The Unofficial Pike Programming Language FAQ

Contents

I. About this document
II. General questions
III. Basics of Pike programming
IV. Advanced Pike programming
V. Pike's module library
VI. Credits

About this document

This is a completely unofficial FAQ list for the Pike programming language, an interpreted (in its current implementation), object-oriented language created by Fredrik Hübinette and made available under the GNU General Public License by Roxen Internet Software (of Sweden), which sponsors its development. This document is unofficial in the sense that its editor has no relationship, business or other, with Roxen Internet Software (RIS) nor with its staff, it has not been reviewed nor approved by RIS and does not comprise part of Pike nor of documentation authored at and provided by RIS. While it is my intent to provide accurate and up to date information, there is no warranty: this document is a volunteer effort, and a work in progress, and it may well fall out of sync with the development of Pike due to limitations of time and knowledge on my part.

This is a work in progress. You are viewing a severely incomplete, early version. When this document becomes a little more complete and (hopefully) useful, the above notice will be made more precise, though it will not change as to intent. All contributions, new questions as well as new or improved answers, are very much welcome, and will be given due credit.

I will try to keep this information up to date for the current "stable" Pike branch (7.2 at the time of this writing); most of it applies as well to 7.0.x. With 0.6x (provided with Roxen 1.3), you're pretty much on your own, look up the docs appropriate to that version.

General questions

1. **What is Pike?**

   Pike is an object-oriented, interpreted programming language with a syntax similar to Java and C, high-level data types, automatic memory management, highly efficient string handling, easy to use APIs for network and database programming, and several years of active development behind it. In the current implementation, Pike code is compiled at runtime to bytecode which is interpreted by a virtual machine. Pike's implementation is heavily optimized, and both the language and its libraries are under continuing development; its performance compares favorably to all scripting languages on the market.

2. **What OS's does Pike run on?**

   Pike is best supported on Solaris and Linux, but it runs on a wide variety of Unix or Unix-like operating systems. With some limitations (hearsay as far as this author is concerned) it can also be made to work on Windows NT (Win32). Work appears to be under way on supporting Win64 as well.

3. **What is Pike good for (pros and cons)?**

   Pike is great for mostly any programming task, but is at its best when you are able to make good use of its high-level datatypes and good string-processing and network I/O facilities. It is probably not optimal for low-level stuff, especially involving much bit-twiddling and byte-by-byte processing. I haven't heard of many device drivers written in Pike ;-)

   A somewhat subjective overview:
   PROS
   - easy to learn and use
   - well-tested in the implementation of a major application (Roxen web server), development sponsored by a stable company, won't go away anytime soon
   - free (as in GPL)
   - familiar C-like syntax for expressions and control statements
   - automatic memory management
   - high-level datatypes, including mappings (associative arrays), classes and objects
   - clean, no-nonsense OOP model, including multiple inheritance and operator overloading (but you can mostly ignore OOP if you want)
   - exception handling
   - no compile/link/run cycle, source code is compiled into bytecode and interpreted at runtime
   - dynamic loading of C modules compiled into shared libs
   - good and efficient string handling
   - a well-designed module system that both organizes Pike's standard library and can be easily used in your own projects
   - excellent handling of network I/O, with an easy to use API
   - convenient built-in support for event-driven programming via a native Pike event loop
   - many useful library modules in the standard distribution

- great performance, beats most other interpreted languages at most tasks

CONS

- Pike is an "implementation-defined" language, lacks a formal definition or standard
- language features and library APIs can and do change in new releases, without much notice
- official documentation is not up to date nor complete
- no native code compilers (yet)
- no facilities for storing or distributing byte-compiled code
- no IDE or source-level debugger (you probably won’t miss the former much though, YMMV)
- exception system is not complete yet, and not yet as useful as it might be
- not optimized for numeric processing
- much smaller user community than (say) Perl or Python’s (the most comparable languages)
- commercial backing, while it exists, is not as powerful as Java’s

4. **Does Pike support databases, graphics, GUI programming, strong crypto, ...**

   Yes, and quite a bit more:

   - Supported RDBMS’s include Mysql, mSQL, Postgres and Oracle. There is also an ODBC interface;
   - The `Image` module supports a wide variety of operations on raster graphics in all common image formats;
   - GUI programming for the X Window System is supported through an interface to the popular GTK library, including Gnome extensions; there is also an interface to the OpenGL API, including GLUT;
   - Pike’s Crypto Toolkit (now fully included in the standard distribution) supports some of the most popular block and stream ciphers (DES, IDEA, and CAST128; RC4), cryptographic hash functions (MD5 and SHA1), and the RSA public key algorithm;
   - Decoding and encoding of MIME messages is supported by a dedicated module;
   - The `Gz` module gives you access to zlib data (de)compression;
   - The `Calendar` module provides a sophisticated set of classes and methods for datetime calculations;
   - Several more specialized modules are in development.

5. **Who uses Pike and for what?**

   The major Pike application is Roxen Internet Software’s Roxen Web Server (GPL), and its proprietary extensions marketed under the name Roxen Platform. For more info, see their website. Roxen is implemented almost entirely in Pike, and is remarkably efficient and stable. It is also highly extensible, via user-supplied modules written in Pike. In fact, most of the Pike programming being done outside of RIS is probably in-house development of Roxen modules by the webserver’s users.

   Another major project is Caudium, a fork of Roxen based on a version currently being phased out by RIS, but with significant new developments.

   One project I don’t know much about is GIME; they state they will be doing their development in Pike, but no new "What’s New" items seem to have appeared on their website for quite a while.

Pike is, however, as I try to argue here, suitable for much more than a webserver extension language, and one of the top purposes of this document is to alert more programmers to its advantages.

Some sample code can be found on the Pike Community website; beware that many of the "hacks" are quite old and will not work under current versions of Pike without modification. Unfortunately, RIS is phasing out this website, and it has ceased to be updated for some time now. Hopefully, whatever is useful of its content will be incorporated into the Roxen Community site.

6. How do I get Pike?

If you decide you are serious about learning and using Pike, you most certainly want to get a recent version from RIS's public CVS repository. This isn't hard to do at all; if you don't have a CVS client on your system yet, and you are running any halfway-decent Linux distribution, just install CVS from your distribution's packages (they all provide it). I'm certain it's just as easy with any of the free BSD's. Check on Roxen Community for how to proceed from there. If you have a decent net connection, the whole process won't take more than a few minutes, plus the time to build Pike from source (half an hour or so on today's machines).

Building Pike from source is quite straightforward in most cases: note that to benefit from most of the modules that interface with external C code, you need to have installed the *development* versions of the corresponding C libs (such as `libmysqlclient`, GTK, the Gnome libs, etc.).

The benefits of doing this are that you'll be sure you have an up to date Pike, including all current bugfixes and feature enhancements; compiling it from source will allow you to have support for those C libraries (and their versions) that are available on *your* system, which Pike can support.

A possible downside is that, since Pike's development is proceeding at a rather fast rate, if you stick to the bleeding edge you will surely sooner or later run into some incompatibilities that will break code you already wrote. A safer compromise may be to stick to the "stable" branch of Pike (7.2 at the time of this writing), but update it regularly to benefit from bugfixes.

Of course, the branch in current development (7.3 nowadays) is also available for your perusal, should you feel adventurous.

A slightly easier alternative is to install Pike from your Linux distribution. If you are running Debian Gnu/Linux (as you should be ;-), Pike and Roxen are available in the main section (and are only an `apt-get` away); however, it may be worthwhile to check out Caudium's website for more up to date Pike packages, including extra modules developed by the Caudium Group and an apt'able repository of deb's.

From Martin Nilsson:
You can download pike for Win32 by downloading Roxen Webserver for Win32. It does not include GL, GTK and some of the "heavy" stuff that would add too much to the size of the WebServer distributions. We have at least once succeded in compiling pike on all the systems listed on this Roxen Community page.

UPDATE: the latest stable source tarball of Pike should be at ftp://ftp.roxen.com/pub/pike/latest-stable/, last I looked this was version 7.2.239, and binary packages for RedHat Linux, Solaris and NT were there as well.

7. **Is there an IDE for Pike?**

   No, currently there is none. However, Pike is so much easier and cleaner then many other languages that the only thing you are likely to miss is a class browser to ease the exploration of the many available (and somewhat poorly documented) library modules. Pike’s interactive mode (known as `Hilfe`, and launched by invoking pike with no filename arguments), is very helpful though (see below).

   For editing Pike code with syntax highlighting you might use the `pike.el` module for (X)Emacs, which is provided in the Pike distribution.

8. **Is it or will it be possible to compile Pike to native code?**

   Not anytime soon. However, to quote the `ANNOUNCE` file from the current distribution:

   > Pike is still under development and the goal is to incorporate those in future versions.
   > - No Pike native compiler or debugger available

   Read whatever you like into this...

   However, if performance is your concern, you are likely to find that interpreted Pike performs well enough for most of your needs. The overhead of runtime bytecode compilation usually matters only for medium to large sized programs that need to be launched often; for short scripts, startup time is negligible, while for a long-running application (such as a webserver) it’s unimportant, as you don’t need to restart it often.

9. **Where do I find more information?**

   The Pike homepage at RIS is your main starting point.

   To get started, an excellent resource on the "Pike way" is the Pike 7.0 tutorial.

   There are several versions of the basic Pike Reference Manual floating around; it is hard to determine which one might be the most up to date at a given time. Places to check are the docs section of the Pike home page, David Hedbor’s page of Pike documentation, and of course Fredrik Hübinette’s Pike page.

   You will almost certainly want to subscribe to the Pike mailing list, if you plan any serious development in Pike.

Basics of Pike programming

1. **What are Pike’s datatypes?**

   Pike has a powerful and clean type system, which provides you with a variety of high-level data types, plus the benefits of compile and run-time type checking. There are arrays, mappings (a.k.a. associative arrays or dictionaries in other languages), classes, objects, and functions are also a first-class data type.

For a start, variables in Pike must be declared, and *some* type information must be given in the declaration, e.g.:

```
int counter=1;
object foo;
object(Stdio.File) outfile;
Stdio.FILE infile;
function(int, string|void: mapping(string:string)) fun;
int|float x;
mixed whatever;
```

are all valid variable declarations in Pike (`object(Classname) foo;` and `Classname foo;` are equivalent). Note that you can declare a variable as a sort of "union" (e.g.`int|float`), specify (optionally) argument and return types for a function, index and value types for a mapping; declare an object as an instance of a specific class, etc.

You may also forfeit (most) compile-time type checking by declaring your variables as `mixed`, i.e. any type at all. This is not usually a good idea (although it works), except when you want to have a sort of function overloading: you can define functions that check the type of their arguments at runtime, and take different actions accordingly.

In more detail: Pike has two sorts of datatypes, the basic types (`int, float, string`) and the pointer types (`array, mapping, multiset, function, program, object`). Variables are passed by value in function calls, but the semantics of this is slightly different depending on whether you are dealing with a basic or pointer type: in the latter case, the "value" is in fact a "pointer" to the actual data object, which is *mutable*, i.e. can be modified "in place". For the basic types, although behind the scenes, they are also represented by references to complex data objects, these objects themselves are immutable (this includes strings, which are shared).

This should be explained a little better, and illustrated with one or two examples.

You can also declare named constants, as in

```
constant PI = 3.141593;
```

here you do not need to specify a type, since the assigned value should be available to the interpreter at the time it compiles your class, meaning it can figure out its type on its own. Named constants are class members just like variables and methods, and are also `public` by default.

2. **OOP features in Pike?**

   Pike's OO features are quite elegant and user-friendly. Learn to take advantage them to organize your code, you'll soon agree that OOP can really make sense, even for rather small projects. The discussion of Pike OOP in the reference manual is actually quite good and highly recommended; anyway, here's a quick recap.

For a start, every file of Pike code you wrote defines a class. A class is simply a data object, of type `program` to Pike's type system, which you use as a container for other data objects; these can be anything at all (that Pike supports), including variables of both basic and pointer types; other classes too.

Pike actually uses *two* keywords referring to classes, in different contexts: the datatype is `program`, as already said; and the `class` keyword is used when you define a class inside another one, i.e. you don't have to put each class definition in a separate file.

In particular, class members can and usually do include functions, which can directly access variables declared at top level of the same class (actually, of any enclosing classes as well), in addition to any arguments that are passed in the function call. But no function definitions (method definitions in OO lingo) are actually required in a class, you can just as well define classes you will use as mere containers for data (like structs or record types in other languages).

Once a class is defined, you can pass it around just like like any other data type, but you can't yet do anything useful with the stuff inside it (data members and methods). To do some work with it, you must first *clone* an object of this class (instantiate, for the more sophisticated). Usually this is done by calling the class as if it were a function, and storing the return value in a variable of type `object` (more specific typing is usually useful, as discussed elsewhere). What this does is it initializes all class variables, and makes the methods available for calling. You can then access the object's members (up to limitations imposed by type modifiers, if you cared to use any), via the *indexing* operator.

In other words, all member identifiers are by default `public`, and are available to any code that holds a handle to an object cloned from the given class. In practice, this usually looks somewhat like

```
  Foo bar = Foo(arg1, arg2);
  gazonk = bar->gurg(whatever);
  bar->crap = zonk();
```

etc. Here the first line clones an object of class Foo, passing `arg1, arg2` to the class constructor (more about this in a minute), and stores it in a variable declared as an object of type Foo; the second line calls a member function of this object (and stores the result); and the third stores the return value of some function call in a member variable of `bar`.

To have something interesting happen when an object of your class is cloned, you must define in it a *constructor*, called `create()`. It should return `void`, and can take any arguments you declare for it; in the function's body, you can do mostly whatever you please. Your constructor will be called automatically whenever an object of your class is cloned. Of course, standard library classes come with their ready-made constructors.

When you're done with an object and want to get rid of it, you can just destroy it by calling `destruct(foo);`. From there on, all references to this object (if any) become invalid. If you need to do some cleanup when an object is destroyed, define in your class a `void destroy()` method -- this is usually necessary only when the object manipulates some external resources.

TBC

3. The preprocessor -- HOWTO?

Pike's preprocessor works pretty much the same as a C preprocessor -- allowing you to use directives such as `#if`, `#ifdef`, `#define` and `#include` in the source code for your Pike programs. `#include <file.h>` by default searches for `file.h` in Pike's systemwide include directory; you can find out where that is by executing `pike --show-paths`. You will seldom, if ever, need to use this directive.

NOTE: do not confuse Pike include files with C include files that are also installed by Pike, and that are used when compiling Pike modules written in C.

One thing that makes using the preprocessor in Pike different is that Pike scripts are compiled at runtime. A way to exploit this is by using a file of `#defines` *in lieu* of a runtime config file for your app. This may not be very elegant, but it's a boon for those who (like me) are too lazy to write a 'real' config file parser.

Some bonus features of Pike's preprocessor:

- `#!` causes the remainder of the line to be ignored (for compatibility with Unix `#!` script magic)
- `#"Put some string here"` extends the string literal syntax by allowing strings with (real) newlines
- `#string "file.txt"` inserts the contents of `file.txt` at the directive's position, as a (quoted and properly escaped) string literal
- You can run the preprocesor on any string value at your program's runtime by calling `cpp()`.
- There is a preprocessor directive `#pike` that defines which version of pike you want to emulate. This should make your scripts invulnerable to pike updates. Example:

  ```
  #pike 7.0
  // Code that should be run as pike 7.0
  #pike 7.2
  // Here you came back a year later and
  // wanted to use
  // a pike 7.2 feature
  #pike 7.0
  // The rest of the program.
  ```

  (thanks again to Martin Nilsson).
- There is a `#charset` directive that allows you to specify in what character set the source file is written; a large number of charsets are supported, including all the iso8859-*, UTF-8 and Unicode (*where is a list?*).

The following preprocessor symbols are predefined in Pike 7.0:

macro_name	expansion
`__LINE__`	Current line number (starts on 1)
`__FILE__`	Current filename
`__DATE__`	Current date "MMM dd yyyy"
`__TIME__`	Current time "hh:mm:ss"
`__dumpdef(X)`	Dump definition of a `#define` (unreliable)
`__PIKE__`	1
`__VERSION__`	major.minor
`__MAJOR__`	major
`__MINOR__`	minor
`__BUILD__`	build
`__AUTO_BIGNUM__`	1 if bignums are enabled.
`__NT__`	1 if WIN32/WIN64.
`__amigaos__`	1 if AmigaOS.

In Pike 7.2 and 7.3 the following preprocessor symbols were added/changed:

macro_name	expansion
`__VERSION__`	Current version major.minor (`#pike`)
`__MAJOR__`	Current version major (`#pike`)
`__MINOR__`	Current version minor (`#pike`)
`__REAL_VERSION__`	major.minor
`__REAL_MAJOR__`	major
`__REAL_MINOR__`	minor
`__REAL_BUILD__`	build (Same as `__BUILD__` for symmetry)

Thanks to Henrik Grubbström for the above.

4. **How do I make my script directly executable?**

   On a Unix(-like) system, any Pike script that defines a `main()` function can be treated as an executable file, provided that the first line of the script starts with

```
#!/usr/local/bin/pike
```

or whatever is the full path to the Pike executable installed on your system. This works just like for e.g. shell scripts and many other interpreters. The Pike interpreter will ignore this line. I've heard of Unix systems that don't obey this convention -- I think you can find them somewhere in the Retrocomputing Museum ;-)

One often-used trick is to start a script with a line

```
#!/usr/bin/env pike
```

instead of the one above; this works around the requirement that the exact full path to the Pike executable be given in the "magic" line, by causing the current value of `PATH` to be searched (see the manpage for `env(1)`).

Trivia: a few times I tried by mistake to execute a Pike script that was missing this line. Funny things happened, like my X display getting strangely messed up. It turned out that when told to execute an ASCII text file with no `#!` magic, my Linux system tries to run it as a shell (`sh`) script (or maybe that's a `bash` feature). Well, the first nonempty line in that script was like `import "Foo";`. Look up the manpage for `import` to solve the riddle (it's part of the ImageMagick suite). Killing the script and typing `xrefresh` at the prompt gets things back to normal.

5. **Can I (or should I) spread my program across multiple files?**
6. **Does Pike have an interactive mode?**

   Yes, it's called 'Hilfe'; according to Martin Nilsson,

   > Hilfe stands for "Hubbes Incremental Lpc FrontEnd". Help in Swedish is "Hjälp".

   (Hilfe happens to be also German for 'help', and Pike is a descendant of the older LPC language); it is launched when you invoke Pike with no filename arguments. Once in Hilfe, type 'help' for a brief summary of available commands. Other than those, you can type in mostly any Pike expression or statement, for immediate evaluation. Input can be continued across multiple lines, line editing is provided via the `readline` library (same as used in `bash`), though with a few glitches.

7. **I made my script return a negative error code from main(), and now it won't exit until I kill it?**

   This is correct: a negative return value from `main()` tells Pike's master program that it should enter asynchronous mode, i.e. the event-processing loop. For this to be useful, you should have first set up some callouts and/or callbacks. Callouts are functions to be called in (approximately) a specified time, while callbacks will be called in response to some external event (data becoming available on an input stream, button pressed on a GUI widget, etc.). To learn how to do this, investigate for a start `Stdio.File.set_nonblocking()` and `call_out()`.

   BTW a negative return code is not meaningful in most OS's: AFAIK the error code returned from a program is `unsigned char` (0-255).

8. **What are Pike's file I/O facilities?**

The `Stdio` module.

Classes `Stdio.File`, `Stdio.FILE`.

TBC

9. **What are Pike's network I/O facilities?**

   The `Stdio.File` class has easy to use methods for handling client socket connections, it's about as simple as

   ```
   Stdio.File fd = Stdio.File();
   fd->connect("pike.roxen.com", 80);
   fd->write("GET / HTTP/1.0\r\n\r\n");
   string reply = fd->read();
   ```

   up to error checking etc. of course. To listen on a port, there's the `Stdio.Port` class, and there's a `Stdio.UDP` for (surprize) UDP communication. The details are pretty well covered in the documentation.

10. **Does Pike support C++/iostream style I/O?**

    Not that I know of. It wouldn't probably be hard to emulate most of those features in Pike, but it seems nobody has bothered.

11. **What are the facilities for processing strings?**

    Builtin string functions and operators.

    The `String` module.

    The `Regexp` module.

    TBC

12. **How do I access OS/Posix facilities?**

    Interfaces to these are implemented mostly as builtin Pike functions, look for them in the mapping returned by `all_constants()`. Most of what you may need is there, e.g. `alarm()`, `chroot()`, `getpid()`, `getpwnam()`, `signal()`, `uname()`, ... Of course some may be missing if you're running Pike on (say) a MS Windows system.

13. **Does Pike have multithreading?**

    Yes, if your OS supports it (well enough).

    TBC.

14. **How do I get started with database programming in Pike?**

    If by 'database programming' you mean working with an RDBMS (such as Mysql or Oracle), first learn a little about SQL -- that part you won't find here (*I might insert a few links sometime*).

OK, now that you've done that: Pike has a generic SQL module, called (appropriately) `Sql`, which is the preferred interface. Most of the real work is done by methods implemented in RDBMS-specific modules, such as `Oracle`, `Mysql` or `Sybase`. Which ones you will actually have support for depends on the (DB-vendor-provided) client libraries you actually have on your system, as mentioned elsewhere.

An easy way to initialize a connection to your database is by cloning an instance of `Sql.sql`, via a call such as

```
object db = Sql.sql("mysql://dbhost", database, user,
password);
```

or even

```
object db = Sql.sql("mysql://user:password@dbhost/database");
```

if you find this more convenient (substitute 'oracle' or whatever for 'mysql' if appropriate), and now you can use methods in `db`, like

```
array(mapping(string:string)) reply = db->query(query_string);
```

here `query_string` holds your SQL statement, and `reply` will hold the rows of the table returned by your query, as mappings keyed by column name.

If you expect your query might return a huge amount of data, consider using the `big_query()` method instead, and fetching the result rows one by one from the `object(Sql.result)` returned by this method.

Much of the description of the `Mysql` module to be found in the Pike reference carries over to the generic `Sql` module. Yes, it is a long-standing deficiency of the reference that it describes `Mysql` and not the generic interface. When you should need DBMS-specific functions not provided by the generic `Sql`, they are available as methods of the object `db->master_sql`.

NOTE: you will almost certainly need to use `db->quote()` to build your query strings; this is used for quoting (rather, escaping) string literals sent in queries to the DB. String quoting rules are, unfortunately, somewhat database-specific. Luckily, `Sql.sql` automatically picks the version appropriate to your DBMS.

Remember that before you're sure you got it right, you can always try it in Hilfe and see pretty quickly if anything goes wrong.

15. **How do I get started with GUI programming?**

If you are not already familiar with the GTK toolkit, look for a GTK tutorial somewhere around the GTK website. For a start, just get acquainted with the basic concepts of GTK's object and event model and have a look at one or two simple code examples in C. GTK maps very nicely to Pike's model of objects and callbacks, and you should find it quite easy to redo those examples in Pike, or create your own. The basic idea is that you use Pike's own event loop, which is entered by returning a negative value from `main()`.

Sometime in the (hopefully) near future I expect to put here a few commented short sample scripts. If you can contribute any of your own, please do.

16. **How do I get started with OpenGL programming?**

    Check out this article on the Roxen Community site.

Advanced Pike programming

1. **Does Pike have support for large integers (bignums)?**

   Yes, as large as will fit in your computer ;-) if it was compiled with support for the Gmp library (Gnu Multiple Precision IIRC). This happens automatically if a suitable version of this library is detected on your system at the time of building Pike. Any integer value that is not less than `0x80000000` (on a 32-bit computer) gets automatically converted to a bignum object.

   Usage of large integers is mostly transparent to the user. Beware though that the builtin `pow()` function returns `float`, even for integer arguments. Use `Gmp.pow()` instead, or write something like `int big = base->pow(exponent);`

   One might say that, if auto bignums are enabled in your Pike, integers actually become bignum objects in disguise, but with optimization for the case when the `int` value fits in a machine integer.

   NB. bitwise operators also work with bignum ints.

   Support for bignum rationals is not provided yet.

2. **Is ‘type’ a datatype? What is it for?**

   Yes, in recent versions of Pike (7.2 for sure) it is a new basic type. Values of type `type` are returned (at least) by the `_typeof()` function and the `typeof()` "special form".

   The former allows you to test the actual, runtime type of an expression. Like any function, `_typeof()` first evaluates its argument; `typeof()`, on the contrary, does not -- it is not actually a function but a keyword that masquerades as a function, and it returns the Pike interpreter’s current notion of the type of what the given expression *would* evaluate to, as a value of type `type`. I.e. fed with the name of a function it will return `function`, or possibly `function(int : void)` (or whatever) if such information is available. Given a function call, it will return (whatever it knows about) the type of the function’s return value, *but the call will not be performed.*

   Note however, that `type` is not (currently) a keyword, i.e. you cannot declare a variable as `type foo;` say. You *can* store type values in named constants or variables of type `mixed`, though.

3. **Does Pike allow for user-defined datatypes?**

   Well, obviously any class definition provides a user-defined (composite) datatype, and this is a most useful feature; but in addition, new features of Pike’s type system (see preceding question) allow you to play some more subtle tricks. A (perhaps silly) example would be (try this code):

```
constant boolean = typeof(0)|typeof(1);
boolean flag = 1;
int main( int argc, array(string) argv )
{
flag = (argc>1)?(int)argv[1]:flag;
write( "Type of flag is %O\n", typeof(flag) );
write( "Value of flag is %O\n", flag );
return 0;
}
```

A related point is that (since Pike 7.2.26 approx.) you can use `typedef` and `enum` declarations in Pike. As far as I can see, their usage and properties are quite alike those of the corresponding C language keywords.

4. **What about the ‘more advanced’ OO features?**

   Well, Pike has at least

   - multiple inheritance
   - data hiding
   - operator overloading
   - the option to have interface declarations separate from implementation
   - ...

   One relevant issue to keep in mind (from Henrik Grubbström):

   > Note that Pike’s typing system uses the "implements"/"looks-like" relation, and not the "inherits"/"is-a" relation. The typing system when comparing objects of different types, compares the types for the public (ie not static) symbols in the objects.

   TBC

5. **How do I extend Pike, or interface it to some C library?**
6. **Are there any other ways to modify Pike’s behavior?**
7. **Can I use mmap() and friends from Pike?**

   Not in Pike 7.2. An interface will be or already is provided in 7.3, the current development version. Note that this can be quite handy: since Pike strings are shared and cannot be altered "in place", changing a few bytes here and there in a string will result in multiple copies being allocated at least temporarily, which may be an issue if you’re dealing with a huge chunk of data.

8. **Can Pike use unix sockets for IPC?**

   Not directly AFAIK. However, unix sockets are used in an indirect way by several modules, when the underlying C libraries use them. Examples include Mysql (connections to a locally running Mysql daemon) and GTK (X protocol connections to a local display).

Pike’s module library

These are essentially all the modules you will find standard with Pike 7.2, except for a few I have omitted -- the latter being either modules for interfacing with the various database servers (Mysql, Oracle, etc.) which are normally accessed via `Sql`, those used internally and not designed for end user access (I presume that this is indicated by a module name that starts with an underscore), and several others that are obviously just stubs. My aim is to have at least a few words of comments for each, regarding purpose, status and usefulness. Of course, I do not mean to duplicate the reference documentation (where it exists), only to provide a quick "hint sheet".

Readers are invited (read: *encouraged*) to send in short, self-contained sample scripts illustrating usage of the individual modules. If appropriate, I will incorporate them into a small repository of Pike code samples I plan to set up. To keep me out of trouble, please indicate the origin of the code (if not written by you) and any possible usage restrictions (public domain will be assumed by default).

1. **ADT**
2. **Array**
3. **Cache**
4. **Calendar**
5. **CommonLog**
6. **Crypto**
7. **Debug**
8. **Filesystem**
9. **GDK**
10. **GL**
11. **GLU**
12. **GLUT**
13. **GTK**
14. **GDBM**
15. **Geography**
16. **Getopt**
17. **Gettext**
18. **Gmp**
19. **Gnome**
20. **Graphics**
21. **Gz**
22. **HTTPLoop**
23. **Image**
24. **Java**
25. **LR**
26. **Languages**
27. **Locale**
28. **MIME**
29. **Math**
30. **Mird**
31. **PDF**

32. **Parser**
33. **Perl**
34. **Pipe**
35. **Process**
36. **Protocols**
37. **Regexp**
38. **Remote**
39. **SANE**
40. **Sql**
41. **Ssleay**
42. **Standards**
43. **Stdio**
44. **String**
45. **Thread**
46. **Tools**
47. **Yabu**
48. **Yp**
49. **spider**

Credits

In addition to information gathered from the Pike mailing list (thanks to everybody) and misc. other sources, the following people have contributed corrections and/or additional information:

- Francesco Chemolli
- Martin Nilsson
- Henrik Grubbström
- Sten Eriksson

Accessed: by 4483 since February 20th.

Robert J. Budzynski

Last modified: 2001-12-03