BDE C++ Coding Standards

October 12, 2023

Contents

1 Introduction

1.1

This document specifies the coding standards to be used for C++ written at Bloomberg. All newly developed C++ software must adhere to these standards.

> **Note:** Adoption of a set of coding standards is done to promote code quality and improve maintainability. A regular and consistent form for the production of code provides benefit to the author, to those who must maintain the code, and to the users of the code who can come to learn how to use new APIs in a predictable manner. Coding standards may also improve portability across platforms, by ensuring that issues are carefully documented for those who may not be aware of issues specific to a platform.
>
> **Note:** Although these standards may be usefully adopted when modifying so-called "legacy" code, there is no requirement that existing code be revised to conform.

1.2

This document is organized into a set of enumerated *rules*, each of which may specify a definition, a requirement, or a recommendation. To signify the meaning of a rule, specific key words are used. In particular,

> *The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" are to be interpreted as described in IETF RFC 2119 (http://www.ietf.org/rfc/rfc2119.txt).*

1.3

A rule shall be denoted by a rule number, relative to the section in which it is defined, and prefixed by the section number. Tools developed to promote conformance with this coding standard must use these rule numbers when reporting a violation.

> **Example:** This rule is Rule 1.3 , part of Section 1.

1.4

Corrections and omissions will be recorded in a separate errata. Periodically, the accumulated errata will be incorporated into a revised version of this document. Rules may be renumbered only with the release of a revised version. Though every attempt will be made to keep rule numbers stable across such versions of this document, this cannot be guaranteed.

1.5

A rule may be accompanied by supplemental notes and/or examples. This supplemental text is non-normative; that is, it is not considered part of the rule.

> **Example:** Both this example and the note in Rule 1.1 are non-normative.

1.6

All code that complies with these standards shall first comply with the C++ 2003 standard (ISO/IEC 14882:2003).

1.7

English shall be the language used for all text in code complying with these standards (see Section 3.4.0).

2 Units of Release

2.1 Package Groups

2.1.1

A *unit of release* (UOR) is either a package group or a standalone package. A *package group* is a broad-based, cohesive grouping of packages that share a common envelope of physical dependencies. A *package* is a collection of components that are logically and physically related. Every component must belong to some package.

2.1.2

A package group must be designated by a name that is exactly three letters long, and must be unique within the enterprise.

2.1.3

The name of a package belonging to a package group is formed by the concatenation of the package group name and a one-to-three-letter mnemonic that identifies the specific package in the package group.

Example: The `bdl` package group contains packages named `bdls`, `bdlma`, `bdlt`, and so on.

2.2 Standalone Packages

2.2.1

A *standalone package* is a package that does not belong to any package group.

2.2.2

The package name of a standalone package must begin with a prefix indicating the type of standalone package, followed by an underscore, followed by the rest of the package name. The prefixes to use are beyond the scope of this document.

2.2.3

An application package is a standalone package that implements an executable and defines a `main` function. No other package may depend on an application package.

2.2.4

The `main` function of an application package must reside in a separate file that is not rendered as a component. This file must use the suffix `.m.cpp`.

The `.m.cpp` file must contain only

a. the relevant `#include` directives

b. the definition of the main function

c. definitions of other entities that do not have external linkage

3 Components

3.1 Physical Design

3.1.1

A *component* is the smallest unit of physical design. All source code in a unit of release (other than the `main` entrypoint of an application; Rule 2.2.4) must reside in one or more components.

3.1.2

The *base name* of a component is a unique character sequence that distinguishes the component from all other components in the same package. The base name should be the same as the name of the principal class of the component if it has one; otherwise, it should be any distinctive name that characterizes the content of the component.

3.1.3

The name of a component is formed by the concatenation of the package name and the lowercased base name of the component, separated by an underscore (_).

Example:

Principal Class	Component Name
bdex::TestOutStreamFormatter	bdex_testoutstreamformatter
bdlt::Date	bdlt_date
ball::LoggerManager	ball_loggermanager

3.1.4

A component consists of a header file (`.h`) and an implementation file (`.cpp`) and has an associated test driver (`.t.cpp`). Each file is named using the component name and the appropriate suffix. All three files must reside in the directory of the package to which the component belongs.

3.1.5

A component must conform to the following rules:

a. The `.cpp` file must include the corresponding `.h` file as its first substantive line of code (Rule 5.5).

b. All constructs with external linkage defined in a `.cpp` file must be declared in the corresponding `.h` file.

c. All constructs with external linkage declared in a `.h` file must be defined within the component, if defined at all.

d. A component's functionality must be accessed only via a `#include` of its header, and never via a forward (`extern`) declaration.

3.1.6

A component that directly uses another component must explicitly include the header for that component. A component is said to *directly use* another component if some construct within the using component makes use of some entity declared in the used component, and the using component needs to see the *definition* of that entity in order to compile. A component must not rely on transitive inclusion of headers for components on which it directly depends.

Note: Note that when using a derived class, it is not necessary to include the component that defines the base class, as the component defining the derived class must have already included the component defining the base class.

3.1.7

A component *b* is said to depend on another component *a* if component *b* includes the header of component *a*. A package *q* is said to depend on another package *r* if package *q* contains a component that depends on any component contained by package *r*.

3.1.8

The graph of dependencies among components, packages, and package groups must be acyclic. That is, there shall be no cycles

a. among components within packages

b. among packages within a package group

c. across package groups

Note: These rules ensure that the graph of dependencies among UORs, packages within a group, and components within a package are all acyclic. Such graphs can be sorted topologically by their distance to external nodes. These distance numbers are referred to as "hierarchical levels"; hierarchies without cycles are said to be "levelizable". Levelizable software implies that there exists at least one ordering in which every component can be tested thoroughly depending only on other components that have already been tested thoroughly.

3.2 Logical Design

3.2.1

Throughout this document, the term *class* shall be used to refer to types declared using either the `class` or `struct` keywords.

3.2.2

Each externally usable class defined in the package-level namespace must be rendered as a separate component, with only the following exceptions:

a. to avoid logical encapsulation spanning physical boundaries (i.e. "no long-distance friendship") (Rule 3.2.3)

b. to avoid cyclic dependencies across component boundaries (Rule 3.1.8)

c. for peer entities that individually provide no useful purpose, but together provide a cohesive solution (i.e., "single solution")

d. for an entity that provides a tiny amount of cohesive functionality atop a substantial class (i.e., "flea on an elephant").

3.2.3

Friendship must not be granted to entities defined in another component. For friend declarations in the same component, see Section 6.6.

3.2.4

Functionality provided to assist the principal class defined in a component that can be implemented efficiently using only the public interface is known as *non-primitive* functionality. Non-primitive functionality should be provided in a utility class with the same name as the principal class plus the added suffix `Util`. Such a utility class should be defined in a separate component.

Example:

```
class ZoneInfo {
    // . . .
};

struct ZoneInfoUtil {
    // This "struct" provides a namespace for utility
    // operations using a "ZoneInfo" object.

    // . . .
};
```

3.2.5

A *local definition* is a definition that is needed by the implementation of the component but is not part of the public interface of the component. Local definitions must be placed in the anonymous namespace in the implementation file (Rule 5.9), with a possible exception for local classes (Rule 3.2.6).

3.2.5.1

While classes, functions, variables, and enums used only within an implementation file must be defined in the unnamed namespace, they may be defined within a namespace `u` within the unnamed namespace. Thus, when the reader encounters the use of an identifier preceded by `u::`, they will know where to find the definition.

3.2.6

A *local class* is a class intended for use only by the implementation. A local class must not be used outside of the component in which it is defined. For implementation reasons, the definition of a local class may be placed in a component's header file; otherwise, it must be placed in the anonymous namespace in the implementation file.

3.2.7

The name of a local class shall be formed by the concatenation of the component base name and a descriptive type name (first letter uppercased), separated by an underscore.

Note: For implementation reasons, the definition of a local class may need to be placed in a component's header file, where it cannot be given internal linkage. To avoid conflicts, therefore, the name of the local class is prefixed by the component name. For consistency, this is also done for local classes that are defined only in the anonymous namespace of the implementation file.

Example:

```
template <class t_TYPE>
struct LinkedList_Link {
    // This "struct" implements the storage required for
    // a linked list of elements.

    bsls::ObjectBuffer<t_TYPE>  d_data;
    LinkedList_Link            *d_next_p;

    LinkedList_Link(const t_TYPE&       value,
                    LinkedList_Link    *next,
                    bslma::Allocator   *basicAllocator)
    : d_data(value, allocator)
    , d_next_p(next)
    { }
};
```

3.3 Namespaces

3.3.1

All logical entities must be defined within the enterprise-wide namespace named `BloombergLP`.

3.3.2

All logical entities must also be declared within a second namespace, known as the *package namespace*, named after the package containing the component. The package namespace must be enclosed by the enterprise-wide namespace.

3.3.3

Only the following entities may be declared directly within a package namespace:

a. classes

b. `swap` functions

c. free operators

All other entities must be declared within a class contained within the package namespace.

3.3.4

When using the name of an entity belonging to a package other than the one for the current component, the name must be qualified with the other package's namespace. Conversely, when using the name of an entity belonging to the current package, the name must not be qualified by the package namespace.

Example: In the following code, `Calendar` does not need to be (and must not be) qualified because it already belongs to the `bdlt` namespace, but `ostream` must be qualified with the `bsl` namespace. Similarly, `Date` must be qualified with the `bdlt` namespace. (See also Section 6.9 for more about free operators).

```
namespace bdlt {

bsl::ostream& operator<<(bsl::ostream&   stream,
                         const Calendar& calendar);
    // . . .

inline
bool Calendar::isBusinessDay(const bdlt::Date& date)
{
    // . . .
}

}
```

Note: The name of the component containing a particular qualified name can be derived from its usage by replacing the namespace scope operator with an underscore, and lowercasing the concatenated name.

3.3.5

A `using namespace` clause must not be used except in the following cases:

a. within test drivers

b. within the `.m.cpp` source file (Rule 2.2.4)

c. within the block scope of a function when using `bdlf::PlaceHolders` (refer to the component documentation for more information)

3.4 General Naming Conventions

3.4.0

All text in source code shall be in English, with the appropriate application-specific jargon allowed when the meaning is unambiguous (though note that the use of abbreviations should always be limited, see Section 3.4.3). This includes all names, documentation, comments, log messages, and internal error messages. This does not include string constants needed for essential behavior (including those used by test drivers to test behavior).

3.4.1

The name of a logical entity should clearly describe the role of the entity. The name of a type or a variable should be a noun. Additionally, the name of a type should indicate capability, not purpose. The name of a variable should indicate purpose, not structure. In general, the name of a method should be a verb phrase (but see also Section 7.1).

3.4.2

The name of a logical entity may be a compound word. If so, all words other than the first must be capitalized. The case of the first character in the name depends on the entity type. The first character of a type name must be an uppercase letter. The first character of a function or variable name must be a lowercase letter.

Example: Consider the following function declaration, which illustrates a type name, a function name, and a variable name:

```
bdlt::Date getNextBusinessDay(const bdlt::Date& initialDate);
```

3.4.3

Entity names, other than names with block scope declared in the bodies of functions, must not use abbreviations, except those adopted as a group convention within a particular development group. The following abbreviations may be used (for the prescribed meaning) anywhere.

Abbreviation	Meaning
`cb`	callback
`dst`	destination
`id`	identifier (not "identity")
`init`	initializ(e,er)
`iter`	iterator
`max`	maximum
`min`	minimum
`msg`	message
`num`	number
`pos`	position
`ptr`	pointer
`ref`	reference
`src`	source
`tmp`	temporary

Note that acronyms, where they are well-understood in the domain in which they are being used, may also be used.

Example: The following are all legal uses of abbreviations:

```
class CalendarBusinessDayConstIter;

int numConnections, srcIndex, dstIndex;
const Type& elementRef;
bsl::string typeId;
const Type *elementPtr;

static void convertUtcToLocalTime(
     bdlt::DatetimeTz                          *resultTime,
     baltzo::Zoneinfo::TransitionConstIterator *resultTransition,
```

```
    const bdlt::Datetime&                       utcTime,
    const baltzo::Zoneinfo&                     timeZone);
```

3.4.4

The name of a template parameter must start with `t_`, with one or more following words in uppercase, separated by underscores. If only a single word is used, it must not consist of a single letter (e.g., `t_T`).

Example:

```
template <class t_VALUE_TYPE, bsl::size_t t_SIZE>
struct array;
```

3.4.4-rationale

4 Header Files

4.1 General Organization

4.1.1

If a component defines or implements a logical entity having external linkage, then that entity must be declared within that component's header file, and only within that component's header file. The corresponding definition must be in the `.cpp` file unless the entity is an inline function, a template, or both.

4.1.2

The organization of a header file must follow the sequence specified here:

a. a prologue (Section 4.2)

b. component-level documentation (Section 8)

c. include directives (Section 4.3)

d. forward declarations (Rule 4.4.2)

e. class definitions, including associated free operator declarations (Section 6)

f. inline function and function template definitions (Section 4.5)

g. inline free operator and free operator template definitions (Section 6.9)

h. trait specializations

i. an epilogue (Section 4.6)

All but the prologue, the epilogue, and the component-level documentation are optional.

4.2 Prologue

4.2.1

The prologue consists of

a. an initial comment stating the filename and language tag

b. the include guard for the file

Example:

```
// mypackagemyclass.h                                        -*-C++-*-
#ifndef INCLUDED_MYPACKAGE_MYCLASS
#define INCLUDED_MYPACKAGE_MYCLASS
```

4.2.2

The language tag for C++ is `-*-C++-*-`. It must appear in the first line of the source file, right-justified to the 79th column.

4.2.3

The name of the include guard macro is formed by the concatenation of the prefix `INCLUDED` and the uppercased component name (Rule 3.1.3), separated by an underscore.

4.3 Include Directives

4.3.1

Include directives must immediately follow the component-level documentation (Section 8). The header file for each component used directly must be included by an explicit include directive (Rule 3.1.6).

4.3.2

OBSOLETE (Rule 14.1)

4.3.3

OBSOLETE (Rule 14.2)

4.3.4

All include directives must use angle brackets (i.e., `'< >'`); the quoted form must not be used.

4.3.5

Include directives must be divided into three groups:

a. components in the same package as this component

b. components from other packages

c. other required headers that are not components (e.g. third-party or platform-specific)

Include directives for each non-empty groups must appear in the order shown. Within each group, the include groups should be ordered alphabetically. Headers belonging to the third group may require a specific, non-alphabetic ordering.

> **Note:** Where avoidable, expensive standard library headers should not be included in a header file. For example, It is usually possible to include `<bsl_iosfwd.h>` instead of `<bsl_iostream.h>` . For components defining class templates with `operator<<` , however, it will be necessary to include `<bsl_iostream.h>` directly in the header file.

4.3.6

Where available, the `bsl` standard headers must be used in place of the STL headers.

4.4 Declarations

4.4.1

Everything following the include directives and prior to the epilogue must be placed inside the enterprise-wide namespace (Rule 3.3.1) (except forward declarations for third-party entities, as necessary). Moreover, class definitions and inline member function definitions must be placed in the package namespace (Rule 3.3.2). In the header file, the following layout must be used to achieve this placement:

```
. . . // forward declarations for non-enterprise entities go here

namespace BloombergLP {

. . . // forward declarations for entities within this unit of release
. . . // but outside of this package go here

. . . // forward declarations are *not* used for entities outside the
. . . // current unit of release - use the corresponding `#include`
. . . // instead

namespace *package* {

. . . // forward declarations for entities in the same package go here

. . . // class definitions, including associated free operator
      // declarations, go here

. . . // inline member function definitions go here

}  // close package namespace

. . . // inline free (operator) definitions go here

. . . // trait specializations go here

}  // close enterprise namespace
```

Note that inline free (operator) *definitions* are placed outside of the package namespace, with no separate banner or category tag (Section 6.9). Note also that package is to be replaced with the component's package name. Finally, note that there must be two spaces preceding the trailing comments, and that blank lines should be used as shown to separate the sections; however, if there are no free operator definitions or trait specializations, there should be no blank line between the lines that close the namespaces.

4.4.2

Per Rule 3.1.5, if a component uses an entity having external linkage, then it must include that component's header file.

When using a class, even if the class definition is not needed for compilation (i.e., no size or layout information is needed), the component should include the component's header file, unless the entity is located in the same unit-of-release (UOR). Even within the same UOR, includes should be preferred over forward declarations absent a reason to use the forward declaration. (see rationale for *4.4.3*).

4.4.3

Forward declarations for classes defined in other packages must not be used. The corresponding *#include* statement should be used instead.

4.4.3-rationale

4.4.4

Forward declarations for classes defined in the same package as the current component must appear immediately after the package namespace is opened. If there are multiple declarations, each declaration should appear on its own line, sorted alphabetically. These forward declarations should only be used to avoid cycles - otherwise, include statements should be used.

4.4.5

Forward declarations for classes defined in the current component, if any, must appear after all other forward declarations, in the order in which they are defined in the source file.

4.5 Inline Definitions

4.5.1

Inline function definitions having external linkage must appear in the header file in a separate section that follows the class definitions but is still inside the package namespace. The section must begin with the comment banner shown:

```
// ============================================================================
//                            INLINE DEFINITIONS
// ============================================================================
```

The banner must extend to the 79th column. The initial `I` in `INLINE` must begin at the 29th column. (Note that the above illustration is not shown with the correct number of columns, due to typesetting constraints).

4.5.2

Inline functions are defined exactly as regular member functions are defined in an implementation file (Section 6.8), except that each function definition must include the keyword `inline` on a line by itself immediately preceding the rest of the function definition. In particular, the rules concerning grouping, ordering, and the appearance of the class-specific implementation comment banner must all be followed (Section 6.8.3, and Rule 6.1.3).

Example:

```
// CLASS METHODS
inline
bool baltzo::LocalTimeDescriptor::isValidUtcOffsetInSeconds(int value)
{
      return value >= -86399 && value <= 86399;
}
```

4.6 Epilogue

4.6.1

The epilogue consists of

a. the `#endif` of the opening include guard

b. the copyright notice

The `#endif` directive must not be followed by a comment.

4.6.2

Unless the files are published for open source (Rule 4.6.3), the copyright notice must appear exactly as shown:

```
// ----------------------------------------------------------------
// NOTICE:
// Copyright 2012 Bloomberg Finance L.P. All rights reserved.
// Property of Bloomberg Finance L.P. (BFLP)
// This software is made available solely pursuant to the
// terms of a BFLP license agreement which governs its use.
// ----------------------- END-OF-FILE ------------------------
```

The year specified in the copyright notice must be the year in which the source file was first created.

Note: If you create a new source file, even if it incorporates previously written code, use the current year in the copyright notice because this new, derivative work did not exist earlier. If the code is subsequently modified, the copyright date should be left as is, because that is the year the work was first published.

4.6.3

For open source files, the copyright notice must appear exactly as shown:

```
// ----------------------------------------------------------------------------
// Copyright 2013 Bloomberg Finance L.P.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
// ----------------------------- END-OF-FILE --------------------------------
```

As in the previous rule Rule 4.6.2, the year specified in the copyright notice must be the year in which the source file was first created.

Note: As in the previous rule Rule 4.6.2, if you create a new source file, even if it incorporates previously written code, use the current year in the copyright notice because this new, derivative

work did not exist earlier. If the code is subsequently modified, the copyright date should be left as is, because that is the year the work was first published.

5 Implementation Files

5.1

An implementation file (`.cpp`) must exist for each component (Rule 3.1.4).

5.2

The organization of an implementation file must follow the sequence specified here:

a. a prologue (Rule 5.3)

b. implementation notes (Rule 5.4)

c. include directives (Rule 5.5)

d. local definitions (Rule 3.2.5)

e. non-inline function definitions (Section 6.8)

f. non-inline free operators (Section 6.9)

g. the copyright notice (Section 4.6)

All but the prologue, the include directive for the component's own header file, and the copyright notice are optional.

5.3

The prologue consists of an initial comment stating the filename and language tag. This is the same as for the header prologue (Section 4.2), except that there is no include guard.

5.4

Implementation notes may be provided if the component's implementation deserves some explanation, statement of invariants, or discussion of possible design and implementation choices. Implementation notes should be placed in the `.cpp` file even if the implementation being described is largely inlined or template-based (and thus placed mostly in the header).

If implementation notes are included, they must follow the prologue and begin as shown:

```
///Implementation Notes
///--------------------
// . . . notes go here
```

Implementation notes may use the markup notation (Section 12)as desired.

Note: Implementation notes do not overlap with other component documentation. They are intended for the developers and maintainers of the component itself. They should clarify implementation issues encountered, considered, and even rejected, as well as the particulars of the implementation chosen. The length of the implementation notes should be proportional to the complexity of the implementation. It is not necessary to discuss straightforward or trivial implementations.

5.5

The first substantive line of code in the implementation file of a component must be an `#include` directive that includes the header file of the component (Rule 3.1.5).

Note: By following this rule, successful compilation of the implementation file confirms that the header file is entirely self-contained.

5.6

Other include directives are specified, as needed, following the required include of the component's own header file. The order of these other include directives should follow the same ordering scheme as is used in header files (Rule 4.3.5).

5.6.1

Any include directives used by the test driver that refer to components in the same package and *not* otherwise used by the component itself must appear in the component's implementation file and appended with the comment:

```
// for testing only
```

The for-testing-only include directives must *not* introduce any cycle of physical dependency into the package (see Rule 3.1.8) and must be sorted alphabetically with include directives of other components from the component's package, if any (Rule 4.3.5).

Note: To maximize the potential for re-use, component authors should aim for a component to have as small a set of dependencies as can be reasonably achieved, including for-testing-only dependencies. The `// for testing only` comment serves to highlight when additional dependencies are added to a component through testing.

5.6.1-rationale

5.7

Redundant include guards must not be used in the implementation file.

5.8

Everything following the include directives and prior to the copyright notice must be placed inside the enterprise-wide namespace. Moreover, all definitions, other than those for non-inline, non-template free operators declared in the header, must be placed in the package namespace as well. The structure that must be used to achieve this is shown:

```
namespace BloombergLP {

namespace *package* {

. . . // anonymous namespace for local definitions goes here

. . . // non-inline function definitions go here

}  // close package namespace

. . . // non-inline free operator definitions go here

}  // close enterprise namespace
```

Note that package should be replaced with the component's package name. Note also that there must be no blank line between the opening of the enterprise and package namespaces. There must also be no blank line between the closing of the package and enterprise namespaces if and only if there are no free operator definitions.

5.9

Local definitions (Rule 3.2.5) that appear in an implementation file must be placed inside an anonymous namespace. The anonymous namespace should appear just after the package namespace is opened.

Example The following layout must be used:

```
namespace {

. . . // local definitions go here

}  // close unnamed namespace
```

The ordering of local definitions in the anonymous namespace may be determined by the needs of the implementation. Where a specific ordering is not required, placing local functions before local class definitions is preferred.

5.10

A *local function* is a free function that is not part of any class in the component. Local functions must be defined `static` in the implementation file, inside the anonymous namespace.

Note: The use of the `static` storage specifier for a local function is technically not necessary for a definition inside the anonymous namespace. The function will be given internal linkage regardless. The use of the explicit storage specifier, however, has an ancillary benefit of making prominent that the definition is local, and on some platforms, it is known to reduce the symbol space consumed in the object file and executable image.

6 Class Definitions

6.1 General Form

6.1.1

Class definitions may appear in a header file or in an implementation file. An externally usable class (Rule 3.2.2) must be placed in the header file. A local class definition may be placed in the header file or in the implementation file (Rule 3.2.5). Regardless of where a class definition appears, it must conform to the rules specified in this section.

6.1.2

A class definition must be immediately preceded by a class definition banner and immediately followed by associated free operator *declarations*, if any (Section 6.9). Class-level documentation (Section 9) must appear immediately following the opening of the class definition.

Example:

```
                    // ===============
                    // class ClassName
                    // ===============

class ClassName {

  . . . // class-level documentation goes here

  . . . // member declarations go here

};

. . . // free operator declarations go here
```

6.1.3

The class definition banner may be centered, or indented by 25 spaces if the class name is shorter than 20 characters. The class definition banner uses double-dashed lines. There must be exactly as many dashes as required to ensure that the banner lines are flush with the class name.

6.1.4

Each entity within a class, as well as each associated free operator, must be fully documented at the point of its declaration. The form and content of the documentation varies depending on the entity type.

> **Note:** The individual entity specifications, along with the more general class-level documentation, and broad component-level documentation, are intended to provide three layers of documentation that together provide a complete description of a component, including all essential information required to use the component as well as information about how it is intended to be used.

6.2 Sections

6.2.1

The body of a class definition is divided into public and private sections using the `public:` and `private:` labels. These labels must appear on lines by themselves, indented exactly two spaces, and preceded by a blank line.

> **Note:** The use of a `protected:` section is discouraged. It is permitted only for a compelling and explicitly documented reason, usually linked to structural inheritance. In general, all members are either public or private.

6.2.2

Each access control section is further divided into one or more categories, based on the kind of declaration that appears there. In order of appearance, these categories are as follows:

- (optional) public section
- public types also used to declare private members (Section 6.3)
- private section
- private type definitions used to declare private members (Section 6.3)
- static and non-static data member declarations (Section 6.5)
- friend declarations (Section 6.6)
- declarations of private methods (Section 6.7)
- (optional) private section
- optional disabled declarations (Rule 6.7.4)
- public section
- type definitions used only in the public interface (Section 6.3)
- declarations of public methods (Section 6.7)

Sections should not be included if they contain no declarations. Note that the optional disabled declarations are always placed in their own separately marked `private` section, even if otherwise contiguous with the previous private section.

6.2.3

Each category must be preceded by a tag comment that identifies the category. The tag must appear immediately before the first declaration in that category, with no intervening blank line. A tag must not be used if the category is otherwise empty.

The following tags are defined:

Tag	Usage
`// TYPES*`	type and constant declarations
`// CLASS DATA`	static data members
`// DATA`	non-static data members
`// FRIENDS`	friendship declarations
`// CLASS METHODS*`	static methods
`// CREATORS*`	constructors and destructors
`// MANIPULATORS*`	non-`const` methods
`// ACCESSORS*`	`const` methods
`// FREE OPERATORS`	free operators
`// NOT IMPLEMENTED`	unimplemented standard methods
`// TRAITS`	traits

Tags that are starred (*) may be used in either a public or a private section. When used in a *private* section, the tag must be preceded by `PRIVATE` (e.g., `PRIVATE TYPES`, `PRIVATE ACCESSORS`, etc.).

Example:

```
public:
  // TYPES . . .

private:
  // PRIVATE TYPES . . .

  // CLASS DATA . . .

  // DATA . . .

  // FRIENDS . . .

  // PRIVATE MANIPULATORS . . .

public:
  // CREATORS . . .

  // MANIPULATORS . . .

  // ACCESSORS . . .
```

6.3 Types Sections

6.3.1

Type definitions used to declare private members are the first definitions that appear in the body of a class. These type definitions may be further divided into two sections: those that are also intended to be part of the public interface, and those that are needed only to declare private members. Note that public types needed only for the declaration of public methods are declared just prior to the sections for the declaration of public methods. Each such section should be introduced with an appropriate access control label and section tag as shown:

```
public:
  // TYPES
  . . . // public types needed for private members go here

private:
  // PRIVATE TYPES
  . . . // private types needed for private members go here

  . . . // all other non-public class member declarations

public:
  // TYPES
  . . . // public types needed only for public methods go here
```

6.3.2

The documentation for a `typedef` immediately follows the declaration, indented one level. The documentation should (1) clearly state that the typedef is intended as an alias, and (2) describe the semantics provided the alias, that is, how the alias is intended to be used.

Example:

```
typedef std::size_t SizeType;
    // 'SizeType' is an alias for an unsigned value, representing
    // the size of an object or the number of elements in a range.
    // 'SizeType' occupies one machine word on the underlying platform.

typedef bsl::function<void(int)> ReadCallback;
    // Invoked as a result of any non-buffered read method, 'ReadCallback'
    // is an alias for a callback function object (functor) that takes as
    // an argument an integer status indicating success (0), an incomplete
    // read (>0), or an error (<0).
```

6.4 Types Section Rules Specific to Enumerations

6.4.1

The recommended name for an enumeration within an enumeration component is `Enum`. (An enumeration component is a component dedicated to that enumeration, with public methods that deal only with the mecahnics of the enumeration such as printing and streaming.) The enumeration component itself is a `struct` that provides the namespace for the enumeration and follows the same naming rules as other components (Rule 3.1.3).

Otherwise, enumerations within components should be named appropriately, as defined by convention (Rule 3.4.1). An enumeration existing solely for defining manifest constants within a component need not be named at all.

6.4.2

The name of an enumerator may be prefixed with e_ or k_, and then shall consist of one or more uppercase words separated by an underscore. The e_ prefix is to be used when the enumerator is part of a related set. The k_ prefix is to be used when the enumerator is an architecturally significant constant (that is, publicly visible in a header file or defined and used extensively within an implementation file).

An exception may be made for an enumerator which defines the result of a template metafunction. Following the example of the C++ standard, the enumerator may be named `value`.

Example:

```
// Enumeration component dedicated to 'Membership'.
struct Membership {
    enum Enum {
        e_ATTENDING,
        e_FORMER,
        e_LIFETIME,
        e_RETIRED,
        e_STUDENT,
        e_SUPPORTING,
        k_NUM_MEMBERSHIP_TYPES
    };
};

// Component defining several enumerations used in its interface.
struct FileOperations {
    enum OpenMode {
        e_READ,
        e_WRITE,
        e_APPEND;
    };

    enum PermissionClass {
        e_USER,
        e_GROUP,
        e_OTHER
    };
};

// Enumeration defining a set of manifest constants.
struct MagicNumbers {
    enum {
```

```
        k_ELF = 0x7f000000 + ('E' << 16) + ('L' << 8) + 'F',
        k_GIF = ('G' << 24) + ('I' << 16) + ('F' << 8) + '8',
        k_JPG = 0xFFD8FFE0,
        k_PS = ('%' << 24) + ('!' << 16) + ('P' << 8) + 'S',
        k_PDF = ('%' << 24) + ('P' << 16) + ('D' << 8) + 'F'
    };
};

// Template metafunction.
template <int t_INDEX>
struct Fibonacci {
    enum { value = Fibonacci<t_INDEX-1>::value + Fibonacci<t_INDEX-2>::value };
};
template <> struct Fibonacci<0> { enum { value = 0 }; };
template <> struct Fibonacci<1> { enum { value = 1 }; };
```

6.4.3

Enumerators should be assigned a value explicitly only if the default initialization is not appropriate. However, if at least one enumerator is assigned a value, all enumerators in a given enumeration should be assigned a value.

6.4.4

The documentation for an `enum` follows the opening brace of the declaration, indented one level. A blank line should separate the comment text from the first enumerator.

6.4.5

The documentation for an `enum` should begin with the phrase "Enumeration used to ...".

6.4.6

Additional documentation may be given for each item in an enumeration. If given, it must follow the same form as is used for documenting data members (Section 11.3).

Example:

```
struct Pattern {
    enum Enum {
        // Enumeration used to distinguish among different stroke
        // patterns.

        e_SOLID,     // (___________) solid line
        e_DOT,       // (. . . . . .) dotted line
        e_DASH,      // (_ _ _ _ _ _) dashed line
        e_DASHDOT,   // (_ . _ . _ .) line with dash-dot pattern
        e_DASHDOTDOT // (_ . . _ . .) line with dash-dot-dot pattern
    };
};
```

6.4.7

If there is only one item in an enumeration, the entire declaration may be written on a single line. If there is more than one item, each item must be written on a separate line.

Example:

```
struct Layout {
    enum { k_DEFAULTCHUNKSIZE = 8 }; // . . . brief doc goes here
```

```
};

struct Type {
    enum Enum {
        // . . . general documentation goes here

        e_INT,
        e_DOUBLE,
        e_STRING,
        e_LIST,
        e_TABLE
    };
};
```

6.5 Data Section

6.5.1

Declarations of data members in a class definition must appear in a private section following the section for type definitions, if any. Each data member must be documented according to the rules for variable documentation (Section 11.4).

6.5.2

Data member declarations may be further divided into sections for static and non-static data members. The section for static data members, if present, must appear before the section for non-static data members. The static data member section must be introduced with the `// CLASS DATA` section comment; the non-static data member section must be introduced with the `// DATA` section comment. Within each section, the layout must follow the rules for the layout of variable declarations (Section 11.3).

Example:

```
private:
  // CLASS DATA
  static int s_staticMember; // this is a static member

  // DATA
  int d_nonStaticMember;     // this is a non-static member
```

6.5.3

In addition to the general naming conventions for all entities (Rule 3.4.1), data members must use the following additional conventions:

- Non-static data members begin with the prefix `d_`.
- Static data members begin with the prefix `s_`.
- Any data member that is a pointer ends with the suffix `_p`.

Note that the `_p` suffix is to be used only for raw pointers. Types such as `ManagedPtr` and `SharedPtr` are not raw pointers, and data members having these types should not use the `_p` suffix.

6.5.4

If there is a data member to hold a supplied allocator, it should be declared as the last data member in the class, and must be given the name `d_allocator_p`.

Note: The placement is by convention. Singling the allocator data member out for special placement seems reasonable because whether or not a class allocates memory is a very general division, and knowing where to look to see if there is an allocator data member removes an unnecessary degree of freedom. Making it **last** forces developers not to rely on it being initialized. So, in the initialization of other data members that take allocators, we pass the incoming allocator argument rather than the `d_allocator_p` data member. If this data member exists but is not placed last, the reason must be documented.

6.5.5

If a data member is a pointer, the comment text should include a parenthetical phrase indicating ownership when it is not clear from context. The parenthetical phrases that may be used are "owned" and "not owned".

Note: In short, if you own a bare pointer, document its lifecycle. If you do not own a bare pointer, you had better make it clear that the object pointed to by the bare pointer must have a lifespan that equals or exceeds the object holding the pointer.

6.6 Friends Section

6.6.1

Friend declarations appear following the section for data members, and must be introduced with the `// FRIENDS` section comment.

6.6.2

Friend declarations must always be private, and must refer only to classes and functions declared in the same component (Rule 3.2.3).

6.7 Method Sections

6.7.1

Within each access control section, methods are divided into separate categories for

- static methods (`// CLASS METHODS`)
- constructors and destructors (`//CREATORS`)
- manipulators (`// MANIPULATORS`)
- accessors (`// ACCESSORS`)

in that order. Each category for which there is at least one method declared must begin with the appropriate tag. The tag must be prefixed with `PRIVATE` when used in a private section.

Example:

```
private:
  // PRIVATE CLASS METHODS
  // . . .

  // PRIVATE MANIPULATORS
  // . . .

public:
  // TYPES
  . . .  // types used only in the public interface go here

  // CLASS METHODS
  . . .

  // CREATORS
  . . .

  // MANIPULATORS
  . . .

  // ACCESSORS
  . . .
```

6.7.2

The section for constructors and destructors, labeled `// CREATORS`, comprises the default constructor, other user-defined constructors, the copy constructor, and the destructor, in that order. Any or all of these may be omitted. (See Section 7.4.)

6.7.3

If the assignment operator is defined, it must appear as the first declaration in the section for manipulators. Other member operators may follow, prior to any non-operator methods.

6.7.4

If the copy constructor and assignment operator are to be disabled, they must be declared in a separate private section labeled with the tag `// NOT IMPLEMENTED`, with no other documentation and no parameter names.

Example:

```
private:
  // NOT IMPLEMENTED
  MyClass(const MyClass&);            // = delete
  MyClass& operator=(const MyClass&); // = delete
```

6.8 Method Definitions

6.8.1

If defined at all, the definition for a method must appear in the component in which it is declared (Rule 3.1.5). The definition may appear in the header file (Section 4.5) or in the implementation file (Section 5).

6.8.2

Methods, whether public or private, must not be *defined* inside the text of a class definition, except within a local class definition.

6.8.3

Method definitions for a particular class must be grouped together following a class implementation banner. The class implementation banner is indented just as for a class definition banner (Rule 6.1.3); however, the class implementation banner uses single-dashed lines, rather than the double-dashed lines used for the class definition.

6.8.4

Following the class implementation banner, the method definitions must be separated into categories and demarcated by category tags, just as is done for declarations in the class definition (Section 6.7).

Example:

```
                            // -------------
                            // class MyClass
                            // -------------

// CREATORS
. . .  // creator method definitions go here
```

6.8.5

Each method must be defined in the same relative order in which it was declared in the class definition.

6.8.6

A method must be defined using the same parameter and type names with which it was declared, except when the parameter is unused in the implementation, in which case the parameter name should be omitted in the signature of the definition.

Note: The omission of parameter names when they are not used will avoid compiler warnings of unused variables on some platforms.

6.9 Free Operators

6.9.1

A component may define a free operator only when a type defined in the same component is used as the type of a parameter in the function signature. (Note that the return type is not part of the signature.) These operators are said to be *associated* with the class. They are typically comparison and output operators, but sometimes also arithmetic operators.

6.9.2

If a class has associated free operators, the following rules apply:

- The free operators must be *declared* in a section immediately following the class definition.
- If they are to be inlined, the free operators must be *defined* in a section immediately following the close of the package namespace.
- The category tag for free operators is `// FREE OPERATORS`.

Note: Inline free operators are defined outside of the package namespace in order to prevent inadvertent self-declaration.

Example: The example below illustrates this pattern:

```
namespace BloombergLP {
namespace mypkg {

class MyClass {
    // . . .
};

// FREE OPERATORS
bool operator!=(const MyClass& lhs, const MyClass& rhs);
    // Return 'true' if . . .

}  // close package namespace

// FREE OPERATORS
inline bool mypkg::operator!=(const MyClass& lhs, const MyClass& rhs)
{
    return !(lhs == rhs);
}

}  // close enterprise namespace
```

7 Method Design and Implementation

7.1 Naming Conventions

7.1.1

Manipulators must be named using a verb phrase.

Example:

```
// MANIPULATORS
void removeAll();

// ACCESSORS
int loadHolidays(bsl::vector<bdlt::Date> *businessDays,
                 const bdlt::Date&        firstDate,
                 const bdlt::Date&        lastDate) const;
```

7.1.2

Accessors may be named without a verb phrase if they meet the following criteria:

- they do not alter any observable state
- they are relatively inexpensive to invoke

Such methods must take only non-modifiable parameters. Methods that do not modify any observable state may still use a verb phrase to suggest a non-trivial cost.

Example:

```
bdlt::Date firstDate() const;

bdlt::Date lastDate() const;

bdlt::Date getNextBusinessDay(const bdlt::Date& initialDate,
                              int               numDays) const;
    // . . . uses a verb phrase because of its non-trivial const
```

7.1.3

Methods that access attribute values must use the attribute name alone. Methods that set attribute values must be prefixed with `set`.

Example:

```
// MANIPULATORS
void setCategory(const char *category);

// ACCESSORS
int category() const;
```

7.1.4

Methods that access an attribute whose value can be one of multiple types determined at runtime may begin with the prefix `the`.

Example:

```
class Choice {
    const bool&        theBool() const;
    const int&         theInt() const;
    const bsl::string& theString() const;
};
```

7.2 Interface Conventions

7.2.1

Function arguments are categorized as one of the following:

outputs	**arguments that are potentially** modified to hold results, or that indicate some information about the output; this includes in/out arguments
inputs	**arguments that contain data to be** processed
parameters	**arguments that control *how* data** is to be processed

7.2.2

Arguments in a function signature must be specified in the following order: outputs, inputs, then parameters.

Note: A canonical form for function signatures limits arbitrary freedom in the specification of a function, and increases predictability in usage.

7.2.3

Inputs and parameters should be passed by value for fundamental, enumerated, and pointer types, and by non-modifiable reference otherwise. An argument must be passed by pointer if its address is retained beyond the end of the function call.

7.2.4

Outputs must be passed by pointer, never by modifiable reference (except where required for consistency with an externally-defined API).

7.2.5

Optional arguments follow required arguments in the same relative order.

Example: This example, showing the inclusion of optional arguments, is from `bcec::TimeQueue`:

```
void popLE(bsl::vector<TimeQueueItem<TYPE> > *buffer,          // output
           const bdlt::TimeInterval&          time,            // param
           int                                maxTimers,       // param
           int                               *newLength = 0,   // opt.out
           bdlt::TimeInterval                *newMinTime = 0); // opt.out
```

7.2.6

Arguments that are tightly bound to the output may be treated as output arguments, rather than parameters, even though they are not modified.

Example: The `dstIndex` parameter below is a parameter that specifies which portion of this queue will be overwritten; it is therefore treated as an output argument.

```
void insert(int                dstIndex,      // output
            const Queue<TYPE>& srcQueue,      // input
            int                srcIndex,      // input
            int                numElements);  // parameter
```

7.2.7

The optional allocator parameter must be passed in the last position (except in the case of function templates having overloads that take a variable number of generic arguments, in which case the allocator should be the first argument and made optional by an additional overload).

Example: The allocator in its preferred position:

```
class Queue {
    explicit Queue(bslma::Allocator *basicAllocator = 0);

    explicit Queue(int               initialLength,
                   bslma::Allocator *basicAllocator = 0);

    Queue(int initialLength, const TYPE&  initialValue,
          bslma::Allocator               *basicAllocator = 0);
};
```

Example: The allocator is the first argument here to avoid overloading ambiguity that would otherwise result:

```
struct BindUtil {
    template <class t_FUNC>
    static inline
    ReturnTypeUnspecified bindA(bslma::Allocator *basicAllocator,
                                t_FUNC);
        // . . .
};
```

7.3 Function Signatures

7.3.1

A function signature, including all of its parameters, must be written on a single line if possible (Rule 11.1.3), but see the exception for overloaded functions (Rule 7.3.5).

Example:

```
void loadBuffer(int *result, int index);
```

7.3.2

If a function signature cannot fit on a single line, then it must be written with one parameter per line. The parameters must be written following the rule for when declarations appear on consecutive lines (Section 11.3).

Example:

```
void loadBuffer(int                *result,
                int                 index,
                const bsl::string&  attribute);
```

7.3.3

The first parameter in a multi-line parameter list should be written on the same line as the function name. If the function name or the parameter is too long, then the first parameter may be started on a separate line, indented such that the longest line in the parameter list is right-justified to the 79th column.

Example:

```
static const bdlat::AttributeInfo *lookupAttributeInfo(
                                                  const char *name,
                                                  int         nameLength);
```

7.3.4

If the combination of the return type, class name, and function name are too long to fit on one line, they may be written on separate lines with no indentation.

Example:

```
template <class t_STREAM_BUF>
inline
GenericByteInStream<t_STREAM_BUF>&
GenericByteInStream<t_STREAM_BUF>::getInt40(bsls::Types::Int64& variable);
```

7.3.5

When multiple function overloads share the same function documentation, and at least one of the overloads requires a multi-line parameter list, all of the functions in the group must be written using a multi-line parameter list, and aligned accordingly.

Example:

```
void loadBuffer(int                *result);
void loadBuffer(int                *result,
                int                 index);
void loadBuffer(int                *result,
                int                 index,
                const bsl::string&  attribute);
```

7.3.6

The `virtual` keyword must be used only for declarations of *new* virtual functions. A function overriding a virtual function from a base class must be marked with `BSLS_KEYWORD_OVERRIDE` immediately after the member function declarator. For code that is only required to comply with the C++ 2011 standard or later, the `override` specifier may be used instead of `BSLS_KEYWORD_OVERRIDE`.

Example:

```
class Acceptor {
    virtual int listen(int port) = 0;

    virtual bool isListening() const = 0;
};
```

```
class SampleAcceptor : public Acceptor {
    int listen(int port) BSLS_KEYWORD_OVERRIDE;

    bool isListening() const BSLS_KEYWORD_OVERRIDE;
};
```

7.3.6-rationale

7.4 Special Rules for Creators

7.4.1

The parameter of a copy constructor must be called `original`. See also Rule 10.1.6 for the standard function-level documentation for a copy constructor.

Example:

```
Date(const Date& original);
```

7.4.2

All constructors (other than copy constructors) callable with exactly one parameter must be declared `explicit`, unless there is a compelling reason to do otherwise. If there is such a reason, it must be documented in the function-level documentation (e.g., using "Note that..."), and marked with an `// IMPLICIT` comment, as shown:

```
StringRef(const char *data); // IMPLICIT
```

7.4.3

All constructors of a class that requires dynamic allocation must have a parameter of type pointer to `bslma::Allocator` in the final position. (See Rule 7.2.7 for more about placement, and Rule 10.1.4 for more about documenting it.) The argument may accept a value of 0 to indicate that the default allocator should be used. This value may be supplied as a default for the argument.

7.4.4

The declarations (and definitions) for certain methods may be omitted to allow the compiler to generate them as needed. This choice is visibly documented by adding to the function signature a `//!` prefix and a `= default` suffix. If the signature spans more than one line, the entire declaration must be commented out using the aforementioned special prefix.

Note: The `//!` prefix comments out the signature, but leaves it there for documentation purposes. The suggested suffix is inspired by the default constructor syntax defined in the C++ 2011 standard.

Example:

```
//! ~MyClass() = default;
    // Destroy this object.
```

7.4.5

Compiler-generated methods that should be omitted in regular and optimized builds may be conditionally declared for safe-mode builds.

Example: A destructor that checks invariants can be enabled only in a safe-build mode.

```
#if defined(BSLS_ASSERT_SAFE_IS_ACTIVE)
    // The following destructor is generated by the compiler, except
    // in ``SAFE`` build modes to enable the checking of class
    // invariants.

    ~MyClass();
        // Destroy this object.
#endif
```

7.4.6

Constructors must initialize members using an initialization list if at all possible to do so. Assignment should not be used.

7.4.7

Constructors must list member initializers in the same order in which they were declared. A constructor should not rely on the order of the declarations of data members for initialization; specifically, a constructor should not use the value of one initialized member to initialize another member without a compelling reason.

> **Note:** Data members are initialized in the order in which they are declared, regardless of the order in which they appear in the initialization list. Using one data member to initialize another can cause hard-to-detect bugs should the members subsequently be reordered.

8 Component-Level Documentation

Component-level documentation appears in the header file, immediately following the prologue (Rule 4.1.2). Component-level documentation should describe the component in sufficient detail to make its purpose, functionality, and usage clear.

8.1 Comment Fields

8.1.1

Component-level documentation is organized as a sequence of tagged comment fields. The following comment tags must be used to introduce each of the comment fields:

`//@PURPOSE:`

`//@CLASSES:`

`//@MACROS:`

`//@SEE_ALSO:`

`//@DESCRIPTION:`

All fields other than `@SEE_ALSO` and `@MACROS` are required, and must appear in the order shown above.

Example:

```
//@PURPOSE: . . .
//
//@CLASSES:
//  MyClass: . . .
//    . . .
//
//@SEE_ALSO: another_component
//
//@DESCRIPTION: . . .
// . . .
```

8.1.2

Each comment field must be introduced with the indicated tag and formatted as specified in the following rules. Each tag must begin on a new line and, other than for the first tag, must be preceded by a commented blank line (i.e., a line with only a comment prefix).

8.1.3

The `@PURPOSE` field should provide a succinct description of the component as a whole. The purpose must be written as an imperative sentence, with the first letter capitalized and ending with a period. The purpose must fit on a single line. For class-specific nomenclature, see Rule 9.1.

Example:

```
//@PURPOSE: Provide an efficient repository of holiday data.
```

8.1.4

The `@CLASSES` field must provide a list of only the externally visible classes in the component. Each class must be listed in the order in which it is declared in the file. Each item in the list consists of the name of the

class, followed by a colon, followed by a brief description of the class. The description is written as a noun phrase, without capitalization or a final period. Each item in the list must appear on its own line and be indented two spaces.

Example:

```
//@CLASSES:
//  bdlt::Date: value-semantic Gregorian date type consistent with Unix
```

8.1.5

The @MACROS field is used only when a major feature of the component is used via macros. The format of a @MACROS field mirrors that of the @CLASSES field, where the macro name is given in the place of the class name. The presence of a @MACROS field does not obviate the need of the @CLASSES field; the @CLASSES field is required even if it is empty.

Example:

```
//@CLASSES:
//
//@MACROS:
//  BSLMF_ASSERT: compile-time assert macro
```

8.1.6

The @SEE_ALSO field may be used to provide the names of related components. The component names should be given in alphabetical order and separated by commas. The field may span multiple lines, but subsequent lines must be indented by two spaces.

Example:

```
//@SEE_ALSO: bdlt_date, bdlt_packedcalendar
```

8.2 Description

8.2.1

The @DESCRIPTION field must provide an overview of the functionality of the component as a whole, with sufficient detail to explain when and how the component would best be used. The description need not be exhaustive. The text of the description must begin on the same line as the tag, following the colon and a single space. Subsequent lines are indented one space. Additional paragraphs are separated by a commented blank line.

Example:

```
//@DESCRIPTION: This component provides a single, simply constrained
// attribute class, 'baltzo::LocalTimeDescriptor', that is used to
// characterize subsets of local time values within time zones. Note
// that this class is consistent with the "local-time types" found in
// the "ZoneInfo" representation of a time zone.
```

Note: The description is, to a certain extent, introductory, and serves to aid the user in selecting and understanding the component in the context of the user's own needs. The full contract is described at the function level.

8.2.2

The description field may have subsections that describe specific features of the component or specific aspects of the problem domain addressed by the component. The author should add subsections as needed for clarity. Subsection headings should be indicated using the markup described in Rule 12.3.1.

Example: Possible subsection headings might be as general as `Performance` or `Thread Safety` , or specific to the particular component, such as `Allocators Versus Pools`.

8.3 Usage

8.3.1

A description must include a usage subsection. The usage subsection must be the last subsection in the description. The usage subsection should provide example code fragments and output to illustrate typical usage and behavior of the component.

8.3.2

The paragraph following the usage subsection header must read as follows:

```
// This section illustrates intended use of this component.
```

8.3.3

Each usage example must be given in a separate sub-subsection of the usage subsection. Each example heading must begin with `Example N`, where N is the example number, starting at 1. The example number must be followed by a colon, then the title of the example.

Example:

```
///Usage
///-----
// This section illustrates intended use of this component.
//
///Example 1: Converting Between UTC and Local Times
///- - - - - - - - - - - - - - - - - - - - - - - - - -
// When using the "ZoneInfo" database, we want to . . .
```

Note: See Rule 12.3.1 for the details of creating sub-headings.

9 Class-Level Documentation

9.1

Class-level documentation appears at the beginning of a class definition. The description should

a. classify the class

b. concisely state the purpose of the class

c. describe the essential features of the interaction of data and functions

d. document class invariants

e. specify guarantees concerning thread, alias and exception safety

Note that this description should not be overly repetitive of the component-level documentation, though some redundancy is to be expected. Although the description is specific to the class, some general rules that apply to all class descriptions should be observed.

Example:

```
class Calendar {
    // This class implements a runtime-efficient, fully
    // value-semantic repository of weekend and holiday information
    // over a *valid* *range* of dates. This valid range ...
```

9.2

Class-level documentation must use standard phrasing where applicable. Standard phrasing applies to

- class nomenclature (Section 9.1.0)
- thread safety (Section 9.2.0)
- exception safety (Section 9.3)
- alias safety (Section 9.4)

The documentation for thread safety, exception safety, and alias safety, where relevant, should be included in the class-level documentation. If essential to the purpose of the component, it may be put in the component-level documentation instead. Where specific to a particular function, it should be put in the function-level documentation instead.

9.1 Class Nomenclature

9.1.1

A class may be categorized according to the nomenclature defined in this section. Where applicable, the standard nomenclature must be used in documentation. In addition, certain class categories have additional category-specific design and descriptive requirements.

Note: The use of a standard nomenclature ensures uniformity and implicitly refers to a well-understood meaning which may not be fully contained in the class-level documentation itself.

9.1.2

A class may be categorized using one of the following terms:

a. A *value-semantic type* is a type used to represent a value. This value is reflected in the *salient attributes* of the class and is defined operationally by the `operator==` function. A

value-semantic type may have other members (e.g., an allocator, a capacity) that do not contribute to the value it represents. Value-semantic types are further divided into those that can be externalized (*full* value semantics) and those that cannot (*in-core* value semantics). All native C types (e.g., `int` and `double`) are considered value-semantic types.

b. An *enumeration* is a value-semantic type that provides a categorization over another type.

c. An *attribute* class is a value-semantic type that is comprised solely of data members of other value-semantic types (including basic vocabulary types). Each data member is known as an *attribute*, and is given a name that is used formulaically in the naming of data members, constructor parameters, accessors, and manipulators. A *simply-constrained* attribute class is one in which the constraints of each attribute (if they exist at all) are independent of each other. A *complex-constrained* attribute class is one in which valid values of one attribute are dependent on the current value of another attribute.

d. A *container* is a value-semantic type that aggregates other types. Implementers of containers should take special note of the rules concerning alias safety (Section 9.4) and exception safety (Section 9.3), and note that there is standard phrasing for iterator `operator==` and `operator!=`.

e. A *utility* class provides a namespace for logically related `static` methods that might otherwise be non-`operator` free functions. A utility class has no instance data members, defines neither constructors nor destructor, and is implemented as a `struct`.

   When the methods all serve to manipulate some principal type (defined in another component), the name of a utility class should be formed by suffixing the name of the principal type manipulated by the utility class with `Util`. A utility class that is used to implement a type, rather than manipulate it, is called an *implementation utility* and should use the suffix `ImpUtil`.

f. A *mechanism* describes a stateful object that does not represent a value (e.g., a socket, an allocator, or a thread pool).

g. A *protocol* is a pure abstract class a class that contains only pure virtual functions, except for the destructor. The destructor must be defined and must not be defined `inline` to force the virtual table to be defined uniquely.

h. An *adapter* provides a concrete implementation of a protocol

i. A *wrapper* provides a C-language interface to some or all of the functionality of some other component. For example, the `z_ball_administration` component provides a C-language function for each method of the `ball::Administration` class.

9.2 Thread Safety

9.2.1

The use of a class in a multi-threaded environment must be considered. Without an explicit statement to the contrary, a client may assume that

- *different* objects of the same class may be manipulated concurrently from *different* threads
- the same object may be accessed concurrently via `const` methods from different threads

Classes that meet these basic criteria are said to be `const` thread-safe. If a class does not meet these criteria, or if a class offers additional support for a multi-threaded environment, differences must be documented. In particular, a class with explicitly documented support for concurrency is said to be *thread aware*; a class in which *any* non-creator method may be called concurrently on the same object is said to be *fully thread-safe*; and a class which requires a multi-threaded environment in order to be used is said to be *thread-enabled*.

9.3 Exception Safety

9.3.1

Any class whose arguments may throw exceptions when used should state their exception-safety guarantees. The following phrasing should be used:

For a class that is exception-neutral,

```
// This class is *exception* *neutral* with no guarantee of rollback:
// If an exception is thrown during the invocation of a method on a
// pre-existing object, the object is left in a valid state, but its
// value is undefined. In no event is memory leaked.
```

For a class that is exception-safe,

```
// This class is *exception* *safe* with guaranteed rollback: if
// an exception is thrown during the invocation of a method on a
// pre-existing instance, the object state is left unaffected. In
// no event is memory leaked.
```

The phrasing can be included in the class-level documentation, or may be specified on a method-by-method basis as appropriate.

9.4 Alias Safety

9.4.1

Aliasing refers to the phenomenon wherein there is more than one reference to the same object through different names. Aliasing generally becomes an issue when both a constant input argument and a modifiable argument refer to the same object. The classic example is when an object is assigned to itself, and the input argument becomes an alias for `*this`.

9.4.2

The assignment operator, where defined, must be alias-safe. This should not be included in the contract.

9.4.3

Methods other than the assignment operator may, but are not required to be, alias-safe. If they are, however, this should be noted in the function-level documentation. If the class as a whole is designed for alias safety, such as for a container class, the following phrasing should be included with the class-level documentation:

```
// Finally, *aliasing* (e.g. using all or part of an object as both
// source and destination) is supported in all cases.
```

10 Function-Level Documentation

Function-level documentation is specified in a comment block immediately following a function signature and indented one level. Function-level documentation for certain functions must follow standard phrasing; all other functions must follow the documentation rules specified in this section.

A function specification consists of five parts, organized in the following sequence:

a. What the function **does** (Section 10.1)

b. What the function **returns** (Section 10.2)

c. More about **essential** behavior (Section 10.3)

d. Conditions leading to **undefined** behavior (Section 10.4)

e. **Note that** (more about usage) (Section 10.5)

The function specification should take the form of a single paragraph. At least one of the first two parts must be present.

10.1 What the Function Does

10.1.1

The first part of a function specification, *what the function does*, must state the primary action(s) of the function. It should be specified in a single imperative sentence. This sentence should introduce each required parameter, and describe how the parameters are used and/or altered during the execution of the function.

Example:

```
void append(double item);
    // Append to the end of this array the value of the specified
    // 'item'.

template <class t_TYPE>
void replace(int dstIndex, const t_TYPE& item);
    // Replace the value at the specified 'dstIndex' in this array with
    // the value of the specified 'item' of the parameterized 'TYPE'.
```

10.1.2

The first reference to each parameter in this part must introduce the parameter with the phrase "the specified" appearing before the parameter name. If a list of parameters is introduced at once, the phrase need appear before only the first parameter name. Subsequent references to a parameter may simply name the parameter.

Example:

```
void setYearDay(int year, int dayOfYear);
    // Set the value of this date to the specified 'year' and 'dayOfYear'
    // The behavior is undefined unless 'year' and 'dayOfYear' represent
    // a valid 'bdlt::Date' value (see 'isValid').
```

10.1.3

An optional parameter should be introduced in a new sentence with the phrase "Optionally specify ...", and then the parameter name itself (in grammatical fashion). Multiple optional parameters may be introduced in the same or in separate sentences.

10.1.4

A constructor of a class that takes an optional allocator must use the following phrasing:

```
// Optionally specify a 'basicAllocator' used to supply memory.
// If 'basicAllocator' is 0, the currently installed default
// allocator is used.
```

10.1.5

A constructor must always begin the first sentence of its function specification with the following standard phrasing:

```
// Create a 'MyClass' object ...
```

Example:

```
baltzo::LocalTimeDescriptor(
                          int                      utcOffsetInSeconds,
                          bool                     dstInEffectFlag,
                          const bslstl::StringRef&  description,
                          bslma::Allocator         *basicAllocator = 0);
    // Create a 'baltzo::LocalTimeDescriptor' object having the
    // specified 'utcOffsetInSeconds', 'dstInEffectFlag', and
    // 'description' attribute values. Optionally specify a
    // 'basicAllocator' used to supply memory. If 'basicAllocator'
    // is 0, the currently installed default allocator is used. The
    // behavior is undefined unless
    // '-86339 <= utcOffsetInSeconds <= 86399'.
```

10.1.6

A copy constructor for a value-semantic type must use the following standard phrasing for its first (and possibly only) sentence:

```
// Create a '*MyClass*' object having the same value as the specified
// 'original' object.
```

10.1.7

A destructor must use the following standard phrasing for its first (and possibly only) sentence:

```
// Destroy this object.
```

Note: The comment is there purely to indicate the absence of anything interesting to report.

10.2 What the Function Returns

10.2.1

The second part of a function specification, *what the function returns*, describes the return value of the function. This part should be specified in a single imperative sentence or phrase beginning with the word "return". When descriptions are brief, this part may be combined with the first part in a single sentence.

Example:

```
int connect(const ADDRESS_TYPE& address) BSLS_KEYWORD_OVERRIDE;
    // Initiate a connection to a peer process at the specified
    // 'address' of the parameterized 'ADDRESS_TYPE'. Return 0
    // on success, and a non-zero value otherwise.

Date& operator+=(int numDays);
    // Increase the value of this object by the specified 'numDays',
    // and return a reference to this modifiable date ...
```

10.2.2

In some cases, the first part of the function specification may be omitted, where what the function returns fully specifies what the function does (e.g., for accessors and many free operators).

Example:

```
int month() const;
    // Return the month of this object as an integer in the
    // range [1..12].
```

10.2.3

When a function returns a result through an output argument, it should specify the value returned in an imperative sentence or phrase beginning with the word "load".

Example:

```
void loadData(bsl::vector<double> *result);
    // Load the accumulated data into the specified 'result'.
```

10.3 More About Essential Behavior

10.3.1

The third part of a functional specification, *more about essential behavior*, provides information about collateral effects and other consequences that are essential to the intended functionality.

Example:

```
const double *data() const;
    // . . .
    // The address returned remains valid until this array is
    // destroyed or modified (i.e., the current capacity is not
    // exceeded).

int connect(const ADDRESSTYPE& address) BSLS_KEYWORD_OVERRIDE;
    // . . .
    // If this socket is in blocking mode, the call waits until
    // a connection is established or an error occurs.
```

10.4 Undefined Behavior

10.4.1

The fourth part of a functional specification, *undefined behavior*, specifies the conditions for which the behavior is undefined.

10.4.2

The "undefined behavior" part should be introduced with the phrase: "The behavior is undefined unless ...", followed by a list of conditions for which the behavior is undefined. If using "unless" would lead to a double negative proposition, use the alternative introductory phrase "The behavior is undefined if ..." instead.

Example:

```
void setYearDay(int year, int dayOfYear);
    // . . .
    // The behavior is undefined unless 'year' and 'dayOfYear'
    // represent a valid 'bdlt::Date' value (see 'isValid').

bdlt::Date& operator++();
    // Increase the value of this date by one day, and return a
    // reference to this modifiable date. The behavior is
    // undefined if the initial value of this date was
    // 'bdlt::Date(9999, 12, 31)'.
```

Note: The preference for the phrase "unless" means that binary expressions written in the contract may be transferred directly for use in a BSLS_ASSERT statement in the corresponding function implementation. If the expression follows "if", the transformation of the expression to an assertion is more complex.

10.4.3

A condition for which the behavior is undefined may be expressed as a binary expression of the form:

<*expression*> <*relational operator*> <*argument*>

Example:

```
double sqrt(double value);
    // . . .
    // The behavior is undefined unless '0 <= value'.
```

10.4.4

If a single expression is constrained by a range, a range expression of the form

<*expression*> <*relational operator*> <*argument*> <*relational operator*> <*expression*>

may be used.

10.4.5

Multiple conditions may be presented in a comma-separated list. If multiple conditions are present, they should be presented in the order in which they appear in the parameter list, followed by an appropriately enumerated order for combinations, where combinations are sorted in order of increasing number of parameters involved.

Note: This rule provides a systematic order for testing.

10.4.6

The function-level documentation should not state that the behavior of passing a null pointer for an output argument is undefined. Pointer arguments are assumed to be valid unless otherwise stated.

Note: The exact criteria of whether a pointer is valid are complex. Among such considerations are that the pointer should denote an address that is part of the address space of the process; that, if it is to be dereferenced, that the memory be readable (or writable if the pointer is not `const`); that, if the pointer refers to an object, the pointer value must be suitably aligned; and so on. Rather than stipulating all of these criteria each time, valid pointers should be assumed by both the contract author and client.

10.5 Note That

10.5.1

The final part of a functional specification, *Note that*, provides supplementary information to clarify behavior. These notes shall not be used to modify or add to the contract.

Example: The content of these notes only clarifies the previously described contract, such as to emphasize a non-obvious implication of the contract, to reinforce proper usage, or to give sample code.

```
void removeHoliday(const bdlt::Date& date);
    // . . .
    // Note that this operation has no effect if 'date' is not a
    // holiday in this calendar, even if it is out of range.

int bind(const ADDRESS& address);
    // . . .
    // Note that, in order to receive connections on a socket, it
    // must have an address associated with it, and when a socket
    // is created, it has no associated address.

int waitForAccept(const bdlt::TimeInterval& timeout) BSLS_KEYWORD_OVERRIDE;
    // . . .
    // Note that once a connection request has been received, a call
    // to 'accept' can be made to establish the connection.
```

10.5.2

If there are multiple notes to be included, subsequent notes begin with the phrase "Also note that...", except that the last one may begin with "Finally note that...".

11 Typographical Conventions

11.1 Source File Mechanics

11.1.1

Lines must be terminated only by an ASCII `LF` (`0x0a`) character. Other line termination sequences (e.g., `CRLF`) are not permitted.

> **Note:** End-of-line treatments are a major issue for tools that work across multiple platforms. This rule pertains specifically to code as it resides in the source code repository. It is reasonable (and desirable) for the tools used to check code in and out of the repository to handle end-of-line conversions to a native form, so long as spurious "diffs" are not introduced by the conversions.

11.1.2

The source file must end with a newline character.

11.1.3

No line may be longer than 79 characters (not including the line terminator).

11.1.4

Trailing whitespace on a line is not permitted.

11.1.5

Where indentation is required for formatting, indentation levels must be indicated using four spaces per level. Tab characters must not be used.

11.2 Variable Declarations

11.2.1

When declaring a variable *var* of type "pointer to `TYPE`", the asterisk (`*`) must be written adjacent to the variable name with no intervening whitespace.

Example:

```
const char *buffer = 0;
```

11.2.2

When declaring a variable var of type "reference to `TYPE`", the ampersand (`&`) must be written as a suffix to `TYPE`.

Example:

```
bsl::string& name = names[index];
```

11.2.3

When declaring a variable var of type "reference to non-modifiable `TYPE`", the declaration must be written with `const` in front.

Example:

```
const bsl::string& name = token.id();
```

11.3 Alignment for Variable Declarations

11.3.1

When multiple declarations appear on consecutive lines, the declarations must be spaced such that:

- the types are aligned
- the identifiers, not counting `*`, are aligned
- there is a column of exactly one blank space between the end of the type, including the `&` suffix, if any, and the beginning of the identifier, including the `*` prefix, if any.

Example:

```
bsl::string  var1;
TYPE        *var2;
const TYPE&  var3;
```

Note: In general, the idea is to line up the types and the variable names into columns. This gets more complicated if you extra qualifiers in there, such as `const` or `volatile` . It is also rare. Use your best judgement; make it look aesthetically pleasing, but don't agonize over it.

11.3.2

If a variable declaration must be split across multiple lines because it is too long, then

a. the alignment point for identifiers must be at least two spaces to the right of the alignment point for types, and

b. each declaration must be separated by a blank line

Example:

```
int numberOfNodes;

std::map<bsls::Types::Int64>, std::vector<bsls::Types::Int64> >
    nodeToAdjacentNodesMap;

bsls::Types::Int64
    rootNode;
```

11.4 Variable Documentation

11.4.1

In most cases, documentation for variables is optional. Data members, however, must be documented. Where variables are documented, they must be documented as described in this section. Note that documentation for parameters is part of the function-level documentation (Section 10).

11.4.2

The documentation for a variable should provide a synonym in the form of a noun phrase to indicate the purpose of the entity. The noun phrase should begin with a lowercase letter and never end with a period.

Example:

```
int dutcOffsetInSeconds;  // signed offset from UTC
```

11.4.3

The comment text for a variable should begin on the same line as the declaration. The comment text must be aligned to a point at least two spaces to the right of the alignment point for the variable name, and at

most two spaces to the right of the rightmost semicolon that appears anywhere in the sequence of declarations of which this variable is a part.

11.4.4

If the comment text itself spans multiple lines, all lines must be aligned, and each declaration (including its comment text) must be separated from adjacent items by a blank line.

Example: Either of the following is acceptable:

```
int         d_utcOffsetInSeconds;  // signed offset from UTC

bool        d_dstInEffectFlag;     // 'true' if Daylight Saving Time
                                   // is in effect, and 'false'
                                   // otherwise

bsl::string d_description;         // *non-canonical* identifier for
                                   // this descriptor
```

```
int         d_utcOffsetInSeconds;
                // signed offset from UTC

bool        d_dstInEffectFlag;
                // 'true' if Daylight Savings Time is in effect, and
                // 'false' otherwise

bsl::string d_description;
                // *non-canonical* identifier for this descriptor
```

11.5 Preprocessor Usage

11.5.1

OBSOLETE (Rule 14.3)

Note: Preprocessor `#if` directives may be used (previously they were not permitted).

11.5.2

In general, the definition of preprocessor macros (other than for include guards) is discouraged. Where used, they must follow the rules of this section.

Note: Preprocessor macros introduce names that are not subject to the scoping rules of the language. In most cases, there are features of the language that can be used in place of macros.

11.5.3

The name of a macro must be all uppercase, with each word in the name separated by an underscore.

11.5.4

The name of a macro with broader scope than a function body (other than an include guard) must be prefixed by the component name of the component in which it is defined.

11.5.5

Macro parameters must be all uppercase. Macro parameters should have more than one letter.

11.5.6

A macro may be used within the body of a function for convenience, so long as the macro is defined after the scope of the function body is opened, and un-defined before the scope of the function body is closed. This technique should be used sparingly.

11.5.7

Macros created within a `.cpp` or `.t.cpp`, may be prefixed by `U_` rather than the full component name, to distinguish them from identifiers defined in a header. Such identifiers are never to be `#undef`'ed without being `#define`'d first, so that if the identifier collides with a macro from the included headers, a compiler warning or error will result.

11.6 Mid-Function Returns

11.6.1

All statements that `return` other than at the closing brace of a function must be marked with a `// RETURN` comment, right-justified to the 79th column.

```
inline
int Date::setYearDayIfValid(int year, int dayOfYear)
{
    enum { k_SUCCESS = 0, k_FAILURE = -1 };

    if (isValidYearDay(year, dayOfYear)) {
        setYearDay(year, dayOfYear);
        return k_SUCCESS;                                     // RETURN
    }

    return k_FAILURE;
}
```

11.6.2

If the `return` statement spans several lines, put the comment on the last line.

```
inline
unsigned int HashUtil::hash0(int key, int modulus)
{
    BSLS_REVIEW(0 < modulus);

    if (4 == sizeof(int)) {
        return static_cast<unsigned int>(key)
             % static_cast<unsigned int>(modulus);            // RETURN
    }
    else {
        return (static_cast<unsigned int>(key) & 0xFFFFFFFF)
              % static_cast<unsigned int>(modulus);           // RETURN
    }
}
```

11.6.3

There should be at least two blank spaces between the trailing `;` of the `return` statement and the first `/` of the comment. If there is insufficient space for the comment on the line(s) containing the `return` statement, insert an empty line containing the (right-justified) comment.

```
return day <= s_cachedDaysInMonth[year - s_firstCachedYear][month];
                                                                // RETURN
```

11.7 Formatting Logical Statements

11.7.1

The consequence (body) of each logical statement is *always* a code block -- i.e., code surrounded by braces -- even if it consists of a single statement. The braces are placed according to the K&R style where:

- the opening brace is on the same line as closing parenthesis of the controlling expression, separated by a single space,
- the closing brace on a separate line aligned under the start of the controlling keyword (e.g., `if`, `while`), and
- the enclosed code is indented one level.

Example:

```
// ...

if (predicate) {
    // ...
}

// ...
```

The above description does not apply to `do`/`while` statements. For this control structure, the opening brace is on the same line as the `do` and the closing brace is aligned under the `do` on the same line as the `while`. As in the other cases, the enclosed code is indented one level:

```
// ...

do {
    // ...
} while (predicate);

// ...
```

11.7.2

The `else` keyword of `if`/`else` statements can be written on a line by itself:

```
if (predicate) {
    // ...
}
else {
    // ...
}
```

11.7.3

`for` statements should be written on a single line if there is sufficient room at the current level of indentation:

```
for (intialization; predicate; update) {
    // ...
}
```

However, if there is insufficient room, put each expression on its own line:

Example:

```
 bsl::vector<bdlt::Date> dates;

 // ...

 for (bsl::vector<bdlt::Date>::iterator it   = dates.begin(),
                                        end  = dates.end();
                                        end != it;
                                        ++it) {
     // ...
}
```

Note: Indenting the declaration of `end` to the same position as `it` is preferred (when possible) as an application of the declaration alignment rule. (11.3.1).

11.7.4

`switch` statements are written with each `case` statement indented by two spaces from the `switch` statement. The contents of the `case` are indented by an additional two spaces (from the `case` keyword), a total of four spaces from the `switch` keyword.

Note: This rule is analogous to the previously presented rules for positioning `public:` and `private:` keywords in class definitions (Rule 6.2.1).

Each `case` enclosed in a code block and followed by a `break;` statement.

Example:

```
// ...

switch (version) {
  case 1: {
    bslx::InStreamFunctions::bdexStreamIn(stream, d_timeZoneId, 1);
    bslx::InStreamFunctions::bdexStreamIn(stream, d_datetimeTz, 1);
  } break;
  default: {
    stream.invalidate();  // unrecognized version number
  } break;
}

// ...
```

12 Markup

12.1

Comment text is text included a source file to document some aspect of the component. Comment text follows the single-line C++ comment prefix `//`. Comments using the C++ multi-line notation (/* */) are not permitted. Note that comment text must follow the typographical conventions for all source files (Section 11).

12.2

All comment text must be written using clear and grammatical language, and in complete sentences, except where explicitly noted otherwise.

12.3

Where appropriate, comment text may use the markup syntax described in this section to enhance the appearance of the text.

> **Note:** The mechanical formatting conventions introduced in this section serve the dual purpose of providing well-structured, easy-to-read documentation in the primary ASCII source files and a markup that can be used to render the same text in a professional, multi-font format suitable for publication.

12.4

Comment text is generally written as a series of filled paragraphs. A single commented blank line may be used to separate multiple paragraphs. Hyphenation must not be used to break a word that is too long to fit on a single line.

> **Note:** When rendered in a proportional font, paragraphs may be filled and line breaks may not appear in the same place as in the original text.

12.1 Basic Markup

12.1.1

Bold, italic, and underscored text may be indicated using the delimiters `'!'`, `'*'`, and `'_'`, respectively. Note that individual words must be delimited individually.

> **Example:** The following source
>
> ```
> // Text may be written in a !bold!, *italic*, or _underscored_
> // proportional font, or may appear in a 'fixed' font. *Consecutive*
> // *multiple* *words* must be _delimited_ _individually_.
> ```
>
> would be rendered as
>
> Text may be written in a **bold**, *italic*, or underscored proportional font, or may appear in a `fixed` font. *Consecutive multiple words* must be delimited individually.

12.1.2

All C++-language keywords, expressions, and any user-defined names, such as variables and classes, must be indicated in the original source text by placing such text between single-quote delimiters. Unlike markup for bold, italic, and underscored text, exactly one set of delimiters must enclose the entire C++ expression.

> **Example:**
>
> ```
> // C++ language expressions, such as '*this' or 'struct' or
> // 'UserDefinedTypeNames', are rendered in a fixed-width font.
> ```

would be rendered as

C++ language expressions, such as `*this` or `struct` or `UserDefinedTypeNames`, are rendered in a fixed-width font.

12.1.3

A single C++ expression in comment text must not break across multiple lines of text (but see Rule 12.2.1).

Example: Consider the following:

```
// Remove from this array, beginning at the specified 'index', the
// specified 'numElements' values.  All values with initial indices
// at or above 'index + numElements' are shifted down by 'numElements'
// index positions.  The behavior is undefined unless '0 <= index',
// '0 <= numelements', and 'index + numElements <= length()'.
```

Note: This rule has proven to be a tremendous help in readability (and accuracy) during code reviews and testing. When the rule results in extremely ragged source text, or one cannot fit even a single expression in one line, consider using a display instead.

12.2 Displays

12.2.1

A *display* is one or more lines of text that are intended to be rendered in a fixed-width font, exactly as typed in the source file. A display is indicated in source text by preceding and following the display text with the character sequence `//.` (known as the "no-fill toggle") on its own line. Within a display, text must initially be indented two characters after the leading comment characters.

Example: The following source text

```
// Code excerpts, such as:
//..
//  const double PI = 3.141592653589793238;
//..
// as well as expressions too long to fit on a single line, such
// as the Gregory-Leibniz approximation for 'PI / 4':
//..
//  1.0 - 1.0 / 3 + 1.0 / 5 - 1.0 / 7 + 1.0 / 9 - ...
//..
// may be inserted within a display.
```

would be rendered as

Code excerpts, such as

```
const double PI = 3.141592653589793238;
```

as well as expressions too long to fit on a single line, such as the Gregory-Leibniz approximation for `PI / 4`:

```
1.0 - 1.0 / 3 + 1.0 / 5 - 1.0 / 7 + 1.0 / 9 - ...
```

may be inserted within a display.

12.2.2

A *list block* is a sequence of lines to be displayed as a list. A list block is indicated by beginning each line in the list block with the character sequence `//:`. The indicator is followed by additional markup indicating the type of list. Three kinds of lists may be used:

- *unordered lists*, indicated with a lowercase `'o'` preceded and followed by a blank
- *ordered lists*, indicated by numbering the list items, starting from 1; single-digit numbers should be preceded by a space.
- *definition lists*, indicated by a word or short phrase followed by a colon.

Example: An unordered list

```
// Each protocol class has to satisfy the following set of
// requirements:
//: o The protocol is abstract: no objects of it can be created.
//: o The protocol has no data members.
//: o The protocol has a virtual destructor.
//: o All methods of the protocol are pure virtual.
//: o All methods of the protocol are publicly accessible.
```

An ordered list

```
// The test will consist of the following steps:
//: 1 Push back into 'dlist', then 'dlist2', then 'dlist3'.
//: 2 Repeat #1.
//: 3 Pop front from 'dlist1', then 'dlist2', then 'dlist3'.
```

A definition list

```
//: 'datetimeTz': date, time and offset from UTC of the local time.
//: 'timeZoneId': unique identifier representing the local time zone.
```

12.2.3

When a list item exceeds one line, the text of the additional lines must be indented three spaces. A list with at least one multi-line item must have an empty line between every list item.

Example:

```
// Global assumptions:
//: o All explicit memory allocators are presumed to use the global,
//:   default, or object allocator.
//:
//: o ACCESSOR methods are 'const' thread-safe.
```

12.2.4

An element in a list may include a sub-list. The mark indicating the type of the list should be aligned with the text of the parent item. A sub-list need not be the same type as the parent.

Example:

```
//: GROWTH STRATEGY: geometrically growing chunk size starting
//:   from 1 (in terms of the number of memory blocks per chunk),
//:   or fixed chunk size, specified as either:
//:   o the unique growth strategy for all pools, or
//:   o an array of growth strategies, one corresponding to each pool.
```

12.3 Headings

12.3.1

Subsections within comment text may may be introduced using a two-line sub-heading, as shown:

```
//
///Sub-Heading
///-----------
// *text begins here*
```

Note that a sub-heading must be preceded by a commented blank line, and immediately followed by the descriptive text on the next line. Sub-headings must be written in title case.

12.3.2

Comment text may include additional nested subsections. Each new level of sub-heading uses a form similar to that shown above, except that space characters are added between each dash, one for each level of depth. The rightmost hyphen must be align with the last character in the heading text; additional space may be added at the beginning of the "underline" as necessary.

Example:

```
///Sub-Sub-Heading
///- - - - - - - -
//
///Sub-Sub-Sub-Sub-Heading
/// -  -  -  -  -  -  -  -
```

12.4 Links

A *link* is a textual reference (the *link text*) to some other location in the documentation set (the *link target*).

Note: When component comments are converted to other representations (e.g., `html`) the BDE markups links are typically converted to clickable hyperlinks.

12.4.1

Each *link* must refer to a unique other location the documentation set. The following are valid link targets:

- A (tagged) comment field (see 8.1) in the current file.
- A heading (see 12.3) within the current file.
- The start of documentation of some other component, package, or package group file.
- A (tagged) comment field or heading in some other component, package, or package group file.

12.4.2

The link is denoted by surrounding the link text by a pair of open { and close } braces.

```
// See {Usage} under {DESCRIPTION} above.
```

Notice that link text to a tagged section does *not* include the @ sign.

Link text must exactly match that of the link target, including font markup. For example, if a heading uses font markup (see 12.1), then the link text to that target must also do so.

```
// See {Using 'MyClass' Directly (*Not* *Recommended*)}.
```

Link text can be broken (between words) across two lines.

```
//            ... See {Using 'MyClass' Directly (*Not*
// *Recommended*)}.
```

12.4.3

Link text to the start of component (package, package group) documentation is denoted by giving the entity name surrounded by single ticks.

```
// See {'bdlt_datetime'} for details.  {'bdlt'} provides an overview
// of the different time conventions used in this package.
```

A heading in another component is denoted by the component name, delimited by single tick characters, a pipe symbol (|), followed the heading text.

```
// See {'bdlt_date'|Usage} for related examples.
```

13 Deprecation

13.1 General Meaning

13.1.1

Code that is slated to be removed from a unit of release is said to be *deprecated*. A deprecation notice must be placed in the appropriate code at least one released version prior to removal.

Note: An important, and often overlooked, phase of the software life-cycle is the removal of old software facilities, typically in lieu of improved, replacement facilities. Before software is removed, the user community should receive notice of the upcoming change, and how to migrate their code. A simple and regular markup for noting deprecation status allows such changes to be readily discovered by software tools, and facilitates reports to the user community on the evolving software base.

13.1.2

The appearance of a deprecation notice implies the following:

a. the facility should not be used in any new code

b. the facility may be removed in a future release

c. the software using the now deprecated facility must be changed prior to the removal of the facility

13.1.3

The deprecation notice must inform the user of the replacement for the deprecated facility, if any.

13.1.4

Specific markup must be used to flag deprecated facilities. Software may be deprecated at several different levels of granularity, as detailed in the following section.

13.2 Markup

13.2.1

To deprecate a component, package, or package group, the `@DEPRECATED` comment field must be used in the corresponding documentation, directly after the `@PURPOSE` section (Section 8.1). All of classes and functions of a deprecated component are implicitly deprecated; however, if any of those items had already been deprecated, there is no need to remove their individual deprecation notices. Indeed, it is advantageous to leave those notices in place as they provide useful information and component history.

Example:

```
//@PURPOSE: Provide helper classes for recording metric values.
//
//@DEPRECATED: Use 'balm_metric' instead.
//
//@CLASSES:
//  baem_Metric: container for recording metric values
```

In the (extremely rare) case that there is no successor facility, one may simply state:

```
//@DEPRECATED: Do not use.
```

Sometimes, the replacement facility is distributed in more than one component, package, or package group. For example, `bcefi` package documentation states:

```
//@DEPRECATED: Use 'bcef' and 'bdef' packages instead.
```

13.2.2

To deprecate a class, the class-level documentation must begin with the deprecation notice, in a separate paragraph. Additionally, deprecated classes must be removed from the `@CLASSES` section of the component-level documentation.

Example:

```
class bcemt::TryLockGuard : public bcemt::LockGuardTryLock<T>
    // !DEPRECATED!: Use 'bcemt::LockGuardTryLock' instead.
    //
    // This class implements a scoped guard that, on construction, ...
```

```
class baedb_UserDb
    // !DEPRECATED!: Use types from 'bsiudb_userdb' instead.
    //
    // This class provides a protocol (or pure interface) for
    // retrieving ...
```

13.2.3

To deprecate a function, the function-level documentation must begin with the deprecation notice, in a separate paragraph.

Example:

```
static int maxSupportedVersion();
    // !DEPRECATED!: Use 'maxSupportedBdexVersion' instead.
    //
    // Return the most current 'bdex' streaming version number
    // supported by this class. (See the package-group-level
    // documentation for more information on 'bdex' streaming of
    // container types.)
```

Sometimes, the deprecation notice cannot be meaningfully expressed in a single line. If so, multiple lines may be used as in this example of one overloaded function deprecated in lieu of another:

```
template <class t_TARGETTYPE, class t_FACTORY>
void load(t_TARGETTYPE  *ptr,
          t_FACTORY     *factory,
          DeleterFunc    deleter);
    // !DEPRECATED!: Instead, use:
    //..
    //  template <class t_TARGETTYPE>
    //  void load(t_TARGETTYPE  *ptr,
    //            void          *factory,
    //            DeleterFunc    deleter);
```

```
    //..
    // Destroy the current managed object (if any) and reinitialize
    // this managed pointer to manage the specified 'ptr' using
    // ...
```

13.2.4

To deprecate an enumerator, the deprecation notice must appear in the enumerator documentation (Rule 6.4.4).

Example:

```
enum Level {
    e_OFF   = 0,      // disable generation of corresponding message
    // ...
    e_TRACE = 192,    // execution trace data
    BAEL_NONE  = 224  // !DEPRECATED!: Do not use.
};
```

14 Appendix 1: Obsolete Rules

This appendix contains the references to the obsolete rules.

14.1

OBSOLETE

Each include directive must have the following form:

```
#ifndef INCLUDED_PACKAGE_COMPONENT
#include <package_component.h>
#endif
```

where *package_component* is replaced with the full component name whose header is to be included. In a sequence of such directives, each block must be separated by a blank line.

14.2

OBSOLETE

When specifying an include directive for a header that is not a component, one must define the include guard explicitly after the header is included, as in the following:

```
#ifndef INCLUDED_PACKAGE_COMPONENT
#include <package_component.h>
#define INCLUDED_PACKAGE_COMPONENT
#endif
```

14.3

OBSOLETE

Preprocessor `#if` directives must not be used.

15 Appendix 2: Rationale

R.3.4 Rationale: Template Parameters

R.3.4.4 Rationale: 3.4.4

The previous version of this rule required upper snake case without the leading `t_`. This convention occasionally resulted in confusing compilation errors when those names happened to collide with macros defined by either the platform or a third-party library. This was a recurring source of issues with releasing BDE software, breaking multiple promotion attempts in 2022 alone. For example, the type template parameter name `OTHER_TYPE`, frequently used for converting constructor templates, collides with a macro defined by the `<net/if.h>` header on the AIX operating system, and thus results in compilation errors on that platform only.

The unusual decision to prefix upper snake case names with a lowercase letter simultaneously achieves two objectives: it minimizes the risk of collision with macros, while making it clear to readers that the entity so named is a template parameter rather than any other kind of entity.

3.4.4

R.4.4 Rationale: Using Includes Rather Than Forward Declarations

R.4.4.3 Rationale: 4.4.3

This standard previously recommended using forward declarations where possible, since forward declarations enable the use of a type name within a translation unit without requiring the compiler to include the header with that type's definition. By avoiding the need to include the header with the definition, the use of a forward declaration can avoid having to recompile when the header changes, thus reducing compile-time coupling and improving compile times when recompilation is necessary.

However, a forward declaration for a type can also impede moving that type to a new namespace or renaming it, a very common type of refactoring. The larger the organization, and the more code using forward declarations, the harder it becomes to migrate and rename types that have been forward declared (making this impediment particularly relevant to Bloomberg).

To understand why forward class declarations impede code migration, imagine a forward declaration in the library `mylib`:

```
// mylib_foo.h                                                -*-C++-*-

namespace calendarlib { class Time; }
```

Now imagine the owner of calendarlib wants to migrate the `Time` type to a new library (`timelib`), and are planning to give it a new name, `PreciseTime` (to distinguish it from a lower precision time-type). To facilitate the migration without breaking existing users of `calendarlib::Time`, the owners of the legacy `Time` type might update the existing (deprecated) header for the type `Time` as follows:

```
// calendarlib_time.h                                         -*-C++-*-

#include <timelib_precisetime.h>

namespace calendarlib {
[[deprecated]] typedef timelib::PreciseTime Time;
}
```

Unfortunately, the old forward declaration in `mylib_foo.h` is now in conflict with the newly refactored code. This inconsistency will result in a compile-time error in either `mylib` itself or in code using `mylib`.

Within the context of Bloomberg, this sort of forward declaration would require the owner of `timelib::Time` to coordinate with each owner of code forward declaring `calendarlib::Time` to update their code prior to performing the refactoring (and potentially a second pass, to recreate the forward declaration, after the release of `PreciseTime` is completed).

Exceptions

Forward declarations are not allowed across unit of release (UOR) boundaries. Within a UOR, `#include` statements should still be preferred, but using forward declarations is still allowed (e.g., to resolve a cyclic dependency).

Note that in future versions of C++, once modules are adopted, it will no longer be permitted for a module's interface to forward declare types from outside the module itself, so using includes is a form of future-proofing.

Impact of using includes

Benchmark performance testing has shown that the switch between forward declares and includes has no real impact on the build time of our libraries or the build of our downstream client code at our enterprise scale on any of our production platforms at Bloomberg.

R.5.6.1 Rationale: 5.6.1

The requirement to append the `// for testing only` labels serve to remind developers that such additions may adversely effect the reuse of the component *and* explain their presence to future developers (or automated tools) lest those directives be deemed superfluous as they are not required by the component itself.

By adding testing-only dependencies to those of the component itself, we ensure that such dependencies do not introduce cycles. Avoiding any test-driver introduced cycles enables a bottom-up testing strategy such that tests for a component can be written in terms of lower-level components that have been tested independently.

5.6.1

R.7.3 Rationale: Function Signatures

***R.7.3.6 Rationale:* 7.3.6**

C++11 introduced the `override` keyword to catch developer mistakes when overloading virtual functions. This section explains the purpose and use of the `override` keyword, enumerates the benefits and drawbacks of its use, and provides rationale the use of `override` in Bloomberg C++ code.

The `override` keyword

The *contextual keyword* [14] `override` can be provided at the end of a member-function declaration to ensure that the decorated function is indeed *overriding* a corresponding `virtual` member function in a base class (i.e., not *hiding* [15] it or otherwise inadvertently introducing a distinct function declaration). Use of this feature express such deliberate intent so that (1) human readers are aware of it and (2) compilers can validate it.

Example

Consider the following polymorphic hierarchy of error-category classes, as we might have defined them using C++03:

```
struct ErrorCategory {
    virtual bool equivalent(const ErrorCode& code, int condition);
```

```
    virtual bool equivalent(int code, const ErrorCondition& condition);
};

struct AutomotiveErrorCategory : ErrorCategory {
    virtual bool equivalent(const ErrorCode& code, int condition);
    virtual bool equivolent(int code, const ErrorCondition& condition);
};
```

Notice that there is a defect in the example above, where `equivalent` has been misspelled, that was not caught by the compiler. Clients calling `equivalent` on `AutomotiveErrorCategory` will incorrectly invoke the base class function. If the function in the base class happens to be defined (i.e., not pure `virtual`) the code may compile and behave unexpectedly at runtime. Moreover, suppose that, over time, the interface is changed by marking the equivalence-checking function const to bring the interface closer to that of `std::error_category`:

```
struct ErrorCategory {
    virtual bool equivalent(const ErrorCode& code, int condition) const;
    virtual bool equivalent(int code, const ErrorCondition& condition) const;
};
```

Without applying the corresponding modification to all classes deriving from `ErrorCategory`, the semantics of the program change due to the derived classes hiding (instead of overriding) the base class's `virtual` member function. Both the errors discussed above would be detected automatically by decorating the `virtual` functions with the `override` keyword:

```
struct AutomotiveErrorCategory : ErrorCategory {
    bool equivalent(const ErrorCode& code, int condition) override;
        // Compile-time error when base class changed.
    bool equivalont(int code, const ErrorCondition& code) override;
        // Compile-time error when first written.
};
```

In addition to leveraging the compiler to ensure that both overloads of `equivalent` indeed override their corresponding virtual functions in the base class, `override` also serves as a clear indication to the human reader of the derived-class author's intent to customize the behavior of `ErrorCategory`. For any given member function, use of `override` necessarily renders any use of `virtual` for that function syntactically and semantically redundant: The only (cosmetic) reason for retaining `virtual` in the presence of override would be that `virtual` appears to the left of the function declaration (as it always has) instead of all the way to the right (as override does now).

Potential pitfalls

Although not a problem with wide-spread use of the feature per se, a codebase that uses override inconsistently (or one that has just started adopting it) might lull developers into mistakenly thinking that such signature changes made across an inheritance hierarchy are guaranteed by the compiler to be exhaustive. For example, altering the signature of a `virtual` member function in a base class and then compiling "the world" will result in errors whenever `override` was used on a corresponding function in the derived classes. After fixing all the errors, the author of the changes might mistakenly believe that all needed changes have been guaranteed by the compiler to have occurred. There might, however, still exist derived-classes implementations for which `override` has not yet been added, and which might now

behave differently at run-time. Static-analysis tools that check for the mandatory use of override (where applicable) would help to address this issue.

Recommendation

Since the overall benefits of wide-spread use of `override` at Bloomberg almost certainly outweigh the (minor) potential pitfall of assuming the compiler will catch every last defect, we recommend always decorating overriding functions with `override`:

```
struct AutomotiveErrorCategory : ErrorCategory {
    bool equivalent(const ErrorCode& code, int condition) const override;
};
```

For code that needs to be compiled in C++03 mode, use `BSLS_KEYWORD_OVERRIDE` instead:

```
struct AutomotiveErrorCategory : ErrorCategory {
    bool equivalent(const ErrorCode& code, int condition) const
                                                  BSLS_KEYWORD_OVERRIDE;
};
```

Furthermore, since in the presence of `override` prefixing the overriding function declarations with `virtual` neither provides additional documentation for human consumption nor prevents bugs when the signature of `virtual` function changes, we recommend omitting the `virtual` keyword from declarations of overriding functions.

7.3.6

---

14 A *"contextual keyword"* is a special identifier that acts like a keyword when used in particular contexts. override is an example as it can be used as a regular identifier outside of member-function declarators.

15 Function-name *hiding* occurs when a member function in a derived class has the same name as one in the base class, but it is not overriding it due to a difference in the function *signature* or because the member function in the base class is not `virtual`. The hidden member function via (a pointer or reference to) a base class will *not* participate in dynamic dispatch; the member function of the base class will be invoked instead. The same code would have invoked the derived class's implementation had the member function of the base class had been *overridden* rather than *hidden*.