Using C++ objects with Tcl

Frank Pilhofer
`fp@informatik.uni-frankfurt.de`

July 3, 1997

Abstract

The "tclobj" package allows you to access C++ classes and to operate C++ objects from the Tcl language. They can be created and deleted, class member functions can be called, and objects can be passed as parameters to other objects' member functions. Using dynamic loading, class definitions can be loaded into the interpreter at runtime.

Some code must be added to the class definition, but existing code does not need to be changed. Each class that you want to be visible from the Tcl level must be derived from a common base class, and two member functions must be overloaded to interface between the two worlds. With some carefully placed preprocessor directives, the new Tcl-dependent part of a class can be easily activated or disabled at compile time (for use outside of Tcl-based applications).

The implementation is pretty efficient. As long as an object is only used in the C++ domain and is not handed down to the Tcl level, the overhead is nil but for two variable initializations. Time is only consumed during the setup of a call from a Tcl procedure into a C++ object and its return due to argument checking and conversion.

This package does not claim to be perfect, as there are some deficiencies and open problems, but can certainly be called interesting.

Contents

1 Introduction **3**

2 The C++ Domain **4**
2.1 The Class Declaration 4
2.2 The Class Implementation 5
2.2.1 Command Table 6
2.2.2 Interface functions 8
2.2.3 Initializer Function 12

3 The Tcl Domain **13**
3.1 Basic Usage 13
3.2 Type Information 14
3.3 Scope resolution 14

4 Dynamic Loading **15**
4.1 Handling dynamic modules 15
4.2 Static linking 16

5 Examples **16**
5.1 HelloWorld 17
5.2 fraction 17
5.3 storage 18
5.4 tclftp 19

1 Introduction

A major feature of the Tcl language is its extensibility using C functions, further enhanced by the introduction of dynamically loadable modules. Two philosophies exist for mixing Tcl and C. The first is to have a C application which just uses an integrated Tcl interpreter to execute rather small scripts that affect the applications runtime behavior. The other is to use Tcl as the glue to compose and control individual and rather independent modules into a larger application. While the focus of the C++ language is certainly on the second principle, this package makes sense in both cases.

This document is not an introduction into C++ or Tcl, but assumes that you are quite fluent in both languages. It helps if you have already played around with C-language extensions to Tcl and dynamic loading.

The code is still in experimental stages, and changes take place on a regular basis, usually requiring more or less changes in class implementations as well. Some portability problems exist; each new system holds its own surprises, mainly due to linking problems. If you can't handle a few problems yourself, this package is probably not (yet) for you. See the later chapters and the distribution's `README` file for some hints if things go wrong.

The documentation is in a similarly rough stage. So let's make this quick and have a look at the package's features and problems (apart from what's been mentioned in the abstract) in loose order.

+ No code changes. While code must be *added*, you do not need to browse through thousands of lines of existing code to look for incompatibilities. And you only need to touch classes that you actually want to be manipulated from the Tcl level.

+ Overloading, virtual functions and abstract base classes. These three major principles of object-oriented programming are fully supported from the Tcl level. Base class's operations are visible in derived classes unless they are overloaded. The new operator refuses to instantiate objects of an abstract class, yet an abstract class can still provide some common functionality.

+ Polymorphism. You can pass an object as a parameter to a function that actually expects a base class.

– Code additions. For each class, you must provide a special constructor and a member function mainly to map the string arguments of a Tcl function call to their proper types.

+ The writing of these two functions is made as easy as possible by macros, many helper functions and a type checker that checks a function call's arguments against a custom table, so that you do not need to do error checking on this part but can simply assume that all arguments have the correct type.

- No multiple inheritance. You can derive a class from two or more base classes, but only one direction of inheritance can be made visible on the Tcl level.
- No conversions in argument passing. While C++ compilers can detect that they can cast an object to the argument's expected type, this is not (yet) supported on the Tcl level (excluding the special case of polymorphism).
- Arguments are always passed by-reference, and objects can only be returned by-reference.
- No direct access to data members or static functions.

See the discussion of the example classes for more information.

2 The C++ Domain

This section describes how to adapt your classes to work with the `tclobj` package and the available extensions that you might find useful.

2.1 The Class Declaration

All classes that you want to be visible from the Tcl level must be derived from the common base class `Tcl_Object`, which is defined in the include file `tclobj.h`. This derivation may be direct or indirect (if `Tcl_Object` is inherited through one of the base classes).

If you want your classes to still compile without Tcl, you can use the following preprocessor directives:

```
#ifdef TCL_OBJECT
class HelloWorld : public Tcl_Object
#else
class HelloWorld
#endif
```

None of the member declarators need to be modified, but three lines must be added. To continue with our declaration of the *HelloWorld* class:

```
#ifdef TCL_OBJECT
public:
  HelloWorld   (Tcl_Interp *, int, char *[], int);
  int Tcl_Proc (Tcl_Interp *, int, char *[], int);
  TCL_OBJECT_DECL_CLASS(HelloWorld);
#endif
```

The first line declares a special constructor, which will be called when the *new* operator is invoked by a Tcl program. The second line declares a special member function which will be called whenever an operation on the object is invoked by a Tcl program. These two functions act as the interface between the two worlds and must both be public members of the class.

The `TCL_OBJECT_DECL_CLASS` macro adds some common declarations. (The reason that `Tcl_Proc` is not considered common is that you will have to write that function yourself.) The macro takes as parameter the (unquoted) name of the class.

A class can be derived from more than one class, but can only inherit Tcl-level procedures from one of them. For more information, see the later chapter on "Multiple Inheritance".[1]

Note that both functions, the constructor and `Tcl_Proc`, must not be inline. The macro declares both public and private members. Do not assume that it ends with a particular kind of access specifier (add your own `public:`, `private:` or `protected:` for following declarations).

You do not need to declare a constructor for abstract base classes. In theory, you could also choose not to declare a Tcl-type constructor for non-abstract classes. Such classes could then never be instantiated on the Tcl level. Abstract base classes *do* however need the `Tcl_Proc` function, because they can offer some functionality to derived classes, for example generic functions using pure virtual members.

2.2 The Class Implementation

To the class's implementation we must add the constructor and the `Tcl_Proc` function, and a call to yet another macro. There are two incarnations of the macro, one for abstract classes that do not provide a Tcl-style constructor, and one for non-abstract classes. Either the one or the other must be called at global level.

```
TCL_OBJECT_DECL_BODY(class, derived-from,
                     command-table, init-function);
```

class The class's type (unquoted). For our example, this would be `HelloWorld`.

derived-from The base class's type (also unquoted).

command-table A table detailing the operations that can be performed on objects of this class from the Tcl level, and their parameter lists. See below.

init-function If non-`NULL`, this function is called once when the class is initialized (but after the initialization of the module's global objects) and can perform custom initialization. The function receives as its sole argument a `Tcl_Interp*` pointer to the current interpreter, must be of "C" linkage, and return the integer `TCL_OK` upon success, else it should set an appropriate error message and return `TCL_ERROR`.

[1] This chapter is not written yet.

Abstract base classes, or classes that you want to appear abstract from the Tcl level, must use the `TCL_OBJECT_DECLARE_ABSTRACT_BODY` macro instead. It has the same arguments as above, but sets up some internal information differently. You can find a complete example in figure 1.

2.2.1 Command Table

The *command-table* is a null-terminated array with elements of type `const TclObj_Comtab`. It defines the signatures of the constructors and member functions available on the Tcl level. The information is used by the library functions upon invokation of a member function to check whether the parameters match an available signature. Each item of the array is composed of four strings:

number A positive serial number of this entry. It should be unique among the constructors or the member functions within a table – you'll see later why. (There may be cases when re-using a number could make sense; so it's *not* explicitely forbidden.)

name The name of the member function, or "new" if this entry defines a constructor. More than one entry may exist with the same name if their argument lists differ (overloading).

args The argument list for this function (see below).

desc An optional short description of this operation (can be `NULL`).

The argument list "args" can be the empty string if the constructor or member function does not take any parameters, or a list of types separated by whitespace and commas (like in a function prototype). The following basic types are supported

int An integer value.

uint Unsigned integer value.

long Long integer.

ulong Unsigned long integer.

double A double-precision floating-point value.

string A string.

bool A boolean value.

ptr A `void*` pointer.

type An object of class `type` or some class derived from `type`.

```
static const TclObj_Comtab testtable[] = {
  /* constructors */
  { 1, "new", "",         "creates empty string object"     },
  { 2, "new", "test",     "copy constructor"                },
  /* member functions */
  { 1, "set", "...",      "set value of object"             },
  { 2, "get", "",         "query value of object"           },
  { 3, "add", "...",      "append text to string"           },
  { 4, "copy","test",     "copy contents of other object"   },
  { 5, "dup", "",         "create new object with our data" },
  { 0, NULL, NULL, NULL }
};

TCL_OBJECT_DECL_BODY (test,Tcl_Object,testtable,NULL);
```

Figure 1: Command table for the "test" class

The last argument in the list can also be three dots "..." which is the same as in C, meaning that any number (including zero) of arguments of any type may follow.

Figure 1 shows the command table from one of the examples, the "test" object, which is some kind of a string class. First, there are two constructors, one without any arguments (the "default constructor"), and one constructor that takes a parameter of its own class as parameter (the "copy constructor"). You are not restricted by this choice of constructors; you need to define neither, and constructors are free to use any number and kinds of parameters.

The `set` and `add` member functions take any number of arguments. `get` just retrieves the object's value and does not need any parameters. `copy` copies the value of another object and, like the copy constructor, takes an argument of its own class.

You will notice that we define the parameters of member functions, but not their return types. This is because with Tcl, our return type is fixed to strings. However, helper functions exist that allow you to return numeric results or references to objects as well.

The purpose of these command tables is to relieve you from the effort of argument checking. Your Tcl-style constructor or the `Tcl_Proc` procedure will only be called if the arguments check out okay. At that time, a matching table entry will be determined; the number of this entry is also passed to your function, and you can then simply enter a `switch` statement with the table entry number.

You need to be careful when overloading member functions with different kinds of numeric arguments, for example when defining a function that can either take an `int` or `double` parameter. Because Tcl does not know about prototyping, we can only look at the argument string. For example, the string "1" can be `bool`, `double`, any of the integer types, and even a `string`.

The algorithm that tries to match an available signature dives down a tree, following the matching nodes, but doesn't do backtracking. Consider the following entries in a command table:

```
{ 1, "proc", "uint, int",    NULL},
{ 2, "proc", "long, double", NULL}
```

Calling this member with the parameters "1 0.5" would fail. Because the "1" is an unsigned integer, the algorithm narrows its search to all members that require an unsigned integer as first parameter. But in the next step, the algorithm notices that "0.5" is not an integer value and fails. Because the algorithm believes the first argument to be an unsigned integer, the second entry is not considered. If, however, the member was called with the parameters "-1 0.5", the second entry would be used, because "-1" is not an unsigned integer.

Such cases are usually easy to avoid. Member functions that take different kinds of numeric arguments tend to implement the same behaviour anyway, where it doesn't matter what kind of integer an argument is actually intended to be.

2.2.2 Interface functions

Both the constructor and the `Tcl_Proc` procedure take the same four arguments. The constructor by definition has no return value, `Tcl_Proc` must return an `int`. Hence, the declarations look like

```
HelloWorld::HelloWorld (Tcl_Interp *interp,
                        int argc, char *argv[],
                        int command)
```

The first argument is the Tcl interpreter. The second and third arguments are the parameter list. The first parameter is actually in `argv[2]`; the zeroth element is the name of the object, and the first element is the command name that is invoked. Fourth argument to the function is the number of the command that is to be executed, taken from the constructor or command table, respectively.

You will usually perform a `switch` on the `command` parameter, cast the arguments to the desired types, call the responsible member function, and return the return value.

Helper functions (or macros) help you to do the job of argument casting. Because the argument lists have already checked out okay, there is no need to check if these operations fail. The first argument to them all is the pointer to the current Tcl interpreter, the second one is the argument `argv[i]`, and the return value is of the type in question.

`TCL_OBJECT_ARG_INT(Tcl_Interp*, char*)`
: Casts the argument to an integer value.

`TCL_OBJECT_ARG_UINT(Tcl_Interp*, char*)`
Casts the argument to an unsigned integer.

`TCL_OBJECT_ARG_LONG(Tcl_Interp*, char*)`
Casts the argument to a long integer.

`TCL_OBJECT_ARG_ULONG(Tcl_Interp*, char*)`
Casts the argument to an unsigned long integer.

`TCL_OBJECT_ARG_DOUBLE(Tcl_Interp*, char*)`
Casts the argument to a floating-point value.

`TCL_OBJECT_ARG_BOOL(Tcl_Interp*, char*)`
Casts the argument to boolean (returned as a true or false integer).

`TCL_OBJECT_ARG_STRING(Tcl_Interp*, char*)`
Casts the argument to a string pointer. Actually, this macro simply returns its second parameter.

`TCL_OBJECT_ARG_PTR(Tcl_Interp*, char*)`
Casts the argument to a `void*` pointer and requires an argument that was returned by a previous call to `TCL_OBJECT_RETURN_PTR()` (see below).

`TCL_OBJECT_ARG_OBJECT(Tcl_Interp*, char*, type)`
Casts the argument to a pointer to a class of the given type, requiring an argument of that type, or any compatible type (a derived class) returned by a previous call to `TCL_OBJECT_RETURN_OBJECT()` (see below). Returns a `type*` pointer.

```
TclObj_CastArgs  (Tcl_Interp*, int argc, char *argv[],
                  char *arglist, ...)
```
Performs a number of casts, similar to `sscanf`. `arglist` is list of argument types like the `args` element of a command table. The argument list in `argv` is interpreted according to the types found there. For each parameter, a pointer to the result type must be given as additional parameter:

- A `char**` pointer for a string parameter.
- Pointers to various integers for the `int`, `uint`, `long` or `ulong` parameter types.
- An `int*` pointer for a boolean value.
- A `double*` pointer for a floating-point value.
- A `void**` pointer for a `ptr` parameter.
- A `Tcl_Object**` pointer for any custom class type.

The argument list passed here must be zero-based, meaning that the first parameter must be found in `argv[0]`. When processing arguments in `Tcl_Proc`, the actual parameter list starts at `argv[2]`, so you would call the function with `argc-2` and `argv+2`.

Similar functions exist to return values of different kinds. Like their above counterparts, they take as first parameter the current interpreter. The functions produce a string representation of the second parameter, and set this string as the interpreter's result. All of them return `TCL_OK`.

`TCL_OBJECT_RETURN_INT(Tcl_Interp*, int)`
: Returns an integer value.

`TCL_OBJECT_RETURN_UINT(Tcl_Interp*, unsigned int)`
: Returns an unsigned integer value.

`TCL_OBJECT_RETURN_LONG(Tcl_Interp*, long)`
: Returns a long integer value.

`TCL_OBJECT_RETURN_ULONG(Tcl_Interp*, unsigned long)`
: Returns an unsigned long value.

`TCL_OBJECT_RETURN_BOOL(Tcl_Interp*, int)`
: Returns a boolean value (true or false).

`TCL_OBJECT_RETURN_DOUBLE(Tcl_Interp*, double)`
: Returns a floating-point value.

`TCL_OBJECT_RETURN_STRING(Tcl_Interp*, char*)`
: Returns a copy of a string (so the pointer may point to volatile memory).

`TCL_OBJECT_RETURN_PTR(Tcl_Interp*, void*)`
: Returns an arbitrary pointer; useful for structures and classes that are not themselves derived from `Tcl_Object` but are needed here and there as parameter or return value.

`TCL_OBJECT_RETURN_OBJECT(Tcl_Interp*, Tcl_Object*)`
: Returns a reference to the given object (`Tcl_Object` is a base class for every Tcl-visible class).

`TclObj_MkResult(Tcl_Interp*, char *arglist, ...)`
: Composes a return value from one or more values according to the list of data types given in `arglist`, which is of the same kind as in a command table. For each argument given in the arglist, an appropriate value must be given in the parameter list:

  - A `char*` pointer for a string.
  - For the various integer types, boolean and floating-point, the value itself.
  - A `void*` pointer for a `ptr` type.
  - A `Tcl_Object*` pointer for any custom class type.

  The result will be a list of all the arguments; the Tcl program can use the list commands to extract the individual items.

```
int
HelloWorld::Tcl_Proc (Tcl_Interp *interp, int argc,
                      char *argv[], int command)
{
  HelloWorld *TheOther;

  switch (command) {
  case 1: /* hello */
    return TCL_OBJECT_RETURN_STRING(interp, hello());

  case 2: /* world */
    return TCL_OBJECT_RETURN_STRING(interp, world());

  case 3: /* copy HelloWorld */
    TheOther = TCL_OBJECT_ARG_OBJECT(interp, argv[2],
                                     HelloWorld);
    operator= (*TheOther);
    return TCL_OBJECT_RETURN_OBJECT(interp, this);
  }
}
```

Figure 2: A copy operator for the “HelloWorld” class

```
  case 3: /* copy HelloWorld */
    TclObj_CastArgs (interp, argc-2, argv+2,
                     "HelloWorld", &TheOther);
    operator= (*TheOther);
    return TclObj_MkResult (interp, "HelloWorld", this);
```

Figure 3: Alternative implemenation of the copy operator

```
extern "C" int
Fraction_Init (Tcl_Interp *interp)
{
  fraction * oneseventh = new fraction (1,7);
  oneseventh->Tcl_Enable (interp, "oneseventh");
  return TCL_OK;
}
```

Figure 4: Initializer for the "fraction" class

Using these helpers, the two necessary interface functions are easy to implement. Figure 2 shows a complete example, the `Tcl_Proc` function of the `HelloWorld` class. It provides the "hello" and "world" members, which just return the strings returned from the corresponding member function. The third member, a copy operator, is a little longer. First, it casts the argument to the desired type. Then, the (compiler-generated) assignment operator is invoked, and finally a reference to the object itself is returned.

To demonstrate the usage of the more generic functions, an alternative implementation of the copy operator is given in figure 3.

You are not restricted to use the mentioned helper functions. You are free to use other means of extracting the data out of the arguments, and you can also set the result yourself, using the functions provided by Tcl.

Although you are guaranteed that `Tcl_Proc` is never called with a `command` value different from the values of the corresponding command table, you should call the macro `TCL_OBJECT_NO_MATCH(Tcl_Interp*)` with the current Tcl interpreter as parameter at the end of the function. It serves as emergency exit and prevents a compiler warning.

2.2.3 Initializer Function

The fourth argument of the `TCL_OBJECT_DECL_BODY()` macro can be NULL, or the name of an initializer function to call when the class is loaded into the interpreter. Apart from setting up your data structures, you can also use this function to make static objects available to the Tcl level.

For example, the "fraction" class wants the constant of one seventh to be available by default. Figure 4 shows how this is done. The initializer function must be declared with "C" linkage and receives the current interpreter as parameter.

From the class `Tcl_Object`, the `Tcl_Enable` member is inherited, which makes the object visible on the Tcl level. Usually, this member is called automatically when an object is returned from `Tcl_Proc()`, but here you need to call the function explicitely. The second parameter is the name you want to assign to the object.

3 The Tcl Domain

3.1 Basic Usage

Basically, only two new Tcl commands are provided, "new" and "delete", which you already know from C++ programming. As Tcl commands, they do just the same.

The first parameter to `new` is the name of a class, and the following arguments are treated as arguments to the constructor. As result, you receive a unique identifier. This identifier is actually a Tcl command which you can then call the object's member functions on. Looking at the definition of the `test` class from figure 1, the following sequence would be valid:

```
set x [new test]
$x set hello world
set y [new test $x]
$y get
delete $x $y
```

First, an empty object of type test is created using the default constructor. The identifier is stored in the variable "x". We then invoke the member function "set" of class test, setting the value of the object to type "hello world". Next, we create another object of type test, but this time using the copy constructor, with the first object as an argument. Invoking the "get" member function should therefore produce "hello world". At last, both objects are destroyed.

The `delete` function is particularly easy to understand. It takes any number of object identifiers as parameter and destroys them, calling their destructors.

A slightly different mechanism to create and destroy objects exists, too. This one is more reminiscent of automatic variables within a function, which is precisely their purpose.

```
test x
x set hello world
test y x
y get
```

This code fragment does much the same as the previous. However, we do not use the `new` operator. The first line creates an object of type test called "x". This is quite different from having a unique identifier created and having this identifier assigned to a variable of name "x". This is apparent in the second line, where we do not need to take the value of "x" to extract the identifier, because "x" itself *is* the identifier.

Another difference is that the two objects "x" and "y" that have been created cannot be destroyed with the `delete` operator. Instead, they are automatically destroyed when scope is lost, thus behaving like local variables.

Still, objects that have been created this way are different from local Tcl variables. Global variables are visible in a procedure unless they are shadowed with a local variable and can not be (and need not be) accessed with Tcl's `global` command.

You must also take care that you do not create objects with the same name as existing Tcl procedures, which would then become inaccessible.

3.2 Type Information

By deriving your classes from the `Tcl_Object` base class, the "info" member function is inherited on the Tcl level. Through this member, a Tcl program can query some information about the object.

***object* info id**
: Prints the object's type and a unique identifier (actually the numeric value of the `this` pointer).

***object* info dep**
: Returns a whitespace-separated list of the object's type and its (linear) derivation tree (the types of the object's base classes). Last in this list is always `Tcl_Object`.

***object* info ops**
: Returns a whitespace-separated list of member functions available for this class, including its base classes (which are prefixed with their type name, for example `Tcl_Object::info`). This list also includes "new", which is in fact no member function but the constructor.

***object* info sig**
: Prints all available signatures of all member functions (including the constructors). This list can get quite lengthy.

object* info *func
: If *func* is a member function available for `object`, print all member functions of this name and their argument lists.

The disadvantage of the "info" member is that you need an existing object. As an alternative, you can also call the "info" operation on the class's type name (excluding "info id", because there is no object to have an identifier). Keep in mind that this is the same syntax as with the creation of "local variables".

3.3 Scope resolution

Like in C++, you can use the scope resolution operator on a member function in order to call a base class's implementation which would normally be shadowed by a derived class's members. For example, if a class overloads the "info" member, but you want to retrieve information about the object using the "info" implementation of `Tcl_Object`, you would use

object `Tcl_Object::info` id

4 Dynamic Loading

Object-oriented programming comes to its full power in a dynamic environment. If an existing class does not fit your purposes, you derive your own class, and attach the new module – *at runtime*! What C++ has been waiting for is now possible with the help of Tcl's generic mechanisms for dynamic loading.

4.1 Handling dynamic modules

"tclobj" itself can be compiled into a shared library and loaded with the "load" command. Classes compiled into shared libraries can also be loaded this way by using the class name prefixed with `c_` as package name. For example, you can load the "fraction" class from the examples with

```
load ./fraction.so c_fraction
```

However, because of the package naming policies of Tcl, this only works if the class name is all in lowercase. Another restriction imposed by Tcl is that it refuses to load more than one package from the same shared library. This can be quite annoying if you have several tiny interrelated classes and would be forced to put each of them into a separate shared library.

For this purpose, additional macros exist in order to "package" one or more class into a package, that upon loading, initializes all of the contained classes. The macros are best explained upon an example from our "storage" example, which defines one base class and three derived classes.

```
TCL_OBJECT_BEGIN_PACKAGE(storage);
TCL_OBJECT_PKG_CLASS(int_storage);
TCL_OBJECT_PKG_CLASS(double_storage);
TCL_OBJECT_PKG_CLASS(string_storage);
TCL_OBJECT_END_PACKAGE();
```

This sequence (which must be placed in a module – not a header file) defines the "storage" package, which is to initialize the classes "int_storage", "double_storage" and "string_storage". The package name must again be all in lowercase, but the class names can be mixed case. After building the module(s) into a shared library, it can be loaded using the package name prefixed with `pkg_`. If the shared library was named "storage.so", the load command would be

```
load ./storage.so pkg_storage
```

If you are familiar with loading Tcl packages from dynamic libraries, you know that each package must define a *Package*`_Init` function. This is exactly what the above macros do. In fact, you can write such a function yourself with a package name of your choice. By looking at the definition of the macros, you can easily mirror their behaviour.

Another macro, `TCL_OBJECT_PKG_EXPECT(`*name*`)`, is allowed within a package declaration. If two classes depend on each other to be available, loading both classes individually would fail because the other would not be available at the time the first is processed. A similar chicken-egg problem exists with tclobj, which checks that all argument types are available. This macro declares that a certain type will be provided at a later time but that it is acceptable within an argument list.

At last, the macro `TCL_OBJECT_PKG_PKG(`*name*`)` allows to include one package into another package declaration. For example if you have several packages in your project, you could write a "project file" that is nothing but one package declaration, including all the other packages in your project.

However, dynamic loading of C++ code turns out to be system-dependent, not being as easy as producing dynamic modules from plain C code. The `README` file that comes with the distribution discusses this problem and possible solutions or workarounds.

4.2 Static linking

If dynamic loading of classes is not supported on your system, you can fall back to linking the class code statically with your application. You will have to build the tclobj package and all of your classes into a custom `tclsh`.

During setup, you must then call the appropriate initialization functions to make the packages and classes available to the interpreter. To set up tclobj itself, call

```
int Tclobj_Init (Tcl_Interp *);
```

For classes, you can then call the initialization function by the name of `C_`*classname*`_Init`. For collections of classes using the above "packaging" mechanism, the initialization function is named `Pkg_`*pkgname*`_Init` (note the case, the first letter and the 'I' are capitalized). To load both the fraction and the storage class, you would do

```
if (Tclobj_Init     (interp) == TCL_ERROR) return TCL_ERROR;
if (C_fraction_Init (interp) == TCL_ERROR) return TCL_ERROR;
if (Pkg_Storage_Init(interp) == TCL_ERROR) return TCL_ERROR;
```

A slight pitfall is that you have to compile the "main" module (the one that contains the Tcl initialization) with a C compiler, but must perform linking with the C++ compiler (so that it adds the necessary C++ libraries to the linker command).

For more information on building your own custom `tclsh`, see the Tcl documentation.

5 Examples

In the `example/` subdirectory, you can find five classes that can be made available in Tcl. I suggest to look at their code while reading the descriptions, and

```
set x [new HelloWorld]
puts "[$x hello] [$x world]"
delete $x
```

Figure 5: Sample session with the "HelloWorld" class

```
set x [new fraction 1 3]
set y [new fraction 2 9]
set z [new fraction 3 27]
$x += $y
$x /= $z
$x value
$x numden
delete $x $y $z
```

Figure 6: Sample session with the "fraction" class

to try out the examples for yourself. All these examples have their purpose; each highlights a different feature and different problems of the package.

5.1 HelloWorld

This is a very tiny "Hello, World" class. It provides the two member functions `hello`, which returns the string "hello" and `world`, which returns "world". Both are made available to Tcl as operations without parameter. See figure 5 for a short sample session.

Because the class name is mixed case, we have put the class into a package of its own. Therefore, you must load the class using the package name `pkg_helloworld`.

5.2 fraction

An arithmetic class, providing a `fraction` data type with an integer numerator and denominator. The Tcl interface provides basic operations for addition, substraction, multiplication and division, plus a number of comparison operators. The "value" member function retrieves the object's value as a floating-point value, and "numden" produces a list with two members, the numerator and denominator.

Function overloading is demonstrated with the assignment operator, which can take three different kinds of arguments, either it can copy the value of another `fraction` object, a single integer value (setting the denominator to 1) or two integers. See figure 6 for a short sample session.

You will notice that while the assignment kind of operators (like "+=") are exported, the binary operators ("+") are not. Why? The binary arithmetic operators have to return an object with the result value. However, the Tcl interface can only return references to objects. Consider the following potential code fragment to implement an addition operator:

```
fraction *other = TCL_OBJECT_ARG_OBJECT(interp, argv[2],
                                        fraction);
fraction result = operator+ (*other);
return TCL_OBJECT_RETURN_OBJECT(interp, &result);
```

However, because the "result" object is local, the reference returned to the Tcl level will be invalid, because the object it points to has been destroyed upon return from the function.

A different solution would be to create the return object on dynamic storage:

```
fraction *other  = TCL_OBJECT_ARG_OBJECT(interp, argv[2],
                                         fraction);
fraction *result = new fraction(operator+ (*other))
return TCL_OBJECT_RETURN_OBJECT(interp, result);
```

This would work. However, the Tcl program must then be aware of this fact, and must remember to destroy the returned "temporary" object sooner or later. Otherwise, more and more garbage would pile up.

The third idea is to return not the object, but its contents, similar to the "numden" operator. The other functions could then be overloaded with versions that take the two integer values as parameter. Still, this solution is not perfect, but it prevents garbage. But to split up the result of an operation (a list of two integers) into the two integers required by the parameter, an `eval` call is needed.

```
$x = [eval $y / $z]
```

This line of code would then divide the fraction object in "y" by the object in "z", and then assign the result to "x". Still, the workaround of having the binary operators return the object's contents, and having the operators take these contents as an argument, is not exactly state-of-the-art, and could be quite a difficult task for more complicated objects. If the object contains non-trivial members, this method would in fact have the same problems as the solution of returning a dynamically allocated object: the Tcl programmer would have to make sure that the destructor is called on the object.

Because a solution cannot yet be offered, only references to object are supported at this time.

5.3 storage

This example demonstrates abstract base classes, virtual functions and inheritance. We define a generic class "storage", which implements a simple storage cell.

The only function common to all kinds of storage is that its value can be queried. Hence, this class defines a pure virtual function "query" which is

```
set x [new int_storage]
set y [new double_storage]
set z [new string_storage]
$x set 42
$y set 3.14159
$z set "hello world"
$x query
$y query
$z query
delete $x $y $z
```

Figure 7: Sample session with the "storage" classes

supposed to return the value as a string. On the Tcl level, however, we provide a "query" member function for the `storage` base class, which returns the value returned from the virtual function.

We then derive three kinds of storage cells, `int_storage`, `double_storage` and `string_storage`. Each class provides an implementation of the `query()` function and a custom `set` function that takes a type-specific argument.

On the Tcl level, however, no implementation of `query` is needed, because it is inherited from the base class. Note that only the base class is derived from `Tcl_Object`, the others inherit the necessary functionality through indirect inheritance by being derived from the base class.

See figure 7 for an example of using the storage classes from Tcl. While the `set` implementation is different for each of the objects, the `query` function is inherited from the base class, but can be invoked like any other member function.

Because the base class and the three derived classes are defined in the same module, a package is defined that initializes all of them. Load the module using a package name of `pkg_storage`.

5.4 tclftp

The other examples were, well, examples. They were custom-designed to fit the necessary requirements. The question is, how well does this approach work with existing classes that were designed without having Tcl in mind?

As a demonstration, we add the necessary hooks to the "ftp" class from the `socket++` library.[2] In fact, this class turned out to be a perfect example of what you might be confronted with: completely undocumented and slightly buggy.[3] Still, it was possible to integrate the class into Tcl without a single code change (apart from the bugfix). After learning how the class worked, the work was done in slightly more than one hour, not counting lunch.

[2] The `socket++` library can be downloaded by anonymous ftp from `ftp.virginia.edu` as `/pub/socket++-1.10.tar.gz`

[3] There's a patch to the `socket++` library at the bottom of `tclftp.c`

```
set x [new tclftp]
$x open ftp.uni-frankfurt.de
$x cd /pub/dist/frank
$x dir
$x get uudeview-0.5.9.tar.gz
$x close
delete $x
```

Figure 8: Sample session with the "tclftp" classes

Usually, some code must be added to a class to allow its integration into Tcl. We use a different approach here. In order to not change the existing code, we derive a new class from the original one, inheriting all its function. We then add the necessary modifications to the derived class.

The `tclftp` class exports all the functions that make sense to ftp operations. You can open and close connections, browse directories, retrieve and upload files. For some reason, re-opening an object does not work; I suspect this is because of a bug in the original class. See figure 8 for a sample session.

This method of integrating derived classes allows you to work with classes without touching the original sources; you do not even need the source code at all. Still, there is a catch. Usually, objects could be passed from and to the Tcl level without spending a thought of where they came into existence. But imagine that in the existing and unchanged code, `ftp` objects are created.

This is not a problem as long as the object is only used in the C++ domain. But because the `ftp` base class does not inherit the necessary Tcl hooks, which are only available through the second derivation of `tclftp`, such an object could not be passed to the Tcl level. This is a rare problem, and in some cases where this problem arises, it is possible to "repackage" the base object into the derived object.