

# Writing C modules for Pike

Julio César Gázquez

April 11, 2001

## 1 Introduction

After you learn Pike and start programming in it, you will start soon structuring your work through modules. You'll write generic modules, or subsystems of your application as Pike modules.

However, sooner or later you'll want to write a module in C. There are some reasons to don't write modules in Pike but in C:

- Speed. If your algorithm needs some extra speed, and you can't refine your logic, maybe you want to recode it in C.
- Accessing libraries. There are a lot of libraries written in C and ready to use. But you need some glue before you can call its functions from your Pike program.

## 2 Before starting

The more obvious requirements are some knowledge of both Pike and C. You should also read the chapter 17 in Hubbe's Tutorial. He explains there the data types used into the interpreter and C modules, and some stuff you maybe can use in your code.

## 3 Anatomy of a module

In a C module you define, as in a Pike module, some programs(classes), with variables(object attributes) and functions(object methods). It also can define global functions. Those are not related to any instance of the programs you define, but to the whole module, acting (maybe?) as class methods.

To achieve all this, a C module usually has the following elements:

- A struct declaring the data you will use in each instance.
- Functions that provide the functionality of each method of each class you want to define.
- A function (*pike\_module\_init*) that allows the definition of the new programs and standalone functions into the Pike environment, as any further initialization the class needs.
- A function (*pike\_module\_exit*) that allows any cleanup needed.
- A pair of functions per program that allows to init and cleanup each object you create. For example, you should take care of giving initial values to your attributes, that should be zero if you don't have any special value, as all Pike variables are created in this way. You shouldn't do this in the create and destroy methods, as those methods can be overridden in a child class.

## 4 A stupid example

We are going to create a simple module. It will allow the creation of a pretty dumb class, that allow us to add a pair of integers. Also, we'll include a standalone function.

```
#include <stdio.h>

#include "global.h"
```

```

#include "stralloc.h"
#include "error.h"
#include "pike_macros.h"
#include "object.h"
#include "constants.h"
#include "interpret.h"
#include "svalue.h"
#include "mapping.h"
#include "builtin_functions.h"
#include "module_support.h"

struct my_data
{
    int a;
    int b;
};
#define this ((struct my_data *) fp->current_storage)

```

So far, we have system and Pike header includes, the struct definition for our object's data, and a useful macro to access those data from our functions.

Before adding the code for the functions, let's see the the trailing part of the file:

```

static void init_callback( struct object *o )
{
    this->a = 0;
    this->b = 0;
}

/*
static void exit_callback( struct object *o )
{
}
*/
void pike_module_init(void)
{
    /* function(string:void) */
    ADD_EFUN("printwithstars", f_printwithstars,tFunc(tStr,tVoid), 0);

    start_new_program();

    ADD_STORAGE(struct my_data);

    set_init_callback( init_callback );
    /* set_exit_callback( exit_callback ); */

    /* function(int:void) */
    ADD_FUNCTION("setA", f_seta,tFunc(tInt,tVoid), 0);

    /* function(int:void) */
    ADD_FUNCTION("setB", f_setb,tFunc(tInt,tVoid), 0);

    /* function(int:void) */
    ADD_FUNCTION("sum", f_sum,tFunc(tVoid,tInt), 0);

    end_class("myclass", 0);
}

void pike_module_exit(void)
{

```

```

}

```

First, we see the object init and cleanup callbacks. Note the using of “this” macro in order to access the attributes of the object. As we don’t need to cleanup those objects, we don’t really define the cleanup. Later is the module initialization. We first define a standalone function (*EFUN*). The macro `ADD_EFUN` has three parameters:

- Pike name of the function.
- The C function that implements it.
- What parameters it takes, and what value it returns.
- An options parameter.

The way of declaring the parameters and return of our function is through the `tFunc` macro. Its basic syntax is:

```

tFunc(paramtype paramtype, returntype)

```

The macros that defines the different types are defined in `svalue.h`. They are:

Macro	Pike equivalent
<code>tVoid</code>	<code>void</code>
<code>tInt</code>	<code>int</code>
<code>tFloat</code>	<code>float</code>
<code>tFlt</code>	<code>float</code>
<code>tString</code>	<code>string</code>
<code>tStr</code>	<code>string</code>
<code>tMixed</code>	<code>mixed</code>
<code>tMix</code>	<code>mixed</code>
<code>tArr(type)</code>	<code>array(type)</code>
<code>tArray</code>	<code>array(mixed)</code>
<code>tMap(type,type)</code>	<code>mapping(type:type)</code>
<code>tMapping</code>	<code>mapping(mixed:mixed)</code>
<code>tSet(type)</code>	<code>multiset(type)</code>
<code>tMultiset</code>	<code>multiset(mixed)</code>
<code>tFunction</code>	<code>function</code>
<code>tObj</code>	<code>object</code>
<code>tProgram</code>	<code>program</code>
<code>tOr(type,type)</code>	<code>(type type)</code>

The options parameters often is left as zero. However, we’ll see a few of them (those I found along the standard modules). You can bitwise-or them if you need it. If you want to see all of them, check `las.h`.

Option	Description
<code>OPT_ASSIGNMENT</code>	Does assignments.
<code>OPT_EXTERNAL_DEPEND</code>	The value returned depends on an external function.
<code>OPT_RETURN</code>	Contains return(s).
<code>OPT_SIDE_EFFECT</code>	Has side effects (it alters something the app itself).
<code>OPT_TRY_OPTIMIZE</code>	Might be worth optimizing .

After defining the efuns, come the program definitions. A program definition start is denoted by a `start_new_program` call. After that, you declare the callbacks, the storage structure, the member functions. The member function definition is quite similar to `efun` definition.

After that, you have two ways of denote the end of the program definition. The simpler is

```

end_class("myclass");

```

where `myclass` is the name given to the program in the Pike environment. Sometimes, however, you will need to clone the program yourself, then you will need a pointer to the program. To do so, you should write instead:

```

myclass_program=end_program();
add_program_constant("myclass", myclass_program, 0);

```

## 5 Writing the guts

The news are not finished. You can't define the functions just as a regular C functions. Instead you need to declare of all them as:

```
void f_myfunction(INT32 args)
```

The parameter `args` is the number of arguments sent to the function. To access the arguments themselves, you need to work with the interpreter's stack.

As Mirar literally said when I ask: "There are tons of ways of doing things". Maybe the simpler way to pick your argument values is using the function `get_all_args`. Look a few examples:

```
get_all_args("Regexp.regexp->match", args, "%s", &str);
get_all_args("init_creds", args, "%o%i%i", &o, &may, &data);
```

You pass the function name in a context proper way, the number of arguments, a scanlike string, and pointers to your work variables.

You can also work directly with the stack. In order to get a fair view of the subject, let's see a few code sections borrowed from `svalue.h`:

```
union anything
{
    struct callable *efun;
    struct array *array;
    struct mapping *mapping;
    struct multiset *multiset;
    struct object *object;
    struct program *program;
    struct pike_string *string;
    INT32 *refs;
    INT_TYPE integer;
    FLOAT_TYPE float_number;
    struct svalue *lval; /* only used on stack */
    union anything *short_lval; /* only used on stack */
};
struct svalue
{
    unsigned INT16 type;
    unsigned INT16 subtype;
    union anything u;
};
```

Then, well see the direct way of deal with parameters

```
void f_cheddar(INT32 args)
{
    if (args!=3)
        SIMPLE_TOO_FEW_ARGS_ERROR("cheddar",3);
    if (sp[-args].type!=T_INT)
        SIMPLE_BAD_ARG_ERROR("cheddar",0,"integer");
    if (sp[1-args].type!=T_STRING)
        SIMPLE_BAD_ARG_ERROR("cheddar",2,"string");
    if (sp[2-args].type!=T_ARRAY)
        SIMPLE_BAD_ARG_ERROR("cheddar",2,"array");

    ...
    INT_TYPE x=sp[-args].u.integer;
    struct pike_string *str=sp[1-args].u.string;
    struct array *arr=sp[2-args].u.array;
}
```

Of course, once we can access our arguments, we use them to whatever we need. Then, we just may need to return the result of our function to its caller. To do so, you can use one of those macros:

```
push_int(n)
push_float(f)
push_str(*str)
push_array(*arr)
push_mapping(*map)
push_program(*prg)
push_multiset(*set)
```

## 6 Compiling the module

To compile the module, you need a command like this:

```
gcc -shared -I /usr/local/pike/7.0.36/include/pike mymod.c -o mymod.so
```

After that, you will be able to use the module as usual.

## Acknowledges

I want to thank to Frederik “Hubbe” Hübinette, Mirar (I guess I’ll never know his real name :-)) and all the Pike developers and Pike community.