Topic `doc.md`

LDoc, a Lua Documentation Tool

Introduction

LDoc is a second-generation documentation tool that can be used as a replacement for **LuaDoc**. It arose out of my need to document my own projects and only depends on the **Penlight** libraries.

It is mostly compatible with LuaDoc, except that certain workarounds are no longer needed. For instance, it is not so married to the idea that Lua modules should be defined using the `module` function; this is not only a matter of taste since this has been deprecated in Lua 5.2.

Otherwise, the output is very similar, which is no accident since the HTML templates are based directly on LuaDoc. You can ship your own customized templates and style sheets with your **own project**, however. You have an option to use Markdown to process the documentation, which means no ugly HTML is needed in doc comments. C/C++ extension modules may be documented in a similar way, although function names cannot be inferred from the code itself.

LDoc can provide integrated documentation, with traditional function comments, any documents in Markdown format, and specified source examples. Lua source in examples and the documents will be prettified.

Although there are a fair number of command-line options, the preferred route is to write a `config.ld` configuration file in Lua format. By convention, if LDoc is simply invoked as `ldoc .` it will read this file first. In this way, the aim is to make it very easy for end-users to build your documentation using this simple command.

Commenting Conventions

LDoc follows the conventions established by Javadoc and later by LuaDoc.

Only 'doc comments' are parsed; these can be started with at least 3 hyphens, or by a empty comment line with at least 3 hypens:

```
--- summary.
-- Description; this can extend over
-- several lines

-----------------
-- This will also do.
```

You can also use Lua block comments:

```
--[[--
 Summary. A description
 ...;
]]
```

Any module or script must start with a doc comment; any other files are ignored and a warning issued. The only exception is if the module starts with an explicit `module` statement.

All doc comments start with a summary sentence, that ends with a period or a question mark. An optional description may follow. Normally the summary sentence will appear in the module contents.

After this descriptive text, there will typically be *tags*. These follow the convention established by Javadoc and widely used in tools for other languages.

```
--- foo explodes text.
-- It is a specialized splitting operation on a string.
-- @param text the string
-- @return a table of substrings
function foo (text)
....
end
```

There are also ‘tparam’ and ‘treturn’ which let you **specify a type**:

```
-- @tparam string text the string
-- @treturn {string,...} a table of substrings
```

There may be multiple ‘param’ tags, which should document each formal parameter of the function. For Lua, there can also be multiple ‘return’ tags

```
--- solvers for common equations.
module("solvers", package.seeall)

--- solve a quadratic equation.
-- @param a first coeff
-- @param b second coeff
-- @param c third coeff
-- @return first root, or nil
-- @return second root, or imaginary root error
function solve (a,b,c)
    local disc = b^2 - 4*a*c
    if disc < 0 then
        return nil,"imaginary roots"
    else
       disc = math.sqrt(disc)
       return (-b + disc)/2*a,
              (-b - disc)/2*a
    end
end

...
```

This is the common module style used in Lua 5.1, but it’s increasingly common to see less ‘magic’ ways of creating modules in Lua. Since `module` is deprecated in Lua 5.2, any future-proof documentation tool needs to handle these styles gracefully:

```
--- a test module
-- @module test

local test = {}

--- first test.
function test.one()
...
end

...

return test
```

Here the name of the module is explicitly given using the ‘module’ tag. If you leave this out, then LDoc will infer the name of the module from the name of the file and its relative location in the filesystem; this logic is also used for the `module(...)` idiom. (How this works and when you need to provide extra information is discussed later.)

It is common to use a local name for a module when declaring its contents. In this case the ‘alias’ tag can tell LDoc that these functions do belong to the module:

```
--- another test.
-- @module test2
-- @alias M

local M = {}

-- first test.
function M.one()
```

```
..
end

return M
```

`M` and `_M` are used commonly enough that LDoc will recognize them as aliases automatically, but 'alias' allows you to use any identifier.

LDoc tries to deduce the function name and the formal parameter names from examining the code after the doc comment. It also recognizes the 'unsugared' way of defining functions as explicit assignment to a variable:

```
--- second test.
M.two = function(...) ... end
```

Apart from exported functions, a module usually contains local functions. By default, LDoc does not include these in the documentation, but they can be enabled using the `--all` flag. They can be documented just like 'public' functions:

```
--- it's clear that boo is local from context.
local function boo(...) .. end

local foo

--- we need to give a hint here for foo
-- @local here
function foo(...) .. end
```

Modules can of course export tables and other values. The classic way to document a table looks like this:

```
--- a useful table of constants
-- @field alpha first correction
-- @field beta second correction
-- @field gamma fudge factor
-- @table constants
```

Here the kind of item is made explicit by the 'table' tag; tables have 'fields' in the same way as functions have parameters.

This can get tedious, so LDoc will attempt to extract table documentation from code:

```
--- a useful table of constants
M.constants = {
    alpha = 0.23, -- first correction
    beta = 0.443, -- second correction
    gamma = 0.01  -- fudge factor
}
```

The rule followed here is `NAME = <table-constructor>`. If LDoc can't work out the name and type from the following code, then a warning will be issued, pointing to the file and location.

Another kind of module-level type is 'field', such as follows:

```
--- module version.
M._VERSION = '0.5'
```

That is, a module may contain exported functions, local functions, tables and fields.

When the code analysis would lead to the wrong type, you can always be explicit.

```
--- module contents.
-- @field _CONTENTS
M._CONTENTS = {constants=true,one=true,...}
```

The order of tags is not important, but as always, consistency is useful. Tags like 'param' and 'return' can be specified multiple times, whereas a type tag like 'function' can only occur once in a comment. The basic rule is

that a single doc comment can only document one entity.

Since 1.3, LDoc allows the use of *colons* instead of @.

```
--- a simple function.
-- string name person's name
-- int: age age of person
-- !person: person object
-- treturn: ?string
-- function check(name,age)
```

However, you must either use the `--colon` flag or set `colon=true` in your `config.ld`.

In this style, types may be used directly if prefixed with '!' or '?' (for type-or-nil)

(see **colon.lua**, rendered **here**)

By default, LDoc will process any file ending in '.lua' or '.luadoc' in a specified directory; you may point it to a single file as well. A 'project' usually consists of many modules in one or more *packages*. The generated `index.html` will point to the generated documentation for each of these modules.

If only one module or script is documented for a project, then the `index.html` generated contains the documentation for that module, since an index pointing to one module would be redundant.

LDoc has a two-layer hierarchy; underneath the project, there are modules, scripts, classes (containing code) and examples and 'topics' (containing documentation). These then contain items like functions, tables, sections, and so forth.

If you want to document scripts, then use `@script` instead of `@module`. New with 1.4 is `@classmod` which is a module which exports a single class.

See References

The tag 'see' is used to reference other parts of the documentation, and 'usage' can provide examples of use; there can be multiple such tags:

```
---------
-- split a string in two.
-- @param s the string
-- @param delim the delimiter (default space)
-- @return first part
-- @return second part
-- @usage local hello,world = split2("hello world")
-- @see split
funtion split2(s,delim) .. end
```

Here it's assumed that 'split' is a function defined in the same module. If you wish to link to a function in another module, then the reference has to be qualified.

References to methods use a colon: `myclass:method`; this is for instance how you would refer to members of a `@type` section.

The example at `tests/complex` shows how @see references are interpreted:

```
complex.util.parse
complex.convert.basic
complex.util
complex.display
complex
```

You may of course use the full name of a module or function, but can omit the top-level namespace – e.g. can refer to the module `util` and the function `display.display_that` directly. Within a module, you can directly use a function name, e.g. in `display` you can say `display_this`.

What applies to functions also applies to any module-level item like tables. New module-level items can be defined and they will work according to these rules.

If a reference is not found within the project, LDoc checks to see if it is a reference to a Lua standard function

or table, and links to the online Lua manual. So references like 'table.concat' are handled sensibly.

References may be made inline using the `@{ref}` syntax. This may appear anywhere in the text, and is more flexible than `@see`. In particular, it provides one way to document the type of a parameter or return value when that type has a particular structure:

```
------
-- extract standard variables.
-- @param s the string
-- @return @{stdvars}
function extract_std(s) ... end

------
-- standard variables.
-- Use @{extract_std} to parse a string containing variables,
-- and @{pack_std} to make such a string.
-- @field length
-- @field duration
-- @field viscosity
-- @table stdvars
```

`@{ref}` is very useful for referencing your API from code samples and readme text.

The link text can be changed from the default by the extended syntax `@{ref|text}`.

You can also put references in backticks, like `\`stdvars\``. This is commonly used in Markdown to indicate code, so it comes naturally when writing documents. It is controlled by the configuration variable `backtick_references` or the `backtick` format; the default is `true` if you use Markdown in your project, but can be specified explicitly in `config.ld`.

To quote such references so they won't be expanded, say @{\ref}.

Custom @see References

It's useful to define how to handle references external to a project. For instance, in the **luaposix** project we wanted to have `man` references to the corresponding C function:

```
------------
-- raise a signal on this process.
-- @see raise(3)
-- @int nsig
-- @return integer error cod
function raise (nsig)
end
```

These see references always have this particular form, and the task is to turn them into online references to the Linux manpages. So in `config.ld` we have:

```
local upat = "http://www.kernel.org/doc/man-pages/online/pages/man%s/%s.%s.html"

custom_see_handler('^(%a+)%((%d)%)$',function(name,section)
    local url = upat:format(section,name,section)
    local name = name .. '(' ..section..')'
    return name, url
end)
```

'^(%a+)%((%d)%)$' both matches the pattern and extracts the name and its section. Then it's a simple matter of building up the appropriate URL. The function is expected to return *link text* and *link source* and the patterns are checked before LDoc tries to resolve project references. So it is best to make them match as exactly as possible.

Module Tags

LDoc requires you to have a module doc comment. If your code style requires license blocks that might look like doc comments, then set `boilerplate=true` in your configuration and they will be skipped.

This comment does not have to have an explicit `@module` tag and LDoc continues to respect the use of `module()`.

There are three types of 'modules' (i.e. 'project-level'); `module`, a library loadable with `require()`, `script`, a program, and `classmod` which is a class implemented in a single module.

There are some tags which are only useful in module comments: `author`,`copyright`, `license` and `release`. These are presented in a special **Info** section in the default HTML output.

The `@usage` tag has a somewhat different presentation when used in modules; the text is presented formatted as-is in a code font. If you look at the script `ldoc` in this documentation, you can see how the command-line usage is shown. Since coding is all about avoiding repetition and the out-of-sync issues that arise, the `@usage` tag can appear later in the module, before a long string. For instance, the main script of LDoc is **ldoc.lua** and you will see that the usage tag appears on line 36 before the usage string presented as help.

`@export` is another module tag that is usually 'detached'. It is for supporting modules that wish to explicitly export their functions **at the end**. In that example, both `question` and `answer` are local and therefore private to the module, but `answer` has been explicitly exported. (If you invoke LDoc with the `-a` flag on this file, you will see the documentation for the unexported function as well.)

`@set` is a powerful tag which assigns a configuration variable to a value *just for this module*. Saying `@set no_summary=true` in a module comment will temporarily disable summary generation when the template is expanded. Generally configuration variables that effect template expansion are modifiable in this way. For instance, if you wish that the contents of a particular module be sorted, then `@set sort=true` will do it *just* for that module.

Sections

LDoc supports *explicit* sections. By default, the implicit sections correspond to the pre-existing types in a module: 'Functions', 'Tables' and 'Fields' (There is another default section 'Local Functions' which only appears if LDoc is invoked with the `--all` flag.) But new sections can be added; the first mechanism is when you **define a new type** (say 'macro'). Then a new section ('Macros') is created to contain these types.

There is also a way to declare ad-hoc sections using the `@section` tag. The need occurs when a module has a lot of functions that need to be put into logical sections.

```
--- File functions.
-- Useful utilities for opening foobar format files.
-- @section file

--- open a file
...

--- read a file
...

--- Encoding operations.
-- Encoding foobar output in different ways.
-- @section encoding

...
```

A section doc-comment has the same structure as a normal doc-comment; the summary is used as the new section title, and the description will be output at the start of the function details for that section; the name is not used, but must be unique.

Sections appear under 'Contents' on the left-hand side. See the **winapi** documentation for an example of how this looks.

Arguably a module writer should not write such very long modules, but it is not the job of the documentation tool to limit the programmer!

A specialized kind of section is `type`: it is used for documenting classes. The functions (or fields) within a type section are considered to be the methods of that class.

```
--- A File class.
-- @type File
```

```
....
--- get the modification time.
-- @return standard time since epoch
function File:mtime()
...
end
```

(In an ideal world, we would use the word 'class' instead of 'type', but this would conflict with the LuaDoc `class` tag.)

A section continues until the next section is found, `@section end`, or end of file.

You can put items into an implicit section using the @within tag. This allows you to put adjacent functions in different sections, so that you are not forced to order your code in a particular way.

With 1.4, there is another option for documenting classes, which is the top-level type `classmod`. It is intended for larger classes which are implemented within one module, and the advantage that methods can be put into sections.

Sometimes a module may logically span several files, which can easily happen with large There will be a master module with name 'foo' and other files which when required add functions to that module. If these files have a @submodule tag, their contents will be placed in the master module documentation. However, a current limitation is that the master module must be processed before the submodules.

See the `tests/submodule` example for how this works in practice.

Differences from LuaDoc

LDoc only does 'module' documentation, so the idea of 'files' is redundant.

One added convenience is that it is easier to name entities:

```
------------
-- a simple module.
-- (LuaDoc)
-- @class module
-- @name simple
```

becomes:

```
------------
-- a simple module.
-- (LDoc)
-- @module simple
```

This is because type names (like 'function', 'module', 'table', etc) can function as tags. LDoc also provides a means to add new types (e.g. 'macro') using a configuration file which can be shipped with the source. If you become bored with typing 'param' repeatedly then you can define an alias for it, such as 'p'. This can also be specified in the configuration file.

LDoc will also work with C/C++ files, since extension writers clearly have the same documentation needs as Lua module writers.

LDoc allows you to attach a *type* to a parameter or return value with `tparam` or `treturn`, and gives the documenter the option to use Markdown to parse the contents of comments. You may also include code examples which will be prettified, and readme files which will be rendered with Markdown and contain prettified code blocks.

Adding new Tags

LDoc tries to be faithful to LuaDoc, but provides some extensions. Aliases for tags can be defined, and new types declared.

```
--- zero function. Two new ldoc features here; item types
-- can be used directly as tags, and aliases for tags
```

```
-- can be defined in config.ld.
-- @function zero_fun
-- @p k1 first
-- @p k2 second
```

Here an alias for 'param' has been defined. If a file `config.ld` is found in the source, then it will be loaded as Lua data. For example, the configuration for the above module provides a title and defines an alias for 'param':

```
title = "testmod docs"
project = "testmod"
alias("p","param")
```

Extra tag *types* can be defined:

```
new_type("macro","Macros")
```

And then used as any other type:

```
-----
-- A useful macro. This is an example of a custom type.
-- @macro first_macro
-- @see second_function
```

This will also create a new module section called 'Macros'.

If your new type has arguments or fields, then specify the name:

```
new_type("macro","Macros",false,"param")
```

(The third argument means that this is not a *project level* tag)

Then you may say:

```
-----
-- A macro with arguments.
-- @macro second_macro
-- @param x the argument
```

And the arguments will be displayed under the subsection 'param'

Inferring more from Code

The qualified name of a function will be inferred from any `function` keyword following the doc comment. LDoc goes further with this kind of code analysis, however.

Instead of:

```
--- first table.
-- @table one
-- @field A alpha
-- @field B beta
M.one = {
    A = 1,
    B = 2;
}
```

you can write:

```
--- first table
-- @table one
M.one = {
```

```
    A = 1, -- alpha
    B = 2; -- beta
}
```

Simularly, function parameter comments can be directly used:

```
------------
-- third function. Can also provide parameter comments inline,
-- provided they follow this pattern.
function mod1.third_function(
    alpha, -- correction A
    beta, -- correction B
    gamma -- factor C
    )
    ...
end
```

As always, explicit tags can override this behaviour if it is inappropriate.

Extension modules written in C

LDoc can process C/C++ files:

```
/***
Create a table with given array and hash slots.
@function createtable
@param narr initial array slots, default 0
@param nrec initial hash slots, default 0
@return the new table
*/
static int l_createtable (lua_State *L) {
....
```

Both `/**` and `///` are recognized as starting a comment block. Otherwise, the tags are processed in exactly the same way. It is necessary to specify that this is a function with a given name, since this cannot be reliably be inferred from code. Such a file will need a module comment, which is treated exactly as in Lua.

An unknown extension can be associated with a language using a call like `add_language_extension('lc','c')` in `config.ld`. (Currently the language can only be 'c' or 'lua'.)

An LDoc feature which is particularly useful for C extensions is *module merging*. If several files are all marked as `@module lib` then a single module `lib` is generated, containing all the docs from the separate files. For this, use `merge=true`.

See **mylib.c** for the full example.

Moonscript Support

1.4 introduces basic support for **Moonscript**. Moonscript module conventions are just the same as Lua, except for an explicit class construct. **list.moon** shows how `@classmod` can declare modules that export one class, with metamethods put explicitly into a separate section.

Basic Usage

For example, to process all files in the 'lua' directory:

```
$ ldoc lua
output written to doc/
```

Thereafter the `doc` directory will contain `index.html` which points to individual modules in the `modules` subdirectory. The `--dir` flag can specify where the output is generated, and will ensure that the directory exists. The output structure is like LuaDoc: there is an `index.html` and the individual modules are in the `modules` subdirectory. This applies to all project-level types, so that you can also get `scripts`, `examples` and

`topics` directories.

If your modules use `module(...)` then the module name has to be deduced. If `ldoc` is run from the root of the package, then this deduction does not need any help – e.g. if your package was `foo` then `ldoc foo` will work as expected. If we were actually in the `foo` directory then `ldoc -b .. .` will correctly deduce the module names. An example would be generating documentation for LuaDoc itself:

```
$ ldoc -b .. /path/to/luadoc
```

Without the `-b` setting the base of the package to the *parent* of the directory, implicit modules like `luadoc.config` will be incorrectly placed in the global namespace.

For new-style modules, that don't use `module()`, it is recommended that the module comment has an explicit `@module PACKAGE.NAME`. If it does not, then `ldoc` will still attempt to deduce the module name, but may need help with `--package/-b` as above.

A special case is if you simply say 'ldoc .'. Then there *must* be a `config.ld` file available in the directory, and it can specify the file:

```
file = "mymod.lua"
title = "mymod documentation"
description = "mymod does some simple but useful things"
```

`file` can of course point to a directory, just as with the `--file` option. This mode makes it particularly easy for the user to build the documentation, by allowing you to specify everything explicitly in the configuration.

In `config.ld`, `file` may be a Lua table, containing file names or directories; if it has an `exclude` field then that will be used to exclude files from the list, for example `{'examples', exclude = {'examples/slow.lua'}}`.

A particular configuration file can be specified with the `-c` flag. Configuration files don't *have* to contain a `file` field, but in that case LDoc does need an explicit file on the command line. This is useful if you have some defaults you wish to apply to all of your docs.

Markdown Support

`format = 'markdown'` can be used in your `config.ld` and will be used to process summaries and descriptions; you can also use the `-f` flag. This requires a markdown processor. LDoc knows how to use:

- **markdown.lua** a pure Lua processor by Niklas Frykholm. For convenience, LDoc comes with a copy of markdown.lua.
- **lua-discount**, a faster alternative (installed with `luarocks install lua-discount`). lua-discount uses the C **discount** Markdown processor which has more features than the pure Lua version, such as PHP-Extra style tables.
- **lunamark**, another pure Lua processor, faster than markdown, and with extra features (`luarocks install lunamark`).

You can request the processor you like with `format = 'markdown|discount|lunamark|plain|backticks'`, and LDoc will attempt to use it. If it can't find it, it will look for one of the other markdown processors; the original `markdown.lua` ships with LDoc, although it's slow for larger documents.

Even with the default of 'plain' some minimal processing takes place, in particular empty lines are treated as line breaks. If the 'backticks' formatter is used, then it's equivalent to using 'process_backticks=trueinconfig.ldand backticks will be expanded into documentation links like@{ref}and converted intoref otherwise.

This formatting applies to all of a project, including any readmes and so forth. You may want Markdown for this 'narrative' documentation, but not for your code comments. `plain=true` will switch off formatting for code.

Processing Single Modules

`--output` can be used to give the output file a different name. This is useful for the special case when a single module file is specified. Here an index would be redundant, so the single HTML file generated contains the module documentation.

```
$ ldoc mylib.lua --> results in doc/index.html
$ ldoc --output mylib mylib.lua --> results in doc/mylib.html
```

```
 $ ldoc --output mylib --dir html mylib.lua --> results in html/mylib.html
```

The default sections used by LDoc are 'Functions', 'Tables' and 'Fields', corresponding to the built-in types 'function', 'table' and 'field'. If `config.ld` contains something like `new_type("macro","Macros")` then this adds a new section 'Macros' which contains items of 'macro' type – 'macro' is registered as a new valid tag name. The default template then presents items under their corresponding section titles, in order of definition.

Getting Help about a Module

There is an option to simply dump the results of parsing modules. Consider the C example `tests/example/mylib.c`':

```
$ ldoc --dump mylib.c
----
module: mylib   A sample C extension.
Demonstrates using ldoc's C/C++ support. Can either use /// or /*** */ etc.

function        createtable(narr, nrec)
Create a table with given array and hash slots.
narr     initial array slots, default 0
nrec     initial hash slots, default 0

function        solve(a, b, c)
Solve a quadratic equation.
a        coefficient of x^2
b        coefficient of x
c        constant
return  {"first root","second root"}
```

This is useful to quickly check for problems; here we see that `createable` did not have a return tag.

LDoc takes this idea of data dumping one step further. If used with the `-m` flag it will look up an installed Lua module and parse it. If it has been marked up in LuaDoc-style then you will get a handy summary of the contents:

```
$ ldoc -m pl.pretty
----
module: pl.pretty       Pretty-printing Lua tables.
* read(s) - read a string representation of a Lua table.
* write(tbl, space, not_clever) - Create a string representation of a Lua table.

* dump(t, ...) - Dump a Lua table out to a file or stdout.
```

You can specify a fully qualified function to get more information:

```
$ ldoc -m pl.pretty.write

function        write(tbl, space, not_clever)
create a string representation of a Lua table.
tbl      {table} Table to serialize to a string.
space    {string} (optional) The indent to use.
              Defaults to two spaces.
not_clever       {bool} (optional) Use for plain output, e.g {['key']=1}.
              Defaults to false.
```

LDoc knows about the basic Lua libraries, so that it can be used as a handy console reference:

```
$> ldoc -m assert

function        assert(v, message)
Issues an error when the value of its argument `v` is false (i.e.,
 nil or false); otherwise, returns all its arguments.
`message` is an error
```

```
 message; when absent, it defaults to "assertion failed!"
v
message
```

Thanks to Mitchell's **Textadept** project, LDoc has a set of `.luadoc` files for all the standard tables, plus **LuaFileSystem** and **LPeg**.

```
$> ldoc -m lfs.lock

function        lock(filehandle, mode, start, length)
Locks a file or a part of it.
This function works on open files; the file
 handle should be specified as the first argument. The string mode could be
 either r (for a read/shared lock) or w (for a write/exclusive lock). The
 optional arguments start and length can be used to specify a starting point
 and its length; both should be numbers.
 Returns true if the operation was successful; in case of error, it returns
 nil plus an error string.
filehandle
mode
start
length
```

Anatomy of a LDoc-generated Page

winapi can be used as a good example of a module that uses extended LDoc features.

The *navigation section* down the left has several parts:

- The project name ('project' in the config)
- A project description ('description')
- ''Contents'' of the current page
- ''Modules'' listing all the modules in this project

Note that `description` will be passed through Markdown, if it has been specified for the project. This gives you an opportunity to make lists of links, etc; any '##' headers will be formatted like the other top-level items on the navigation bar.

'Contents' is automatically generated. It will contain any explicit sections, if they have been used. Otherwise you will get the usual categories: 'Functions', 'Tables' and 'Fields'.

'Modules' will appear for any project providing Lua libraries; there may also be a 'Scripts' section if the project contains Lua scripts. For example, **LuaMacro** has a driver script `luam` in this section. The **builtin** module only defines macros, which are defined as a *custom tag type*.

The *content section* on the right shows:

- The module summary and description
- The contents summary, per section as above
- The detailed documentation for each item

As before, the description can use Markdown. The summary contains the contents of each section as a table, with links to the details. This is where the difference between an item's summary and an item's description is important; the first will appear in the contents summary. The item details show the item name and its summary again, followed by the description. There are then sections for the following tags: 'param', 'usage', 'return' and 'see' in that order. (For tables, 'Fields' is used instead of 'Parameters' but internally fields of a table are stored as the 'param' tag.)

By default, the items appear in the order of declaration within their section. If `sort=true` then they will be sorted alphabetically. (This can be set per-module with **@set**.)

You can of course customize the default template, but there are some parameters that can control what the template will generate. Setting `one=true` in your configuration file will give a *one-column* layout, which can be easier to use as a programming reference. You can suppress the contents summary with `no_summary`.

Customizing the Page

A basic customization is to override the default UTF-8 encoding using `charset`. For instance, Brazillian

software would find it useful to put `charset='ISO-8859-1'` in `config.ld`, or use the @charset tag for individual files.

Setting `no_return_or_parms` to `true` will suppress the display of 'param' and 'return' tags. This may appeal to programmers who dislike the traditional @tag soup xDoc style and prefer to comment functions just with a description. This is particularly useful when using Markdown in a stylized way to specify arguments:

```
---------
-- This extracts the shortest common substring from the strings _s1_ and _s2_
function M.common_substring(s1,s2)
```

Here I've chosen to italicise parameter names; the main thing is to be consistent.

This style is close to the Python **documentation standard**, especially when used with `no_summary`.

It is also very much how the Lua documentation is ordered. For instance, this configuration file formats the built-in documentation for the Lua global functions in a way which is close to the original:

```
project = 'Lua'
description = 'Lua Standard Libraries'
file = {'ldoc/builtin',exclude = {'ldoc/builtin/globals.lua'}}
no_summary = true
no_return_or_parms = true
format = 'discount'
```

Generally, using Markdown gives you the opportunity to structure your documentation in any way you want; particularly if using lua-discount and its **table syntax**; the desired result can often be achieved then by using a custom style sheet.

Examples

It has been long known that documentation generated just from the source is not really adequate to explain *how* to use a library. People like reading narrative documentation, and they like looking at examples. Previously I found myself dealing with source-generated and writer-generated documentation using different tools, and having to match these up.

LDoc allows for source examples to be included in the documentation. For example, see the online documentation for **winapi**. The function `utf8_expand` has a `@see` reference to 'testu.lua' and following that link gives you a pretty-printed version of the code.

The line in the `config.ld` that enables this is:

```
examples = {'examples', exclude = {'examples/slow.lua'}}
```

That is, all files in the `examples` folder are to be pretty-printed, except for `slow.lua` which is meant to be called from one of the examples. To link to an example, use a reference like `@{testu.lua}` which resolves to 'examples/testu.lua.html'.

Examples may link back to the API documentation, for instance the example `input.lua` has a `@{spawn_process}` inline reference.

By default, LDoc uses a built-in Lua code 'prettifier'. Reference links are allowed in comments, and also in code if they're enclosed in backticks. Lua and C are known languages.

lxsh can be used (available from LuaRocks) if you want something more powerful. `pretty='lxsh'` will cause `lxsh` to be used, if available.

Readme files

Like all good Github projects, Winapi has a `readme.md`:

```
readme = "readme.md"
```

This goes under the 'Topics' global section; the 'Contents' of this document is generated from the second-level (##) headings of the readme.

Readme files are always processed with the current Markdown processor, but may also contain `@{ref}`

references back to the documentation and to example files. Any symbols within backticks will be expanded as references, if possible. As with doc comments, a link to a standard Lua function like `@{os.execute}` will work as well. Any code sections will be pretty-printed as Lua, unless the first indented line is '@plain'. (See the source for this readme to see how it's used.)

Another name for `readme` is `topics`, which is more descriptive. From LDoc 1.2, `readme/topics` can be a list of documents. These act as a top-level table-of-contents for your documentation. Currently, if you want them in a particular order, then use names like `01-introduction.md` etc, which sort appropriately.

The first line of a document may be a Markdown `#` title. If so, then LDoc will regard the next level as the subheadings, normally second-level `##`. But if the title is already second-level, then third-level headings will be used `###`, and so forth. The implication is that the first heading must be top-level relative to the headings that follow, and must start at the first line.

A reference like `@{string.upper}` is unambiguous, and will refer to the online Lua manual. In a project like Penlight, it can get tedious to have to write out fully qualified names like `@{pl.utils.printf}`. The first simplification is to use the `package` field to resolve unknown references, which in this case is 'pl'. (Previously we discussed how `package` is used to tell LDoc where the base package is in cases where the module author wishes to remain vague, but it does double-duty here.) A further level of simplification comes from the `@lookup` directive in documents, which must start at the first column on its own line. For instance, if I am talking about `pl.utils`, then I can say `@lookup utils` and thereafter references like `@{printf}` will resolve correctly.

If you look at the source for this document, you will see a `@lookup doc.md` which allows direct references to sections like **this** with `@{Readme_files|this}`.

Remember that the default is for references in backticks to be resolved; unlike @ references, it is not an error if the reference cannot be found.

The *sections* of a document (the second-level headings) are also references. This particular section can be refered to as `@{doc.md.Resolving_References_in_Documents}` – the rule is that any non-alphabetic character is replaced by an underscore.

Any indented blocks are assumed to be Lua, unless their first line is `@plain`. New with 1.4 is github-markdown-style fenced code blocks, which start with three backticks optionally followed by a language. The code continues until another three backticks is found: the language can be `c`,'cpp' or `cxx` for C/C++, anything else is Lua.

Tag Modifiers

Ay tag may have *tag modifiers*. For instance, you may say `@param[type=number]` and this associates the modifier `type` with value `number` with this particular param tag. A shorthand has been introduced for this common case, which is `@tparam <type> <parmname> <comment>`; in the same way `@treturn` is defined.

This is useful for larger projects where you want to provide the argument and return value types for your API, in a structured way that can be easily extracted later.

These types can be combined, so that "?string|number" means "ether a string or a number"; "?string" is short for "?|nil|string". However, for this last case you should usually use the `opt` modifier discussed below.

There is a useful function for creating new tags that can be used in `config.ld`:

```
tparam_alias('string','string')
```

That is, "@string" will now have the same meaning as "@tparam string"; this also applies to the optional type syntax "?|T1|T2".

From 1.3, the following standard type aliases are predefined:

- `string`
- `number`
- `int`
- `bool` Lua 'boolean' type
- `func` 'function' (using 'function' would conflict with the type)
- `tab` 'table'
- `thread`

When using 'colon-style' (**colon.lua**) it's possible to directly use types by prepending them with '!'; '?' is also naturally understood.

```
            <type>
```

The exact form of is not defined, but here is one suggested scheme:

- `number` — a plain type
- `Bonzo` — a known type; a reference link will be generated
- `{string,number}` — a 'list' tuple of two values, built from type expressions
- `{A=string,N=number}` — a 'struct', ditto (But it's often better to create a named table and refer to it)
- `{Bonzo,...}` — an array of Bonzo objects
- `{[string]=Bonzo,...}` — a map of Bonzo objects with string keys
- `Array(Bonzo)` — (assuming that Array is a container type)

The `alias` function within configuration files has been extended so that alias tags can be defined as a tag plus a set of modifiers. So `tparam` is defined as:

```
alias('tparam',{'param',modifiers={type="$1"}})
```

As an extension, you're allowed to use '@param' tags in table definitions. This makes it possible to use type alias like '@string' to describe fields, since they will expand to 'param'.

Another modifier understood by LDoc is `opt`. For instance,

```
---- testing [opt]
-- @param one
-- @param[opt] two
-- @param three
-- @param[opt] four
function two (one,two,three,four)
end
----> displayed as: two (one [, two], three [, four])
```

This modifier can also be used with type aliases. If a value is given for `opt` then LDoc can present this as the default value for this optional argument.

```
--- a function with typed args.
-- If the Lua function has varargs, then
-- you may document an indefinite number of extra arguments!
-- @string name person's name
-- @int age
-- @string[opt='gregorian'] calender optional calendar
-- @int[opt=0] offset optional offset
-- @treturn string
function one (name,age,...)
end
----> displayed as: one (name, age [, calender='gregorian' [, offset=0]])
```

(See **four.lua**, rendered **here**)

An experimental feature in 1.4 allows different 'return groups' to be defined. There may be multiple `@return` tags, and the meaning of this is well-defined, since Lua functions may return multiple values. However, being a dynamic language it may return a single value if successful and two values (`nil`,an error message) if there is an error. This is in fact the convention for returning 'normal' errors (like 'file not found') as opposed to parameter errors (like 'file must be a string') that are often raised as errors.

Return groups allow a documenter to specify the various possible return values of a function, by specifying *number* modifiers. All `return` tags with the same digit modifier belong together as a group:

```
-----
-- function with return groups.
-- @return[1] result
-- @return[2] nil
-- @return[2] error message
function mul1() ... end
```

This is the first function in **multiple.lua**, and the **output** shows how return groups are presented, with an **Or** between the groups.

This is rather clumsy, and so there is a shortcut, the `@error` tag which achieves the same result, with helpful type information.

Currently the `type`,`opt` and `<digit>` modifiers are the only ones known and used by LDoc when generating HTML output. However, any other modifiers are allowed and are available for use with your own templates or for extraction by your own tools.

Fields allowed in `config.ld`

Same meaning as the corresponding parameters:

- `file` a file or directory containing sources. In `config.ld` this can also be a table of files and directories.
- `project` name of project, used as title in top left
- `title` page title, default 'Reference'
- `package` explicit base package name; also used for resolving references in documents
- `all` show local functions, etc as well in the docs
- `format` markup processor, can be 'plain' (default), 'markdown' or 'discount'
- `output` output name (default 'index')
- `dir` directory for output files (default 'doc')
- `colon` use colon style, instead of @ tag style
- `boilerplate` ignore first comment in all source files (e.g. license comments)
- `ext` extension for output (default 'html')
- `one` use a one-column layout
- `style`, `template`: together these specify the directories for the style and and the template. In `config.ld` they may also be `true`, meaning use the same directory as the configuration file.
- `merge` allow documentation from different files to be merged into modules without explicit @submodule tag

These only appear in the configuration file:

- `description` a short project description used under the project title
- `full_description` when you *really* need a longer project description
- `examples` a directory or file: can be a table
- `readme` or `topics` readme files (to be processed with Markdown)
- `pretty` code prettify 'lua' (default) or 'lxsh'
- `charset` use if you want to override the UTF-8 default (also @charset in files)
- `sort` set if you want all items in alphabetical order
- `no_return_or_parms` don't show parameters or return values in output
- `backtick_references` whether references in backticks will be resolved. Happens by default when using Markdown. When explicit will expand non-references in backticks into `<code>` elements
- `plain` set to true if `format` is set but you don't want code comments processed
- `wrap` ??
- `manual_url` point to an alternative or local location for the Lua manual, e.g. 'file:///D:/dev/lua/projects/lua-5.1.4/doc/manual.html'
- `no_summary` suppress the Contents summary
- `custom_see_handler` function that filters see-references
- `not_luadoc` set to `true` if the docs break LuaDoc compatibility
- `no_space_before_args` set to `true` if you do not want a space between a function's name and its arguments.
- `template_escape` overrides the usual '#' used for Lua code in templates. This needs to be changed if the output format is Markdown, for instance.

Available functions are:

- `alias(a,tag)` provide an alias `a` for the tag `tag`, for instance `p` as short for `param`
- `add_language_extension(ext,lang)` here `lang` may be either 'c' or 'lua', and `ext` is an extension to be recognized as this language
- `add_section`
- `new_type(tag,header,project_level)` used to add new tags, which are put in their own section `header`. They may be 'project level'.
- `tparam_alias(name,type)` for instance, you may wish that `Object` becomes a new tag alias that means

`@tparam Object.`

- `custom_see_handler(pattern,handler)`. If a reference matches `pattern`, then the extracted values will be passed to `handler`. It is expected to return link text and a suitable URI. (This match will happen before default processing.)

Annotations and Searching for Tags

Annotations are special tags that can be used to keep track of internal development status. The known annotations are 'todo', 'fixme' and 'warning'. They may occur in regular function/table doc comments, or on their own anywhere in the code.

```
--- Testing annotations
-- @module annot1
...
--- first function.
-- @todo check if this works!
function annot1.first ()
    if boo then

    end
    --- @fixme what about else?
end
```

Although not currently rendered by the template as HTML, they can be extracted by the `--tags` command, which is given a comma-separated list of tags to list.

```
D:\dev\lua\LDoc\tests> ldoc --tags todo,fixme annot1.lua
d:\dev\lua\ldoc\tests\annot1.lua:14: first: todo check if this works!
d:\dev\lua\ldoc\tests\annot1.lua:19: first-fixme1: fixme what about else?
```

Generating HTML

LDoc, like LuaDoc, generates output HTML using a template, in this case `ldoc/html/ldoc_ltp.lua`. This is expanded by the powerful but simple preprocessor devised originally by **Rici Lake** which is now part of Lake](http://lua-users.org/wiki/SlightlyLessSimpleLuaPreprocessor) which is now part of Penlight. There are two rules – any line starting with '#' is Lua code, which can also be embedded with '$(…)'.

```
<h2>Contents</h2>
<ul>
# for kind,items in module.kinds() do
<li><a href="#$(no_spaces(kind))">$(kind)</a></li>
# end
</ul>
```

This is then styled with `ldoc.css`. Currently the template and stylesheet is very much based on LuaDoc, so the results are mostly equivalent; the main change that the template has been more generalized. The default location (indicated by '!') is the directory of `ldoc_ltp.lua`.

You will notice that the built-in templates and stylesheets end in `.lua`; this is simply to make it easier for LDoc to find them. Where you are customizing one or both of the template and stylesheet, they will have their usual extensions.

You may customize how you generate your documentation by specifying an alternative style sheet and/or template, which can be deployed with your project. The parameters are `--style` and `--template`, which give the directories where `ldoc.css` and `ldoc.ltp` are to be found. If `config.ld` contains these variables, they are interpreted slightly differently; if they are true, then it means 'use the same directory as config.ld'; otherwise they must be a valid directory relative to the ldoc invocation.

An example of fully customized documentation is `tests/example/style`: this is what you could call 'minimal Markdown style' where there is no attempt to tag things (except emphasizing parameter names). The narrative alone *can* to be sufficient, if it is written well.

There are two other stylesheets available in LDoc since 1.4; the first is `ldoc_one.css` which is what you get from `one=true` and the second is `ldoc_pale.css`. This is a lighter theme which might give some relief from the

heavier colours of the default. You can use this style with `style="!pale"` or `-s !pale`. See the **Lake** documentation as an example of its use.

Of course, there's no reason why LDoc must always generate HTML. `--ext` defines what output extension to use; this can also be set in the configuration file. So it's possible to write a template that converts LDoc output to LaTex, for instance. The separation of processing and presentation makes this kind of new application possible with LDoc.

From 1.4, LDoc has some limited support for generating Markdown output, although only for single files currently. Use `--ext md` for this. 'ldoc/html/ldoc_md_ltp.lua' defines the template for Markdown, but this can be overriden with `template` as above. It's another example of minimal structure, and provides a better place to learn about these templates than the rather elaborate default HTML template.

Internal Data Representation

The `--dump` flag gives a rough text output on the console. But there is a more customizeable way to process the output data generated by LDoc, using the `--filter` parameter. This is understood to be a fully qualified function (module + name). For example, try

```
$ ldoc --filter pl.pretty.dump mylib.c
```

to see a raw dump of the data. (Simply using `dump` as the value here would be a shorthand for `pl.pretty.dump`.) This is potentially very powerful, since you may write arbitrary Lua code to extract the information you need from your project.

For instance, a file `custom.lua` like this:

```
return {
   filter = function (t)
      for _, mod in ipairs(t) do
         print(mod.type,mod.name,mod.summary)
      end
   end
}
```

Can be used like so:

```
~/LDoc/tests/example$ ldoc --filter custom.filter mylib.c
module  mylib   A sample C extension.
```

The basic data structure is straightforward: it is an array of 'modules' (project-level entities, including scripts) which each contain an `item` array (functions, tables and so forth).

For instance, to find all functions which don't have a @return tag:

```
return {
   filter = function (t)
      for _, mod in ipairs(t) do
         for _, item in ipairs(mod.items) do
            if item.type == 'function' and not item.ret then
               print(mod.name,item.name,mod.file,item.lineno)
            end
         end
      end
   end
}
```

The internal naming is not always so consistent; `ret` corresponds to @return, and `params` corresponds to @param. `item.params` is an array of the function parameters, in order; it is also a map from these names to the individual descriptions of the parameters.

`item.modifiers` is a table where the keys are the tags and the values are arrays of modifier tables. The standard tag aliases `tparam` and `treturn` attach a `type` modifier to their tags.

*generated by **LDoc 1.4.0***