C Sphinx Extension Documentation

Release 1.3.0

Nick G.

Sep 21, 2023

CONTENTS:

A basic attempt at extending Sphinx and autodoc to work with C files.

The idea is to add support for similar directives that autodoc provides. i.e.

A function in `my_c_file.c`:

```
/**
 * A simple function that adds.
 *
 * @param a: The initial value
 * @param b: The value to add to `a`
 *
 * @returns The sum of `a` and `b`.
 *
 *
int my_adding_function(int a, int b) {
    return a + b;
    }
```

Could be referenced in documentation as:

```
.. autocfunction:: my_c_file.c::my_adding_function
```

With the resulting documentation output of:

int my_adding_function(int a, int b)

A simple function that adds.

Parameters

- **a** - The initial value
- **b** - The value to add to *a*

Returns

The sum of *a* and *b*

CHAPTER

ONE

REQUIRES

- clang
- beautifulsoup4

CHAPTER

TWO

SIMILAR TOOLS

- hawkmoth a sphinx extension that which will document all of a C file. It supports being able to regex list files and have those files be documented.
- breathe A doxygen output to sphinx tool.

2.1 Directives

Most C constructs are available as an autodoc style directive.

Do to the fact that autodoc directives are set up into a global namespace and not in domain specific namespace. The directives used for auto documenting C constructs will have a `c` moniker in their names.

Keeping consistent with autodoc, the term **member** is used generically to refer to any sub construct of another C construct. For example in this documentation any usage a C function may be referred to as a member of a C module.

2.1.1 Common Options

`:noindex:`

By default the documentation instances are added to the index. To prevent an instance from being added to the index provide this option.

`.. autocmodule::` `filename`

Create documentation for *filename*. The *filename* is relative to *c_autodoc_roots*. This directive can be used for both C source files as well as C header files.

The contents of the first documentation comment will be utilized as the documentation for *filename*.

`:members:`

Specify which members to recursively document.

This option has 4 states:

- Omitted will result in the `members` entry of autodoc_default_options being used. If `members` is omitted from autodoc_default_options then no members will be automatically documented.
- Specified as `:no-members:`, no members will be automatically documented.
- Specified with no arguments, `:members:`, then all supported C constructs at the file level will be recursively documented.
- Specified with a comma separated list of names, `:members: function_a, struct_b`, only the file members specified will be recursively documented.

`:private-members:`

Specify if private members are to be documented. Since C doesn't actually have a true idea of public and private, the following rules are used to determine these characteristics.

Members are public if:

- They are in a header file, ends in *.h*. By nature header files are meant to be included in other files, thus anything declared there is deemed public.
- They can be visible outside of the compilation unit. These are things the linker can get access to. Mainly this means functions and variables that are not static.

Just as for the standard autodoc options, one can use the negated form of `:no-private-members:` to selectively turn off this option on a per module basis.

`:undoc-members:`

Specify if undocumented members are to show in the documentation. This option is off by default.

Just as for the standard autodoc options, one can use the negated form of `:no-undoc-members:` to selectively turn off this option when it has been enabled in autodoc_default_options.

`.. autocfunction:: filename::function`

Document the function `function` from file `filename`.

The documentation will first attempt to utilize the output provided by clang. Clang can parse Doxygen style markup. So if the function parameter documentation or the function return value documentation is seen in the clang parsing, then the parsed clang documentation will be used. Otherwise the raw function comment block will be utilized.

`.. autocmacro:: filename::some_define`

Document a C macro. Both macro constants as well as function like macros.

`.. autoctype:: filename::typedef`

Document a typedef

`.. autocenum:: filename::enum_name`

Document a enum

`:members:`

Specify which members to recursively document.

This option has 4 states:

- Omitted will result in the `members` entry of autodoc_default_options being used. If `members` is omitted from autodoc_default_options then no members will be automatically documented.
- Specified as `:no-members:`, no members will be automatically documented.
- Specified with no arguments, `:members:`, then all enumerator constants will be documented.
- Specified with a comma separated list of names, `:members: field_a, field_b`, only the items specified will be recursively documented.

`.. autocenumerator:: filename::enum_name.enumerator`

Document a enumerator. One of the constant values of an enum.

`.. autocstruct:: filename::struct_name`

Document a struct

`:members:`

Specify which members to recursively document.

This option has 4 states:

- Omitted will result in the `members` entry of autodoc_default_options being used. If `members` is omitted from autodoc_default_options then no members will be automatically documented.
- Specified as `:no-members:`, no members will be automatically documented.
- Specified with no arguments, `:members:`, then all fields will be recursively documented.
- Specified with a comma separated list of names, `:members: field_a, field_b`, only the items specified will be recursively documented.

`.. autocunion:: filename::union_name`

Document a union

`:members:`

Specify which members to recursively document.

This option has 4 states:

- Omitted will result in the `members` entry of autodoc_default_options being used. If `members` is omitted from autodoc_default_options then no members will be automatically documented.
- Specified as `:no-members:`, no members will be automatically documented.
- Specified with no arguments, `:members:`, then all fields will be recursively documented.
- Specified with a comma separated list of names, `:members: field_a, field_b`, only the items specified will be recursively documented.

`.. autocdata:: filename::variable`

Document a file level variable.

`.. autocmember:: filename::struct.field`

Document the specified field of a struct or union.

Note: This is one of the overloaded uses of the term **member**. This name was used to keep consistent with the member wording of the C domain.

2.2 Example C File

Below is an example of a auto documenting an entire C file. Click on any of the `[source]` links to jump to the location in the raw source file.

2.2.1 Begin Autodocumented C File

This is a file comment. The *first* comment in the file will be grabbed. Often times people put the copyright in these. If that is the case then you may want to utilize the pre processing hook, *c-autodoc-pre-process*.

One may notice that this comment block has a string of *** along the top and the bottom. For the file comment these will get stripped out, however for comments on other c constructs like macros, functions, etc. clang is often utilized and it does not understand this pattern, so the *c-autodoc-pre-process* hook may be something to use to sanitize these kind of comments.

MY_COOL_MACRO(_a, _b)

A function like macro

An attempt will be made to derive the arguments of the macro. It will probably work in most instances…

Function like macros can be documented with the `:param:` and `:returns:` fields. One could even utilize the napoleon extension to format something like:

Parameters

- **_a** – The time of day as derived from the current temperature.
- **_b** – The place to be.

Returns

The predicted value of stocks based on *_a*.

TOO_SIMPLE

A simple macro definition

struct **a_struct_using_member_comments**

Structures can be documented.

When the structure is anonymous and hidden inside a typedef, like this one, it will be documented using the typedefed name.

The members can be documented with individual comments, or they can use a members section. This example struct documents the members with individual comments.

float **first_member**

The first member of this specific structure using a trailing comment, notice the < after the comment start

int **second_member**

This member is documented with a comment proceeding the member.

typedef int **a_typedef_type**

A plain old typedef

enum **anon_example_3952103300**

Anonymous enums are supported, so that the enumerators can be documented.

Note: That one will not be able to autodoc the enum directly it will only be included by autodocing a module. Since it's name will be built up dynamically

---

enumerator **AN_ANONYMOUSE_1**

The first enumerator from an anonymous enum.

enumerator **AN_ANONYMOUSE_2**

The second enumerator from an anonymous enum.

struct **inline_struct_variable**

Even structures defined in variables can be handled.

int **a**

float **b**

struct **members_documented_with_napoleon**

This example structure uses the *Members:* section and lets napoleon format the members.

int **one**

The first member of parent struct

struct **two**

This is a structure declared in the parent struct its children are documented below.

float **nested_one**

The nested member documentation

int **nested_two**

The second nested member documentation

float **three**

The third member of parent struct

int **my_func**(float hello, char what)

This is a function comment. The parameters from this are much easier to derive than those from a function like macro so they should always be correct.

Since the backend parser is clang and clang supports doxygen style comments One can document functions using normal doxygen style markup.

Parameters

- **hello:** – The amount of hello appericiations seen so far.
- **what:** – The common reply character seen.

Returns

The increase on hello's in order to maintain politeness.

int **napoleon_documented_function**(int yes, int another_one)

One can also use Goolge style docstrings with napoleon for documenting functions.

Note: Functions do not support mixing doxygen style and napoleon style documentation.

Parameters

- **yes** – A progressive rock band from the 70s.
- **another_one** – Yet one more parameter for this function.

Returns

The square root of 4, always.

enum **some_enum**

If you want to document the enumerators with napoleon then you use the section title *Enumerators:*.

enumerator **THE_FIRST_ENUM**

Used for the first item

Documentation in a comment for THE_FIRST_ITEM. Note this is trailing, for some reason clang will apply leading comments to *all* the enumerators

enumerator **THE_SECOND_ENUM**

Second verse same as the first.

enumerator **THE_THIRD_ENUM**

Not once, note twice, but thrice.

enumerator **THE_LAST_ENUM**

Just to be sure.

int **some_flag_variable**

File level variables can also be documented

2.3 Configuring

2.3.1 Configuration Variables

c_autodoc_roots

A list of directories which will be used to search for the files provided in the *Directives*. The directories are relative to documentation source directory, often where `conf.py` is.

The list of directories will be searched in order so if duplicate named files exist the first one encountered in the directory list will be used.

Example:

```
c_autodoc_roots = ['my/source/dir', 'other/source/dir']
```

Then a directive of the form:

```
.. autocfunction:: some_file.c::some_function
```

would be searched first as `my/source/dir/some_file.c` then, if not found, it would be searched as `other/source/dir/some_file.c`. Again this relative to the top documentation source directory.

c_autodoc_compilation_database

Path to a compilation database. The compilation database is relative to the documentation source directory, often where `conf.py` is.

The compilation database will be used as the source of compile options for each file. If a file is listed more than once in the compilation database, only the first instance of the file will be used. Of importance is the `directory` entry for each file. The `directory` entry will be passed to libclang via the working-directory flag. The `-working-directory` allows for the includes and other path relative arguments to be handled consistently.

Note: Currently libclang only supports compilation databases named `compile_commands.json`.

---

c_autodoc_compilation_args

A list of arguments to pass to libclang. This can be used for setting common defines used only for documentation and/or avoiding areas of the code that have trouble parsing for documentation.

For example the following would result in libclang parsing the source files with the defines `SPHINX_DOCS` and `SIMULATION`:

```
c_autodoc_compilation_args = ["-DSPHINX_DOCS", "-DSIMULATION"]
```

`c_autodoc_compilation_args` are added for *all* files being processed. `c_autodoc_compilation_args` will be applied *after* any arguments provided by *c_autodoc_compilation_database*.

2.3.2 Events

c-autodoc-pre-process

There are times a project needs to perform some form of pre-processing of a source file prior to the build up of the auto-documentation. The *c-autodoc-pre-process* is a sphinx event that will be triggered prior to the parsing of the C file.

c-autodoc-pre-process(app, filename, contents, *args)

Parameters

- **app** – the Sphinx application object
- **filename** – The full filename being parsed
- **contents** – The file contents. This is a list with one item, the file contents. Modify the list in place. Only the first element will be looked at by the parser.
- **args** – Unused, but provides compatibility for future expansions.

An example which replaces all instances of "the" with "this":

```
def pre_process(app, filename, contents, *args):
    file_body = contents[0]

    modified_contents = file_body.replace("the", "this")

    # replace the list to return back to the sphinx extension
    contents[:] = [modified_contents]

app.connect("c-autodoc-pre-process", pre_process)
```

autodoc-process-docstring

Since this is extending the autodoc functionality the autodoc events are available as well. Of particular interest may be the autodoc-process-docsting which will be emitted for every C construct.

2.4 Napoleon

This extension also provides a way to extend napoleon to work with C constructs.

Enabling this feature simply requires adding the napoleon sub package of this extension to the list of desired sphinx extensions:

```
extensions = [
    'sphinx_c_autodoc.napoleon',
    ]
```

Note: Currently only the Google style docstrings are supported.

Using this extension will take:

```
/**
 * This example structure uses the `Members:` section and lets napoleon format
 * the members.
 *
 * Members:
 *     one: The first member of parent struct
 *     two: This is a structure declared in the parent struct its children are
 *         documented below.
 *         Members:
 *             nested_one: The nested member documentation
 *             nested_two: The second nested member documentation
 *     three: The third member of parent struct
 *
 */
struct members_documented_with_napoleon
{
    int one;
    struct {
        float nested_one;
        int nested_two;
    } two;
    float three;
};
```

and convert it into

struct **members_documented_with_napoleon**

This example structure uses the *Members:* section and lets napoleon format the members.

int **one**

The first member of parent struct

struct **two**

This is a structure declared in the parent struct its children are documented below.

float **nested_one**

The nested member documentation

int **nested_two**

The second nested member documentation

float **three**

The third member of parent struct

2.5 Viewcode

Similar to the sphinx python extension viewcode. This extension provides a way to also list the contents of C files and link between the documentation to the C file contents.

Enabling this feature simply requires adding the viewcode sub package of this extension to the list of desired sphinx extensions:

```
extensions = [
    'sphinx_c_autodoc.viewcode',
    ]
```

This functionality can be seen with the *Example C File*, there will be tags of the form `[source]` which can be clicked on and will follow the link to the html version of the C file.

In order for this to work, the C domain directives have been expanded to support a `:module:` option. This option is the name of the C file, relative to *c_autodoc_roots*.

The auto directives will automatically populate this option. If one needs to use non auto directives then this option will need to be manually specified for the source code file to be populated.

2.6 apidoc

The *sphinx-c-apidoc* provides a way to generate documentation files for a C directory. It is meant to fulfill a similar role to the sphinx-apidoc command for python packages.

2.6.1 sphinx-c-apidoc

```
usage: sphinx-c-apidoc [-h] [-o OUTPUT_PATH] [-f] [-t TEMPLATEDIR]
                       [--tocfile TOCFILE] [-d MAXDEPTH]
                       [--header-ext HEADER_EXT] [--source-ext SOURCE_EXT]
                       source_path
```

source_path

Path to C source files to be documented

-h, --help

show this help message and exit

-o <output_path>, **--output-path** <output_path>

Directory to place the output files. If it does not exist, it is created

-f, --force

Force overwriting of any existing generated files

-t <templatedir>, **--templatedir** <templatedir>

Template directory for template files

--tocfile <tocfile>

Filename for the root table of contents file (default: files)

-d <maxdepth>

Maximum depth for the generated table of contents file(s). (default: 4)

--header-ext <header_ext>

The extension(s) to use for header files (default: ["h"])

--source-ext <source_ext>

The extension(s) to use for source files (default: ["c"])

2.6.2 Generated Documentation Files

The generated documentation files will follow the same directory structure of the provide C source directory.

For example:

```
a_project
├── file_1.h
├── file_2.c
├── some_dir
    ├── another_dir
        ├── file_3.h
        └── file_4.c
```

would result in:

```
doc_dir
├── files.rst
├── file_1_h.rst
├── file_2_c.rst
├── some_dir
    ├── some_dir.rst
    ├── another_dir
        ├── another_dir.rst
        ├── file_3_h.rst
        └── file_4_c.rst
```

Where *a_project* was provided as the `source_path` and *doc_dir* was provided as the `--output-path`. `files.rst` is the root index or table of contents file. By default it only contains references to the other documentation files in the same directory and any index files in sub directories.

`another_dir.rst` is the index or table of contents file for the directory *another_dir* it will only contain references the files in that directory as well as any index files in subsequent sub directories.

2.6.3 Templates

There are three jinja templates that are utilized for generating the documentation files. These can be overridden by passing a directory, via the `--templatedir` option, containing any of the templates to override.

header.rst.jinja2

Controls the generation of files deemed to be header files, the `--header-ext` option.

Will be passed 2 arguments:

- `filepath` The relative path to the file. For `file_3.h`, from the example above in *Generated Documentation Files*, this would be `some_dir/another_dir/file_3.h`
- `filename` The name of the file, without relative directory. For `file_3.h`, from the example above in *Generated Documentation Files*, this would be `file_3.h`

source.rst.jinja2

Controls the generation of files deemed to be source files, the `--source-ext` option.

Will be passed 2 arguments:

- `filepath` The relative path to the file. For `file_4.c`, from the example above in *Generated Documentation Files*, this would be `some_dir/another_dir/file_4.c`
- `filename` The name of the file, without relative directory. For `file_4.c`, from the example above in *Generated Documentation Files*, this would be `file_4.c`

toc.rst.jinja2

Controls the generation of index or table of contents files.

Will be passed 3 arguments:

- `title` The name of the index file without extension.
- `maxdepth` The `-d` option.
- `doc_names` The list of documentation files in the directory as well as those in subdirectories.

2.7 Minimum Supported Versions

Keeping with the industry support of Python versions https://www.python.org/downloads/, this package will try to support the currently supported Python versions. The minimum supported dependency versions will be the newest version of the dependency that was available when the minimum supported Python version was released.

Dropping support for a dependency version is considered a major change and will result in a major version bump.

Planned support calendar. The end dates aren't December 31st, they're the October end of life dates for the associated Python version.

2020	2021	2022	2023	2024	2025	2026	2027
Python 3.8					xxxx	xxxx	xxxx
Python 3.9						xxxx	xxxx
xxxx	Python 3.10						xxxx
xxxx	xxxx	Python 3.11					
Sphinx 3.1					xxxx	xxxx	xxxx
xxxx	Sphinx 4.x						xxxx
xxxx	xxxx	Sphinx 5.x					
xxxx	xxxx	Sphinx 6.x					
xxxx	xxxx	xxxx	Sphinx 7.x				
Clang 6.x					xxxx	xxxx	xxxx
Clang 7.x					xxxx	xxxx	xxxx
Clang 8.x					xxxx	xxxx	xxxx
Clang 9.x					xxxx	xxxx	xxxx
Clang 10.x					xxxx	xxxx	xxxx
Clang 11.x						xxxx	xxxx
xxxx	xxxx	Clang 12.x					xxxx
xxxx	xxxx	Clang 13.x					xxxx
xxxx	xxxx	Clang 14.x					
xxxx	xxxx	xxxx	Clang 15.x				
xxxx	xxxx	xxxx	Clang 16.x				
Beautiful Soup 4							

Note: The different Clang versions are *not* exercised in CI. Most of the Python API for Clang seems pretty stable and as long as the most recent version of Clang keeps working it's hoped that it will catch most issues.

2.8 Change Log

This document records all notable changes to sphinx-c-autodoc. This project adheres to Semantic Versioning.

2.8.1 v1.3.1-dev (unreleased)

2.8.2 v1.3.0 (2023-09-21)

Added

- Support for Sphinx 7

Removed

- Support for python 3.7

2.8.3 v1.2.2 (2023-06-28)

Fixed

- Failed parsing of enumerators which were defined based on macros #174

2.8.4 v1.2.1 (2023-06-12)

Fixed

- Regression for processing anonymous structs in versions of Clang prior to Clang 16. #166

2.8.5 v1.2.0 (2023-06-11)

Added

- Support for Clang 16. Clang 16 changed the way anonymous constructs are represented when walking the AST. These differences are now accounted for.

2.8.6 v1.1.1 (2022-12-21)

Fixed

- Packaging * Incorrectly pinned Sphinx, Clang, BeautifulSoup4 * Was missing *sphinx-c-apidoc* entry point

2.8.7 v1.1.0 (2022-12-20)

Added

- Support for Sphinx 5

2.8.8 v1.0.0 (2021-09-12)

Added

- Dual licensing with MIT license.
- Support for global compilation args. Compilation args can now be specified with the `c_autodoc_compilation_args` configuration value.

Fixes

- Fix parsing of multi-paragraph function documentation that used doxygen style markup. Previously multi-paragraph function documentation which used doxygen style markup would get merged in to one paragraph. Now the multi-paragraph nature is preserved.

2.8.9 v0.4.0 (2020-12-27)

Changed

- Undocumented constructs are no longer added to the documentation by default. To maintain previous behavior add *:undoc-members:* to the project's autodoc_default_options.

Added

- *:undoc-members:* and *:no-undoc-members:* option for the autocmodule directive. This option set allows for controlling the listing of undocumented constructs. The default is to not list undocumented constructs.

Fixes

- Fix file level variables with unknown types. Previously variables with unknown types would cause an error in Sphinx processing.
- Fix documentation of members that are arrays. Previously struct members that were array types would cause an error as the array size was put between the type and the member name.
- Fix viewcode processing of empty files. Previously an exception would be raised when the viewcode extension tried to process empty files.
- Call out Sphinx 3.1 as minimum version in `setup.py`. Previously the Sphinx version in setup.py called out 3.0 or greater. This was incorrect as features from Sphinx 3.1 are being utilized.

2.8.10 v0.3.1 (2020-10-24)

Added

- The `sphinx-c-apidoc` command. This command provides users the ability to quickly build up a set of documentation files for a C directory.
- Support for compile flags via a compilation database. A compilation database can now be specified with the `c_autodoc_compilation_database` configuration value.

Fixes

- No longer passing *typedef* to auto documentation of types. Previously the *typedef* keyword was being provided to sphinx as part of the signature. This resulted in losing the ability to link to the type in html documentation.
- Display of function arguments with unknown array types. Previously when a function had an argument that was an array and an unknown type, it would result in being empty in the output documentation. Now the full function signature is provided token by token, with the exception of comments.

2.8.11 v0.3.0 (2020-08-22)

Changed

- Changed to support sphinx version 3. Due to significant changes between sphinx 2 and 3, sphinx 2 is no longer supported.

Fixes

- typedef function and function pointers with unknown return types were not being properly handled. These now get handled, but the unknown types are whatever clang provides, which is usually `int`.
- typedef unions with unknown member types caused in index error. This has been fixed and these unknown member types are evaluated by clang to be `int`.
- Fix comments in function declarations showing up in documentation. When comments were placed inbetween parameter types and the parameter names, and the type was unknown to clang, the fallback parsing would take the declaration character for character (consolidating whitespace). This resulted in comments being pulled in to the documentation verbatim. Now comments will explicitly be skipped over when the fall back parsing for function declarations happens.

2.8.12 v0.2.0 (2020-04-04)

Added

- Viewcode functionality which allows for listing the source C files and providing links between the documentation and the C source listings.
- *:private-members:* and *:no-private-members:* option for the autocmodule directive. This option set allows for controlling the documentation of constructs based on what is visible outside of the module. For header files this means everything will still be documented. For standard source files only non static functions and non static variables will be auto documented if the :private-members: is not specified, or the :no-private-members: is specified.

Fixes

- Anonymous enumerations which were contained in a typedef were being documented twice. Once as the typedef and once as anonymous. Now they are only documnted as part of the typedef.

2.8.13 v0.1.1 (2020-03-15)

Fixes

- C module is not resolved relative to the document root, #1.
- C module can not be specified in a sub directory, #2.

2.8.14 v0.1.0 (2020-03-07)

- Initial public release

2.9 Developing

2.9.1 Developer Notes

Common Terms

Construct:
In order to find a common word to describe things such as C functions, C types, C struct members a common term is needed. This common term is _construct_.

AST:
Abstract Syntax Tree or AST is a common term used to describe the break down of a C source file into its components. Normally an AST goes all the way down to things like if conditions and other constructs. However for the use in autodoc there is no reason to break down the contents of a function. Structures, unions and enums are further broken down into their member or enumerator constants though.

For this extension the common AST is just a simple dictionary which has the following entries:

`name` (str):
The name of the construct, i.e. function name, variable name etc.

`start_line` (int):
The line number in the source code where this construct starts. Line numbers are 1 based, not 0 based.

`end_line` (int):
The line number in the source code where this construct ends. For some constructs this line will be the same as the start. Line numbers are 1 based, not 0 based.

`children` (list):
A list of child constructs with the same fields as this. Children may be things such as struct members. Or functions within a file where the file is the root node. An empty list is still provided for no children.

The file itself should be the root node.

2.9.2 Contributing

The source code is available on github at `https://github.com/speedyleion/sphinx-c-autodoc`

The main development of this project utilizes tox. The project itself does not provide a libclang implementation so this must be installed and available on your system. Some common tox environments are:

- `tox -e py38` to run the tests.
- `tox -e py38-cov` to run the tests for coverage.
- `tox -e docs` to generate the documentation.

Note: Prior to contributing via a pull request please ensure all tox environments pass by running `tox`, with no additional arguments.

Coverage

The tests should currently have 100% coverage, . This means any contributions should maintain that level of coverage. It is understandable that the amount of coverage can be a touchy topic and that some feel 100% is gold plating. To be perfectly honest 80-90% was what this author used to target, until a peer made the convincing argument:

> It is better to have a 100% coverage report with known coverage exceptions than to have 10-20% of the code in an unknown state.

Once a project has 100% coverage keeping that coverage is usually fairly cheap. One should only be contributing functionality that is desired and if it's desired then there should be tests to help maintain its correctness. There are times where edge cases may be harder to hit and in such cases it is acceptable to have a `# pragma: no cover` with some form of documentation describing why that coverage wasn't able to be covered in automated tests.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

I

M

N

S

T