The SWIG Preprocessor 8

Introduction

Before an interface file is processed by SWIG, it is first processed by the SWIG preprocessor[1]. The SWIG preprocessor closely mimics its C counterpart, in that it handles conditional compilation, file inclusion, and macro expansion. However, it differs from the normal C preprocessor in that it leaves comments intact (for the documentation system), allows preprocessing to be disabled, and doesn't introduce quite as much clutter as you would normally get running a header file through the C preprocessor.

Preprocessing steps

The following steps are performed by the SWIG preprocessor :

- Trigraph sequences are replaced by the equivalents. This is disabled by default, but can be enabled by running SWIG with the `-trigraphs` option.
- Conditional compilation.
- File inclusion with `%include`, `%extern`, and `%import` directives.
- Macro expansion.

While preprocesing closely mimics that of C, the following C preprocessor directives are ignored:

- `#include`
- `#line`
- `#error`
- `#pragma`
- `#import`

To find out what the preprocessor is doing, use the command '`swig -E`'. This will output the results of preprocessing to stdout.

Scope of preprocessing

Preprocessing is applied to all portions of an interface file except to

- Code included inside a `%{,%}` block.
- C preprocessor directives preceded by a %.

1. The SWIG preprocessor is new in SWIG 1.2.

This first rule is necessary because %{,%} is used to insert supporting C code into SWIG's output. C preprocessor directives appearing here are used during the compilation of the SWIG generated wrapper code, not during the interface generation process. The second rule can be used to force a directive to pass through the preprocessor unchanged. This is sometimes necessary in code fragments such as the following :

```
%addmethod Foo {
       void bar(double x) {
%#ifdef DEBUG
              printf("Value = %g\n", x);
%#endif
       }
};
```

Normally this code is processed by the SWIG preprocessor and all preprocessor directives stripped. However, in this case, the output after preprocessing will be :

```
%addmethod Foo {
       void bar(double x) {
#ifdef DEBUG
              printf("Value = %g\n", x);
#endif
       }
};
```

The C preprocessor directives will now appear in the wrapper code generated by SWIG.

Defining macros

Macros can be defined using the `#define` directive. The rules are the same as for the C preprocessor except that simple constants such as

```
#define PI  3.14159
```

are passed through to the SWIG parser where they are turned into scripting language constants. Macros with arguments can also be defined such as :

```
#define ADD(a,b)  a+b
```

These macros will be expanded by the SWIG preprocessor when appropriate, but they are ignored during the creation of wrapper code. When defining macros, the # and ## operators may be used to create strings and perform concatenation. For example :

```
#define STRING(a)      #a
```

will generate the string "a", and

```
#define CONCAT(a,b)     a ## b
```

will generate the symbol `ab`.

Macros can be undefined using the `#undef` directive.

Conditional compilation

Conditional compilation can be performed using the `#ifdef`, `#ifndef`, `#else`, `#endif`, `#if`, and `#elif` directives in exactly the same manner as in the C preprocessor. For the `#if` and `#elif` directives, a constant integral expression can be evaluated to determine truth. The `defined()` macro can also be used to determine if a symbol has been defined. For example :

```
#if ((MAJOR_VERSION == 2) && (MINOR_VERSION == 1)) || defined(__STDC__)
...
#endif
```

Of course, I'm assuming that you're intimately familar with the C preprocessor already....

Predefined symbols

The following symbols are defined by SWIG :

```
__STDC__          Always defined (indicates that SWIG supports ANSI)
__cplusplus       Defined when the -c++ option has been given.
__FILE__          Name of the SWIG interface file being processed.
__DATE__          A string literal containing the date of compilation.
__TIME__          A string literal containing the time of compilation.
SWIG              Always defined when SWIG is processing a file
SWIGTCL           Defined when using Tcl
SWIGTCL8          Defined when using Tcl8.0
SWIGPERL          Defined when using Perl
SWIGPERL4         Defined when using Perl4
SWIGPERL5         Defined when using Perl5
SWIGPYTHON        Defined when using Python
SWIGGUILE         Defined when using Guile
SWIGWIN           Defined when running SWIG under Windows
SWIGMAC           Defined when running SWIG on the Macintosh
```

Common uses for macros

Since SWIG is primarily concerned with C declarations, not C code, macros may have a much more limited use in SWIG than might be the case in a full C program. However, here are a few useful tricks that can be used.

Processing complicated header files

With macros, SWIG can process header files such as the following :

```
#ifdef __STDC__
#define _ANSI_ARGS_(a)   a
#endif
#define EXTERN extern
...
EXTERN void foo _ANSI_ARGS((double x, double y));
```

Wrapping C++ templates

The problem with templates is that SWIG is only able to wrap a specific instantiation of a template. One useful trick for doing this is described in chapter 3, but we can extend it with a macro as follows :

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

// Now define a macro that mirrors the template class definition
#define LIST_TEMPLATE(name,type)   \
%{\
typedef List<type> name; \
%}\
class name {\
public:\
     name();\
    ~name();\
     void append(type); \
     int length(); \
     type get(int n); \
};

// Now create wrappers around a bunch of different lists
LIST_TEMPLATE(IntList,int)
LIST_TEMPLATE(DoubleList,double)
LIST_TEMPLATE(VectorList, Vector *)
LIST_TEMPLATE(StringList,char *)
```

Certainly not the most elegant approach, but remarkably simple considering that SWIG does not yet have full template support.

Cool tricks and avoidance of code duplication

Macros can also be used for avoiding code duplication and performing other cool operations. For example, here's a method for producing array helper functions for any datatype :

```
#define MAKE_NAME(a,b)    a ## b
#define ARRAY_HELP(name,type) \
%inline %{  \
type *MAKE_NAME(new_,name)(int size) { \
     return (type *) malloc(size*sizeof(type)); \
} \
void MAKE_NAME(delete_,name)(type *obj) {  \
     free((char *) obj); \
} \
\
type *MAKE_NAME(get_,name)(type *obj, int index) { \
     return obj+index; \
}\
void MAKE_NAME(set_,name)(type *obj, int index, type value) { \
     obj[index] = value; \
}\
%}

// Now provide a bunch of helper functions for arrays
```

```
ARRAY_HELP(double,double)
ARRAY_HELP(int,int)
ARRAY_HELP(VectorPtr, Vector *)
```

Similar tricks can also be performed when defining typemaps and exception handlers.

The obfuscated SWIG code contest?

While I will probably live to regret adding macro support, it provides a very powerful mechanism for developers of SWIG library files, working with large nasty systems, creating virtually unintelligible interface files, or simply blowing your whole leg off. Have fun!