`MPark/WG21` User's Guide

Framework for Writing C++ Committee Proposals

Document #:	D0000R0
Date:	2026-06-26
Project:	Programming Language C++
Audience:	WG21

Reply-to:	Michael Park <mcypark@gmail.com>

Contents

1 Introduction

`MPark/WG21` is a framework for writing proposals for the C++ Standards Committee, built on top of Pandoc.

In short, you write **Markdown** and the framework produces the paper either in **HTML** or **PDF** (via LaTeX).

The framework provides Markdown extensions that are specifically useful for C++ proposals.

Couple of examples are:

— Modifying Wording Text
— Embedded Markdown in Code

2 Goals

There are two goals for this framework, not necessarily in order:

— Ease of **authoring** papers
— Ease of **reviewing** papers

3 Getting Started

3.1 Requirements

The framework downloads and maintains a pinned version of Pandoc.

For HTML output, only the following dependencies are required:

— `git`
— `curl`
— `make`
— `python3`
— `python3-venv`

For PDF output, the framework also depends on `xelatex`.

3.1.1 macOS

```
brew install python make

# For PDF output
brew install --cask mactex
```

3.1.2 Ubuntu

```
sudo apt-get install git curl make python3 python3-venv

# For PDF output
sudo apt-get install texlive-xetex
```

3.1.3 Debian

```
sudo apt-get install git curl make python3 python3-venv

# For PDF output
sudo apt-get install texlive-xetex \
                     texlive-fonts-recommended \
                     texlive-latex-recommended \
                     texlive-latex-extra
```

3.2 Integration

Add this repository to your paper repository as a git submodule:

```
git submodule add https://github.com/mpark/wg21.git
```

3.3 Project Layouts

The framework provides two Makefile fragments for common project layouts.

3.3.1 Flat Project Layout

Use `flat.mk` when all papers live in one directory and outputs should be written to a common output directory.

```
wg21-papers/
|-- wg21 (submodule)
|-- Makefile
|-- p2806r4.md
|-- p2996r13.md
`-- generated/
    |-- p2806r4.html
    `-- p2996r13.html
```

In the top-level `Makefile`:

```
include wg21/flat.mk
```

Markdown files in the repository root become build targets. By default, outputs are written under `generated/`.

For example:

```
make p2806r4.html  # builds generated/p2806r4.html
make p2806r4.pdf   # builds generated/p2806r4.pdf
```

You may also build all papers at once:

```
make       # builds all papers in all formats
make html  # builds all papers in HTML format
make latex # builds all papers in LaTeX format
make pdf   # builds all papers in PDF format
```

To use a different output directory, set `OUTDIR` before the include:

```
OUTDIR := out
include wg21/flat.mk
```

If a top-level `defaults.yaml` or `requirements.txt` exists, it is picked up automatically. To use a different file, set `DEFAULTS` or `REQUIREMENTS` before the include.

See mpark/wg21-papers for an example use of this layout.

3.3.2 Per-paper Project Layout

Use `paper.mk` when each paper has its own directory. Outputs are written in that paper directory.

```
wg21-papers/
|-- wg21 (submodule)
|-- p2806/
|   |-- Makefile
|   |-- p2806r4.md
|   `-- p2806r4.html
`-- p2996/
    |-- Makefile
    |-- p2996r13.md
    `-- p2996r13.html
```

In each per-paper `Makefile`:

```
include ../wg21/paper.mk
```

Same-stem targets work automatically. For example:

```
cd p2806
make p2806r4.html  # builds from p2806r4.md
```

You may also introduce explicit source-to-output mappings. Suppose you have:

```
wg21-papers/
|-- wg21 (submodule)
`-- p2806-do-expr/
    |-- Makefile
    |-- do-expr.md
    `-- p2806r4.html
```

In `p2806-do-expr/Makefile`:

```
include ../wg21/paper.mk

p2806r4.html: do-expr.md
```

With this, you can do:

```
cd p2806-do-expr
make p2806r4.html  # builds from do-expr.md

# or just...
make               # also builds p2806r4.html from do-expr.md
```

See brevzin/cpp_proposals for an example use of this layout.

4 Formatting

This framework provides support for various common elements for C++ proposals.

4.1 Title

The title is specified in a YAML metadata block.

For example, the title of this document is generated from:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
---
```

[*Note:* `date: today` generates today's date in `YYYY-MM-DD` (ISO 8601) format. — *end note*]

YAML lists can be used to specify multiple audiences and authors:

```
  ---
  title: "`MPark/WG21` User's Guide"
  subtitle: "Framework for Writing C++ Committee Proposals"
  document: D0000R0
  date: today
- audience: WG21
+ audience:
+   - Library Evolution Working Group
+   - Library Working Group
  author:
    - name: Michael Park
      email: <mcypark@gmail.com>
+   - name: Barry Revzin
+     email: <barry.revzin@gmail.com>
---
```

4.2 Table of Contents

By default, a table of contents is generated. For an example, see the table of contents of this document.

4.2.1 Metadata: `toc`

To disable the table of contents entirely, set the `toc` metadata to `false`.

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
toc: false
---
```

4.2.2 Metadata: `toc-depth`

The default depth of table of contents is 3. That is, given headers like

```
# Design Overview

## Types of Patterns

### Primary Patterns

#### Wildcard Pattern
```

The table of contents will not show `#### Wildcard Pattern` since it's 4 levels deep.

To set the depth manually, set `toc-depth` to the desired number.
For example, to make `#### Wildcard Pattern` show up:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
toc-depth: 4
---
```

4.3 Headings

Both Setext and ATX styles are available:

Markdown Source	Rendered Output
`Header 1` `========`	Header 1
`Header 2` `--------`	Header 2
`### Header 3`	Header 3
`#### Header 4`	Header 4

4.3.1 Disable from Section Numbering: `-` or `.unnumbered`

Add the `-` or `.unnumbered` class to a header to exclude it from section numbering, using the Pandoc extension `header_attributes`:

```
# Miscelleneous {-}
```

4.3.2 Exclude from Table of Contents: `.unlisted`

Add the `.unlisted` class **in addition to** `-` or `.unnumbered` to a header to exclude it from the table of contents, using the Pandoc extension: `header_attributes`:

```
# Miscelleneous {- .unlisted}
```

4.3.3 Implicit Header References

Markdown Source	Rendered Output
`Given a heading like:` `## Algorithm Return Type {- .unlisted}` `Reference with any of the following ways:` `- [Algorithm Return Type]` `- [Algorithm Return Type][]` `- [Custom Text 1][Algorithm Return Type]` `- [Custom Text 2](#algorithm-return-type)`	Given a heading like: **Algorithm Return Type** Reference with any of the following ways: — Algorithm Return Type — Algorithm Return Type — Custom Text 1 — Custom Text 2

The `#algorithm-return-type` identifier is automatically generated by the Pandoc extension: `auto_identifiers`.

Relevant Pandoc extensions: `implicit_header_references`, `shortcut_reference_links`

4.3.4 Header References by ID with Automatic Text

This extension interprets `[](#header-identifier)` as a reference to a header where the heading text is automatically used.

The main usage is to first specify an explicit identifier on a header, like:

```
## Algorithm Return Type {#return-type}
```

then refer to it like `[](#return-type)`, which will be rendered as: Algorithm Return Type.

The main advantage is that when the header text changes, the reference remains stable and the new heading text is automatically rendered.

4.4 Inline Formatting

4.4.1 Emphasis

Use asterisks (`*`) and underscores (`_`) to emphasize inline text.

Markdown Source	Rendered Output
`Some of these words *are emphasized*.`	Some of these words *are emphasized*.
`Some of these words _are emphasized also_.`	Some of these words *are emphasized also*.
`Use two asterisks for **strong emphasis**.`	Use two asterisks for **strong emphasis**.
`Or, __use two underscores instead__.`	Or, **use two underscores instead**.

For emphasizing **part** of a word, asterisks are **required**.

Markdown Source	Rendered Output
`feas**ible**, not feas__able__`	feas**ible**, not feas__able__

This is the Pandoc extension: `intraword_underscores`.

4.4.2 Superscript, Subscript

Wrap caret (`^`) for superscripts and tilde (`~`) for subscripts.

Markdown Source	Rendered Output
`2^10^ is 1024`	2^{10} is 1024

Markdown Source	Rendered Output
`` `constexpr`~opt~ means optional ``	`constexpr`$_{\text{opt}}$ means optional

This is the Pandoc extension: `superscript_subscript`.

4.4.3 Highlighting

Wrap `==`, or use `[highlighted text]{.mark}` to highlight some text.

Markdown Source	Rendered Output
`This is a ==highlighted **text**==.`	This is a highlighted **text**.
`Also, [highlight *text*]{.mark}`	Also, highlight *text*

4.5 Wording

4.5.1 Modifying Text

Small, inline changes are bracketed `Span` elements that look like the following:

Markdown Source	Rendered Output
`Let's add some [new **text**]{.add}.`	Let's add some new **text**.
`Remove some [old *text*]{.rm} now.`	Remove some ~~old *text*~~ now.
`Substitute: [old *text*](new **text**){.sub}`	Substitute: ~~old *text*~~new **text**

[*Note:* Substitutions are essentially just shorthand for `[old text]{.rm}[new text]{.add}`. — *end note*]

Markdown Source

```
The optional *attribute-specifier-seq* appertains to the [label](*general-label*){.sub}.
The only use of a [label with an *identifier*](*label*){.sub} is as the target of
a `goto`, [`break`, or `continue`]{.add}. No two [label](*label*){.sub}s in a function
shall have the same *identifier*. A [label](*general-label*){.sub} can be used
[in a `goto` statement]{.rm} before its introduction by a *labeled-statement*.
```

Rendered Output

The optional *attribute-specifier-seq* appertains to the ~~label~~*general-label*. The only use of a ~~label with an *identifier*~~*label* is as the target of a `goto`, `break`, or `continue`. No two ~~label~~*label*s in a function shall have the same *identifier*. A ~~label~~*general-label* can be used ~~in a `goto` statement~~ before its introduction by a *labeled-statement*.

Large changes are fenced `Div` blocks with `::: add` for additions, `::: rm` for removals, and close with `:::`.

Markdown Source

````
Modify section [format.functions]{.sref}

::: add
> ```
> template<class... Args>
>   string format(const locale& loc, string_view fmt, const Args&... args);
> ```
>
> *Returns*: `vformat(loc, fmt, make_format_args(args...))`.
:::
````

Rendered Output

Modify section 28.5.5 Formatting functions [format.functions]

```
template<class... Args>
  string format(const locale& loc, string_view fmt, const Args&... args);
```

Returns: `vformat(loc, fmt, make_format_args(args...))`.

Markdown Source

```
Remove [expr.post.incr]{.sref}/2:

::: rm
> The operand of postfix `--` is decremented analogously to the postfix `++` operator.
>
> [For prefix increment and decrement, see [expr.pre.incr].]{.note}
:::
```

Rendered Output

Remove 7.6.1.6 Increment and decrement [expr.post.incr]/2:

> The operand of postfix `--` is decremented analogously to the postfix `++` operator.
>
> [*Note:* For prefix increment and decrement, see [expr.pre.incr]. — *end note*]

4.5.2 Paragraph Numbers

4.5.2.1 Paragraph Number Element: `.pnum`

Paragraph number elements are bracketed `Span` elements that look like: `[2]{.pnum}` and `[2.1]{.pnum}`.

Markdown Source

```
[1]{.pnum} In this subclause, "before" and "after" refer to the "happens before"
relation ([intro.multithread]).

[2]{.pnum} The *lifetime* of an object or reference is a runtime property of
the object or reference. [...]

  - [2.1]{.pnum} storage with the proper alignment and size for type `T` is obtained, and

    [...]

  - [2.4]{.pnum} if `T` is a class type, the destructor call starts, or
  - [2.5]{.pnum} the storage which the object occupies is released, or is reused by
    an object that is not nested within _o_ ([intro.object]).

[...]

::: add
[x]{.pnum} Some new paragraph here
:::

[6]{.pnum} A program may end the lifetime of an object of class type without
invoking the destructor, by reusing or releasing the storage as described above.
```

Rendered Output

1 In this subclause, “before” and “after” refer to the “happens before” relation ([intro.multithread]).

2 The *lifetime* of an object or reference is a runtime property of the object or reference. [...]

(2.1) — storage with the proper alignment and size for type `T` is obtained, and

[...]

(2.4) — if `T` is a class type, the destructor call starts, or

(2.5) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* ([intro.object]).

[...]

x Some new paragraph here

6 A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

Within `::: wording` blocks, use `#` within a paragraph number element to automatically fill in that part. For example:

Markdown Source

```
::: wording
[#]{.pnum} Automatically starts at 1.

[5]{.pnum} Existing paragraph pinned at 5.

[#]{.pnum} Automatically continues to 6.

- [#.#]{.pnum} Automatically starts a nested numbering at (6.1).

::: add
- [#.#]{.pnum} Automatically continues a nested numbering at (6.2).

[x]{.pnum} Added paragraph that does not affect the next automatic number.
:::

[#]{.pnum} Automatically continues to 7.
:::
```

Rendered Output

1 Automatically starts at 1.

5 Existing paragraph pinned at 5.

6 Automatically continues to 6.

(6.1) — Automatically starts a nested numbering at (6.1).

(6.2) — Automatically continues a nested numbering at (6.2).

x Added paragraph that does not affect the next automatic number.

7 Automatically continues to 7.

The automatic numbering resets for each `::: wording` div.

4.5.2.2 List-based Paragraphs (Experimental)

List-based paragraphs are list elements (`ordered_lists` and `bullet_lists`), within a bracketed `Div` element, `::: wording`.

At the top-level, use `#.` for automatic numbering, and `1.` for pinned numbering. For example:

Markdown Source

```
::: wording
#. In this subclause, "before" and "after" refer to the "happens before" relation
   ([intro.multithread]).
#. The *lifetime* of an object or reference is a runtime property of the object
   or reference. [...]

[...]
```

```
6. A program may end the lifetime of an object of class type without invoking
   the destructor, by reusing or releasing the storage as described above.
:::
```

Rendered Output

1 In this subclause, “before” and “after” refer to the “happens before” relation ([intro.multithread]).

2 The *lifetime* of an object or reference is a runtime property of the object or reference. [...]

[...]

6 A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

Use nested bullet lists for sublists which are also automatically numbered, and `1.` for partial pinning. For example, if we pin `3.` within paragraph `2`, partial pinning will work out to `(2.3)`.

Markdown Source

```
::: wording
#. In this subclause, "before" and "after" refer to the "happens before" relation
   ([intro.multithread]).
#. The *lifetime* of an object or reference is a runtime property of the object
   or reference. [...]

   - storage with the proper alignment and size for type `T` is obtained, and

     [...]

   4. if `T` is a class type, the destructor call starts, or
   - the storage which the object occupies is released, or is reused by
     an object that is not nested within _o_ ([intro.object]).

[...]

6. A program may end the lifetime of an object of class type without invoking
   the destructor, by reusing or releasing the storage as described above.
:::
```

Rendered Output

1 In this subclause, “before” and “after” refer to the “happens before” relation ([intro.multithread]).

2 The *lifetime* of an object or reference is a runtime property of the object or reference. [...]

(2.1) — storage with the proper alignment and size for type `T` is obtained, and

[...]

(2.4) — if `T` is a class type, the destructor call starts, or

(2.5) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* ([intro.object]).

[...]

6 A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

Lastly, use `x.` to mark a paragraph number `x`. This is most useful within an `::: add` block where you want to insert new paragraphs without having to manually shift the numbers.

Markdown Source

```
::: wording
#. In this subclause, "before" and "after" refer to the "happens before" relation
   ([intro.multithread]).
```

```
#. The *lifetime* of an object or reference is a runtime property of the object
   or reference. [...]

   - storage with the proper alignment and size for type `T` is obtained, and

     [...]

   4. if `T` is a class type, the destructor call starts, or
   - the storage which the object occupies is released, or is reused by
     an object that is not nested within _o_ ([intro.object]).

[...]

::: add
x. Some new paragraph here
:::

6. A program may end the lifetime of an object of class type without invoking
   the destructor, by reusing or releasing the storage as described above.
:::
```

Rendered Output

1 In this subclause, “before” and “after” refer to the “happens before” relation ([intro.multithread]).

2 The *lifetime* of an object or reference is a runtime property of the object or reference. [...]

(2.1) — storage with the proper alignment and size for type `T` is obtained, and

[...]

(2.4) — if `T` is a class type, the destructor call starts, or

(2.5) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* ([intro.object]).

[...]

x Some new paragraph here

6 A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

[*Note:* In order to nest content (code blocks, nested list, etc) properly within another list, be sure to add a blank line and indent to line up with the first non-space character after the list marker. See Pandoc extension: `block_content_in_list_items`.

[*Example:*

Preceding blank line	Indent to line up
`#. Some paragraph` `~~~~~~~~~~~~~~~~~~~~` ` ```` `   code` ` ````	`#. Some paragraph` `   ~` ` ```` `   code` ` ````

— *end example*]

For nested lists, the blank line may be omitted:

[*Example:*

```
#. Some paragraph
   - Nested bullet list
```

— *end example*]

— *end note*]

4.5.3 Code Changes

Code changes can be either shown in a `diff` code block like so:

Markdown Source

````
```diff
  template <size_t I, class T1, class T2>
-   constexpr typename tuple_element<I, pair<T1, T2>>::type&
+   constexpr tuple_element_t<I, pair<T1, T2>>&
      get(pair<T1, T2>&) noexcept;
```
````

Rendered Output

```
  template <size_t I, class T1, class T2>
-   constexpr typename tuple_element<I, pair<T1, T2>>::type&
+   constexpr tuple_element_t<I, pair<T1, T2>>&
      get(pair<T1, T2>&) noexcept;
```

Alternatively, Embedded Markdown can be used to show the changes inline:

Markdown Source

````
```
template <size_t I, class T1, class T2>
  constexpr @[typename]{.rm}@ tuple_element@[_t]{.add}@<I, pair<T1, T2>>@[::type]{.rm}@&
    get(pair<T1, T2>&) noexcept;
```
````

Rendered Output

```
template <size_t I, class T1, class T2>
  constexpr typename tuple_element_t<I, pair<T1, T2>>::type&
    get(pair<T1, T2>&) noexcept;
```

[*Note:* For inline changes like this, prefer no-syntax-highlighting by using ``` to avoid too many colors. — *end note*]

4.5.4 Examples

Smaller, inline examples are bracketed Span elements that looks like `[example text]{.example}`.

Markdown Source

```
[`T x = T(T(T()));` value-initializes `x`.]{.example}
```

Rendered Output

[*Example:* `T x = T(T(T()));` value-initializes `x`. — *end example*]

Large examples are fenced Div blocks with `::: example`.

Markdown Source	Rendered Output
`` ::: example A simple example of a class definition is ```cpp struct tnode { char tword[20]; int count; tnode* left; tnode* right; }; ``` ::: ````	[*Example:* A simple example of a class definition is `struct tnode { char tword[20]; int count; tnode* left; tnode* right; };` — *end example*]

Within `::: wording` blocks, examples are numbered automatically. You may add the class `-` or `.unnumbered` to omit the number, or specify the `num` attribute like `num=5` to pin a number. The examples after a pinned number will increment from that number.

Markdown Source

````
::: wording
[`T x = T(T(T()));` value-initializes `x`.]{.example}

[`int a = do { 42 };`]{.example}

::: {.example num=5}
```cpp
auto a = do { do_return 1; };           // OK, deduces int
auto b = do -> long { do_return 1; };   // OK, explicit type is long
```
:::
:::
````

Rendered Output

[*Example 1:* `T x = T(T(T()));` value-initializes `x`. — *end example*]

[*Example 2:* `int a = do { 42 };` — *end example*]

[*Example 5:*

```cpp
auto a = do { do_return 1; };           // OK, deduces int
auto b = do -> long { do_return 1; };   // OK, explicit type is long
```

— *end example*]

4.5.5 Notes

Smaller, inline notes are bracketed `Span` elements that looks like `[note text]{.note}`.

Markdown Source

```
[Padding bits have unspecified value, but cannot cause traps.]{.note}
```

Rendered Output

[*Note:* Padding bits have unspecified value, but cannot cause traps. — *end note*]

Large notes are fenced `Div` blocks with `::: note`.

Markdown Source

````
::: note
An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`"
independently of the cv-qualifiers *cv1* and *cv2*.

```cpp
int a;
const int b = a;
int c = b;
```
:::
````

Rendered Output

[*Note:* An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*.

```cpp
int a;
const int b = a;
int c = b;
```

— *end note*]

Within `::: wording` blocks, notes are numbered automatically. You may add the class `-` or `.unnumbered` to omit the number, or specify the `num` attribute like `num=5` to pin a number. The notes after a pinned number will increment from that number.

Markdown Source

````
::: wording
[Padding bits have unspecified value, but cannot cause traps.]{.note}

::: note
An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`"
independently of the cv-qualifiers *cv1* and *cv2*.

```cpp
int a;
const int b = a;
int c = b;
```
:::
:::
````

Rendered Output

[*Note 1:* Padding bits have unspecified value, but cannot cause traps. — *end note*]

[*Note 2:* An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*.

```cpp
int a;
const int b = a;
int c = b;
```

— *end note*]

Use `ednote` for editorial notes:

Markdown Source	Rendered Output
`[This is a drive-by fix.]{.ednote}`	[Editor's note: This is a drive-by fix.]

Markdown Source

```
::: ednote
Throughout the wording, we say that a reflection (an object of type `std::meta::info`)
represents some source construct, while splicing that reflection designates that source
construct. For instance, `^^int` represents the type `int` and `[: ^^int :]` designates
the type `int`.
:::
```

Rendered Output

[Editor's note: Throughout the wording, we say that a reflection (an object of type `std::meta::info`) represents some source construct, while splicing that reflection designates that source construct. For instance, `^^int` represents the type `int` and `[: ^^int :]` designates the type `int`.]

Use `draftnote` for drafting notes:

Markdown Source

```
[An `audience` attribute addresses a specific audience]{.draftnote audience="the reader"}
```

Rendered Output

[Drafting note for the reader: An `audience` attribute addresses a specific audience]

Markdown Source

```
::: draftnote
We don't think we have to change anything here, since if `E` is a *splice-specifier*
that can be interpreted as a *splice-expression*, the requirements already fall out
based on how paragraphs 1 and 3 are already worded
:::
```

Rendered Output

[Drafting note: We don't think we have to change anything here, since if `E` is a *splice-specifier* that can be interpreted as a *splice-expression*, the requirements already fall out based on how paragraphs 1 and 3 are already worded]

To specify an audience for the fenced `Div` block, you'll need `::: {.draftnote audience="the reader"}`.

These editorial notes are not automatically numbered.

Finally, in the relatively common situation where an example appears within a note, you can simply nest them:

Markdown Source

````
::: note
The declaration of a class name takes effect immediately after the *identifier*
is seen in the class definition or *elaborated-type-specifier*.

::: example
```cpp
class A * A;
```
first specifies `A` to be the name of a class and then redefines it as the name of
a pointer to an object of that class. This means that the elaborated form `class A`
must be used to refer to the class. Such artistry with names can be confusing and
is best avoided.
:::
:::
````

Rendered Output

[*Note:* The declaration of a class name takes effect immediately after the *identifier* is seen in the class definition or *elaborated-type-specifier*.

[*Example:*

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided. — *end example*]

— *end note*]

4.5.6 Grammar

Use line blocks (`|`) in order to preserve the leading spaces.

Markdown Source

```
> | _selection-statement:_
> |     `if constexpr`_~opt~_ `(` _init-statement~opt~_ _condition_ `)` _statement_
> |     `if constexpr`_~opt~_ `(` _init-statement~opt~_ _condition_ `)` _statement_ `else` _stateme
> |     `switch (` _init-statement~opt~_ _condition_ `)` _statement_
> |     [`inspect` `constexpr`~_opt_~ `(` _init-statement~opt~_ _condition_ `)` `{`
>           _inspect-case-seq_
>        `}`]{.add}
>
> ::: add
> | _inspect-case-seq:_
> |     _inspect-case_
> |     _inspect-case-seq_ _inspect-case_
>
> | _inspect-case:_
> |     _attribute-specifier-seq~opt~_ _inspect-pattern_ _inspect-guard~opt~_ `:` _statement_
>
> | _inspect-pattern:_
> |     _wildcard-pattern_
> |     _identifier-pattern_
> |     _constant-pattern_
> |     _structured-binding-pattern_
> |     _alternative-pattern_
> |     _binding-pattern_
> |     _extractor-pattern_
>
> | _inspect-guard:_
> |     `if (` _expression_ `)`
> :::
```

Rendered Output

selection-statement:
- `if constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `)` *statement*
- `if constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `)` *statement* `else` *statement*
- `switch (` *init-statement*$_{opt}$ *condition* `)` *statement*
- `inspect constexpr`$_{opt}$ `(` *init-statement*$_{opt}$ *condition* `) {` *inspect-case-seq* `}`

inspect-case-seq:
- *inspect-case*
- *inspect-case-seq inspect-case*

inspect-case:
- *attribute-specifier-seq*$_{opt}$ *inspect-pattern inspect-guard*$_{opt}$ `:` *statement*

inspect-pattern:
- *wildcard-pattern*
- *identifier-pattern*
- *constant-pattern*

structured-binding-pattern
alternative-pattern
binding-pattern
extractor-pattern

inspect-guard:
`if` (*expression*)

4.6 Code

4.6.1 Inline Code

Use backticks like `` `int x = 0;` `` for inline code.

We can use the Pandoc extension `inline_code_attributes` to specify which language should be used for syntax highlighting:

Markdown Source	Rendered Output
`` `auto value = std::format("{}", bar);`{.cpp} ``	`auto value = std::format("{}", bar);`
`` `let value = format!("{bar}");`{.rust} ``	`let value = format!("{bar}");`
`` `value = f"{bar}"`{.python} ``	`value = f"{bar}"`

[*Note:* **Inline Code gets C++ syntax highlighting by default.** — *end note*]

Since we're writing a C++ proposal, many, if not most of the inline code will be C++. Adding the `{.cpp}` attribute everywhere can become very verbose, very fast. Thus, inline code is implicitly interpreted as `{.cpp}`. For example, `` `int x = 0;` `` is implicitly `` `int x = 0;`{.cpp} ``.

For no-syntax-highlighting, use `{.default}` like `` `int x = 0;`{.default} ``.

Markdown Source	Rendered Output
`` `constexpr int x = 0;` ``	`constexpr int x = 0;`
`` `constexpr int x = 0;`{.default} ``	`constexpr int x = 0;`

4.6.2 Code Block

Use three or more backticks to start a code block.

Markdown Source	Rendered Output
`` ``` int main() { return 0; } ``` ````	`int main() {` `  return 0;` `}`
`` ```cpp int main() { return 0; } ``` ````	`int main() {` `  return 0;` `}`

[*Note:* Unlike Inline Code, code blocks are not implicitly C++. — *end note*]

Add the `.numberLines` class to a code block to number the lines, and the `startFrom=N` attribute to specify the starting number, using the Pandoc extension: `fenced_code_attributes`.

Markdown Source	Rendered Output
``cpp {.numberLines} int main() { return 0; } ```	1 int main() { 2 return 0; 3 }
``cpp {.numberLines startFrom=8} int main() { return 0; } ```	8 int main() { 9 return 0; 10 }

4.6.3 Embedded Markdown

Code in C++ proposals often needs small pieces of wording markup inside it. With embedded Markdown enabled, text surrounded by `@` is parsed as Markdown and then placed back into the code element. This is useful for italicized terms, exposition-only names, modifying text, and stable names.

Embedded Markdown is **enabled by default** for `cpp`, `default`, and `diff` code classes. This is essentially:

— Inline code: `` `code` ``, `` `code`{.cpp} ``, `` `code`{.default} ``, `` `code`{.diff} `` and
— Code blocks that start with ```, ```cpp, ```default, or ```diff

Markdown Source

```
Recall the `static_cast` syntax: `static_cast < @*type-id*@ > ( @*expression*@ )`.
```

Rendered Output

Recall the `static_cast` syntax: `static_cast < type-id > ( expression )`.

Because italicized wording terms are so common, `$text$` is provided as a shorthand for `@*text*@`.

Markdown Source

```
Recall the `static_cast` syntax: `static_cast < $type-id$ > ( $expression$ )`.
```

Rendered Output

Recall the `static_cast` syntax: `static_cast < type-id > ( expression )`.

A more elaborate example with modifying text:

Markdown Source

```
```cpp
template <@[invocable](class){.sub}@ F@[, class]{.add}@>
struct $as-receiver$ {
@[private:]{.rm}@
  @[using invocable_type = std::remove_cvref_t<F>;]{.rm}@
  @[invocable_type](F){.sub}@ f_;
@[public:]{.rm}@
  @[explicit *as-receiver*(invocable_type&& f)]{.rm}@
  @[*as-receiver*(*as-receiver*&& other) = default;]{.rm}@
  void set_value() @[noexcept(is_nothrow_invocable_v<F&>)]{.add}@ {
    invoke(f_);
  }
  @[[[noreturn]]]{.add}@ void set_error(std::exception_ptr) @[noexcept]{.add}@ {
    terminate();
  }
  void set_done() noexcept {}
```
```

```
};
```
```

**Rendered Output**

```
template <invocableclass F, class>
struct as-receiver {
private:
  using invocable_type = std::remove_cvref_t<F>;
  invocable_typeF f_;
public:
  explicit as-receiver(invocable_type&& f)
  as-receiver(as-receiver&& other) = default;
  void set_value() noexcept(is_nothrow_invocable_v<F&>) {
    invoke(f_);
  }
  [[noreturn]] void set_error(std::exception_ptr) noexcept {
    terminate();
  }
  void set_done() noexcept {}
};
```

4.6.3.1 Code within Embedded Markdown

Suppose we want to **add** a parameter `int i` to a function `f`:

Markdown Source	Rendered Output
``cpp void f(@[int i]{.add}@); ```	`void f(int i);`

This is perfectly fine, and quite intuitive. However, consider if the parameter is something more complicated, like `Widget *const *ptr`:

Markdown Source	Rendered Output
``cpp void f(@[*Widget* *const *ptr]{.add}@); ```	`void f(Widget const ptr);`

Now, the pointers have disappeared and `const` is italicized. This is because the **full text** within `@` is treated as Markdown, which is exactly what we want for the `*Widget*` part of it, for example. It's just not want we want for the `*const *` part of it.

There are a couple of ways to resolve this issue:

1. Introduce an **inline code within the embedded Markdown**:

Markdown Source	Rendered Output
``cpp void f(@[`$Widget$ *const *ptr`]{.add}@); ```	`void f(Widget *const *ptr);`

   Given that `` `$Widget$ *const *ptr` `` is exactly how it would be written if we were writing inline code, this is a decent solution.
```

However, this approach can't really handle markup more complicated than italicizing because the parsing of `@` is extremely naive in its current implementation.

2. **Escape the special Markdown characters** that you want interpreted literally. Any symbol can be escaped with a backslash like `\*`, and is interpreted literally by the Pandoc extension: `all_symbols_escapable`.

Markdown Source	Rendered Output
`` ```cpp void f(@[*Widget* *const *ptr]{.add}@); ``` ````	`void f(Widget *const *ptr);`

This approach can handle more complicated markup requirements. For example, if `Widget` needed to be bolded instead, we can just do:

Markdown Source	Rendered Output
`` ```cpp void f(@[**Widget** *const *ptr]{.add}@); ``` ````	`void f(Widget *const *ptr);`

4.6.3.2 Opt-out for Default Languages: `.raw`

Add the `.raw` class to opt-out of embedded Markdown, using Pandoc extensions `inline_code_attributes` or `fenced_code_attributes`.

One example of a problem involving emails:

Markdown Source	Rendered Output
`` ```cpp {.raw} auto emails = { "a@mail.com", "b@mail.com" }; ``` ````	`auto emails = { "a@mail.com", "b@mail.com" };`
`` ```cpp auto emails = { "a@mail.com", "b@mail.com" }; ``` ````	`auto emails = { "amail.com", "bmail.com" };`

The `@` characters "disappear" without the `.raw` class because the text between the two `@`, `domain.com", "bar` is parsed as Markdown, and placed back into the full range of `@domain.com", "bar@`.

4.6.3.3 Opt-in for Other Languages: `.embed_md`

Add the `.embed_md` class to opt-in for embedded Markdown, using Pandoc extensions `inline_code_attributes` or `fenced_code_attributes`.

Markdown Source	Rendered Output
`` ```rust fn main() { @**println**@!("hello!"); } ``` ````	`fn main() {` `    @**println**@!("hello!");` `}`

Markdown Source	Rendered Output
``rust {.embed_md} fn main() { @**println**@!("hello!"); } ```	fn main() { println!("hello!"); }
`s.trim_@[left](start){.sub}@()`{.rust}	s.trim_@[left](start){.sub}@()
`s.trim_@[left](start){.sub}@()`{.rust .embed_md}	s.trim_~~left~~start()

4.6.3.4 Overriding the Delimiters: `md`, `em`

The final escape hatch, is to change the default Markdown and italics delimiters from `@` and `$` respectively, to something else.

Add the attribute `md=<symbol>|none` to set the Markdown delimiter or disable, and `em=<symbol>|none` to set the italics delimiter or disable.

Consider this `bash` example:

Markdown Source

```
```bash
rsync -av "$@" "deploy@$target:/srv/app/"
```
```

Rendered Output

```
rsync -av "$@" "deploy@$target:/srv/app/"
```

If we want to bold the `"$@"` for example, we need to enable embedded Markdown. However, the two occurences each of `@` and `$` make the default delimiters (`@` and `$`) unusable. In this example, we set the Markdown delimiter to be `%` instead, and disable the italics delimiter entirely.

Markdown Source

```
```bash {md=% em=none}
rsync -av %**"$@"**% "deploy@$target:/srv/app/"
```
```

Rendered Output

```
rsync -av "$@" "deploy@$target:/srv/app/"
```

[*Note:* Effectively, `.embed_md` is `md=@ em=$` and `.raw` is `md=none em=none`. — *end note*]

4.7 Comparison Tables

Comparison Tables are fenced `Div` blocks that open with `::: cmptable` and close with `:::`. Fenced code blocks are the only elements that actually get added to Comparison Tables, except that the last header (if any) before a fenced code block is attached to the cell above.

Markdown Source

```
::: cmptable
### Before
```cpp
switch (x) {
  case 0: std::cout << "got zero"; break;
  case 1: std::cout << "got one"; break;
  default: std::cout << "don't care";
}
```
```

````
### After
```cpp
x match {
  0 => do { std::cout << "got zero" };
  1 => do { std::cout << "got one" };
  _ => do { std::cout << "don't care" };
};
```
:::
````

Rendered Output

Before	After
`switch (x) {` `  case 0: std::cout << "got zero"; break;` `  case 1: std::cout << "got one"; break;` `  default: std::cout << "don't care";` `}`	`x match {` `  0 => do { std::cout << "got zero" };` `  1 => do { std::cout << "got one" };` `  _ => do { std::cout << "don't care" };` `};`

Each fenced code block is pushed onto the current row, and horizontal rules (`---`) are used to move to the next row.

Markdown Source

````
::: cmptable

### Before
```cpp
switch (x) {
  case 0: std::cout << "got zero"; break;
  case 1: std::cout << "got one"; break;
  default: std::cout << "don't care";
}
```

### After
```cpp
x match {
  0 => do { std::cout << "got zero" };
  1 => do { std::cout << "got one" };
  _ => do { std::cout << "don't care" };
};
```

---

```cpp
if (s == "foo") {
  std::cout << "got foo";
} else if (s == "bar") {
  std::cout << "got bar";
} else {
  std::cout << "don't care";
}
```

```cpp
````
````

```
s match {
  "foo" => do { std::cout << "got foo" };
  "bar" => do { std::cout << "got bar" };
  _ => do { std::cout << "don't care" };
};
```

:::

Rendered Output

Before	After
`switch (x) { case 0: std::cout << "got zero"; break; case 1: std::cout << "got one"; break; default: std::cout << "don't care"; }`	`x match { 0 => do { std::cout << "got zero" }; 1 => do { std::cout << "got one" }; _ => do { std::cout << "don't care" }; };`
`if (s == "foo") { std::cout << "got foo"; } else if (s == "bar") { std::cout << "got bar"; } else { std::cout << "don't care"; }`	`s match { "foo" => do { std::cout << "got foo" }; "bar" => do { std::cout << "got bar" }; _ => do { std::cout << "don't care" }; };`

The last block quote `> caption` (if any) is used as the caption.

Markdown Source

````
::: cmptable

> Put your caption here

### Before
```cpp
switch (x) {
  case 0: std::cout << "got zero"; break;
  case 1: std::cout << "got one"; break;
  default: std::cout << "don't care";
}
```

### After
```cpp
x match {
  0 => do { std::cout << "got zero" };
  1 => do { std::cout << "got one" };
  _ => do { std::cout << "don't care" };
};
```

:::
````

Rendered Output

Table 24: Put your caption here

Before	After
`switch (x) {` `  case 0: std::cout << "got zero"; break;` `  case 1: std::cout << "got one"; break;` `  default: std::cout << "don't care";` `}`	`x match {` `  0 => do { std::cout << "got zero" };` `  1 => do { std::cout << "got one" };` `  _ => do { std::cout << "don't care" };` `};`

4.8 Stable Names

Stable names come in two flavors: *explicit* and *implicit* .

[*Note:* Run `make update` to re-fetch and update the local databases, including stable names. — *end note*]

4.8.1 Implicit Stable Names

Implicit stable names are `shortcut_reference_links` such as `[stable.name]`. It automatically links to https://eel.is/c++draft/stable.name, if such a stable name exists. This is typically used in quoting standard paragraphs like this:

Markdown Source

```
> [...] whose lifetime has begun and has not ended ([basic.life]).
```

Rendered Output

> [...] whose lifetime has begun and has not ended ([basic.life]).

4.8.2 Explicit Stable Names

Explicit stable names are bracketed `Span` elements that look like: `[stable.name]{.sref}`. By default, a leading section number and the section title is automatically rendered.

For example:

Markdown Source	Rendered Output
`Modify section [basic.life]{.sref}:`	Modify section 6.8.4 Lifetime [basic.life]:

This is also useful in a header when updating large bodies of wording:

Markdown Source	Rendered Output
`## [basic.life]{.sref} {- .unlisted}`	**6.8.4 Lifetime [basic.life]**

[*Note:* The `{- .unlisted}` is applied to the header itself, to disable the section numbering and to exclude it from the table of contents — *end note*]

Lastly, you may also add the class `-` or `.unnumbered` to to omit the section number.

This is useful if you prefer to use the regular header numbering instead:

Markdown Source	Rendered Output
`## [basic.life]{- .sref}`	**4.9 Lifetime [basic.life]**

See Numbering of Explicit Stable Names for how to disable numbering at the document-level.

4.8.3 Paragraph Link

A suffix `/pnum` can be added to link to a specific paragraph number, where `pnum` is a dot-separated integers like `1` or `2.1`.

Markdown Source	Rendered Output
`Refer to a specific paragraph [basic.life]/1:`	Refer to a specific paragraph [basic.life]/1:
`Change [basic.life]{.sref}/2.1 as follows:`	Change 6.8.4 Lifetime [basic.life]/2.1 as follows:

4.9 Citations

In-text citations look like this: `[@paper]`

Markdown Source

```
This is a proposal for a reduced initial set of features to support static
reflection in C++. Specifically we are mostly proposing a subset of features
suggested in [@P1240R2].
```

Rendered Output

This is a proposal for a reduced initial set of features to support static reflection in C++. Specifically we are mostly proposing a subset of features suggested in [P1240R2].

You may also include the title of the paper by adding the `.title` class, like `[@paper]{.title}` which generates:

Markdown Source

```
This is a proposal for a reduced initial set of features to support static
reflection in C++. Specifically we are mostly proposing a subset of features
suggested in [@P1240R2]{.title}.
```

Rendered Output

This is a proposal for a reduced initial set of features to support static reflection in C++. Specifically we are mostly proposing a subset of features suggested in [P1240R2] (Scalable Reflection).

4.10 References

4.10.1 Automatic References

The bibliography is automatically generated from https://wg21.link/index.yaml for citations of the following types.

Type	Identifier
Paper	**N***xxxx* / **P***xxxx***R***n*
Issue	**CWG***xxxx* / **EWG***xxxx* / **LWG***xxxx* / **LEWG***xxxx* / **FS***xxxx*
Editorial	**EDIT***xxx*
Standing Document	**SD***x*

The `[@P1240R2]` example from Citations produces a bibliography entry: [P1240R2] in References.

[*Note:* Run `make update` to re-fetch and update the local databases, including the automatic references. — *end note*]

4.10.2 Manual References

Manual references are specified in a YAML metadata block similar to Title, typically at the bottom of the document.

Markdown Source

```
The `id` field is for in-text citations (e.g., [@PAT]),
and `citation-label` is the label for the reference.

Typically `id` and `citation-label` are kept the same.

---
references:
  - id: PAT
    citation-label: Patterns
    title: "Pattern Matching in C++"
    author:
      - family: Park
        given: Michael
    URL: https://github.com/mpark/patterns
---
```

Rendered Output

The `id` field is for in-text citations (e.g., [Patterns]), and `citation-label` is the label for the reference.

Typically `id` and `citation-label` are kept the same.

This produces a bibliography entry [Patterns] in References.

5 Configurations

5.1 Default Language for Code Elements

As mentioned in sections Inline Code and Code Block, inline code elements are C++ syntax highlighted by default, while code blocks are not.

The default configuration is:

```
---
highlighting:
  inline-code: cpp
  code-block: default
---
```

You could change this for your document by adding both or either entries to the YAML metadata block. For example, if you also want code blocks to be C++ by default:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
highlighting:
  code-block: cpp
---
```

or if you want inline-code to be treated normally, not as C++:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
highlighting:
  inline-code: default
---
```

5.2 Embedded Markdown by Default Code Classes

Embedded Markdown is enabled by default for `cpp` and `default` code elements.

While the `.embed_md` class should be suffcient for most use cases to enable embedded Markdown as needed, if you want to change the default, you can set the `embedded-md-code-classes` YAML metadata. For example:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
embedded-md-code-classes:
  - cpp
  - default
  - diff
  - nasm
  - rust
---
```

This is an **override** for the existing list of `cpp` and `default`, so if it's desired to keep `cpp` and `default` to have embedded Markdown enabled by default, they must be listed again.

5.3 Numbering of Explicit Stable Names

By default *explicit* stable names such as `[basic.life]{.sref}` renders with a leading section number and the section title.

As mentioned in Explicit Stable Names, you can disable numbering at the element-level via `[basic.life]{- .sref}` or `[basic.life]{.unnumbered .sref}`. To disable at the document-level, you can specify `number-srefs: false` in the YAML metadata:

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
number-srefs: false
---
```

5.4 Unicode Fonts

If building for PDF (via LaTeX) output with Unicode characters, you may want to select specific Fonts for rendering. For example, `monofont` can be specified in the YAML metadata block to select a font for code elements.

```
---
title: "`MPark/WG21` User's Guide"
subtitle: "Framework for Writing C++ Committee Proposals"
document: D0000R0
date: today
audience: WG21
author:
  - name: Michael Park
    email: <mcypark@gmail.com>
monofont: "DejaVu Sans Mono"
---
```

"DejaVu Sans Mono" provides glyphs for a large amount of the Unicode characters. If you want the list of available fonts on your system, most supported systems will produce a list via the command-line tool `fc-list`.

6 License

Distributed under the Boost Software License, Version 1.0.

7 Resources

— Blog Post: How I format my C++ papers
— Lightning Talk @ C++Now 2019: WG21 Paper in Markdown

8 References

[P1240R2] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, Faisal Vali. 2022-01-14. Scalable Reflection.
https://wg21.link/p1240r2
[Patterns] Michael Park. Pattern Matching in C++.
https://github.com/mpark/patterns