MWrap User Guide

D. Bindel

March 4, 2009

1 Introduction

MWrap is an interface generation system in the spirit of SWIG or matwrap. From a set of augmented MATLAB script files, MWrap will generate a MEX gateway to desired C/C++ function calls and MATLAB function files to access that gateway. The details of converting to and from MATLAB's data structures, and of allocating and freeing temporary storage, are hidden from the user.

MWrap files look like MATLAB scripts with specially formatted extra lines that describe calls to C/C++. For example, the following code is converted into a call to the C `strncat` function:

```
$ #include <string.h>
function foobar;

  s1 = 'foo';
  s2 = 'bar';
  # strncat(inout cstring[128] s1, cstring s2, int 127);
  fprintf('Should be foobar: %s\n', s1);
```

There are two structured comments in this file. The line beginning with `$` tells MWrap that this line is C/C++ support code, and the line beginning with `#` describes a call to the C `strncat` function. The call description includes information about the C type to be used for the input arguments, and also describes which parameters are used for input and for output.

From this input file, MWrap produces two output files: a C file which will be compiled with the MATLAB `mex` script, and a MATLAB script which calls that MEX file. If the above file is `foobar.mw`, for example, we would generate an interface file `foobar.m` and a MEX script `fbmex.mex` with the lines

```
mwrap -mex fbmex -m foobar.m foobar.mw
mwrap -mex fbmex -c fbmex.c  foobar.mw
mex fbmex.c
```

At this point, we can run the `foobar` script from the MATLAB prompt:

```
>> foobar
Should be foobar: foobar
```

Versions of GNU Octave released after mid-2006 support much of MATLAB's MEX interface. Simple MWrap-generated code should work with GNU Octave. To compile for GNU Octave, use `mkoctfile --mex` instead of invoking `mex`. GNU Octave does not implement MATLAB's object model, so code that uses MATLAB's object-oriented facilities will not function with GNU Octave.

2 MWrap command line

The `mwrap` command line has the following form:

```
mwrap [-mex outputmex] [-m output.m] [-c outputmex.c] [-mb]
      [-list] [-catch] [-c99complex] [-cppcomplex]
      infile1 infile2 ...
```

where

- `-mex` specifies the name of the MEX function that the generated MATLAB functions will call. This name will generally be the same as the prefix for the C/C++ output file name.
- `-m` specifies the name of the MATLAB script to be generated.
- `-c` specifies the name of the C MEX file to be generated. The MEX file may contain stubs corresponding to several different generated MATLAB files.
- `-mb` tells MWrap to redirect MATLAB function output to files named in the input. In this mode, the processor will change MATLAB function output files whenever it encounters a line beginning with `@`. If `@` occurs alone on a line, MATLAB output will be turned off; if the line begins with `@function`, the line will be treated as the first line of a function, and the m-file name will be deduced from the function name; and otherwise, the characters after `@` (up to the next set of white space) will be treated as a filename, and MWrap will try to write to that file.
- `-list` tells MWrap to print to the standard output the names of all files that would be generated from redirect output by the `-mb` flag.
- `-catch` tells MWrap to surround library calls in try/catch blocks in order to intercept C++ exceptions.
- `-c99complex` tells MWrap to use the C99 complex floating point types as the default `dcomplex` and `fcomplex` types.
- `-cppcomplex` tells MWrap to use the C++ complex floating point types as the default `dcomplex` and `fcomplex` types.

3 Interface syntax

3.1 Input line types

MWrap recognizes six types of lines, based on the first non-space characters at the start of the line:

- C support lines begin with `$`. These lines are copied into the C code that makes up the MEX file. Typically, such support lines are used to include any necessary header files; they can also be used to embed short functions.
- Blocks of C support can be opened by the line `$[` and closed by the line `$]`. Like lines beginning with `$`, lines that fall between the opening and closing markers are copied into the C code that makes up the MEX file.
- C call lines begin with `#`. These lines are parsed in order to generate an interface function as part of the MEX file and a MATLAB call line to invoke that interface function. C call lines can refer to variables declared in the local MATLAB environment.
- Input redirection lines (include lines) begin with `@include`. The input file name should not be quoted in any way.
- Output redirection lines begin with `@`. Output redirection is used to specify several generated MATLAB scripts with a single input file.
- Comment lines begin with `//`. Comment lines are not included in any output file.
- All other lines are treated as ordinary MATLAB code, and are passed through to a MATLAB output file without further processing.

3.2 C call syntax

The complete syntax for MWrap call statements is given in Figure **??**. Each statement makes a function or method call, and optionally assigns the output to a variable. For each argument or return variable, we specify the type and also say whether the variable is being used for input, for output, or for both. Variables are given by names which should be valid identifiers in the local MATLAB environment where the call is to be made. Input arguments can also be given numeric values, though it is still necessary to provide type information.

There are three types of call specifications. Ordinary functions not attached to a class can be called by name:

```
# do_something();
```

To create a new C++ object instance, we use the `new` command

```
# Thermometer* therm = new Thermometer(double temperature0);
```

And once we have a handle to an object, we can invoke methods on it

```
# double temperature = therm->Thermometer.get_temperature();
```

Object deletion is handled just like an ordinary function call

```
# delete(Thermometer* therm);
```

Intrinsic operators like `sizeof` can also be invoked in this manner. The type specifications are *only* used to determine how MWrap should handle passing data between MATLAB and a C/C++ statement; the types specified in the call sequence should be compatible with a corresponding C/C++ definition, but they need not be identical to the types in a specific function or method declaration.

An MWrap type specification consists of three parts. The first (optional) part says whether the given variable will be used for input (`input`), for output (`output`), or for both. The second part gives the basic type name for the variable; this may be an intrinsic type like `int` or `double`, a string, an object type, or a MATLAB intrinsic (see Section **??**). Finally, there may be modifiers to specify that this is a pointer, a reference, or an array. Array and string arguments may also have explicitly provided size information. In the example from the introduction, for example the argument declaration

```
inout cstring[128] s1
```

tells MWrap that `s1` is a C string which is used for input and output, and that the buffer used should be at least 128 characters long.

Identifiers in MWrap may include C++ scope specifications to indicate that a function or method belongs to some namespace or that it is a static member of a class. That is, it is valid to write something like

```
std::ostream* os = foo->get_ostream();
```

Scoped names may be used for types or for method names, but it is an unchecked error to use a scoped name for a parameter or return variable.

4 Variable types

MWrap recognizes several different general types of variables as well as constant expressions:

4.1 Numeric types

Scalars are intrinsic numeric types in C/C++: `double`, `float`, `long`, `int`, `char`, `ulong`, `uint`, `uchar`, `bool`, and `size_t`. These are the numeric types that MWrap knows about by default, but if necessary, new numeric types can be declared using `typedef` commands. For example, if we wanted to use `uint32_t` as a numeric type, we would need the line

```
# typedef numeric uint32_t;
```

Ordinary scalars cannot be used as output arguments.

Scalar pointers are pointers to the recognized numeric intrinsics. They are assumed to refer to *one* variable; that is, a `double*` in MWrap is a pointer to one double in memory, which is different from a double array (`double[]`).

statement := returnvar `=` func `(` args `);`
:= func `(` args `);`
:= `typdef numeric` *type-id* `;`
:= `typdef dcomplex` *type-id* `;`
:= `typdef fcomplex` *type-id* `;`
:= `class` *child-id* `:` *parent-id* `,` *parent-id* `,` ...

func := *function-id*
:= `FORTRAN` *function-id*
:= *this-id* `.` *class-id* `->` *method-id*
:= `new` *class-id*

args := arg `,` arg `,` ... | ϵ
arg := iospec type *var-id*
:= ispec type *value*
returnvar := type *var-id*

iospec := `input` | `output` | `inout` | ϵ
ispec := `input` | ϵ
type := *type-id* | *type-id* `*` | *type-id* `&` | *type-id* `[` dims `]` | *type-id* `[` dims `]` `&`
dims := dim `,` dim `,` ... | ϵ
dim := *var-id* | *number*

Figure 1: MWrap call syntax

Scalar references are references to the recognized numeric intrinsics.

Arrays store arrays of recognized numeric intrinsics. They may have explicitly specified dimensions (in the case of pure return arrays and pure output arguments, they *must* have explicitly specified dimensions), but the dimensions can also be automatically determined from the input data. If only one dimension is provided, return and output arrays are allocated as column vectors.

If a function is declared to return an array or a scalar pointer and the C return value is NULL, MWrap will pass an empty array back to MATLAB. If an empty array is passed to a function as an input array argument, MWrap will interpret that argument as NULL.

Array references are references to numeric arrays, such as in a function whose C++ prototype looks like

```
void output_array(const double*& data);
```

Array references may only be used as output arguments, and the array must have explicitly specified dimensions. If the value of the data pointer returned from the C++ function is NULL, MWrap will pass an empty array back to MATLAB.

Complex scalars pose a special challenge. C++ and C99 provide distinct complex types, and some C89 libraries define complex numbers via structures. If the `-cppcomplex` or `-c99complex` flags are specified, `mwrap` will automatically define complex double and single precision types `dcomplex` and `fcomplex` which are bound to the C++ or C99 double-precision and single-precision complex types. More generally, we allow complex numbers which are conceptually pairs of floats or doubles to be defined using `typedef fcomplex` and `typedef dcomplex` commands. For example, in C++, we might use the following commands to set up a double complex type `cmplx` (which is equivalent to the `dcomplex` type when the `-cppcomplex` flag is used):

```
$ #include <complex>
$ typedef std::complex<double> cmplx;   // Define a complex type
$ #define real_cmplx(z) (z).real()      // Accessors for real, imag
$ #define imag_cmplx(z) (z).imag()      //  (req'd for complex types)
$ #define setz_cmplx(z,r,i) *z = dcomplex(r,i)

# typedef dcomplex cmplx;
```

The macro definitions `real_cmplx`, `imag_cmplx`, and `setz_cmplz` are used by MWrap to read or write the real and imaginary parts of a number of type `cmplx`. Similar macro definitions must be provided for any other complex type to be used.

Other than any setup required to define what will be used as a complex type, complex scalars can be used in all the same ways that real and integer scalars are used.

4.2 Strings

Strings are C-style null-terminated character strings. They are specified by the MWrap type `cstring`. A `cstring` type is not equivalent to a `char[]` type, since the latter is treated as an array of numbers (represented by a double vector in MATLAB) in which zero is given no particular significance.

The dimensions can be of a `cstring` can explicitly specified or they can be implicit. When a C string is used for output, the size of the corresponding character buffer *must* be given; and when a C string is used for input, the size of the corresponding character buffer should not be given.

If a function is declared to return a C string and the return value is NULL, MWrap will pass back the scalar 0.

4.3 Objects

Objects are variables with any base type other than one of the recognized intrinsics (or the `mxArray` pass-through – see below). This can lead to somewhat startling results when, for example, MWrap decides a `size_t` is a dynamic object type (this will only give surprises if one tries to pass in a numeric value). If a function or method returns an object, MWrap will make a copy of the return object on the heap and pass back that copy.

Object references are treated the same as objects, except that when a function returns an object reference, MWrap will return the address associated with that reference, rather than making a new copy of the object.

Object pointers may either point to a valid object of the appropriate type or to NULL (represented by zero). This is different from the treatment of objects and object references. When a NULL value is specified for a `this` argument, an object argument, or an object reference argument, MWrap will generate a MATLAB error message.

If the wrapped code uses an object hierarchy, you can use MWrap class declarations so that valid casts to parent types will be performed automatically. For example, the declaration

```
# class Child : Parent1, Parent2;
```

tells MWrap that an object of type `Child` may be passed in where an object of type `Parent1` is required. The generated code works correctly with C++ multiple inheritance.

Objects cannot be declared as output or inout parameters, but that just means that the identity of an object parameter does not change during a call. There's nothing wrong with changing the internal state of the object.

By default, MWrap stores non-NULL object references in strings. However, for MATLAB 2008a and onward, MWrap will also interpret as objects any classes with a readable property `mwptr`. This can be used, for example, to implement class wrappers using the new `classdef` keyword. In order to use this feature, the macro `R2008OO` must be defined by adding the argument `-DR2008OO` to the `mex` compile line.

4.4 `mxArray`

The *mxArray* type in MWrap refers to MATLAB's basic object type (also called `mxArray`). `mxArray` arguments can be used as input or output arguments (but not as inout arguments), or as return values. On input, `mxArray` is mapped to C type `const mxArray*`; on output, it is mapped to `mxArray**`; and on return, it is mapped to `mxArray*`. For example, the line

```
# mxArray do_something(mxArray in_arg, output mxArray out_arg);
```

is compatible with a function defined as

```
mxArray* do_something(const mxArray* in_arg, mxArray** out_arg);
```

Note that the header file `mex.h` must be included for this function declaration to make any sense.

The primary purpose for the mxArray pass through is to allow specialized routines to read the internal storage of MATLAB sparse matrices (and possibly other structures) for a few routines without giving up the convenience of the MWrap framework elsewhere.

4.5 Auto-converted objects

If there is a natural mapping from some MATLAB data structure to a C/C++ object type, you can use a typedef to tell MWrap to perform automatic conversion. For example, if we wanted `Foo` to be automatically converted from some MATLAB data structure on input, then we would add the line

```
# typedef mxArray Foo;
```

With this declaration, `Foo` objects are automatically converted from `mxArray` to the corresponding C++ type on input, and back to `mxArray` objects on output. It is assumed that MWrap *owns the argument objects* and *does not own the return objects*. This feature should not be used when the C++ side keeps a pointer or reference to a passed object, or when the caller is responsible for deleting a dynamically allocated return object.

Auto-converted objects rely on the following user-defined functions:

```
Foo* mxWrapGet_Foo(const mxArray* a, const char** e);
mxArray* mxWrapSet_Foo(Foo* o);
Foo* mxWrapAlloc_Foo();
void mxWrapFree_Foo(Foo* o);
```

Not all functions are needed for all uses of an auto-converted type. The functions play the following roles:

1. The `mxWrapGet_Foo` function is used to convert an input argument to the corresponding C++ representation. If an error occurs during conversion, the second argument should be pointed toward an error message string. It is assumed that this conversion function will catch any thrown exceptions.

2. The `mxWrapSet_Foo` function is used to convert an output argument or return value to the corresponding C++ representation.

3. The `mxWrapAlloc_Foo` function is used to allocate a new temporary for use as an output argument.

4. The `mxWrapFree_Foo` function is used to deallocate a temporary created by `mxWrapGet_Foo` or `mxWrapAlloc_Foo`.

The point of auto-converted objects is to simplify wrapper design for codes that make heavy use of things like C++ vector classes (for example). The system does *not* provide the same flexibility as the `mxArray` object, nor is it as flexible as a sequence of MWrap calls to explicitly create and manage temporaries and their conversion to and from MATLAB objects.

At present, the behavior when you try to involve an auto-converted object in an inheritance relation is undefined. Don't try it at home.

4.6 Constants

The *const* type in MWrap refers to a C/C++ symbolic constant or global variable. The name of the variable is output directly into the compiled code. For example, to print a string to `stderr`, we can write

```
# fprintf(const stderr, cstring s);
```

5 Example

An event queue stores pairs (i, t) pairs, i is an identifier for some event and t is a time associated with the event. Events can be added to the queue in whatever order, but they are removed in increasing order by time. In this example, we bind to a C++ event queue implementation based on the C++ standard template library priority queue. The example code is in `example/eventq/eventq_class.mw` and `example/eventq/eventq_handle.mw`; an alternate version of the code in `example/eventq/eventq_plain.mw` illustrates a different way of organizing the same interface. The `example/eventq2` subdirectory provides yet another implementation, this one capable of storing arbitrary MATLAB arrays rather than just integers.

5.1 Event queue using old MATLAB OO

We begin by defining an event as a pair (double, int), and an event queue as an STL priority queue of such pairs, sorted in descending order:

```
$ #include <queue>
$
$ typedef std::pair<double, int>                        Event;
$ typedef std::priority_queue< Event,
$                              std::vector<Event>,
$                              std::greater<Event> >  EventQueue;
```

Now we specify the code to wrap the individual methods. For this example, we will take advantage of the object oriented features in MATLAB, and map the methods of the C++ event queue class onto methods of a MATLAB wrapper class called `eventq`. We begin with bindings for the constructor and destructor. We will compile the MATLAB functions for the interface using the `-mb` flag, so that we can specify these functions (and all the others) in the same file:

```
@ @eventq/eventq.m -------------------------------------

function [qobj] = eventq();

qobj = [];
# EventQueue* q = new EventQueue();
qobj.q = q;
qobj = class(qobj, 'eventq');


@ @eventq/destroy.m ------------------------------------

function destroy(qobj);

q = qobj.q;
# delete(EventQueue* q);
```

The `empty` method returns a `bool`, but MWrap does not know about `bool` variables. A `bool` result can be saved as an integer, though, so we will simply do that:

```
@ @eventq/empty.m --------------------------------------

function [e] = empty(qobj)

q = qobj.q;
# int e = q->EventQueue.empty();
```

Because `pop_event` needs to return two values (the event identifier and the time), we use reference arguments to pass out the information.

```
@ @eventq/pop_event.m ----------------------------------

function [id, t] = pop_event(qobj)

$ void pop_event(EventQueue* q, int& id, double& t) {
$     t  = q->top().first;
$     id = q->top().second;
$     q->pop();
$ }
$
q = qobj.q;
# pop_event(EventQueue* q, output int& id, output double& t);
```

In MATLAB, it may make sense to simultaneously push several events. However, our event queue class only provides basic interfaces to push one event at a time. We could write a MATLAB loop to add events to the queue one at a time, but for illustrating how to use MWrap, it is better to write the loop in C:

```
@ @eventq/push_event.m -----------------------------------

function push_event(qobj, id, t)

$ void push_events(EventQueue* q, int* id, double* t, int m)
$ {
$     for (int i = 0; i < m; ++i)
$         q->push(Event(t[i], id[i]));
$ }
$
q = qobj.q;
m = length(id);
# push_events(EventQueue* q, int[m] id, double[m] t, int m);
```

5.2 Event queue using new MATLAB OO

Starting with MATLAB 7.6 (release 2008A), MATLAB supports a new single-file OO programming style. Particularly convenient for writing wrappers is the *handle* class system, which allows the user to define destructors that are called automatically when an instance is destroyed by the system (because all references to the instance have gone out of scope). As a programming convenience, MWrap automatically interprets a class with the property `mwptr` as a container for an MWrap object[1]. For example, the following file provides an alternate implementation of the event queue class described in the previous section.

```
$ #include <queue>
$
$ typedef std::pair<double, int>                      Event;
$ typedef std::priority_queue< Event,
$                              std::vector<Event>,
$                              std::greater<Event> >  EventQueue;

@ eventqh.m -------------------------------------------

classdef eventqh < handle

  properties
    mwptr
  end

  methods

    function [qobj] = eventqh(obj)
```

[1]This functionality is only enabled when `-DR2008OO` is specified as an argument on the MEX command line. This restriction is in place so that the files generated by MWrap can remain compatible with Octave and with older versions of MATLAB.

```
    # EventQueue* q = new EventQueue();
    qobj.mwptr = q;
  end

  function delete(q)
    #delete(EventQueue* q);
  end

  function e = empty(q)
    # int e = q->EventQueue.empty();
  end

  function [id, t] = pop(q)
    $ void pop_event(EventQueue* q, int& id, double& t) {
    $     t  = q->top().first;
    $     id = q->top().second;
    $     q->pop();
    $ }
    # pop_event(EventQueue* q, output int& id, output double& t);
  end

  function push(q, id, t)
    $ void push_events(EventQueue* q, int* id, double* t, int m)
    $ {
    $     for (int i = 0; i < m; ++i)
    $         q->push(Event(t[i], id[i]));
    $ }
    m = length(id);
    # push_events(EventQueue* q, int[m] id, double[m] t, int m);
  end

 end
end
```

This implementation of the event queue class allows for automatic cleanup:

```
q = eventqh();   % Construct a new queue
clear q;         % The C++ object gets correctly deleted here
```

Warning: When using MATLAB handle classes for automatic cleanup, be sure to avoid situations where multiple MATLAB handle objects have been given responsible for deleting a single C/C++ object. If you need to have more than one MATLAB handle for a single C/C++ object, I recommend using a reference counted pointer class as an intermediate[2].

[2]For more information on reference counted pointer classes, I recommend reading *More Effective C++* by Scott Meyers.

6 FORTRAN bindings

It is possible to use MWrap to bind FORTRAN functions (though the generated MEX file is still a C/C++ file). FORTRAN bindings can be specified using the `FORTRAN` keyword immediately before a function name; for example:

```
# double sum = FORTRAN dasum(int n, double[n] x, int 1);
```

FORTRAN parameters are treated differently from C parameters in the following ways:

1. MWrap does not allow objects to be passed into FORTRAN functions.
2. Scalar and reference arguments are automatically converted to pointer arguments from the C side to match FORTRAN call-by-reference semantics.
3. A warning is generated when passing C strings into FORTRAN. The generated code will work with compilers that produce f2c-compatible code (including g77/g95), but it will not work with all FORTRAN compilers.
4. Only simple numeric values can be returned from FORTRAN. A warning is generated when returning complex values, as different FORTRAN compilers follow different protocols when returning complex numbers. The generated code for complex return types will work with some f2c-compatible compilers, but by no means all.

Internally, MWrap defines macros for different FORTRAN name-mangling conventions, and it declares appropriate prototypes (and protects them from C++ compiler name mangling). By default, MWrap assumes that the f2c name mangling convention is being used (this convention is followed by Sun FORTRAN, g77, and g95); however, the following flags can be passed to the `mex` script to change this behavior:

- `-DMWF77_CAPS` – Assume the FORTRAN compiler uses all-caps names and no extra underscores. Used by Compaq FORTRAN (I think).
- `-DMWF77_UNDERSCORE1` – Append a single underscore to an all-lower-case name. Used by the Intel FORTRAN compiler.

It is possible to use the `typedef numeric` construct to introduce new types corresponding to FORTRAN data types. For example, if the header file `f2c.h` is available (and the types defined therein are appropriate for the compiler) we might have

```
% Use the f2c integer type...

$ #include "f2c.h"
# typedef numeric integer;
# double sum = FORTRAN dasum(integer n, double[n] x, integer 1);
```

No attempt is made to automatically produce these type maps, though.

7 Logging

For profiling and coverage testing, it is sometimes useful to track the number of calls that are made through MWrap. If `mymex` is the name of the generated MEX file, then we can access profile information as follows:

```
% Enable profiler
mymex('*profile on*');

% Run some tests here...

% Print to screen and to a log file
mymex('*profile report*');
mymex('*profile log*', 'log.txt');

% Disable profiler
mymex('*profile off*');
```

The MEX file will be locked in memory as long as profiling is active.