Perl Extension Building with SWIG

David M. Beazley
Dept. of Computer Science
University of Chicago
Chicago, IL 60637

David Fletcher
Fusion MicroMedia, Corp.
Longmont, CO 80501

Dominique Dumont
Hewlett Packard
Lab TID
5 Ave Raymond Chanas
38053 Grenoble cedex 9
France

Abstract

SWIG (Simplified Wrapper and Interface Generator) is a freely available tool that integrates Perl, Python, Tcl, and other scripting languages with programs written in C, C++, and Objective-C. This paper provides an introduction to SWIG and shows how it can be used to construct Perl extension modules. In addition, a number of applications in which SWIG has been utilized are described. While SWIG is similar to other Perl extension building tools such as xsubpp and h2xs, SWIG has a number of unique features that help simplify the task of creating Perl extension modules. Many of these features are described as well as limitations and future directions. This paper is primarily intended for developers who are interested in combining Perl with applications written in C or C++ as well as current SWIG users who are interested in learning more about some of SWIG's advanced features.

1 Introduction

One of Perl's greatest strengths is its ability to simplify hard programming tasks as well as being able to solve the odd and varied computing problems that occur on a day-to-day basis. While it would be nice to use Perl (or other high-level languages) for everything, this is simply not practical for many applications. In fact, performance critical tasks, low-level systems programming, and complex data structures are likely to be implemented in a compiled language such as C or C++ (and may be easier to manage in such languages). Furthermore, developers often need to work with a wide variety of existing applications and "legacy" systems that are written in such languages.

The integration of Perl and code written in compiled languages has a number of practical benefits. First, it allows existing C/C++ applications to be incorporated into a high-level interpreted environment. This environment provides greater flexibility and often simplifies development since debugging and testing can be performed using Perl scripts. Second, Perl can serve as a powerful user interface. In other words, rather than writing a user interface from scratch, it is possible to use a Perl interpreter instead. This also allows other for other possibilities such as graphical user interface development with Perl/Tk. Finally, Perl provides developers with a mechanism for assembling and controlling software components. Rather than creating a huge monolithic package, C/C++ programs can be packaged as collections of Perl extension modules. As a result, programs become more modular and easier to maintain. Furthermore, it is even possible to combine entirely different programs together within a shared Perl interpreter.

This paper provides an introduction and overview of SWIG, a tool designed to integrate C code with a variety of scripting languages including Perl, Python, and Tcl. Currently, SWIG can construct Perl extension modules on Unix and Windows-NT systems. It also supports the ActiveState Perl for Windows and Perl4. SWIG has been freely available since February, 1996 and has been previously described in *Advanced Perl Programming*, *The Perl Journal*, and *Dr. Dobb's Journal*[1, 2, 3]. In addition, SWIG is packaged with a 300 page user manual describing its use [4]. The goal of this paper is not to repeat all of this information, but to provide an overview of SWIG, demonstrate the use of some of its more advanced features, and describe some of the ways that it is currently being used. The authors include the developer of SWIG and two of SWIG's foremost Perl experts who have made substantial contributions to SWIG's development.

2 Perl Extension Building

To interface Perl with code written in C or C++, it is necessary to write wrappers that serve as a glue layer between the Perl interpreter and the underlying C code.

These wrappers are responsible for converting data between Perl and C, reporting errors, and other tasks. Perl is packaged with several tools for creating these wrappers. One such tool is `xsubpp`, a compiler that takes interface definitions written in a special language known as XS and converts them into wrappers. For example, suppose that you had the following C function:

```
int fact(int n);
```

To wrap this function into a Perl module with `xsubpp`, you would write the following XS file:

```
/* file : example.xs */
extern int fact(int n);
MODULE = Example     PACKAGE = Example
int
fact(n)
     int    n
```

When processed with `xsubpp`, the following wrapper file is produced

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
extern int fact(int n);

XS(XS_Example_fact)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Example::fact(n)");
    {
        int     n = (int)SvIV(ST(0));
        int     RETVAL;
        RETVAL = fact(n);
        ST(0) = sv_newmortal();
        sv_setiv(ST(0), (IV)RETVAL);
    }
    XSRETURN(1);
}

XS(boot_Example)
{
    dXSARGS;
    char* file = __FILE__;
    XS_VERSION_BOOTCHECK ;
    newXS("Example::fact",
           XS_Example_fact, file);
    ST(0) = &sv_yes;
    XSRETURN(1);
}
```

To use the module, the wrapper code must be compiled and linked into a shared library that can be dynamically loaded into the Perl interpreter. The easiest way to do this is with the MakeMaker utility by writing a script as follows:

```
# file : Makefile.PL
use ExtUtils::MakeMaker;
WriteMakefile(
   'NAME' => 'Example',
   'OBJECT' => 'example.o fact.o'
);
```

This script is then used to create a Makefile and module as follows:

```
unix > perl Makefile.PL
unix > make
unix > make install
```

Finally, in addition to creating the C component of the extension module, it is necessary to write a `.pm` file that is used to load and initialize the module. For example,

```
# file : Example.pm
package Example;
require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
bootstrap Example;
1;
```

At this point, you should have a working Perl extension module. In principle, building a Perl extension requires an XS specification for every C function that is to be accessed. To simplify the process of creating these specifications, Perl includes `h2xs`, a tool that converts C header files to XS descriptions. While useful, `h2xs` is somewhat limited in its ability to handle global variables, structures, classes, and more advanced C/C++ features. As a result, `h2xs` can be somewhat difficult to use with more complex applications.

3 SWIG Overview

In a nutshell, SWIG is a specialized compiler that transforms ANSI C/C++ declarations into scripting language extension wrappers. While somewhat similar to `h2xs`, SWIG has a number of notable differences. First, SWIG is much less internals oriented than XS. In other words, SWIG interfaces can usually be constructed without any knowledge of Perl's internal operation. Second, SWIG is designed to be extensible and general purpose. Currently, wrappers can be generated for Perl, Python, Tcl, and Guile. In addition, experimental modules for MATLAB and Java have been developed. Finally, SWIG supports a larger subset of C and C++ including structures, classes, global variables, and inheritance. This section provides a tour of SWIG and describes many of its interesting features.

3.1 A Small Taste

As a first example, suppose that you wanted to build a Perl interface to Thomas Boutell's gd graphics library.[1] Since gd is a C library, images are normally created by writing C code such as follows:

```
#include "gd.h"
int main() {
  gdImagePtr  im;
  FILE       *out;
  int         blk,wht;

  /* Create an image */
  im=gdImageCreate(200,200);

  /* Allocate some colors */
  b=gdImageColorAllocate(im,0,0,0);
  w=gdImageColorAllocate(im,255,255,255);

  /* Draw a line */
  gdImageLine(im,20,50,180,140,w);

  /* Output the image */
  out=fopen("test.gif","wb");
  gdImageGif(im,out);
  fclose(out);

  /* Clean up */
  gdImageDestroy(im);
}
```

By building a Perl interface to gd, our goal is to write similar code in Perl. Thus, the functionality of the gd library must be exposed to the Perl interpreter. To do this, a SWIG interface file can be written as follows:

```
// File : gd.i
%module gd
%{
#include "gd.h"
%}

typedef gdImage *gdImagePtr;

gdImagePtr gdImageCreate(int sx, int sy);
void       gdImageDestroy(gdImagePtr im);
void       gdImageLine(gdImagePtr im,
                       int x1, int y1,
                       int x2, int y2,
                       int color);
int   gdImageColorAllocate(gdImagePtr im,
                           int r, int g, int b);
void gdImageGif(gdImagePtr im, FILE *o);
// File I/O (explained shortly)
FILE *fopen(char *name, char *mode);
void  fclose(FILE *);
```

In this file, the ANSI C prototypes for every function that we would like to access from Perl are listed. In addition, a number of SWIG directives (which are always preceded by a "%") appear. The `%module` directive specifies the name of the extension module. The `%{, %}` block is used to insert literal code into the output wrapper file.[2] In this case, we simply include the "`gd.h`" header file. Finally, a few file I/O functions also appear. While not part of gd, these functions are needed to manufacture file handles used by several gd functions.

To run SWIG, the following command is executed:

```
unix > swig -perl5 gd.i
Generating wrappers for Perl 5
```

This produces two files, `gd_wrap.c` and `gd.pm`. The first file contains C wrapper functions that appear similar to the output that would have been generated by `xsubpp`. The `.pm` file contains supporting Perl code needed to load and use the module.

To build the module, the wrapper file is compiled and linked into a shared library. This process varies on every machine (consult the man pages), but the following steps are performed on Linux:

```
% gcc -fpic -c gd_wrap.c \
  -Dbool=char \
  -I/usr/lib/perl5/i586-linux/5.004/CORE
% gcc -shared gd_wrap.o -lgd -o gd.so
```

At this point, the module is ready to use. For example, the earlier C program can be directly translated into the following Perl script:

```
#!/usr/bin/perl
use gd;

# Create an image
$im = gd::gdImageCreate(200,200);

# Allocate some colors
$b=gd::gdImageColorAllocate($im,0,0,0);
$w=gd::gdImageColorAllocate($im,255,
                            255,255);

# Draw a line
gd::gdImageLine($im,20,50,180,140,$w);

# Output the image
$out=gd::fopen("test.gif","wb");
gd::gdImageGif($im,$out);
```

[1] gd is a freely available graphics library for producing GIF images and can be obtained at `www.boutell.com/gd/gd.html`. A Perl module to gd, developed by Lincoln Stein, is also available on CPAN so interested readers are encouraged to compare the results of using SWIG against an existing Perl extension.

[2] This syntax is derived from lex and yacc.

```
gd::fclose($out);

# Clean up
gd::gdImageDestroy($im);
```

3.2 Input Files

In the gd example, SWIG was given a special interface file containing a list of the C declarations to be included in the Perl module. When working with a large C library, interface files can often be constructed by copying an existing header file and modifying it slightly. However, in some cases, it is possible to include a header file as follows:

```
%module
%{
#include "gd.h"
%}

// Grab the declarations from gd.h
%include "gd.h"

// Some file I/O functions
FILE *fopen(char *name, char *mode);
void  fclose(FILE *);
```

The `%include` directive tells SWIG to include a file and parse all of the declarations it contains. In this case, the interface would now wrap every function in the gd library as opposed to the half-dozen functions listed in the first example.

SWIG also includes a C preprocessor that can be used for macro expansion and conditional compilation. If a new application is being written with SWIG in mind, header files can be written as follows:

```
#ifdef SWIG
%module gd
%{
#include "gd.h"
%}
#endif

/* C declarations */
...
```

With this approach, the file can serve as both a valid C header file and as an interface specification. The `SWIG` symbol is only defined when SWIG is parsing so special directives can be easily hidden from the C compiler as needed.

Finally, for the truly lazy, SWIG can sometimes be run directly on C header and source files. For example,

```
% swig -perl5 -module gd gd.h
% swig -perl5 -module example example.c
```

Most users, however, use a mix of dedicated interface files and header files.

3.3 Data Model

The most critical part of interfacing Perl to C programs is the management of data. Since Perl and C utilize a different set of internal datatypes, wrapper generators are responsible for producing code that marshals data and objects between languages. For fundamental types such as `int` and `double` the conversion process is straightforward. However, pointers, arrays, structures, and objects complicate the process. Furthermore, since most C/C++ programs make extensive use of these datatypes, it is important for wrapper generators to support as many of these datatypes as possible.

3.3.1 Pointers

SWIG maps C pointers and C++ references into Perl blessed references. These references contain both the value of the pointer itself, plus a type-signature. In the gd example, pointers were used to manage both images and files. If one were to print out the value a pointer, it would appear as follows:

```
gdImagePtr=SCALAR(0x80b9914)
```

SWIG uses the type-signature to perform run-time checking of all pointer values. These checks emulate many of the checks that would have been performed by a C compiler. When an invalid Perl datatype or pointer of invalid type is used, a run-time error is generated. For example,

```
% perl
use gd;
$f = gd::fopen("test.gif","w");
gd::gdImageLine($f,20,50,180,140,0);
Type error in argument 1 of gdImageLine.
Expected gdImagePtr. at - line 3.
```

Type-checking is based on the name of each datatype. However, the type-checker also keeps track of C++ inheritance hierarchies and `typedef` definitions. Thus, an acceptable pointer type includes any alternate names that might have been created with a `typedef` declaration as well as any derived datatypes in C++.

When pointers are manipulated in Perl, they are opaque values. That is, pointers can be created and passed around to other C functions, but they can not be dereferenced directly. Thus, in the example, it is difficult (or impractical) for a user to directly manipulate the internal representation of an image from the Perl interpreter. Furthermore, SWIG, by default, handles all pointers in a uniform manner. Thus, datatypes such as `FILE *` are represented as blessed references even though such types may appear remarkably similar to other Perl datatypes such as file handles.

3.3.2 Arrays

SWIG maps all arrays into pointers where the "value" of an array is simply a pointer to the first element in the array. This is the same model used by C compilers and like C, SWIG performs no bounds or size checking. Thus, a function such as

```
void foo(double a[4][4]);
```

would accept *any* object of type `double *`. It is up to the user to ensure that the pointer is valid and that it points to memory that has been properly allocated.

3.3.3 Structures and Objects

Finally, all structures and objects are represented as pointers. This includes cases where objects are manipulated by value. For example, the functions

```
double dot_product(Vector a, Vector b);
Vector cross_product(Vector a, Vector b);
```

are transformed by SWIG into the following wrappers:[3]

```
double
wrap_dot_product(Vector *a,Vector *b)
{
  return dot_product(*a,*b);
}
Vector *
wrap_cross_product(Vector *a,Vector *b)
{
  Vector *r;
  r=(Vector *) malloc(sizeof(Vector));
  *r=cross_product(*a,*b);
  return r;
}
```

The representation of objects by reference avoids the problem of marshaling objects between a C and Perl representation–a process that would be extremely difficult for very complicated C datatypes. It also provides better performance since manipulating references is more efficient than copying object data back and forth between languages. Finally, the use of references closely matches the way in which most C/C++ programs already handle objects.

The downside to this approach is that objects are opaque in Perl. This prevents users from examining their contents directly. In addition, SWIG wrappers occasionally need to perform implicit memory allocations as shown above. It is up the user to free the resources used by such functions (or learn to live with a memory leak). Of course, this naturally brings us to the next topic.

[3] When C++ is used, SWIG uses the default copy constructor instead of `malloc`.

3.3.4 Memory Management

SWIG maintains a strict separation between the management of Perl and C objects. While Perl uses reference counting to keep track of its own objects, this scheme is not extended to C/C++ extensions created with SWIG. Thus, when Perl destroys a blessed reference containing the value of a C pointer, only the pointer value disappears, not the underlying C data that it points to.

From a user standpoint, SWIG generated C/C++ extensions follow the same memory management rules as the underlying application. Thus, if a program relies on `malloc` and `free` to allocate and deallocate objects, these will also be used from the Perl interpreter. Likewise, a C++ extension typically requires explicit invocation of constructors and destructors. Furthermore, for functions that implicitly allocate memory as in the previous section, it is up to the user to explicitly destroy the result using `free` or a C++ destructor. While such a scheme may seem problematic, it is no less problematic than memory management in C (which may or may not be a good thing depending on your point of view). Even if it were possible to have Perl automatically manage C/C++ objects, this would be an inherently dangerous affair–especially since Perl has no way to know how an underlying C application really operates. Furthermore, it would be a fatal error for Perl to deallocate objects that were still in use. Therefore, SWIG leaves memory management largely up the user.

3.3.5 Pointers, Arrays, and Perl

A common confusion among some novice users is the difference between C datatypes and similar Perl datatypes. In particular, Perl references are not the same as a C pointers and Perl arrays are not the same as C arrays. Differences also apply to other datatypes such as files (this is the reason that the simple example included prototypes for `fopen` and `fclose`). The primary reason for these differences is that objects in Perl have a different internal representation than objects in C. For example, a Perl array is represented as a collection of references to Perl objects which may be of mixed types. The internal representation of this array is entirely different than what would be used for a normal C array. Therefore, it is impossible to take a Perl array and pass it in unmodified form to an arbitrary C function.

The difference between Perl and C datatypes often arises with C functions such as the following:

```
/* Plot some points */
void
plotpts(gdImagePtr im, int x[], int y[],
        int npts,  int c)
{
```

```
   for (int i = 0; i < npts; i++) {
      gdImageSetPixel(im,x[i],y[i],c);
   }
}
```

Ideally, a user might want to pass Perl arrays as arguments as follows:

```
@a = (10,20,30,40);
@b = (50,70,60,200);
gd::plotpts($im,\@a,\@b,4,1); # Error!
```

However, this script generates a type error instead of acting as one might expect. While such behavior may seem restrictive or bizarre, SWIG has been deliberately designed to operate in this manner. In fact, there are even benefits to this approach. If Perl arrays were to be used as C arrays, a copy would be made, verified for type-correctness, and deallocated every time an array was passed to a C function. For large arrays, this would introduce a substantial performance overhead. Space requirements are also a concern for some C programs. For example, a numerical application might manipulate arrays with millions of elements. Converting such arrays to and from a Perl representation would clearly introduce substantial memory and performance overhead. In contrast, manipulating pointers to such arrays is easy and efficient.

It should also be noted that SWIG provides a variety of customization options that can be used to change its behavior. In fact, one can even make SWIG map Perl arrays into C arrays if desired. Therefore, most restrictions can be eliminated with a little extra work. Some of these customization techniques are described shortly.

3.4 Helper Functions

Sometimes the Perl interface constructed by SWIG is lacking in functionality or is difficult to use. For example, in the previous section, a function operating on C arrays was presented. To construct C arrays from Perl, it is necessary to add some additional functions to the SWIG interface. This can be done using the `%inline` directive as follows:

```
// Add some helper functions for C arrays
%inline %{
int *int_array(int size) {
   return (int *)
      malloc(sizeof(int)*size);
}
void int_destroy(int *a) {
   free(a);
}
void int_set(int *a, int i, int val) {
   a[i] = val;
}
int int_get(int *a, int i) {
   return a[i];
}
%}
```

When SWIG builds the scripting interface, these functions become part of the extension module and can be used as follows:

```
# Convert a Perl array into a C int array
sub create_array {
 $len = scalar(@_);
 $ia = gd::int_array($len);
 for ($i = 0; $i < $len; $i++) {
    val = shift;
    gd::int_set($ia,$i,$val);
 }
 return $ia;
}

@a = (10,20,30,40);
@b = (50,70,60,200);
$ia = create_array(@a);# Create C arrays
$ib = create_array(@b);
gd::plotpts($im,$ia,$ib,4,1);
...
gd::int_destroy($ia);
gd::int_destroy($ib);
```

3.5 Classes and Structures

While SWIG represents all objects as opaque pointers, the contents of an object can be examined and modified through the use of accessor functions as follows:

```
/* Extract data from an object */
double Point_x_get(Point *p) {
    return p->x;
}
/* Invoke a C++ member function */
int Foo_bar(Foo *f) {
    return f->bar();
}
```

From a Perl script, a user simply passes an object pointer to accessor functions to extract internal information or invoke member functions.

While it is possible to write accessor functions manually, SWIG automatically creates them when it is given structure and class definitions. For example, in the gd library, the following structure is used to contain image information:

```
typedef struct gdImageStruct {
        unsigned char ** pixels;
        int sx;
        int sy;
        int colorsTotal;
...
} gdImage;
```

If this structure definition were placed in the SWIG interface file, accessor functions would automatically be created. These could then be used to extract information about images as follows:

```
#!/usr/bin/perl
use gd;
$im = gd::gdImageCreate(400,300);
# Print out the image width
print gd::gdImage_sx_get($im), "\n";
```

Accessor functions are also created for C++ classes and Objective-C interfaces. For example, the class definition

```
class List {
public:
   List();
  ~List();
   void insert(Object *);
   Object *get(int i);
   int  length();
   ...
};
```

is translated into the following accessor functions:

```
List *new_List() {
   return new List;
}
void delete_List(List *l) {
   delete l;
}
void List_insert(List *l, Object *o) {
   l->insert(o);
}
...
```

3.6 Shadow Classes and Perl Objects

As an optional feature, the accessor functions created by SWIG can be used to write Perl wrapper classes (this is enabled by running SWIG with the `-shadow` option). While all the gory details can be found in the SWIG Users Manual, the general idea is that the accessor functions can be encapsulated in a Perl class that mimics the behavior of the underlying object. For example,

```
package List;
@ISA = qw( example );
sub new {
    my $self = shift;
    my @args = @_;
    $self = new_List(@args);
    return undef if (!defined($self));
    bless $self, "List";
    my %retval;
    tie %retval, "List", $self;
    return bless \%retval,"List";
}
sub DESTROY {
    delete_List(@_);
}
sub insert {
    return $result = List_insert(@_);
}
...
```

This class provides a wrapper around the underlying object and is said to "shadow" the original object. Shadow classes allow C and C++ objects to be used from Perl in a natural manner. For example,

```
$l = new List;
$l->insert($o);
...
$l->DESTROY();
```

For C structures, access to various attributes are provided through tied hash tables. For the gd library, members of the image data structure could be accessed as follows:

```
$im = gd::gdImageCreate(400,400);
$width = $im->{sx};
$height = $im->{sy};
...
```

The other significant aspect of shadow classes is that they allow Perl to perform a limited form of automatic memory management for C/C++ objects. If an object is created from Perl using a shadow class, the `DESTROY` method of that class automatically invokes the C++ destructor when the object is destroyed. As a result, C/C++ objects wrapped by shadow classes can be managed using the same reference counting scheme utilized by other Perl datatypes.

3.7 Class Extension

When building object-oriented Perl interfaces, it is sometimes useful to modify or extend objects with new capabilities. For example, the gd library defines the following data structure for defining points:

```
typedef struct {
    int x,y;
} gdPoint;
```

To make this structure more useful, we can add constructors, destructors, and various methods to it (regardless of whether it is implemented in C or C++). To do this, the SWIG `%addmethods` directive can be used as follows:

```
/* Add some methods to points */
%addmethods gdPoint {
/* Create a point or array of points */
gdPoint(int npts = 1) {
  return (gdPoint *)
      malloc(sizeof(gdPoint)*npts);
}
/* Destroy a point */
~gdPoint() {
  free(self);
}
/* Array indexing */
gdPoint *get(int i) {
  return self+i;
}
/* A debugging function */
void output() {
  printf("(%d,%d)\n",self->x,self->y);
}
};
```

Now, in the Perl interface `gdPoint` will appear just like a class with constructors, destructors, and methods. For example,

```
# Create a point
$pt = new gdPoint;
$pt->{x} = 20;
$pt->{y} = 50;
$pt->output();

# Create an array of points
$pts = new gdPoint(10);
for ($i = 0; $i < 10; $i++) {
   $p = $pts->get($i);
   $p->{x} = $i;
   $p->{y} = 10*$i;
}

# Pass the points to a function
gd::gdImagePolygon($im,$pts,10,1);
...
```

The class extension mechanism is also a powerful way to repackage existing functionality. For example, the `gdImage` structure and various functions in the gd library could be combined into a Perl class as follows:

```
%addmethods gdImage {
gdImage(int w, int h) {
  return gdImageCreate(w,h);
}
~gdImage() {
  gdImageDestroy(self);
}
int
colorAllocate(int r, int g, int b) {
  return
    gdImageColorAllocate(self,r,g,b);
}
void
line(int x1,int y1,int x2,int y2,int c){
  gdImageLine(self,x1,y1,x2,y2,c);
}
...
};
```

Users can now write scripts as follows:

```
#!/usr/bin/perl
use gd;
$im=new gdImage(400,400);
$black=$im->colorAllocate(0,0,0);
$white=$im->colorAllocate(255,255,255);
$im->line(20,50,180,140,$white);
...
```

With these simple modifications, our interface is already looking remarkably similar to that used in the GD module on CPAN. However, more improvements will be described shortly.

3.8 Access Control and Naming

In certain instances, it may be useful to restrict access to certain variables and class members. Hiding objects is easy–simply remove them from the interface file. Providing read-only access can be accomplished using the `%readonly` and `%readwrite` directives. For example,

```
// Create read-only variables
%readonly
int foo;         // Read-only
double bar;      // Read-only
%readwrite

// Create read-only class members
class List {
...
%readonly
    int length;  // Read-only member
%readwrite
...
}
```

When read-only mode is used, attempts to modify a value from Perl result in a run-time error.

Another common problem is changing the name of various C declarations. For example, a C function name may conflict with an existing Perl keyword or subroutine. To fix this problem, the `%name` directive can be used. For example,

```
%name(cpack) void pack(Object *);
```

creates a new command "cpack." If name conflicts occur repeatedly, the `%rename` directive can be used to change all future occurrences of a particular identifier as follows:

```
%rename pack cpack;
```

The renaming operations can also be applied to C/C++ class and structure names as needed. For example,

```
%name(Image) class gdImage {
...
}
```

3.9 Exception handling

To catch errors, SWIG allows users to create user-defined exception handlers using the `%except` directive. These handlers are responsible for catching and converting C/C++ runtime errors into Perl errors. As an example, the following error handler can be used to catch errors in the standard C library:

```
%except(perl5) {
   errno = 0;
   $function
   if (errno) {
      die(strerror(errno));
   }
}
```

When defined, the exception handling code is placed into all of the wrapper functions. In the process, the `$function` token is replaced by the actual C function call. For the example shown, the exception handler resets the `errno` variable and calls the C function. If the value of `errno` is modified to a non-zero value, an error message is extracted from the C library and reported back to Perl.

While catching errors in the C library has been illustrated, exception handlers can also be written to catch C++ exceptions or to use any special purpose error handling code that might be present in an application.

3.10 Typemaps

Typemaps are one of SWIG's most powerful features and the primary means of customization. Simply stated, a typemap is a small bit of C code that can be given to SWIG to modify the way that it processes specific datatypes. For instance, Perl arrays can be converted into C arrays, Perl references can be substituted for pointers, and so forth. This section briefly introduces typemaps and their use. However, typemaps are a complicated topic so it is impossible to cover all of the details here and interested readers are strongly advised to consult the SWIG documentation.

3.10.1 Example: Output Values

As a first typemap example, consider a function that returns values through its parameters as follows:

```
void
imagesize(gdImagePtr im,int *w,int *h) {
    *w = gdImageSX(im);
    *h = gdImageSY(im);
}
```

As is, this function would be difficult to use because the user must write helper functions to manufacture, dereference, and destroy integer pointers. These functions might be used as follows:

```
$wptr = new_integer();# Create an 'int *'
$hptr = new_integer();
imagesize($im, $wptr, $hptr);
$w = integer_value($wptr);# Dereference
$h = integer_value($hptr);
delete_integer($wptr);    # Clean up
delete_integer($hptr);
```

A more elegant solution is to use the SWIG typemap library in the interface file as follows:

```
%include typemaps.i
void
imagesize(gdImagePtr im, int *OUTPUT,
          int *OUTPUT);
```

Now, in the Perl script, it is possible to do this:

```
($w,$h) = imagesize($im);
```

In a similar spirit, it is also possible to use Perl references. For example:

```
%include typemaps.i
void
imagesize(gdImagePtr im, int *REFERENCE,
          int *REFERENCE);
```

Now in Perl:

```
# Return values in $w and $h
imagesize($im,\$w,\$h);
```

To implement this behavior, the file `typemaps.i` defines a collection of typemap "rules" that are attached to specific datatypes such as `int *OUTPUT` and `int *REFERENCE`. The creation of these rules is now discussed.

3.10.2 Creating New Typemaps

All wrapper functions perform a common sequence of internal "operations." For example, arguments must be converted from Perl into a C representation, a function's return value must be converted back into Perl, argument values might be checked, and so forth. SWIG gives each of these operations a unique name such as "in" for input parameter processing, "out" for returning values, "check" for checking values, and so forth. Typemaps allow a user to re-implement these operations for specific datatypes by supplying small fragments of C code that SWIG inserts into the resulting wrapper code.

To illustrate, consider the gd example. In the original interface file, two functions were included to open and close files. These were required because SWIG normally maps all pointers (including files) into blessed references. Since a blessed reference is not the same as a Perl file handle, it is not possible to pass Perl files to functions expecting a `FILE *`. However, this is easily changed with a typemap as follows:

```
%typemap(perl5,in) FILE * {
   $target = IoIFP(sv_2io($source));
}
```

This declaration tells SWIG that whenever a `FILE *` appears as a function parameter, it should be converted using the supplied C code. When generating wrappers, the typemap code is inserted into all wrapper functions where a `FILE *` is involved. In the process the `$source` and `$target` tokens are replaced by the names of C local variables corresponding to the Perl and C representations of an object respectively. As a result, this typemap allows Perl files to be used in a natural manner. For example,

```
open(OUT,">test.gif") || die "error!\n";

# Much better than before
gd::gdImageGif($im,*OUT);
```

Certain operations, such as output values, are implemented using a combination of typemaps as follows:

```
%typemap(perl5,ignore)
int *OUTPUT(int temp) {
  $target = &temp;
}
%typemap(perl5,argout) int *OUTPUT {
  if (argvi >= items) {
     EXTEND(sp,1);
  }
  $target = sv_newmortal();
  sv_setiv($target,(IV) *($source));
  argvi++;
}
```

In this case, the "ignore" typemap tells SWIG that a parameter is going to be ignored and that the Perl interpreter will not be supplying a value. Since the underlying C function still needs a value, the typemap sets the value of the parameter to point to a temporary variable `temp`. The "argout" typemap is used to return a value held in one of the function arguments. In this case, the typemap extends the Perl stack (if needed), and creates a new return value. The `argvi` variable is a SWIG-specific variable containing the number of values returned to the Perl interpreter (so it is incremented for each return value).

The C code supplied in each typemap is placed in a private scope that is not visible to any other typemaps or other parts of a wrapper function. This allows different typemaps to be used simultaneously–even if they define variables with the same names. This also allows the same typemap to be used more once in the same wrapper function. For example, the previous section used the `int *OUTPUT` typemap twice in the same function without any adverse side-effects.

3.10.3 Typemap Libraries

Writing new typemaps is a somewhat magical art that requires knowledge of Perl's internal operation, SWIG, and the underlying application. Books such as *Advanced Perl Programming* and the man pages on extending and embedding the Perl interpreter will prove to be quite useful. However, since writing typemaps from scratch is difficult, SWIG provides a way for typemaps to be placed in a library and utilized without knowing their internal implementation details. To illustrate, suppose that you wanted to write some generic typemaps for checking the value of various input parameters. This could be done as follows:

```
// check.i
// typemaps for checking argument values
%typemap(perl5,check) Number POSITIVE {
  if ($target <= 0)
    croak("Expected a positive value");
}

%typemap(perl5,check) Pointer *NONNULL {
  if ($target == NULL)
    croak("Received a NULL pointer!");
}
```

To use these typemaps, a user could include the file `check.i` and use the `%apply` directive. The `%apply` directive simply takes existing typemaps and makes them work with new datatypes. For example:

```
%include check.i

// Force 'double px' to be positive
%apply Number Positive { double px };

// Force these pointers to be NON-NULL
%apply Pointer NONNULL { FILE *,
                         Vector *,
                         Matrix *,
                         gdImage * };

// Now some functions
double log(double px);// 'px' positive
double dot_product(Vector *, Vector *);
...
```

In this case, the typemaps we defined for checking different values have been applied to a variety of new datatypes. This has been done without having to examine the implementation of those typemaps or having to look at any Perl internals. Currently, SWIG includes a number of libraries that operate in this manner.

3.11 Other SWIG Features

SWIG has a number of other features that have not been discussed. In addition to producing wrapper code, SWIG also produces simple documentation files. These describe the contents of a module. In addition, C comments can be used to provide descriptive text in the documentation file. SWIG is also packaged with a library of useful modules that include typemaps and interfaces to common libraries. These libraries can simplify the construction of scripting interfaces.

3.12 Putting it All Together

In the first part of this section, a minimal interface to the gd library was presented. Now, let's take a look at a more substantial version of that interface.

```
// gd.i
%module gd
%{
#include "gd.h"
%}

// Make FILE * work
%typemap(perl5,in) FILE * {
   $target = IoIFP(sv_2io($source));
}

// Grab the gd.h header file
%include "gd.h"

// Extend the interface a little bit
%addmethods gdImage {
gdImage(int w, int h) {
    return gdImageCreate(w,h);
}
~gdImage() {
    gdImageDestroy(self);
}
... etc ...
};

%addmethods gdPoint {
... etc ...
}

// Wrap the fonts (readonly variables)
%readonly
%include "gdfontt.h"
%include "gdfonts.h"
%include "gdfontmb.h"
%include "gdfontl.h"
%include "gdfontg.h"
%readwrite
```

Finally, here is a simple script that uses the module. Aside from a few minor differences, the script is remarkably similar to the first example given in the standard GD module documentation.

```
use gd;

$im = new gdImage(100,100);
$white= $im->colorAllocate(255,255,255);
$black= $im->colorAllocate(0,0,0);
$red= $im->colorAllocate(255,0,0);
$blue= $im->colorAllocate(0,0,255);
$im->transparentcolor($white);
$im->interlaced(1);
$im->rectangle(0,0,99,99,$white);
$im->arc(50,50,95,75,0,360,$blue);
$im->fill(50,50,$red);
open(IMG, ">test.gif");
$im->gif(*IMG);
close(IMG);
```

4 Interface Building Strategies

SWIG simplifies the construction of Perl extensions because it hides Perl-specific implementation details and allows programmers to incorporate C/C++ applications into a Perl environment using familiar ANSI C/C++ syntax rules. In addition, SWIG interfaces are generally specified in a less formal manner than that found in XS or component architectures such as CORBA and COM. As a result, many users are surprised to find out how rapidly they can create Perl interfaces to their C/C++ applications. However, it is a misperception to think that SWIG can magically take an arbitrary C/C++ header file

and instantly turn it into a useful Perl module. This section describes some of the issues and solution strategies for effectively using SWIG.

4.1 Wrapping an Existing Program

Building a Perl interface to an existing application generally involves the following steps :

1. Locate header files and other sources of C declarations.
2. Copy header files to interface files.
3. Edit the interface file and add SWIG directives.
4. Remove or rewrite the application's `main()` function if necessary.
5. Run SWIG, compile, and link into a Perl extension module.

While it is theoretically possible to run SWIG directly on a C header file, this rarely results in the best scripting interface. First, a raw header file may contain problematic declarations that SWIG doesn't understand. Second, it is usually unnecessary to wrap every function and variable in a large library. More often than not, there are internal functions that make little sense to use from Perl. By copying header files to a separate interface file, it is possible to eliminate these functions and clean things up with a little editing.[4] Finally, the underlying application may require a few slight modifications. For example, Perl supplies its own `main()` function so if an application also contains `main()`, it will have to be removed, rewritten, or not linked into the extension module.

4.2 Evolutionary Interface Building

After a Perl interface is first built, its use will expose any problems and limitations. These problems include functions that are awkward to use, poor integration with Perl datatypes, missing functionality, and so forth. To fix these problems, interface files can be enhanced with helper functions, typemaps, exception handlers, and other declarations. Since interfaces are easily regenerated, making such changes is a relatively straightforward process. However, as a result, SWIG interfaces tend to be built in an evolutionary and iterative manner rather than being formally specified in advance.

[4] An alternative approach to copying header files is to modify the header files using conditional compilation to add SWIG directives or to remove unnecessary functions.

4.3 Traps and Pitfalls

Finally, there are a number of subtle problems that sometimes arise when transforming a C/C++ program into a Perl extension module. One of these problems is the issue of implicit execution order dependencies and reentrant functions. From the Perl interpreter, users will be able to execute functions at any time and in any order. However, in many C programs, execution is precisely defined. For example, a precise sequence of function calls might be performed to properly initialize program data. Likewise, it may only be valid to call certain functions once during a single execution. From Perl, it is easy for a user to violate these constraints–resulting in a potential program crash or incorrect behavior. To fix these problems, applications can sometimes be modified by introducing additional state variables. For example, to prevent repeated execution, a function can be modified as follows:

```
void foo() {
   static int called = 0;
   if (called) return;
   ...
   called = 1;
}
```

It is also possible to catch such behavior using exception handlers. For example,

```
%except(perl5) {
   static int called = 0;
   if (called)
         croak("Already executed!\n");
   $function
   called = 1;
}
// List all non-reentrant functions
void foo();
...
// Clear the exception handler
%except(perl5);
```

Another common problem is that of improper memory management. As previously mentioned, SWIG extensions use the same memory management techniques as C. Therefore, careless use may result in memory leaks, dangling pointers, and so forth. A somewhat more obscure memory related problem is caused when a C program overwrites Perl data. This can be caused by a function such as the following:

```
void geterror(char *msg) {
     strcpy(msg,strerror(errno));
}
```

This function copies a string into memory pointed to by `msg`. However, in the wrapper function, the value of

`msg` is really a pointer to data buried deep inside a Perl scalar value. When the function overwrites the value, it corrupts the value of the Perl scalar value and can cause the Perl interpreter to crash with a memory addressing error or obscure run-time error. Again, this sort of problem can usually be fixed with the use of typemaps. For example, it is possible to turn the `msg` parameter into an output value as follows :

```
// Use a temporary array for the result
%typemap(perl5,ignore)
char *msg (char temp[512]) {
    $target = temp;
}
// Copy output into a new Perl scalar
%typemap(perl5,argout) char *msg {
  if (argvi >= items) {
     EXTEND(sp,1);
  }
  $target = sv_newmortal();
  sv_setpv($target,$source);
  argvi++;
}
```

5 Applications

SWIG is currently being used in an increasing variety of applications. This section describes some of the ways in which has been used. A number of advanced SWIG/Perl interfacing techniques such as typemaps and callback functions are also described.

5.1 Electronic CAD

SWIG plays a pivotal role in the development process of BADGER, an electronic computer-aided design system, being developed by Fusion MicroMedia, used in the design of integrated circuits and other electronic components. BADGER is a fully object-oriented, modular, and highly extensible system, running under various flavors of the UNIX operating system as well as Windows-NT.

The core components in BADGER are constructed in C++ and are delivered as a set of shared (dynamically loaded) libraries. The libraries are *not* directly linked into an executable program. Instead, each library comes with an extension language (EL) interface that is generated by SWIG, allowing the library to be used within a Perl program.[5] The combination of a powerful EL and well-tuned, application-specific software results in a system that is potent, flexible, and easy to use.

For the most part, SWIG is used in a "normal" fashion: a description of the classes contained within a library is presented to SWIG, and it generates an EL interface that allows the code within that library to be accessed from an EL. There are two interesting facets to the use of SWIG within BADGER: the use of "smart references," and the use of callbacks from C++ to the EL,

5.1.1 Smart References

Suppose a Perl program calls a function defined by BADGER (and wrapped with SWIG) in order to create and return some object. Any Perl variable used to refer to that object really holds a *handle* to the object, implemented as a blessed reference containing the object's type and its memory address. Although the implementation is a bit more involved, the handle, in effect, acts like a pointer in C. Now, suppose another function within BADGER is called that causes the original object to be destroyed. Severe problems will occur if the Perl variable is supplied to another BADGER function, because the variable refers to a non-existent object. The reason for the difficulty is that the extension language expects to have control over the lifetime of the object, but the external system (BADGER) cannot meet this expectation.

It is possible to design BADGER so that the extension language has complete control over the lifetime of all the objects within the system. Unfortunately, this approach results in a system that is too closely tied to the implementation of a particular language, and adding a new extension language to the mix is difficult. An alternate solution that is simple to implement and is portable, is to introduce "smart references" (also called *proxies*) into the design [5, pg. 207]. In effect, a smart reference is an object that has the same set of operations as a "real" object, but the smart reference's implementation consists solely of a single pointer to a "real" object of the appropriate type.

The extension language interfaces within BADGER have been crafted so that the extension language manipulates smart references and that the lifetime of a smart reference is completely under the control of the extension language. Under most circumstances, the extension language performs an operation on the smart reference, and the smart reference then attempts to transfer the operation to the real object. If the real object has been destroyed then the smart reference will have been invalidated (it points to `nil`). In this case, the operation is aborted and, if possible, an exception is raised in the extension language. BADGER contains the necessary machinery to invalidate any smart references that point to an object being destroyed.

Modern C++ compilers, with their support for templates, run-time type identification, and so forth, provide the means to automatically construct smart reference classes. For a variety of reasons, we are not able to

[5] For now, Perl is the only supported extension language. Tcl and Java will be supported in the future.

always utilize modern compilers. Hence, we have created the implementations of the smart references manually, which is a tedious process. Fortunately, this task can be mostly automated by creating our own code generator as part of SWIG. This is a simple matter, as SWIG is a modular software system.

5.1.2 Callbacks

The extension language interface produced by SWIG allows functions defined in the external system to be called from within an extension language. Unfortunately, the interface produced by SWIG does not support the calling of extension language functions within C, C++, or Objective-C. The ability to invoke functions bidirectionally is needed by BADGER, so support for callbacks from C++ to Perl has been developed.[6] The basic approach is this:

- Define a function.
- Register the function.
- Perform some operation that causes the registered function to be invoked.

To make this work, BADGER provides an abstract base class in C++ called `Trigger`, so called because a function associated with objects of this class is invoked when an event of some kind occurs. BADGER also provides the machinery to associate `Trigger` objects with an event name and with one or more objects internal to the system. When an internal object "receives" an event, it examines the set of registered functions looking for a match. If a match is found then the `Trigger` object is invoked, and the name of the event and the object that received the event are supplied as arguments.

BADGER provides a number of classes derived from `Trigger` that specialize its behavior for certain extension languages, for C++, or for an object request broker. For example, the `Perl5Trigger` class is derived from `Trigger` and it specializes its base class by storing a pointer to a Perl function reference (an `SV*`), and by providing the machinery needed to invoke that Perl function.

For example, consider the following Perl fragment:

```
sub MyFcn {
  my $EventName = shift;
  my $Object = shift;
  # ... rest of function here.
}
my $Object = BadgerFunction(....);
my $Name = "Can't find file";
Badger::RegisterByObject($Name,
                $Object, \&MyFcn);
$Object->ReadFile("Bogus name");
```

The `MyFcn()` Perl function is the callback (trigger) function, and it is registered with `$Object` using the event name called "`Can't find file`". Now, suppose that the `$Object->ReadFile()` operation fails. Internally, BADGER will note the failure, determine the appropriate event name, attempt to find a Trigger object associated with that event, and if found, will "invoke the Trigger" by calling the appropriate member function. For the example above, this means that the `MyFcn()` function will be called with `$Object` and "`Can't find file`" supplied as arguments. The function may require more information such as the file name (that could not be opened), and it might find this information by "pulling" data from the external library using the functions wrapped by SWIG.

The `RegisterByObject()` function is responsible for creating an object of the `Perl5Trigger` class, and for creating the association between the `Perl5Trigger`, the event name, and the object receiving the event. There is a bit of typemap trickery involved when intercepting the arguments from Perl:

```
%typemap(perl5,in) SV* pFcn {
  if (!SvROK($source))
    croak("Expected a reference.\n");
  $target = SvRV($source);
}
void
RegisterByObject(
   const char* pcEventName,
   Ref* pRef, SV* pFcn);
```

The final portion of the system left to describe is the implementation of the `Perl5Trigger::Invoke()` member function, which is responsible for calling the Perl function from the C++ side of the world. The implementation of this, taken nearly verbatim from the Advanced Perl Programming book ([1, pg. 353]), looks like this:

```
bool
Perl5Trigger::
Invoke(const char* pcEventName,
       void* pObject,
       const char* pcTypeName) {
  dSP;
  ENTER;
  SAVETMPS;
  PUSHMARK(sp);
  SV* pSV = sv_newmortal();
  sv_setpv(pSV, (char*) pcEventName);
  XPUSHs(pSV);
  pSV = sv_newmortal();
```

[6] For now, callbacks only work with Perl. Support for callbacks with Tcl and Java will be added later.

```
  sv_setref_pv(pSV,(char*)pcTypeName,
               pObject);
  XPUSHs(pSV);
  pSV = sv_newmortal();
  sv_setpv(pSV,(char*)pcTypeName);
  XPUSHs(pSV);
  PUTBACK;
  int n = perl_call_sv(
          this->pPerl5Fcn,
          G_SCALAR);
  SPAGAIN;
  if (n == 1)
    n = POPi;
  PUTBACK;
  FREETMPS;
  LEAVE;
  return n == 0 ? false : true;
}
```

5.1.3 Benefits And Limitations

The benefits that SWIG provides to BADGER are enormous:

- Not counting custom code (*e.g.*, language-specific callbacks), an extension language interface can be developed in a day, compared with weeks for a hand-crafted approach.
- SWIG supports the use of multiple extension languages with ease.
- The resulting solution is flexible, and the results can be tailored to meet the needs of complex applications (*e.g.*, callbacks, smart references, and so on).

SWIG does have limitations, but so far, none of these limitations has proven to be a real impediment. It also appears that most of these limitations will be eradicated, once SWIG has its own extension language interface (see Section 7).

5.2 TCAP and SCCP from HP OpenCall

One of the well known pitfalls of systematic library testing is the creation of a huge number of small C programs–each designed to perform a single test. More often than not, these C programs have a lot of common code that is copied from one test case to the other. Testing is further complicated by the tedious process of editing, compiling, and executing each of these programs.

To solve this problem, SWIG can be used to incorporate libraries into Perl extension modules where test cases can be implemented as Perl scripts. As a result, the compile-execute cycle is no longer a problem and Perl scripts can be used to implement common parts of various test cases.

This section describes the integration of Perl with an API that is part of a HP OpenCall telecom product developed at HP Grenoble. The API provides access to the TCAP and SCCP layers of the SS7 protocol and consists of about 20 function and 60 structure declarations. Furthermore, most function parameters are pointers to deeply nested structures such as follows:

```
typedef enum {
  ...
} tc_address_nature;

typedef struct  {
  ...
  tc_address_nature  nature;
  ...
} tc_global_title;

typedef struct tc_address_struct {
  ...
  tc_global_title    gt;
  ...
} tc_address;
```

From a Perl users' point of view, the functionality offered by the SWIG generated module must be not be very different from the underlying C API. Otherwise, test writers may be confused by the Perl API and testing will be unnecessarily complicated. Fortunately, SWIG addresses this problem because Perl interfaces are specified using C syntax and the resulting interface closely resembles the original API.

5.2.1 Creating the SWIG Interface

To wrap the C API, there were three choices: copy and modify the header files into a SWIG interface file, feed the header files directly to SWIG, or write an interface file that includes some parts of the header files. The first choice requires the duplication of C definitions–a task that is difficult to manage as the API evolves (since it is hard to maintain consistency between the interface file and header files). The second choice may work if the header files are written in a very clean way. However, it can break down if header files are too complicated. Therefore, a mix of header files and interfaces was utilized.

As part of the interface building process, header files were to be included directly into interface files. This is easily done using the `%include` directive, but a number of problematic nested structure declarations had to be fixed. For example,

```
struct tcStat {
  ...
  union {
    ...
```

```
    struct stat_p_abort {
      int value;
      tc_p_abort_cause  p_abort;
    } abort;
    ...
  } p;
} tc_stat;
```

To make this structure more manageable in SWIG, it can be split into smaller pieces and rewritten as follows:

```
typedef struct {
  int value;
  tc_p_abort_cause  p_abort;
} tc_stat_abort;

struct TcStat {
  ...
  tc_stat_abort  abort;
  ...
};
```

Such changes have no impact on user code, but they simplify the use of SWIG.

In addition to splitting, a number of structures in the header files were to be hidden from the SWIG compiler. While this could be done using a simple `#ifndef SWIG` in the code, this could potentially result in a huge customer problem if they also defined a `SWIG` macro in their compilation process. Therefore, conditional compilation was implemented using some clever C comments that were parsed by vpp (See the Text::Vpp module) during the build of the SWIG interface. For example,

```
/*
HP reserved comment
@if not $_hp_reserved_t
*/
typedef struct {
  int           length;
  unsigned char datas[MAX_ABORT_LEN];
} tc_u_abort;
/*
@endif
 */
```

5.2.2 Shadow Classes

By default, SWIG converts structure definitions into accessor functions such as

```
tc_global_title *
tc_address_gt_get(tc_address *);

tc_address_nature
tc_global_title_nature_set(
   tc_global_title *t,
   tc_address_nature val);
```

Unfortunately, using such functions is somewhat unfriendly from Perl. For example, to set a single value, it would be necessary to write the following:

```
$param = new_tc_address();
tc_global_title_nature_set(
    tc_address_gt_get($param),
    $value);
```

Fortunately, shadow classes solve this problem by providing object-oriented access to the underlying C structures. As a result, it is possible to rewrite the above Perl code as follows:

```
$parm = new tc_address;
$param->{gt}{nature} = $value;
```

Needless to say, this approach is much easier for users to grasp.

5.2.3 Customization With Typemaps

To improve the Perl interface, a number of typemaps were defined for various parts of the interface. One use of typemaps was in structures such as the following:

```
typedef struct {
    ...
    tc_u_abort  abort_reason;
    ...
} tc_dialog_portion;
```

Since `tc_u_abort` is defined by the structure shown earlier, SWIG normally tries to manipulate it through pointers. However, a typemap can be defined to change this behavior. In particular, it was decided that testers should be able to set and get this value using BCD encoded strings such as follows:

```
my $dialog = new tc_dialog_portion;
$dialog->{abort_reason} = '0f456A';

# Or
print "User abort reason is \
$dialog->{abort_reason} \n";
```

To do this, a typemap for converting BCD Perl strings into an appropriate byte sequence were developed. In addition, the typemap performs a few sanity checks to prevent invalid values.

```
%typemap (perl5,in) tc_u_abort *
($basetype temp)
{
  int i;
  STRLEN len;
  short tmp;
  char *str;
```

```
  $target = &temp;

  /* convert  scalar to char* */
  str = SvPV($source,len);
  /* check if  even # of char */
  if ( (len % 2) != 0 ) {
    croak("Uneven # of char");
  }
  /* set length field */
  $target->length = (len/2);
  if ((len/2) > (sizeof($basetype)-1))
  {
    croak("Too many bytes\n");
  }
  for (i=0; i < $target->length; i++)
  {
    if (sscanf(str,"%2hx",&tmp) != 1)
      croak("sscanf failed on %s, \
             is it hexa ?\n",str);
      $target->datas[i] = tmp;
      str+=2;
  }
}
```

To return the byte buffer back to Perl as a string, a somewhat simpler typemap is used:

```
%typemap (perl5,out) tc_u_abort *
{
  int i;
  $target=newSVpvf("%x",
              $source->datas[0]);

  for (i=1; i<$source->length; i++) {
    sv_catpvf($target,"%x",
              $source->datas[i]);
  }
  argvi++;
}
```

SWIG typemaps were also used to fix a few other functions. For example, some functions required an address parameter encoded as a two-element array. By default, SWIG wraps this parameter as a pointer, but this leaves the Perl writer with the painful tasks of creating and filling a C array with sensible values using the SWIG pointer library or helper functions. Fortunately, with typemaps, it was possible to create and set this parameter using Perl hashes as follows:

```
# $address is an ordinary perl hash
# $address will be used as an array
$address->{pc} = 10;
$address->{ssn}= 12;
...
SCCP_oamcmd($cnxId, $time, undef,
  $address, $command, $cmd_parms);
```

The typemap implementing this behavior is as follows:

```
%typemap (perl5,in) SccpOamAddress*
{
  HV* passedHash;
  SV** valuePP;
  SccpOamAddress tempAddress;
  if (!SvOK($source)) {
      /* we were passed undef */
      tempAddress[0] = 0;
      tempAddress[1] = 0;
  } else {
    if (!SvROK($source))
      croak("Not a reference\n");
    if (SvTYPE(SvRV($source))!=SVt_PVHV)
      croak("Not a hash ref\n");

    passedHash=(HV*)SvRV($source);
    valuePP =
       hv_fetch(passedHash,"ssn",3,0);

    if (*valuePP == NULL)
      croak("Missing 'ssn' key\n");
    tempAddress[1] = SvIV(*valuePP);
    valuePP =
       hv_fetch(passedHash,"pc",2,0);
    if (*valuePP == NULL)
      croak("Missing 'pc' key\n");
    tempAddress[0] = SvIV(*valuePP);
  }
  $target = &tempAddress;
}

/* SccpOamAddress is returned as
{'ssn'=>ssn_value, 'pc'=>pc_value} */
%typemap (perl5,out) SccpOamAddress*
{
  HV* passedHash;
  SV* theSsn;
  SV* thePc;

  thePc  = newSViv((*$source)[0]);
  theSsn = newSViv((*$source)[1]);
  passedHash = newHV();
  hv_store(passedHash,"ssn",3,theSsn,0);
  hv_store(passedHash,"pc",2,thePc,0);
  $target = newRV_noinc((SV*)passedHash);
  argvi ++;
}
```

5.2.4 Statistics

Table 1 shows the amount of code associated with `.i` files and header files as well as the amount of code generated by SWIG (`.C` and `.pm` files). While it was necessary to write a few `.i` files, the size of these files is small in comparsion to the generated output files.

Table 1: TCAP and SCPP Modules

	.i files	.h files	.C files	.pm files
TCAP	434	977	16098	3561
SCPP	364	494	13060	2246

5.2.5 Results

Overall, SWIG saved time when providing Perl access to the TCAP and SCCP libraries. While it took some time and hard work to write the typemaps, the SWIG approach has several advantages compared to XS or the pure C approach:

- The interface files are quite short so if they are well documented, a new SWIG user should not have any major problems maintaining them.
- A new version of the API is wrapped with a 'make' command, so there is no need to edit any file. In most cases the interface files can remain unmodified, provided there are no weird constructs introduced in the new version of the API.
- New comments added in the header files will be automatically added in the documentation files generated by SWIG.
- If necessary, new helper functions may be added in the .i files without impacting other parts of the code or typemaps. This allows a new user to do it without reading the whole SWIG manual.
- Typemaps that deal with basic types or simple structures are reusable and can be used with other APIs.

For those who are considering SWIG's advanced features, the learning curve is a little steep at first, but the rewards are great because SWIG advanced features will enable you to provide an improved interface to the Perl user.

6 Limitations

Currently, SWIG is being used by hundreds of users in conjunction with a variety of applications. However, the current implementation of SWIG has a number of limitations. Some of these limitations are due to the fact that SWIG is not a full C/C++ parser. In particular, the following features are not currently supported:

- Variable length arguments (...)
- Pointers to functions.
- Templates.
- Overloaded functions and operators.
- C++ Namespaces.
- Nested class definitions.

When these features appear in a SWIG input file, a syntax error or warning message is generated. To eliminate these warnings, problematic declarations can either be removed from the interface, hidden with conditional compilation, or wrapped using helper functions and other SWIG directives.

A closely related problem is that certain C/C++ programs are not easily scripted. For example, programs that make extensive use of advanced C++ features such as templates, smart pointers, and overloaded operators can be extremely troublesome to incorporate into Perl. This is especially the case for C++ programs that override the standard behavior of pointers and deferencing operations—operations that are used extensively by SWIG generated wrapper code.

In addition, SWIG does not provide quite as much flexibility as `xsubpp` and other Perl specific extension building tools. In order to be general purpose, SWIG hides many of the internal implementation details of each scripting language. As a result, it can be difficult to accomplish certain tasks. For example, one such situation is the handling of functions where arguments are implicitly related to each other as follows:

```
void foo(char *str, int len) {
   // str = string data
   // len = length of string data
   ...
}
```

Ideally, it might be desirable to pass a single Perl string to such a function and have it expanded into a data and length component. Unfortunately, SWIG has no way to know that the arguments are related to each other in this manner. Furthermore, the current typemap mechanism only applies to single arguments so it can not be used to combine arguments in this manner. XS, on the other hand, is more closely tied to the Perl interpreter and consequently provides more power in the way that arguments can be converted and passed to C functions.

Finally, SWIG is still somewhat immature with respect to its overall integration with Perl. For example, SWIG does not fully support Perl's package and module naming system. In other words, SWIG can create a module "Foo", but can't create a module "Foo::Bar." Likewise, SWIG does not currently utilize MakeMaker and other utilities (although users have successfully used

SWIG with such tools). In addition, some users have reported occasional problems when SWIG modules are used with the Perl debugger and other tools.

7 Future Directions

Future development of SWIG is focused on three primary areas. First, improved parsing and support for more advanced C++ are being added. These additions include support for overloaded functions and C++ namespaces. Limited support for wrapping C++ templates may also be added. Second, SWIG's code generation abilities are being improved. Additions include more flexible typemaps and better access to scripting-language specific features. Finally, an extension API is being added to the SWIG compiler. This API will allow various parts of the SWIG compiler such as the preprocessor, parser, and code generators to be accessed through a scripting language interface. In fact, this interface will even allow new parsers and code generators to be implemented entirely in Perl.

8 Acknowledgments

SWIG would not be possible without the feedback and contributions of its users. While it is impossible to acknowledge everyone individually, a number of people have been instrumental in promoting and improving SWIG's Perl support. In particular, Gary Holt provided many of the ideas used in the shadow class mechanism. We would also like to thank John Buckman, Scott Bolte, and Sriram Srinivasan, for their support of SWIG. We also thank the University of Utah and Los Alamos National Laboratory for their continued support.

9 Availability

SWIG is freely available on CPAN at

`www.perl.com/CPAN/authors/Dave_Beazley`

Additional information is also available on the SWIG homepage at `www.swig.org`. An active mailing list of several hundred subscribers is also available.

References

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

[2] Scott Bolte. SWIG. *The Perl Journal*, 2(4):26–31, Winter 1997.

[3] D.M. Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal*, (282):30–36, Feb 1998.

[4] D.M. Beazley. SWIG users manual. Technical Report UUCS-98-012, University of Utah, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.