# Automatic Generation of Tcl Bindings for C and C++ Libraries

Wolfgang Heidrich
University of Waterloo, Canada

Philipp Slusallek
Universitaet Erlangen-Nuernberg, Germany

The major problem which arises when trying to attach existing C or C++ library functions to Tcl, is that they do not normally receive their arguments using this argc/argv mechanism. The developer has to write a C wrapper function, which parses the parameter list, converts the string of each argument to the correct C type, and passes these arguments to the C function. Return values and other output arguments must then be converted back into strings in order to be stored in a Tcl variable.

However, all wrapper functions are very similar to each other. Their main functionality, that is argument parsing and translation, can be created automatically if sufficient information about argument types is provided. The authors of *Wafe* use specification files with a special syntax for the description of C functions, widgets and special widget properties. These specification files are then parsed by a Perl script which creates the appropriate C code.

The problem becomes even more complicated if not only functions, but also C++ objects are to be accessed. Not only must a wrapper function be created for each public member function, but since C(++) data can not be addressed directly from Tcl, string handles need to be assigned, where each handle represents one C++ object on "the Tcl side" of the application. Tables that translate handles to and from C++ objects can be implemented using the hash tables provided by Tcl.

Moreover, our main goal was to transparently encapsulate C++ functionality in [incr Tcl] classes and objects. This means that [incr Tcl] classes have to be built, with every member function calling the corresponding C++ function wrapper (see Figure 2 and Section 3.2). All necessary code should be created without human intervention, if at all possible.

## 2 Structure of Itcl++

To meet these requirements, we decided to use a two step strategy. In the first step, C or C++ header files are parsed and specification files are created from function declarations and C++ class definitions. Additional information from a type declaration file, which contains information about complex C data types like structures or enumerations, is used.

The functionality of the specification files is a superset of those used in the *Wafe* project. In contrast to *Wafe*, which uses a proprietary file format, our specification files are just Tcl scripts which are executed with a predefined set of functions. This approach ensures that our code generators have enough flexibility to handle even very complex situations, as arbitrary Tcl commands may be executed within the specification file.

The generation of specification files is partially based on heuristics, since the semantics of a parameter can not always be determined by just evaluating its declaration. Consider, for example, the following function.

```
void foo( Foo *f );
```

No decision is possible about whether `f` is supposed to be a pointer to an object of type `Foo`, or an array of `Foo` objects: the two alternatives obviously require different conversion code. In cases where ambiguities occur, heuristics must be used, and a warning message is generated. Decisions made in this step may be overridden by simply editing the specification file.

This specification file, perhaps with some modifications made by hand, now contains all the information required to create C++ and [incr Tcl] code in the second step. Every semantic ambiguity in the specification file should now have been resolved, so that code generation can take place without further intervention.

Both [incr Tcl] and C (or C++) code is generated, where the [incr Tcl] part is only necessary if arrays or C++ objects are used. As mentioned above, both code generators are Tcl scripts which define a set of functions and then call the specification files. Another Tcl script is available to generate manual pages from the specification.

A diagram showing the interaction of all the parts is shown in Figure 1.

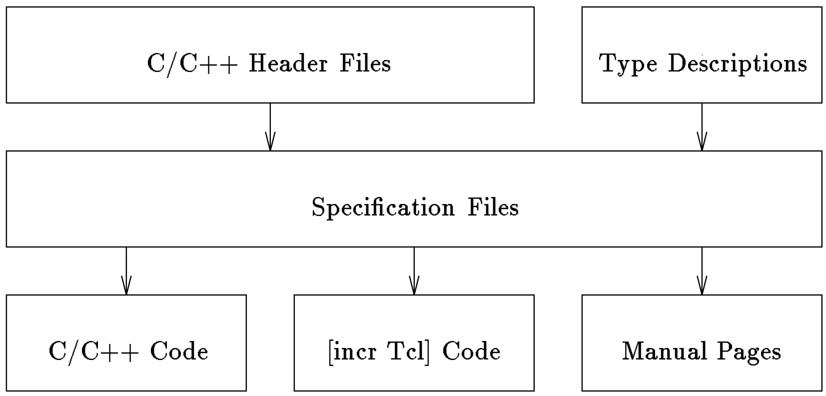


Figure 1: The building blocks of Itcl++: C and C++ header files are parsed, and a specification file is created using type information from a separate file. Other tools are used to generate C/C++ and [incr Tcl] code from these specification files.

# 3 Generating Code from Specification Files

The specification files contain one command for every C function which is to be mapped to Tcl. The following example specifies that the C function "`foo`", which takes an integer, and produces an integer as a return value, should be made available in Tcl as a command which also has name "`foo`".

```
command int {} foo {
  in int {cname value}
  cmdCode {returnVar= foo( value );}
}
```

The line starting with keyword `in` declares the input parameter for the function, with the second entry being the C type, and the third entry being a list of option/value pairs. The option `cname` in the above example is used to assign a name to the variable used in the C code of the wrapper function. If no name is specified, the names "`localVar`", "`localVar1`" and so on are used. The same mechanism applies to return values. In our example the option list for the return type is empty, and the default variable name "`returnVar`" is used.

The line starting with `cmdCode` contains the C code which is to be executed after parameters have been parsed, converted and stored in C variables. In our example the C function "`foo`" is called with its parameter. Then the result, an integer, is stored in the C variable `returnVar`, which is the default name for the variable holding the return value. The code for converting the incoming Tcl parameter, a string, to the correct C type, and for converting the return value back to Tcl, is generated automatically.

Another way of returning values which is often used in C is to pass a pointer to a variable as a function parameter. These semantics can be specified as follows:

```
command void {} bar {
  out int {cname outValue}
  cmdCode {bar( &outValue );}
}
```

True call by reference semantics as available in C++ can be specified in a similar way.

To allow the conversion of very complex data types, which might require temporary memory, clean up code may also be specified. The clean up code usually frees dynamically allocated memory and is executed as the last command in the wrapper function, after all output and return values have been written back to Tcl.

In the following we will show how the different C and C++ data types can be mapped to Tcl. In Section 4 we will describe how the specification files can be generated automatically.

### 3.1 Basic C Types

For the conversion of C types between Tcl and C, only C code needs to be generated. No Tcl or [incr Tcl] code is necessary, and no C++ features are used, except for array types, which we will discuss below.

A separate type specification file, which again is a Tcl script, contains the necessary information to convert values between C and Tcl. Simple C types, like integers, floats, characters and strings can be converted using `scanf` for input , and `printf` for output arguments. This information is stored in a Tcl array variable called "`conversion`":

```
set conversion(scanf,int)   "%d"
set conversion(printf,int)  "%d"
```

Using this information, the following C code will be generated for every integer input variable. Recall that the default name for the C variable used to hold the value is "`localVar`".

```
/* argv[i] holds the i-th parameter */
if( sscanf( argv[i], "%d",
            &localVar )
    != 1 )
  /* error handling */
  ...
```

A similar technique is used to create the output code.

More complex C data types like enumerations do not have a "natural" representation in Tcl. However, string constants can be used to represent the C constants in Tcl. Consider the following C type declaration.

```
typedef enum {
  Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday, Sunday
} Days;
```

The following lines in the type file specifies that type `Days` is an enumeration, with Tcl string "Mon" corresponding to the C constant `Monday`, "Tue" corresponding to `Tuesday`, and so on.

```
set conversion(enum,Days,Mon) Monday
set conversion(enum,Days,Tue) Tuesday
...
```

The code generated from this specification is a cascaded `if` statement.

```
if( !strcmp( argv[i], "Mon" ) )
  localVar= Monday;
else if( !strcmp( argv[i], "Tue" ) )
  localVar= Tuesday;
...
```

The same principle can be applied when a set of C preprocessor macros can be passed as a parameter. Many commercial libraries use macros instead of enumerations, because the latter are not supported by older compilers. In this case the macro names can be mapped to Tcl strings by introducing a pseudo enumeration type in the type file.

There are two different possibilities on how to handle C structures. One approach is to create a system of *handles*, and only pass these handles to Tcl, instead of the real data. Read or write access to single components of the structure would then be implemented by a call to a C function. We will describe such a system of handles when we discuss the conversion of C++ objects in Section 3.2.

For C structures, however, this approach to access the components from Tcl is overly complicated. We therefore chose to convert structures to associative arrays in Tcl. This means that all components of a structure are passed to Tcl and stored in an array, where the component names are used as indices.

To see how structure types can be specified in the type file, consider the following C type definition for a structure holding information about an employee.

```
typedef struct {
  char  *name;
  int   sin;
  float salary;
} Employee;
```

The entry in the type file which specifies this C structure is a comma separated list of components and their types:

```
set conversion(struct,Employee) \
  "(char *) name,int sin,float salary"
```

After the code for this type has been generated, its components can be accessed in Tcl using normal array syntax. For example the social insurance number of an employee can be accessed like this:

```
employee(sin)
```

The code generated for the conversion of structures recursively applies type conversion to each of the components of the structure.

Like structures, arrays can be handled in two different ways. Again we have to choose between a system based on handles, and one based on the direct mapping of the array contents to a Tcl data structure such as a list. The latter approach, however, bears the problem of deciding at runtime how many elements a passed array contains.

Another problem arises with this method when it is used in a C++ context. Suppose an array is returned from a function call. All its elements will be extracted, and put into a Tcl list. Later, when we want to pass this array to another C++ function, all elements will be transferred back, and a new array will be constructed. While this copy semantics is not usually a problem for arrays of simple types, it might be extremely harmful in the case of arrays of C++ objects, since for each element in the array a constructor will be called.

Therefore we decided to use the same system of handles as for C++ objects to support arrays. In Tcl, these handles are encapsulated in [incr Tcl] objects, which have methods for reading and writing single entries, as well as assigning lists of values to arrays.

## 3.2 C++ Classes

The syntax for the specification files presented above can easily be extended to C++ classes. The following is a specification of a simple counter class containing methods for incrementing and decrementing the counter by a given value, and a method for querying the current value. Furthermore it has one constructor, and one destructor. Since destructors in C++ do not take arguments, the existence of a public destructor needs only be specified with a binary flag.

```
class Counter {} {
  constructor Constructor {
    cmdCode{returnVar= new Counter();}
  }

  destructor

  member int {} getValue {
    cmdCode \
      {returnVar= self->getValue();}
  }

  member void {} += {
    in int {cname value}
    cmdCode \
      {self->operator+=( value );}
  }

  member void {} -= {
    in int {}
    cmdCode \
      {self->operator-=( localVar );}
  }
}
```

Since C++, unlike [incr Tcl], allows for more than one constructor in every class, every C++ constructor is mapped to a [incr Tcl] procedure. These procedures will call the [incr Tcl] constructor to create a new object, and will then invoke the corresponding C++ constructor. A more detailed description of the generated [incr Tcl] code can be found in [Heidrich, 1994].

Instead of having different wrappers for each public function of a class, we decided to group these functions into four categories (constructor, destructor, static and non-static member), for each of which we create one wrapper in order to prevent replication of code. The function call scheme is illustrated in Figure 2.

In order to keep track of the C++ objects referenced by [incr Tcl], we use an object server, which consists of two hash tables. One hash table maps C++ pointers to [incr Tcl] object names and is used for handling return values. It is important that this table contains pointers to all C++ subobjects of every registered C++ object. That is, for each registered C++ object the table contains one entry for

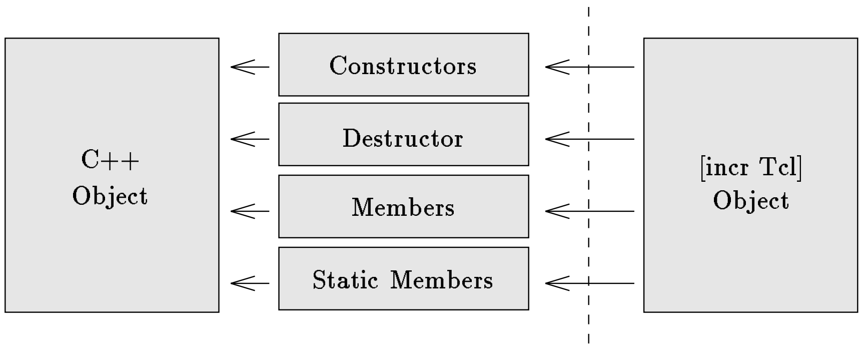


Figure 2: Access of C++ member functions through [incr Tcl]. [incr Tcl] members call C++ wrappers for constructors, destructors, methods and static methods, respectively. These functions convert function input parameters from Tcl to C++, execute the C++ object member and finally convert output parameters and the return value back to Tcl.

each class in the inheritance hierarchy of the object.

The second table maps [incr Tcl] objects to C++ object pointers. Again we need different pointers for every class in the inheritance graph of the object, so we can not just take the [incr Tcl] object name as a hash key, since there is no one-to-one correspondence between names and pointers. Instead, we have to assign a unique handle to each [incr Tcl] subobject. This handle is stored in a private `self` variable within every subobject.

During construction of an [incr Tcl] object, the constructor wrapper recursively calls the wrappers of the superclasses, with each wrapper registering the corresponding subobject and initializing the `self` variable (see Figure 3).

The object server is used to convert the arguments when C++ functions are called from [incr Tcl]. If a member function takes an object as one of its parameters, the wrapper function simply queries the object handler for the address of this object. Since every [incr Tcl] object is assigned a C++ object at creation time, this pointer always exists.

If, however, a C++ function returns a pointer to an object, this object may or may not be registered, that is, there may or may not be a corresponding [incr Tcl] object. If such an object exists, its name should be returned to [incr Tcl], otherwise a new [incr Tcl] object must be created and registered with the returned C++ object in the object server. In this case, if the object handler is not able to find a matching entry in its hash table, it creates a new [incr Tcl] object, which in turn registers itself with the object handler (see Figure 4).

One problem that arises with complex class hierarchies is that [incr Tcl] does not support repeated inheritance, i.e. a class may only occur once in the inheritance graph of another class. Thus the inheritance graph for every class may only be a tree instead of an arbitrary DAG as in C++, for example, when using virtual base classes. This means that C++ class hierarchies that use this feature cannot be completely mapped to [incr Tcl], but rather the [incr Tcl] hierarchy has to be cut below the point where this problem would occur.

While this is clearly a restriction, in practice the consequences do not seem to be too striking, since in C++ this feature is most often used to provide groups of classes with low level functionality, for example being writable to some sort of stream. In cases where [incr Tcl] is used as an high level interface on top of a C++ hierarchy, which seems to be the most appropriate range of application, one could as well do without such low level functionality. Nonetheless, we think that [incr Tcl] should be changed to support the full C++ semantics of inheritance in the future.

# 4 Generation of the Specification Files

The specification files can be generated by a Tcl script which parses a list of ANSI C or C++ header

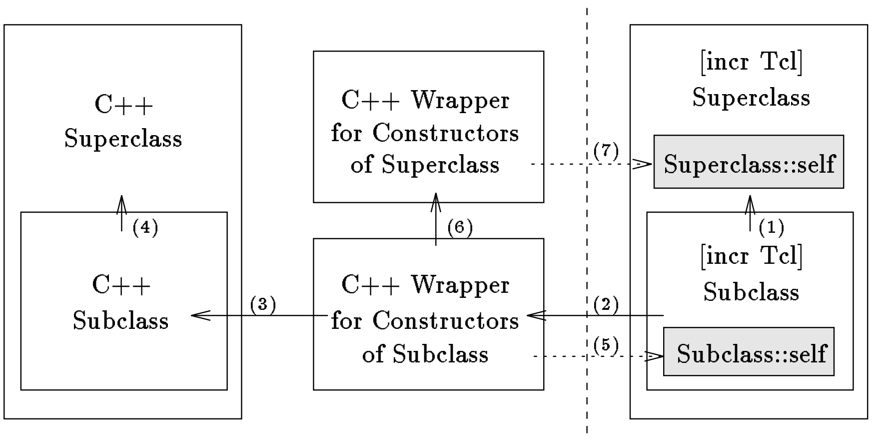


Figure 3: Whenever a new object is created, [incr Tcl] first executes the constructor of each [incr Tcl] base class (1). Afterwards, the C++ wrapper for the constructor is called (2), and a new C++ object is being created (3,4). Then the wrappers for the constructors of each baseclass are called recursively (6). They register the new object with the object handler, and initialize the `self` variables of the [incr Tcl] object (5,7).

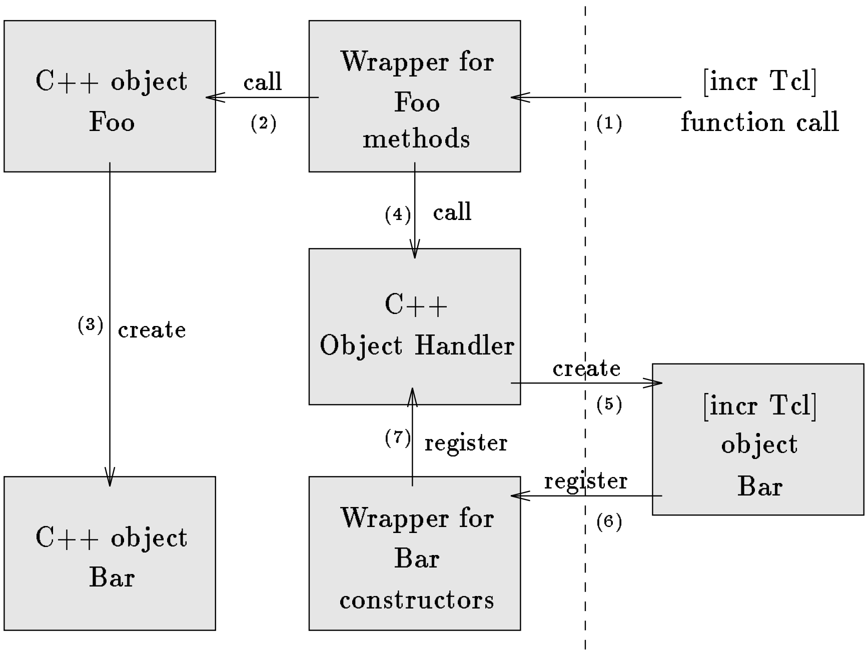


Figure 4: A member function of object `Foo` has been called (1,2), which returns a newly created object `Bar` (3). The `Foo` wrapper queries the object handler for the name of the corresponding [incr Tcl] object (4). Since the object handler is not able to find the name in the hash table, it creates a new [incr Tcl] object (5), which in turn registers itself through the constructor wrapper (6,7).

files. It uses the information from the type description files and from built-in rule tables to figure out the semantics of a given parameter.

As mentioned above, some of these rules need to be heuristic. Most of these heuristics deal with the problem of how to interpret pointer parameters: as arrays or as output values. The rules will determine, that, for example an argument of type "`char *`" is usually a string, while an argument of type "`int *`" is probably an output value. Whenever a heuristic rule is used, a warning message will be generated in the output file, so that it is easy to verify the correctness of the decision.

The parser also instantiates C++ templates. This is done by looking for simple type definitions which involve template types, but it is also possible for the programmer to directly specify which instances should be generated for a given template.

In the future, the type information could also be extracted automatically from the definition of enumerations and structures. It is clear, however, that the mapping of C macros to pseudo types mentioned in 3.1 would still have to be specified manually.

# 5 Results

## 5.1 Use of Itcl++ with our own Class Hierarchy

Itcl++ was originally developed for the use in an object oriented rendering system called VISION, which is currently under development at the computer graphics laboratory of the University of Erlangen [Slusallek, 1995].

The heuristics used for C++ parsing have been developed using the classes of the VISION hierarchy as a reference. However the C++ classes have not been changed to accommodate Itcl++.

The VISION system currently consists of about 250 classes, making extensive use of advanced C++ features such as templates. About 100 of the high level classes with a total of over 650 member functions have been mapped to [incr Tcl].

Heuristics have proven to work very well for this project: After inserting some type declarations in the types file, correct decisions have been made for *all* parameters of the 650 functions.

As a result, we are able to start Itcl++ from a "makefile", so that code is now generated completely automatically, without the need for human intervention.

We now use the [incr Tcl] interface to do initialization and configuration of our application, to describe scenes for our rendering system, and to test and debug new classes.

## 5.2 Use of Itcl++ with the OpenInventor Class Library

We tested Itcl++ with the commercial OpenInventor class library [Strauss, 1992] from Silicon Graphics. OpenInventor is an object oriented 3-D toolkit, which provides means to display and interactively manipulate complex scenes, using the 3D graphics library OpenGL.

For our testing purposes we chose 32 classes with 190 member functions, mainly geometric objects and manipulators. The C++ parser detected 13 ambiguities, all relating to parameters of type `char *`. Based on the heuristic rules, these parameters were interpreted as strings, not as pointers to `char`. In all cases this interpretation turned out to be the right one, so that no further human intervention has been necessary.

A specification file of 839 lines was used to create 8204 lines (about 18 KB) of C++ code. This averages to about 43 lines of code per member function.

# 6 Conclusion and Future Extensions

We have presented Itcl++, a tool for automatically generating Tcl/[incr Tcl] interfaces for C and C++ libraries. We have shown that it is possible to map C types to Tcl, and whole C++ class hierarchies to equivalent hierarchies in [incr Tcl]. The approach has been demonstrated using examples from a rendering class library and a commercial graphics library.

Directions for future development include the improvement of the heuristics used in the parsing step, the automatic generation of the type specifications from C and C++ header files, and providing direct read and write access to C variables.

Itcl++ is freely available to the research community. Please contact the authors for details.

# 7 Acknowledgments

We would like to thank Gustav Neumann and Stefan Nusser, who implemented a specification syntax and a related code generator for the *Wafe* program [Neumann, 1993]. We used their Perl implementation as a starting point for Itcl++. Gustav Neumann also made some suggestions concerning the syntax of our specification files. We would also like to thank the members of the VISION project, for which the tool was originally written since they tested early versions of Itcl++ and provided valuable feedback. Finally we would like to thank Fabrice Jaubert for reviewing an early version of the paper and making useful suggestions to improve its quality.

## References

[Beier, 1994] Beier, E. (1994). Tcl meets 3D – interpretative access to object-oriented graphics. In *Proc: 2nd Tcl/Tk Workshop, New Orleans, 1994*, pages 159–170.

[Heidrich, 1994] Heidrich, W., Slusallek, P., and Seidel, H.-P. (1994). Using C++ class libraries from an interpreted language. In *Proceedings of TOOLS USA '94*.

[McLennan, 1993] McLennan, M. J. (1993). [incr Tcl]: Object – Oriented Programming in Tcl. In *Proc: 1st Tcl/Tk Workshop, University of California at Berkeley, 1993.*

[Neumann, 1993] Neumann, G. and Nusser, S. (1993). Wafe — an X toolkit based frontend for application programs in various programming languages. In *Proc: Usenix Winter Conference, 1993.*

[Ousterhout, 1990] Ousterhout, J. K. (1990). Tcl: an embedded command language. In *Proc: Usenix Winter Conference, 1990.*

[Ousterhout, 1994] Ousterhout, J. K. (1994). An introduction to Tcl and Tk. Addison Wesley.

[Slusallek, 1995] Slusallek, P. and Seidel, H.-P. (1995). Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1).

[Strauss, 1992] Strauss, P. S. and Carey, R. (1992). An Object–Oriented 3D graphics toolkit. In *ACM Computer Graphics*. SIGGRAPH '92 Conference Proceedings.

[Welch, 1994] Welch, B. (1994). Practical programming in Tcl and Tk. To be published. Prentice Hall.