

Tutorials on Writing C Modules in Pike

Andreas Finnman

March 4, 2003

Contents

1	The C API	4
1.1	Overview of This Tutorial	4
1.2	The Absolute Minimum	4
1.3	set_init_callback()	4
1.4	set_exit_callback()	5
1.5	start_new_program()	5
1.6	end_class()	5
1.7	ADD_STORAGE()	6
1.8	Pike_fp->current_storage	6
1.9	ADD_FUNCTION()	7
1.10	tFunc	7
1.11	The Stack	8
1.11.1	Pike_sp	8
1.11.2	svalue	8
1.12	Interacting with the Stack	9
1.12.1	args	9
1.12.2	Pop Functions	10
1.12.3	Push Functions	10
1.13	Pike_error()	11
2	The CMOD API	12
2.1	Overview of This Tutorial	12
2.2	INIT - keyword	12
2.3	EXIT - keyword	12
2.4	PIKECLASS - keyword	12
2.5	PIKEFUN - keyword	13
2.6	CVAR - keyword	13
2.7	THIS - keyword	13
2.8	RETURN - keyword	14
3	Writing a C Module	15
3.1	Overview of This Tutorial	15
3.2	The Files of a Module	15
3.3	Getting Started	15
3.4	Pike Level and C Level	16
3.5	Define the Pike Level Interface	18

3.5.1	Without CMOD	18
3.5.2	With CMOD	19
3.6	Write the Functions	20
3.6.1	Without CMOD	20
3.6.2	With CMOD	21
3.7	Error Handling	21
3.7.1	Argument Checking without CMOD	21
3.7.2	Argument Checking with CMOD	21
3.7.3	Throwing Exceptions	22
3.8	Init and Exit Routines	22
3.9	An Example - The Rev Module	22
3.10	Define the Pike Level Interface	23
3.10.1	Without CMOD	23
3.10.2	With CMOD	25
3.11	Write the Functions	26
3.11.1	Without CMOD	26
3.11.2	With CMOD	27

1 The C API

1.1 Overview of This Tutorial

This tutorial contains explanations of the most common functions, macros, data structures and variables that a programmer needs to know about in order to do the C code implementation part of a C module. Note that this is no reference manual, nor is it a manual on how to write C modules. Words printed in bold in the text are of programming specific meaning. It can be a variable, function, data structure, data type or a macro. Using bold is an attempt to clarify their meaning in the running text. The text that is written in courier font is example code. It exists in order to show definitions of functions, macros and variables, and to illustrate how they are used.

1.2 The Absolute Minimum

In order to be recognized as a module by Pike, the least a module must define are the two functions:

```
void pike_module_init()  
void pike_module_exit()
```

As one can tell by the name, **pike_module_init()** is the function run when the module is being initialized. In **pike_module_init()** the whole module is so to say defined. By using (for instance) the functions and macros listed below, every component of the module (class, method, variable) can be defined within the Pike language and mapped to its corresponding C level implementation.

The **pike_module_exit()** function is as the name indicates run when the module exits. Therefore it can contain clean up that is to be done upon exit.

1.3 set_init_callback()

```
/*Prototype:*/  
void set_init_callback(void (*init_callback)(struct object *));
```

set_init_callback() sets the initialization callback of a class. The example line of code sets the function **init_my_module()** as the initialization callback of the corresponding class.

```
/*Example of usage:*/  
set_init_callback(init_my_module);
```

1.4 set_exit_callback()

```
/*Prototype:*/  
void set_exit_callback(void (*exit_callback)(struct object *));
```

set_exit_callback() sets the exit callback of a class. The example line of code sets the function **exit_my_module()** as the exit callback of the corresponding class.

```
/*Example of usage:*/  
set_exit_callback(exit_my_module);
```

1.5 start_new_program()

The **start_new_program()** is a kind of starting mark that is set first of all when defining a class in **pike_module_init()**. After this come all definitions, settings and so on. It is called without any arguments. Even though it is not entirely correct, Java and C++ programmers can think of **start_new_program()** as a 'class {' statement.

```
/*Example of usage:*/  
start_new_program();
```

1.6 end_class()

```
/*Macro:*/  
end_class(NAME, FLAGS)
```

end_class() is a terminating mark for **start_new_program()**. It takes two arguments. The first one is the name by which the class defined between the **start_new_program()** and the **end_class()**, will be called. The second argument is a flag. The following example line of code results in a public class called 'MyClass'.

```
/*Example of usage:*/  
end_class("MyClass", 0);
```

The fact that **start_new_program()** is terminated by **end_class()** is a bit confusing. It is this way because the words 'program' and 'class' are used synonymously in Pike and for some reason the two macros happened to be named this way.

1.7 ADD_STORAGE()

```
/*Macro:*/  
ADD_STORAGE(X)
```

ADD_STORAGE() is used to allocate a space in memory. It takes only one argument namely the name of the data type for which storage is to be provided. The macro is used to implement things like object variables. The following example line of code placed in a class makes sure a storage for the struct **number_and_letter** is created upon each instantiation of the class.

```
/*Example of usage:*/  
struct number_and_letter{  
    int number;  
    char letter;  
};  
ADD_STORAGE(number_and_letter);
```

The first **ADD_STORAGE()** that is placed in a class will yield a handle that can be accessed for instance when writing a function for that class. This handle is kept track of so that the storage can be accessed as long as the corresponding object exists. When writing the implementation of a class this handle is then the `Pike_fp->current_storage` variable.

If a second storage is added to a class, it will be found in an offset from the first storage. Same goes for a third storage if it is added. The programmer will have to keep track of the offsets manually.

1.8 Pike_fp->current_storage

```
/*Variable:*/  
Pike_fp->current_storage
```

This variable contains a pointer to the current storage. Current storage means the storage belonging to the object that is currently worked with. The pointer is used to access storage when writing the implementations of class methods and such. The following example shows how a 'THIS' pointer is defined and used in order to access the object variable **value**.

```
/*Example of usage:*/  
struct data{
```

```

    int value;
    int label;
};
#define THIS (struct data *)Pike_fp->current_storage
THIS->value = 14;

```

1.9 ADD_FUNCTION()

```

/*Macro:*/
ADD_FUNCTION(NAME, FUNC, TYPE, FLAGS)

```

ADD_FUNCTION() is a macro to define a Pike level method and map it to a C level function in the module. It takes four arguments. The first argument is the Pike level name of the function (or method as it is called on the Pike level). This is the name a Pike programmer will call the method by. Note that this argument is a string (array of characters). The second argument is the name of the corresponding C level function. That means that when the Pike method with the name equal to the first argument is invoked, Pike will look for the C function with a name equal to the second argument and run that function. The third argument specifies what type of method it is and what arguments the method takes. As it is seen in the example the macro **tFunc()** is used in order to specify the third argument. The fourth argument is a flag that specifies the modifier of the method.

The following example of usage adds a method that has one string as argument and an integer as second optional argument. The return value of the method is of type array.

```

/*Example of usage:*/
ADD_FUNCTION("my_method", my_method_C_impl,
            tFunc(tString tOr(tInt,tVoid),tArray), 0);

```

A Pike declaration of the method would look like this:

```

array my_method(string str, int|void number)

```

1.10 tFunc

```

/*Macro:*/
tFunc(arg1 arg2 arg3 ... argn, ret)

```

This macro specifies a function. The arguments of the macro specify the type of the specified Pike method's arguments and return value respectively. Note here that all arguments of the specified function are listed on a row separated only by a space. After the comma comes the type of the return value of the specified Pike method. The example below specifies a method that takes an **int** as first argument, a **string** as second argument, and has a third optional argument of type **int**. The return type is **int**. Note that the Pike types are meant here.

```
/*Example of usage:*/  
tFunc(tInt tString tOr(tInt,tVoid), tInt)
```

1.11 The Stack

All arguments to, and return values from, underlying C functions are passed through the stack. The elements on the stack are `svalue` structs so that all kinds of data structures can be placed "on" it.

1.11.1 Pike_sp

`Pike_sp` is the global stack pointer. `Pike_sp[0]` always points to the top of the stack. `Pike_sp[-1]` is the top element on the stack, `Pike_sp[-2]` the element below, and so on. The example shows a typical fetching of an integer from the top element of the stack.

```
/*Example of usage:*/  
int number = Pike_sp[-1].u.integer;
```

When a C level function is called and the arguments are passed to it by the stack, it can find its first argument in `Pike_sp[-args]`, its second in `Pike_sp[1-args]` and its *n*'th argument in `Pike_sp[n-1-args]`. This means that argument *i* is found in `Pike_sp[i-1-args]` for every $1 \leq i \leq \text{args}$. The elements of **Pike_sp** are of type **struct svalue**.

1.11.2 svalue

```
/*Definition of data type svalue:*/  
union anything  
{  
    INT_TYPE integer; /* Union initializations
```

```

                                assume this first. */
struct callable *efun;
struct array *array;
struct mapping *mapping;
struct multiset *multiset;
struct object *object;
struct program *program;
struct pike_string *string;
struct pike_type *type;
INT32 *refs;
struct ref_dummy *dummy;
FLOAT_TYPE float_number;
struct svalue *lval;          /* only used on stack */
union anything *short_lval; /* only used on stack */
void *ptr;
};

struct svalue
{
    unsigned INT16 type;
    unsigned INT16 subtype;
    union anything u;
};

```

The **svalue** is a data structure to store references to any data type that a programmer might need to pass through the stack.

1.12 Interacting with the Stack

1.12.1 args

Usually the C level function that correspond to a Pike level method are invoked with the argument **args**. **args** is an integer that specifies how many 'arguments' the C level function should take. Note here that when speaking of 'arguments' in this context we do not mean the actual arguments of the C function, but the arguments of the Pike method that it implements. Each underlying C function has only one actual argument and that is the **args** parameter. The **args** top items on the stack then is the data that the C function has to fetch and use. In other words,

what the `args` parameter actually does specify is with how many arguments the overlying Pike method was invoked. For the module programmer `args` is then number of items that his underlying C function has to remove from the stack.

1.12.2 Pop Functions

```
/*Prototype:*/  
void pop_n_elems(INT32 elems);
```

`pop_n_elems()` is a function that pops `n` elements from the stack. Normally this function is called at the end of a C level function that has 'arguments' on the stack. The most usual call is:

```
/*Example of usage:*/  
pop_n_elems(args);
```

This pops the top `args` elements from the stack. If the stack has not been altered since the function was called then that is the same as removing the 'arguments' from the stack. There is also a function called `pop_stack()`.

```
/*Example of usage:*/  
pop_stack();
```

This function can be called instead of `pop_n_elems()` if top element on the stack is to be removed.

1.12.3 Push Functions

Return values are pushed to the stack. Since Pike has a lot of different data types there are a lot of push functions. They all work more or less the same. What they do is that they take the parameter and push it to the stack. The call to push a variable called `arg` of type `foo` would be `push_foo(arg)`. Two examples of push functions are listed below:

```
/*Example of usage:*/  
push_int(1378);
```

`push_int()` pushes an integer on the stack. The example pushes 1378 on the top of the stack.

```
/*Example of usage:*/
    struct pike_string *str = make_shared_binary_string("",0);
    push_string(str);
```

push_string() pushes a **pike_string** to the stack. The example shows how the empty string is pushed to the stack.

1.13 Pike_error()

This function is used to throw exceptions when errors occur. A call to **Pike_error()**, like the one in the example, raises an exception. The argument is a message which is printed to standard error if the exception is not caught.

```
/*Example of usage:*/
    Pike_error("Too many arguments in call to method");
```

2 The CMOD API

2.1 Overview of This Tutorial

This tutorial describes the CMOD interface in Pike. It contains descriptions of the most common CMOD keywords and lists some examples that illustrates how these keywords are used. The words printed in bold in this tutorial have some programming specific meaning. It can be a function, method, class, variable, macro or CMOD keyword.

2.2 INIT - keyword

INIT is used to set an initialization callback. Each class ought to have an **INIT**. What stands in between the curly brackets of **INIT** is what will be done upon initialization of an object of that class. In the example below two object variables called **number_of_iterations** and **current_string** are assigned their initial values.

```
/*Example of usage:*/
INIT{
    THIS->number_of_iterations = 0;
    THIS->current_string = "";
}
```

2.3 EXIT - keyword

EXIT is used to set an exit callback. What stands in between the curly brackets of **EXIT** is what will be done just before an object of that class is being garbage collected. In the example memory allocated and held as an object variable is freed before the object is garbage collected.

```
/*Example of usage:*/
EXIT{
    free(THIS->buffer);
}
```

2.4 PIKECLASS - keyword

This keyword is used to add a Pike class. It is followed by the name of the class that is to be added. In the example, a class called **MyClass** is added.

```
/*Example of usage:*/  
    PIKECLASS MyClass{}
```

2.5 PIKEFUN - keyword

PIKEFUN adds a method to the class. It is followed by the corresponding Pike prototype of the method that is to be added. The whole implementation of the method is written in between the curly brackets. The example shows how a method named **my_method()** is added.

```
/*Example of usage:*/  
    PIKEFUN array my_method(string arg1, int|void arg2){}
```

my_method takes a string as first argument, has an optional second integer argument, and returns an array. Note here that all data types are Pike types. That is, the type of the method and the types of the arguments are all specified using the Pike data type keywords.

2.6 CVAR - keyword

CVAR is used to declare object variables for internal use. **CVAR** is followed by the C declaration of the variable. **CVAR** provides storage for the variable whose declaration follows right after. The example statement placed in a class would result in an object global integer variable named **number**. **number** would then be accessed using the **THIS** pointer.

```
/*Example of usage:*/  
    CVAR int number;
```

2.7 THIS - keyword

THIS is a pointer to the current object. It is used mainly to access items within the object such as object variables. In the example an object variable **number** is incremented by five.

```
/*Example of usage:*/  
    THIS->number += 5;
```

2.8 RETURN - keyword

RETURN returns its argument from the method in whose implementation it is placed to the caller. What this actually means is to first remove **args** elements from the stack and then push its argument to the stack. The example is a return of 0.

```
/*Example of usage:*/  
    RETURN(0);
```

3 Writing a C Module

3.1 Overview of This Tutorial

This tutorial contains some information on what a programmer needs to keep in mind when writing a module for Pike using the C or CMOD API. The aim is to describe how C level implementation is to be done, not how Pike level design is done. It is however an attempt to describe the writing of a module both using CMOD and not using CMOD. The descriptions are first of general kind. Later on in this tutorial these descriptions are exemplified by a very small module called 'Rev'.

The document contains some terms that are C or CMOD API specific. Therefore it is recommended to read the documentation of the C API and the documentation of the CMOD API before or in parallel with this tutorial.

3.2 The Files of a Module

A C/CMOD module normally consists of the following files

1. The C/CMOD code files
2. A `configure.in` file
3. A `Makefile.in` file
4. A `testsuite.in` file

This is the minimum set of files needed in the source of a module. The C/CMOD code files can be one or more files. They contain the whole C/CMOD implementation of the module. The `configure.in` file is a file for the `configure` script. The `configure` script is responsible for generating the global `Makefile`. In `configure.in` conditions and tests are written for when and if the module is to be build.

The `Makefile.in` is a file needed to make additions to the global `Makefile` so that the module is built when the Pike is built. Each `Makefile.in` contains a list of the targets needed in order to make that specific module.

3.3 Getting Started

In order to write a new C module for Pike the first thing we need to do is to add a new module to our Pike source tree. To do that we need to create a new directory

for the module in our module tree. In that directory we need a skeleton for our new module. A skeleton in this context means a working set of the four files mentioned in the previous section. A skeleton that contains the files ¹ necessary and some guiding instructions ² for getting started can be found on the Pike web site. There is also a module with a hello world application³, that compiles out of the box and can serve as a guiding example.

Once the skeleton files are added and adjusted so that the new module ought to exist the Pike has to be reconfigured and recompiled. After that, if our module exists we can start to write C/CMOD code.

3.4 Pike Level and C Level

One way to think of a module, is to divide it into Pike level and C level. The items on the Pike level are abstract entities through which the Pike programmer is interfaced. The C level on the other hand contains the implementation of all Pike level items. That means that all Pike methods that belong to a C module in reality are C functions invoked through a Pike level interface. Therefore writing the C code of a module can be divided into two major steps.

First, the Pike level interface needs to be defined. The module programmer has to define the components of the module. In other words, he has to define the classes, methods, constants, and variables that his module is going to contain. Where needed he also has to map the components to the C level item (in most cases a function) that will represent the implementation of that component.

When the Pike interface has been defined, what we have is stubs for the whole module. This is where the second major step starts: to turn these stubs into a working module. In plain English: the code for the C functions that are the implementation of the defined Pike methods has to be written.

The following two sections describe more in detail how these two steps are conducted. There are major differences depending on if we use CMOD or not, that is why both sections are divided into two parts.

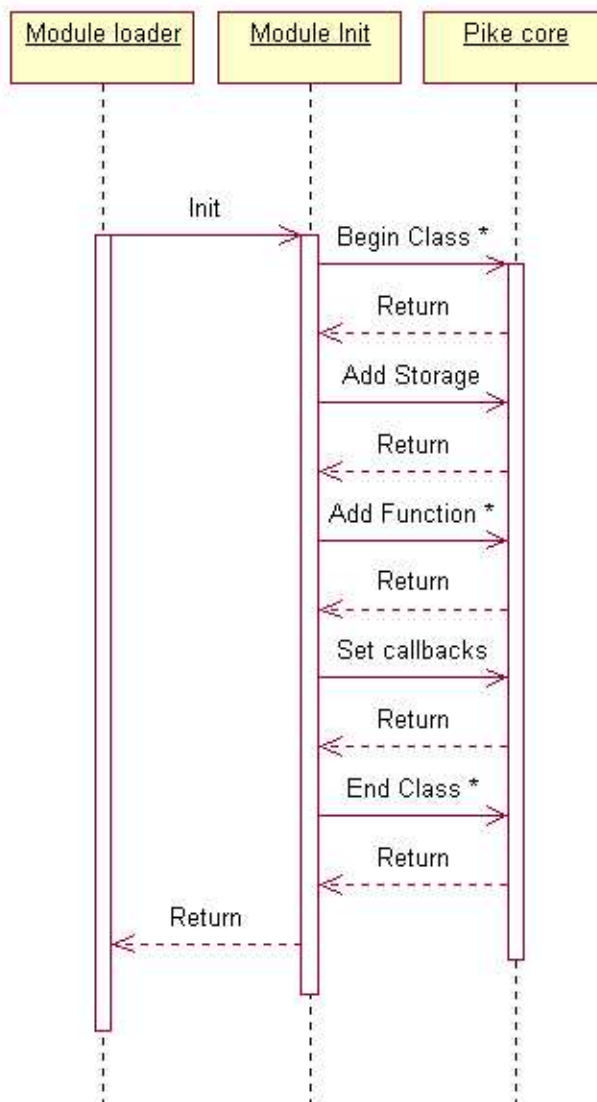


Figure 1: Sequence chart, initialization of a module

3.5 Define the Pike Level Interface

3.5.1 Without CMOD

Let us assume we want to write a new C module and that a Pike level design of that module already has been done. Let us also assume we want to do it the old fashion way and not use CMOD. As mentioned in the section above we need to define our Pike level interface. The question is how is that done? The keyword here, if we are not using CMOD, is the **pike_module_init()** function. Even though it might not be entirely correct from a technical point of view, we can say that the whole module with all classes and methods is defined in this function. The sequence chart in figure 5.1 illustrates what happens when a module is initialized. As it can be seen in the chart the initialization is so to speak the definition of the Pike level interface since each component in the module is added here.

To add a class **start_new_program()** and **end_class()** calls are placed in the **pike_module_init()** function. These two calls when used together add an empty class to the module.

If we want to add a method to that empty class we use the macro **ADD_FUNCTION()**. We place the call to **ADD_FUNCTION()** somewhere in between the calls to **start_new_program()** and **end_class()** of the class that we want the method to belong to.

To add private object variables the macro **ADD_STORAGE()** is useful. Variables that need to be kept in a storage, are those that need to be 'remembered' over several function calls. In other words, variables that in some way represent the inner state of an object need to be kept in storage. To add storage to a class we need to place a call to **ADD_STORAGE()** somewhere in between the call to **start_new_program()** and the call to **end_class()** of the class that we want the storage to belong to. The first call to **ADD_STORAGE()** after a call to **start_new_program()** will yield a handle. Storages added by other calls will lie within an offset of this storage. The module programmer will have to keep track of that offset manually if he chooses to add more than one storage. Therefore, an easy way to handle multiple object variables without adding more than one storage is to pack them all into one struct on the C level. Then this struct is made the first and only added storage of the class. Every object variable can then be accessed using the **Pike_fp->current_storage** pointer. So in order to add all private ob-

¹<http://pike.ida.liu.se/projects/docs/cmods/skeleton.tgz>

²<http://pike.ida.liu.se/projects/docs/cmods/steps.xml>

³<http://pike.ida.liu.se/projects/docs/cmods/hello.tgz>

ject variables, we first create a struct that contains all the variables needed. Then place a call to **ADD_STORAGE()** with that struct as only argument, somewhere in between the corresponding **start_new_program()** and **end_class()** .

Last in each definition of a class we need to set the initialization and exit callbacks. The init callback is set by adding a call to **set_init_callback()**. The exit callback is set by adding a call to **set_exit_callback()**.

In order for the now defined Pike level interface to compile, stubs for the functions that will be written later on, have to be added. All C functions that represent the implementation of a Pike method need to look like this:

```
static void function_name(INT32 args).
```

That is just the way the C API works. If we have defined a method (using **ADD_FUNCTION()**) that is specified to be implemented by the C function **function_name()**, a function like the one just mentioned will be requested.

Further all functions that have been set as init- or exit callback also need to exist to make the Pike level interface compile.

3.5.2 With CMOD

Using CMOD when writing the module makes it a little easier to define the Pike level interface. What needs to be done is in abstract:

- Add classes using a **PIKECLASS** statement. Everything that belongs to that class is then placed in between its curly brackets.
- Use **PIKEFUN ...(...)** to add a method to a class. Place it somewhere in between the curly brackets of the **PIKECLASS** statement of the class that the method is supposed to belong to.
- Use **CVAR ...** to add a private object variable. Place the variable declaration in the class that it is supposed to belong to.

Note that even though we use CMOD, the functions **pike_module_init()** and **pike_module_exit()** need to exist. They can be kept empty though.

As we can see CMOD is very convenient since the design can easily be translated to an implementation. If we for instance have a UML design model the mapping from the most basic UML components to CMOD is straight forward. As it can be seen in figure 5.2, UML components in this particular case have intuitive CMOD equivalents, so the translation is easy.



Figure 2: A class modeled with UML and its CMOD correspondence

3.6 Write the Functions

Once the classes and methods have been defined and mapped one still needs to do the 'real' implementation of the module. Since one common reason for writing a C module is to take advantage of already existing C libraries, much of the implementation has, in many cases, already been done. However, what always needs to be done is to fit the interface of the C library to the Pike interface of the module in a good way. Again, there are some differences what needs to be done, depending on if we are using CMOD or not.

3.6.1 Without CMOD

We have stubs for the functions, so then we can go on writing the code. What needs to be done in each function that implements a method is in abstract:

1. Read the parameters from the stack. The argument `args` specifies how many parameters that should be read. Pointers to the parameters can be found in **Pike_sp[-args]**, **Pike_sp[1-args]**, **Pike_sp[2-args]** and so on..
2. Make function calls, assignments, and so on until we have the desired result, i.e. until the return value is right and all variables in the storage have their desired value. In other words, write the actual program.
3. Pop the parameters from the stack and push the return value from 2) to the stack.

As we can see all input to and output from the functions is passed through the stack.

3.6.2 With CMOD

If we are using CMOD, then we have our skeleton to start writing the code. That is because CMOD generates some code that accesses the stack for us, so we can simply go ahead and write the implementation of the methods in plain C. In the end of each method (or should we call it function, since that is what it actually is) a **RETURN** statement should be placed. That **RETURN** removes the arguments from the stack and pushes the return value to it.

There are however two special cases where some extra code has to be written. The first one is when we have a method with optional arguments. Then CMOD does not manage the stack for us, instead we will have to check the `args` parameter and if the arguments exist, fetch them from the stack, just the same way as it is done when writing modules without CMOD.

The other special case is when we implement a method of type void. At the end of that method we can not use **RETURN** since there is nothing to return. Instead a **pop_n_elems(args)** needs to be done in order to keep the stack up to date.

3.7 Error Handling

One important thing the programmer that implements a module has to take care of is error handling. As we said earlier the module implementor's main task is to make the pieces fit each other. In order for that to work the input (and output) parameters must be valid. Since Pike does not provide adequate type checking, each module has to do that itself. The absolute minimum of error handling is to check the arguments in the implementation of each method.

3.7.1 Argument Checking without CMOD

As stated above, arguments must be checked in each method. What needs to be checked is that the number of arguments that the method is invoked with is valid, and that all the arguments that the method is invoked with have the right type. If anyone of these checks give a negative result an exception should be thrown.

3.7.2 Argument Checking with CMOD

CMOD provides automatic type checking and therefore reduces the amount error checking code a programmer must write manually. CMOD generates code for

argument checking in the implementation of the methods. The types of the arguments and the number of arguments with which the method is invoked is checked, and if something is wrong, the Pike exception handler is invoked.

The special case is if we have methods with optional argument(s). Then the arguments need to be checked manually in the same way as when CMOD is not used.

3.7.3 Throwing Exceptions

If an error of exceptional kind occurs, then an exception should be thrown. But before we can invoke the exception handler, we need to be certain that we do not hold any memory that will not be released once the exception handler is called. If we do not release all such memory before we call the exception handler there is a great risk that the method leaks memory. If the exceptional event in any way broke the inner state of an object, then it is probably a good idea to restore it before invoking the exception handler. When we have assured our selves that it is safe to call the exception handler, it can be called using **Pike_error()**.

3.8 Init and Exit Routines

To assure that the initial state of an object always is the desired, each class need to have a proper initialization routine. It is not safe to only set initial values in the **create()** method of a class since this method is not certain to be run. This is the case when a class is being inherited. Therefore, initialization of classes is required to be done in the initialization callback. So we need to add an initialization for each class in the module.

If needed, proper exit routines should also be written. Their task is to do cleanup before the object is garbage collected.

3.9 An Example - The Rev Module

This is a very simple C module. This text contains extracts of its code. The whole source code file can be found at the end of this appendix. A CMOD implementation of this module does not exist but some extracts of the code has been translated to CMOD.

The module has one class called **ReversedString**. This class has two methods: the mandatory **create()** method, and a method **write()**. The **ReversedString** class takes the 49 first characters of a string, reverses the string, and stores it as an object

variable. A call to the method **write()** writes the content of the reversed string **nr_of_times** times to the screen. As we can see the module is totally useless, it is just an example to illustrate how a module is written.

3.10 Define the Pike Level Interface

3.10.1 Without CMOD

As stated earlier Pike needs the function **pike_module_init()** and the function **pike_module_exit()** to recognize the module as a module.

Since the ReversedString module is that simple the **pike_module_init()** function was just the following few lines:

```
struct data{
    char buf[50];
}

void pike_module_init(void)
{
    start_new_program();
    ADD_STORAGE(struct data);
    ADD_FUNCTION("create", reversed_string_create,
                tFunc(tString, tVoid), 0);
    ADD_FUNCTION("write", reversed_string_write,
                tFunc(tInt, tVoid), 0);
    set_init_callback(init_reversed_string);
    set_exit_callback(exit_reversed_string);
    end_class("ReversedString", 0);
}
```

The effect of each line of code is described below.

```
start_new_program();
```

Tells the system that this is the beginning of a new class.

```
ADD_STORAGE (struct data);
```

Creates storage for a struct data. This is the first **ADD_STORAGE()** in the class. As a consequence of this the struct data is accessed through the global variable **Pike_fp->current_storage** which points directly to the struct.

```
ADD_FUNCTION("create", reversed_string_create,  
            tFunc(tString, tVoid),0);
```

Adds the Pike level **create()** method, or in other words the constructor, of the class and maps it to the **reversed_string_create()** C level function. The first parameter of **ADD_FUNCTION()** is the name of the Pike level method, the second parameter is the underlying C level function to which it is mapped, that means the low level function that is called when the high level method is invoked. The third parameter specifies what data types the Pike level method's arguments and return values are. The fourth argument specifies whether the Pike level method is static or public or...

```
ADD_FUNCTION("write", reversed_string_write,  
            tFunc(tInt, tVoid), 0);
```

This line adds the **write()** method. It works in the same way as the addition of the create method.

```
set_init_callback(init_reversed_string);
```

This line sets the init callback. Each time a ReversedString object is initialized **init_reversed_string()** will be run.

```
set_exit_callback(exit_reversed_string);
```

This line sets the exit callback. The **exit_reversed_string()** function will be run each time a ReversedString object is about to be garbage collected.

```
end_class("ReversedString", 0);
```

This function call is the terminating mark for the class that started at the **start_new_program()** call ends here, and that the name of that class is 'ReversedString'.

So that was the whole **pike_module_init()**. The module also needs a **pike_module_exit()** function. In this case there is nothing to be done in that function. Still it needs to exist, and that is why there is such an empty function.

3.10.2 With CMOD

The ReversedString realized with CMOD instead looks as follows:

```
PIKECLASS ReversedString{
    CVAR char reversed_string[50];
    PIKEFUN void create(string input){
        /*write the implementation of
        create here */
    }
    PIKEFUN void write(int nr_of_times){
        /*write the implementation of
        write here */
    }
    INIT{
        /*write the init callback here*/
    }
    EXIT{
        /*write the exit callback here*/
    }
}

void pike_module_init(){}
void pike_module_exit(){}
```

As we can see the declaration is straight forward and do not need that much explanation. The **CVAR** call results in that storage is added for a fifty character long **char** array. **PIKEFUN** adds the method that follows right after the keyword. **INIT** and **EXIT** add the init- and exit callback respectively. Finally, **pike_module_init()** and **pike_module_exit** need to exist in order for the module to be recognized as a module, they can be kept empty though.

Now, having handled the 'mandatory' parts, let us take a look at the functions that implements the functionality of the module.

3.11 Write the Functions

This section contains some description of how the Rev module works.

3.11.1 Without CMOD

```
reversed_string_create()
```

This is the function that will be run upon creation of a ReversedString object, because we set that earlier in **pike_module_init()**. It does the following:

1. Checks whether the constructor was called with one argument, i.e. if `args == 1`, throws an exception if not
2. Checks whether the only argument is a **pike_string**, throws an exception if not.
3. Takes the 49 first characters, using the **Pike_fp->current_storage** pointer, reverses the order of them, and stores them in storage.
4. Removes one argument from the stack, i.e. does a **pop_n_elems(args)**

```
reversed_string_write()
```

This is just a function to generate some output in order to test if everything in the create function and the storage works. It does the following:

1. Checks the argument, i.e. check that `args == 1` and that the argument is an **int**.
2. If the argument check succeeded, then it fetches the argument from the stack, and reads the content of the storage.
3. Prints the content of the storage on the screen

The callback functions

The init and exit callback functions exist in this module, but since nothing needs to be done in them they are left empty. These functions need to take a **struct object*** as argument, or else warnings will be generated when the module is compiled.

3.11.2 With CMOD

```
PIKEFUN void create(string input){}
```

This is where the implementation of the **create()** method would be written. What it would do is:

1. Read the 49 first characters from **input**, reverse the order of them and copy them to **reversed_string**.
2. Pop the parameters from the stack,

```
PIKEFUN void write(int nr_of_times){}
```

This is then where the implementation of **write()** would be written. What it would do is:

1. Print the content of **reversed_string nr_of_times** times
2. Pop **args** elements from the stack

INIT and EXIT

In this case neither INIT nor EXIT need to contain anything, since nothing needs to be done upon initialization or exit of a **ReversedString** object. Therefore they are kept empty.

```

/**
 *File: reversed.c
 *Description: The code of the Rev module */

#include "global.h"

#include "interpret.h"
#include "svalue.h"
#include "stralloc.h"
#include "array.h"
#include "pike_macros.h"
#include "program.h"
#include "stralloc.h"
#include "object.h"
#include "pike_types.h"
#include "threads.h"
#include "dynamic_buffer.h"

/* This must be included last! */
#include "module_magic.h"

/*PLEASE NOTE THAT this code is not that good from a programming point
of view, it is only intended to serve as an example of a simple
C module*/

struct data{
    char buf[50];
};

static void reversed_string_create(INT32 args){
    struct pike_string *input;
    char *s;
    char tmpbuf[50];
    char retbuf[50];
    int i;

    int j = 0;

    if (args != 1){
        Pike_error("Method ReversedString->create() takes ONE argument");
    }

    /* a little bit messy, assigns s a pointer to the storage of the reversed string*/
    s = (char *)&((struct data *)Pike_fp->current_storage->buf);
    input = Pike_sp[-args].u.string;

    strncpy((char*)&tmpbuf, (char *)&input->str,49); /* copy the input string to a buffer*/
    tmpbuf[49]='\0';
    /* find the terminating null-character*/
    for(i = 0; i < 50; i++){
        if (tmpbuf[i]=='\0'){
            break;
        }
    }
    /*reverse the string, result in retbuf[]*/

```

```

while (i >= 0 && j < 50){
    i--;
    retbuf[j] = tmpbuf[i];
    j++;
}
retbuf[j] = '\0';
/* store the reversed string, and pop arguments from stack*/
strcpy(s, (char *)&retbuf);
pop_n_elems(args);
}

static void reversed_string_write(INT32 args){
    int i = 0;
    int nr_of_times;

    if (args != 1){
        Pike_error("Method ReversedString->create() takes ONE argument");
    }
    nr_of_times = Pike_sp[-args].u.integer;
    for(i = 0; i < nr_of_times; i++){
        printf("%s", (char *)&(((struct data *)Pike_fp->current_storage)->buf));
    }
    pop_n_elems(args);
}

static void init_reversed_string(struct object *o){}

static void exit_reversed_string(struct object *o){}

void pike_module_exit(void) {}

void pike_module_init(void)
{
    start_new_program(); /*begin of class*/

    ADD_STORAGE(struct data);
    /*add the methods create and write*/
    ADD_FUNCTION("create", reversed_string_create, tFunc(tString, tVoid), 0);
    ADD_FUNCTION("write", reversed_string_write, tFunc(tInt, tVoid), 0);
    set_init_callback(init_reversed_string);
    set_exit_callback(exit_reversed_string);
    end_class("ReversedString",0);
}

```