Exception Handling 7

In some cases, it is desirable to catch errors that occur in C functions and propagate them up to the scripting language interface (ie. raise an exception). By default, SWIG does nothing, but you can create a user-definable exception handler using the `%except` directive.

The %except directive

The `%except` directive allows you to define an exception handler. It works something like this :

```
%except(python) {
        try {
        $function
        }
        catch (RangeError) {
               PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
               return NULL;
        }
}
```

As an argument, you need to specify the target language. The exception handling C/C++ code is then enclosed in braces. The symbol `$function` is replaced with the real C/C++ function call that SWIG would be ordinarily make in the wrapper code. The C code you specify inside the `%except` directive can be anything you like including custom C code and C++ exceptions.

To delete an exception handler, simply use the `%except` directive with no code. For example :

```
%except(python);             // Deletes any previously defined handler
```

Exceptions can be redefined as necessary. The scope of an exception handler is from the point of definition to the end of the file, the definition of a new exception handler, or until the handler is deleted.

Handling exceptions in C code

C has no formal mechanism for handling exceptions so there are many possibilities. The first approach is to simply provide some functions for setting and checking an error code. For example :

```
/* File : except.c */
```

```
static char error_message[256];
static int error_status = 0;

void throw_exception(char *msg) {
       strncpy(error_message,msg,256);
       error_status = 1;
}

void clear_exception() {
       error_status = 0;
}
char *check_exception() {
       if (error_status) return error_message;
       else return NULL;
}
```

To work, functions will need to explicitly call `throw_exception()` to indicate an error occurred. For example :

```
double inv(double x) {
       if (x != 0) return 1.0/x;
       else {
              throw_exception("Division by zero");
              return 0;
       }
}
```

To catch the exception, you can write a simple exception handler such as the following (shown for Perl5) :

```
%except(perl5) {
       char *err;
       clear_exception();
       $function
       if ((err = check_exception())) {
              croak(err);
       }
}
```

Now, when an error occurs, it will be translated into a Perl error. The downside to this approach is that it isn't particularly clean and it assumes that your C code is a willing participant in generating error messages. (This isn't going to magically add exceptions to a code that doesn't have them).

Exception handling with longjmp()

Exception handling can also be added to C code using the `<setjmp.h>` library. This usually isn't documented very well (at least not in any of my C books). In any case, here's one implementation that uses the C preprocessor :

```
/* File : except.c
   Just the declaration of a few global variables we're going to use */

#include <setjmp.h>
jmp_buf exception_buffer;
```

```
int exception_status;

/* File : except.h */
#include <setjmp.h>
extern jmp_buf exception_buffer;
extern int exception_status;

#define try if ((exception_status = setjmp(exception_buffer)) == 0)
#define catch(val) else if (exception_status == val)
#define throw(val) longjmp(exception_buffer,val)
#define finally else

/* Exception codes */

#define RangeError     1
#define DivisionByZero 2
#define OutOfMemory    3
```

Now, within a C program, you can do the following :

```
double inv(double x) {
       if (x) return 1.0/x;
       else {throw(DivisionByZero);
}
```

Finally, to create a SWIG exception handler, write the following :

```
%{
#include "except.h"
%}

%except(perl5) {
       try {
              $function
       } catch(RangeError) {
              croak("Range Error");
       } catch(DivisionByZero) {
              croak("Division by zero");
       } catch(OutOfMemory) {
              croak("Out of memory");
       } finally {
              croak("Unknown exception");
       }
}
```

At this point, you're saying this sure looks alot like C++ and you'd be right (C++ exceptions are often implemented in a similar manner). As always, the usual disclaimers apply--your mileage may vary.

Handling C++ exceptions

Handling C++ exceptions is almost completely trivial (well, all except for the actual C++ part). A typical SWIG exception handler will look like this :

```
%except(perl5) {
```

```
        try {
                $function
        } catch(RangeError) {
                croak("Range Error");
        } catch(DivisionByZero) {
                croak("Division by zero");
        } catch(OutOfMemory) {
                croak("Out of memory");
        } catch(...) {
                croak("Unknown exception");
        }
}
```

The exception types need to be declared as classes elsewhere, possibly in a header file :

```
class RangeError {};
class DivisionByZero {};
class OutOfMemory {};
```

Newer versions of the SWIG parser should ignore exceptions specified in function declarations. For example :

```
double inv(double) throw(DivisionByZero);
```

Defining different exception handlers

By default, the `%except` directive creates an exception handler that is used for all wrapper functions that follow it. Creating one universal exception handler for all functions may be unwieldy and promote excessive code bloat since the handler will be inlined into each wrapper function created. For this reason, the exception handler can be redefined at any point in an interface file. Thus, a more efficient use of exception handling may work like this :

```
%except(python) {
        ... your exception handler ...
}
/* Define critical operations that can throw exceptions here */

%except(python);         // Clear the exception handler

/* Define non-critical operations that don't throw exceptions */
```

Applying exception handlers to specific datatypes.

An alternative approach to using the `%except` directive is to use the "except" typemap. This allows you to attach an error handler to specific datatypes and function name. The typemap is applied to the return value of a function. For example :

```
%typemap(python,except) void * {
        $function
        if (!$source) {
                PyExc_SetString(PyExc_MemoryError,"Out of memory in $name");
                return NULL;
        }
```

```
}

void *malloc(int size);
```

When applied, this will automatically check the return value of `malloc()` and raise an exception if it's invalid. For example :

```
Python 1.4 (Jan 16 1997)  [GCC 2.7.2]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from example import *
>>> a = malloc(2048)
>>> b = malloc(1500000000)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
MemoryError: Out of memory in malloc
>>>
```

Since typemaps can be named, you can define an exception handler for a specific function as follows :

```
%typemap(python,except) void *malloc {
        ...
}
```

This will only be applied to the `malloc()` function returning `void *`. While you probably wouldn't want to write a different exception handler for every function, it is possible to have a high degree of control if you need it. When typemaps are used, they override any exception handler defined with `%except`.

Using The SWIG exception library

The `exception.i` library file provides support for creating language independent exceptions in your interfaces. To use it, simply put an "`%include exception.i`" in your interface file. This creates a function `SWIG_exception()` that can be used to raise scripting language exceptions in a portable manner. For example :

```
// Language independent exception handler
%include exception.i

%except {
        try {
                $function
        } catch(RangeError) {
                SWIG_exception(SWIG_ValueError, "Range Error");
        } catch(DivisionByZero) {
                SWIG_exception(SWIG_DivisionByZero, "Division by zero");
        } catch(OutOfMemory) {
                SWIG_exception(SWIG_MemoryError, "Out of memory");
        } catch(...) {
                SWIG_exception(SWIG_RuntimeError,"Unknown exception");
        }
}
```

As arguments, `SWIG_exception()` takes an error type code (an integer) and an error message

string. The currently supported error types are :

```
SWIG_MemoryError
SWIG_IOError
SWIG_RuntimeError
SWIG_IndexError
SWIG_TypeError
SWIG_DivisionByZero
SWIG_OverflowError
SWIG_SyntaxError
SWIG_ValueError
SWIG_SystemError
SWIG_UnknownError
```

Since the `SWIG_exception()` function is defined at the C-level it can be used elsewhere in SWIG. This includes typemaps and helper functions. The exception library provides a language-independent exception handling mechanism, so many of SWIG's library files now rely upon the library as well.

Debugging and other interesting uses for %except

Since the `%except` directive allows you to encapsulate the actual C function call, it can also be used for debugging and tracing operations. For example :

```
%except(tcl) {
        printf("Entering function : $name\n");
        $function
        printf("Leaving function : $name\n");
}
```

allows you to follow the function calls in order to see where an application might be crashing.

Exception handlers can also be chained. For example :

```
%except(tcl) {
        printf("Entering function : $name\n");
        $except
        printf("Leaving function : $name\n");
}
```

Any previously defined exception handler will be inserted in place of the "`$except`" symbol. As a result, you can attach debugging code to existing handlers if necessary. However, it should be noted that this must be done before any C/C++ declarations are made (as exception handlers are applied immediately to all functions that follow them).

More Examples

By now, you know most of the exception basics. See the SWIG Examples directory for more examples and ideas. Further chapters show how to generate exceptions in specific scripting languages.