like to be able to call the member functions of the object as shown in Figure 2.

We want our member functions to be callable both from C++ and from Tcl. We also wish to do some validation and conversion operations, e.g., to make sure a date is specified in an acceptable format and to convert it to the **Date** object type. Finally, note that all member functions (including those with a **void** return type) should be capable of returning not only values in their natural range, but also Tcl errors.

```
tcl> Employees#5->insert 012-34-5678 \
       {John Doe} 6/23/94 20000
tcl> Employees#5->raise 012-34-5678 5
21000
tcl> set tuple \
       [Employees#5->get 012-34-5678]
012-34-5678 {John Doe} 6/23/94 21000
```

Figure 2: Tcl Access to Employees

# 3 The Solution

Our solution consists of three components: straightforward modifications of the member functions we wish to call from Tcl and the class to which they belong (the *target* member functions and class), a mapping class for each data type used in the signatures of these functions, and a supporting C++ class library. We will discuss each of these components of the solution in turn.

## 3.1 Target Class Modifications

To make the Employees class accessible from Tcl, we rewrite it as shown in Figure 3.

The **Employees** class now inherits from the **Handle** class (Figure 4) and defines a **dynamic_class** member function that returns the string "Employees". This enables us to associate a string handle with each object of type **Employees**. (We use handles of the form *Classname#unique_integer* and we do not reuse ids, to avoid the synchronization problems that can occur if a front-end

```
class Employees
: virtual public Handle
{
public:
  Employees();
  virtual char *dynamic_class();
  TclMap<void>& insert(TclMap<char *> ssn,
                       TclMap<char *> name,
                       TclMap<Date> birthdate,
                       TclMap<int> salary);
  TclMap<Tuple>& get(TclMap<char *> ssn);
  TclMap<int>& raise(TclMap<char *> ssn, TclMap<int> percent);
protected:
  Hashtable<Tuple> employees;
private:
  TclCmdCallback_4<Employees,
                   TclMap<void>&,
                   TclMap<char *>,
                   TclMap<char *>,
                   TclMap<Date>,
                   TclMap<int> > insert_callback;
  TclCmdCallback_1<Employees,
                   TclMap<Tuple>&,
                   TclMap<int> > get_callback;
  TclCmdCallback_2<Employees,
                   TclMap<int>&,
                   TclMap<char *>,
                   TclMap<int> > raise_callback;
}
```

Figure 3: Modified Employees Class

uses handles on objects that are frequently created and deleted.)

The TclMap template has a separate implementation for each of its argument types. These mapping classes convert Tcl strings to values of particular C++ data types (validating their representations along the way). They also perform mapping functions in the other direction, from C++ data type (or Tcl error) to Tcl string and return code. The implementation of TclMap classes is discussed in the next section.

The **TclCmdCallback** template classes (with the help of our **TclInterpreter** class, discussed later) implement the bindings between Tcl commands and the member functions of individual **Employees** objects. These classes are part of the supporting library. There must be one for member functions with zero arguments, one for member functions with one argument, etc; the integer suffix on the **TclCmdCallback** class names indicates the number of member function arguments. The arguments to the **TclCmdCallback** template classes specify the class of the target object and the signa-

```
class Handle
{
public:
  Handle();
  virtual ~Handle();
  virtual char *handle();
protected:
  virtual char *dynamic_class();
  ...
};
```

Figure 4: The Handle Class Interface

ture of the target member function.

Now, let's take a look at how the **Employees** class is implemented. First, the **TclCmdCallback** callbacks must be initialized as shown in Figure 5. Their constructors take three arguments: the command name (to which the handle and "->" are prepended), the **this** pointer, and the address of the target function.

```
Employees()
: insert_callback
    ("insert", this, &insert),
  get_callback("get", this, &get)
  raise_callback("raise", this, &raise)
{}
```

Figure 5: TclCmdCallback Initialization

We are now left with only the implementation of the target member functions. Figure 6 shows how this is done for the **raise** member function, which gives a particular employee a raise and returns the new salary. This example demonstrates that **TclMap**<*type*> objects can be used in the same manner as *type* objects, as **ssn** is used as an indexing operator to the **employees** object (which takes a **char *** argument) and the **percent** argument is used in an arithmetic expression just as an **int** would be used. This can accomplished by way of C++ operator and/or conversion operator overloading in the **TclMap** class. The expression on the **return** statement can also be of the natural (**int**) type as shown. This is accomplished via a constructor for the **TclMap** class.

```
TclMap<int>& Employees::
raise(TclMap<char *> ssn,
      TclMap<int> percent)
{
  static TclMap<int> result;
  employees[ssn].salary *=
    1.0 + percent/100.0;
  return result =
    employees[ssn].salary;
}
```

Figure 6: Member Function Implementation

We would have preferred to implement member functions as shown in Figure 7. Unfortunately a bug in one of our C++ compilers forces us to return the result through a reference to a static variable as shown in Figure 6.

```
TclMap<int> Employees::
raise(TclMap<char *> ssn,
      TclMap<int> percent)
{
  employees[ssn].salary *=
    1.0 + percent/100.0;
  return employees[ssn].salary;
}
```

Figure 7: Preferred Return Value Syntax

Now suppose we wish to return an error from the **raise** function if **percent** is less than 0. Suppose that if **raise** is called from C++, a negative return code is sufficient, while if it is called from Tcl, an error message should be returned. We can accomplish this through the use of the **TclError** class as shown in Figure 8. The **TclMap** class will have a constructor with a **TclError** argument that will store the error flag and error string in the **TclMap<int>** result object, from whence they can be retrieved by the calling code, either automatically via a conversion operator (returning -1) for a C++ call, or manually by the **TclCmdCallback** class which returns the result to Tcl.

```
TclMap<int>& Employees::
raise(TclMap<char *> ssn,
      TclMap<int> percent)
{
  static TclMap<int> result;
  if (percent < 0)
    return result =
      TclError("percent < 0");
  employees[ssn].salary *=
    1.0 + percent/100.0;
  return result =
    employees[ssn].salary;
}
```

Figure 8: Member Function Implementation

## 3.2 The TclMap Classes

The **TclMap** classes perform the mappings between Tcl strings and C++ data types and vice versa. The support library contains some of these mappers; others must be created by the implementor of the target classes.

The **TclMap** template class has a trivial default implementation with a single template argument specifying the C++ type, as shown in Figure 9. There must be a separate implementation of this class for each type used in the signatures of member functions exposed to Tcl. Using template syntax is convenient because name mangling is done by the C++ compiler. For example, a mapping class for **char *** can be called **TclMap<char *>** instead of **TclMap_char_asterisk**.

```
template<class TYPE>
class TclMap
{
};
```

Figure 9: Default TclMap

Consider the **TclMap<int>** implementation shown in Figure 11. The first constructor allows expressions of the form

```
return result = 1;
```

from within target functions returning the **TclMap<int>&** type. The second constructor allows these member functions to return errors to Tcl. The third constructor is the one used by the appropriate **TclCmdCallback** class; its responsibility is to parse an integer from the **argc** and **argv** arguments and update these arguments accordingly. Note that simple syntactic errors (not enough arguments, or a string that doesn't represent an **int**) can be caught at this time, so that the member function need never be called with invalid arguments. The **Tcl_GetInt** function should be used instead of the **atoi** function so that parsing errors can be reported; the latter was used here to simplify the example. The **int&** operator makes it possible for the **TclMap<int>** class to be used as an **int**, both within target function implementations having a **TclMap<int>** parameter and from calling C++ functions when the member function returns **TclMap<int>&**. It will not always be so easy to make **TclMap**<*type*> objects act like *type* objects; sometimes the *type* operators will have to be implemented in the **TclMap**<*type*> class itself and passed through to the **value** member, because C++ won't be able to tell what underlying *type* is intended. The **string_value** function returns a string representation for the type. This member function is used by the **TclCmdCallback** class to convert results back to Tcl strings. Again, a simple **sprintf** is used here for simplicity; a more robust method should be used.

```
class TclMap_SuperClass
{
public:
  TclMap_SuperClass();
  TclMap_SuperClass
    (const TclMap_SuperClass&);
  TclMap_SuperClass(const TclError&);

  virtual void set_error
    (const char *&);
  virtual char *string_value() = 0;
  virtual int error();

protected:
  int Error;
  char * Error_string;
};
```

Figure 10: TclMap_SuperClass

More elaborate mapping classes are, of course, possible. The natural C++ type can be any C++ type, not just predefined types like **int**. For example, a target member function argument's natural form might be a pointer to a user-defined class and the associated **TclMap** implementation can map between the object's handle and its pointer. Single mappers can also parse multiple arguments, or they can map optional arguments and provide the appropriate default if they are not specified. Simple mappers can also be defined to pass **argc**, **argv**, the **TclInterpreter** pointer or the **ClientData** pointer directly to target functions. The mappers needn't all use the **TclMap**<*type*> syntax, but they must be subclasses of the **TclMap_SuperClass** type.

## 3.3 The Support Library

In addition to mappers for common basic data types, the support library contains the **TclMap_SuperClass** (Figure 10) and **TclError** classes, the **TclCmdCallback** template classes and the **TclInterpreter** class.

The single argument version of the **TclCmdCallback** class is shown in Figure 12. The **target** and **function** parameters are saved by the constructor as members of the class, and the **interpreter** and **clientData** parameters, which have reasonable defaults, are passed up to the **TclCmdCallback** superclass to be saved, along with the Tcl command name. The **TclCmdCallback** superclass registers the callback with the interpreter. When the associated Tcl command is invoked, the function call operator is called with the **argc** and **argv** parameters received from Tcl. The function call operator, in turn, calls the various **TclMap** constructors to parse the arguments. After each of these, it checks to see if the **TclMap** constructor detected an error; after all are parsed it ensures that there are no straggling arguments. If there are parsing errors, it returns a Tcl error without calling the target member function. Otherwise it uses the C++ **->*** operator to call the target function, sets the Tcl result appropriately, and returns. The other **TclCmdCallback** classes are similar.

Our **TclInterpreter** class is a subclass of the Extended Tcl **TclInterp_cl** class. It defines its own versions of the **CreateCommand** member functions for registering **TclCmdCallback** objects, as shown in Figure 13. It type-casts the **callback** pointer to the **ClientData** type and registers a callback with Tcl using the

```
class TclMap<int>
: public TclMap_SuperClass
{
public:
  TclMap(const int value)
  : value(value)
  {
  }

  TclMap(const TclError& tcl_error)
  : TclMap_SuperClass(tcl_error),
    value(0)
  {
  }

  TclMap(int& argc, char**&argv, TclInterpreter *const, ClientData)
  : value(0)
  {
    if (argc-- < 1) set_error("too few arguments");
    else value = atoi(argv++[0]);
  }

  operator int&()
  {
    return value;
  }

  char *string_value()
  {
    if (Error) return Error_string;
    static char buffer[100];
    sprintf(buffer, "%d", value);
    return value;
  }

protected:
  int value;
};
```

Figure 11: TclMap<int>

```
template<class TARGET, class RETURNS, class ARGUMENT>
class TclCmdCallback_1
: public TclCmdCallback
{
public:
  typedef RETURNS (TARGET::*FUNCTION)(ARGUMENT);

  TclCmdCallback_1 (const char *const name,
                    TARGET *target,
                    FUNCTION function,
                    TclInterpreter *const interpreter =
                      Global::global->TclInterpreter_External,
                    ClientData clientData = 0)
  : TclCmdCallback(name, interpreter, clientData),
    target(target),
    function(function)
  {
  }

  virtual int operator() (int argc, char **argv) const
  {
    argc--; argv++;
    ARGUMENT argument = ARGUMENT(argc, argv, interpreter, clientData);
    if (argument.error())
      {
        interpreter->SetResult(argument.string_value());
        return argument.error();
      }
    if (argc > 0) {
      interpreter->SetResult("too many arguments");
      return TCL_ERROR;
    }
    RETURNS result((target->*function)(argument));
    interpreter->SetResult(result.string_value());
    return result.error();
  }

protected:
  TARGET *target;
  FUNCTION function;
};
```

Figure 12: TclCmdCallback_1

command name stored in the **callback** object and a single **CallCmdProc** function. The purpose of this function is to call the **TclCmdCallback** function call operator; its definition is shown in Figure 14.

```
void TclInterpreter::
CreateCommand
(const char *cmdName,
 Tcl_CmdProc *proc,
 ClientData clientData = NULL,
 Tcl_CmdDeleteProc *deleteProc = NULL)
{
  TclInterp_cl::CreateCommand
    (cmdName,
     proc,
     clientData,
     deleteProc);
}

void TclInterpreter::
CreateCommand
(TclCmdCallback *callback)
{
  CreateCommand(callback->name,
                CallCmdProc,
                ClientData(callback));
}
```

Figure 13: TclInterpreter::CreateCommand

```
static int CallCmdProc
(ClientData callback,
 Tcl_Interp *,
 int argc,
 char **argv)
{
  return
    (*(TclCmdCallback*)callback)
       (argc, argv);
}
```

Figure 14: TclInterpreter::CreateCommand

# 4 Future Work

As the ABC system development continues we hope to refine and expand the Tcl/C++ binding method discussed in this paper. The current implementation fails to expose the entire C++ interface in Tcl; in the short term this will be the focus of work on binding. Some thought has gone into automating target class modifications to make this method even easier to use. These improvements are discussed individually in the following list.

**New/Delete** C++ allows dynamic instantiation and destruction of objects. It would often be useful to expose this capability to the Tcl level. Current thinking is to require two static functions in each class, **TclNew** and **TclDelete** which would be installed in the Tcl interpreter as "*ClassName*::**new**" and "*ClassName*::**delete**" respectively.

*ClassName*::**new** would parse its parameters and, as described in the section below on overloading, select the appropriate constructor for the object, returning the handle for the new object to Tcl. *ClassName*::**delete** would take a handle, validate that it was of the appropriate class, and delete the associated object.

**Overloading** C++ allows overloading of a function by providing multiple parameter lists, differentiating between the various function bodies at compile time by the signature of the function call. A simple solution to this problem is to append some sort of signature indicator to the function name registered with the Tcl interpreter (to, in effect, mangle the function name into the Tcl name space). This is not satisfying as it defeats the ability to use a single name with multiple implementations chosen by the types of the calling parameters.

A more sophisticated solution is to provide a single callback with the name shared by the overloaded functions. Using the mapping functions, it would be possible to determine from the parameters provided which implementation to call. It might not be possible to determine the signature unambiguously. Current thinking is that the overloaded functions would be tried in increasing order of generality so the most specific version matching the parameter list would be called; it is not clear that such an order can always be found.

**Operators** C++ allows the redefinition of the meaning of various operators. It would be nice to expose these operators in Tcl and even nicer to bring along the convenience of overloading the operator syntax so expressions in Tcl and C++ would look the same. Current thinking is that exposing the operators is not that difficult; simply modify their signatures as with any other function and give them names in the Tcl interpreter. It may be possible to install a Tcl procedure with the same name as a handle and allow that procedure to handle the parsing of those operator expressions the class understands.

**Code Generation** Target class modifications are, on the whole, mechanical. It would be nice to provide automatic support for translating from a basic C++ class to one which could provide a Tcl interface. With a fairly complete library of mapping functions for built-in and locally-developed variable types, it would be possible to expose almost all of our C++ objects' interfaces.

Among the difficulties in providing such automation are the need to build dispatch functions for overloaded function names and constructors and intelligently mapping parameters. Mapping simple types by putting them in a **TclMap** template can be supported, perhaps even using optional parameter types when a default value is provided. There is currently no vision of a general solution for more complex mappings.

## 5 Conclusion

This paper describes a method of binding C++ member functions to Tcl procedures which is type-safe, effective, and easy to use. This simplifies the building of debugging scaffolds and the use of front-end processes to monitor and control other processes on the object level. This binding method makes it easier for the ABC system to use Tcl as its primary method of inter-process communication. Future work will increase the functionality and usability of this method.

## References


[1] John Menges and Mark Parris. Tcl and Tk Use in the Artifact Based Collaboration System. In *Tcl/Tk Workshop Proceedings*, pages 66–70, Berkeley, California, June 1993.

[2] B. C. Smith, L. A. Rowe, and S. C Yen. Tcl Distributed Programming. In *Tcl/Tk Workshop Proceedings*, pages 50–51, Berkeley, California, June 1993.

[3] J. B. Smith and F. D. Smith. ABC: A Hypermedia System for Artifact-Based Collaboration. In *Proceedings of Hypertext '91*, December 1991.