AutoIndex

John Maddock

Table of Contents

Overview

AutoIndex is a tool for taking the grunt work out of indexing a Boostbook/Docbook document (perhaps generated by your Quickbook file mylibrary.qbk, and perhaps using also Doxygen autodoc) that describes C/C++ code.

Traditionally, in order to index a Docbook document you would have to manually add a large amount of `<indexterm>` markup: in fact one `<indexterm>` for each occurrence of each term to be indexed.

Instead AutoIndex will automatically scan one or more C/C++ header files and extract all the *function*, *class*, *macro* and *typedef* names that are defined by those headers, and then insert the `<indexterm>`s into the Docbook XML document for you.

AutoIndex can also scan using a list of index terms specified in a script file, for example index.idx. These manually provided terms can optionally be regular expressions, and may allow the user to find references to terms that may not occur in the C++ header files. Of course providing a manual list of search terms in to index is a tedious task (especially handling plurals and variants), and requires enough knowledge of the library to guess what users may be seeking to know, but at least the real 'grunt work' of finding the term and listing the page number is automated.

AutoIndex creates index entries as follows:

for each occurrence of each search term, it creates two index entries:

1. The search term as the *primary index key* and the *title of the section it appears in* as a subterm.

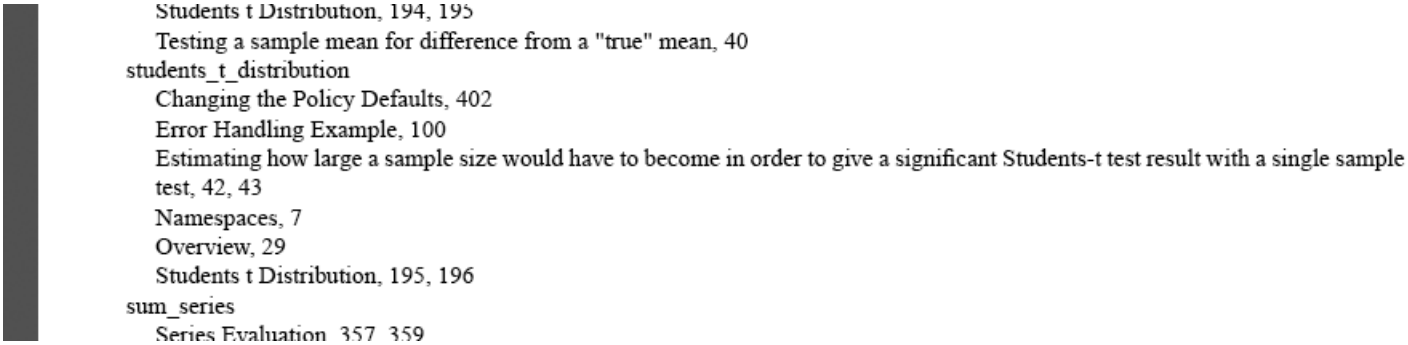
2. The section title as the main index entry and the search term as the subentry.

Thus the user has two chances to find what they're looking for, based upon either the section name or the *function*, *class*, *macro* or *typedef* name.

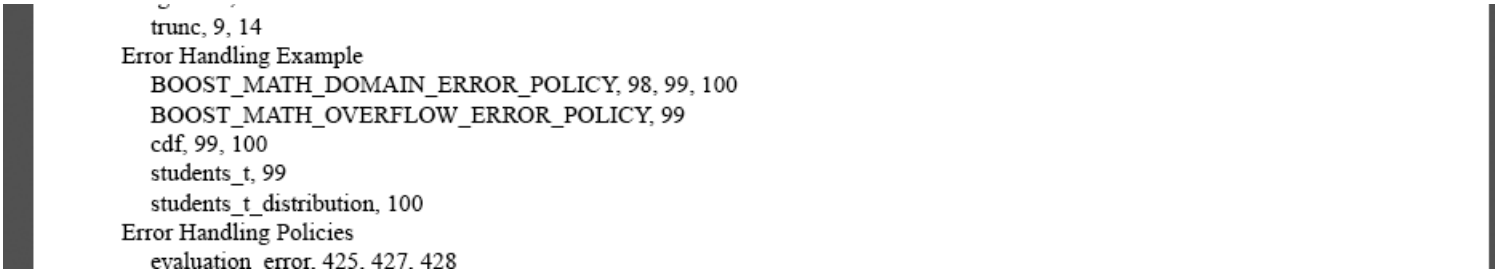
Note

This behaviour can be changed so that only one index entry is created (using the search term as the key and not using the section name except as a sub-entry of the search term).

So for example in Boost.Math the class name `students_t_distribution` has a primary entry that lists all sections the class name appears in:



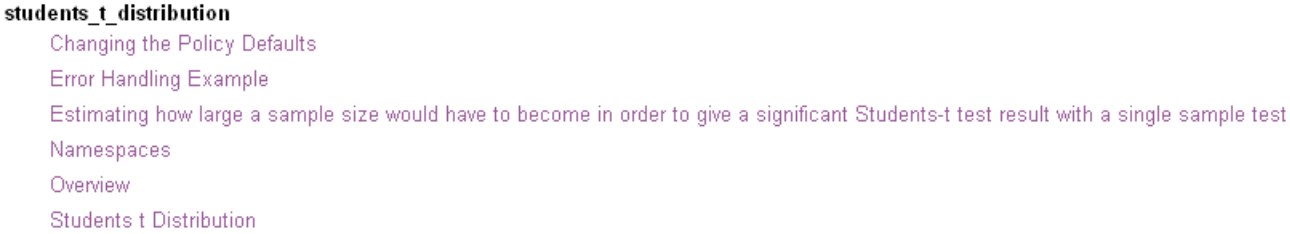
Then those sections also have primary entries, which list all the search terms those sections contain:



Of course these automated index entries may not be quite what you're looking for: often you'll get a few spurious entries, a few missing entries, and a few entries where the section name used as an index entry is less than ideal. So AutoIndex provides some powerful regular expression based rules that allow you to add, remove, constrain, or rewrite entries. Normally just a few lines in AutoIndex's script file are enough to tailor the output to match the author's expectations (and thus hopefully the index user's expectations too!).

AutoIndex also supports multiple indexes (as does Docbook), and since it knows which search terms are *function*, *class*, *macro* or *typedef* names, it can add the necessary attributes to the XML so that you can have separate indexes for each of these different types. These specialised indexes only contain entries for the *function*, *class*, *macro* or *typedef* names, *section names* are never used as primary index terms here, unlike the main "include everything" index.

Finally, while the Docbook XSL stylesheets create nice indexes complete with page numbers for PDF output, the HTML indexes look a lot less good, as these use section titles in place of page numbers... but as AutoIndex uses section titles as index entries this leads to a lot of repetition, so as an alternative AutoIndex can be instructed to construct the index itself. This is faster than using the XSL stylesheets, and now each index entry is a hyperlink to the appropriate section:



With internal index generation there is also a helpful navigation bar at the start of each Index:

Finally, you can choose what kind of XML container wraps an internally generated index - this defaults to `<section>...</section>` but you can use either command line options or Boost.Build Jamfile features, to select an alternative wrapper - for example *appendix* or *chapter* would be good choices, whatever fits best into the flow of the document. You can even set the container wrapper to type *index* provided you turn off index generation by the XSL stylesheets, for example by setting the following build requirements in the Jamfile:

```
<format>html:<auto-index-internal>on       # Use internally generated indexes.
<auto-index-type>index                     # Use <index>...</index> as the XML wrapper.
<format>html:<xsl:param>generate.index=0   # Don't let the XSL stylesheets generate indexes.
```

Getting Started and Tutorial

Step 1: Build the AutoIndex tool

cd into `tools/auto_index/build` and invoke bjam as:

```
bjam release
```

Optionally pass the name of the compiler toolset you want to use to bjam as well:

```
bjam release gcc
```

Now open up your `user-config.jam` file and at the end of the file add the line:

```
using auto-index : full-path-of-executable-auto-index.exe ;
```

Note

This declaration must go towards the end of `user-config.jam`, or in any case after the Boostbook initialisation.

Also note that Windows users must use forward slashes in the paths in `user-config.jam`

Finally note that `tools/auto_index/auto-index.jam` gets copied into the same directory as the rest of the Boost.Build tools (under `tools/build/v2/tools` in your main Boost tree): this is a temporary fix that will go away if the tool is accepted into Boost.

Caution

If you move to a new machine you will need to do this! An error message will warn about missing `auto-index.jam`.

Step 2: Configure Boost.Build jamfile to use AutoIndex

Assuming you have a Jamfile for building your documentation that looks something like:

```
boostbook standalone
    :
        mylibrary
    :
        # build requirements go here:
    ;
```

Then add the line:

```
using auto-index ; ↵
```

to the start of the Jamfile, and then add whatever auto-index options you want to the *build requirements section*, for example:

```
boostbook standalone
    :
        mylibrary
    :
        # Build requirements go here:

        # <auto-index>on (or off) one turns on (or off) indexing:
        <auto-index>on

        # Turns on (or off) auto-index-verbose for diagnostic info.
        # This is highly recommended until you have got all the many details correct!
        <auto-index-verbose>on

        # Choose the indexing method (separately for html and PDF) - see manual.
        # Choose indexing method for PDFs:
        <format>pdf:<auto-index-internal>off

        # Choose indexing method for html:
        <format>html:<auto-index-internal>on

        # Set the name of the script file to use (index.idx is popular):
        <auto-index-script>index.idx
        # Commands in the script file should all use RELATIVE PATHS
        # otherwise the script will not be portable to other machines.
        # Relative paths are normally taken as relative to the location
        # of the script file, but we can add a prefix to all
        # those relative paths using the <auto-index-prefix> feature.
        # The path specified by <auto-index-prefix> may be either relative or
        # absolute, for example the following will get us up to the boost root
        # directory for most Boost libraries:
        <auto-index-prefix>../../..

        # Tell Quickbook that it should enable indexing.
        <quickbook-define>enable_index ;

    ;
```

Available Indexing Options

The available options are:

<auto-index>off/on	Turns indexing of the document on, defaults to "off", so be sure to set this if you want AutoIndex invoked!
<auto-index-internal>off/on	Chooses whether AutoIndex creates the index itself (feature on), or whether it simply inserts the necessary DocBook markup so that the DocBook XSL stylesheets can create the index. Defaults to "off".

<auto-index-script>filename	Specifies the name of the script to load.
<auto-index-no-duplicates>off/on	When *on* AutoIndex will only index a term once in any given section, otherwise (the default) multiple index entries per term may be created if the term occurs more than once in the section.
<auto-index-section-names>off/on	When *on* AutoIndex will use create two index entries for each term found - one uses the term itself as the primary index key, the other uses the enclosing section name. When off the index entry that uses the section title is not created. Defaults to "on"
<auto-index-verbose>off/on	Defaults to "off". When turned on AutoIndex prints progress information - useful for debugging purposes during setup.
<auto-index-prefix>filename	Optionally specifies a directory to apply as a prefix to all relative file paths in the script file. You may wish to do this to reduce typing of pathnames, and/or where the paths can't be located relative to the script file location, typically if the headers are in the Boost trunk, but the script file is in Boost sandbox. For Boost standard library layout, `<auto-index-prefix>../../..` will get you back up to the 'root' of the Boost tree, so `!scan-path boost/mylibrary/` is where your headers will be, and `libs/mylibrary` for other files. Without a prefix all relative paths are relative to the location of the script file.
<auto-index-type>element-name	Specifies the name of the XML element to enclose internally generated indexes in: defaults to *section*, but could equally be *appendix* or *chapter* or some other block level element that has a formal title. The actual list of available options depends upon the document type, the following table gives the available options:

Document Type	**Available Index Types**
book	appendix index article chapter reference part
article	section appendix index sect1
library	See Chapter
chapter	section index sect1
part	appendix index article chapter reference
appendix	section index sect1
preface	section index sect1
qandadiv	N/A: an index would have to be placed within a subsection of the document.
qandaset	N/A: an index would have to be placed within a subsection of the document.
reference	N/A: an index would have to be placed within a subsection of the document.
set	N/A: an index would have to be placed within a subsection of the document.

Making AutoIndex optional

It is considerate to make the **use of auto-index optional** in Boost.Build, to allow users who do not have AutoIndex installed to still be able to build your documentation.

This also very convenient while you are refining your documentation, to allow you to decide to build indexes, or not: building indexes can take long time, if you are just correcting typos, you won't want to wait while you keep rebuilding the index!

One method of setting up optional AutoIndex support is to place all AutoIndex configuration in a the body of a bjam if statement:

```
if --enable-index in  [ modules.peek : ARGV ]
  {
     ECHO "Building the  docs with automatic index generation enabled." ;

     using auto-index ;
     project : requirements
          <auto-index>on
          <auto-index-script>index.idx

           ... other auto-index options here...

        # And tell Quickbook that it should enable indexing.
        <quickbook-define>enable_index
    ;
  }
  else
  {
    ECHO "Building the my_library docs with automatic index generation disabled. To get an auto-
index, try building with --enable-index." ;
  }
```

You will also need to add a conditional statement at the end of your Quickbook file, so that the index(es) is/are only added after the last section if indexing is enabled.

```
[? enable_index
'''
  <index/>
'''
]
```

To use this jamfile, you need to cd to your docs folder, for example:

```
cd \boost-sandbox\guild\mylibrary\libs\mylibrary\doc
```

and then run `bjam` to build the docs without index, for example:

```
bjam -a html > mylibrary_html.log
```

or with index(es)

```
bjam -a html --enable-index > mylibrary_html_index.log
```

Tip

Always send the output to a log file. It will contain of lot of stuff, but is invaluable to check if all has gone right, and else diagnose what has gone wrong.

Tip

A return code of 0 is not a reliable indication that you have got what you really want - inspecting the log file is the only certain way.

Tip

If you upgrade compiler version, for example MSVC from 9 to 10, then you may need to rebuild Autoindex to avoid what Microsoft call a 'side-by-side' error. And make sure that the autoindex.exe version you are using is the new one.

Step 3: Add indexes to your documentation

To add a single "include everything" index to a BoostBook/Docbook document, (perhaps generated using Quickbook, and perhaps also using Doxygen reference section), add `<index/>` at the location where you want the index to appear. The index will be rendered as a separate section called "Index" when the documentation is built.

To add multiple indexes, then give each one a title and set its `type` attribute to specify which terms will be included, for example to place the *function*, *class*, *macro* or *typedef* names indexed by *auto_index* in separate indexes along with a main "include everything" index as well, one could add:

```
<index type="class_name">
<title>Class Index</title>
</index>

<index type="typedef_name">
<title>Typedef Index</title>
</index>

<index type="function_name">
<title>Function Index</title>
</index>

<index type="macro_name">
<title>Macro Index</title>
</index>

<index/>
```

Note

Multiple indexes like this only work correctly if you tell the XSL stylesheets to honor the "type" attribute on each index as by default . You can turn the feature on by adding `<xsl:param>index.on.type=1` to your projects requirements in the Jamfile.

In Quickbook, you add the same markup but enclose it between two triple-tick ''' escapes, thus

```
'''<index/>''' ↵
```

If you are writing a Quickbook document with Doxygen reference documentation, the position of a `[xinclude autodoc.xml]` line in the Quickbook file determines the location of the Doxygen references section. You will almost certainly want this as well.

```
[xinclude autodoc.xml] # Using Doxygen reference documentation.
```

You can control the *displayed name* of the Doxygen reference section thus by adding to the end of the Doxygen autodoc section in your jamfile.

```
<xsl:param>"boost.doxygen.reftitle=Boost.mylibrary C++ Reference"
```

Note

AutoIndex knows nothing of the XML `xinclude` element, so if you're writing raw Docbook XML then you may want to run this through an XSL processor to flatten everything to one XML file before passing to AutoIndex. If you're using Boostbook or quickbook though, this all happens for you anyway, and AutoIndex will index the whole document including any sections included with `xinclude`.

If you are using auto-index's internal index generation on

```
<auto-index-internal>on
```

(usually recommended for HTML output, and not the default) then you can also decide what kind of XML wrapper the generated index is placed in. By default this is a `<section>...</section>` XML block (this replaces the original `<index>...</index>` block). However, depending upon the structure of the document and whether or not you want the index on a separate page - or else on the front page after the TOC - you may want to place the index inside a different type of XML block. For example if your document uses `<chapter>` top level content rather than `<section>`s then it may be preferable to place the index in a `<chapter>` or `<appendix>` block. You can also place the index inside an `<index>` block if you prefer, in which case the index does not appear in on a page of its own, but after the TOC in the HTML output.

You control the type of XML block used by setting the `<auto-index-type>element-name` attribute in the Jamfile, or via the `index-type=element-name` command line option to auto-index itself. For example, to place the index in an appendix, your Jamfile might look like:

```
using quickbook ;
using auto-index ;

xml mylibrary : mylibary.qbk ;
boostbook standalone
    :
        mylibrary
    :
        # auto-indexing is on:
        <auto-index>on

        # PDFs rely on the XSL stylesheets to generate the index:
        <format>pdf:<auto-index-internal>off

        # HTML output uses auto-index to generate the index:
        <format>html:<auto-index-internal>on

        # Name of script file to use:
        <auto-index-script>index.idx

        # Set the XML wrapper for HML Indexes to "appendix":
        <format>html:<auto-index-type>appendix

        # Turn on multiple index support:
        <xsl:param>index.on.type=1
```

Step 4: Create the .idx script file - to control what to terms to index

AutoIndex works by reading a script file that tells it what terms to index.

If your document contains largely text, and only a small amount of simple C++, and/or if you are using Doxygen to provide a C++ Reference section (that lists the C++ elements), and/or if you are relying on the indexing provided from a Standalone Doxygen Index, you may decide that a index is not needed and that you may only want the text part indexed.

But if you want C++ classes functions, typedefs and/or macros AutoIndexed, optionally, the script file also tells which other C++ files to scan.

At its simplest, it will scan one or more headers for terms that should be indexed in the documentation. So for example to scan "myheader.hpp" the script file would just contain:

```
!scan myheader.hpp
!scan mydetailsheader.hpp
```

Or, more likely in practice, so we can recursively scan through directories looking for all the files to scan whose **name matches a particular regular expression**:

```
!scan-path "boost/mylibrary" ".*.hpp" true ↵
```

Each argument is whitespace separated and can be optionally enclosed in "double quotes" (recommended).

The final *true* argument indicates that subdirectories in `/boost/math/mylibrary` should be searched recursively in addition to that directory.

Caution

The second *file-name-regex* argument is a regular expression and not a filename GLOB!

Caution

The scan-path is modified by any setting of <auto-index-prefix>. The examples here assume that this is `<auto-index-prefix>../../..` so that `boost/mylibrary` will be your header files, `libs/mylibrary/doc` will contain your documentation files and `libs/mylibrary/example` will contain your examples.

You could also scan any examples (.cpp) files, typically in folder `/mylibrary/lib/example`.

```
# All example source files, assuming no sub-folders.
!scan-path "libs/mylibrary/example" ".*\.cpp"
```

Often the *scan* or *scan-path* rules will bring in too many terms to search for, so we need to be able to exclude terms as well:

```
!exclude type
```

Which excludes the term "type" from being indexed.

We can also add terms manually:

```
foobar
```

will index occurrences of "foobar" and:

```
foobar \<\w*(foo|bar)\w*\>
```

will index any whole word containing either "foo" or "bar" within it, this is useful when you want to index a lot of similar or related words under one entry, for example:

```
reflex
```

Will only index occurrences of "reflex" as a whole word, but:

```
reflex \<reflex\w*\>
```

will index occurrences of "reflex", "reflexing" and "reflexed" all under the same entry *reflex*. You will very often need to use this to deal with plurals and other variants.

This inclusion rule can also restrict the term to certain sections, and add an index category that the term should belong to (so it only appears in certain indexes).

Finally the script can add rewrite rules, that rename section names that are automatically used as index entries. For example we might want to remove leading "A" or "The" prefixes from section titles when AutoIndex uses them as an index entry:

```
!rewrite-name "(?i)(?:A|The)\s+(.*)" "\1"
```

Step 5: Add Manual Index Entries to Docbook XML - Optional

If you add manual `<indexentry>` markup to your Docbook XML then these will be passed through unchanged. Please note however, that if you are using AutoIndex's internal index generation then it only recognises `<primary>` and `<secondary>` elements within the `<indexterm>`. `<tertiary>`, `<see>` and `<seealso>` elements are not currently recognised and auto-index will emit a warning if these are used.

Likewise none of the attributes which can be applied to these elements are used when AutoIndex generates the index itself, with the exception of the `<type>` attribute.

Step 6: Build the Docs

Using Boost.Build you build the docs with either:

```
bjam release > mylibrary_html.log
```

To build the html docs or:

```
bjam pdf release > mylibrary_pdf.log
```

To build the pdf.

During the build process you should see AutoIndex emit a message in the log file such as:

```
Indexing 990 terms... ↵
```

If you don't see that, or if it's indexing 0 terms then something is wrong!

Likewise when index generation is complete, AutoIndex will emit another message:

```
38 Index entries were created.
```

Again, if you see that 0 entries were created then something is wrong!

Examine the log file, and if the cause is not obvious, make sure that you have `<auto-index-verbose>on` and that any needed `!debug regular-expression` directives are in your script file.

Step 7: Iterate - to refine your index

Creating a good index is an iterative process, often the first step is just to add a header scanning rule to the script file and then generate the documentation and see:

- What's missing.
- What's been included that shouldn't be.
- What's been included under a poor name.

Further rules can then be added to the script to handle these cases and the next iteration examined, and so on.

Tip

If you don't understand why a particular term is (or is not) present in the index, try adding a *!debug regular-expression* directive to the script file.

Restricting which Sections are indexed for a particular term

You can restrict which sections are indexed for a particular term. So assuming that the docbook document has the usual hierarchical names for section ID's hierarchical names for section IDs(as Quickbook generates, for example), you can easily place a constraint on which sections are examined for a particular term.

For example, if you want to index occurrences of Lord Kelvin's name, but only in the introduction section, you might then add:

```
Kelvin "" ".*introduction.*"
```

to the script file, assuming that the section ID of the intro is "some_library_or_chapter_name.introduction".

This would avoid an index entry every time ° kelvin is found, something the user is unlikely to find helpful.

Script File (.idx) Reference

The following elements can occur in a script:

Comments and blank lines

Blank lines consisting of only whitespace are ignored, so are lines that **start with a #**. (But, of course, you can't append # comments onto the end of a line!).

Inclusion of Index terms

```
term [regular-expression1 [regular-expression2 [category]]]
```

term — *Term to index.*

The index term will form a primary entry in the Index with the section title(s) containing the term as secondary entries, and also will be used as a secondary entry beneath each of the section titles that the index term occurs in.

regular-expression1 — *Index term Searcher.*

An optional regular expression: each occurrence of the regular expression in the text of the document will result in one index term being emitted.

If the regular expression is omitted (default) or is "", then the *index term* itself will be used as the search text - and only occurrence of whole words matching *index term* will be indexed.

For example:

```
foobar
```

will index occurrences of "foobar" in any section, but

```
foobar \<\w*(foo|bar)\w*\>
```

will index any whole word containing either "foo" or "bar" within it. This is useful when you want to index a lot of similar or related words under one entry.

```
reflex
```

will only index occurrences of "reflex" as a whole word, but:

```
reflex \<reflex\w*\>
```

will index occurrences of "reflex", "reflexes", "reflexing" and "reflexed" ... all under the same entry reflex.

You will very often need to use this to deal with plurals and other variants.

regular-expression2

Section(s) Selector.

A constraint that specifies which sections are indexed for *term*: only if the ID of the section matches *regular-expression2* exactly will that section be indexed for occurrences of *term*.

For example, to limit indexing to just **one specific section** (but not sub-sections below):

```
myclass "" "mylib.examples"
```

For example, to limit indexing to specific sections, **and sub-sections below**:

```
myclass "" "mylib.examples.*"
```

will index occurrences of "myclass" as a whole word, but only in sections whose section ID **begins** "mylib.examples",

```
myclass "\<myclass\w*\>" "mylib.examples.*"
```

and will also index plurals myclass, myclasses, myclasss ...

while:

```
myclass "" "(?!mylib.introduction.*).*"
```

will index occurrences of "myclass" in any section, except those whose section IDs begin "mylib.introduction".

```
reflex "\<reflex\w*\>" "mylib.introduction.*"
```

If this third section selection field is omitted (the default) or is "", then **all sections** are indexed for this term.

category

Index Category Constraint.

Optionally a category to place occurrences of *index term* in. If you have multiple indexes then this is the name assigned to the indexes "type" attribute.

For example:

myclass "" "" class_name

Will index occurances of *myclass* and place them in the class-index if there is one.

Source File Scanning

```
!scan source-file-name
```

Scans the C/C++ source file *source-file-name* for definitions of *function*s, *class*s, *macro*s or *typedef*s and makes each of these a term to be indexed. Terms found are assigned to the index category "function_name", "class_name", "macro_name" or "typedef_name" depending on how they were seen in the source file. These may then be included in a specialised index whose "type" attribute has the same category name.

Important

When actually indexing a document, the scanner will not index just any old occurrence of the terms found in the source files. Instead it searches for class definitions or function or typedef declarations. This reduces the number of spurious matches placed in the index, but may also miss some legitimate terms: refer to the *define-scanner* command for information on how to change this.

Directory and Source File Scanning

```
!scan-path directory-name file-name-regex [recurse]
```

directory-name	The directory to scan: this should be a path relative to the script file (or to the path specified with the prefix=path option on the command line) and should use all forward slashes in its file name.
file-name-regex	A regular expression: any file in the directory whose name matches the regular expression will be scanned for terms to index.
recurse	An optional boolean value - either "true" or "false" - that indicates whether to recurse into subdirectories. This defaults to "false"

Excluding Terms

```
!exclude term-list
```

Excludes all the terms in whitespace separated *term-list* from being indexed. This should be placed *after* any *!scan* or *!scan-path* rules which may result in the terms becoming included. In other words this removes terms from the scanners internal list of things to index.

Rewriting Section Names

```
!rewrite-id regular-expression new-name
```

regular-expression	A regular expression: all section ID's that match the expression exactly will have index entries *new-name* instead of their title(s).
new-name	The name that the section will appear under in the index.

```
!rewrite-name regular-expression format-text
```

regular-expression	A regular expression: all sections whose titles match the regular expression exactly, will have index entries composed of the regular expression match combined with the regex format string *format-text*.
format-text	The Perl-style format string used to reformat the title.

For example:

```
!rewrite-name "(?:A|An|The)\s+(.*)" "\1"
```

Will remove any leading "A", "An" or "The" from all index entries - thus preventing lots of entries under "The" etc!

Defining or Changing the File Scanners

```
!define-scanner type file-search-expression xml-regex-formatter term-formatter id-filter file↵
name-filter
```

When a source file is scanned using the `!scan` or `!scan-path` rules, then the file is searched using a series of regular expressions to look for classes, functions, macros or typedefs that should be indexed. A set of default regular expressions are provided for this (see below), but sometimes you may want to replace the defaults, or add new scanners. The arguments to this rule are:

type	The *type* to which items found using this rule will assigned, index terms created from the source file and then found in the XML, will have the type attribute set to this value, and may then appear in a specialized index with the same type attribute
file-search-expression	A regular expression that is used to scan the source file for index terms, the result of a match against this expression will be transformed by the next two arguments.
xml-regex-formatter	A regular expression format string that extracts the salient information from whatever matched the *file-search-expression* in the source file, and creates *a new regular expression* that will be used to search the document being indexed for occurrences of this index term.
term-formatter	A regular expression format string that extracts the salient information from whatever matched the *file-search-expression* in the source file, and creates the index term that will appear in the index.
id-filter	Optional. A regular expression that restricts the section-id's that are searched in the document being indexed: only sections whose ID attribute matches this expression exactly will be considered for indexing terms found by this scanner.
filename-filter	Optional. A regular expression that restricts which files are scanned by this scanner: only files whose file name matches this expression exactly will be scanned for index terms to use. Note that the filename matched against this may well be an absolute path, and contain either forward or backward slash path separators.

If, when the first file is scanned, there are no scanners whose *type* is "class_name", "typedef_name", "macro_name" or "function_name", then the defaults are installed. These are equivalent to:

```
!define-scanner class_name "^[[:space:]]*(tem↵
pla[[:space:]]*<[^;:{]*>[[:space:]]*(class|struct)[[:space:]]*(\w+([[:blank:]]*([^)]*))?[[:space:]]*)*(\w+)[[:space:]]*(<[^;:{]*>)?[[:space:]]*(\{|:[^;\{()]*{)""(?class|struct)[^;{]*\<\>\s[^;{]*\{"\5
!define-scanner typedef_name "typedef[^;{}#]+?(\w+)\s*;"  "typedef[^;]+\\<\1\\>\\s*;" "\1"
!define-scanner "macro_name" "^\s*#\s*define\s+(\w+)" "\\<\1\\>" "\1"
!define-scanner "func↵
tion_name" "\w+\s+(\w+)\s*\([^\)]*\)\s*[;{]" "\\<\\w+\\>\\s+\\<\1\\>\\s*\\([^;{]*\\)\\s*[;{]" "\1"
```

Note that these defaults are not installed if you have provided your own versions with these *type* names. In this case if you want the default scanners to be in effect as well as your own, you should include the above in your script file. It is also perfectly allowable to have multiple scanners with the same *type*, but with the other fields differing.

Finally you should note that the default scanners are quite strict in what they will find, for example the class scanner will only create index entries for classes that have class definitions of the form:

```
class my_class : public base_classes
{
   // etc
```

In the documentation, so that simple mentions of the class name will *not* get indexed, only the class synopsis if there is one. If this isn't how you want things, then include the *class_name* scanner definition above in your script file, and change the *xml-regex-formatter* field to something more permissive, for example:

```
!define-scanner class_name "^[[:space:]]*(tem↵
plate[[:space:]]*<[^;:{]+>[[:space:]]*)?(class|struct)[[:space:]]*(\<\w*\>([[:blank:]]*\([^)]*\))?[[:space:]]*)*(\<\w*\>)[[:space:]]*(<[^;:{]+>)?[[:space:]]*(\{|:[^;\{()]*\{)" "\\<\5\\>" \5
```

Will look for *any* occurrence of whatever class names the scanner may find in the documentation.

Debugging scanning

If you see a term in the index, and you don't understand why it's there, add a *debug* directive:

```
!debug regular-expression
```

Now, whenever *regular-expression* matches either the found index term, or the section title it appears in, or the *type* field of a scanner, then some diagnostic information will be printed that will look something like:

```
Debug term found, in block with ID: spirit.qi.reference.parser_concepts.parser
Current section title is: Notation
The main index entry will be : Notation
The indexed term is: parser
The search regex is: [P|p]arser
The section constraint is: .qi.reference.parser_concepts.
The index type for this entry is: qi_index
```

This can produce a lot of output in your log file, but until you are satisfied with your file selection and scanning process, it is worth switching it on.

Understanding The AutoIndex Workflow

1. Load the script file (usually index.idx) and process it one line at a time, producing one or more index term per (non-comment) line.

2. Reading all lines builds a list of *terms to index*. Some of those may be terms defined (by you) directly in the script file, others may be terms found by scanning C++ header and source files that were specified by the *!scan-path* directive.

3. Once the complete list of *terms to index* is complete, it loads the Docbook XML file. (If this comes from Quickbook/Doxygen/Boostbook/Docbook then this is the complete documentation after conversion to Docbook format).

4. AutoIndex builds an internal Document Object Model (DOM) of the Docbook XML. This internal representation then gets scanned for occurrences of the *terms to index*. This scanning works at the XML paragraph level (or equivalent sibling such as a table or code block) - so all the XML encoding within a paragraph gets flattened to plain text.
   This flattening means the regular expressions used to search for *terms to index* can find anything that is completely contained within a paragraph (or code block etc).

5. For each term found then an *indexterm* Docbook element is inserted into the Document Object Model (DOM) (provided internal index generation is off),

6. Also the AutoIndex's internal index representation gets updated.

7. Once the whole XML document has been indexed, then, if autoindex has been instructed to generate the index itself, it creates the necessary XML and inserts this into the Document Object Model (DOM).

8. Finally the whole Document Object Model (DOM) is written out as a new Docbook XML file, and normal processing of this continues via the XSL stylesheets (with xsltproc) to actually build the final human-readable docs.

XML Handling

AutoIndex is rather simplistic in its handling of XML:

- When indexing a document, all block content at the paragraph level gets collapsed into a single string for matching against the regular expressions representing each index term. In other words, for the most part, you can assume that you're indexing plain text when writing regular expressions.
- Named XML entities for &, ", ', < or > are converted to their corresponding characters before indexing a section of text. However, decimal or hex escape sequences are not currently converted.
- Index terms are assumed to be plain text (whether they originate from the script file or from scanning source files) and the characters &, ", < and > will be escaped to & " < and > respectively.

Command Line Reference

The following command line options are supported by auto_index:

in=infilename	Specifies the name of the XML input file to be indexed.
out=outfilename	Specifies the name of the new XML file to create.
scan=source-filename	Specifies that *source-filename* should be scanned for terms to index.
script=script-filename	Specifies the name of the script file to process.
--no-duplicates	If a term occurs more than once in the same section, then include only one index entry.
--internal-index	Specifies that auto_index should generate the actual indexes rather than inserting `<indexterm>`s and leaving index generation to the XSL stylesheets.
--no-section-names	Prevents auto_index from using section names as index entries.
prefix=pathname	Specifies a directory to apply as a prefix to all relative file paths in the script file.
index-type=element-name	Specifies the name of the XML element to enclose internally generated indexes in: defaults to *section*, but could equally be *appendix* or *chapter* or some other block level element that has a formal title.