Quickbook 1.4

Joel de Guzman
Eric Niebler

Table of Contents

Introduction

"Why program by hand in five days what you can spend five years of your life automating?"

-- Terrence Parr, author ANTLR/PCCTS

Well, QuickBook started as a weekend hack. It was originally intended to be a sample application using Spirit. What is it? What you are viewing now, this documentation, is autogenerated by QuickBook. These files were generated from one master:

quickbook.qbk

Originally named QuickDoc, this funky tool that never dies, evolved into a funkier tool thanks to Eric Niebler who resurrected the project making it generate BoostBook instead of HTML. The BoostBook documentation format is an extension of DocBook, an SGML or XML based format for describing documentation.

QuickBook is a WikiWiki style documentation tool geared towards C++ documentation using simple rules and markup for simple formatting tasks. QuickBook extends the WikiWiki concept. Like the WikiWiki, QuickBook documents are simple text files. A single QuickBook document can generate a fully linked set of nice HTML and PostScript/PDF documents complete with images and syntax- colorized source code.

Features include:

• generate BoostBook xml, to generate HTML, PostScript and PDF

• simple markup to link to Doxygen-generated entities

• macro system for simple text substitution

- simple markup for italics, bold, preformatted, blurbs, code samples, tables, URLs, anchors, images, etc.

- automatic syntax coloring of code samples

- CSS support

Change Log

Version 1.3

- Quickbook file inclusion [include].

- Better xml output (pretty layout). Check out the generated XML.

- Regression testing facility: to make sure your document will always be compatible (full backward compatibility) regardless of changes to QuickBook.

- Code cleanup and refactoring.

- Allow phrase markup in the doc-info.

- Preformatted code blocks via ``code`` (double ticks) allows code in tables and lists, for example.

- Quickbook versioning; allows full backward compatibility. You have to add [quickbook 1.3] to the doc-info header to enable the new features. Without this, QuickBook will assume that the document is a pre-1.3 document.

- Better (intuitive) paragraph termination. Some markups may terminate a paragraph. Example:

```
[section x]
blah...
[endsect]
```

- Fully qualified section and headers. Subsection names are concatenated to the ID to avoid clashing. Example: doc_name.sect_name.sub_sect_name.sub_sub_sect_name

- Better   and whitespace handling in code snippets.

- [xinclude] fixes up the relative path to the target XML file when input_directory is not the same as the output_directory.

- Allow untitled tables.

- Allow phrase markups in section titles.

- Allow escaping back to QuickBook from code, code blocks and inline code.

- Footnotes, with the [footnote This is the footnote] syntax.

- Post-processor bug fix for escaped XML code that it does not recognize.

- Replaceable, with the [~replacement] syntax.

Version 1.4

- Generic Headers

- Code changes to allow full recursion (i.e. Collectors and push/pop functions)

- Various code cleanup/maintenance

- Templates!

- [conceptref] for referencing BoostBook <concept> entities.

- Allow escape of spaces. The escaped space is removed from the output. Syntax: \ .

- Nested comments are now allowed.

- Quickbook blocks can nest inside comments.

- Import facility.

- Callouts on imported code

- Simple markups can now span a whole block.

- Blurbs, Admonitions and table cells (see Tables) may now contain paragraphs.

- \n and [br] are now deprecated.

- Conditional Generation. Ala C++ #ifdef.

- Searching of included and imported files in an extensible search path with --include-path (-I) option.

Syntax Summary

A QuickBook document is composed of one or more blocks. An example of a block is the paragraph or a C++ code snippet. Some blocks have special mark-ups. Blocks, except code snippets which have their own grammar (C++ or Python), are composed of one or more phrases. A phrase can be a simple contiguous run of characters. Phrases can have special mark-ups. Marked up phrases can recursively contain other phrases, but cannot contain blocks. A terminal is a self contained block-level or phrase-level element that does not nest anything.

Blocks, in general, are delimited by two end-of-lines (the block terminator). Phrases in each block cannot contain a block terminator. This way, syntax errors such as un-matched closing brackets do not go haywire and corrupt anything past a single block.

Comments

Can be placed anywhere.

```
[/ comment (no output generated) ]
```

```
[/ comments can be nested [/ some more here] ]
```

```
[/ Quickbook blocks can nest inside comments. [*Comment this out too!] ]
```

Phrase Level Elements

Font Styles

```
['italic], [*bold], [_underline], [^teletype], [-strikethrough]
```

will generate:

italic, **bold**, underline, `teletype`, ~~strikethrough~~

Like all non-terminal phrase level elements, this can of course be nested:

```
[*['bold-italic]]
```

will generate:

bold-italic

Replaceable

When you want content that may or must be replaced by the user, use the syntax:

```
[~replacement]
```

This will generate:

replacement

Quotations

```
["A question that sometimes drives me hazy: am I or are the others crazy?]--Einstein
```

will generate:

"A question that sometimes drives me hazy: am I or are the others crazy?"--Einstein

Note the proper left and right quote marks. Also, while you can simply use ordinary quote marks like "quoted", our quotation, above, will generate correct DocBook quotations (e.g. <quote>quoted</quote>).

Like all phrase elements, quotations may be nested. Example:

```
["Here's the rule for bargains: ["Do other men, for they would do you.] That's
the true business precept.]
```

will generate:

"Here's the rule for bargains: 'Do other men, for they would do you.' That's the true business precept."

Simple formatting

Simple markup for formatting text, common in many applications, is now supported:

```
/italic/, *bold*, _underline_, =teletype=
```

will generate:

italic, **bold**, underline, teletype

Unlike QuickBook's standard formatting scheme, the rules for simpler alternatives are much stricter [1] .

• Simple markups cannot nest. You can combine a simple markup with a nestable markup.

• Simple markups cannot contain any other form of quickbook markup.

---

[1] Thanks to David Barrett, author of Qwiki, for sharing these samples and teaching me these obscure formatting rules. I wasn't sure at all if Spirit, being more or less a formal EBNF parser, can handle the context sensitivity and ambiguity.

- A non-space character must follow the leading markup

- A non-space character must precede the trailing markup

- A space or a punctuation must follow the trailing markup

- If the matching markup cannot be found within a block, the formatting will not be applied. This is to ensure that un-matched formatting markups, which can be a common mistake, does not corrupt anything past a single block. We do not want the rest of the document to be rendered bold just because we forgot a trailing '*'. A single block is terminated by two end of lines or the close bracket: ']'.

- A line starting with the star will be interpreted as an unordered list. See Unordered lists.

Table 1. More Formatting Samples

Markup	Result
`*Bold*`	**Bold**
`*Is bold*`	**Is bold**
`* Not bold* *Not bold * * Not bold *`	* Not bold* *Not bold * * Not bold *
`This*Isn't*Bold (no bold)`	This*Isn't*Bold (no bold)
`(*Bold Inside*) (parenthesis not bold)`	(**Bold Inside**) (parenthesis not bold)
`*(Bold Outside)* (parenthesis bold)`	**(Bold Outside)** (parenthesis bold)
`3*4*5 = 60 (no bold)`	3*4*5 = 60 (no bold)
`3 * 4 * 5 = 60 (no bold)`	3 * 4 * 5 = 60 (no bold)
`3 *4* 5 = 60 (4 is bold)`	3 **4** 5 = 60 (4 is bold)
`*This is bold* this is not *but this is*`	**This is bold** this is not **but this is**
`*This is bold*.`	**This is bold**.
`*B*. (bold B)`	**B**. (bold B)
`['*Bold-Italic*]`	***Bold-Italic***
`*side-by*/-side/`	**side-by**-*side*

As mentioned, simple markups cannot go past a single block. The text from "have" to "full" in the following paragraph will be rendered as bold:

```
Baa baa black sheep, *have you any wool?
Yes sir, yes sir, three bags full!*
One for the master, one for the dame,
And one for the little boy who lives down the lane.
```

Baa baa black sheep, **have you any wool? Yes sir, yes sir, three bags full!** One for the master, one for the dame, And one for the little boy who lives down the lane.

But in the following paragraph, bold is not applied:

```
Baa baa black sheep, *have you any wool?
Yes sir, yes sir, three bags full!
One for the master, one for the dame,
And one for the little boy who lives down the lane.
```

Baa baa black sheep, *have you any wool? Yes sir, yes sir, three bags full! One for the master, one for the dame, And one for the little boy who lives down the lane.

Inline code

Inlining code in paragraphs is quite common when writing C++ documentation. We provide a very simple markup for this. For example, this:

```
This text has inlined code `int main() { return 0; }` in it.
```

will generate:

This text has inlined code `int main() { return 0; }` in it. The code will be syntax highlighted.

> **Note**
>
> We simply enclose the code with the tick: "`", not the single quote: "'". Note too that `some code` is preferred over [^some code].

Code blocks

Preformatted code simply starts with a space or a tab (See Code). However, such a simple syntax cannot be used as phrase elements in lists (See Ordered lists and Unordered lists), tables (See Tables), etc. Inline code (see above) can. The problem is, inline code does not allow formatting with newlines, spaces, and tabs. These are lost.

We provide a phrase level markup that is a mix between the two. By using the double-tick, instead of the single-tick, we are telling QuickBook to use preformatted blocks of code. Example:

```
``
    #include <iostream>

    int main()
    {
        std::cout << "Hello, World!" << std::endl;
        return 0;
    }
``
```

will generate:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Source Mode

If a document contains more than one type of source code then the source mode may be changed dynamically as the document is processed. All QuickBook documents are initially in C++ mode by default, though an alternative initial value may be set in the Document section.

To change the source mode, use the `[source-mode]` markup, where `source-mode` is one of the supported modes. For example, this:

```
Python's [python] `import` is rather like C++'s [c++] `#include`. A
C++ comment `// looks like this` whereas a Python comment [python]
`# looks like this`.
```

will generate:

Python's `import` is rather like C++'s `#include`. A C++ comment `// looks like this` whereas a Python comment `#looks like this`.

Table 2. Supported Source Modes

Mode	Source Mode Markup
C++	`[c++]`
Python	`[python]`

> **Note**
>
> The source mode strings are lowercase.

line-break

```
[br]
```

> **Warning**
>
> `[br]` is now deprecated. Blurbs, Admonitions and table cells (see Tables) may now contain paragraphs.

Anchors

```
[#named_anchor]
```

A named anchor is a hook that can be referenced by a link elsewhere in the document. You can then reference an anchor with `[link named_anchor Some link text]`. See Anchor links, Section and Heading.

Links

```
[@http://www.boost.org this is [*boost's] website....]
```

will generate:

this is **boost's** website....

URL links where the link text is the link itself is common. Example:

```
see http://spirit.sourceforge.net/
```

so, when the text is absent in a link markup, the URL is assumed. Example:

```
see [@http://spirit.sourceforge.net/]
```

will generate:

see http://spirit.sourceforge.net/

Anchor links

You can link within a document using:

```
[link section_id.normalized_header_text The link text]
```

See sections Section and Heading for more info.

refentry links

In addition, you can link internally to an XML refentry like:

```
[link xml.refentry The link text]
```

This gets converted into `<link linkend="xml.refentry">The link text</link>`.

Like URLs, the link text is optional. If this is not present, the link text will automatically be the refentry. Example:

```
[link xml.refentry]
```

This gets converted into `<link linkend="xml.refentry">xml.refentry</link>`.

Code Links

If you want to link to a function, class, member, enum, concept, global, or header in the reference section, you can use:

```
[funcref fully::qualified::function_name The link text]
[classref fully::qualified::class_name The link text]
[memberref fully::qualified::member_name The link text]
[enumref fully::qualified::enum_name The link text]
[macroref MACRO_NAME The link text]
[conceptref ConceptName The link text]
[headerref path/to/header.hpp The link text]
[globalref fully::qualified::global The link text]
```

Again, the link text is optional. If this is not present, the link text will automatically be the function, class, member, enum, macro, concept, global, or header name. Example:

```
[classref boost::bar::baz]
```

would have "boost::bar::baz" as the link text.

Escape

The escape mark-up is used when we don't want to do any processing.

```
'''
escape (no processing/formatting)
'''
```

Escaping allows us to pass XML markup to BoostBook or DocBook. For example:

```
'''
<emphasis role="bold">This is direct XML markup</emphasis>
'''
```

This is direct XML markup

> ⚠ **Important**
>
> Be careful when using the escape. The text must conform to BoostBook/DocBook syntax.

Single char escape

The backslash may be used to escape a single punctuation character. The punctuation immediately after the backslash is passed without any processing. This is useful when we need to escape QuickBook punctuations such as [and]. For example, how do you escape the triple quote? Simple: \'\'\'

\n has a special meaning. It is used to generate line breaks.

> ⊗ **Warning**
>
> \n and [br] are now deprecated. Blurbs, Admonitions and table cells (see Tables) may now contain paragraphs.

The escaped space: \   also has a special meaning. The escaped space is removed from the output.

Images

```
[$image.jpg]
```

Footnotes

As of version 1.3, QuickBook supports footnotes. Just put the text of the footnote in a `[footnote]` block, and the text will be put at the bottom of the current page. For example, this:

```
[footnote A sample footnote]
```

will generate this [2] .

Macro Expansion

```
__a_macro_identifier__
```

See Macros for details.

Template Expansion

```
[a_template_identifier]
```

See Templates for details.

Conditional Generation

Like C++ `#ifdef`, you can generate phrases depending on the presence of a macro. Example:

```
[? __to_be__ To be or not to be]
```

Here, the phrase "To be or not to be" will only be generated if the macro symbol __to_be__ has been previously defined. The phrase above will not do anything since we haven't defined __to_be__. Now, let's define the symbol:

```
[def __to_be__]
```

And try again:

To be or not to be

Yes! [3]

Block Level Elements

Document

Every document must begin with a Document Info section, which should look like this:

---

[2] A sample footnote
[3] Conditional Generation makes quickbook turing complete.

```
[document-type The Document Title
    [quickbook 1.3]
    [version 1.0]
    [id the_document_name]
    [dirname the_document_dir]
    [copyright 2000 2002 2003 Joe Blow, Jane Doe]
    [purpose The document's reason for being]
    [category The document's category]
    [authors [Blow, Joe], [Doe, Jane]]
    [license The document's license]
    [source-mode source-type]
]
```

Where document-type is one of:

• book

• article

• library

• chapter

• part

• appendix

• preface

• qandadiv

• qandaset

• reference

• set

quickbook 1.3 declares the version of quickbook the document is written for. In its absence, version 1.1 is assumed.

version, id, dirname, copyright, purpose, category, authors, license, last-revision and source-mode are optional information.

source-type is a lowercase string setting the initial Source Mode. If the source-mode field is omitted, a default value of c++ will be used.

Section

Starting a new section is accomplished with:

```
[section:id The Section Title]
```

where *id* is optional. id will be the filename of the generated section. If it is not present, "The Section Title" will be normalized and become the id. Valid characters are a-z, A-Z, 0-9 and _. All non-valid characters are converted to underscore and all upper-case are converted to lower case. Thus: "The Section Title" will be normalized to "the_section_title".

End a section with:

```
[endsect]
```

Sections can nest, and that results in a hierarchy in the table of contents.

xinclude

You can include another XML file with:

```
[xinclude file.xml]
```

This is useful when file.xml has been generated by Doxygen and contains your reference section.

Paragraphs

Paragraphs start left-flushed and are terminated by two or more newlines. No markup is needed for paragraphs. QuickBook automatically detects paragraphs from the context. Block markups [section, endsect, h1, h2, h3, h4, h5, h6, blurb, (block-quote) ':', pre, def, table and include] may also terminate a paragraph.

This is a new paragraph...

Lists

Ordered lists

```
# One
# Two
# Three
```

will generate:

1. One

2. Two

3. Three

List Hierarchies

List hierarchies are supported. Example:

```
# One
# Two
# Three
    # Three.a
    # Three.b
    # Three.c
# Four
    # Four.a
        # Four.a.i
        # Four.a.ii
# Five
```

will generate:

1. One

2. Two

3. Three

    a. Three.a

    b. Three.b

    c. Three.c

4. Fourth

    a. Four.a

        i. Four.a.i

        ii. Four.a.ii

5. Five

Long List Lines

Long lines will be wrapped appropriately. Example:

```
# A short item.
# A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
# A short item.
```

1. A short item.

2. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item.

3. A short item.

Unordered lists

```
* First
* Second
* Third
```

will generate:

- First

- Second

- Third

Mixed lists

Mixed lists (ordered and unordered) are supported. Example:

```
# One
# Two
# Three
    * Three.a
    * Three.b
    * Three.c
# Four
```

will generate:

1. One

2. Two

3. Three

   • Three.a

   • Three.b

   • Three.c

4. Four

And...

```
# 1
    * 1.a
        # 1.a.1
        # 1.a.2
    * 1.b
# 2
    * 2.a
    * 2.b
        # 2.b.1
        # 2.b.2
            * 2.b.2.a
            * 2.b.2.b
```

will generate:

1. 1

   • 1.a

     a. 1.a.1

     b. 1.a.2

   • 1.b

2. 2

   • 2.a

   • 2.b

     a. 2.b.1

     b. 2.b.2

- 2.b.2.a

- 2.b.2.b

Code

Preformatted code starts with a space or a tab. The code will be syntax highlighted according to the current Source Mode:

```
#include <iostream>

int main()
{
    // Sample code
    std::cout << "Hello, World\n";
    return 0;
}
```

```
import cgi

def cookForHtml(text):
    '''"Cooks" the input text for HTML.'''

    return cgi.escape(text)
```

Macros that are already defined are expanded in source code. Example:

```
[def __array__ [@http://www.boost.org/doc/html/array/reference.html array]]
[def __boost__ [@http://www.boost.org/libs/libraries.htm boost]]

    using __boost__::__array__;
```

Generates:

```
using boost::array;
```

Escaping Back To QuickBook

Inside code, code blocks and inline code, QuickBook does not allow any markup to avoid conflicts with the target syntax (e.g. c++). In case you need to switch back to QuickBook markup inside code, you can do so using a language specific *escape-back* delimiter. In C++ and Python, the delimiter is the double tick (back-quote): "``" and "``". Example:

```
void ``[@http://en.wikipedia.org/wiki/Foo#Foo.2C_Bar_and_Baz foo]``()
{
}
```

Will generate:

```
void foo()
{
}
```

When escaping from code to QuickBook, only phrase level markups are allowed. Block level markups like lists, tables etc. are not allowed.

Preformatted

Sometimes, you don't want some preformatted text to be parsed as C++. In such cases, use the `[pre ... ]` markup block.

```
[pre

    Some *preformatted* text                        Some *preformatted* text

        Some *preformatted* text            Some *preformatted* text

            Some *preformatted* text    Some *preformatted* text

]
```

Spaces, tabs and newlines are rendered as-is. Unlike all quickbook block level markup, pre (and Code) are the only ones that allow multiple newlines. The markup above will generate:

```
Some preformatted text                        Some preformatted text

    Some preformatted text            Some preformatted text

        Some preformatted text    Some preformatted text
```

Notice that unlike Code, phrase markup such as font style is still permitted inside `pre` blocks.

Blockquote

```
[:sometext...]
```

> Indents the paragraph. This applies to one paragraph only.

Admonitions

```
[note This is a note]
[tip This is a tip]
[important This is important]
[caution This is a caution]
[warning This is a warning]
```

generates DocBook admonitions:

Note

This is a note

Tip

This is a tip

> ⚠️ **Important**
>
> This is important

> ⚠️ **Caution**
>
> This is a caution

> ✖️ **Warning**
>
> This is a warning

These are the only admonitions supported by DocBook. So, for example `[information This is some information]` is unlikely to produce the desired effect.

Headings

```
[h1 Heading 1]
[h2 Heading 2]
[h3 Heading 3]
[h4 Heading 4]
[h5 Heading 5]
[h6 Heading 6]
```

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

Headings 1-3 [h1 h2 and h3] will automatically have anchors with normalized names with `name="section_id.normalized_header_text"` (i.e. valid characters are `a-z`, `A-Z`, `0-9` and `_`. All non-valid characters are converted to underscore and all upper-case are converted to lower-case. For example: Heading 1 in section Section 2 will be normalized to `section_2.heading_1`). You can use:

```
[link section_id.normalized_header_text The link text]
```

to link to them. See Anchor links and Section for more info.

Generic Heading

In cases when you don't want to care about the heading level (1 to 6), you can use the *Generic Heading*:

```
[heading Heading]
```

The *Generic Heading* assumes the level, plus one, of the innermost section where it is placed. For example, if it is placed in the outermost section, then, it assumes *h2*.

Headings are often used as an alternative to sections. It is used particularly if you do not want to start a new section. In many cases, however, headings in a particular section is just flat. Example:

```
[section A]
[h2 X]
[h2 Y]
[h2 Z]
[endsect]
```

Here we use h2 assuming that section A is the outermost level. If it is placed in an inner level, you'll have to use h3, h4, etc. depending on where the section is. In general, it is the section level plus one. It is rather tedious, however, to scan the section level everytime. If you rewrite the example above as shown below, this will be automatic:

```
[section A]
[heading X]
[heading Y]
[heading Z]
[endsect]
```

They work well regardless where you place them. You can rearrange sections at will without any extra work to ensure correct heading levels. In fact, with *section* and *heading*, you have all you need. *h1..h6* becomes redundant. *h1..h6* might be deprecated in the future.

Macros

```
[def macro_identifier some text]
```

When a macro is defined, the identifier replaces the text anywhere in the file, in paragraphs, in markups, etc. macro_identifier is a string of non- white space characters except ']'. A macro may not follow an alphabetic character or the underscore. The replacement text can be any phrase (even marked up). Example:

```
[def sf_logo [$http://sourceforge.net/sflogo.php?group_id=28447&type=1]]
sf_logo
```

Now everywhere the sf_logo is placed, the picture will be inlined.

> ### Tip
>
> It's a good idea to use macro identifiers that are distinguishable. For instance, in this document, macro identifiers have two leading and trailing underscores (e.g. `__spirit__`). The reason is to avoid unwanted macro replacement.

Links (URLS) and images are good candidates for macros. **1**) They tend to change a lot. It is a good idea to place all links and images in one place near the top to make it easy to make changes. **2**) The syntax is not pretty. It's easier to read and write, e.g. `__spirit__` than `[@http://spirit.sourceforge.net Spirit]`.

Some more examples:

```
[def :-)          [$theme/smiley.png]]
[def __spirit__   [@http://spirit.sourceforge.net Spirit]]
```

(See Images and Links)

Invoking these macros:

```
Hi __spirit__  :-)
```

will generate this:

Hi Spirit

Predefined Macros

Quickbook has some predefined macros that you can already use.

Table 3. Predefined Macros

Macro	Meaning	Example
__DATE__	Today's date	2009-Feb-19
__TIME__	The current time	05:23:57 PM
__FILENAME__	Quickbook source filename	d:\data\boost\branches\1.38.0\doc\pdf\..\..\tools\quick-book\doc\quickbook.qbk

Templates

Templates provide a more versatile text substitution mechanism. Templates come in handy when you need to create parameterizable, multi-line, boilerplate text that you specify once and expand many times. Templates accept one or more arguments. These arguments act like place-holders for text replacement. Unlike simple macros, which are limited to phrase level markup, templates can contain block level markup (e.g. paragraphs, code blocks and tables).

Example template:

```
[template person[name age what]

Hi, my name is [name]. I am [age] years old. I am a [what].

]
```

Template Identifier

Template identifiers can either consist of:

• An initial alphabetic character or the underscore, followed by zero or more alphanumeric characters or the underscore. This is similar to your typical C/C++ identifier.

• A single character punctuation (a non-alphanumeric printable character)

Formal Template Arguments

Template formal arguments are identifiers consisting of an initial alphabetic character or the underscore, followed by zero or more alphanumeric characters or the underscore. This is similar to your typical C/C++ identifier.

A template formal argument temporarily hides a template of the same name at the point where the template is expanded. Note that the body of the `person` template above refers to `name age` and `what` as `[name] [age]` and `[what]`. `name age` and `what` are actually templates that exist in the duration of the template call.

Template Body

The template body can be just about any QuickBook block or phrase. There are actually two forms. Templates may be phrase or block level. Phrase templates are of the form:

```
[template sample[arg1 arg2...argN] replacement text... ]
```

Block templates are of the form:

```
[template sample[arg1 arg2...argN]
replacement text...
]
```

The basic rule is as follows: if a newline immediately follows the argument list, then it is a block template, otherwise, it is a phrase template. Phrase templates are typically expanded as part of phrases. Like macros, block level elements are not allowed in phrase templates.

Template Expansion

You expand a template this way:

```
[template_identifier arg1..arg2..arg3]
```

At template expansion, you supply the actual arguments. The template will be expanded with your supplied arguments. Example:

```
[person James Bond..39..Spy]
[person Santa Clause..87..Big Red Fatso]
```

Which will expand to:

Hi, my name is James Bond. I am 39 years old. I am a Spy.

Hi, my name is Santa Clause. I am 87 years old. I am a Big Red Fatso.

> ⚠️ **Caution**
>
> A word of caution: Templates are recursive. A template can call another template or even itself, directly or indirectly. There are no control structures in QuickBook (yet) so this will always mean infinite recursion. QuickBook can detect this situation and report an error if recursion exceeds a certain limit.

Each actual argument can be a word, a text fragment or just about any QuickBook phrase. Arguments are separated by the double dot `".."` and terminated by the close parenthesis.

Nullary Templates

Nullary templates look and act like simple macros. Example:

```
[template alpha[]'''&#945;''']
[template beta[]'''&#946;''']
```

Expanding:

```
Some squigles...[*[alpha][beta]]
```

We have:

Some squiggles...αβ

The difference with macros are

- The explicit template expansion syntax. This is an advantage because, now, we don't have to use obscure naming conventions like double underscores (e.g. __alpha__) to avoid unwanted macro replacement.

- The template is expanded at the point where it is invoked. A macro is expanded immediately at its point of declaration. This is subtle and can cause a slight difference in behavior especially if you refer to other macros and templates in the body.

The empty brackets after the template identifier (alpha[]) indicates no arguments. If the template body does not look like a template argument list, we can elide the empty brackets. Example:

```
[template aristotle_quote Aristotle: [*['Education is the best provision
for the journey to old age.]]]
```

Expanding:

```
Here's a quote from [aristotle_quote].
```

We have:

Here's a quote from Aristotle: **Education is the best provision for the journey to old age.**.

The disadvantage is that you can't avoid the space between the template identifier, aristotle_quote, and the template body "Aristotle...". This space will be part of the template body. If that space is unwanted, use empty brackets or use the space escape: "\ ". Example:

```
[template tag\ _tag]
```

Then expanding:

```
`struct` x[tag];
```

We have:

struct x_tag;

You have a couple of ways to do it. I personally prefer the explicit empty brackets, though.

Simple Arguments

As mentioned, arguments are separated by the double dot "..". If there are less arguments passed than expected, QuickBook attempts to break the last argument into two or more arguments following this logic:

- Break the last argument into two, at the first space found ('', '\n', \t' or '\r').

- Repeat until there are enough arguments or if there are no more spaces found (in which case, an error is reported).

For example:

```
[template simple[a b c d] [a][b][c][d]]
[simple w x y z]
```

will produce:

wxyz

"w x y z" is initially treated as a single argument because we didn't supply any `".."` separators. However, since `simple` expects 4 arguments, "w x y z" is broken down iteratively (applying the logic above) until we have "w", "x", "y" and "z".

QuickBook only tries to get the arguments it needs. For example:

```
[simple w x y z trail]
```

will produce:

wxyz trail

The arguments being: "w", "x", "y" and "z trail".

It should be obvious now that for simple arguments with no spaces, we can get by without separating the arguments with `".."` separators. It is possible to combine `".."` separators with the argument passing simplification presented above. Example:

```
[simple what do you think ..m a n?]
```

will produce:

what do you think man?

Punctuation Templates

With templates, one of our objectives is to allow us to rewrite QuickBook in QuickBook (as a qbk library). For that to happen, we need to accommodate single character punctuation templates which are fairly common in QuickBook. You might have noticed that single character punctuations are allowed as template identifiers. Example:

```
[template ![bar] <hey>[bar]</hey>]
```

Now, expanding this:

```
[!baz]
```

We will have:

```
<hey>baz</hey>
```

Blurbs

```
[blurb :-) [*An eye catching advertisement or note...]

    __spirit__ is an object-oriented recursive-descent parser generator framework
    implemented using template meta-programming techniques. Expression templates
    allow us to approximate the syntax of Extended Backus-Normal Form (EBNF)
    completely in C++.
]
```

will generate this:

An eye catching advertisement or note...

Spirit is an object-oriented recursive-descent parser generator framework implemented using template meta-programming techniques. Expression templates allow us to approximate the syntax of Extended Backus-Normal Form (EBNF) completely in C++.

Note

Prefer admonitions wherever appropriate.

Tables

```
[table A Simple Table
    [[Heading 1] [Heading 2] [Heading 3]]
    [[R0-C0]      [R0-C1]      [R0-C2]]
    [[R1-C0]      [R1-C1]      [R1-C2]]
    [[R2-C0]      [R2-C1]      [R2-C2]]
]
```

will generate:

Table 4. A Simple Table

Heading 1	Heading 2	Heading 3
R0-C0	R0-C1	R0-C2
R2-C0	R2-C1	R2-C2
R3-C0	R3-C1	R3-C2

The table title is optional. The first row of the table is automatically treated as the table header; that is, it is wrapped in <thead>...</thead> XML tags. Note that unlike the original QuickDoc, the columns are nested in [cells...]. The syntax is free-format and allows big cells to be formatted nicely. Example:

```
[table Table with fat cells
    [[Heading 1] [Heading 2]]
    [
        [Row 0, Col 0: a small cell]
        [
            Row 0, Col 1: a big fat cell with paragraphs

            Boost provides free peer-reviewed portable C++ source libraries.

            We emphasize libraries that work well with the C++ Standard Library.
            Boost libraries are intended to be widely useful, and usable across
            a broad spectrum of applications. The Boost license encourages both
            commercial and non-commercial use.
        ]
    ]
    [
        [Row 1, Col 0: a small cell]
        [Row 1, Col 1: a small cell]
    ]
]
```

and thus:

Table 5. Table with fat cells

Heading 1	Heading 2
Row 0, Col 0: a small cell	Row 0, Col 1: a big fat cell with paragraphs Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.
Row 1, Col 0: a small cell	Row 1, Col 1: a small cell

Here's how to have preformatted blocks of code in a table cell:

```
[table Table with code
    [[Comment] [Code]]
    [
        [My first program]
        [``
            #include <iostream>

            int main()
            {
                std::cout << "Hello, World!" << std::endl;
                return 0;
            }
        ``]
    ]
]
```

Table 6. Table with code

Comment	Code
My first program	```#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}``` |

Variable Lists

```
[variablelist A Variable List
    [[term 1] [The definition of term 1]]
    [[term 2] [The definition of term 2]]
    [[term 3] [
    The definition of term 3.

    Definitions may contain paragraphs.
    ]]
]
```

will generate:

A Variable List

term 1   The definition of term 1

term 2   The definition of term 2

term 3   The definition of term 3.

              Definitions may contain paragraphs.

The rules for variable lists are the same as for tables, except that only 2 "columns" are allowed. The first column contains the terms, and the second column contains the definitions. Those familiar with HTML will recognize this as a "definition list".

Include

You can include one QuickBook file from another. The syntax is simply:

```
[include someother.qbk]
```

The included file will be processed as if it had been cut and pasted into the current document, with the following exceptions:

- The __FILENAME__ predefined macro will reflect the name of the file currently being processed.

- Any macros defined in the included file are scoped to that file.

The [include] directive lets you specify a document id to use for the included file. When this id is not explicitly specified, the id defaults to the filename ("someother", in the example above). You can specify the id like this:

```
[include:someid someother.qbk]
```

All auto-generated anchors will use the document id as a unique prefix. So for instance, if there is a top section in someother.qbk named "Intro", the named anchor for that section will be "someid.intro", and you can link to it with [link someid.intro The Intro].

Import

When documenting code, you'd surely need to present code from actual source files. While it is possible to copy some code and paste them in your QuickBook file, doing so is error prone and the extracted code in the documentation tends to get out of sync with the actual code as the code evolves. The problem, as always, is that once documentation is written, the tendency is for the docs to languish in the archives without maintenance.

QuickBook's import facility provides a nice solution.

Example

You can effortlessly import code snippets from source code into your QuickBook. The following illustrates how this is done:

```
[import ../test/stub.cpp]
[foo]
[bar]
```

The first line:

```
[import ../test/stub.cpp]
```

collects specially marked-up code snippets from stub.cpp and places them in your QuickBook file as virtual templates. Each of the specially marked-up code snippets has a name (e.g. foo and bar in the example above). This shall be the template identifier for that particular code snippet. The second and third line above does the actual template expansion:

```
[foo]
[bar]
```

And the result is:

This is the **foo** function.

This description can have paragraphs...

- lists

26

- etc.

And any quickbook block markup.

```
std::string foo()
{
    // return 'em, foo man!
    return "foo";
}
```

This is the **bar** function

```
std::string bar()
{
    // return 'em, bar man!
    return "bar";
}
```

Some trailing text here

Code Snippet Markup

Note how the code snippets in stub.cpp get marked up. We use distinguishable comments following the form:

```
//[id
some code here
//]
```

The first comment line above initiates a named code-snippet. This prefix will not be visible in quickbook. The entire code-snippet in between `//[id` and `//]` will be inserted as a template in quickbook with name id. The comment `//]` ends a code-snippet This too will not be visible in quickbook.

Special Comments

Special comments of the form:

```
//` some [*quickbook] markup here
```

and:

```
/*` some [*quickbook] markup here */
```

will be parsed by QuickBook. This can contain quickbook *blocks* (e.g. sections, paragraphs, tables, etc). In the first case, the initial slash-slash, tick and white-space shall be ignored. In the second, the initial slash-star-tick and the final star-slash shall be ignored.

Special comments of the form:

```
/*<- this C++ comment will be ignored ->*/
```

or

```
/*<-*/ "this c++ code  will be ignored" /*->*/
```

or

```
//<-
private:
    int some_member;
//->
```

can be used to inhibit code from passing through to quickbook. All text between the delimeters will simply be ignored.

Callouts

Special comments of the form:

```
/*< some [*quickbook] markup here >*/
```

will be regarded as callouts. These will be collected, numbered and rendered as a "callout bug" (a small icon with a number). After the whole snippet is parsed, the callout list is generated. See Callouts for details. Example:

```
std::string foo_bar() ❶
{
    return "foo-bar"; ❷
}
```

❶   The *Mythical* FooBar. See Foobar for details
❷   return 'em, foo-bar man!

This is the actual code:

```
//[ foo_bar
std::string foo_bar() /*< The /Mythical/ FooBar.
                         See [@http://en.wikipedia.org/wiki/Foobar Foobar for details] >*/
{
    return "foo-bar"; /*< return 'em, foo-bar man! >*/
}
//]
```

The callouts bugs are placed exactly where the special callout comment is situated. It can be anywhere in the code. The bugs can be rather obtrusive, however. They get in the way of the clarity of the code. Another special callout comment style is available:

```
/*<< some [*quickbook] markup here >>*/
```

This is the line-oriented version of the callout. With this, the "bug" is placed at the very left of the code block, away from the actual code. By placing it at the far left, the code is rendered un-obscured. Example:

```
class x
{
public:

    ❶x() : n(0)
    {
    }

    ❷~x()
    {
    }

    ❸int get() const
    {
        return n;
    }

    ❹void set(int n_)
    {
        n = n_;
    }
};
```

❶   Constructor
❷   Destructor
❸   Get the `n` member variable
❹   Set the `n` member variable

See the actual code here: ../../tools/quickbook/test/stub.cpp

Installation and configuration

This section provides some guidelines on how to install and configure BoostBook and Quickbook under several operating systems.

Before continuing, it is very important that you keep this in mind: if you try to build some documents and the process breaks due to misconfiguration, be absolutely sure to delete any `bin` and `bin.v2` directories generated by the build before trying again. Otherwise your configuration fixes will not take any effect.

Mac OS X

Section contributed by Julio M. Merino Vidal

The following instructions explain how to install Docbook XML, Docbook XSL and Doxygen in a Mac OS X system, how to configure Boost.Build v2 to recognize them and how to build and install Quickbook. They were taken from a 10.4 (Tiger) machine so it is likely that they also apply to future versions; they may not work with older ones, though.

The text below assumes you want to install all the necessary utilities in a system-wide location, allowing any user in the machine to have access to them. Therefore, all files will be put in the `/usr/local` hierarchy. If you do not want this, you can choose any other prefix such as `~/Applications` for a single-user installation.

Mac OS X comes with `xsltproc` and all related libraries preinstalled, so you do not need to take any extra steps to set them up. It is probable that future versions will include them too, but these instructions may not apply to older versions.

To get started:

1. Download Docbook XML 4.2 and unpack it inside `/usr/local/share/xml/docbook/4.2`.

2. Download the latest Docbook XSL version and unpack it. Put the results in `/usr/local/share/xsl/docbook`, thus effectively removing the version number from the directory name (for simplicity).

3. Add the following to your `user-config.jam` file, which should live in your home directory (`/Users/<your_username>`). You must already have it somewhere or otherwise you could not be building Boost (i.e. missing tools configuration).

```
using xsltproc ;

using boostbook
    : "/usr/local/share/xsl/docbook"
    : "/usr/local/share/xml/docbook/4.2"
    ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds and install a system-wide Quickbook instead:

1. Go to Quickbook's source directory (`BOOST_ROOT/tools/quickbook`).

2. Build the utility by issuing `bjam --v2`.

3. Copy the resulting `quickbook` binary (located under the `BOOST_ROOT/bin.v2` hierarchy) to a safe place. Following our previous example, you can install it into: `/usr/local/bin`.

4. Add the following to your `user-config.jam` file:

```
using quickbook
    : "/usr/local/bin/quickbook" ;
    ;
```

Additionally, if you need to build documentation that uses Doxygen, you will need to install it too:

1. Go to the downloads section and get the disk image (`dmg` file) for Mac OS X.

2. Open the disk image and drag the Doxygen application to your `Applications` folder to install it.

3. Add the following to your `user-config.jam` file:

```
using doxygen
    : /Applications/Doxygen.app/Contents/Resources/doxygen
    ;
```

Alternatively, you may want to install all the prerequistes through a package system to avoid manual management of the installations. In that case, check out pkgsrc.

Windows 2000, XP, 2003, Vista

Section contributed by Julio M. Merino Vidal

The following instructions apply to any Windows system based on Windows 2000, including Windows XP, Windows 2003 Server and Windows Vista. The paths shown below are taken from a Windows Vista machine; you will need to adjust them to match your system in case you are running an older version.

1. First of all you need to have a copy of `xsltproc` for Windows. There are many ways to get this tool, but to keep things simple, use the binary packages made by Igor Zlatkovic. At the very least, you need to download the following packages: `iconv`, `zlib`, `libxml2` and `libxslt`.

2. Unpack all these packages in the same directory so that you get unique `bin`, `include` and `lib` directories within the hierarchy. These instructions use `C:\Users\example\Documents\boost\xml` as the root for all files.

3. From the command line, go to the `bin` directory and launch `xsltproc.exe` to ensure it works. You should get usage information on screen.

4. Download Docbook XML 4.2 and unpack it in the same directory used above. That is: `C:\Users\example\Documents\boost\xml\docbook-xml`.

5. Download the latest Docbook XSL version and unpack it, again in the same directory used before. To make things easier, rename the directory created during the extraction to `docbook-xsl` (bypassing the version name): `C:\Users\example\Documents\boost\xml\docbook-xsl`.

6. Add the following to your `user-config.jam` file, which should live in your home directory (`%HOMEDRIVE%%HOMEPATH%`). You must already have it somewhere or otherwise you could not be building Boost (i.e. missing tools configuration).

```
using xsltproc
    : "C:/Users/example/Documents/boost/xml/bin/xsltproc.exe"
    ;

using boostbook
    : "C:/Users/example/Documents/boost/xml/docbook-xsl"
    : "C:/Users/example/Documents/boost/xml/docbook-xml"
    ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds:

1. Go to Quickbook's source directory (`BOOST_ROOT\tools\quickbook`).

2. Build the utility by issuing `bjam --v2`.

3. Copy the resulting `quickbook.exe` binary (located under the `BOOST_ROOT\bin.v2` hierarchy) to a safe place. Following our previous example, you can install it into: `C:\Users\example\Documents\boost\xml\bin`.

4. Add the following to your `user-config.jam` file:

```
using quickbook
    : "C:/Users/example/Documents/boost/xml/bin/quickbook.exe"
    ;
```

Debian, Ubuntu

The following instructions apply to Debian and its derivatives. They are based on a Ubuntu Edgy install but should work on other Debian based systems.

First install the `bjam`, `xsltproc`, `docbook-xsl` and `docbook-xml` packages. For example, using `apt-get`:

```
sudo apt-get install xsltproc docbook-xsl docbook-xml
```

If you're planning on building boost's documentation, you'll also need to install the `doxygen` package as well.

Next, we need to configure Boost Build to compile BoostBook files. Add the following to your `user-config.jam` file, which should be in your home directory. If you don't have one, create a file containing this text. For more information on setting up `user-config.jam`, see the Boost Build documentation.

```
using xsltproc ;

using boostbook
    : /usr/share/xml/docbook/stylesheet/nwalsh
    : /usr/share/xml/docbook/schema/dtd/4.2
    ;

# Remove this line if you're not using doxygen
using doxygen ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds:

1. Go to Quickbook's source directory (`BOOST_ROOT/tools/quickbook`).

2. Build the utility by issuing `bjam --v2`.

3. Copy the resulting `quickbook` binary (located under the `BOOST_ROOT/bin.v2` hierarchy) to a safe place. The traditional location is `/usr/local/bin`.

4. Add the following to your `user-config.jam` file, using the full path of the quickbook executable:

```
using quickbook
    : /usr/local/bin/quickbook
    ;
```

Editor Support

Editing quickbook files is usually done with text editors both simple and powerful. The following sections list the settings for some editors which can help make editing quickbook files a bit easier.

☞ You may submit your settings, tips, and suggestions to the authors, or through the docs Boost Docs mailing list.

Scintilla Text Editor

Section contributed by Dean Michael Berris

The Scintilla Text Editor (SciTE) is a free source code editor for Win32 and X. It uses the SCIntilla source code editing component.

ⓘ SciTE can be downloaded from http://www.scintilla.org/SciTE.html

You can use the following settings to highlight quickbook tags when editing quickbook files.

```
qbk=*.qbk
lexer.*.qbk=props
use.tabs.$(qbk)=0
tab.size.$(qbk)=4
indent.size.$(qbk)=4
style.props.32=$(font.base)
comment.stream.start.props=[/
comment.stream.end.props=]
comment.box.start.props=[/
comment.box.middle.props=
comment.box.end.props=]
```

Thanks to Rene Rivera for the above SciTE settings.

KDE Support

boost::hs::quickbook

boost::hs::quickbook is a syntax highlighting designed to work with Katepart. It can be used in KWrite, Kate, Konqueror and KDevelop, and supports all the constructs of Quickbook 1.4 including tables, list, templates and macros.

.qbk loaded in a text editor

```
[table Code examples
[[   Name      ][                   Code                ][        Description
[[for loop      ][ ` for(int k=0; k<10; k++) v+=k; `       ][Sums some numbers.
[[while loop    ][ ` { int k; while( k < 10 ) { v+=k; k++; } } ` ][Same effect.
[[infinite loop ][ ` while( true ) { v+=1; } `             ][Not a good example.
]
```

.qbk loaded with boost::hs support

```
[table Code examples
[[   Name      ][                   Code                ][        Description
[[for loop      ][ ` for(int k=0; k<10; k++) v+=k; `       ][Sums some numbers.
[[while loop    ][ ` { int k; while( k < 10 ) { v+=k; k++; } } ` ][Same effect.
[[infinite loop ][ ` while( true ) { v+=1; } `             ][Not a good example.
]
```

html generated from this .qbk file

Table 7. Code examples

Name	Code	Description
for loop	`for(int k=0; k<10; k++) v+=k;`	Sums some numbers.
while loop	`{ int k; while( k < 10 ) { v+=k; k++ } }`	Same effect.
infinite loop	`while( true ) { v+=1; }`	Not a good example.

Code Folding

boost::hs goes far beyond simple coloring. One useful thing you can get the editor to do is to mark regions. They appear in a small grey line and each region can be folded or unfolded independently.

Auto Comment / Uncomment

Another important feature is the possibility to auto-comment or uncomment some piece of code (*Tools - Comment*). Commented regions can be uncommented simple calling the *uncomment* command while being in it.

Styles reference

Name	Style	Description
plain text	normal black	Plain text at each level.
formatted text	formatted black	Bold, italic, underline and mixes. Tele-type, replaceable, strikeout.
structure	light blue	All quickbook structures characters ([,], [block-type, simple formating boundaries, lists keywords (*, #)
macros	red	Names in macro definitions, macros insertion if it is used the __xxx__ proposed sintaxis.
templates	red	Names in template definitions
anchors	red	All the keywords that are used to link quickbooks together.
comments	italic light gray	Inside the commentaries.
tables	HTML like	Reveal the structure, bold title, higlighted HTML like columns titles.
variable lists	HTML like	Reveal the structure, bold title, bold HTML like items names.
c++ code	cpp Kate syntax	Code blocks and inline code.
paths	green	Image, files and web paths
IDE specific	dark blue	IDE commands

About boost::hs

boost::hs::quickbook is a component of boost::hs, a syntax highlighting for C++, doxygen, Boost.Build jamfiles and QuickBook. boost::hs has his own page here.

> **Note**
>
> boost::hs::cpp support QuickBook code import comments style!

Installing boost::hs

There exist an ongoing effort to push boost::hs upstream to the KatePart project. In a few months KDE may have native Quickbook support! For the moment you must download and install it.

You can download boost::hs from here.

> **Note**
>
> A copy of boost::hs::quickbook and boost::hs::cpp is available in `boost/tools/quickbook/extra/katepart`.

In order to install it you must copy the content in the folder **katepart/syntax/** to the appropriate katepart syntax folder in your machine. In general this folder will be in **/usr/share/apps/katepart/syntax**. A bash script named *install.sh* is included that copy the files to this folder.

Frequently Asked Questions

Can I use QuickBook for non-Boost documentation?

QuickBook can be used for non-Boost documentation with a little extra work.

> *Faq contributed by Michael Marcin*

When building HTML documentation with BoostBook a Boost C++ Libraries header is added to the files. When using QuickBook to document projects outside of Boost this is not desirable. This behavior can be overridden at the BoostBook level by specifying some XSLT options. When using Boost Build version 2 (BBv2) this can be achieved by adding parameters to the BoostBook target declaration.

For example:

```
using quickbook ;

xml my_doc : my_doc.qbk ;

boostbook standalone
    :
        my_doc
    :
        <xsl:param>boost.image.srcimages/my_project_logo.png
        <xsl:param>boost.image.alt"\"My Project\""
        <xsl:param>boost.image.w=100
        <xsl:param>boost.image.h=50
        <xsl:param>nav.layout=none
    ;
```

Is there an easy way to convert BoostBook docs to QuickBook?

There's a stylesheet that allows Boostbook generated HTML to be viewed as quickbook source, see http://svn.boost.org/trac/boost/wiki/QuickbookSourceStylesheetProject, so it's then just a cut and paste job to convert the BoostBook to QuickBook (which IMO is a whole lot easier to edit and maintain).

--John Maddock

Quick Reference

Table 8. Syntax Compendium

To do this...	Use this...	See this...
comment	`[/ some comment]`	Comments
italics	`['italics]` or `/italics/`	Font Styles and Simple formatting
bold	`[*bold]` or `*bold*`	Font Styles and Simple formatting
<u>underline</u>	`[_underline]` or `_underline_`	Font Styles and Simple formatting
`teletype`	`[^teletype]` or `=teletype=`	Font Styles and Simple formatting
~~strikethrough~~	`[-strikethrough]`	Font Styles and Simple formatting
replaceable	`[~replaceable]`	Replaceble
source mode	`[c++]` or `[python]`	Source Mode
inline code	`` `int main();` ``	Inline code
code block	` ``int main();`` `	Code
code escape	` ``from c++ to QuickBook`` `	Escaping Back To QuickBook
line break	`[br]` or `\n`	line-break **DEPRECATED**
anchor	`[#anchor]`	Anchors
link	`[@http://www.boost.org Boost]`	Links
anchor link	`[link section.anchor Link text]`	Anchor links
refentry link	`[link xml.refentry Link text]`	refentry links
function link	`[funcref fully::qualified::func-tion_name Link text]`	function, class, member, enum, macro, concept or header links
class link	`[classref fully::quali-fied::class_name Link text]`	function, class, member, enum, macro, concept or header links
member link	`[memberref fully::quali-fied::member_name Link text]`	function, class, member, enum, macro, concept or header links
enum link	`[enumref fully::quali-fied::enum_name Link text]`	function, class, member, enum, macro, concept or header links
macro link	`[macroref MACRO_NAME Link text]`	function, class, member, enum, macro, concept or header links
concept link	`[conceptref ConceptName Link text]`	function, class, member, enum, macro, concept or header links
header link	`[headerref path/to/header.hpp Link text]`	function, class, member, enum, macro, concept or header links

To do this...	Use this...	See this...
global link	`[globalref fully::quali-fied::global Link text]`	function, class, member, enum, macro, concept or header links
escape	`'''escaped text (no pro-cessing/formatting)'''`	Escape
single char escape	`\c`	Single char escape
images	`[$image.jpg]`	Images
begin section	`[section The Section Title]`	Section
end section	`[endsect]`	Section
paragraph	No markup. Paragraphs start left-flushed and are terminated by two or more newlines.	Paragraphs

ordered list	`# one`
	`# two`
	`# three`
		Ordered lists

unordered list	`* one`
	`* two`
	`* three`
		Unordered lists

code	No markup. Preformatted code starts with a space or a tab.	Code
preformatted	`[pre preformatted]`	Preformatted
block quote	`[:sometext...]`	Blockquote
heading 1	`[h1 Heading 1]`	Heading
heading 2	`[h2 Heading 2]`	Heading
heading 3	`[h3 Heading 3]`	Heading
heading 4	`[h4 Heading 4]`	Heading
heading 5	`[h5 Heading 5]`	Heading
heading 6	`[h6 Heading 6]`	Heading
macro	`[def macro_identifier some text]`	Macros
template	`[template[a b] [a] body [b]]`	Templates
blurb	`[blurb advertisement or note...]`	Blurbs

To do this...	Use this...	See this...
admonition	`[warning Warning text...]`	Admonitions

table	`[table Title`
	`[[a][b][c]]`
	`[[a][b][c]]`
	`]`
		Tables

variablelist	`[variablelist Title`
	`[[a][b]]`
	`[[a][b]]`
	`]`
		Variable Lists

include	`[include someother.qbk]`	Include
conditional generation	`[? symbol phrase]`	Conditional Generation