BDE Libraries: An Orientation

Steven Breitstein
sbreitstein@bloomberg.net
Bloomberg LP

Thursday, September 22, 2016

CppCon2016, Bellevue, Washington, USA

Introduction

What is BDE?

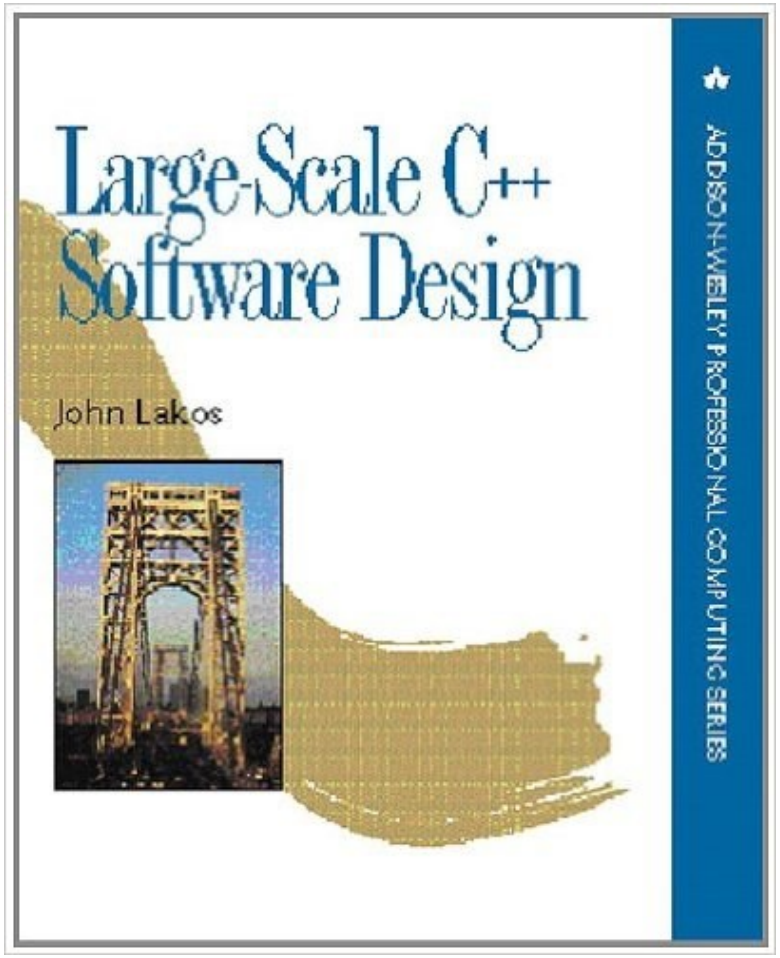
Bloomberg **D**evelopment **E**nvironment (BDE)

- Sometimes**, B**asic **D**evelopment **E**nvironment
- Even, **B**DE **D**evelopment **E**nvironment (recursive)

The term, *BDE*, applies to:

- Our suite of low-level libraries, the foundation of much critical infrastructure at Bloomberg LP
- The Bloomberg team who creates/maintains those libraries
- The methodology we follow
  - Inspired by John Lakos

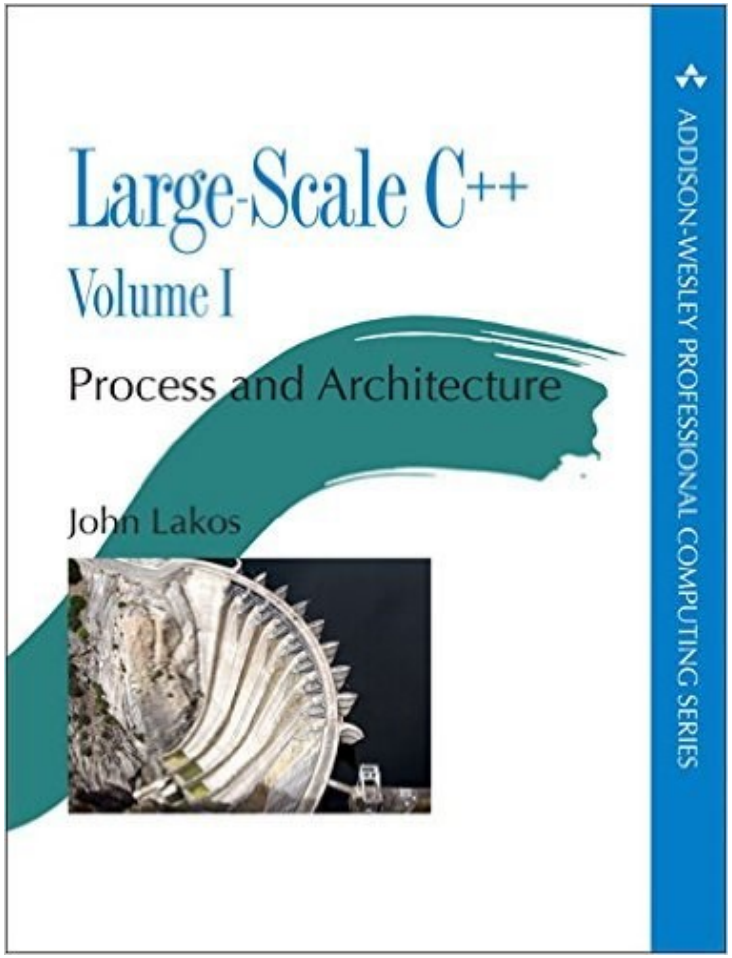
Background: Lakos ‘96



Background: Lakos Talks

Event	Title
LiveLessons	Applied Hierarchical Reuse Using Bloomberg's Foundation Libraries
CppCon 2014	Defensive Programming Done Right.
CppCon 2015	Value Semantics: It ain’t about the syntax!
CppCon 2016	Advanced Levelization Techniques

Background: Lakos ‘17



BDE Libraries

- *Foundation classes* for development throughout Bloomberg
  - For building infrastructure
  - Found at: **`https://github.com/bloomberg/bde`**
  - Over 900 *documented* and *tested* components
    - Organized into 47 packages, which are
    - Organized into 5 package groups (libraries == UOR)

BDE Libraries

- Address needs best solved once for entire enterprise, e.g.:
  - Date/time/timezone/calendar types and utilities
  - Logging and metrics
  - Command-line parsing, tokenization
  - Thread and event management
  - Data marshalling and networking

BDE Libraries

- Broad Features
  - Many areas of application
  - Levelized, component-based code
  - Taxonomy of class categories
  - Abundant, detailed, consistent documentation
  - Narrow contracts
  - Defensive Programming
  - Runtime polymorphic memory allocation
  - Thorough test drivers (published)

What is a Component?

Example:

- Component, **`bdlt_date`**
- Defines, class **`Date`**
- In namespace **`bdlt`**

Actually **`BloombergLP::bdlt::Date`**

- Consists of files
  - **`bdlt_date.h`**
  - **`bdlt_date.cpp`**
- Has test driver **`bdlt_date.t.cpp`**

What is a Component?

A **`.h / .cpp`** file pair with common base name such that:

1. The **`.cpp`** file includes the **`.h`** as the first substantive line.
2. Everything defined in the component to have external linkage is declared in the **`.h`** file.
3. Everything declared in the **`.h`** (if defined anywhere) is defined in the component.
4. Clients **`#include`** the **`.h`** file.

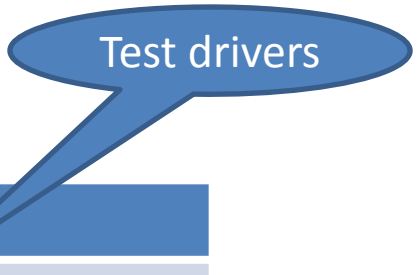
Design Rules:

1. No cyclic dependencies (per **`#include`**'s).
2. Any friendships stop at component borders.

What is a Package?

- Example, package **`bdlt`** contains:

Component	Constituent Files
`bdlt_date`	`bdlt_date.{h,cpp,t.cpp}`
`bdlt_datetime`	`bdlt_datetime.{h,cpp,t.cpp}`
`bdlt_dateutil`	`bdlt_dateutil.{h,cpp,t.cpp}`
`bdlt_time`	`bdlt_time.{h,cpp,t.cpp}`
et al.	



- Having no cyclic dependencies among themselves
- Defining symbols in namespace **`bdlt`**
  - E.g., **`bdlt::Date`**, **`bdlt::Datetime`**, **`bdlt::Time`**
- Source directory **`groups/bdl/bdlt`**

What is a Package?

- A set of *components* logically related by functionality *and* having a common envelope of dependencies.
- Having no cyclic dependencies among the components.
- Having names with a common prefix sequence – the package name – separated by an underscore.
- Defining their symbols in a namespace matching the package name.

What is a Package Group?

- Example, package group **`bdl`** contains:

Package	Description
`bdlb`	Utilities on basic types
`bdlc`	Specialized containers
`bdlmt`	Thread pools and event schedulers
`bdlt`	Date and time vocabulary types, and utilities
et al.	

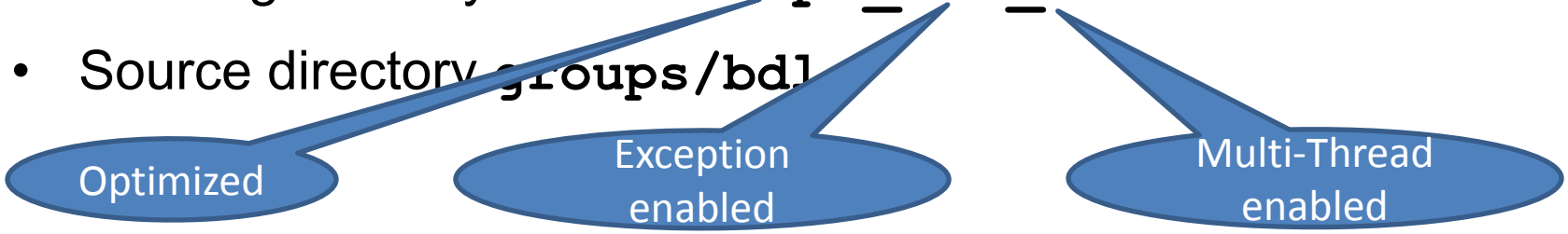
- Having no cyclic dependencies among themselves
- Defining a library: **`-lbdl.opt_exc_mt`**
- Source directory **`groups/bdl`**

What is a Package Group?

- Example, package group **`bdl`** contains:

Package	Description
`bdlb`	Utilities on basic types
`bdlc`	Specialized containers
`bdlmt`	Thread pools and event schedulers
`bdlt`	Date and time vocabulary types, and utilities
et al.	

- Having no cyclic dependencies among themselves
- Defining a library: **`-lbdl.opt_exc_mt`**
- Source directory **`groups/bdl`**



What is a Package Group?

- A set of *packages* having a common envelope of dependencies and, often, logically related functonality.
- Having no cyclic dependencies among the packages.
- Member package names have a common three letter prefix in *all* package names.
  - Note that when naming a package, the intended package group must be known.
- Defines a unit-of-release, a library.

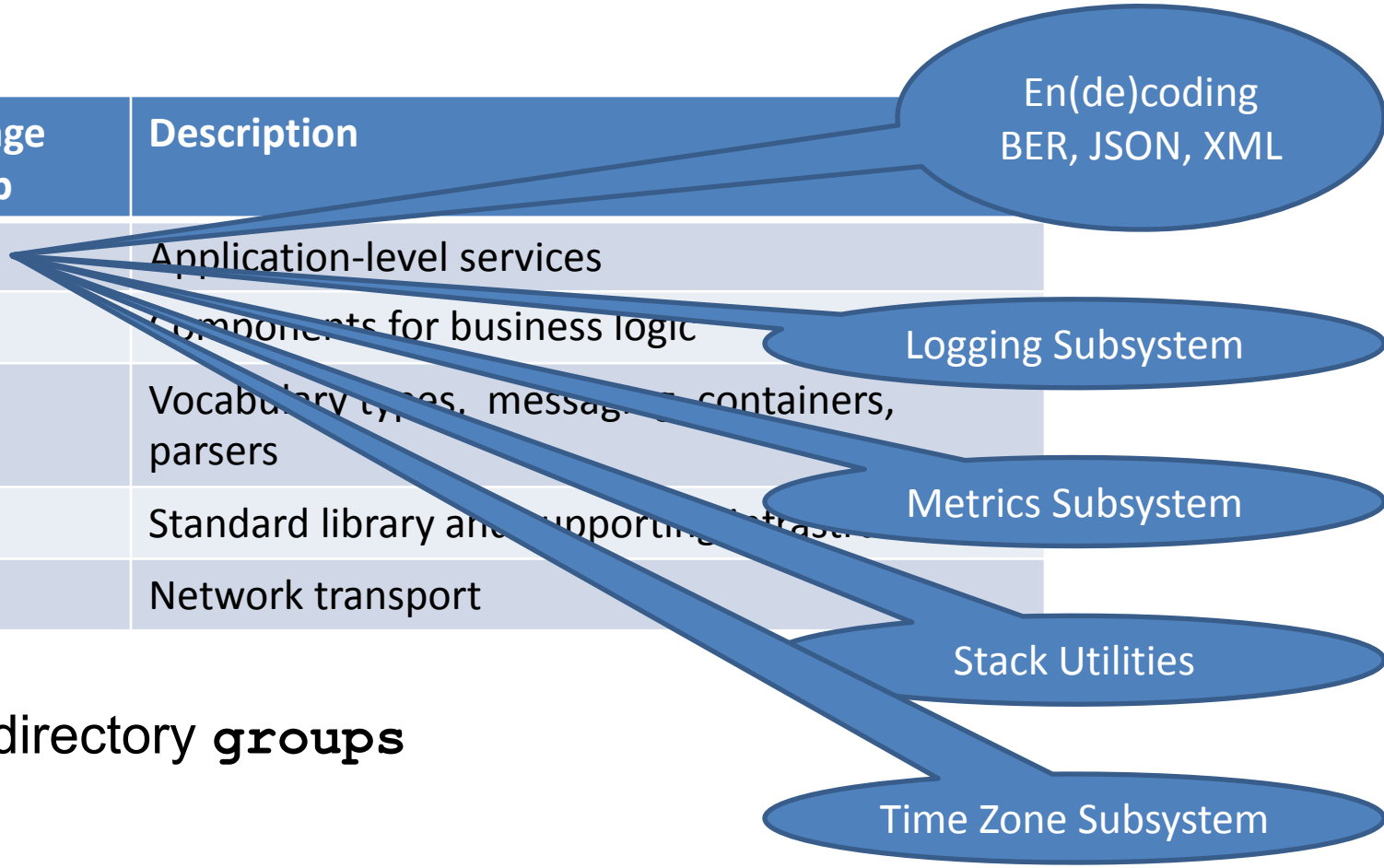
The Package Groups

Package Group	Description
`bal`	Application-level services
`bbl`	Components for business logic
`bdl`	Vocabulary types, messaging, containers, parsers
`bsl`	Standard library and supporting infrastructure
`btl`	Network transport

- Source directory **`groups`**

The Package Groups

Package Group	Description
bal	Application-level services
bbl	Components for business logic
bdl	Vocabulary types, messaging, containers, parsers
bsl	Standard library and supporting infrastructure
btl	Network transport



- Source directory **`groups`**

The Package Groups

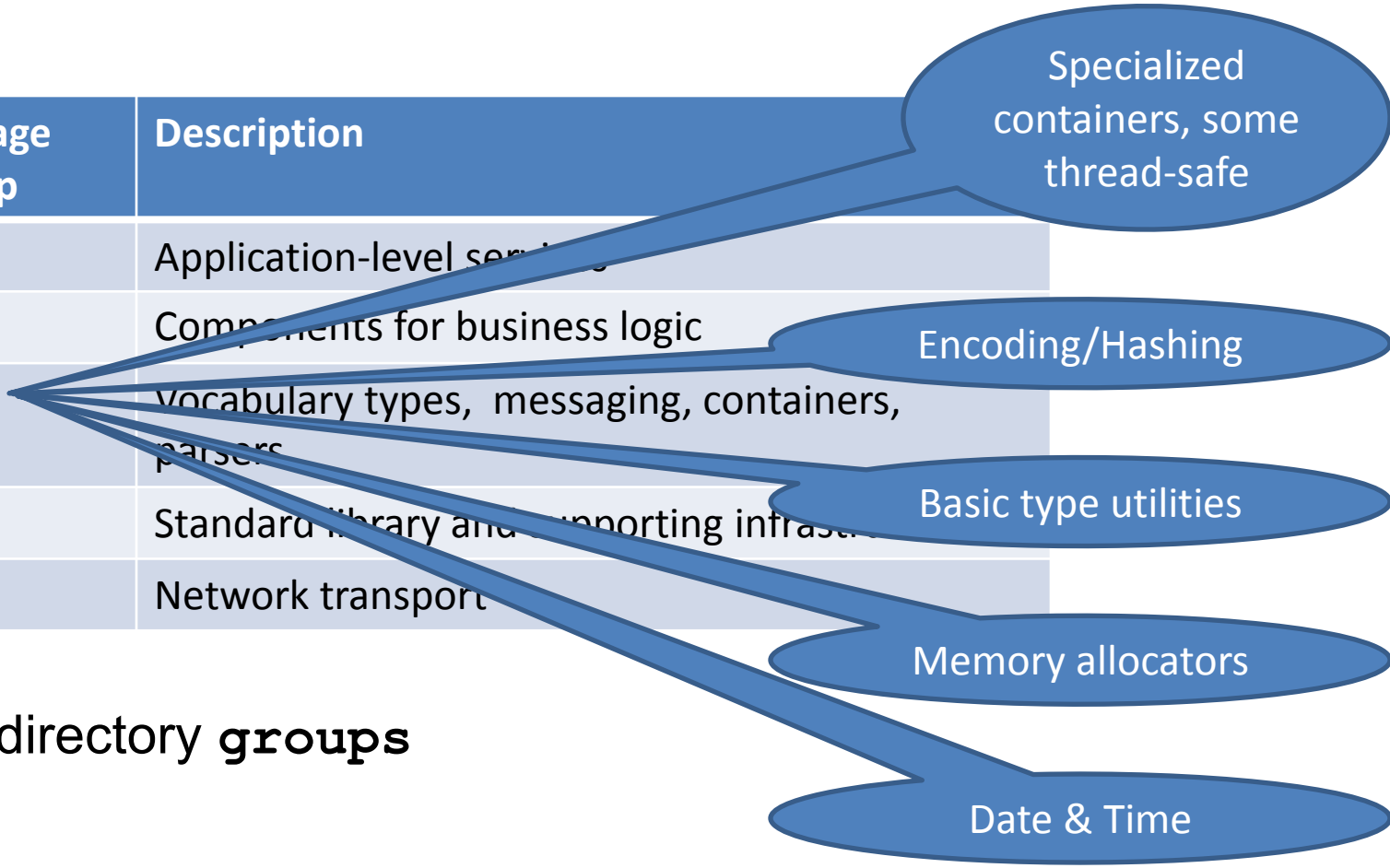
Package Group	Description
bal	Application level services
bbl	Components for business logic
bdl	Vocabulary types, messaging, containers, parsers
bsl	Standard library and supporting infrastructure
btl	Network transport

Standard business day-count utilities

- Source directory **groups**

The Package Groups

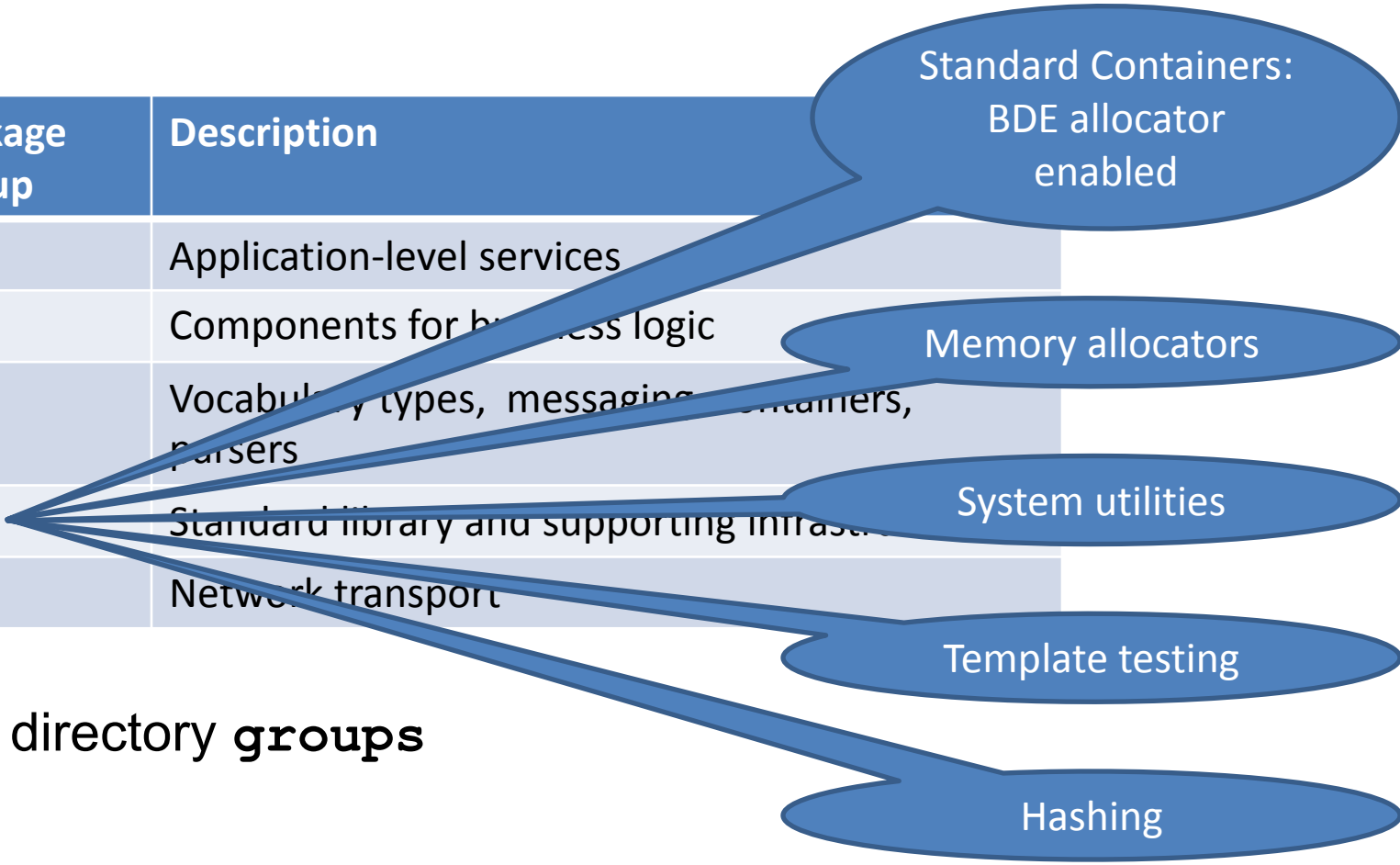
Package Group	Description
bal	Application-level ser…
bbl	Comp…ents for business logic
bdl	Vocabulary types, messaging, containers, parsers
bsl	Standard …brary and …pporting infrast…
btl	Network transport



- Source directory **`groups`**

The Package Groups

Package Group	Description
bal	Application-level services
bbl	Components for b[illegible]ess logic
bdl	Vocabu[illegible]y types, messaging [illegible]ntainers, p[illegible]arsers
bsl	Standard library and supporting infrast[illegible]
btl	Netwo[illegible]k transport



- Source directory **groups**

The Package Groups

Package Group	Description
bal	Application-level services
bbl	Components for business logic
bdl	Vocabulary types, messaging, cont… parsers
bsl	Standard library and supporting i…
btl	Network transport

Not recommended

Required by higher library

Some useful stuff

- Source directory **groups**

```
btls_leakybucket
btls_ratelimiter
btls_reservationguard
```

The Package Groups

Hierarchy

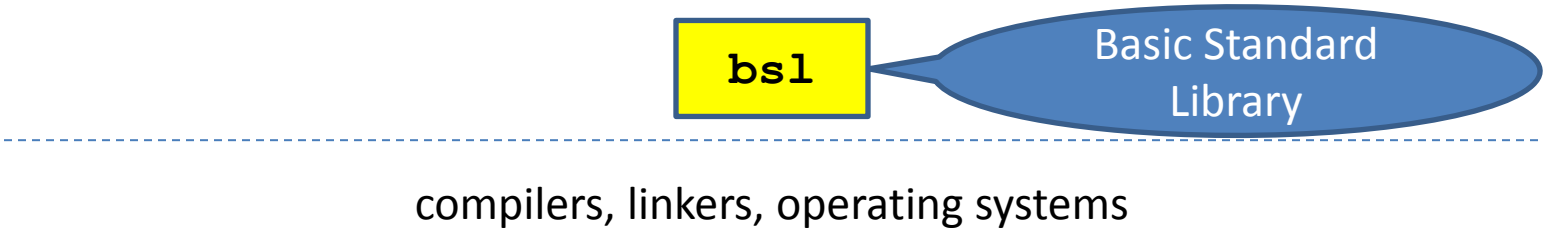
The Package Groups

Hierarchy

compilers, linkers, operating systems

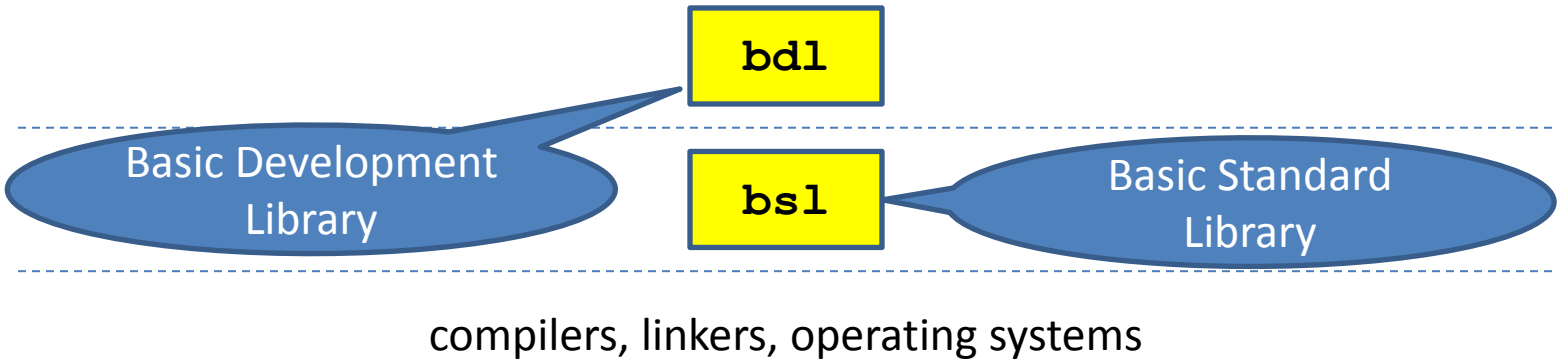
The Package Groups

Hierarchy

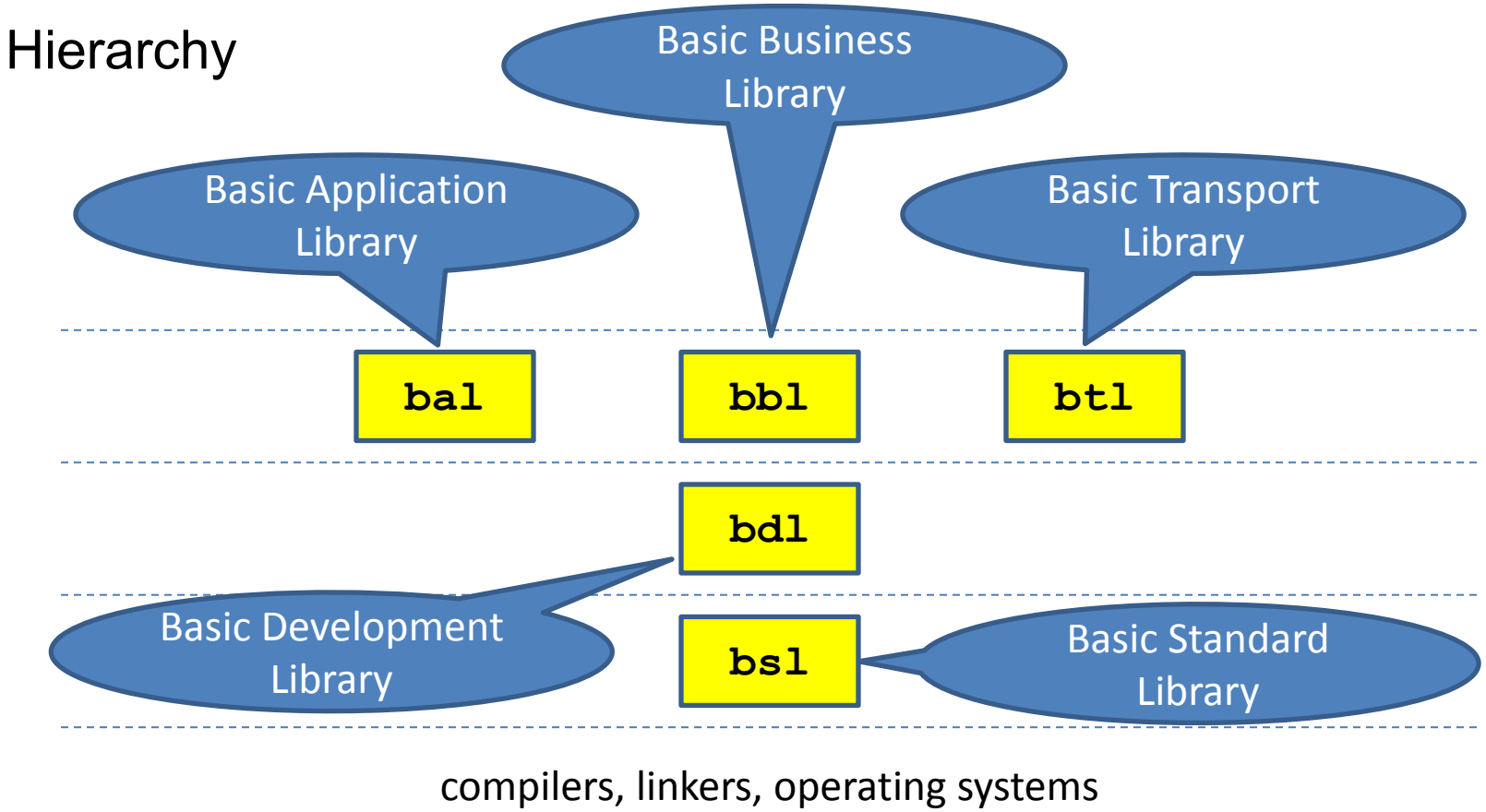


The Package Groups

Hierarchy



The Package Groups



Goals

To acquire:

- Familiarity with the BDE mind-set
- Ability to:
  - Navigate BDE documentation/resources
  - Program within narrow contracts
  - Invoke defensive programming facilities

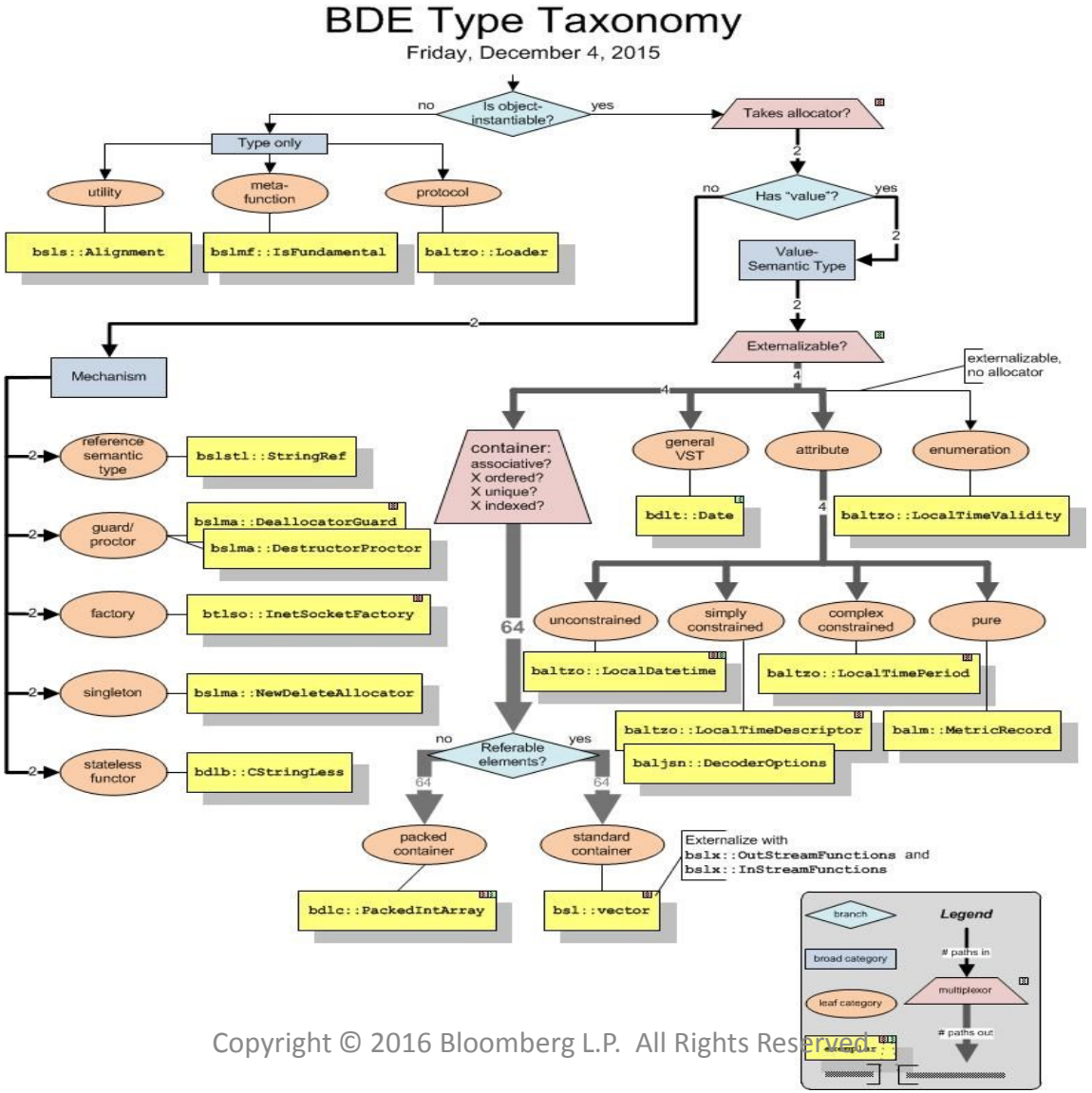
Documentation

Starting Point

`https://github.com/bloomberg/bde`

- BDE wiki, library overviews and links to
  - Tutorials
  - Coding Standard
  - How to contribute
- Quick-Start Guide
  - Build instructions
- Online Library Documentation
  - **`doxygen`** pages

Taxonomy of Classes



Taxonomy of Classes

Each type of class has

1. a well-defined organization,
2. an expected set of methods, and
3. common naming conventions.

Taxonomy of Classes

- *Value-semantic class,* defines a *value*

`bdlt::Date`, `bdlt::DayOfWeek`, `bdlt::Datetime`, `bdlt::Calendar`, `bdlt::PackedCalendar`

- *Mechanism,* has *state*, but not value

`bdlma::BufferedSequentialAllocator`, `bdlma::MultipoolAllocator`

- *Utility,* namespace for functions

`bdlt::DateUtil`, `bdlt::CurrentTime`

- *Protocol,* pure abstract class

`bdlt::CalendarLoader`, `bslma::Allocator`

Taxonomy of Classes

- *Value-semantic class,* defines a *value*

  `bdlt::Date`, **`bdlt::DayOfWeek`**, **`bdlt::Datetime`**, **`bdlt::Calendar`**, **`bdlt::PackedCalendar`**

- *Mechanism,* has *state*, but not value

  `bdlma::BufferedSequentialAllocator`, **`bdlma::MultipoolAllocator`**

- *Utility,* namespace for functions

  `bdlt::DateUtil`, **`bdlt::CurrentTime`**

- *Protocol,* pure abstract class

  `bdlt::CalendarLoader`, **`bslma::Allocator`**

Documentation

Quick Links: bal | bbl | bdl | bsl | btl

Main Page | Related Pages | Components | Namespaces | Classes | Files

Components

Here is a table of all package groups (a.k.a., groups), packages, and components:

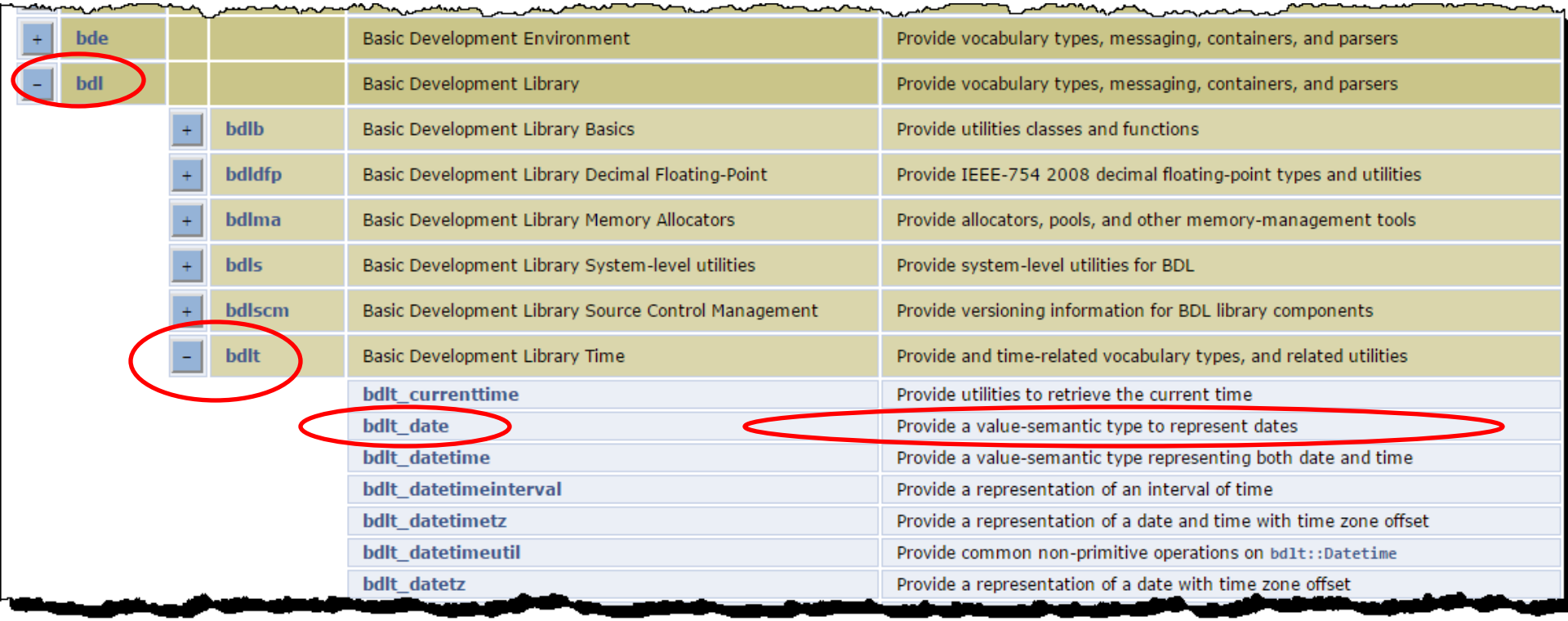
Collapse All Groups | Expand All Packages

	Group		Package	Mnemonic/Component	Purpose
+	bal			Basic Application Environment	Provide application-level support services
+	bbl			Basic Business Library	Provide a foundation for component-based business logic development
+	bdl			Basic Development Library	Provide vocabulary types, messaging, containers, and parsers
+	bsl			Basic Standard Library	Provide a comprehensive foundation for component-based development
+	btl			Basic Transport Environment	Provide support, services, and frameworks for transport (IPC)

Collapse All Groups | Expand All Packages

Documentation

DOXYGEN



+ bde		Basic Development Environment	Provide vocabulary types, messaging, containers, and parsers
− bdl		Basic Development Library	Provide vocabulary types, messaging, containers, and parsers
	+ bdlb	Basic Development Library Basics	Provide utilities classes and functions
	+ bdldfp	Basic Development Library Decimal Floating-Point	Provide IEEE-754 2008 decimal floating-point types and utilities
	+ bdlma	Basic Development Library Memory Allocators	Provide allocators, pools, and other memory-management tools
	+ bdls	Basic Development Library System-level utilities	Provide system-level utilities for BDL
	+ bdlscm	Basic Development Library Source Control Management	Provide versioning information for BDL library components
	− bdlt	Basic Development Library Time	Provide and time-related vocabulary types, and related utilities
		bdlt_currenttime	Provide utilities to retrieve the current time
		bdlt_date	Provide a value-semantic type to represent dates
		bdlt_datetime	Provide a value-semantic type representing both date and time
		bdlt_datetimeinterval	Provide a representation of an interval of time
		bdlt_datetimetz	Provide a representation of a date and time with time zone offset
		bdlt_datetimeutil	Provide common non-primitive operations on `bdlt::Datetime`
		bdlt_datetz	Provide a representation of a date with time zone offset

- Package group **`bdl`**
- Package **`bdlt`**
- Component **`bdlt_date`**

Components

Package bdlt

[Package Group bdl]

Provide date and time vocabulary types, and related utilities. More...

Components

Component bdlt_calendar
Provide fast repository for accessing weekend/holiday information.

Component bdlt_calendarreverseiteratoradapter
Provide reverse iterator adapter for calendar iterators.

Component bdlt_currenttime
Provide utilities to retrieve the current time.

Component bdlt_date
Provide a value-semantic type to represent dates.

Component bdlt_datetime

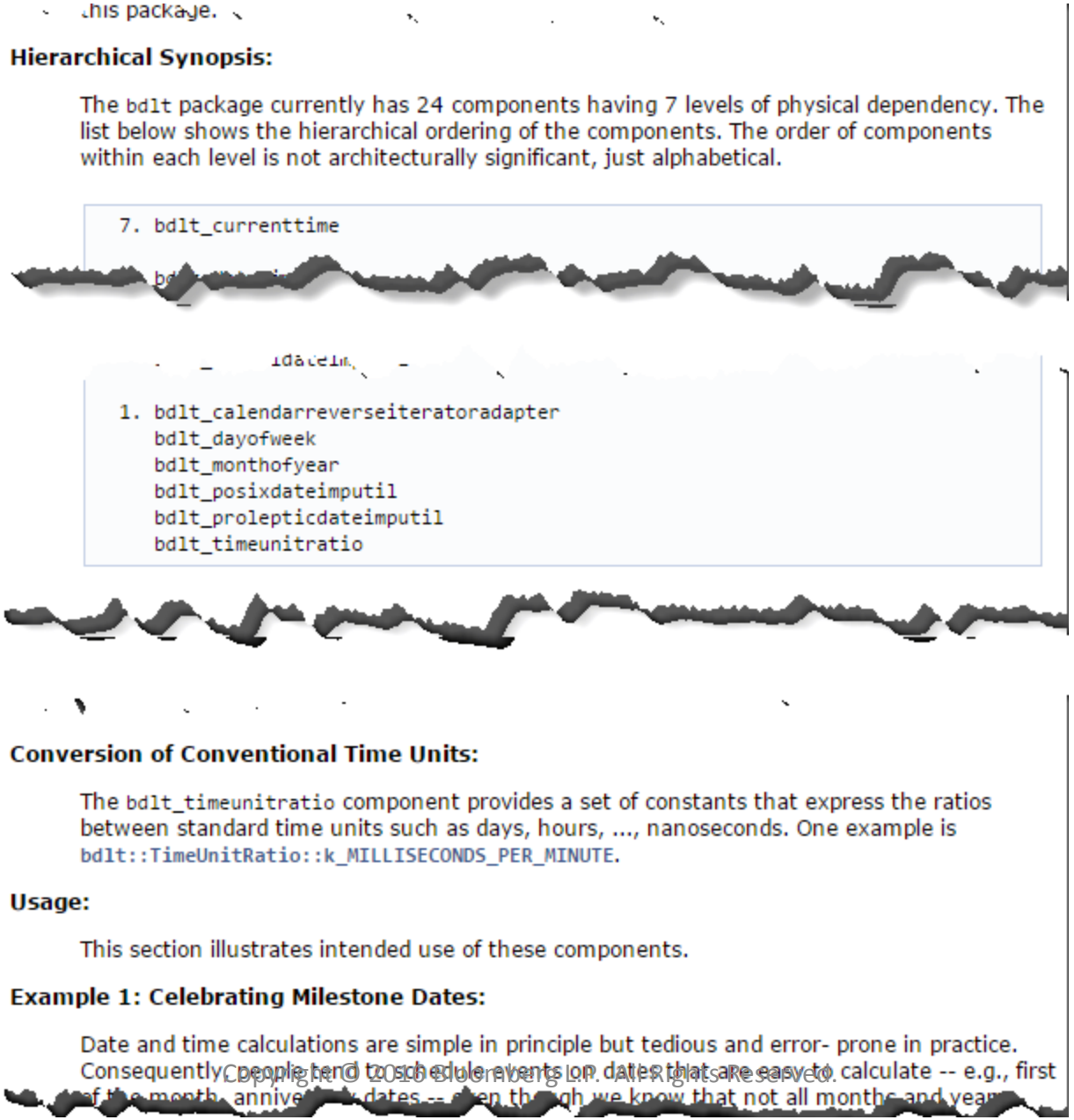
time

Detailed Description

Outline

- Purpose
- MNEMONIC: Basic Development Library Time (bdlt)
- Description
  - Hierarchical Synopsis
  - Component Synopsis
  - Value Types
    - Value Types: `Date`, `Time` and `Datetime`
      - Singular `Time` and `Datetime` Values
    - Timezone Augmented Value Types: `DateTz`, `TimeTz` and `DatetimeTz`
    - Enumerated Values
  - Utilities
    - Obtaining Current Date, Time, and Local-Time Offset Values
    - Advanced Date Arithmetic: `bdlt::DateUtil`
    - Conversion of `Date`, `Time` and `Datetime` Values
    - Conversion of Conventional Time Units
  - Usage
    - Example 1: Celebrating Milestone Dates

Documentation



this package.

Hierarchical Synopsis:

The bdlt package currently has 24 components having 7 levels of physical dependency. The list below shows the hierarchical ordering of the components. The order of components within each level is not architecturally significant, just alphabetical.

7. bdlt_currenttime

1. bdlt_calendarreverseiteratoradapter
   bdlt_dayofweek
   bdlt_monthofyear
   bdlt_posixdateimputil
   bdlt_prolepticdateimputil
   bdlt_timeunitratio

Conversion of Conventional Time Units:

The bdlt_timeunitratio component provides a set of constants that express the ratios between standard time units such as days, hours, ..., nanoseconds. One example is bdlt::TimeUnitRatio::k_MILLISECONDS_PER_MINUTE.

Usage:

This section illustrates intended use of these components.

Example 1: Celebrating Milestone Dates:

Date and time calculations are simple in principle but tedious and error- prone in practice. Consequently, people tend to schedule events on dates that are easy to calculate -- e.g., first of the month, anniversary dates -- even though we know that not all months and years

Documentation

DOXYGEN

Component bdlt_date

[Package bdlt]

Purpose:

Provide a value-semantic type to represent dates.

Classes:

bdlt::Date value-semantic date type consistent with the Unix calendar

See also:

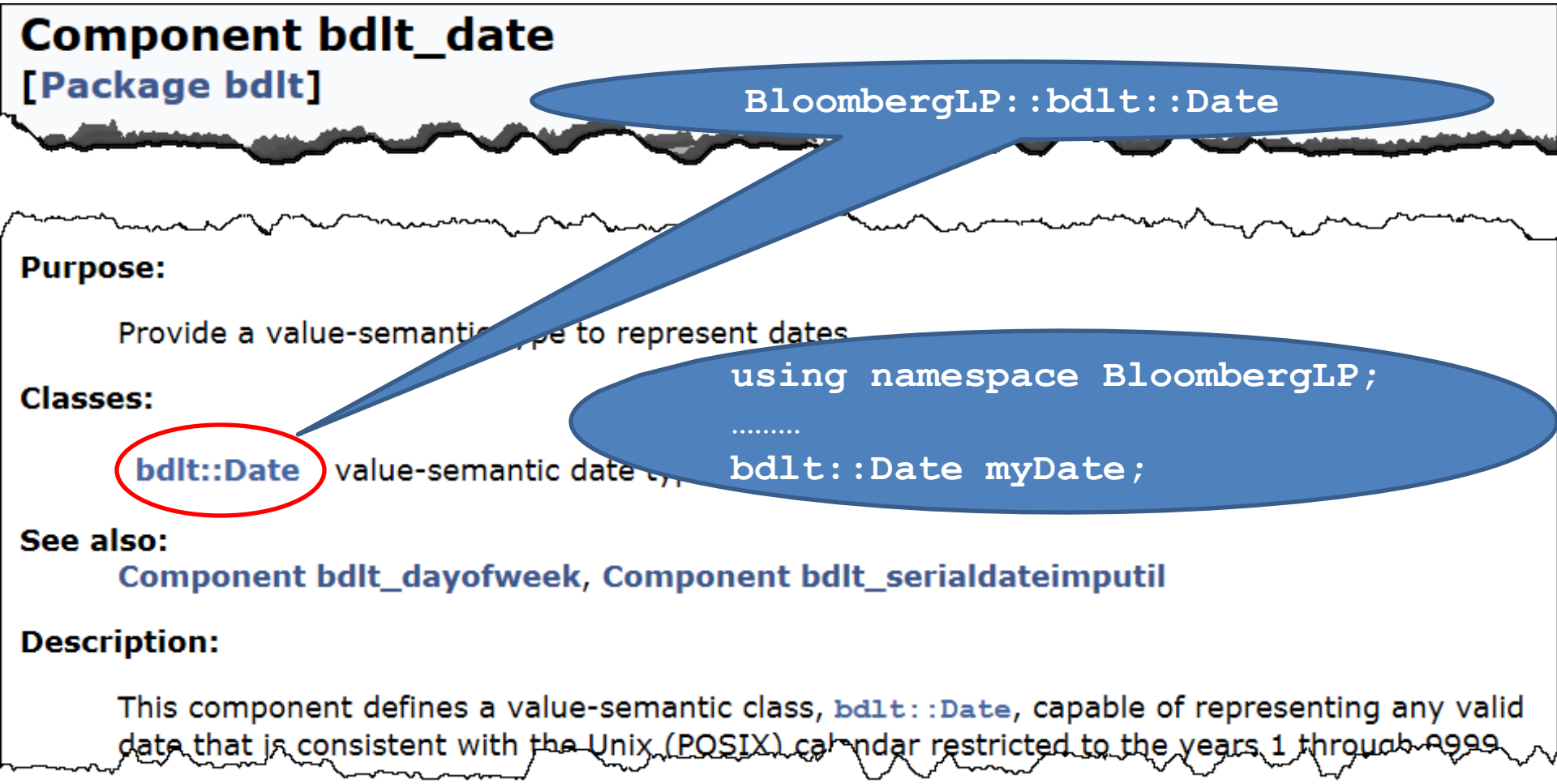
Component bdlt_dayofweek, Component bdlt_serialdateimputil

Description:

This component defines a value-semantic class, `bdlt::Date`, capable of representing any valid date that is consistent with the Unix (POSIX) calendar restricted to the years 1 through 9999

Documentation

DOXYGEN

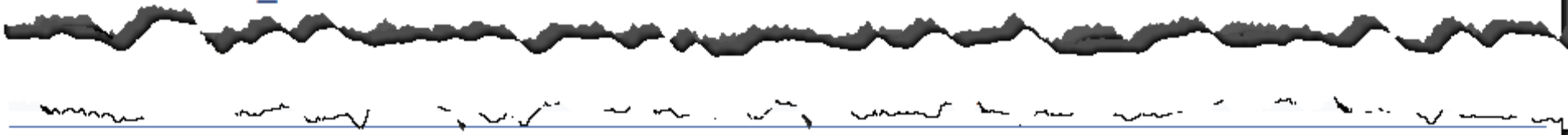


Documentation

bdlt::Date Class Reference

| Friends

```
#include <bdlt_date.h>
```



Detailed Description

This class implements a complex-constrained, value-semantic type for representing dates according to the Unix (POSIX) calendar. Each object of this class *always* represents a *valid* date value in the range `[0001JAN01 .. 9999DEC31]` inclusive. The interface of this class supports `Date` values expressed in terms of either year/month/day (the canonical representation) or year/day-of-year (an alternate representation). See **Valid Date Values and Their Representations** for details.

Constructor & Destructor Documentation

bdlt::Date::Date ( )

Create a `Date` object having the earliest supported date value, i.e., having a year/month/day representation of `0001/01/01`.

Component Up Close
`bdlt_date`
(with commentary)

Component-Level Doc

```
// bdlt_date.h                                                    -*-C++-*-
#ifndef INCLUDED_BDLT_DATE
#define INCLUDED_BDLT_DATE

#ifndef INCLUDED_BSLS_IDENT
#include <bsls_ident.h>
#endif
BSLS_IDENT("$Id: $")

//@PURPOSE: Provide a value-semantic type to represent dates.
//
//@CLASSES:
//  bdlt::Date: value-semantic date type consistent with the Unix calendar
//
//@SEE_ALSO: bdlt_dayofweek, bdlt_serialdateimputil
//
//@DESCRIPTION: This component defines a value-semantic class, 'bdlt::Date',
// capable of representing any valid date that is consistent with the Unix
// (POSIX) calendar restricted to the years 1 through 9999 (inclusive):
//..
//  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/cal.html
//..
// "Actual" (i.e., natural) day and date calculations are supported directly by
```

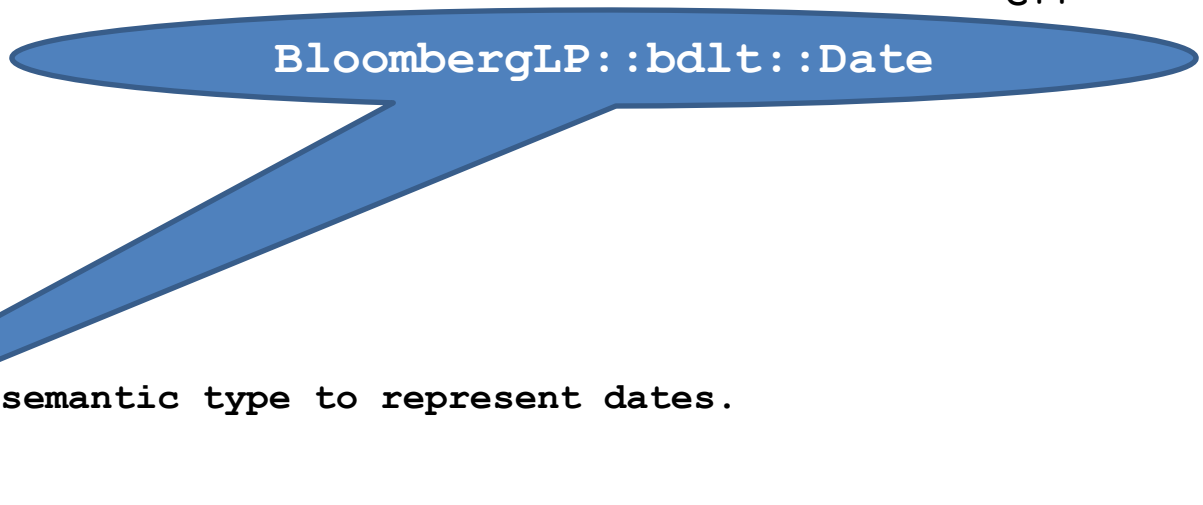
Component-Level Doc

```
// bdlt_date.h                                                    -*-C++-*-
#ifndef INCLUDED_BDLT_DATE
#define INCLUDED_BDLT_DATE

#ifndef INCLUDED_BSLS_IDENT
#include <bsls_ident.h>
#endif
BSLS_IDENT("$Id: $")

//@PURPOSE: Provide a value-semantic type to represent dates.
//
//@CLASSES:
//  bdlt::Date: value-semantic date type consistent with the Unix calendar
//
//@SEE_ALSO: bdlt_dayofweek, bdlt_serialdateimputil
//
//@DESCRIPTION: This component defines a value-semantic class, 'bdlt::Date',
// capable of representing any valid date that is consistent with the Unix
// (POSIX) calendar restricted to the years 1 through 9999 (inclusive):
//..
//  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/cal.html
//..
// "Actual" (i.e., natural) day and date calculations are supported directly by
```

Component-Level Doc




```
// bdlt_date.h                                                        -*-C++-*-
#ifndef INCLUDED_BDLT_DATE
#define INCLUDED_BDLT_DATE

#ifndef INCLUDED_BSLS_IDENT
#include <bsls_ident.h>
#endif
BSLS_IDENT("$Id: $")

//@PURPOSE: Provide a value-semantic type to represent dates.
//
//@CLASSES:
//  bdlt::Date: value-semantic date type consistent with the Unix calendar
//
//@SEE_ALSO: bdlt_dayofweek, bdlt_serialdateimputil
//
//@DESCRIPTION: This component defines a value-semantic class, 'bdlt::Date',
// capable of representing any valid date that is consistent with the Unix
// (POSIX) calendar restricted to the years 1 through 9999 (inclusive):
//..
//  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/cal.html
//..
// "Actual" (i.e., natural) day and date calculations are supported directly by
```

Component-Level Doc

```
// bdlt_date.h                                                    -*-C++-*-
#ifndef INCLUDED_BDLT_DATE
#define INCLUDED_BDLT_DATE

#ifndef INCLUDED_BSLS_IDENT
#include <bsls_ident.h>
#endif
BSLS_IDENT("$Id: $")

//@PURPOSE: Provide a value-semantic type to represent dates.
//
//@CLASSES:
//  bdlt::Date: value-semantic date type consistent with the Unix calendar
//
//@SEE_ALSO: bdlt_dayofweek, bdlt_serialdateimputil
//
//@DESCRIPTION: This component defines a value-semantic class, 'bdlt::Date',
// capable of representing any valid date that is consistent with the Unix
// (POSIX) calendar restricted to the years 1 through 9999 (inclusive):
//..
//  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/cal.html
//..
// "Actual" (i.e., natural) day and date calculations are supported directly by
```

BloombergLP::bdlt::Date

```
using namespace BloombergLP;
.....
bdlt::Date myDate;
```

Component-Level Doc

```
///Usage
///-----
// This section illustrates intended use of this component.
//
///Example 1: Basic Use of 'bdlt::Date'
/// - - - - - - - - - - - - - - - - - - - -
// The following snippets of code illustrate how to create and use a
// 'bdlt::Date' object.
//
// First, we create a default date 'd1':
//..
//  bdlt::Date d1;           assert(   1 == d1.year());
//                           assert(   1 == d1.month());
//                           assert(   1 == d1.day());
//..
// Next, we set 'd1' to July 4, 1776:
//..
```

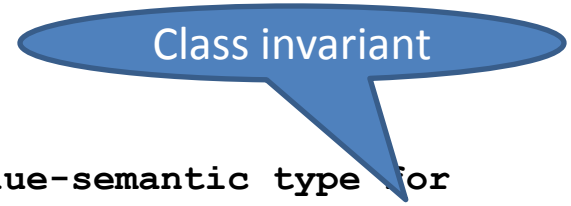
Pedagogical asserts of object state

Component Up Close

Class-Level Doc

```
                    // ==========
                    // class Date
                    // ==========

class Date {
    // This class implements a complex-constrained, value-semantic type for
    // representing dates according to the proleptic Gregorian calendar.  Each
    // object of this class *always* represents a *valid* date value in the
    // range '[0001JAN01 .. 9999DEC31]' inclusive.  The interface of this class
    // supports 'Date' values expressed in terms of either year/month/day (the
    // canonical representation) or year/day-of-year (an alternate
    // representation).  See {Valid Date Values and Their Representations} for
    // details.
```



Tacit Guarantees

- In the absence of other specifications, assume each class guarantees
  - **`const`** Thread-Safety
  - Exception Agnostic
  - Alias Safety (if pertinent)
- **`bsldoc_glossary`** defines terms
- Stronger or weaker guarantees are explicitly documented

Exceptions

Exception Agnostic

- Basic exception safety guarantee
  - objects left in valid state
- Exception Neutral
  - any exceptions pass through
- Does not rely on exception syntax
  - BDE eschews **`try`**/**`catch`** blocks
  - BDE uses “proctors”, i.e., guards with **`release`** methods, an RAII-like technique

Exceptions

- BDE does not throw exceptions, except…
  - As required by the Standard, e.g., **`bsl::bad_alloc`**
  - Testing the exception guarantees of our code
- Whence exceptions?
  - Exceptions from legacy code (at Bloomberg)
  - Exceptions from user installed callbacks

Function-Level Doc

```
// CLASS METHODS
static bool isValidYearMonthDay(int year, int month, int day);
    // Return 'true' if the specified 'year', 'month', and 'day' represent
    // a valid value for a 'Date' object, ……

// CREATORS
Date();
    // Create a 'Date' object having the earliest supported valid date value
    // i.e., "year/month/day" representation of '0001/01/01'.

Date(const Date& original);
    // Create a 'Date' object having the value of the specified 'original'
    // date.

// MANIPULATORS
Date& operator=(const Date& rhs);
    // Assign to this date object the value of the specified 'rhs' date,
    // and return a reference providing modifiable access to this object.

// ACCESSORS
void getYearDay(int *year, int *dayOfYear) const;
    // Load, into the specified 'year' and 'dayOfYear', the respective
    // 'year' and 'dayOfYear' attribute values of this date.
```

Function-Level Doc

A contract with user (and tester) in dead-regular form:

1. *Required*: What the function *does* (a verb); e.g.,
   `Set`, **`Load`**, **`Create`**, **`Destroy`**, **`Append`**, …
2. What the function returns -- sometimes all it "does"
   - Optional arguments, if any
3. All other essential behavior
4. Conditions leading to undefined behavior:
   `The behavior is undefined unless`…
5. Clarifications:
   `Note that` …

First mention of parameter name prefaced by "the specified".

A Value-Semantic Type

```
// bdlt_date.h                                                    -*-C++-*-
#ifndef INCLUDED_BDLT_DATE
#define INCLUDED_BDLT_DATE

#ifndef INCLUDED_BSLS_IDENT
#include <bsls_ident.h>
#endif
BSLS_IDENT("$Id: $")

//@PURPOSE: Provide a value-semantic type to represent dates.
//
//@CLASSES:
//  bdlt::Date: value-semantic date type using the proleptic Gregorian calendar
//
//@SEE_ALSO: bdlt_dayofweek, bdlt_serialdateimputil
//
//@DESCRIPTION: This component defines a value-semantic class, 'bdlt::Date',
// capable of representing any valid date that is consistent with the proleptic
// Gregorian calendar restricted to the years 1 through 9999 (inclusive):
//..
//  http://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar
//..
// "Actual" (i.e., natural) day and date calculations are supported directly by
```

Taxonomy of Classes: Value-Semantic

Value-semantic Types

- Are *Regular Types* a la Alex Stepanov
- Define a *value* from the *salient* attributes of the class
- That are compared by **`operator==`**

General Value-Semantic Methods

Expected operations of a Value-Semantic Type (*VST*)

- Default and copy constructors

```
VST();
VST(attribute1, attribute2, ... );
VST(const VST& original);
```

- Assignment

```
VST& operator=(const VST& rhs);
```

- Equality (and non-equality)

```
bool operator==(const VST& lhs,
                const VST& rhs);
```

- Attribute “getters” and “setters”
  - BDE has different naming convention

General Value-Semantic Methods

Expected operations of a Value-Semantic Type (*VST*)

- ```
  bsl::ostream& print(
                bsl::ostream& stream,
                int level = 0,
                int spacesPerLevel = 4) const;
  ```
- ```
  bsl::ostream& operator<<(
                           bsl::ostream& stream,
                           const VST&    object);
  ```
- Print value in some “human-readable” format
  - Intended to aid debugging and testing
  - Subject to change
  - Do not write (machine) parsers for this format

Date-Specific Operations

Also, any other methods *appropriate* to the type

For example, **`bdlt::Date`** methods include:

```
bdlt::Date& bdlt::Date::operator++();

bdlt::Date& bdlt::Date::operator-=(int numDays);

bool operator>(const bdlt::Date& lhs,
               const bdlt::Date& rhs);

void setYearDay(int year, int dayOfYear);
```

Other VST classes (e.g., an employee record) do not have any additional methods; they simply contain related attributes.

Valid Date Values

`bdlt::Date`, represents dates in range
`[ 1/1/1 .. 12/31/9999 ]`

- *Except* September [3 .. 13], 1752
- A POSIX Standard
  - Proleptic-Gregorian dates available

Class (static) methods inform if a date value is valid for this class

```
bdlt::Date::isValidYearMonthDay(year,
                                month,
                                day);

bdlt::Date::isValidYearDay(year,
                           dayOfYear);
```

Using `bdlt::Date`

Default date

```
bdlt::Date d1; assert(   1 == d1.year());
               assert(   1 == d1.month());
               assert(   1 == d1.day());
```

Careful of argument order

Set some value

```
d1.setYearMonthDay(1776, 7, 4);
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Not a leap year

Set a date with validation

```
int status = d1.setYearMonthDayIfValid(1900, 2, 29);
               assert(   0 != status);
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Correctly fails

No change

Using `bdlt::Date`

Default date

```
bdlt::Date d1; assert(   1 == d1.year());
               assert(   1 == d1.month());
               assert(   1 == d1.day());
```

Set some value

```
d1.setYearMonthDay(7, 4, 1776); // BAD IDEA
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Careful of argument order

Set a date with validation

```
int status = d1.setYearMonthDayIfValid(1900, 2, 29);
               assert(   0 != status);
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Not a leap year

Correctly fails

No change

Using `bdlt::Date`

Default date

```
bdlt::Date d1; assert(   1 == d1.year());
               assert(   1 == d1.month());
               assert(   1 == d1.day());
```

Careful of argument order

Set some value

```
d1.setYearMonthDay(1776, 7, 4);
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Not a leap year

Set a date with validation

```
int status = d1.setYearMonthDayIfValid(1900, 2, 29);
               assert(   0 != status);
               assert(1776 == d1.year());
               assert(   7 == d1.month());
               assert(   4 == d1.day());
```

Correctly fails

No change

Using `bdlt::Date`

Validate a date *value* and create a date *object*:

```
bool isValid = bdlt::Date::isValidYearMonthDay(1900,
                                               2,
                                               28);
               assert(isValid);
bdlt::Date d2(1900, 2, 28);
```

Date arithmetic

```
int dayOfYear = d2.dayOfYear();
++d2;         assert(dayOfYear + 1 == d2.dayOfYear());
d2 += 3;      assert(dayOfYear + 4 == d2.dayOfYear());
              assert(1900 == d2.year());
              assert(   3 == d2.month());
              assert(   4 == d2.day());

int dateDiff = bdlt::Date(2011, 1, 31)
             - bdlt::Date(2011, 2,  1);
               assert(-1 == dateDiff);
```

Narrow Contracts

'

Narrow versus Wide Contracts

Terminology

- *Narrow contacts*
  - Place preconditions on input parameters and/or object state.
  - Required preconditions for contracted behavior are always clearly stated.
  - Otherwise, the behavior is undefined.
  - Narrow contracts are a common feature of BDE code.

- *Wide contracts*
  - Have *no* constraints on input or object state.

Undefined Behavior

Typical of BDE classes, **`bdlt::Date`** documentation often mentions *undefined behavior*; e.g.:

```
// CREATORS
 Date(int year, int month, int day);
     // Create a 'Date' object having the value
     // represented by the specified 'year', 'month', and
     // 'day'.  The behavior is undefined unless 'year',
     // 'month', and 'day' represent a valid 'Date' value
     // (see 'isValidYearMonthDay').
```

Undefined Behavior

```
// MANIPULATORS
Date& operator++();
      // Set this object to have the value that is one day
      // later than its current value, and return a
      // reference providing modifiable access to this
      // object.  The behavior is undefined if the
      // year/month/day representation of the current value
      // is '9999/12/31'.

void setYearDay(int year, int dayOfYear);
     // Set this object to have the value represented by
     // the specified 'year' and 'dayOfYear'.  The behavior
     // is undefined unless 'year' and 'dayOfYear'
     // represent a valid 'Date' value (see
     // 'isValidYearDay').
```

Undefined Behavior

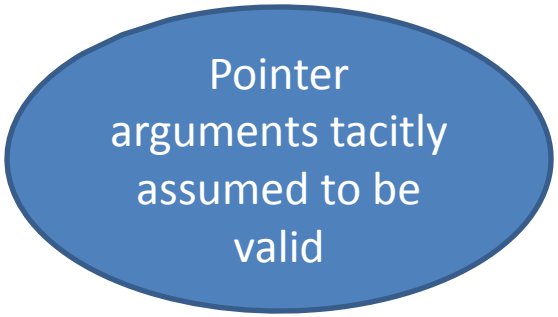
Undefined behavior is a key concept in BDE.

- If *preconditions* for an action are met, the contracted behavior can be assumed correct (heavily tested).
- If preconditions for an operation are *not* met, subsequent behavior is *not defined*; *anything* (or *nothing*) may happen
  - In some build modes, BDE asserts before allowing undefined behavior.
  - The *entire* process is corrupt, not just one part.
  - Whatever does happen might change over time.
- Preconditions are explicitly stated in the documentation.
  - **`The behavior is undefined unless`**…
  - Tacit precondition: pointer arguments are valid.

Undefined Behavior

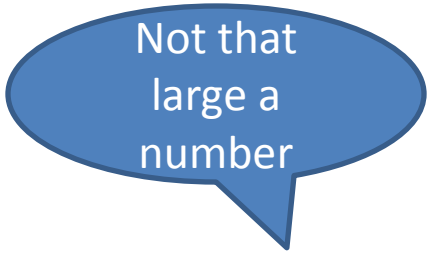
```
// ACCESSORS
void getYearDay(int *year, int *dayOfYear) const;
    // Load, into the specified 'year' and
    // 'dayOfYear', the respective 'year' and
    // 'dayOfYear' attribute values of this date.
```

Tacit Preconditions



Undefined Behavior

Developers are responsible for avoiding undefined behavior, e.g.:




```
bdlt::Date d1(0, 0, 0);                          // BAD IDEA

bdlt::Date d2; --d2;                             // BAD IDEA

bdlt::Date d3 = bdlt::Date() + 3652061;          // BAD IDEA

bdlt::Date d4; d4.setYearDay(d4.year(), 367);
                                                 // BAD IDEA
```

Undefined Behavior

```
int findDayOfInterest(bdlt::Date *result,
                      bdlt::Date  first,
                      bdlt::Date  last)
  // Load to the specified 'result' the earliest date
  // of interest between the specified 'first' and
  // 'last' dates inclusive.  Return 0 on success and a
  // non-zero value otherwise.  The behavior is
  // undefined unless 'first <= last'.
{
    for (bdlt::Date date = first;
                    date < last; // BAD IDEA
                  ++date) {
        if (isInteresting(date)) {
            *result = date;
            return 0;                              // RETURN
        }

    return -1;
}
```

Undefined Behavior

One often hears people say…

- "Valid **`bdlt::Date`** object"
  - Redundant.
  - Each **`bdlt::Date`** *object* always holds a valid value.
  - Invariant of the **`bdlt::Date`** *class* .
- "Invalid **`bdlt::Date`** object"
  - Contradiction.
  - If its value is invalid, it is *not* a **`bdlt::Date`** *object.*
  - *No normal contractual behavior can be assumed.*

Defensive Programming

Preconditions Asserts

- Programming within narrow contracts can be challenging. (More than pointers? Integer overflow?)
- BDE code has compile-time conditional assertion checks on preconditions, when possible/practical.
- Defensive checks are **not** part of the contracts.
  - Redundant code that exists to aid development.
  - Not a substitute for thorough testing.
  - Assume inactive in production code.
- Implemented using **`BSLS_ASSERT`** macros.

Preconditions Asserts

```
int findNextFreeDate(bdlt::Date            *freeDate,
                     const bdlt::Date&      targetDate,
                     const bdlt::Calendar&  calendar)
    // Load, into the specified 'freeDate', first non-business date on or after
    // the specified 'targetDate', where business dates are determined by the
    // specified 'calendar'.  Return 0 on success, and a non-zero value with no
    // effect at 'freeDate' otherwise (i.e., all days in 'calendar' are
    // business days).  The behavior is undefined unless 'targetDate' is in the
    // range of 'calendar'.
{
    BSLS_ASSERT(freeDate);
    BSLS_ASSERT(0 < calendar.length());
    BSLS_ASSERT(calendar.isInRange(targetDate));

    //…
}
```

Preconditions Asserts

```
int findNextFreeDate(bdlt::Date             *freeDate,
                     const bdlt::Date&       targetDate,
                     const bdlt::Calendar&   calendar)
    // Load, into the specified 'freeDate', first non-business date on or after
    // the specified 'targetDate', where business dates are determined by the
    // specified 'calendar'.  Return 0 on success, and a non-zero value with no
    // effect at 'freeDate' otherwise (i.e., all days in 'calendar' are
    // business days).  The behavior is undefined unless 'targetDate' is in the
    // range of 'calendar'.
{
    BSLS_ASSERT(freeDate);
    BSLS_ASSERT(0 < calendar.length());
    BSLS_ASSERT(calendar.isInRange(targetDate));

    //…
}
```

Partial test of pointer validity

Implicit precondition on **`calendar`**

`BSLS_ASSERT` Macros

The defensive programming macros have two degrees of freedom:

- Which assertions are instantiated
- What happens when an assertion fails

`BSLS_ASSERT` Macros

- **`BSLS_ASSERT`**
  - Max overhead roughly 5 to 10%
- **`BSLS_ASSERT_SAFE`**
  - More than ~10% overhead
  - e.g., **`inline`** methods
- **`BSLS_ASSERT_OP`**
  - Checks needed even in optimized mode
  - e.g., **`BSLS_ASSERT_OP(!"Unreachable");`**

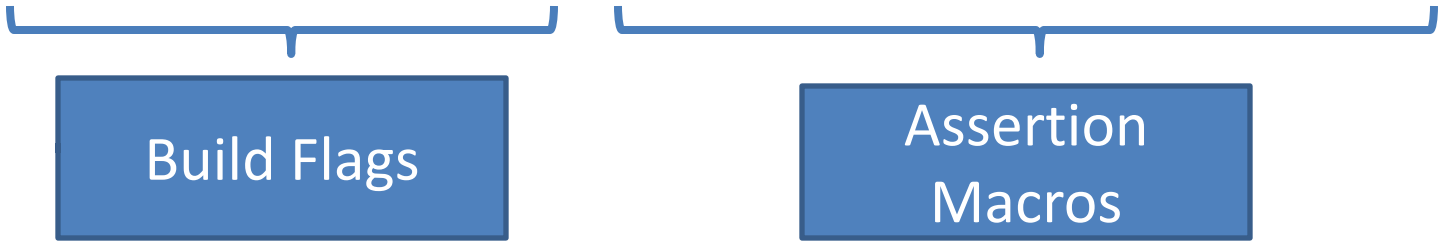
`BSLS_ASSERT` Macros

Separate responsibilities:

- The *library* developer categorizes and installs the checks in the source code.
- The *application* builder decides the appropriate level of defensive checking.

BSLS_ASSERT Build Targets

BSLS_ASSERT_...	..._SAFE	BSLS_ASSERT	..._OPT
..._LEVEL_ASSERT_SAFE	Active	Active	Active
..._LEVEL_ASSERT		Active	Active
..._LEVEL_ASSERT_OPT			Active
..._LEVEL_NONE			



`BSLS_ASSERT` Build Targets

- For example, build flag **`BSLS_ASSERT_LEVEL_ASSERT_SAFE`** enables all tests
- Default: Specifying none of these flags enables all but the "safe" (i.e., most expensive) checks
- See **`bsls_assert.h for`** details
- For **`waf`** builds
  - Use the **`--safe`** option to **`waf configure`**
  - See **`https://bloomberg.github.io/bde-tools/waf.html`**

`BSLS_ASSERT` Handlers

Three handlers provided by **`bsls::Assert`**

- **`failAbort`**, the default handler
- **`failSleep`**, debug the live process
- **`failThrow`**, throws the **`bsls::AssertTestException`**

Log and Continue

- Return from handler (and enter undefined behavior?)
- Why?
  - Want to instrument legacy code
  - Yet, must not disrupt working services
  - See **`balst_assertionlogger`**

Conclusion

'

Summary

- **https://github.com/bloomberg/bde**
- The BDE libraries provide wide utility.
- The documentation is clear, complete, and extremely regular across libraries.
- The taxonomy of classes aides in comprehension.
- Clearly documented, appropriately narrow contracts allow efficient implementations for experienced developers.
- Defensive programming checks aid correctness.

The End