A Perl-C++ Interface with applications to the Qt library

Roberto De Leo
deleo@crs4.it

Crs4
Uta, Cagliari, Italia

Abstract: *In this paper we present the parser we wrote to produce automatically Perl bindings for C++ libraries.*

1 Introduction

When developing the basics of a package, it would be desirable to be able to focus only on the algorithm part avoiding all the intricacies related only to the language structure, like variables declarations, memory management and compilation issues.

In particular, coding in C and C++ involves always complicated operations that have nothing to do with the actual algorithm being developed but just with the complexity of the language itself.

This problem can be effectively solved builiding Perl bindings for the library. Larry Wall's language [WCS96] indeed gets rid of all these problems (no type declaration is needed, the language is interpreted and needs no compiling and memory management is automatic) allowing the programmer to focus solely on building the algorithm.

Of course all this has a drawback, namely Perl code runs always slower than the corresponding C/C++ code, but in the developing phase this problem tipically has no relevance at all and, once the development is over, the close similarity between Perl and C/C++ syntax makes a pretty standard and fast job to convert all code from one language to the other. Moreover, the most complex operations are done by the C++ functions that are just called from Perl and therefore the difference of speed may be not so big after all.

This approach has also another great advantage: it allows to offer classes on highly complex C/C++ libraries such as OpenCascade to a much broader set of students, because in this way the students do not need to invest a not neglectable amount of time in learning C/C++ and in fighting with the compiler and with linking libraries: Perl modules, once installed, work without needing any setup from the user that can transparently use all classes simply calling the right Perl module.

A thorough article about pros and cons of using scripting languages as opposed to using compiled ones can be find in a John K. Ousterhout (father of Tcl/Tk) article [Ous98a,Ous98b]. In particular from that article it appears evident as scripting languages require in general a much smaller number of lies of code and tend to increase productivity and software reuse.

For the same reasons scripting interfaces have already been generated at Crs4: for example an object oriented interface to the C library Shapes had been implemented by G. Zanetti, giving us a good starting point for our progresses.

We are in particular interested in a Perl interface to the Qt C++ library [Dal02], mainly because it is the most used widgets library that runs on the three main OSs: MS Windows, Unix and MacOS. The Qt library should show particularly useful in our case to provide a simple and reusable interface to our future OpenCascade applications.

A PerlQt interface had been developed years ago by Ashley Winters (his web page, containing the PerlQt package, was once http://www.accessone.com/~jql/ but it seems it is not anymore existing) under perl5.005 for versions 1.04 and 2.0 of Qt but unfortunately is not anymore mantained and in the meantime both Qt and Perl have undergone radical changes that do not allow anymore to continue using those old versions. On the other side luckily the interface for version 1.04 uses exactly the same kind of interface we chose to use and it helped a lot our work.

2 Building Perl-C/C++ bindings

The Perl language is written in C (the new version will be actually written in C++ but it will take still many months before it will be available) and so in principle it is possibile to write in C "from scratch" bindings for a C/C++ library. On the other side doing that would be very tedious and it would require an extremely good knowledge of complicated Perl internals, so usually such interfaces are created through an intermediate "interface description" which hides most of the intricacies of the interface code.

There exist mainly two of such interfaces to build Perl bindings for C/C++ libraries: XS and SWIG. Once the interface code has been written, an interpreter is called to produce C code that will be linked to the Perl library allowing finally to be able to call library functions directly from Perl code. Both packages contain also a wrapper able to parse automatically C/C++ header files and produce the relative interface code.

The SWIG interface is developed by David Beazley (http://www.swig.org/) and implements C/C++ bindings for several scripting languages, including Perl, Python and Java. Unfortunately the SWIG automatic code generator of the interface from header files does not work very well dealing with complex packages.

The XS one [Sri97] is the native Perl glue for C/C++ code and it is provided together with the Perl package. Unfortunately also its automatic interface generator does not produce good code for complex libraries, that makes clear that to generate such interfaces is needed to develop a new wrapper.

Between the two we chose XS for a few reasons. First of all, an XS interface had already been developed at Crs4 by G.Zanetti for the Shapes C library, so that it was possible to start generating the interface having a working example by hand. On the other side, the fact that XS is included in the Perl distribution gives more warranties that a version of the interface will be kept in sync with the new versions of Perl. Moreover, most of CPAN packages (http://www.cpan.org/) are written using XS and so learning XS has the advantage of becoming able to customize those package in case of need.

Unfortunately, both XS and SWIG suffer by lack of documentation. A good list of links to XS docs is available at [Kei01]. Mainly, the docs we used in our work are the PerlGuts [P5P02], PerlXS [Roe96] and PerlXSTut [Oka99] man pages, the XS packages of PerlQt-1.04 and the XS package of Shapes developed here at Crs4.

XS provides support by default only for the simplest data types, namely number types and pointers to the *char* type, both in input and in output and the interface format for not overloaded functions is fairly simple: for example a function **f** with a header like

```
int f(int a, double b, char* c);
```

requires the following interface code

```
MODULE = Test        PACKAGE = Test
int
f(a,b,c)
    int a
    double b
    char* c
```

that in turn will be translated in the C code

```
XS(XS_Test_f)
{
   dXSARGS;
   if (items != 3)
      Perl_croak(aTHX_ "Usage: Test::f(a, b, c)");
```

```
    {
        int a = (int)SvIV(ST(0));
        double b = (double)SvNV(ST(1));
        char* c = (char *)SvPV(ST(2),PL_na);
        int RETVAL;
        dXSTARG;
        RETVAL = f(a, b, c);
        XSprePUSH; PUSHi((IV)RETVAL);
    }
    XSRETURN(1);
}
```

The meaning of this code is more or less the following: when the function **Test::f** is called from Perl, provided the **Test** package has been loaded, all arguments are stored inside an array **ST** of variables of type **SV*** (SV is an acronym for Scalar Value). To retrieve these args three functions are provided: **SvIV** to convert scalar values to integers, **SvNV** to convert them to floating point numbers and **SvPV** to convert them to strings (i.e. **char***).
Finally, the C function is evaluated with the three arguments and its return value stored in the variable **RETVAL** and pushed on top of the stack, so that it can be retrieved as output of the Perl function from the Perl program.
Support for default values is provided through an **if** instructions that counts the elements of **ST** and fills the missing ones with the values provided in the interface. The format is the most natural possible: for a function

```
int f(int a, double b = 3.4, char* c = "test");
```

we must provide the interface code

```
MODULE = Test        PACKAGE = Test
int
f(a,b=3.4,c="test")
    int a
    double b
    char* c
```

that tranforms in

```
XS(XS_Test_f)
{
    dXSARGS;
    if (items < 1 || items > 3)
```

```
        Perl_croak(aTHX_ "Usage: bobCookBook::Ex6::f(a, b=3.4, c =test)");
    {
        int a = (int)SvIV(ST(0));
        double b;
        char* c;
        int RETVAL;
        dXSTARG;
        if (items < 2)
          b = 3.4;
        else {
          b = (double)SvNV(ST(1));
        }
        if (items < 3)
          c = "test";
        else {
          c = (char *)SvPV(ST(2),PL_na);
        }
        RETVAL = f(a, b, c);
        XSprePUSH; PUSHi((IV)RETVAL);
    }
    XSRETURN(1);
}
```

The support for overloaded functions is slightly more complicated, because the C language does not support overloading and therefore it must be implemented in some non standard way. The method indeed is to create a unique function that contains all the overloading brothers, and every functions is chosen again through an **if** instruction. In particular, functions that differ only by arguments types that are indistinguishable by Perl, such as **int** and **long**, must be differntiated through the introduction of an **ALIAS** flag as well explained in the PerlXS man page. To every alias it will correspond in Perl a different function name, so that it will be possible to the C interface to understand which function to call.

The most complicated part of the XS interface is the support of more complex variable types such as pointers and double pointers, except for the **char*** case that is well supported.

For numerical types we used the same trick used by G. Zanetti in Shapes: using the code extracted from PP.pm file of the PDL (http://pdl.perl.org) Perl modules, a library for scientific computations, we can extract a pointer to the structure of PDL's core C routines and use it to

extract the pointer from a PDL variable using the **SvPDLV** functions, use the pointer to call the C function and finally return the output value to Perl.
For example a function

void h(int* a);

will generate at the end of the process a C code like

```
XS(XS_Test_h)
{
  dXSARGS;
  if (items != 1)
    Perl_croak(aTHX_ "Usage: Test::h(a)");
  {
    int* a;
    if (SvIOK(ST(0)) && SvIV(ST(0)) == 0) {
     a = (int *) NULL ;
    } else {
     pdl *a_pdl = (pdl *)(PDL->SvPDLV(ST(0)));
     a = (int *) ((a_pdl)->data) ;
    }
    h(a);
  }
  XSRETURN_EMPTY;
}
```

When a pointer of this kind is given in output, we bring back the problem to the previous one adding the output at the end of the interface definition.

The case of double pointers is much more complicated, and it makes use of perl pointers to arrays besides as before of pdl objects.

Let us start from the **char**** type: through the function **SvRV**, that extracts the pointer to the array, we create a perl **AV** pointer containing the data and extract one by one the strings from it using the function **av_fetch** putting them in a buffer **char**** variable, and finally set the C argument equal to the address of the buffer and use it in to call the C function.

For example a function

void h(char a);**

will generate at the end of the process a C code like

```
XS(XS_Test_h)
{
```

```
  dXSARGS;
  if (items != 1)
    Perl_croak(aTHX_ "Usage: Test::h(a)");
  {
    char** a;
    AV *ar ;
    char **buf;
    if (!SvROK(ST(0)) || SvTYPE(SvRV(ST(0))) != SVt_PVAV) {
    croak(a is not of a reference to an array);
    } else {
      int i;
      ar = (AV *) SvRV(ST(0)) ;
      buf = (char**) calloc(av_len(ar),sizeof(char*));
      for (i = 0 ; i <= av_len(ar) ; i++) {
        char *array = (char*)SvPV(*av_fetch(ar, i, 0), PL_na);
        buf[i] = array;
      }
    a = &(buf[0]) ;
    }
    h(a);
  }
  XSRETURN_EMPTY;
}
```

For numerical data types the algorithm is almost identical, except that the input array will contain PDL variables instead of strings and so we will have to extract the data from each variable using as before **SvPDLV**.

The support for C++ classes types is the most complicated. To make them the same kind of objects of native Perl classes, that are of type **HV**, i.e. pointers to associative arrays, we follow the implementation introduced by Ashley Winter in PerlQt 1.0. The standard implementation presented by the standard PerlXS documentation implements classes that interface C++ classes as **SV*** instead of a **HV*** and this introduces an asymmetry that can be an indesirable source of confusion.

In detail, we define two C functions called **objectify_ptr**, that takes a **SV*** containing the pointer of the C++ class and creates an associative array containing it, and **extract_ptr**, that takes an associative array containing a pointer of a C++ class and returns its pointer.

3 Generating a C++ intermediate library

Even though we can recover in Perl all functionalities of a C++ library, although at the price of some ALIAS, there is still something we cannot accomplish so easily. Many libraries indeed have defined internally a loop that constantly calls methods called “callbacks” that are meant to be overrided by the library classes, in order to provide an easy mechanism to customize the library behaviour in correspondance with some event. An example can be the way to redraw a window of a widget library, or often callbacks are used to define the action of the library in response to a mouse event.

The problem with callbacks is exactly that are called internally. The C++ library indeed knows nothing of the Perl interface and in particular knows nothing of the Perl classes that inherit from C++ classes through the XS interface, and therefore if nothing else is done it will not be possible to use from Perl any of the library callback. For libraries focused on other aspects, like the visualization library VTK (http://www.kitware.com/) or the modeling library OpenCascade (http://www.opencascade.org/), this may not be a big deal, but it becomes a big problem for widget libraries such as Qt (http://www.troll.no/) or GTK (http://www.gtk.org/) that are basically callbacks reservoirs.

In our knowledge, no solution for this problem is documented in any standard XS docs web page or book up to now. It is possible though to find a working example of such interface in the package PerlQt 1.0, by Ashley Winters, developed under the old perl5.00, and in fact what we did is to study the structure of such interface and sync it with the new stable perl 5.6.1 release. The main difficulty we met is the fact that many functions used in this interface are non documented Perl internal functions or at the best scarcely documented in the PerlGuts man page, so that the only way to understand their behaviour is to examine Perl source code and pass through a trial-and-error long phase.

After a tiring examination of the PerlQt 1.0 code the following picture became clear: the only way to make the C++ library be “Perl-aware”, so that C++ callbacks can be overrided by Perl functions, is to add a C++ layer, i.e. build a parallel dummy C++ library that contains a new class for each original C++ class, and this new class overrides exactly the set of virtual functions of the original one, set that contains the set of all callback functions. The PerlXS code finally will interface this new library, that will function as a cushion between Perl and the original C++ library.

The idea is that inside each of the dummy functions we put internal Perl code that checks whether a function with that name has been implemented in the calling Perl class: if such function is found then another Perl internal function will take care of redirecting the C++ function to the Perl one, while if the function is not overrided in Perl then the C++ original function is executed.

4 The Parser

All libraries interesting enough have a number of functions too high for a construction by hand of a Perl interface, so it is needed to use some automatic program able to parse the library header files and generate from them all the code, i.e. the dummy C++ library and the PerlXS interface code.

As we have already noticed two parsers are provided both with the SWIG and XS packages, but unfortunately they did not prove until now to have any success with huge C++ packages such as Qt, VTK or OpenCascade. This is why we decided to build a new parser able to extract as many C++ functions as possible and moreover to keep track of inheritances, structures, enumerations and constants. We called it SWING just because its aim was similar to the SWIG package.

After a first phase it became clear that the parsing code tended to become pretty complicated as all possible features were added and every new correction needed more and more time to be done, so we decided to convert all SWING code to the present object oriented format.

Five classe now share the package code: Package, Class, Function, Argument and Header.

The first one is used mainly to store all C++ library classes data plus extra informations like the list of types we do not support.

The Class class instead is one of the two most complex: it contains calls for parsing C++ classes code, that is easily collected by the parser as all lines between the opening and the closing curly brackets corresponding to a C++ class definition; to add and retrieve class's functions, structures and enumerates and finally to generate the beginning of the C++ and XS file and the XS conditionals for its overrided functions

The Function class analogously takes care of the functions data passed to it by the Class package, easily spotted as the code between the opening and the closing parenthesis of the function declaration. In particular it contains all calls to parse its code and store all of its flags and args and finally to produce the function C++ and XS code.

The last two classes are used to store the single function arguments and their flags (Argument) and the list of all header files to call (Header) for a Class contained in a particular header file.

Finally, the main SWING script takes care of creating the Makefile of the package and a few extra files.

5 Qt Quirks

For generic C++ libraries the parser described above should be enough to produce a working Perl-C++ interface implementing also callbacks.

Unfortunately Qt, one the main reasons for the development of the SWING package, contains other quirks inside: slots and signals. These two new concepts are just two new kinds of special functions introduced to make easier customizing the communication between widgets: slots are virtual functions whose action can be triggered by the action of an event called signal.

The implementation of this schema is not trivial at the Qt level itself: to be able to use signals from a derived class indeed a script called **moc** is provided with the Qt distribution to generate extra C++ code to store the slot and signal functions in an internal reservoir called "metaObject".

To implement also this feature in the Perl interface we must therefore add in the dummy C++ interface exactly the same instructions automatically generated by the **moc** script. Such job had been done already by Winters in his PerlQt1.0 but the mechanism changed a little since then, so we had also this time to procedd by trial and error to find out how the things worked and how to upgrade them to newer versions of Perl and Qt.

6 Conclusions

Lot of efforts have been put in understanding how to build an automatic and robust Perl interface to C++ libraries in order to support pointers and double pointers, overrids of functions and callbacks.

Almost no documentation exist about how to accomplish this result, and in our knowledge no such parser is available, nor under GPL license nor for sale.

On the other side, there are many reasons to think that having such package would be of huge value for Crs4, allowing on one side to speed up algorithms creation and on the other side to make much easier to use complex code in low level computer science classes.

Almost all work now has been completed and its application to the Qt case is close to be operative. Next important step will be to apply all this to the even bigger library OpenCascade. In this case though the only real problem seems to come from its size, as the native callback calls will not be used as all applications should be embedded in a Qt environment and so will depend on Qt callbacks.

Bibliography

[Dal02] M. K. Dalheimer, *Programming with Qt, 2nd Edition*, O'Reilly and Associates, 2002

[Kei01] John Keiser, *Gluing C++ And Perl Together*, http://www.johnkeiser.com/perl-xs-c++.html

[Oka99] Jeff Okamoto, *PerlXS Tutorial*, http://www.perldoc.com/perl5.6/pod/perlxstut.html

[Ous98a] John K. Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, IEEE Computer Magazine, March 1998, http://home.pacbell.net/ouster/scripting.html

[Ous98b] J. Ousterhout, *Additional Information for Scripting White Paper*, http://home.pacbell.net/ouster/scriptextra.html

[P5P02] Perl 5 Porters, *Introduction to the Perl API*, http://www.perldoc.com/perl5.6/pod/perlguts.html

[Roe96] Dean Roehrich, *XS language reference manual*, http://www.perldoc.com/perl5.6/pod/perlxs.html

[Sri97] Sriram Srinivasan, *Advanced Perl Programming*, O'Reilly and Associates, 1997

[WCS96] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl, Second Edition*, O'Reilly and Associates, ISBN 1-56592-149-6, 1996.