

# Defensive Programming

Done *Right.*

John Lakos

Monday, September 8, 2014

# Copyright Notice

© 2014 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

# Abstract

In our component-based development methodology, each developer is responsible for ensuring that the software he or she creates is easy to understand and use, and not especially easy to misuse. One common form of misuse is to invoke a library function or method under circumstances where not all of its preconditions are satisfied, leading to **undefined behavior**. Contracts having undefined behavior are not necessarily undesirable, and (for many engineering reasons) are often optimal. Most would agree that a well-implemented library **should do something other than silently continue when a precondition violation is detected**, although these same folks might not agree on what specific action should be taken. Unfortunately, validating preconditions implies writing additional code that will execute at runtime. More code runs slower, and some would fairly argue that they should not be forced to pay for redundant runtime checks in the library software they use. Whether and to what extent library functions should validate their preconditions, and what should happen if a precondition violation is detected are questions that are best answered on an application-by-application basis – i.e., by the owner of `main`. “Defensive Programming Done Right” makes it all possible.

In this talk, we begin by reviewing the basic concepts of **Design-By-Contract (DbC)**, and what we mean by the term **“Defensive Programming” (DP)**. We then explore our overall approach to institutionalizing defensive programming in robust reusable library software such that each application can conveniently specify both the runtime budget (e.g., none, some, lots) for defensive checking, and also the specific action to be taken (e.g., abort, throw, spin) should a precondition violation occur. Along the way, we touch on how modern compilers and linkers work, binary compatibility, and the consequences of possibly violating the one-definition rule in mixed-mode builds. We conclude the talk by describing and then demonstrating our **“negative testing” strategy (and supporting test apparatus)** for readily verifying, in our component-level test drivers, that our defensive checks detect and report out-of-contract client use as intended. Actual source for the supporting utility components will be presented throughout the talk and made available afterwards.

# What's The Problem?

# What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.

# What's The Problem?

Large-Scale C++ Software Design:

- Involves many subtle logical and physical aspects.
- Requires an ability to isolate and modularize **logical functionality** within discrete, fine-grain **physical components**.

# What's The Problem?

## Large-Scale C++ Software Design:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grain physical components.
- Requires the designer to delineate **logical behavior** precisely, while managing the **physical dependencies** on other subordinate components.

# What's The Problem?

## Large-Scale C++ Software:

- Involves many subtle *logical* and *physical* aspects.
- Requires an ability to isolate and modularize logical functionality within discrete, fine-grain physical components.
- Requires the designer to delineate logical behavior precisely, while managing the physical dependencies on other subordinate components.
- **Is broken unless clients of even well-designed, carefully documented, and thoroughly tested software use it properly!**

# Purpose of this Talk

# Purpose of this Talk

Provide a solid introduction to designing and specifying **logically** (and **physically**) sound components, their interfaces, & their contracts:

# Purpose of this Talk

Provide a solid introduction to designing and specifying logically (and physically) sound components, their interfaces, & their contracts:

- Emphasize the importance of documenting what *is* and *is not* *defined behavior*.

# Purpose of this Talk

Provide a solid introduction to designing and specifying logically (and physically) sound components, their interfaces, & their contracts:

- Emphasize the importance of documenting what is and is not defined behavior.
- Show how *undefined behavior* can **reduce costs** associated with initial development, testing, and maintenance, as well as **improve runtime** and **reduce code size**.

# Purpose of this Talk

Provide a solid introduction to designing and specifying logically (and physically) sound components, their interfaces, & their contracts:

- Emphasize the importance of documenting what is and is not defined behavior.
- Show how *undefined behavior* can reduce costs associated with initial development, testing, and maintenance, as well as improve runtime and reduce code size.
- Demonstrate how we can exploit **undefined behavior** to detect **latent defects** in client software using the **bsls\_assert** facility.

# Outline

1. Brief Review of Physical Design

2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)

4. Implementing Defensive Checks

Using the **bsls\_assert** Component

5. Negative Testing

Using the **bsls\_asserttest** Component

# Outline

## 1. Brief Review of Physical Design

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

## 4. Implementing Defensive Checks

Using the `bsls_assert` Component.

## 5. Negative Testing

Using the `bsls_asserttest` Component.

## 1. Brief Review of Physical Design

# Logical versus Physical Design

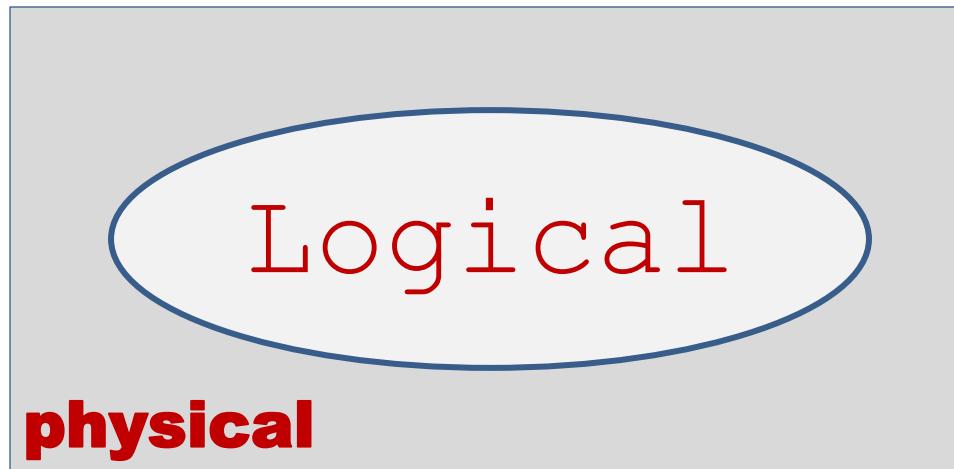
What distinguishes *Logical* from *Physical* Design?



## 1. Brief Review of Physical Design

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?

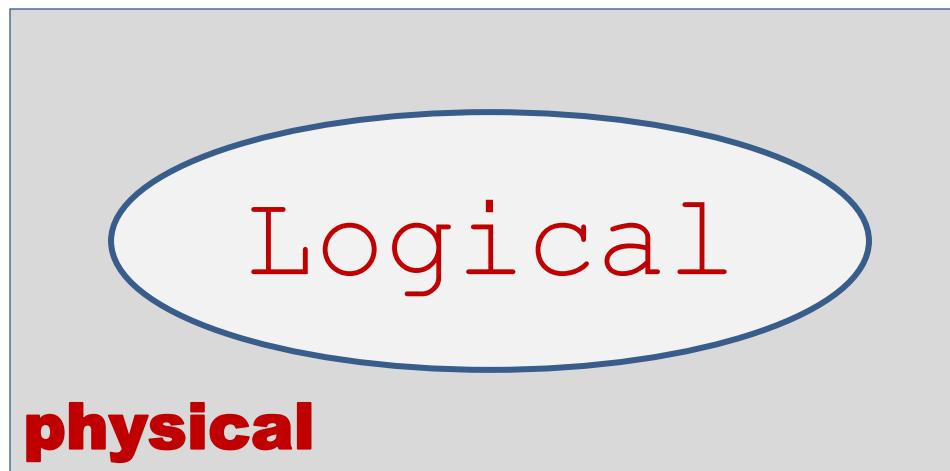


**Logical:** Classes and Functions

## 1. Brief Review of Physical Design

# Logical versus Physical Design

What distinguishes *Logical* from *Physical* Design?



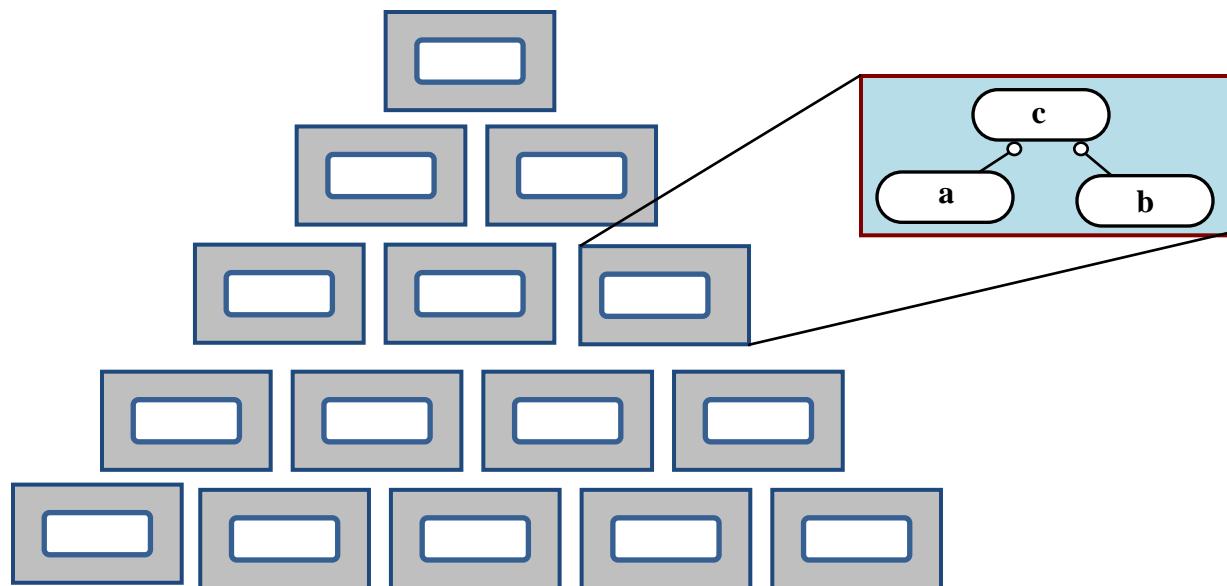
**Logical:** Classes and Functions

**Physical:** Files and Libraries

## 1. Brief Review of Physical Design

# Logical versus Physical Design

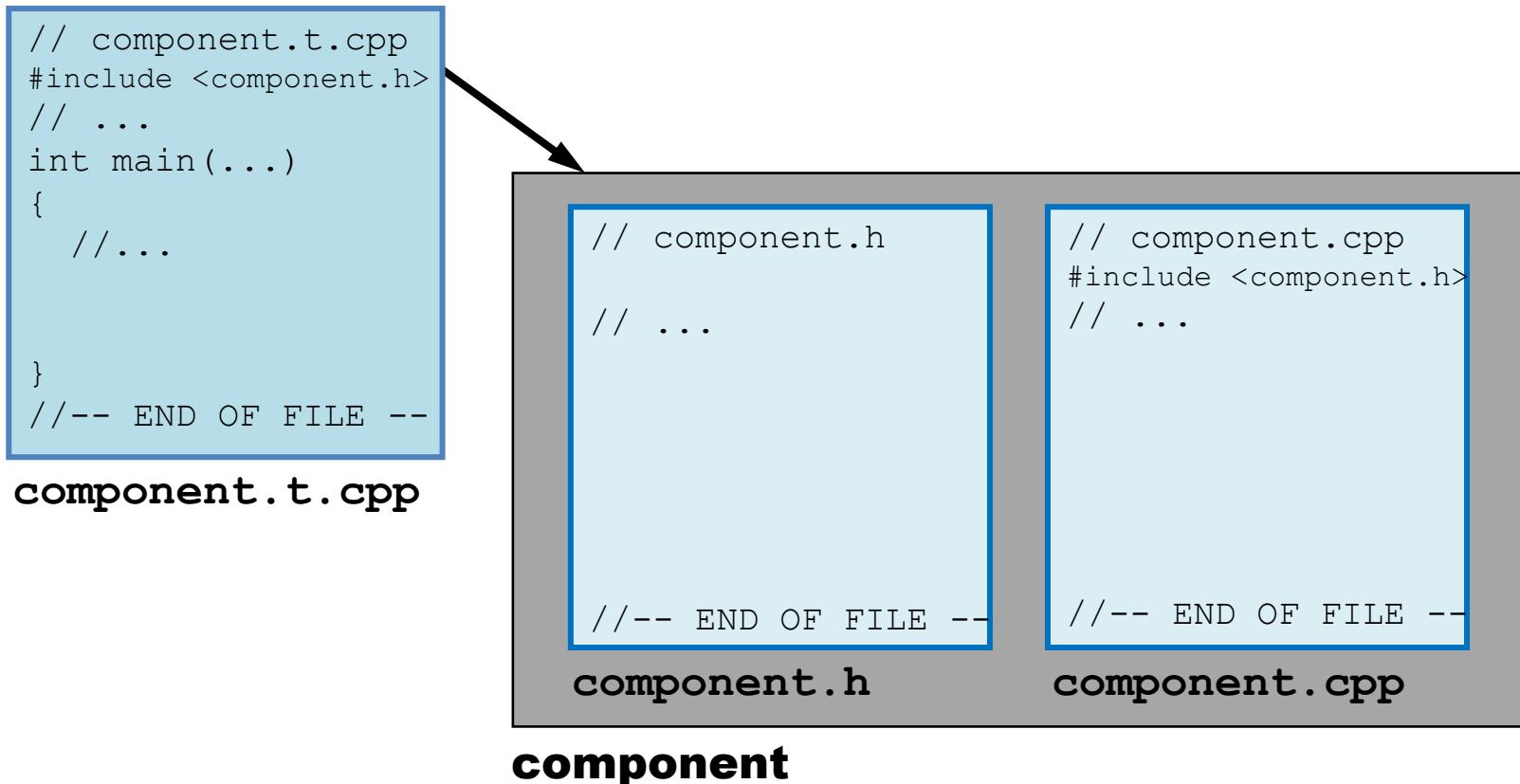
*Logical* content aggregated into a  
*Physical* hierarchy of **components**



## 1. Brief Review of Physical Design

# *Component: Uniform Physical Structure*

## A Component Is Physical



## 1. Brief Review of Physical Design

# *Component: Uniform Physical Structure*

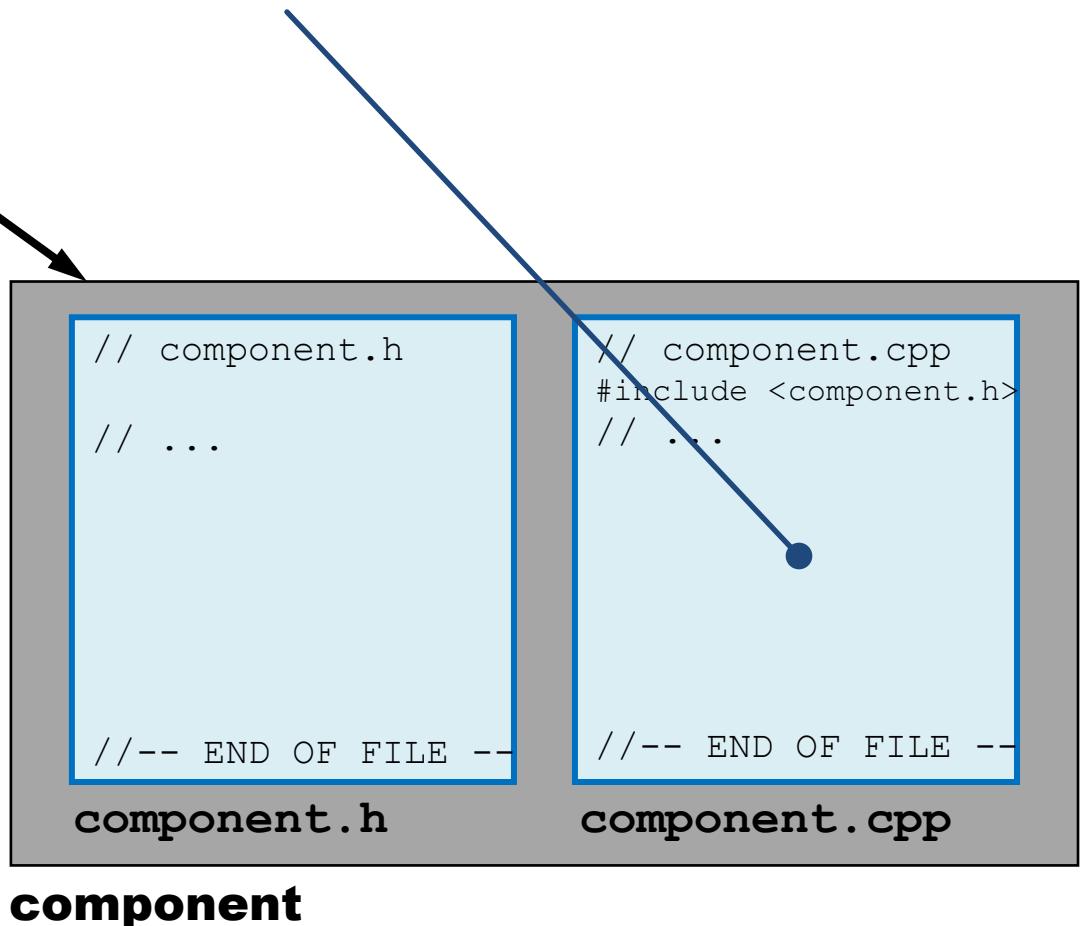
### Implementation

```
// component.t.cpp
#include <component.h>
// ...
int main(...)

{
    //...
}

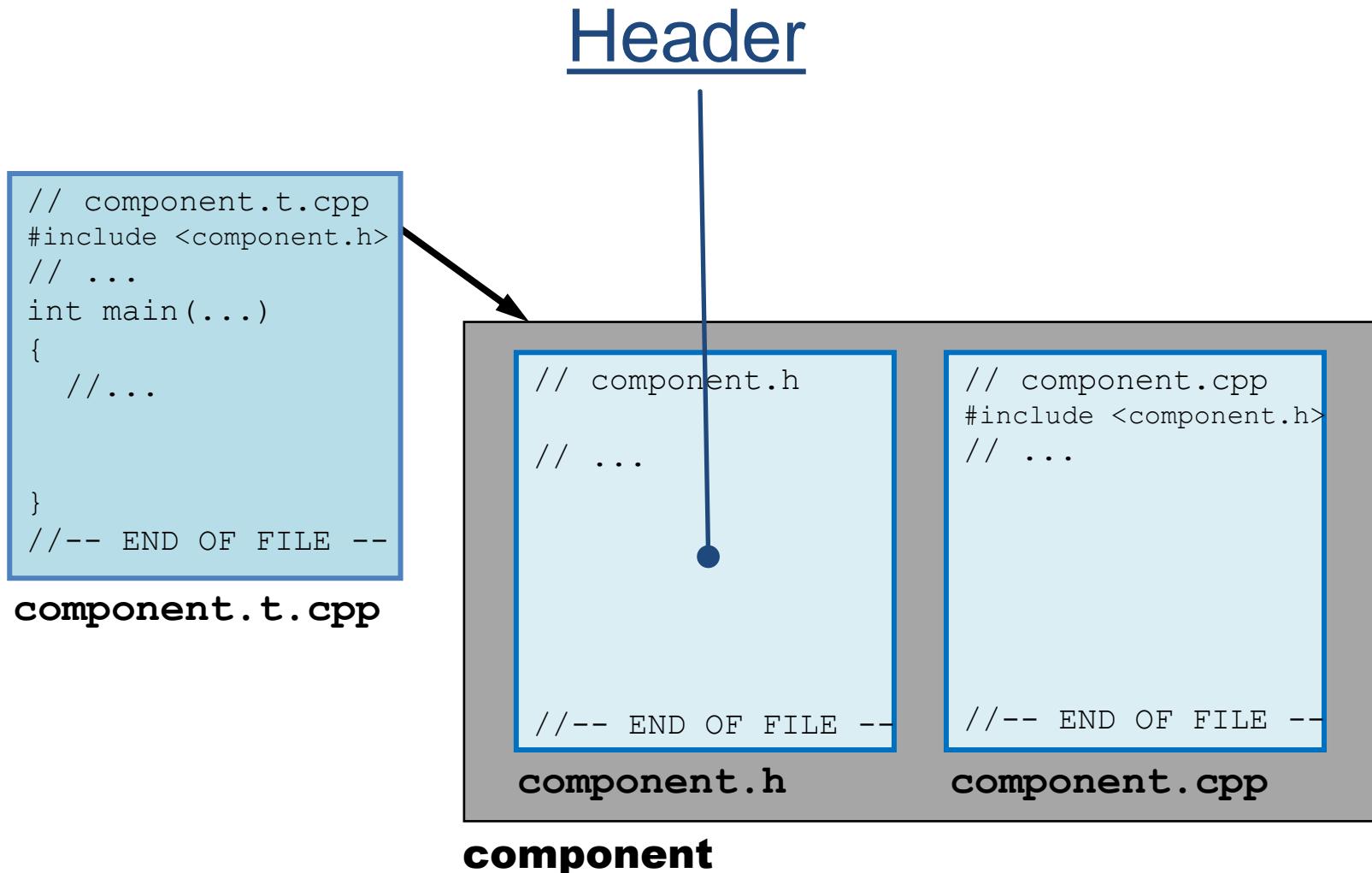
//-- END OF FILE --
```

**component.t.cpp**



## 1. Brief Review of Physical Design

# *Component: Uniform Physical Structure*



## 1. Brief Review of Physical Design

# *Component: Uniform Physical Structure*

### Test Driver

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
    //...
}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h
```

```
// ...
```

```
//-- END OF FILE --
```

**component.h**

```
// component.cpp
```

```
#include <component.h>
// ...
```

```
//-- END OF FILE --
```

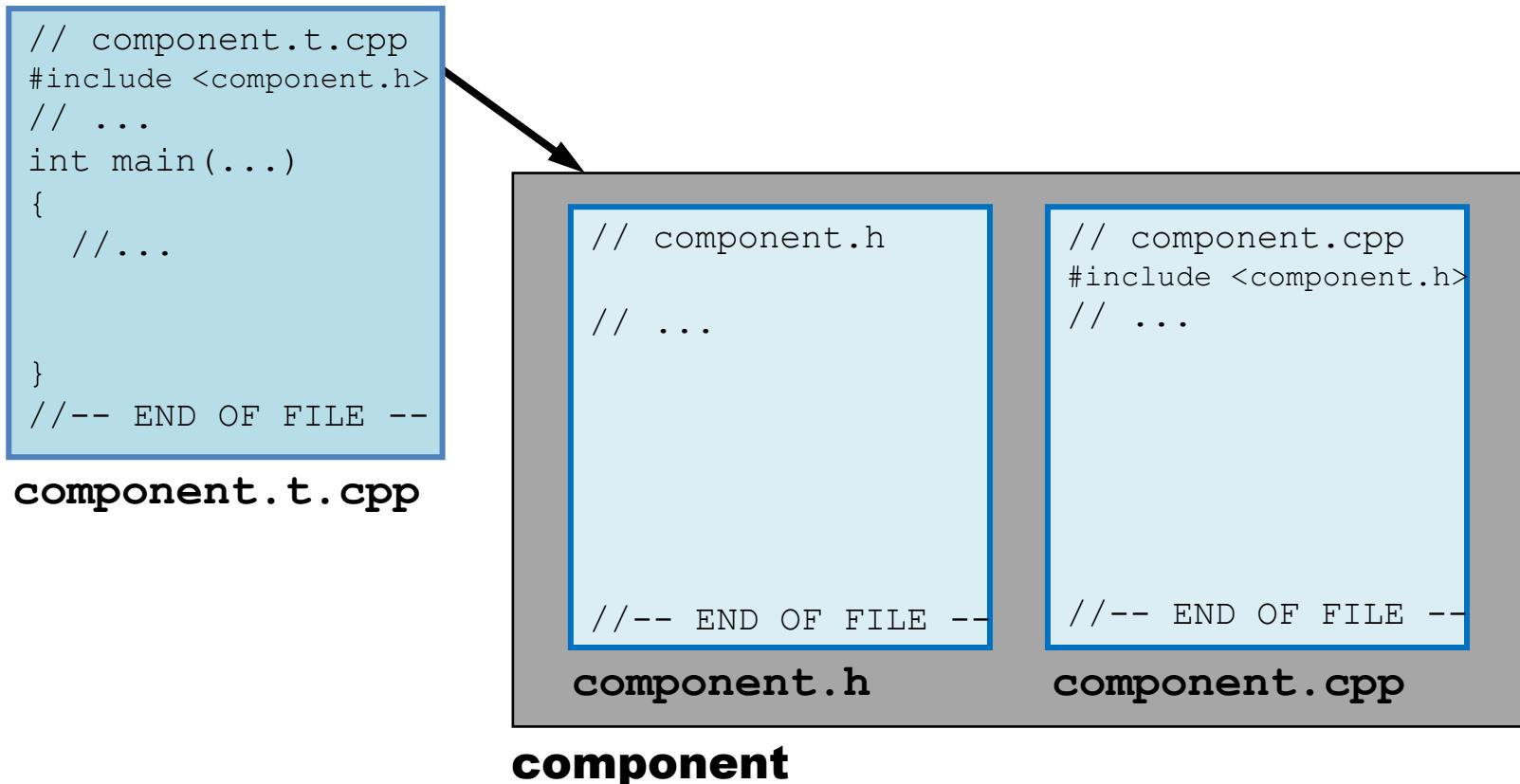
**component.cpp**

### **component**

## 1. Brief Review of Physical Design

# *Component: Uniform Physical Structure*

## The Fundamental Unit of Design



## 1. Brief Review of Physical Design

# Logical Relationships

**PointList**

**PointList\_Link**

**Polygon**

**Point**

**Shape**

## 1. Brief Review of Physical Design

# Logical Relationships

PointList

PointList\_Link

Polygon

Point

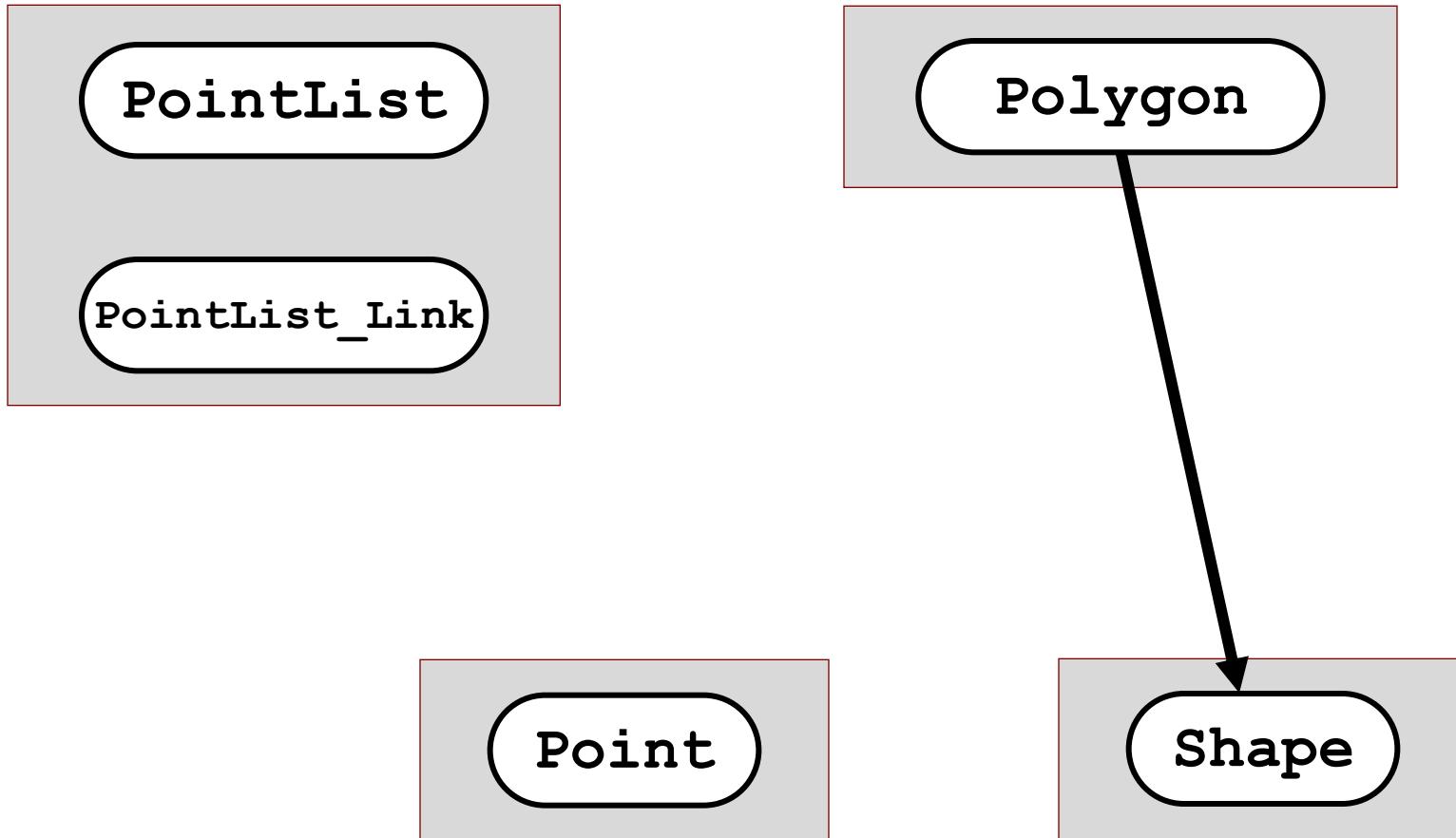
Shape



Is-A

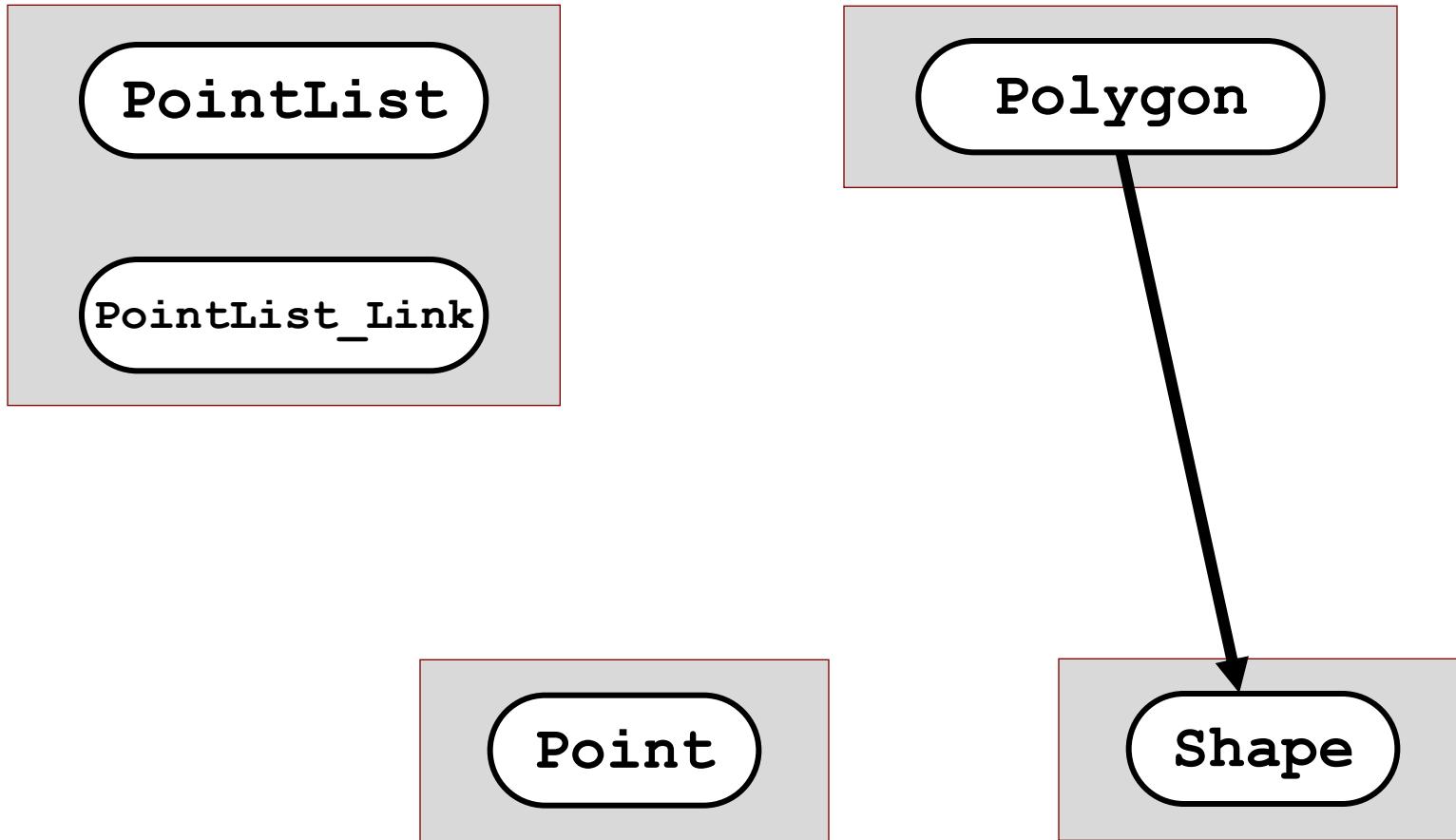
## 1. Brief Review of Physical Design

# Logical Relationships



## 1. Brief Review of Physical Design

# Logical Relationships



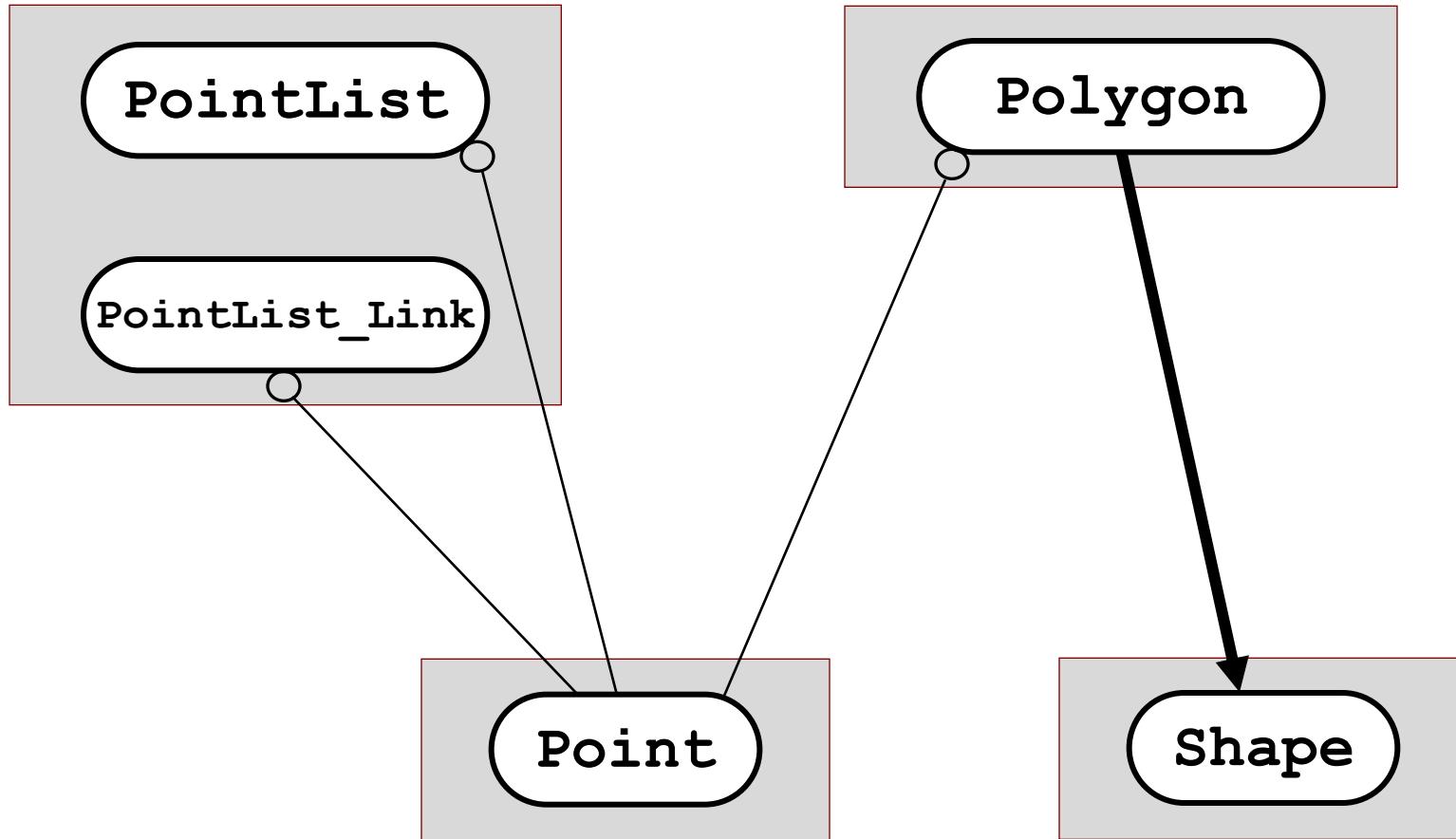
Uses-in-the-Interface



Is-A

## 1. Brief Review of Physical Design

# Logical Relationships

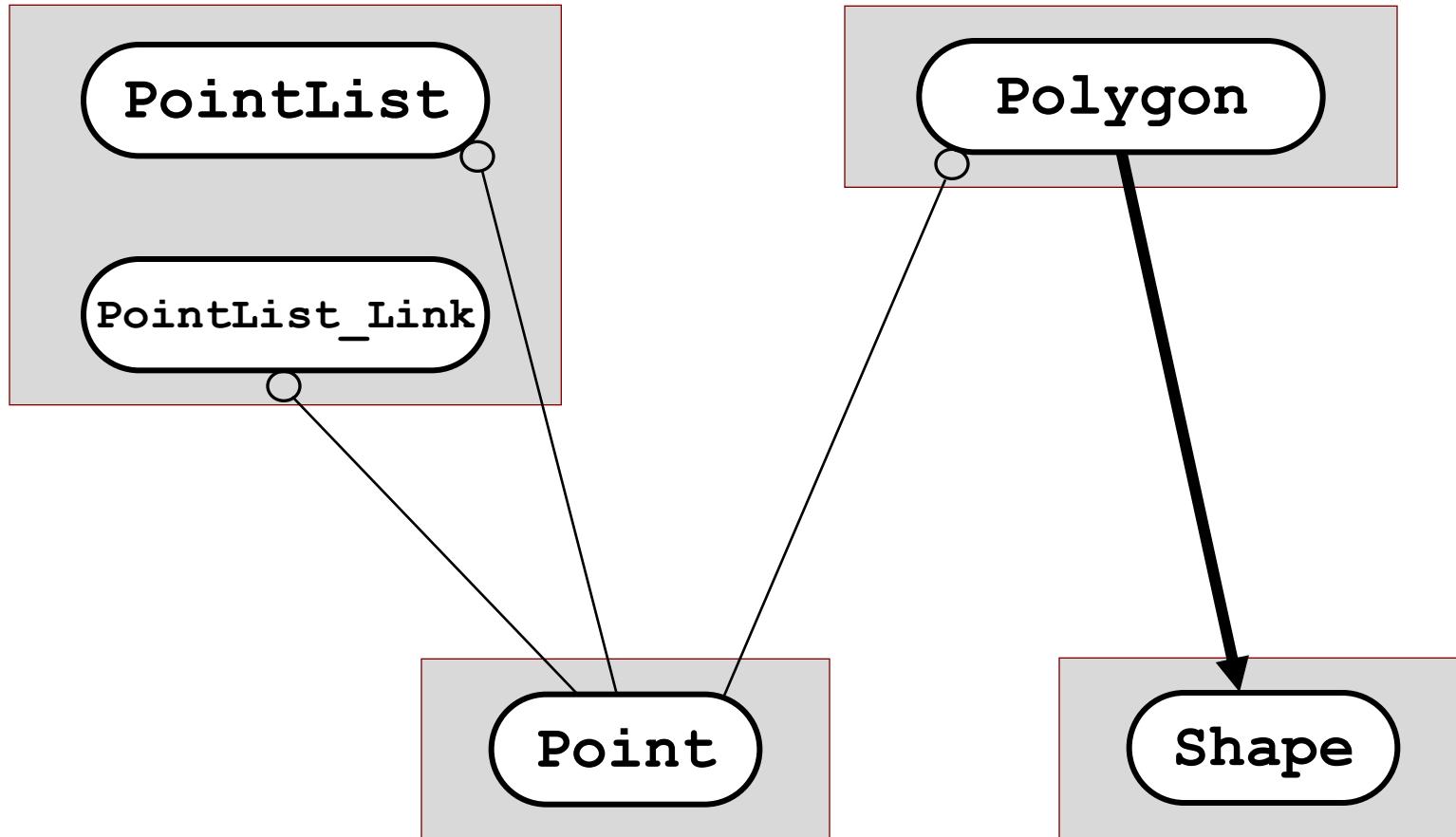


○ ————— Uses-in-the-Interface

————→ Is-A

## 1. Brief Review of Physical Design

# Logical Relationships



○ —————

Uses-in-the-Interface

● —————

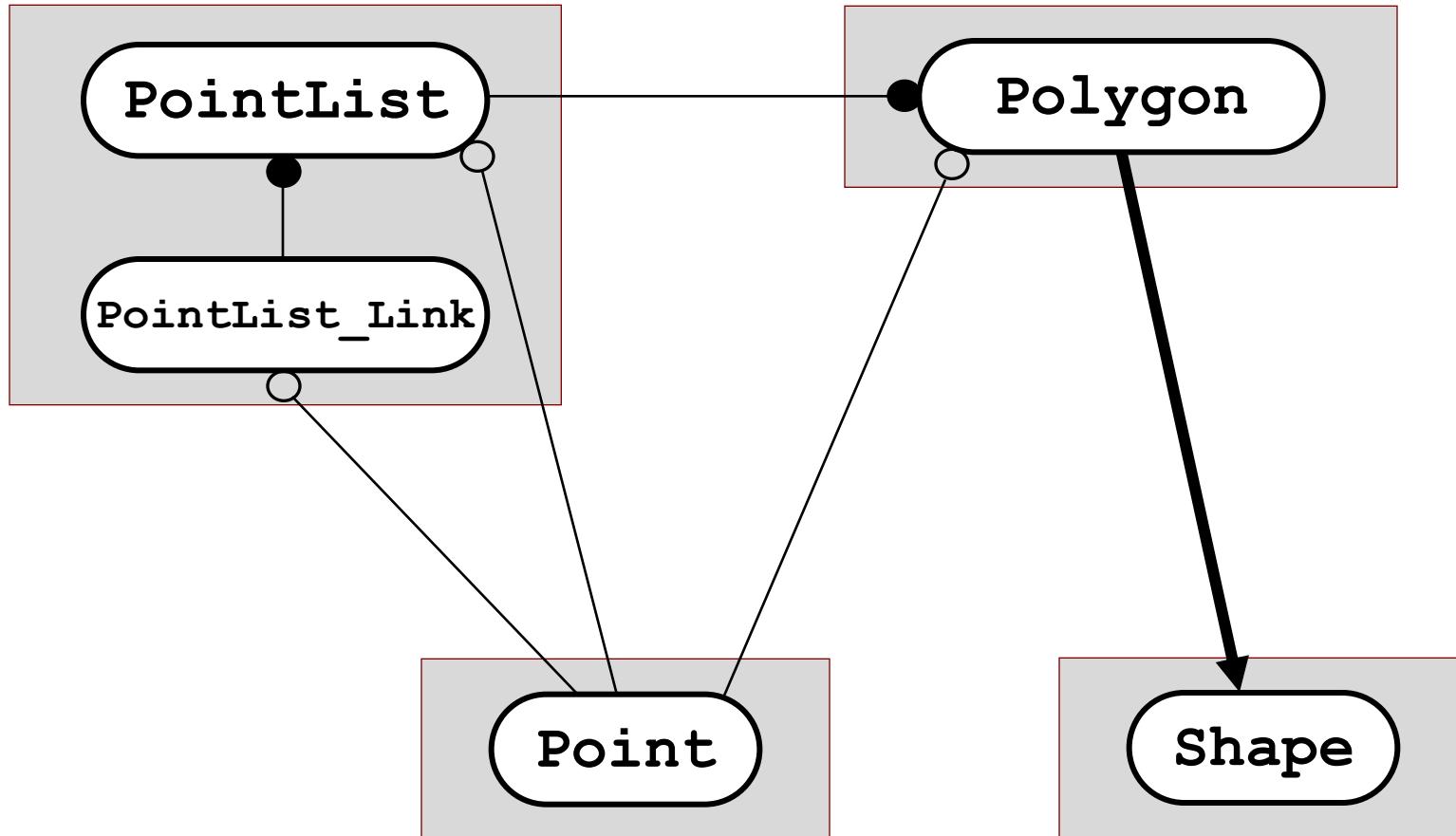
Uses-in-the-Implementation

————→

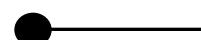
Is-A

## 1. Brief Review of Physical Design

# Logical Relationships



Uses-in-the-Interface



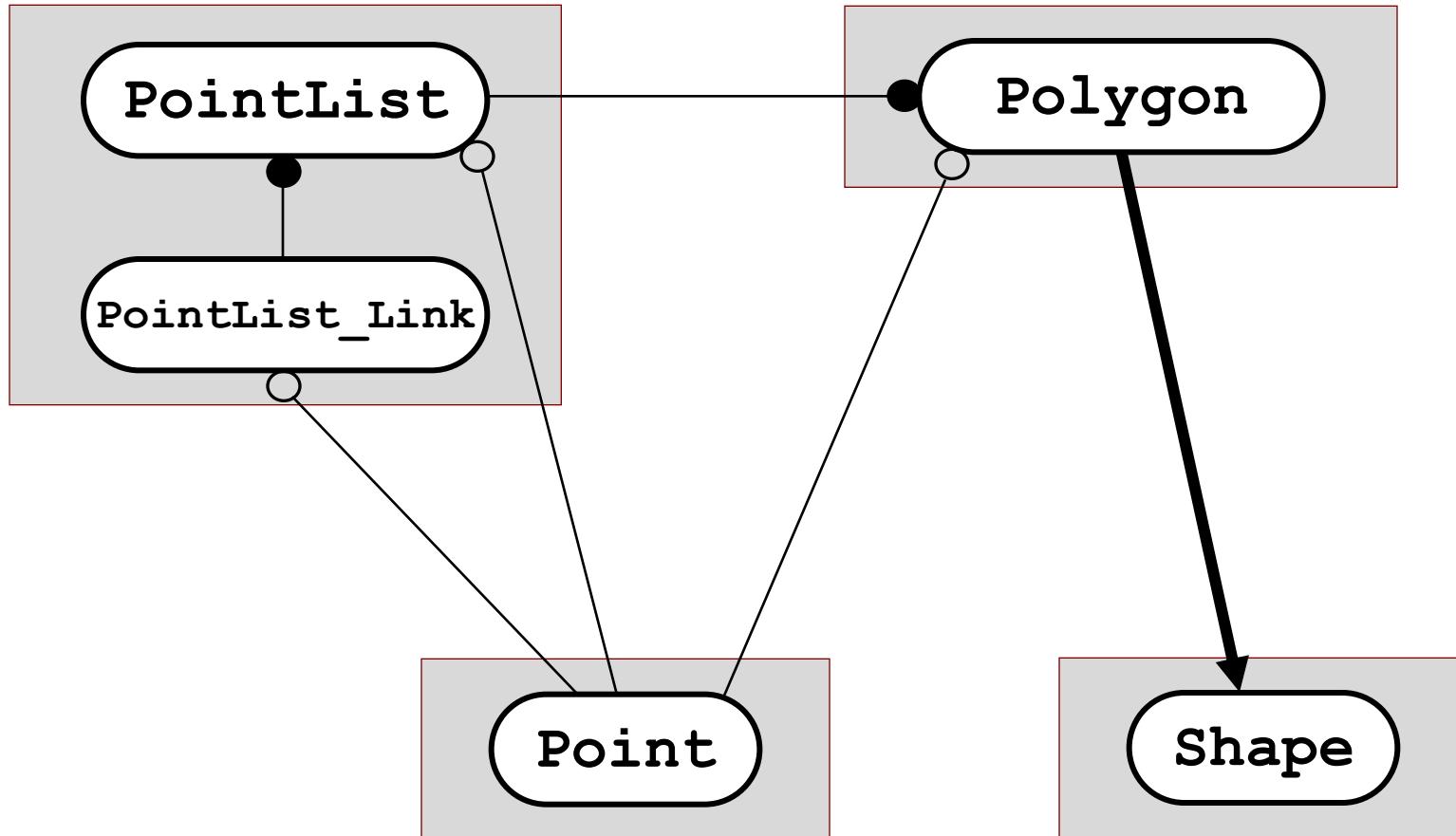
Uses-in-the-Implementation



Is-A

## 1. Brief Review of Physical Design

# Logical Relationships

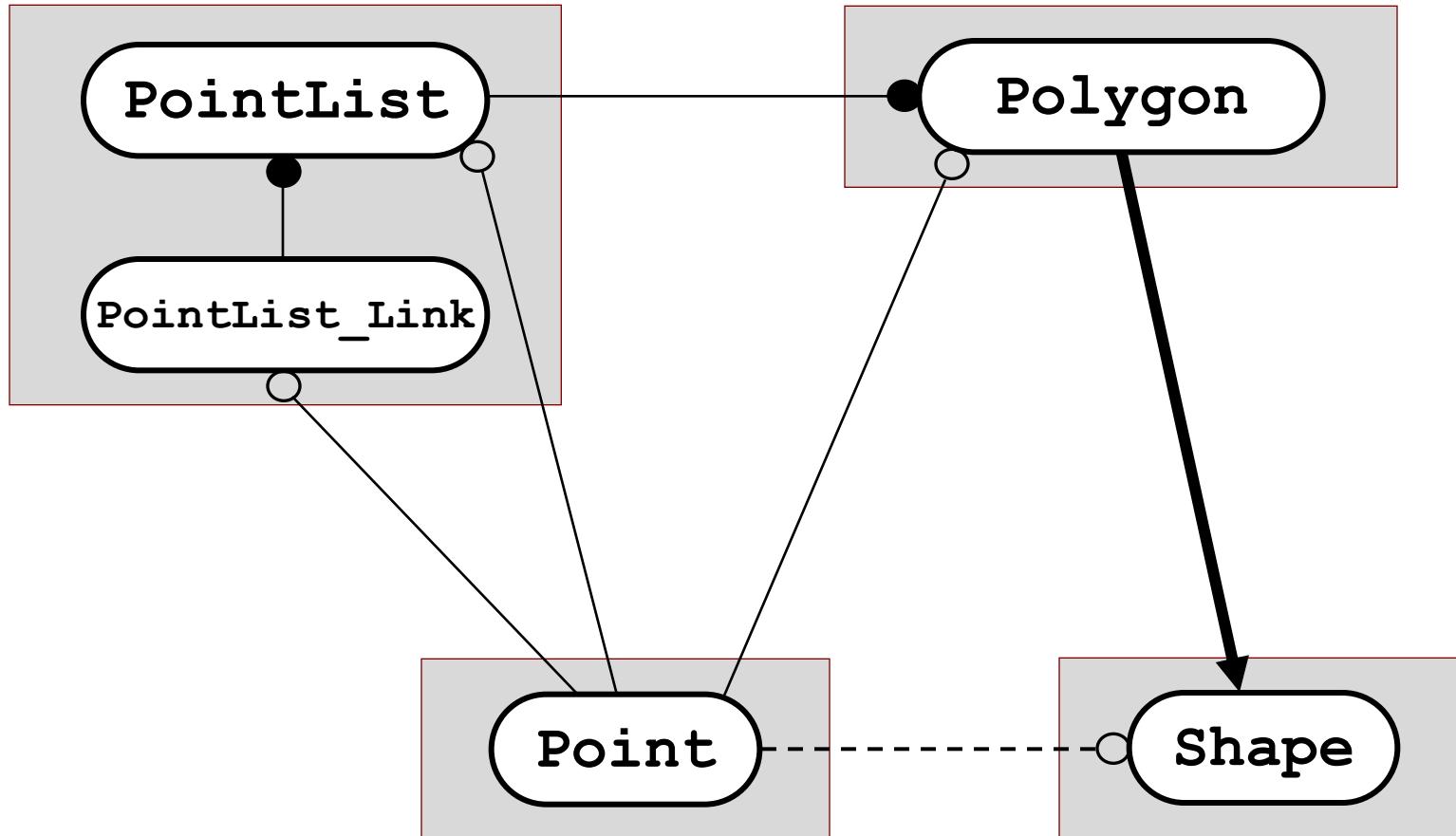


○— Uses-in-the-Interface  
●— Uses-in-the-Implementation

○----- Uses in name only  
→ Is-A

## 1. Brief Review of Physical Design

# Logical Relationships

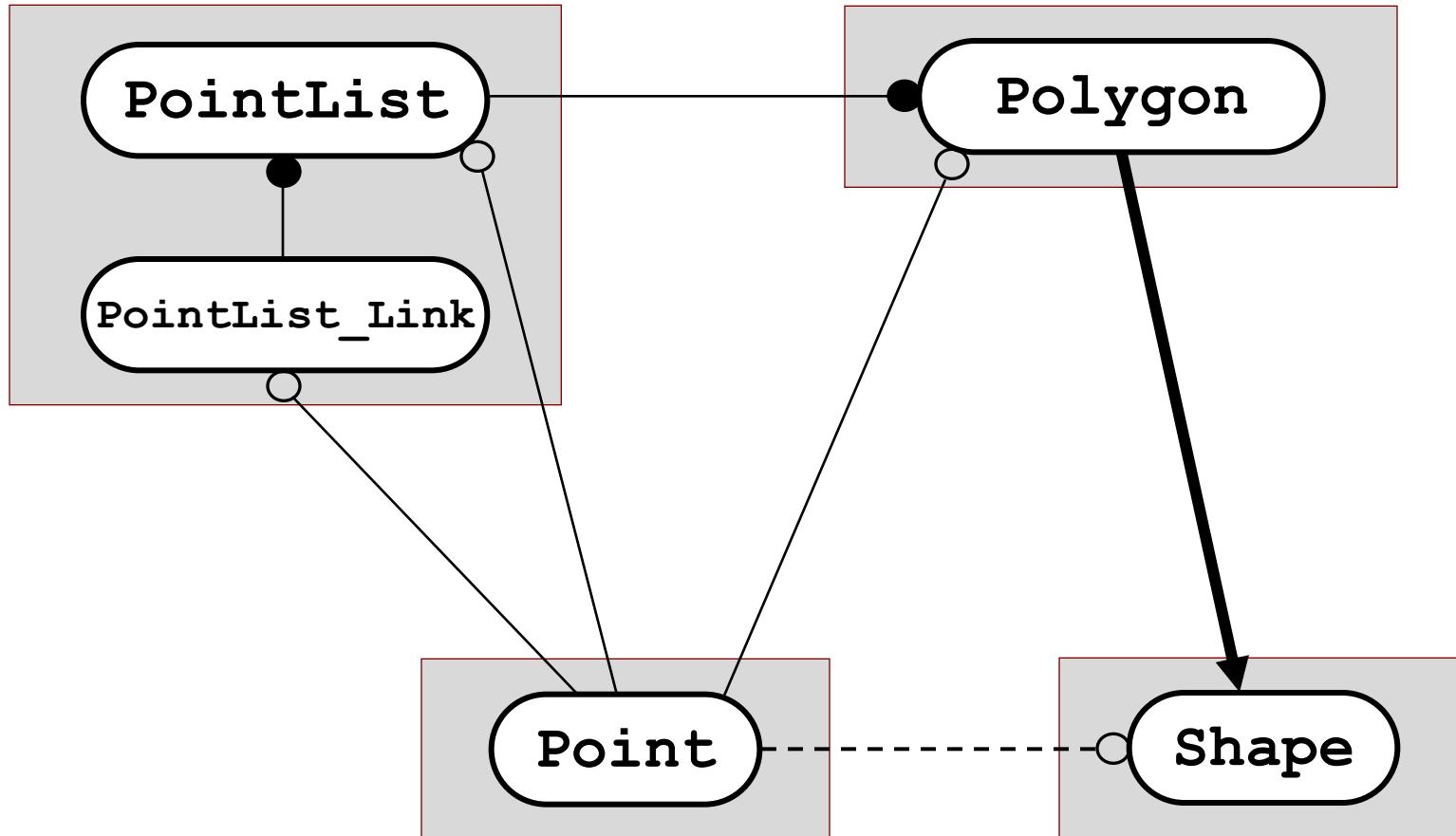


○————— Uses-in-the-Interface  
●————— Uses-in-the-Implementation

○ - - - - - Uses in name only  
———— Is-A

## 1. Brief Review of Physical Design

# Implied Dependency

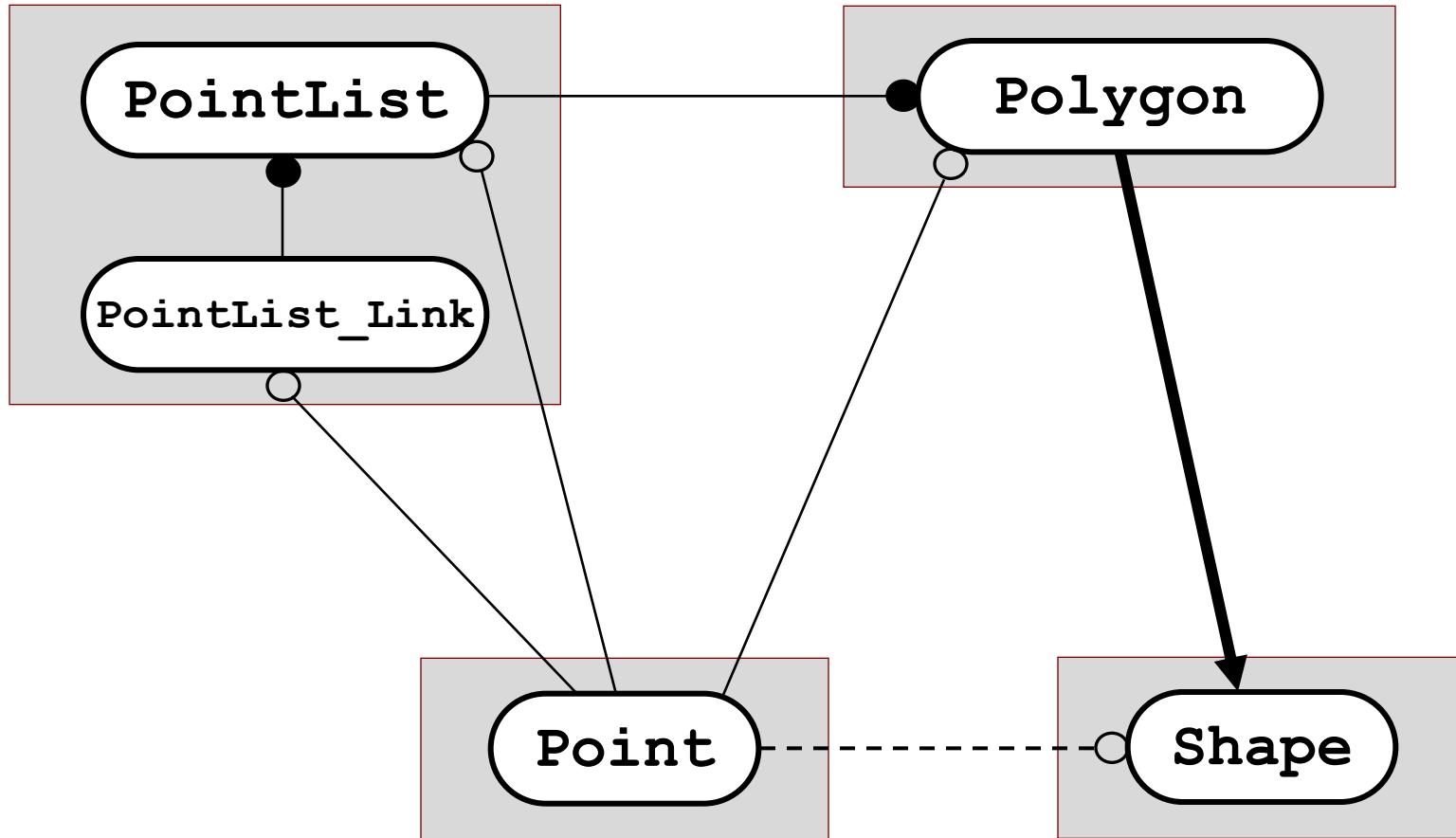


—○— Uses-in-the-Interface  
—●— Uses-in-the-Implementation

○----- Uses in name only  
→ Is-A

## 1. Brief Review of Physical Design

# Implied Dependency

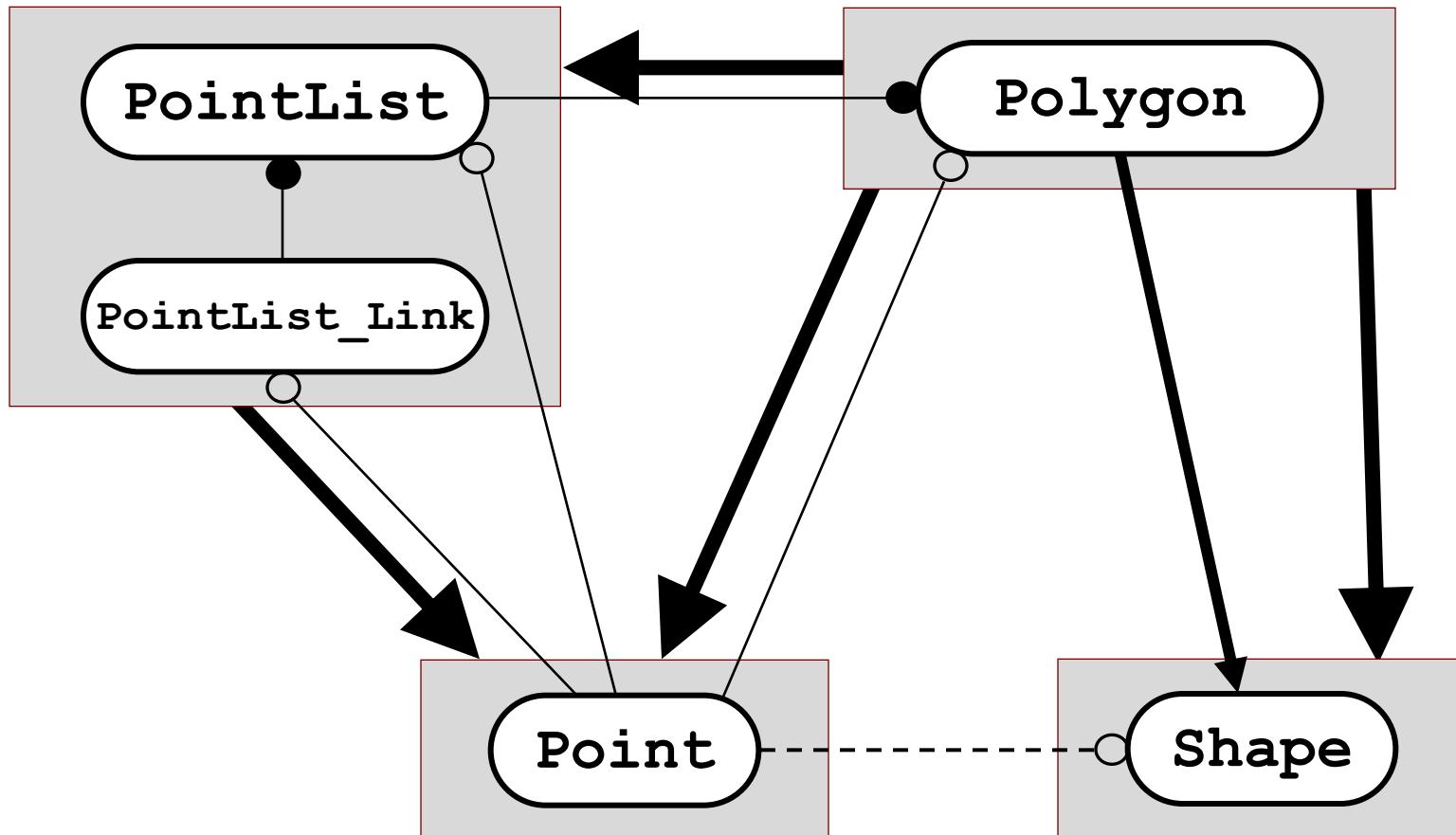


○— Uses-in-the-Interface  
●— Uses-in-the-Implementation

→ Depends-On  
○--- Uses in name only  
→ Is-A

## 1. Brief Review of Physical Design

# Implied Dependency

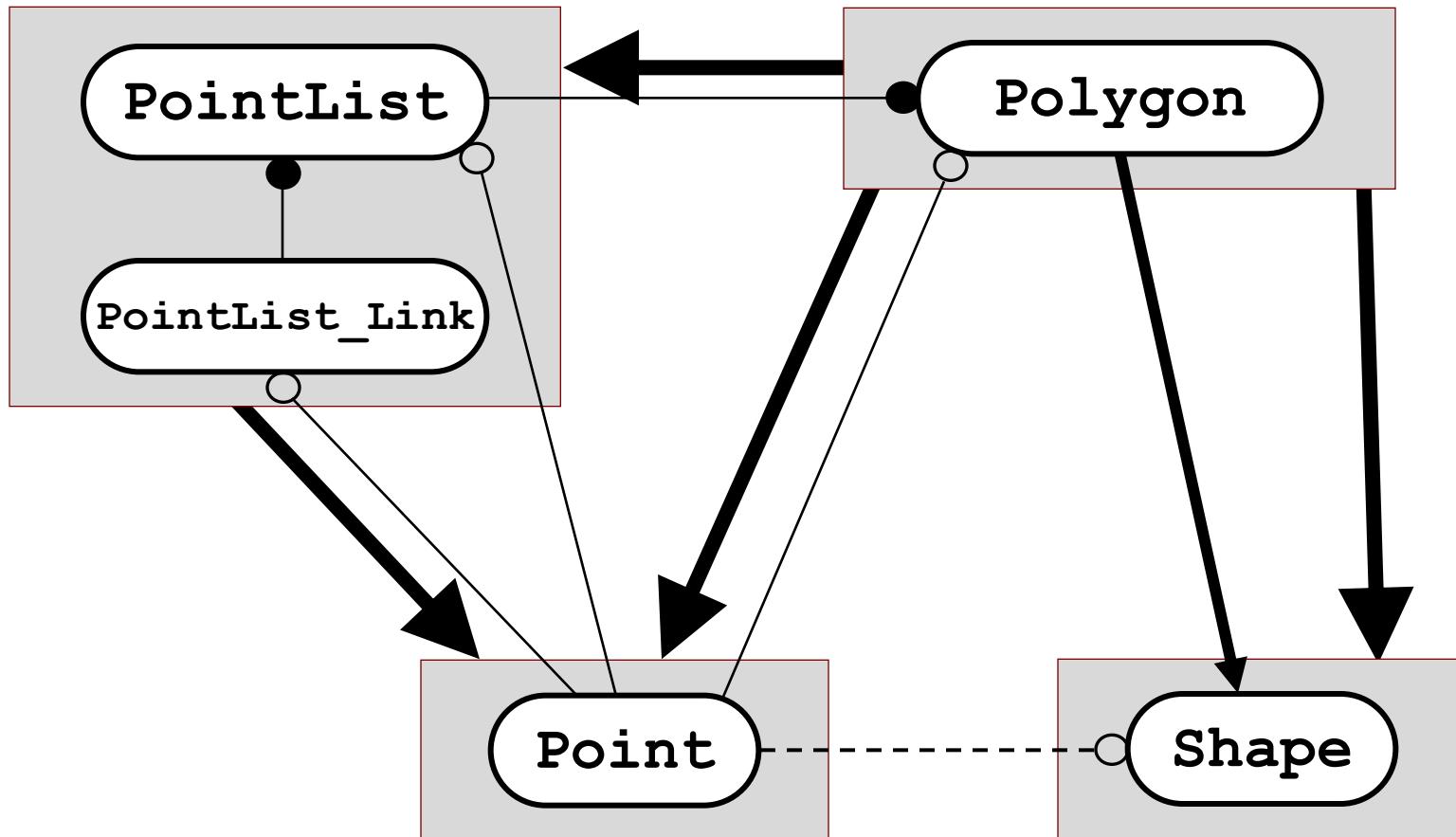


○—> Uses-in-the-Interface  
●—> Uses-in-the-Implementation

→ Depends-On  
○-----> Uses in name only  
→ Is-A

# 1. Brief Review of Physical Design

## Level Numbers

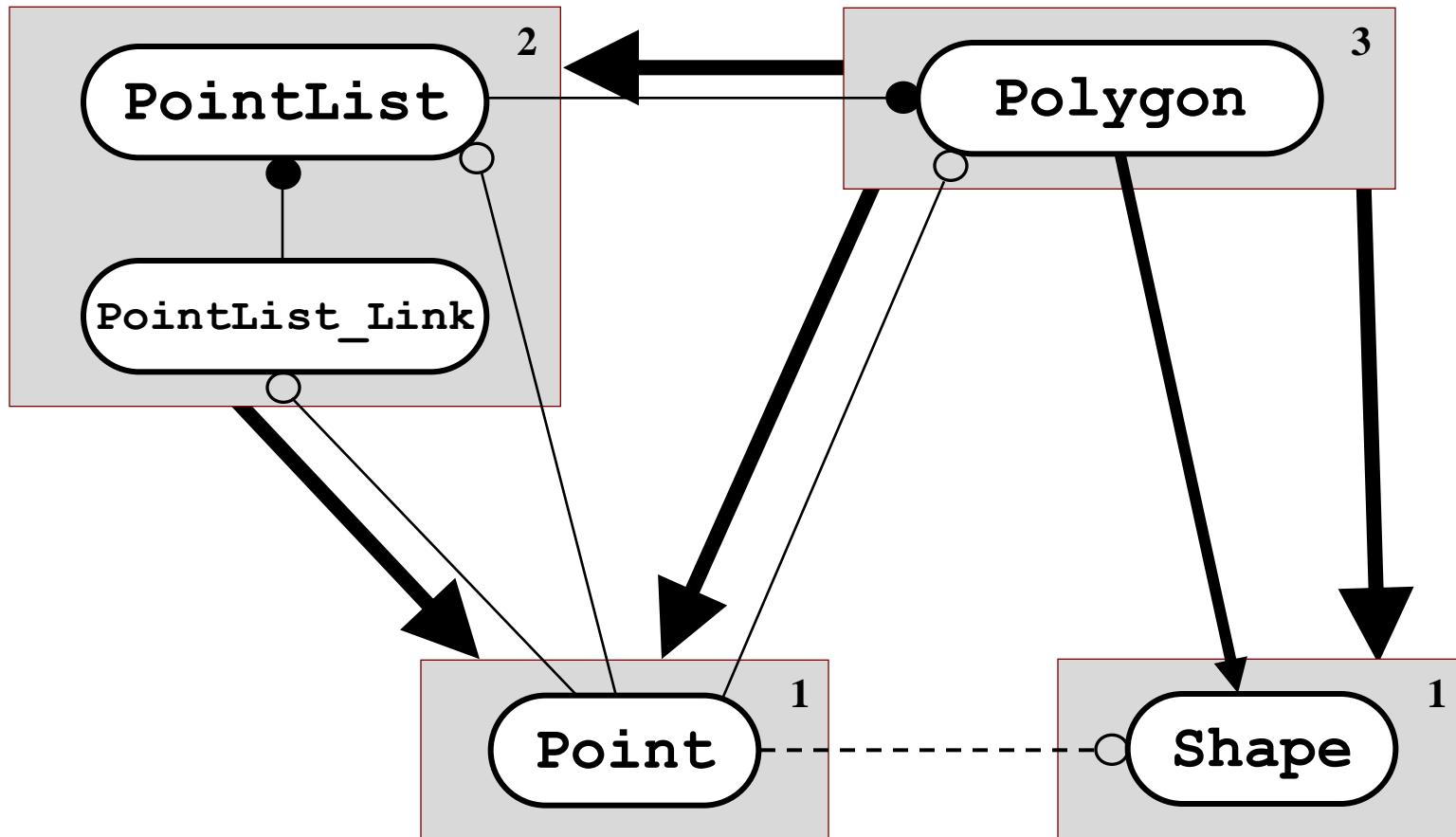


○—→ Uses-in-the-Interface  
●—→ Uses-in-the-Implementation

→ Depends-On  
○----- Uses in name only  
→ Is-A

# 1. Brief Review of Physical Design

## Level Numbers



○— Uses-in-the-Interface  
●— Uses-in-the-Implementation

→ Depends-On  
○--- Uses in name only  
→ Is-A

1. Brief Review of Physical Design

End of Section

Questions?

## 1. Brief Review of Physical Design

# What Questions are we Answering?

- What distinguishes *Logical* from *Physical* Design?
- How do we infer physical relationships (*Depends-On*) from logical relationships (e.g., *Is-A* and *Uses*)?
- What are *level numbers*?

# Outline

1. Brief Review of Physical Design
2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior
3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)
4. Implementing Defensive Checks

Using the **bsls\_assert** Component
5. Negative Testing

Using the **bsls\_asserttest** Component

# Outline

1. Brief Review of Physical Design
2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior
3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)
4. Implementing Defensive Checks

Using the `bsls_assert` Component
5. Negative Testing

Using the `bsls_asserttest` Component

## 2. Interfaces and Contracts

# Interfaces and Contracts

What do we mean by *Interface* versus *Contract* for

- A *Function*?
- A *Class*?
- A *Component*?

## 2. Interfaces and Contracts

# Interfaces and Contracts Function

```
std::ostream& print(std::ostream& stream,  
                    int           level      = 0,  
                    int           spacesPerLevel = 4) const;
```

## 2. Interfaces and Contracts

# Interfaces and Contracts Function

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;
```

Types Used  
In the Interface

## 2. Interfaces and Contracts

# Interfaces and Contracts Function

```
std::ostream& print(std::ostream& stream,
                     int           level          = 0,
                     int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

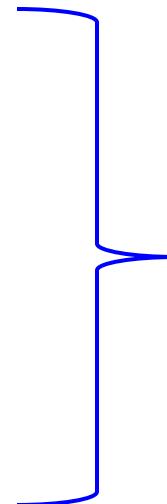
# Interfaces and Contracts Class

```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```

## 2. Interfaces and Contracts

# Interfaces and Contracts Class

```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```



Public  
Interface

## 2. Interfaces and Contracts

# Interfaces and Contracts Class

```
class Date {  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
  
    Date(const Date& original);  
  
    // ...  
};
```

## 2. Interfaces and Contracts

# Interfaces and Contracts Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
  
        Date(const Date& original);  
  
        // ...  
};
```

## 2. Interfaces and Contracts

# Interfaces and Contracts Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
  
    // ...  
};
```

## 2. Interfaces and Contracts

# Interfaces and Contracts Class

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
  
    // ...  
};
```

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

## Component

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

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

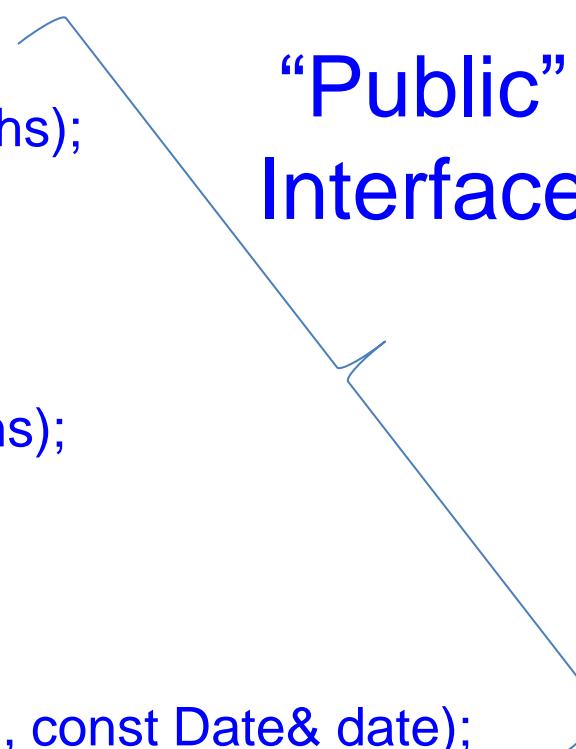
```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

## Component

- bool operator==(const Date& lhs, const Date& rhs);
  - bool operator!=(const Date& lhs, const Date& rhs);
  - std::ostream& operator<<(std::ostream& stream, const Date& date);
- 
- “Public” Interface

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

# Component

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

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

```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

# Component

```
bool operator==(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
// value, and 'false' otherwise. Two 'Date' objects have the same  
// value if their respective 'year', 'month', and 'day' attributes  
// have the same value.
```

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

```
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

# Component

```
bool operator==(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
// value, and 'false' otherwise. Two 'Date' objects have the same  
// value if their respective 'year', 'month', and 'day' attributes  
// have the same value.  
  
bool operator!=(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates do not have the  
// same value, and 'false' otherwise. Two 'Date' objects do not have  
// the same value if any of their respective 'year', 'month', or 'day'  
// attributes do not have the same value.  
  
std::ostream& operator<<(std::ostream& stream, const Date& date);
```

## 2. Interfaces and Contracts

# Interfaces and Contracts

```
class Date {  
    // ...  
public:  
    // ...  
};
```

# Component

```
bool operator==(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates have the same  
// value, and 'false' otherwise. Two 'Date' objects have the same  
// value if their respective 'year', 'month', and 'day' attributes  
// have the same value.  
  
bool operator!=(const Date& lhs, const Date& rhs);  
// Return 'true' if the specified 'lhs' and 'rhs' dates do not have the  
// same value, and 'false' otherwise. Two 'Date' objects do not have  
// the same value if any of their respective 'year', 'month', or 'day'  
// attributes do not have the same value.  
  
std::ostream& operator<<(std::ostream& stream, const Date& date);  
// Format the value of the specified 'date' object to the specified  
// output 'stream' as 'yyyy/mm/dd', and return a reference to 'stream'.  
58
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

Precondition

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);  
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

## Precondition

For a Stateless Function:  
Restriction on *syntactically legal* inputs.

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
```

```
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
```

```
// Return the square root of the specified
// 'value'. The behavior is undefined unless
// '0 <= value'.
```

**Postcondition**

## 2. Interfaces and Contracts

# Preconditions and Postconditions Function

```
double sqrt(double value);
```

```
// Return the square root of the specified  
// 'value'. The behavior is undefined unless  
// '0 <= value'.
```

## Postcondition

For a Stateless Function:  
What it “returns”.

## 2. Interfaces and Contracts

# Preconditions and Postconditions Object Method

## 2. Interfaces and Contracts

# Preconditions and Postconditions Object Method

- Preconditions: What must be true of both (object) state and method inputs; otherwise the behavior is undefined.

## 2. Interfaces and Contracts

# Preconditions and Postconditions Object Method

- Preconditions: What must be true of both (object) state and method inputs; otherwise the behavior is undefined.
  
- Postconditions: What must happen as a function of (object) state and input if all Preconditions are satisfied.

## 2. Interfaces and Contracts

# Preconditions and Postconditions Object Method

a.k.a.  
*Essential  
Behavior*

- Preconditions: What must be true of both (object) state and input if all Preconditions are satisfied.
- Postconditions: What must happen as a function of (object) state and input if all Preconditions are satisfied.

Note that *Essential Behavior* refers to a superset of *Postconditions* that includes behavioral guarantees, such as runtime complexity.

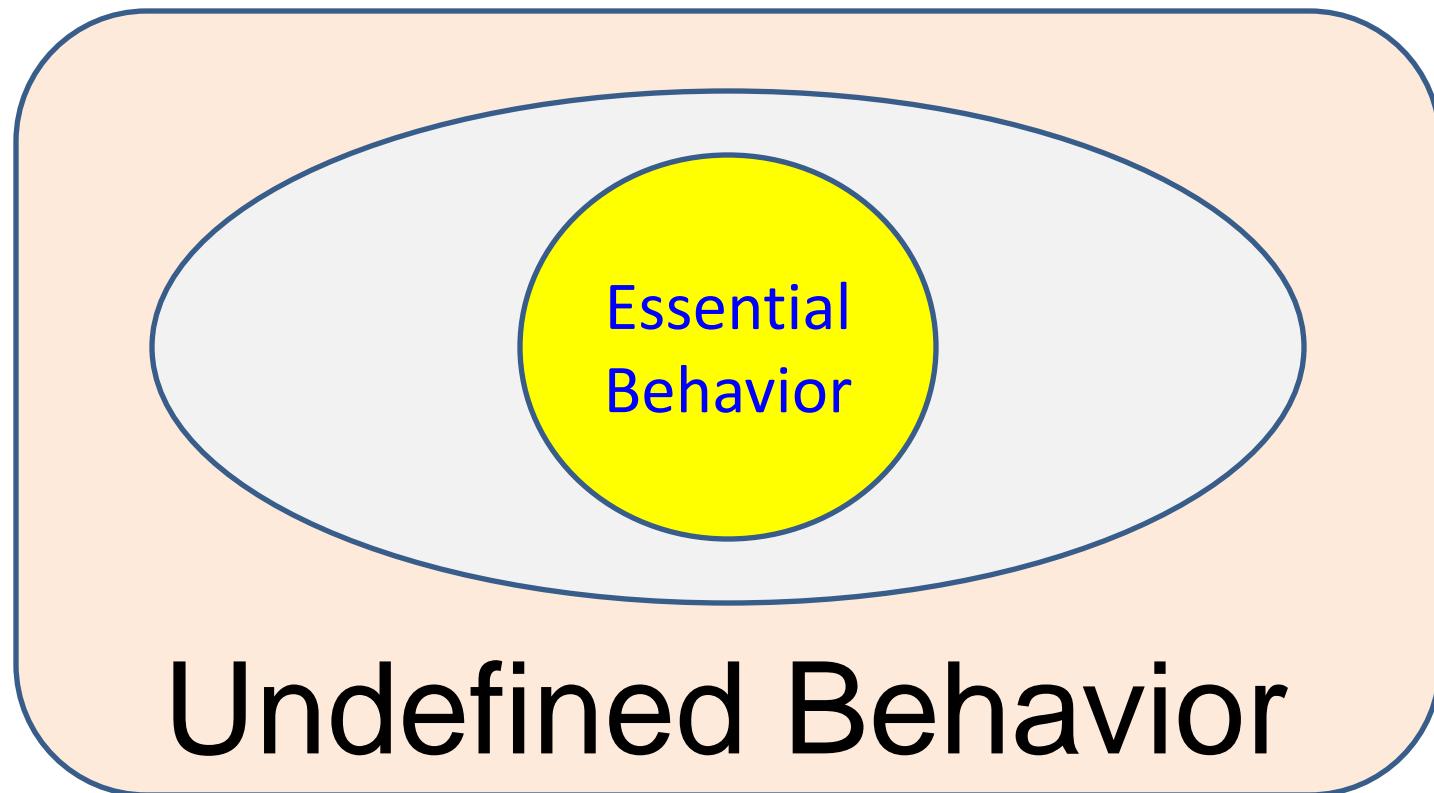
a.k.a.  
*Essential  
Behavior*

- Preconditions: What must happen as a function of (object) state and input if all Postconditions are satisfied.
- Postconditions: What must happen as a function of (object) state and input if all Preconditions are satisfied.

Observation By  
**Kevlin Henny**

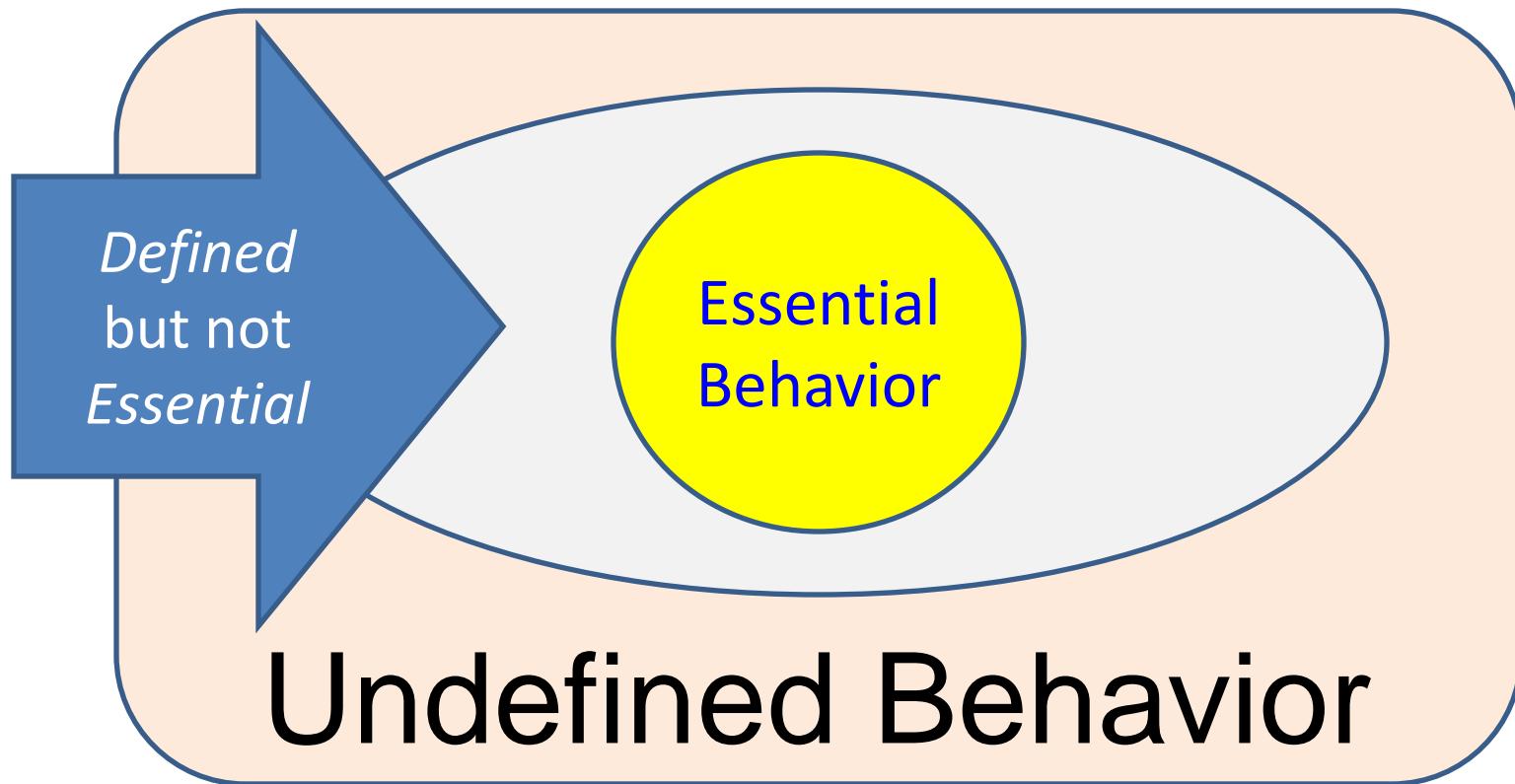
## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior



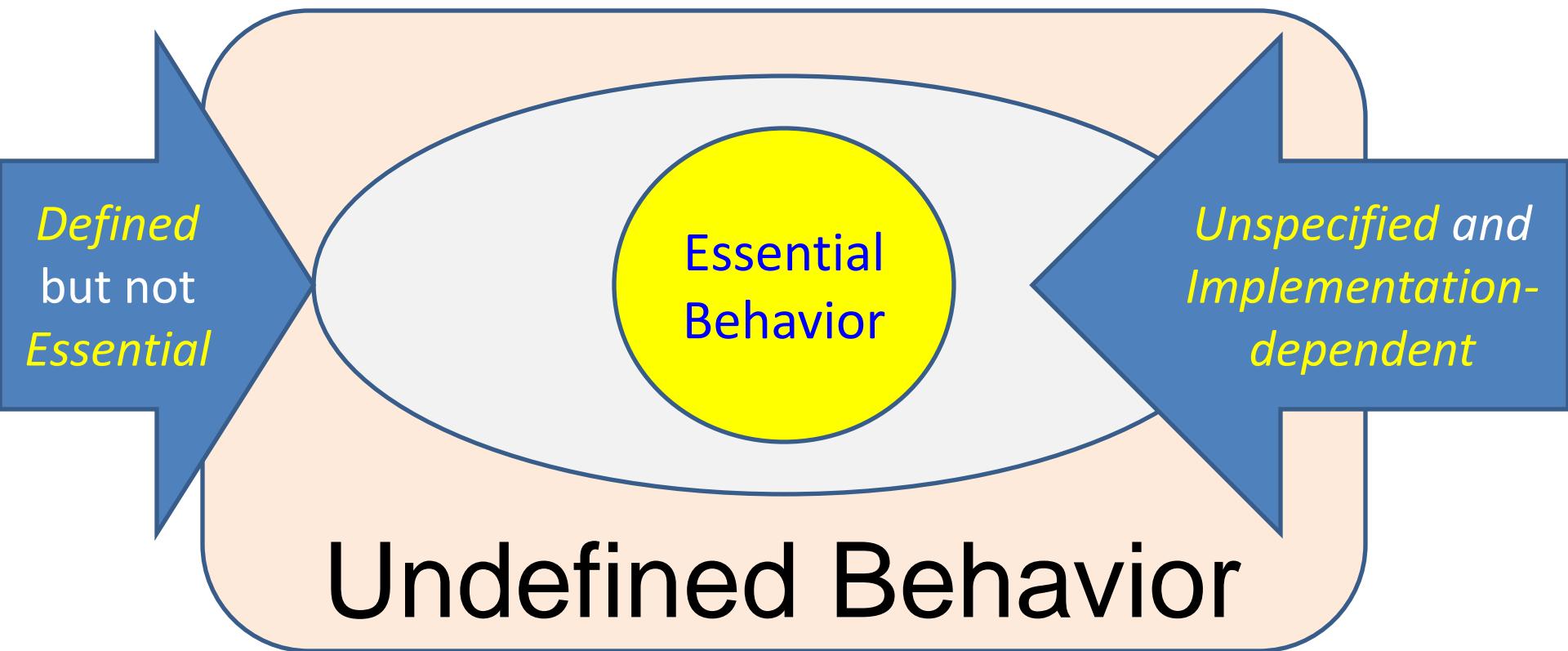
## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior



## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior



## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,
                     int           level        = 0,
                     int           spacesPerLevel = 4) const;
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
std::ostream& print(std::ostream& stream,  
                    int          level      = 0,  
                    int          spacesPerLevel = 4) const;  
  
// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.
```

# Preconditions and Postconditions Defined & Essential

Any  
Undefined  
Behavior?

```
std::ostream& print(std::ostream& stream,
```

|     |                |             |
|-----|----------------|-------------|
| int | level          | = 0,        |
| int | spacesPerLevel | = 4) const; |

```
// Format this object to the specified output 'stream' at the (absolute
// value of) the optionally specified indentation 'level', and return a
// reference to 'stream'. If 'level' is specified, optionally specify
// 'spacesPerLevel', the number of spaces per indentation level for
// this and all of its nested objects. If 'level' is negative,
// suppress indentation of the first line. If 'spacesPerLevel' is
// negative, format the entire output on one line, suppressing all but
// the initial indentation (as governed by 'level'). If 'stream' is
// not valid on entry, this operation has no effect.
```

# Preconditions and Postconditions Defined & Essential

Any  
Non-Essential  
Behavior?

```
std::ostream& print(std::ostream& stream,  
                    int      level      = 0,  
                    int      spacesPerLevel = 4) const;
```

// Format this object to the specified output 'stream' at the (absolute  
// value of) the optionally specified indentation 'level', and return a  
// reference to 'stream'. If 'level' is specified, optionally specify  
// 'spacesPerLevel', the number of spaces per indentation level for  
// this and all of its nested objects. If 'level' is negative,  
// suppress indentation of the first line. If 'spacesPerLevel' is  
// negative, format the entire output on one line, suppressing all but  
// the initial indentation (as governed by 'level'). If 'stream' is  
// not valid on entry, this operation has no effect.

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/ 'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantics.  
    // a valid date between the dates 0001/01/01  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/month'/day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantics.  
    // a valid date between the dates 0001/01/01  
    // 9999/12/31 inclusive.  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions Defined & Essential Behavior

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
    //...  
public:  
    Date(int year, int month, int day);  
    // Create a valid date from the specified  
    // 'day'. The behavior is undefined if  
    // represents a valid date in the range [0001/01/01, 9999/12/31].  
    Date(const Date& original);  
    // Create a date having the value of the specified 'original' date.  
    // ...  
};
```



## 2. Interfaces and Contracts

# Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
public:  
    Date(int year, int month, int day);  
        // Create a valid date from the specified 'year', 'month', and  
        // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
        // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
    Date(const Date& original);  
        // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
            // Create a valid date from the specified 'year', 'month', and  
            // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
            // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
        Date(const Date& original);  
            // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.
```

```
//...
```

```
p
```

**Question:** Must the code itself preserve invariants even if one or more Preconditions of a method's contract is violated?

```
};
```

e.

## 2. Interfaces and Contracts

# Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
            // Create a valid date from the specified 'year', 'month', and  
            // 'day'. The behavior is undefined unless 'year'/month'/day'  
            // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
        Date(const Date& original);  
            // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

## 2. Interfaces and Contracts

# Preconditions and Postconditions (Object) Invariants

```
class Date {  
    // This class implements a value-semantic type representing  
    // a valid date between the dates 0001/01/01 and  
    // 9999/12/31 inclusive.  
  
    //...  
  
    public:  
        Date(int year, int month, int day);  
            // Create a valid date from the specified 'year', 'month', and  
            // 'day'. The behavior is undefined unless 'year'/month'/day'  
            // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
        Date(const Date& original);  
            // Create a date having the value of the specified 'original' date.  
    // ...  
};
```

**Answer: No!**

## 2. Interfaces and Contracts

What happens  
when behavior  
is undefined  
is undefined!

public:

```
Date(int year, int month, int day);  
    // Create a valid date from the specified 'year', 'month', and  
    // 'day'. The behavior is undefined unless 'year'/'month'/'day'  
    // represents a valid date in the range [0001/01/01 .. 9999/12/31].  
  
Date(const Date& original);  
    // Create a date having the value of the specified 'original' date.  
// ...  
};
```

Postconditions  
variants

semantic type representing  
001/01/01 and

Answer: No!

## 2. Interfaces and Contracts

# Design by Contract (DbC)

“If you give me valid input\*,  
I will behave as advertised;  
otherwise, all bets are off!”

\*including state

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. **What it does.**
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. What it does.
2. **What it returns.**
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. Note that...

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. What it does.
2. What it returns.
- 3. *Essential Behavior.***
- 4. *Undefined Behavior.***
5. Note that...

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. ***Undefined Behavior.***
5. Note that...

## 2. Interfaces and Contracts

# Design by Contract Documentation

There are five aspects:

1. What it does.
2. What it returns.
3. *Essential Behavior.*
4. *Undefined Behavior.*
5. **Note that...**

2. Interfaces and Contracts

# Design by Contract

## Verification

## 2. Interfaces and Contracts

# Design by Contract Verification

➤ **Preconditions:**

## 2. Interfaces and Contracts

# Design by Contract Verification

### ➤ **Preconditions:**

- ✓ RTFM (Read the Manual).

## 2. Interfaces and Contracts

# Design by Contract

## Verification

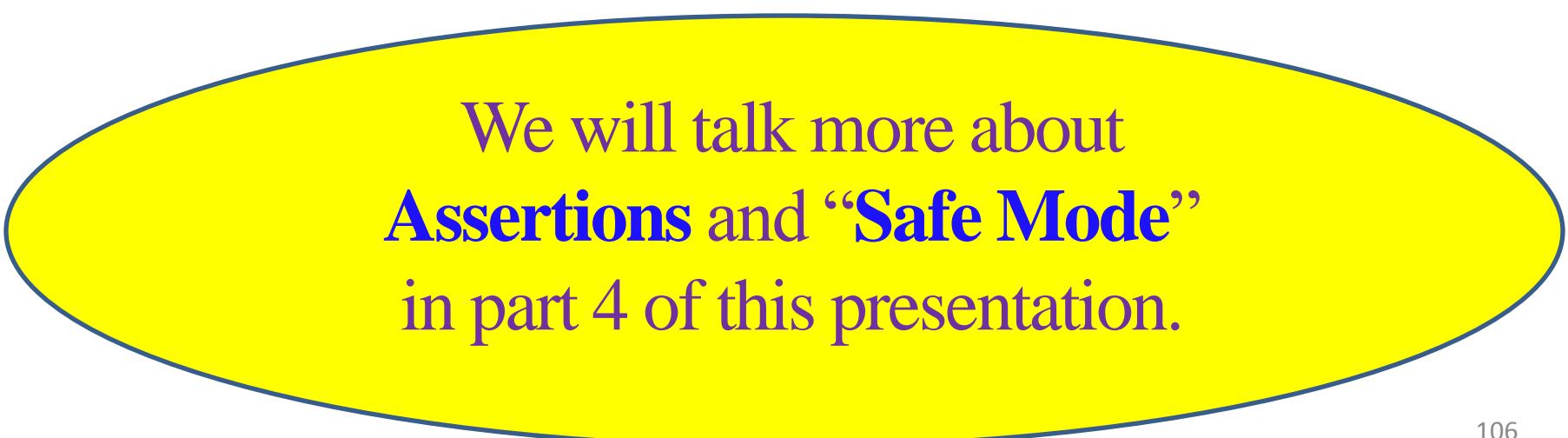
- **Preconditions:**
  - ✓ RTFM (Read the Manual).
  - ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).

## 2. Interfaces and Contracts

# Design by Contract Verification

### ➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).



We will talk more about  
**Assertions** and “**Safe Mode**”  
in part 4 of this presentation.

## 2. Interfaces and Contracts

# Design by Contract Verification

- **Preconditions:**
  - ✓ RTFM (Read the Manual).
  - ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).
- **Postconditions:**

## 2. Interfaces and Contracts

# Design by Contract

## Verification

- **Preconditions:**
  - ✓ RTFM (Read the Manual).
  - ✓ Assert (only in ‘*debug*’ or ‘*safe*’ mode).
- **Postconditions:**
  - ✓ Component-level test drivers.

## 2. Interfaces and Contracts

# Design by Contract Verification

### ➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘debug’ or ‘safe’ mode).

### ➤ Postconditions:

- ✓ Component-level test drivers.

### ➤ Invariants:

## 2. Interfaces and Contracts

# Design by Contract Verification

### ➤ Preconditions:

- ✓ RTFM (Read the Manual).
- ✓ Assert (only in ‘debug’ or ‘safe’ mode).

### ➤ Postconditions:

- ✓ Component-level test drivers.

### ➤ Invariants:

- ✓ Assert invariants in the destructor.

## 2. Interfaces and Contracts

End of Section

# Questions?

## 2. Interfaces and Contracts

# What Questions are we Answering?

- What is the idea behind *Design-by-Contract (DbC)*?
- What do we mean by *Interface* versus *Contract* for a *function*, a *class*, or a *component*?
- What do we mean by *preconditions*, *postconditions*, and *invariants*?
- What do we mean by *essential* & *undefined behavior*?
- Must the code itself preserve invariants even if one or more preconditions of a contract are violated?
- How do we document the contract for a function?
- How do we ensure that postconditions are satisfied?
- How do we test to make sure invariants are preserved?

# Outline

1. Brief Review of Physical Design
2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior
3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)
4. Implementing Defensive Checks

Using the **bsls\_assert** Component
5. Negative Testing

Using the **bsls\_asserttest** Component

# Outline

1. Brief Review of Physical Design

2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)

4. Implementing Defensive Checks

Using the `bsls_assert` Component

5. Negative Testing

Using the `bsls_asserttest` Component

### 3. ‘Good’ Contracts

# Defensive Programming

3. ‘Good’ Contracts

# Defensive Programming (DP)

- What is it?

### 3. ‘Good’ Contracts

## Defensive Programming (DP)

- What is it?

**Redundant Code** that provides runtime checks to detect and report (but not “handle” or “hide”) defects in software.

3. ‘Good’ Contracts

# Defensive Programming (DP)

- What is it?
- Is it Good or Bad?

### 3. 'Good' Contracts

## Defensive Programming (DP)

- What is it?
- Is it Good or Bad?

**Both:** It adds overhead, but can help identify defects early in the development process.

### 3. 'Good' Contracts

# Defensive Programming (DP)

- What is it?
- Is it Good or Bad?
- Which is Better: DP or DbC?

### 3. ‘Good’ Contracts

## Defensive Programming (DP)

- What is it?
- Is it Good or Bad?
- Which is Better: DP or DbC?

Do you ride the bus to school  
or do you take your lunch?

### 3. ‘Good’ Contracts

## Defensive Programming

# What are we defending against?

### 3. ‘Good’ Contracts

## Defensive Programming

What are we defending against?

- Bugs in software  
that we use in our  
implementation?

### 3. ‘Good’ Contracts

## Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?

### 3. 'Good' Contracts

## Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients.

### 3. ‘Good’ Contracts

## Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

### 3. ‘Good’ Contracts

## Defensive Programming

What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

### 3. 'Good' Contracts

## Defensive Programming

### What are we defending against?

- Bugs in software that we use in our implementation?
- Bugs we introduce into our own implementation?
- Misuse by our clients?

### 3. 'Good' Contracts

## Defensive Programming

What are we defending against?

MISUSE BY  
OUR CLIENTS

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

Pejorative terms:

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

Pejorative terms:

- ***Fat***<sub>(not proper inheritance)</sub> Interface

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

### Pejorative terms:

- ***Fat*** (not proper inheritance) Interface
- ***Large*** (not minimal/primitive) Interface

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

# Pejorative terms:

- ***Fat***<sub>(not proper inheritance)</sub> Interface
- ***Large***<sub>(not minimal/primitive)</sub> Interface
- ***Wide***<sub>(no preconditions)</sub> Contract

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

How about it must return 0?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
size_t strlen(const char *s)
{
    if (!s) return 0;           } Wide
    // ...
}
```

How about it must return 0?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
size_t strlen(const char *s)
{
    if (!s) return 0;           } Wide
    // ...
}
```

Likely to mask a defect

How about it must return 0?

### 3. 'Good' Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined behavior:*

```
size_t strlen(const char *s)
{
    ...
    !s, return 0
}
```

**More Code**  
**Runs slowly to mask a defect**  
**More Slow!**  
Wide

How about it must return 0?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

What should happen with the following call?

```
std::size_t x = std::strlen(0);
```

Undefined Behavior

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
size_t strlen(const char *s)
{
    assert(s);
    // ...
}
```

} Narrow

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
size_t strlen(const char *s)
{
    // ...
}
```

}] Narrow

### 3. 'Good' Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
size_t strlen(const char*s)  
{  
    /.../  
}
```

*Just Don't Pass 0!* } Narrow

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

Should

Date:: setDate(int, int, int);

Return a status?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

Should

Date::setDate(int, int, int);

Return a status?

Absolutely  
Not!

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

I “know” this date is valid (It’s my birthday)!

```
date.setDate(3, 8, 59);
```

Therefore, why should I bother to check status?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

I “know” this date is valid (It’s my birthday)!

```
date.setDate(3, 8, 59);
```

Therefore, why should I bother to check status?

```
date.setDate(1959, 3, 8);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

I “know” this date is valid (It’s my birthday)!

```
date.setDate(1959, 3, 8);
```

Therefore, why should I bother to check status?

```
date.setDate(1959, 3, 8);
```

**Double Fault!!**

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- Returning status implies a wide interface contract.

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- Returning status implies a wide interface contract.
- Wide contracts prevent defending against such errors in any build mode.

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
void Date:: setDate(int y,  
                    int m,  
                    int d)  
{  
    d_year = y;  
    d_month = m;  
    d_day = d;  
}
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
void Date:: setDate(int y,  
                     int m,  
                     int d)  
{  
    assert(isValid(y,m,d));  
    d_year = y;  
    d_month = m;  
    d_day = d;  
}
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
void Date:: setDate(int y,  
                    int m,  
                    int d)  
{  
    assert(isValid(y,m,d));  
    d_year = y;  
    d_month = m;  
    d_day = d;  
}
```

**Narrow Contract:**  
Checked Only In  
“Debug Mode”

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
int Date:: setDateIfValid(int y,  
                           int m,  
                           int d)  
{  
    if (!isValid(y, m, d)) {  
        return !0;  
    }  
    d_year = y;  
    d_month = m;  
    d_day = d;  
    return 0;  
}
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

```
int Date:: setDateIfValid(int y,  
                           int m,  
                           int d)  
{  
    if (!isValid(y, m, d)) {  
        return !0;  
    }  
    d_year = y;  
    d_month = m;  
    d_day = d;  
    return 0;  
}
```

**Wide Contract:**  
Checked in  
**Every** Build Mode

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined?

```
TYPE& vector<TYPE>::operator[] (int idx);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined?

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract?

```
TYPE& vector<TYPE>::at (int idx);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract?

```
TYPE& vector<TYPE>::at (int idx);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

- Should what happens be part of the contract? **If it is, then it's defined behavior!**

```
TYPE& vector<TYPE>::at (int idx);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

Must check  
in every  
build mode!

What happens be part of the  
**If it is, then it's defined behavior!**

```
TYPE& vector<TYPE>::at (int idx);
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*

- What should happen when the behavior is undefined? **It depends on the build mode.**

```
TYPE& vector<TYPE>::operator[] (int idx);
```

Must check  
in every  
build mode!

What happens be part  
**If it is, then it's defi**

```
TYPE& vector<TYPE>::at (in
```

**Bad Idea!** or!

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? If so, what should it be?

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? If so, what should it be?

```
if (idx < 0)           idx = 0;
```

```
if (idx > length())  idx = length();
```

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx < 0)           idx = 0;  
if (idx > length())   idx = length();  
idx = abs(idx) % (length() + 1);
```

### 3. 'Good' Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? If so, what should it be?

**More Code**

```
if (idx < 0)           idx = 0;  
if (idx > length())   idx = length();  
idx = abs(idx)         (length() + 1);
```

**Runs Slower!**

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx > length()) idx = length();  
if (idx < 0) idx = max(-idx, -length());  
idx = abs(idx) % (length() + 1);
```

**Would Serve Only  
To Mask Defects**

### 3. 'Good' Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? If so, what should it be?

```
if (idx < 0) {  
    if (idx > length()) {  
        // ...  
    } else {  
        // ...  
    }  
}  
else {  
    if (idx > length()) {  
        // ...  
    } else {  
        // ...  
    }  
}  
idx = abs(idx) % (length() - 1);
```

**Undefined Behavior**

Serve Only  
To Me

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? **Answer: No!**

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? **Answer: No!**

```
void insert(int idx, const TYPE& value)
{
    assert(0 <= idx); assert(idx <= length());
    // ...
}
```

### 3. 'Good' Contracts

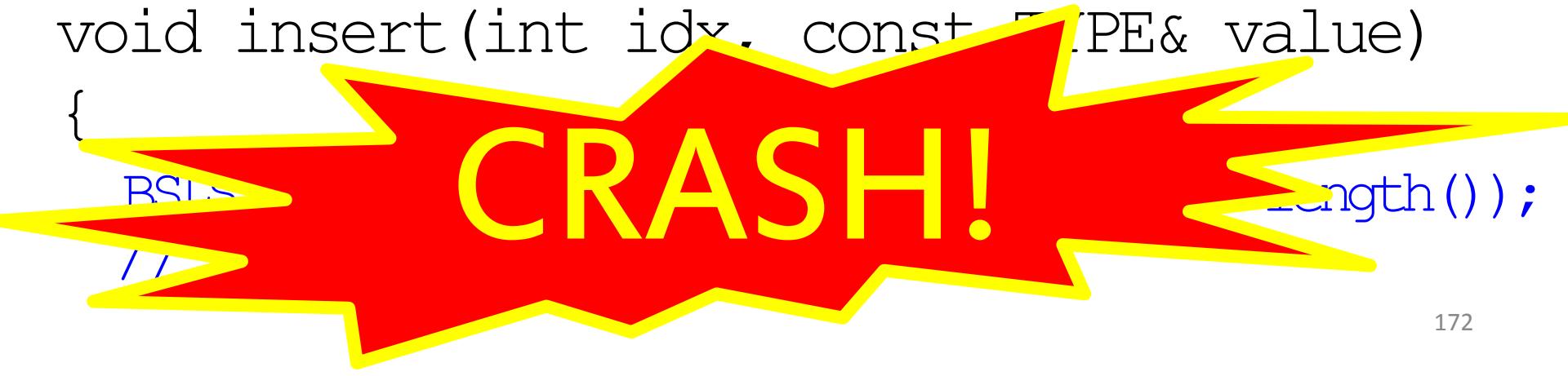
## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()` or less than zero? **Answer: No!**

```
void insert(int idx, const TYPE& value)
{
    // ...
```



CRASH!

```
... length());
```

### 3. 'Good' Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? **Answer: No!**

```
void insert(int id... con... TYPE& val...){  
    BST C...  
    // ...  
}
```

Or, as we will soon see, ...  
Something Much Better!

### 3. ‘Good’ Contracts

## Narrow versus Wide Contracts

*Narrow Contracts Imply Undefined Behavior:*  
Should the behavior for

```
void insert(int idx, const TYPE& value);
```

be defined when `idx` is greater than `length()`  
or less than zero? **Answer: No!**

```
void insert(int idx, const TYPE& value)
{
    BSLS_ASSERT(0 <= idx); BSLS_ASSERT(idx <= length());
    // ...
}
```

### 3. ‘Good’ Contracts

## Appropriately Narrow Contracts

### 3. ‘Good’ Contracts

## Appropriately Narrow Contracts

Narrow, but not too narrow.

### 3. ‘Good’ Contracts

## Appropriately Narrow Contracts

Narrow, but not too narrow.

Should the behavior for

```
void replace(int index,  
            const TYPE& value,  
            int numElements);
```

be defined when `index` is `length()` and  
`numElements` is zero?

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

Narrow, but not too narrow.

Should the behavior for



```
void replace(int index,  
            const TYPE& value,  
            int numElements);
```

be defined when `index` is `length()` and  
`numElements` is zero?



### 3. 'Good' Contracts

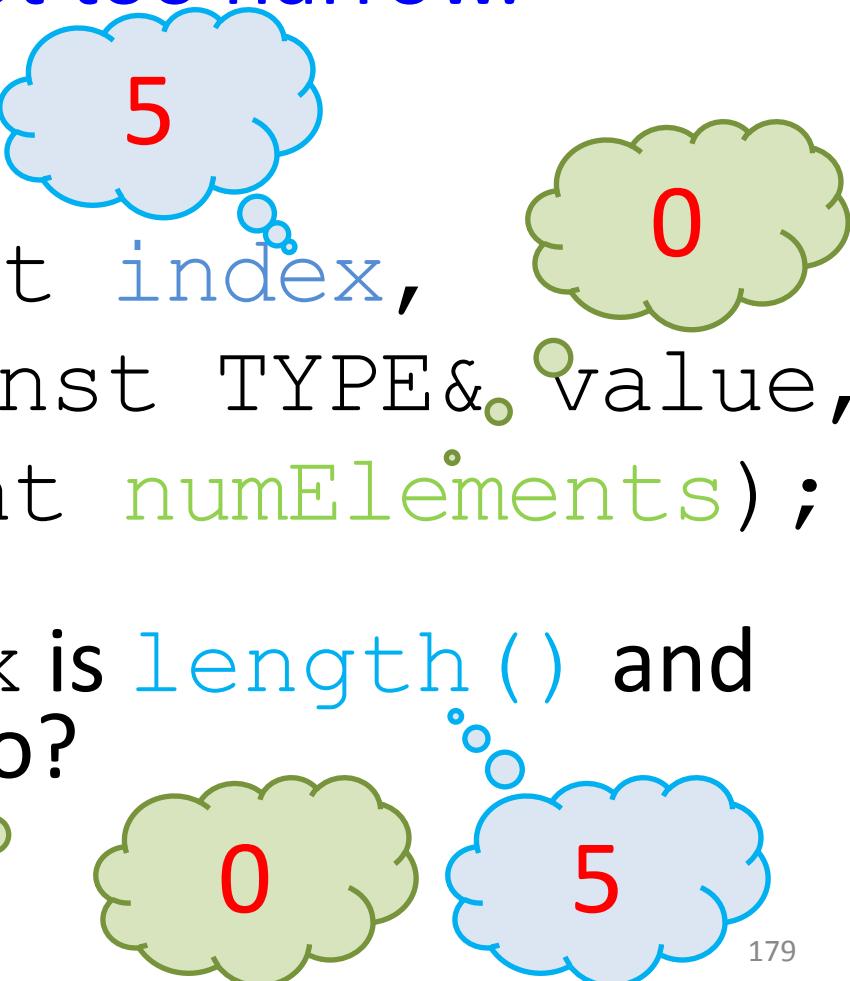
## Appropriately Narrow Contracts

Narrow, but not too narrow.

Should the behavior for

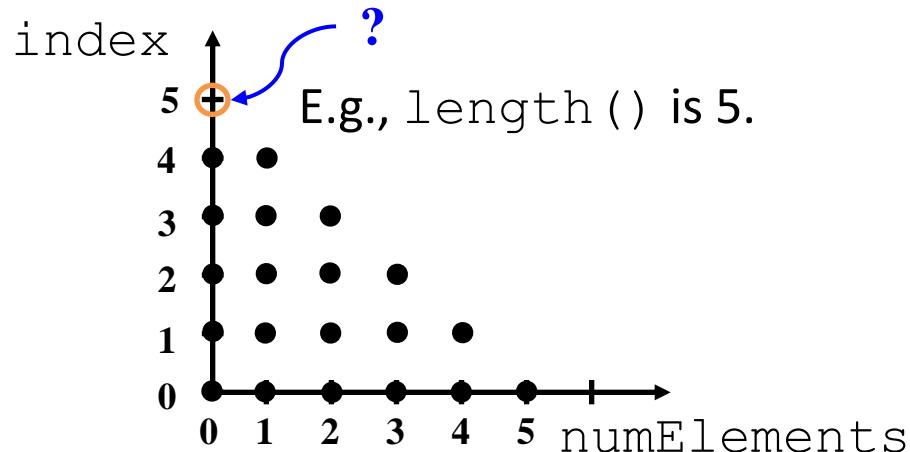
```
void replace(int index,  
            const TYPE& value,  
            int numElements);
```

be defined when `index` is `length()` and  
`numElements` is zero?



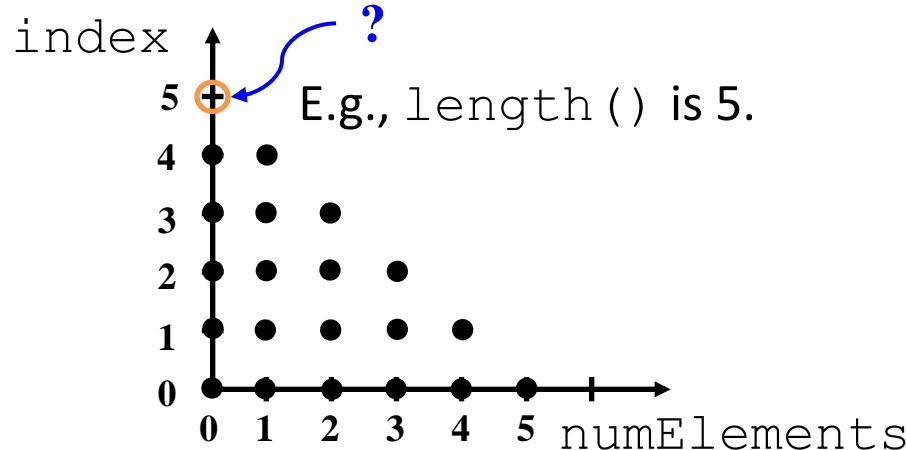
### 3. 'Good' Contracts

## Appropriately Narrow Contracts



### 3. 'Good' Contracts

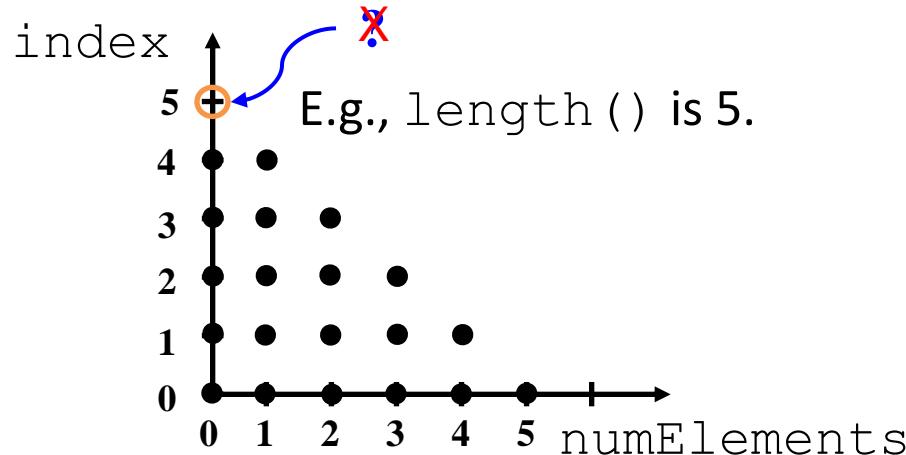
## Appropriately Narrow Contracts



```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

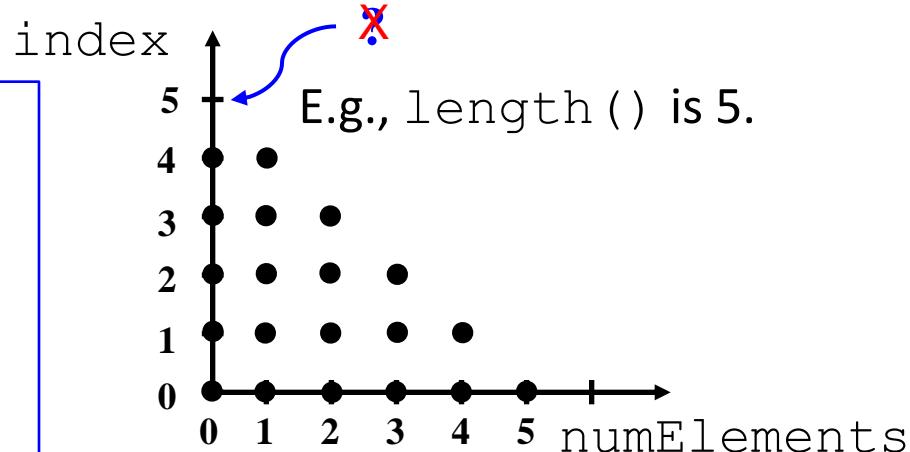


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

Now a client  
would have  
to check for this  
special case.

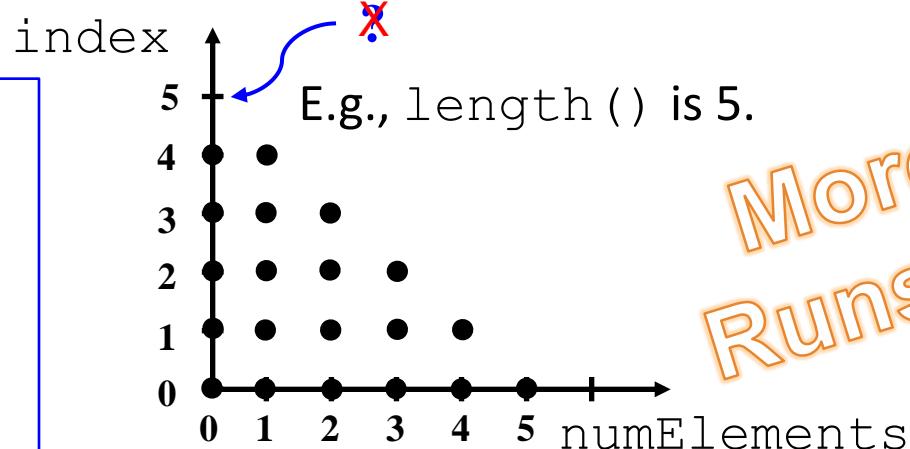


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

Now a client  
would have  
to check for this  
special case.



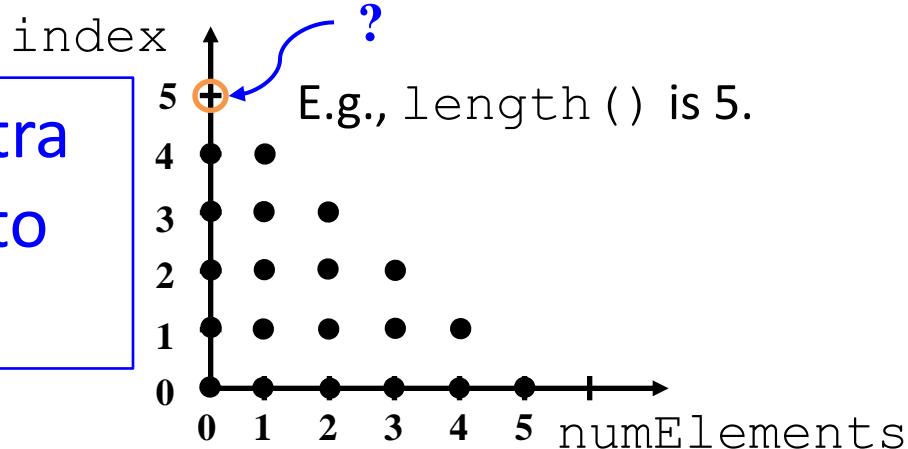
More Code  
Runs Slower!

```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

Assuming no extra code is needed to handle it ...

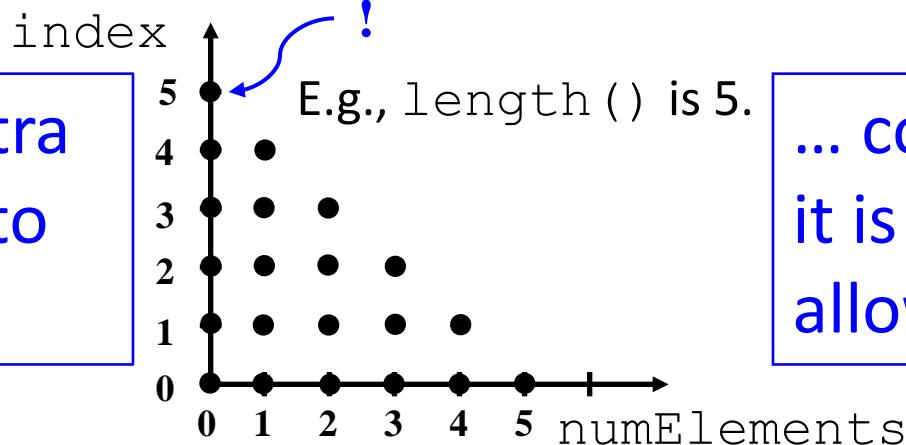


```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. 'Good' Contracts

## Appropriately Narrow Contracts

Assuming no extra code is needed to handle it ...



... continuity says it is natural to allow it.

```
void replace(int index,  
            const TYPE& value,  
            int numElements)  
{  
    assert(0 <= index);  
    assert(0 <= numElements);  
    assert(index + numElements <= length());  
    // ...  
}
```

### 3. ‘Good’ Contracts

# Contracts and Exceptions

### 3. ‘Good’ Contracts

## Contracts and Exceptions

Preconditions always Imply Postconditions:

### 3. ‘Good’ Contracts

## Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.

### 3. ‘Good’ Contracts

## Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.
- This failure should be due to physical limitations of the underlying hardware (not its current state), and such that it would not have occurred on a “larger” machine.
- Memory allocation alone is the one physical resource that is inseparably intertwined with object management.
- `std::bad_alloc` is arguably the only exception that belongs in any library (allocator) component’s contract.
- `abort()` should be considered a viable alternative to `throw` in virtually all cases (if exceptions are disabled).

### 3. ‘Good’ Contracts

## Contracts and Exceptions

Preconditions always Imply Postconditions:

- If a function cannot satisfy its contract (given valid preconditions) it must not return normally.
- This failure should be due to physical limitations of the underlying hardware (not its current state), and such that it would not have occurred on a “larger” machine.
- Memory allocation alone is the one physical resource that is inseparably intertwined with object management.
- `std::bad_alloc` is arguably the only exception that belongs in any library (allocator) component’s contract.
- `abort()` should be considered a viable alternative to `throw` in virtually all cases (if exceptions are disabled).
- Good library components are *exception-neutral* (via RAII).

3. ‘Good’ Contracts  
End of Section

Questions?

### 3. ‘Good’ Contracts

## What Questions are we Answering?

- What do we mean by *Defensive Programming (DP)*?
- What do we mean by a *narrow* versus a *wide* contract?
  - Should `std::strlen(0)` be required to do something reasonable?
  - Should `Date:: setDate(int, int, int)` return a status?
- What should happen when the behavior is undefined?
  - Should what happens be part of the component-level contract?
- What about the behavior for these specific interfaces:
  - Should `operator[](int index)` check to see if `index` is less than zero or greater than `length()`?
    - And what should happen if `index` is out of range?
  - Should `insert(int index, const TYPE& value)` be defined when `index` is greater than `length()` or less than zero?
  - Should `replace(int index, const TYPE& value, int numElements)` be defined when `index` is `length()` and `numElements` is zero?

# Outline

1. Brief Review of Physical Design
2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior
3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)
4. Implementing Defensive Checks

Using the **bsls\_assert** Component
5. Negative Testing

Using the **bsls\_asserttest** Component

# Outline

1. Brief Review of Physical Design

2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

4. Implementing Defensive Checks

Using the **bsls\_assert** Component

5. Negative Testing

Using the **bsls\_asserttest** Component

## 4. Implementing Defensive Checks

# Addressing Client Misuse

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

What should happen if the client misuses library code?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?
- b. be more careful?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?
- b. be more careful?
- c. have tested more thoroughly?

## 4. Implementing Defensive Checks

### Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?
- b. be more careful?
- c. have tested more thoroughly?
- d. ask the library component to warn us?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?
- b. be more careful?
- c. have tested more thoroughly?
- d. ask the library component to warn us?

How?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

What should happen if the client misuses library code?

As application developers, should we...

- a. be fired?
- b. be more careful?
- c. have tested more thoroughly?
- d. ask the library component to warn us?

How? (Please don't say "Return status.")

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?
4. How much of the application's CPU time should we spend trying to detect misuse of our code?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?
4. How much of the application's CPU time should we spend trying to detect misuse of our code?
  - a. Less than 5%

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?
4. How much of the application's CPU time should we spend trying to detect misuse of our code?
  - a. Less than 5%
  - b. 5% to 20%

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?
4. How much of the application's CPU time should we spend trying to detect misuse of our code?
  - a. Less than 5%
  - b. 5% to 20%
  - c. More than 20%, but not more than a constant factor.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As Library developers...

1. Should we document our code better?
2. Should we try to detect undefined behavior?
3. Can we detect *all* undefined behavior?
4. How much of the application's CPU time should we spend trying to detect misuse of our code?
  - a. Less than 5%
  - b. 5% to 20%
  - c. More than 20%, but not more than a constant factor.
  - d. Sky's the limit: factor of  $O[\log(n)]$ ,  $O[n]$ , or more?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?
- f. Spin, waiting to break into a debugger?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

As library developers, what should happen if we detect client misuse?

- a. Be fired?
- b. Ignore it, and proceed on? (See a.)
- c. Return immediately, but normally? (See a.)
- d. Immediately terminate the program?
- e. Throw an exception?
- f. Spin, waiting to break into a debugger?
- g. Something else?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?
4. Is there something sensible to do?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?
4. Is there something sensible to do?
  - a. Save client work before terminating the program.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?
4. Is there something sensible to do?
  - a. Save client work before terminating the program.
  - b. Log the error, abandon the current transaction, & proceed.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

How do we as an enterprise decide what to do?

It depends...

1. How mature is the software?
2. Are we in *alpha*, *beta*, or *production*?
3. Is this a performance-critical application?
4. Is there something sensible to do?
  - a. Save client work before terminating the program.
  - b. Log the error, abandon the current transaction, & proceed.
  - c. Send a message to the console room and just wait.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?

## 4. Implementing Defensive Checks

# Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. The owner of the application, who:

## 4. Implementing Defensive Checks

# Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. **The owner of the application, who:**
  - i. Is responsible for building the application.

## 4. Implementing Defensive Checks

# Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. **The owner of the application, who:**
  - i. Is responsible for building the application.
  - ii. Owns main .

## 4. Implementing Defensive Checks

# Addressing Client Misuse

Who should decide...

1. How much time the library component should spend checking for preconditions?
2. What happens if preconditions are violated?

Should it be...

- a. The (reusable) library component developer?
- b. The developer of the immediate client?
- c. **The owner of the application, who:**
  - i. Is responsible for building the application.
  - ii. Owns main .



#### 4. Implementing Defensive Checks

## Addressing Client Misuse

High-Level Requirements:

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

High-Level Requirements:

1. Make it relatively easy for each **application owner** to specify:

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

### High-Level Requirements:

1. Make it relatively easy for each **application owner** to specify:
  - I. Roughly how much runtime the library is to spend checking for contract violations by its clients.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

### High-Level Requirements:

1. Make it relatively easy for each **application owner** to specify:
  - I. Roughly how much runtime the library is to spend checking for contract violations by its clients.
  - II. What to do in the event such a violation is detected.

#### 4. Implementing Defensive Checks

## Addressing Client Misuse

### High-Level Requirements:

1. Make it relatively easy for each **application owner** to specify:
  - I. Roughly how much runtime the library is to spend checking for contract violations by its clients.
  - II. What to do in the event such a violation is detected.
2. Make it relatively easy for (even high-level) **library developers** to implement such (compile- and runtime) configurable checks in their component source code.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

Provide 3 Kinds of BSLS\_ASSERT\* macros for library developers to use.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

Provide 3 Kinds of BSLS\_ASSERT\* macros for library developers to use. **By default:**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

Provide 3 Kinds of BSLS\_ASSERT\* macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

Provide 3 Kinds of BSLS\_ASSERT\* macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]  
Active if not -DBDE\_BUILD\_TARGET\_OPT  
or if -DBDE\_BUILD\_TARGET\_SAFE

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part I):

Provide 3 Kinds of BSLS\_ASSERT\* macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]  
Active if not -DBDE\_BUILD\_TARGET\_OPT  
or if -DBDE\_BUILD\_TARGET\_SAFE
- **BSLS\_ASSERT\_SAFE(EXPR)** [ $> 20\%$  to ?]  
Active only if -DBDE\_BUILD\_TARGET\_SAFE

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Plan (Part II):

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility ...

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility ...

```
typedef  
    void (*Handler) (const char *text,  
                      const char *file,  
                      int           line);
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility along with  
three “off-the-shelf” failure handlers:

```
typedef  
    void (*Handler) (const char *text,  
                      const char *file,  
                      int           line);
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility along with  
three “off-the-shelf” failure handlers:

```
typedef  
    void (*Handler) (const char *text,  
                      const char *file,  
                      int           line);
```

1. **failAbort**: Print to stderr and terminate.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility along with  
three “off-the-shelf” failure handlers:

```
typedef  
    void (*Handler) (const char *text,  
                      const char *file,  
                      int           line);
```

1. **failAbort**: Print to stderr and terminate.
2. **failThrow**: Wrap info in std::logic\_error.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Plan (Part II):

Provide a global callback facility along with  
three “off-the-shelf” failure handlers:

```
typedef  
    void (*Handler) (const char *text,  
                      const char *file,  
                      int           line);
```

1. **failAbort**: Print to stderr and terminate.
2. **failThrow**: Wrap info in std::logic\_error.
3. **failSleep**: Print to stderr and spin/sleep.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Plan (Part II):

Provide a global callback facility along with three “off-the-shelf” failure handlers:

```
typedef void (*Handler) (const char *text,  
                         const char *file,  
                         int line);
```

1. Based on recent feedback from the C++ Standards Committee’s LEWG, we are moving to a single **struct** to capture all of the handler argument information.
2. sleep.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 1 ([Library](#)): Using **BSLS\_ASSERT**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.h  
// ...
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.h
// ...
struct MathUtil {
    static double factorial(int n);
    // ...
}; // ...
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.h
// ...
struct MathUtil {
    static double factorial(int n);
    // Return the product of values from 1
    // to the specified 'n', or 1 if 'n' is
    // 0. The behavior is undefined unless
    // '0 <= n <= 100'. Note that
    // 'factorial(1)' is 1.

    // ...
};

// ...
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 1 ([Library](#)): Using **BSLS\_ASSERT**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp
```

```
// ...
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp  
#include <our_mathutil.h>
```

```
// ...
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp
#include <our_mathutil.h>

// ...

double MathUtil::factorial(int n)
{
}

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp
#include <our_mathutil.h>
#include <bsls_assert.h>
// ...
```

```
double MathUtil::factorial(int n)
{
```

```
}
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp
#include <our_mathutil.h>
#include <bsls_assert.h>
// ...

double MathUtil::factorial(int n)
{
    BSLS_ASSERT(0 <= n); BSLS_ASSERT(n <= 100);

}

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 1 (**Library**): Using **BSLS\_ASSERT**

```
// our_mathutil.cpp
#include <our_mathutil.h>
#include <bsls_assert.h>
// ...

double MathUtil::factorial(int n)
{
    BSLS_ASSERT(0 <= n); BSLS_ASSERT(n <= 100);

    // FACTORIAL IMPLEMENTATION GOES HERE.

}
// ...
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    double z = our::MathUtil::factorial(-5);
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    double z = our::MathUtil::factorial(-5);
    // ...
}
```

```
$ CC -o a.out main.cpp my_client.cpp our_mathutil.cpp ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 1 (Client): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    double z = our::MathUtil::factorial(-5);
    // ...
}
$ CC -o a.out main.cpp my_client.cpp our_mathutil.cpp ... .
```

We must also link with  
**bsls\_assert.o**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    double z = our::MathUtil::factorial(-5);
    // ...
}

$ CC -o a.out main.cpp my_client.cpp our_mathutil.cpp ...
$ ./a.out
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 1 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_mathutil.h>
// ...
void someFunction()
{
    // ...
    double z = our::MathUtil::factorial(-5);
    // ...
}
```

```
$ CC -o a.out main.cpp my_client.cpp our_mathutil.cpp ...
$ ./a.out
```

```
Assertion failed: 0 <= n, file our_mathutil.cpp, line 365
Abort (core dumped)
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 2 ([Library](#)): Using **BSLS\_ASSERT\_SAFE**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
```

```
// ...
```

```
// ...
```

```
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
class Square {
public:
    // ...
};

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
class Square {
    double d_width;
public:
    // ...
};

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
};

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
        // ... The behavior is undefined unless
        // '0.0 <= width'.
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
        // ... The behavior is undefined unless
        // '0.0 <= width'.
};
// ...
inline void Square::setWidth(double width) {
}
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 2 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
// ...
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
        // ... The behavior is undefined unless
        // '0.0 <= width'.
};
// ...
inline void Square::setWidth(double width) {
    d_width = width;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 2 (Library): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
#include <bsls_assert.h>
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
        // ... The behavior is undefined unless
        // '0.0 <= width'.
};
// ...
inline void Square::setWidth(double width) {
    d_width = width;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 2 (Library): Using **BSLS\_ASSERT\_SAFE**

```
// our_square.h
// ...
#include <bsls_assert.h>
// ...
class Square {
    double d_width;
public:
    // ...
    void setWidth(double width);
        // ... The behavior is undefined unless
        // '0.0 <= width'.
};
// ...
inline void Square::setWidth(double width) {
    BSLS_ASSERT_SAFE(0.0 <= width);
    d_width = width;
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
```

```
// ...
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>

// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    our::Square s;
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    our::Square s;
    s.setWidth(-3.14);
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    our::Square s;
    s.setWidth(-3.14);
    // ...
}
```

```
$ CC -o a.out -DBDE_BUILD_TARGET_SAFE \
      main.cpp my_client.cpp our_square.cpp ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 2 (**Client**): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    our::Square s;
    s.setWidth(-3.14);
    // ...
}
```

```
$ CC -o a.out -DBDE_BUILD_TARGET_SAFE \
      main.cpp my_client.cpp our_square.cpp ...
$ ./a.out
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 2 (Client): Using **failAbort** (by default)

```
// my_client.cpp
#include <my_client.h>
#include <our_square.h>
// ...
void someFunction()
{
    our::Square s;
    s.setWidth(-3.14);
    // ...
}
```

```
$ CC -o a.out -DBDE_BUILD_SAFE \
      main.cpp my_client.cpp our_square.cpp ...
$ ./a.out
Assertion failed: 0 <= width, file our_square.h, line 142
Abort (core dumped)
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be quite expensive.
- `BSLS_ASSERT`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be quite expensive.
- `BSLS_ASSERT`
  - Checks that are cheap, but not negligible.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be quite expensive.
- `BSLS_ASSERT`
  - Checks that are cheap, but not negligible.
- `BSLS_ASSERT_OPT`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**,

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

Rule-of-thumb (guideline):

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

Rule-of-thumb (guideline):

- ❖ Inline functions:

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

Rule-of-thumb (guideline):

- ❖ Inline functions: use `BSLS_ASSERT_SAFE`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

Rule-of-thumb (guideline):

- ❖ Inline functions: use `BSLS_ASSERT_SAFE`
- ❖ Non-inline functions:

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Which `BSLS_ASSERT*` should be used?

- `BSLS_ASSERT_SAFE`
  - Checks that could be **quite expensive**.
- `BSLS_ASSERT`
  - Checks that are **cheap**, but **not negligible**.
- `BSLS_ASSERT_OPT`
  - Checks that are either (a) **not observable**, or (b) **critical**.

Rule-of-thumb (guideline):

- ❖ **Inline functions:** use `BSLS_ASSERT_SAFE`
- ❖ **Non-inline functions:** use `BSLS_ASSERT`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
    Kpoint(int x, int y);
};

};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
    Kpoint(int x, int y);
    // ... The behavior is undefined unless
    // '-1000 <= x <= 1000' and '-1000 <= y <= 1000'.
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
    Kpoint(int x, int y);
    // ... The behavior is undefined unless
    // '-1000 <= x <= 1000' and '-1000 <= y <= 1000'.
};

inline
Kpoint::Kpoint(int x, int y)
: d_x(x)
, d_y(y)
{

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
    Kpoint(int x, int y);
    // ... The behavior is undefined unless
    // '-1000 <= x <= 1000' and '-1000 <= y <= 1000'.
};
```

### **inline**

```
Kpoint::Kpoint(int x, int y)
: d_x(x)
, d_y(y)
{
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 3 (**Library**): Using **BSLS\_ASSERT\_SAFE**

```
// our_kpoint.h
// ...
class Kpoint {
    short int d_x;
    short int d_y;
public:
    Kpoint(int x, int y);
    // ... The behavior is undefined unless
    // '-1000 <= x <= 1000' and '-1000 <= y <= 1000'.
};

inline
Kpoint::Kpoint(int x, int y)
: d_x(x)
, d_y(y)
{
    BSLS_ASSERT_SAFE(-1000 <= x); BSLS_ASSERT_SAFE(x <= 1000);
    BSLS_ASSERT_SAFE(-1000 <= y); BSLS_ASSERT_SAFE(y <= 1000);
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 4 ([Library](#)): Using **BSLS\_ASSERT**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
};

};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
void resize(double loadFactor);
};

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
    void resize(double loadFactor);
        // Adjust the size of the underlying hash table to be
        // approximately the current number of elements divided
        // by the specified 'loadFactor'.
};

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
    void resize(double loadFactor);
        // Adjust the size of the underlying hash table to be
        // approximately the current number of elements divided
        // by the specified 'loadFactor'. The behavior is
        // undefined unless '0 < loadFactor'.
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
    void resize(double loadFactor);
        // Adjust the size of the underlying hash table to be
        // approximately the current number of elements divided
        // by the specified 'loadFactor'. The behavior is
        // undefined unless '0 < loadFactor'.
};

// our_hashtable.cpp
// ...
void HashTable::resize(double loadFactor)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
    void resize(double loadFactor);
        // Adjust the size of the underlying hash table to be
        // approximately the current number of elements divided
        // by the specified 'loadFactor'. The behavior is
        // undefined unless '0 < loadFactor'.
};

// our_hashtable.cpp
// ...
void HashTable::resize(double loadFactor)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 4 (**Library**): Using **BSLS\_ASSERT**

```
// our_hashtable.h
// ...
class HashTable {
    // ...
public:
    // ...
    void resize(double loadFactor);
        // Adjust the size of the underlying hash table to be
        // approximately the current number of elements divided
        // by the specified 'loadFactor'. The behavior is
        // undefined unless '0 < loadFactor'.
};

// our_hashtable.cpp
// ...
void HashTable::resize(double loadFactor)
{
    BSLS_ASSERT(0 < loadFactor);
    // ...
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 5 ([Library](#)): Using **BSLS\_ASSERT\_OPT**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 ([Library](#)): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
};

};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (**Library**): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
void executeTrade(int scalingFactor);
};

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (**Library**): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.
};

};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (**Library**): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.  The behavior is undefined unless
        // '0 <= scalingFactor' and '100' evenly divides
        // 'scalingFactor'.
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (**Library**): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.  The behavior is undefined unless
        // '0 <= scalingFactor' and '100' evenly divides
        // 'scalingFactor'.
};

// our_tradingsystem.cpp
// ...
void TradingSystem::executeTrade(int scalingFactor)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (**Library**): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.  The behavior is undefined unless
        // '0 <= scalingFactor' and '100' evenly divides
        // 'scalingFactor'.
};

// our_tradingsystem.cpp
// ...
void TradingSystem::executeTrade(int scalingFactor)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Library): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.  The behavior is undefined unless
        // '0 <= scalingFactor' and '100' evenly divides
        // 'scalingFactor'.
};

// our_tradingsystem.cpp
// ...
void TradingSystem::executeTrade(int scalingFactor)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Library): Using **BSLS\_ASSERT\_OPT**

```
// our_tradingsystem.h
// ...
class TradingSystem {
    // ...
public:
    // ...
    void executeTrade(int scalingFactor);
        // Execute the current trade using the specified
        // 'scalingFactor'.  The behavior is undefined unless
        // '0 <= scalingFactor' and '100' evenly divides
        // 'scalingFactor'.
};

// our_tradingsystem.cpp
// ...
void TradingSystem::executeTrade(int scalingFactor)
{
    BSLS_ASSERT_OPT(0 <= scalingFactor);
    BSLS_ASSERT_OPT(0 == scalingFactor % 100);
    // ...
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 5 (**Client**): Using **failSpin** (explicitly)

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 5 (**Client**): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    // All useful work goes on here ...
    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}

// my_client.cpp
// ...
void clientFunction(our::TradingSystem *objectPtr) {
    // ...
}
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}

// my_client.cpp
// ...
void clientFunction(our::TradingSystem *objectPtr) {
    // ...
    objectPtr->executeTrade(10022);
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}

// my_client.cpp
// ...
void clientFunction(our::TradingSystem *objectPtr) {
    // ...
    objectPtr->executeTrade(10022);
    // ...
}

$ CC -o a.out -DBDE_BUILD_TARGET_OPT \
      main.cpp my_client.cpp our_tradingsystem.cpp
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}

// my_client.cpp
// ...
void clientFunction(our::TradingSystem *objectPtr) {
    // ...
    objectPtr->executeTrade(10022);
    // ...
}

$ CC -o a.out -DBDE_BUILD_TARGET_OPT \
        main.cpp my_client.cpp our_tradingsystem.cpp
$ ./a.out
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 5 (Client): Using **failSpin** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failSpin);
    // All useful work goes on here ...
    return 0;
}

// my_client.cpp
// ...
void clientFunction(our::TradingSystem *objectPtr) {
    // ...
    objectPtr->executeTrade(10022);
    // ...
}

$ CC -o a.out -DBDE_BUILD_TARGET_OPT \
        main.cpp my_client.cpp our_tradingsystem.cpp
$ ./a.out
```

**(printed message sent to console room, waiting...)**

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

## 4. Implementing Defensive Checks

### Hard vs. Soft Undefined Behavior

This is a critical concept, yet it may hurt your head.

## 4. Implementing Defensive Checks

### Hard vs. Soft Undefined Behavior

This is a critical concept, yet it may hurt your head. Listen up!

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

There are two separate notions of *undefined behavior (UB)*:

## 4. Implementing Defensive Checks

# Hard vs. Soft Undefined Behavior

There are two separate notions of *undefined behavior (UB)*:

1. Language Undefined Behavior (*Hard UB*)

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

There are two separate notions of *undefined behavior (UB)*:

1. Language Undefined Behavior (*Hard UB*)
2. Library Undefined Behavior (*Soft UB*)

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

There are two separate notions of ***undefined behavior (UB)***:

1. Language Undefined Behavior (*Hard UB*)
2. Library Undefined Behavior (*Soft UB*)

“With **Language** UB, anything can happen:  
Your cat could get pregnant.” – Marshall Clow

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

There are two separate notions of ***undefined behavior (UB)***:

1. Language Undefined Behavior (*Hard UB*)
2. Library Undefined Behavior (*Soft UB*)

“With **Language** UB, anything can happen:  
Your cat could get pregnant.” – Marshall Clow

With **Library** (i.e. **Soft**) UB (a.k.a. a **Contract Violation**), nothing dire has happened yet, but it could easily lead to **Language** (i.e., **Hard**) UB.

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

***Soft*** Undefined Behavior is Relative:

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

**Soft** Undefined Behavior is Relative:

- If you are not privy to the implementation, then a **contract violation (*Soft UB*)** might lead to **language undefined behavior (*Hard UB*)**.

#### 4. Implementing Defensive Checks

## Hard vs. Soft Undefined Behavior

**Soft** Undefined Behavior is Relative:

- If you are not privy to the implementation, then a contract violation (**Soft UB**) might lead to language undefined behavior (**Hard UB**).
- If, however, you are part of the component
  - a. **implementation**
  - b. **test driver**

then a contract violation can remain **Soft UB!**

## 4. Implementing Defensive Checks

# Hard vs. Soft Undefined Behavior

Is passing 0 for **string** undefined behavior?

```
bsl::size_t bsl::strlen(const char *string)
// ... The behavior is undefined unless
// 'string' is null-terminated.
{
    BSLS_ASSERT_SAFE(string);
    const char *p = string;
    while (*p){
        ++p;
    }
    return p - string;
}
```

This is hard undefined behavior.

## 4. Implementing Defensive Checks

# Hard vs. Soft Undefined Behavior

Is passing 0 for **name** undefined behavior?

```
class Account {  
    std::string d_name;  
    // ...  
public:  
    // ...  
    void setName(const char *name)  
    // ... The behavior is undefined unless  
    // 'name' is null-terminated.  
    {  
        BSLS_ASSERT_SAFE(name);  
        d_name = name;  
    }  
}
```

This is hard  
undefined  
behavior.

## 4. Implementing Defensive Checks

# Hard vs. Soft Undefined Behavior

Is passing 0 for **result** undefined behavior?

```
void badLibraryFunction(std::string& result, ...)  
    // Load, into the specified 'result', the ...  
{  
    BSLS_ASSERT_SAFE(&result); // <- Too late!  
    // ...  
}
```

```
int someClientFunction(std::string *resultId, ...)  
    // ... The behavior is undefined unless 'resultId'  
    // is null-terminated.  
{  
    // ...  
    badLibraryFunction(*resultId, ...);  
    // ...  
}
```

This is hard  
undefined  
behavior.

## 4. Implementing Defensive Checks

# Hard vs. Soft Undefined Behavior

Is passing 0 for **result** undefined behavior?

```
void goodLibraryFunction(std::string *result, ...)  
    // Load, into the specified 'result', the ...  
{  
    BSLS_ASSERT_SAFE(result); // <- Not too late.  
    // ...  
}
```

```
int someClientFunction(std::string *resultId, ...)  
    // ... The behavior is undefined unless 'resultId'  
    // is null-terminated.  
{  
    // ...  
    goodLibraryFunction(resultId, ...);  
    // ...  
}
```

This is soft  
undefined  
behavior.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 6 (Client): Using **failThrow** (explicitly)

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 6 (**Client**): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 6 (Client): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failThrow);

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 6 (Client): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls_Assert::setFailureHandler(&bsls_Assert::failThrow);

    try {

    }
    catch (const std::logic_error& exception)  {

    }

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 6 (Client): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls Assert::setFailureHandler(&bsls Assert::failThrow);

    // SET UP INTERNAL STATE TO BE SAVED ON LOGIC ERROR.

    try {

    }
    catch (const std::logic_error& exception)  {

    }

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 6 (Client): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls Assert::setFailureHandler(&bsls Assert::failThrow);

    // SET UP INTERNAL STATE TO BE SAVED ON LOGIC ERROR.

    try {

        // ALL USEFUL WORK GOES HERE.
    }
    catch (const std::logic_error& exception)  {

    }

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 6 (Client): Using **failThrow** (explicitly)

```
int main(int argc, const char *argv[])
{
    bsls Assert::setFailureHandler(&bsls Assert::failThrow);

    // SET UP INTERNAL STATE TO BE SAVED ON LOGIC ERROR.

    try {

        // ALL USEFUL WORK GOES HERE.
    }
    catch (const std::logic_error& exception)  {

        // SAVE INTERNAL STATE AND EXIT.
    }

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Ex. 6 (Client): Using **failThrow** (explicitly)

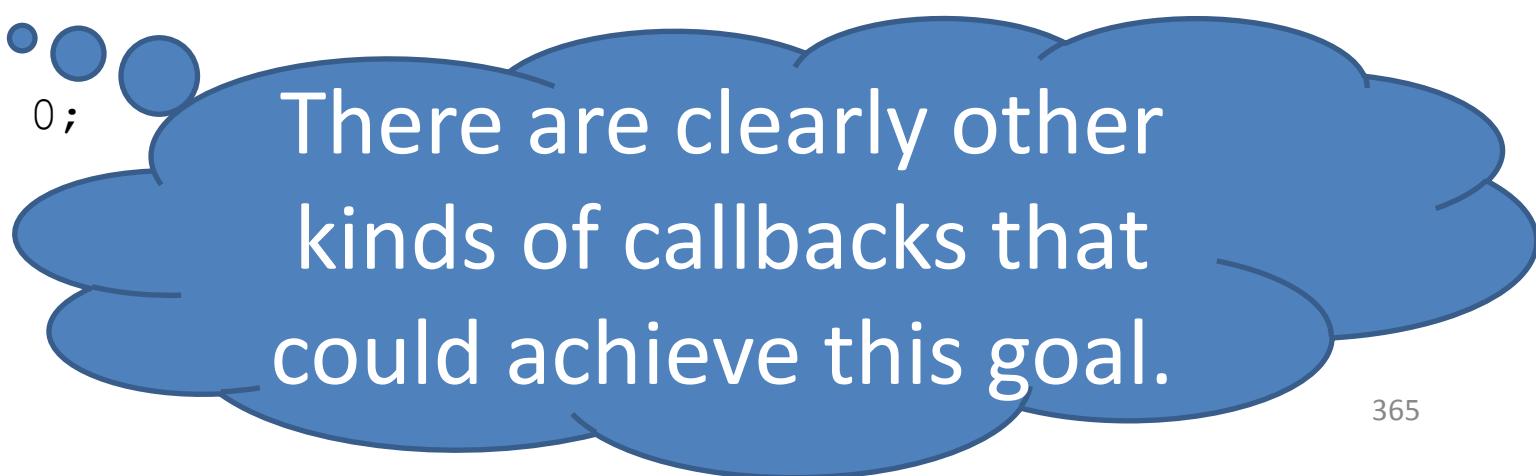
```
int main(int argc, const char *argv[])
{
    bsls Assert::setFailureHandler(&bsls Assert::failThrow);

    // SET UP INTERNAL STATE TO BE SAVED ON LOGIC ERROR.

    try {

        // ALL USEFUL WORK GOES HERE.
    }
    catch (const std::logic_error& exception) {

        // SAVE INTERNAL STATE AND EXIT.
    }
    return 0;
}
```



There are clearly other kinds of callbacks that could achieve this goal.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Support for Conditional Compilation

Corresponding to each Assertion Level:

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Support for Conditional Compilation

Corresponding to each Assertion Level:

- **BSLS\_ASSERT\_OPT**
  
- **BSLS\_ASSERT**
  
- **BSLS\_ASSERT\_SAFE**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Support for **Conditional Compilation**

Corresponding to each Assertion Level:

- BSLS\_ASSERT\_OPT  
**BSLS\_ASSERT\_OPT\_IS\_ACTIVE**
- BSLS\_ASSERT  
**BSLS\_ASSERT\_IS\_ACTIVE**
- BSLS\_ASSERT\_SAFE  
**BSLS\_ASSERT\_SAFE\_IS\_ACTIVE**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 7 (**Library**): Using **BSLS\_ASSERT\*\_IS\_ACTIVE**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

### Ex. 7 (**Library**): Using **BSLS\_ASSERT\*\_IS\_ACTIVE**

```
class MyDate {  
public:  
  
    // ...  
  
}; // ...  
// ...  
  
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 7 (**Library**): Using **BSLS\_ASSERT\*\_\_IS\_ACTIVE**

```
class MyDate {  
    int d_serialDate; // The valid serial range is [1 .. 3652061].  
public:  
  
    // ...  
  
}; // ...  
// ...  
  
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 7 (**Library**): Using **BSLS\_ASSERT\*\_IS\_ACTIVE**

```
class MyDate {  
    int d_serialDate; // The valid serial range is [1 .. 3652061].  
public:  
    MyDate();  
    // Create a 'MyDate' object having the value '0001Jan01'.  
    // ...  
    // ...  
};  
// ...  
  
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 7 (**Library**): Using **BSLS\_ASSERT\*\_IS\_ACTIVE**

```
class MyDate {  
    int d_serialDate; // The valid serial range is [1 .. 3652061].  
public:  
    MyDate();  
    // Create a 'MyDate' object having the value '0001Jan01'.  
    // ...  
    // ...  
};  
// ...  
inline MyDate::MyDate() : d_serialDate(1) {} // 0001Jan01  
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 7 (Library): Using **BSLS\_ASSERT\*\_\_IS\_ACTIVE**

```
class MyDate {
    int d_serialDate; // The valid serial range is [1 .. 3652061].
public:
    MyDate();
    // Create a 'MyDate' object having the value '0001Jan01'.
    // ...
#ifndef BSLS_ASSERT_SAFE_IS_ACTIVE
    ~MyDate();
    // Destroy this object.
#endif
    // ...
};
// ...

inline MyDate::MyDate() : d_serialDate(1) { } // 0001Jan01
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 7 (Library): Using **BSLS\_ASSERT\*\_\_IS\_ACTIVE**

```
class MyDate {
    int d_serialDate; // The valid serial range is [1 .. 3652061].
public:
    MyDate();
    // Create a 'MyDate' object having the value '0001Jan01'.
    // ...
#ifndef BSLS_ASSERT_SAFE_IS_ACTIVE
    ~MyDate();
    // Destroy this object.
#endif
    // ...
};

// ...

inline MyDate::MyDate() : d_serialDate(1) {} // 0001Jan01
// ...

#ifndef BSLS_ASSERT_SAFE_IS_ACTIVE
inline MyDate::~MyDate()
{
    BSLS_ASSERT_SAFE(1 <= d_serialDate); // 0001Jan01
    BSLS_ASSERT_SAFE(d_serialDate <= 3652061); // 9999Dec31
}
#endif
```

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility?

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? **No.**

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? **No.**
  - i. All assertion-level modes must be ABI compatible.

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? **No.**

- i. All assertion-level modes must be ABI compatible.
- ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? No.
  - i. All assertion-level modes must be ABI compatible.
  - ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.
2. Violating the One-Definition Rule?

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? **No.**
  - i. All assertion-level modes must be ABI compatible.
  - ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.
2. Violating the One-Definition Rule? **Yes.**

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? No.
  - i. All assertion-level modes must be ABI compatible.
  - ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.
2. Violating the One-Definition Rule? Yes.
  - ❖ Use of `bsls_assert` in header files could violate the ODR in a mixed assertion-level build – e.g., within

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? No.
  - i. All assertion-level modes must be ABI compatible.
  - ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.
2. Violating the One-Definition Rule? Yes.
  - ❖ Use of `bsls_assert` in header files could violate the ODR in a mixed assertion-level build – e.g., within
    - i. Inline-Function Definitions.

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

What are the implications?

1. ABI Incompatibility? No.
  - i. All assertion-level modes must be ABI compatible.
  - ii. Generally speaking, the only permitted side-effect is to invoke the failure handler.
2. Violating the One-Definition Rule? Yes.
  - ❖ Use of `bsls_assert` in header files could violate the ODR in a mixed assertion-level build – e.g., within
    - i. Inline-Function Definitions.
    - ii. Function-Template Definitions.

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

|  |  |  |
|--|--|--|
|  |  |  |
|  |  |  |
|  |  |  |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

|                                     |  |  |
|-------------------------------------|--|--|
| Client Build Mode<br>ACTIVE ASSERTS |  |  |
|                                     |  |  |
|                                     |  |  |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

|                   |  |  |
|-------------------|--|--|
| Client Build Mode |  |  |
| ACTIVE ASSERTS    |  |  |
| BSLS_ASSERT_OPT   |  |  |
| BSLS_ASSERT       |  |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      |                                      |  |
|                                     |                                      |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      |  |
|                                     |                                      |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      |             |
|                                     |                                      |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ?           |
|                                     |                                      |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT           |
|-------------------------------------|--------------------------------------|-----------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | Active or Not Active? |
|                                     |                                      |                       |
|                                     |                                      |                       |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ?           |
|                                     |                                      |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT       |
|-------------------------------------|--------------------------------------|-------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | <b>NOT ACTIVE</b> |
|                                     |                                      |                   |
|                                     |                                      |                   |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | NOT ACTIVE  |
| BSLS_ASSERT_OPT                     |                                      | ?           |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | NOT ACTIVE  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | ?           |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | NOT ACTIVE  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | ?           |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT           |
|-------------------------------------|--------------------------------------|-----------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | <b>NOT ACTIVE</b>     |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | Active or Not Active? |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | NOT ACTIVE  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | ?           |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

All Functions Defined in the .cpp File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT       |
|-------------------------------------|--------------------------------------|-------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | <b>NOT ACTIVE</b> |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | <b>ACTIVE</b>     |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
|                                     |                                      |  |
|                                     |                                      |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

## Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT_SAFE |
|-------------------------------------|--------------------------------------|------------------|
|                                     |                                      |                  |
|                                     |                                      |                  |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT_SAFE |
|-------------------------------------|--------------------------------------|------------------|
|                                     |                                      |                  |
|                                     |                                      |                  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT_SAFE |
|-------------------------------------|--------------------------------------|------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      |                                      |                  |
|                                     |                                      |                  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE |
|-------------------------------------|--|------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE |                  |
|                                     |  |                  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE |
|-------------------------------------|--|------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | ?                |
|                                     |  |                  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE         |
|-------------------------------------|--|--------------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | (PROBABLY)<br>NOT ACTIVE |
|                                     |  |                          |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS                | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE         |
|--|--|--------------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | (PROBABLY)<br>NOT ACTIVE |
| BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE |  |                          |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS                | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE         |
|--|--|--------------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | (PROBABLY)<br>NOT ACTIVE |
| BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | BSLS_ASSERT_OPT<br>BSLS_ASSERT                     |                          |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS                | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE         |
|--|--|--------------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | (PROBABLY)<br>NOT ACTIVE |
| BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | ?                        |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Inline Functions Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS                | Library Build Mode<br>ACTIVE ASSERTS               | BSLS_ASSERT_SAFE         |
|--|--|--------------------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | (PROBABLY)<br>NOT ACTIVE |
| BSLS_ASSERT_OPT<br>BSLS_ASSERT<br>BSLS_ASSERT_SAFE | BSLS_ASSERT_OPT<br>BSLS_ASSERT                     | (PROBABLY)<br>ACTIVE     |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
|                                     |                                      |  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |  |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
|                                     |                                      |  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |  |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      |                                      |  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS |  |
|-------------------------------------|--------------------------------------|--|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      |  |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |  |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      |             |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |             |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ?           |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ??          |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ??          |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ??          |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       |             |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ??          |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | ?           |

## 4. Implementing Defensive Checks

# Mixed-Mode (Assertion-Level) Builds Violating the One-Definition Rule (ODR) ...

### Function Templates Defined in the .h File

| Client Build Mode<br>ACTIVE ASSERTS | Library Build Mode<br>ACTIVE ASSERTS | BSLS_ASSERT |
|-------------------------------------|--------------------------------------|-------------|
| BSLS_ASSERT_OPT<br>BSLS_ASSERT      | BSLS_ASSERT_OPT                      | ??          |
| BSLS_ASSERT_OPT                     | BSLS_ASSERT_OPT<br>BSLS_ASSERT       | ??          |

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.
2. Semantically substitutable:

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.
2. Semantically substitutable:
  - I. W.R.T. Component-Level Contract ...

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.
2. Semantically substitutable:
  - I. W.R.T. Component-Level Contract ...
    - ❖ **NOTHING!** (Undefined Behavior)

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.
2. Semantically substitutable:
  - I. W.R.T. Component-Level Contract ...
    - ❖ **NOTHING!** (Undefined Behavior)
  - II. W.R.T. Library-Level Understanding ...

#### 4. Implementing Defensive Checks

## Mixed-Mode (Assertion-Level) Builds

Violating the One-Definition Rule (ODR) ...

Q: Why is violating ODR here OK?

A: Because all definitions are:

1. Syntactically (ABI) compatible.
2. Semantically substitutable:
  - I. W.R.T. Component-Level Contract ...
    - ❖ **NOTHING!** (Undefined Behavior)
  - II. W.R.T. Library-Level Understanding ...
    - ❖ See something, Say Something.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp  
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>
static bool globalEnableOurPrintingFlag = true;
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>
static bool globalEnableOurPrintingFlag = true;
static void
ourFailureHandler(const char *text, const char *file, int line)
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>

static bool globalEnableOurPrintingFlag = true;

static void
ourFailureHandler(const char *text, const char *file, int line)
    // Print the specified expression 'text', 'file' name, and
    // 'line' number to 'stdout' as a comma-separated list,
    // replacing null string-argument values with empty strings
    // (unless printing has been disabled by setting to 'false'
    // the 'globalEnableOurPrintingFlag' variable); then
    // unconditionally abort the process.
{
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>

static bool globalEnableOurPrintingFlag = true;

static void
ourFailureHandler(const char *text, const char *file, int line)
    // Print the specified expression 'text', 'file' name, and
    // 'line' number to 'stdout' as a comma-separated list,
    // replacing null string-argument values with empty strings
    // (unless printing has been disabled by setting to 'false'
    // the 'globalEnableOurPrintingFlag' variable); then
    // unconditionally abort the process.
{
    if (!text) text = "";
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>

static bool globalEnableOurPrintingFlag = true;

static void
ourFailureHandler(const char *text, const char *file, int line)
    // Print the specified expression 'text', 'file' name, and
    // 'line' number to 'stdout' as a comma-separated list,
    // replacing null string-argument values with empty strings
    // (unless printing has been disabled by setting to 'false'
    // the 'globalEnableOurPrintingFlag' variable); then
    // unconditionally abort the process.
{
    if (!text) text = "";
if (!file) file = "";

}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>

static bool globalEnableOurPrintingFlag = true;

static void
ourFailureHandler(const char *text, const char *file, int line)
    // Print the specified expression 'text', 'file' name, and
    // 'line' number to 'stdout' as a comma-separated list,
    // replacing null string-argument values with empty strings
    // (unless printing has been disabled by setting to 'false'
    // the 'globalEnableOurPrintingFlag' variable); then
    // unconditionally abort the process.
{
    if (!text) text = "";
    if (!file) file = "";
    if (globalEnableOurPrintingFlag) {
        std::printf("%s, %s, %d\n", text, file, line);
    }
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Creating a custom handler.

```
// main.cpp
// ...
#include <bsls_assert.h>

static bool globalEnableOurPrintingFlag = true;

static void
ourFailureHandler(const char *text, const char *file, int line)
    // Print the specified expression 'text', 'file' name, and
    // 'line' number to 'stdout' as a comma-separated list,
    // replacing null string-argument values with empty strings
    // (unless printing has been disabled by setting to 'false'
    // the 'globalEnableOurPrintingFlag' variable); then
    // unconditionally abort the process.
{
    if (!text) text = "";
    if (!file) file = "";
    if (globalEnableOurPrintingFlag) {
        std::printf("%s, %s, %d\n", text, file, line);
    }
    std::abort();
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp  
// ...
```

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp  
// ...
```

```
int main(int argc, const char *argv[])
{
```

```
    return 0;  
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

    ...

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

bsls_Assert::setFailureHandler(f);

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

    bsls_Assert::setFailureHandler(f);

bsls_Assert::invokeHandler("str1", "str2", 3);

    return 0;
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

    bsls_Assert::setFailureHandler(f);

    bsls_Assert::invokeHandler("str1", "str2", 3);

    return 0;
}
```

```
$CC -o a.out main.cpp
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

    bsls_Assert::setFailureHandler(f);

    bsls_Assert::invokeHandler("str1", "str2", 3);

    return 0;
}

$CC -o a.out main.cpp
$ ./a.out
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 8 (**Client**): Installing/Invoking custom handler.

```
// main.cpp
// ...

int main(int argc, const char *argv[])
{
    bsls_Assert::Handler f = &::ourFailureHandler;

    bsls_Assert::setFailureHandler(f);

    bsls_Assert::invokeHandler("str1", "str2", 3);

    return 0;
}
```

```
$CC -o a.out main.cpp
$ ./a.out
str1, str2, 3
Abort (core dumped)
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [**< 5%**]

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [**< 5%**]  
Always active.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]  
Active if not -DBDE\_BUILD\_TARGET\_OPT

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]  
Active if not `-DBDE_BUILD_TARGET_OPT`  
or if `-DBDE_BUILD_TARGET_SAFE`

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [ $5\% \text{ to } 20\%$ ]  
Active if not `-DBDE_BUILD_TARGET_OPT`  
or if `-DBDE_BUILD_TARGET_SAFE`
- **BSLS\_ASSERT\_SAFE(EXPR)** [ $> 20\% \text{ to } ?$ ]

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Recall our Plan (Part I):

Provide 3 Kinds of **BSLS\_ASSERT\*** macros for library developers to use. **By default:**

- **BSLS\_ASSERT\_OPT(EXPR)** [ $< 5\%$ ]  
Always active.
- **BSLS\_ASSERT(EXPR)** [5% to 20%]  
Active if not -DBDE\_BUILD\_TARGET\_OPT  
or if -DBDE\_BUILD\_TARGET\_SAFE
- **BSLS\_ASSERT\_SAFE(EXPR)** [ $> 20\%$  to ?]  
Active only if -DBDE\_BUILD\_TARGET\_SAFE

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Assertion-Level Overrides:

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET \*** switches by placing exactly one of the following **overrides** on the build line:

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

### Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET** \* switches by placing exactly one of the following **overrides** on the build line:

-D**BSLS\_ASSERT\_LEVEL\_NONE**

NO assert macros will be active.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET \*** switches by placing exactly one of the following **overrides** on the build line:

-DBSLS\_ASSERT\_LEVEL\_NONE

NO assert macros will be active.

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT Only BSLS\_ASSERT\_OPT will be active.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET** \* switches by placing exactly one of the following **overrides** on the build line:

- D**BSLS\_ASSERT\_LEVEL\_NONE** NO assert macros will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_OPT** Only **BSLS\_ASSERT\_OPT** will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT** Only **BSLS\_ASSERT\_SAFE** will **NOT** be active.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET** \* switches by placing exactly one of the following **overrides** on the build line:

- D**BSLS\_ASSERT\_LEVEL\_NONE** NO assert macros will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_OPT** Only **BSLS\_ASSERT\_OPT** will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT** Only **BSLS\_ASSERT\_SAFE** will NOT be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE** ALL assert macros will be active.

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET** \* switches by placing exactly one of the following **overrides** on the build line:

- D**BSLS\_ASSERT\_LEVEL\_NONE** NO assert macros will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_OPT** Only **BSLS\_ASSERT\_OPT** will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT** Only **BSLS\_ASSERT\_SAFE** will **NOT** be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE** ALL assert macros will be active.

Example of an optimized build, but with all three **BSLS\_ASSERT\*** macros enabled:

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Assertion-Level Overrides:

We can override the default assertion level derived from the **BDE\_BUILD\_TARGET \*** switches by placing exactly one of the following **overrides** on the build line:

- D**BSLS\_ASSERT\_LEVEL\_NONE** NO assert macros will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_OPT** Only **BSLS\_ASSERT\_OPT** will be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT** Only **BSLS\_ASSERT\_SAFE** will **NOT** be active.
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE** ALL assert macros will be active.

Example of an optimized build, but with all three **BSLS\_ASSERT\*** macros enabled:

```
CC -DBDE_BUILD_TARGET_OPT \
    -DBSLS_ASSERT_LEVEL_ASSERT_SAFE ...
```

#### 4. Implementing Defensive Checks

## Binary-Incompatible Defensive Checks

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

*Dereferencing an iterator invalidated by modifying its associated container is **undefined behavior**.*

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

*Dereferencing an iterator invalidated by modifying its associated container is undefined behavior.*

Check requires additional instance Data:

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

*Dereferencing an iterator invalidated by modifying its associated container is undefined behavior.*

Check requires additional instance Data:

- I. The **container** needs to maintain an additional linked list of back pointers.

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

*Dereferencing an iterator invalidated by modifying its associated container is undefined behavior.*

Check requires additional instance Data:

- I. The **container** needs to maintain an additional linked list of back pointers.
- II. Each **iterator** needs to maintain an additional pointer to its link in the list.

## 4. Implementing Defensive Checks

# Binary-Incompatible Defensive Checks

Motivating example: *Checked Iterators*

*Dereferencing an iterator invalidated by modifying its associated container is undefined behavior.*

Check requires additional instance Data:

- I. The container needs to maintain an additional linked list of back pointers.
- II. Each iterator needs to maintain an additional pointer to its link in the list.

Such checks are **NOT** binary compatible!

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 9 (**Library**): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 9 (**Library**): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_string.h
// ...
class String {
    // ...
    char* d_array_p;
    std::size_t d_length;
    std::size_t d_capacity;
    bslma_Allocator* d_allocator_p;

    // ...
}

};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 9 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_string.h
// ...
class String {
    // ...
    char* d_array_p;
    std::size_t d_length;
    std::size_t d_capacity;
    bslma_Allocator* d_allocator_p;

    // ...
    void removeAll() {
        d_length = 0;
    }
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 9 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_string.h
// ...
class String {
    // ...
    char                         *d_array_p;
    std::size_t                   d_length;
    std::size_t                   d_capacity;
    bslma_Allocator              *d_allocator_p;
#if defined(BDE_BUILD_TARGET_SAFE_2)
    bsl::list<iterator *>  d_backPointers;
#endif
    // ...
    void removeAll()  {
        d_length = 0;
    }
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 9 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_string.h
// ...
class String {
    // ...
    char                         *d_array_p;
    std::size_t                   d_length;
    std::size_t                   d_capacity;
    bslma_Allocator              *d_allocator_p;
#if defined(BDE_BUILD_TARGET_SAFE_2)
    bsl::list<iterator *>  d_backPointers;
#endif
    // ...
    void removeAll()  {
#if defined(BDE_BUILD_TARGET_SAFE_2)
        // Use 'd_backpointers' to invalidate all iterators.
#endif
        d_length = 0;
    }
};
```

#### 4. Implementing Defensive Checks

## Disproportionally Expensive Checks

#### 4. Implementing Defensive Checks

## Disproportionally Expensive Checks

Motivating example: *Sorted Order*

## 4. Implementing Defensive Checks

# Disproportionally Expensive Checks

## Motivating example: *Sorted Order*

*Binary search in a sorted array is only  $O[\log(n)]$ , but verifying that the array is in fact sorted is  $O[n]$ .*

## 4. Implementing Defensive Checks

# Disproportionally Expensive Checks

## Motivating example: *Sorted Order*

*Binary search in a sorted array is only  $O[\log(n)]$ , but verifying that the array is in fact sorted is  $O[n]$ .*

- Checks arising from binary incompatible builds are also typically very expensive – e.g., by a factor  $O[n]$ .

## 4. Implementing Defensive Checks

# Disproportionally Expensive Checks

## Motivating example: *Sorted Order*

*Binary search in a sorted array is only  $O[\log(n)]$ , but verifying that the array is in fact sorted is  $O[n]$ .*

- ❑ Checks arising from binary incompatible builds are also typically very expensive – e.g., by a factor  $O[n]$ .
- ❑ Other checks, even though not binary incompatible, could be prohibitively expensive, even in a safe build.

## 4. Implementing Defensive Checks

# Disproportionally Expensive Checks

## Motivating example: *Sorted Order*

*Binary search in a sorted array is only  $O[\log(n)]$ , but verifying that the array is in fact sorted is  $O[n]$ .*

- ❑ Checks arising from binary incompatible builds are also typically very expensive – e.g., by a factor  $O[n]$ .
- ❑ Other checks, even though not binary incompatible, could be prohibitively expensive, even in a safe build.
- ❑ **ARBITRARY CHOICE:**

## 4. Implementing Defensive Checks

# Disproportionally Expensive Checks

## Motivating example: *Sorted Order*

*Binary search in a sorted array is only  $O[\log(n)]$ , but verifying that the array is in fact sorted is  $O[n]$ .*

- Checks arising from binary incompatible builds are also typically very expensive – e.g., by a factor  $O[n]$ .
- Other checks, even though not binary incompatible, could be prohibitively expensive, even in a safe build.
- **ARBITRARY CHOICE:** We choose NOT to put checks with a higher (worst-case) runtime Big-Oh complexity in “normal” safe builds of our software.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
// ...
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    // ...
    // ...
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    // ...
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    static bool isMember(const std::vector<int>& a, int v);
    // ...
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    static bool isMember(const std::vector<int>& a, int v);
    // ... The behavior is undefined unless 'a' is sorted.
    // ...
};
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    static bool isMember(const std::vector<int>& a, int v);
        // ... The behavior is undefined unless 'a' is sorted.
    // ...
};

// my_algorithmutil.cpp
// ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    static bool isMember(const std::vector<int>& a, int v);
        // ... The behavior is undefined unless 'a' is sorted.
    // ...
};

// my_algorithmutil.cpp
// ...
bool AlgorithmUtil::isMember(const std::vector<T>& a, int v)
{
    // ...
}
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Ex. 10 (Library): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

```
// my_algorithmutil.h
// ...
#include <bsls_assert.h>
// ...
struct AlgorithmUtil {
    static bool isSorted(const std::vector<int>& a);
    // ...
    static bool isMember(const std::vector<int>& a, int v);
    // ... The behavior is undefined unless 'a' is sorted.
    // ...
};

// my_algorithmutil.cpp
// ...
bool AlgorithmUtil::isMember(const std::vector<T>& a, int v)
{
#if defined(BDE_BUILD_TARGET_SAFE_2)
    BSLS_ASSERT_SAFE(isSorted(a));
#endif
    // ...
}
```

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

**-DBDE\_BUILD\_TARGET\_SAFE\_2**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

-DBDE\_BUILD\_TARGET\_SAFE\_2

- I. All assertion levels are enabled (by default) in SAFE-2 mode.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

-DBDE\_BUILD\_TARGET\_SAFE\_2

- I. All assertion levels are enabled (by default) in SAFE-2 mode.
- II. Use **BSLS\_ASSERT\_LEVEL \*** overrides to reduce the default assertion level in SAFE-2 mode.

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

-DBDE\_BUILD\_TARGET\_SAFE\_2

- I. All assertion levels are enabled (by default) in SAFE-2 mode.
- II. Use BSLS\_ASSERT\_LEVEL \* overrides to reduce the default assertion level in SAFE-2 mode. E.g.,

**CC -DBDE\_BUILD\_TARGET\_SAFE\_2**

#### 4. Implementing Defensive Checks

## Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

-DBDE\_BUILD\_TARGET\_SAFE\_2

- I. All assertion levels are enabled (by default) in SAFE-2 mode.
- II. Use BSLS\_ASSERT\_LEVEL \* overrides to reduce the default assertion level in SAFE-2 mode. E.g.,

```
CC -DBDE_BUILD_TARGET_SAFE_2 \
    -DBSLS_ASSERT_LEVEL_ASSERT
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

Ex. 10 (Client): Using **BDE\_BUILD\_TARGET\_SAFE\_2**

A program must be built uniformly with (or without)

-DBDE\_BUILD\_TARGET\_SAFE\_2

- I. All assertion levels are enabled (by default) in SAFE-2 mode.
- II. Use BSLS\_ASSERT\_LEVEL \* overrides to reduce the default assertion level in SAFE-2 mode. E.g.,

```
CC -DBDE_BUILD_TARGET_SAFE_2 \
    -DBSLS_ASSERT_LEVEL_ASSERT \
        myalgorithms.cpp ...
```

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT  
-DBDE\_BUILD\_TARGET\_SAFE  
-DBDE\_BUILD\_TARGET\_SAFE\_2

BSLS\_ASSERT\_OPT(...)  
BSLS\_ASSERT(...)  
BSLS\_ASSERT\_SAFE(...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE  
-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT  
-DBSLS\_ASSERT\_LEVEL\_ASSERT  
-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

#### DEFAULT

- D**BDE\_BUILD\_TARGET\_OPT**
- D**BDE\_BUILD\_TARGET\_SAFE**
- D**BDE\_BUILD\_TARGET\_SAFE\_2**

| <code>BSLS_ASSERT_OPT(...)</code> | <code>BSLS_ASSERT(...)</code> | <code>BSLS_ASSERT_SAFE(...)</code> |
|-----------------------------------|-------------------------------|------------------------------------|
| @                                 | @                             |                                    |
| @                                 |                               |                                    |
| @                                 | @                             | @                                  |
| @                                 | @                             | @                                  |

### BSLS Assertion Overrides

- D**BSLS\_ASSERT\_LEVEL\_NONE**
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_OPT**
- D**BSLS\_ASSERT\_LEVEL\_ASSERT**
- D**BSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE**

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT

-DBDE\_BUILD\_TARGET\_SAFE

-DBDE\_BUILD\_TARGET\_SAFE\_2

BSLS\_ASSERT\_OPT(...)

BSLS\_ASSERT(...)

BSLS\_ASSERT\_SAFE(...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT

-DBSLS\_ASSERT\_LEVEL\_ASSERT

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT

-DBDE\_BUILD\_TARGET\_SAFE

-DBDE\_BUILD\_TARGET\_SAFE\_2

BSLS ASSERT OPT(...)

BSLS ASSERT (...)

BSLS ASSERT\_SAFE (...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT

-DBSLS\_ASSERT\_LEVEL\_ASSERT

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT

-DBDE\_BUILD\_TARGET\_SAFE

**-DBDE\_BUILD\_TARGET\_SAFE\_2**

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT

-DBSLS\_ASSERT\_LEVEL\_ASSERT

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

BSLS\_ASSERT\_OPT(...)  
BSLS\_ASSERT(...)  
BSLS\_ASSERT\_SAFE(...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT

-DBDE\_BUILD\_TARGET\_SAFE

-DBDE\_BUILD\_TARGET\_SAFE\_2

BSLS ASSERT OPT (...) BSLS ASSERT (...) BSLS ASSERT SAFE (...) BSLS ASSERT SAFE (...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT

-DBSLS\_ASSERT\_LEVEL\_ASSERT

-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

-DBDE\_BUILD\_TARGET\_OPT  
-DBDE\_BUILD\_TARGET\_SAFE  
-DBDE\_BUILD\_TARGET\_SAFE\_2

### BSLS Assertion Overrides

-DBSLS\_ASSERT\_LEVEL\_NONE  
-DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT  
-DBSLS\_ASSERT\_LEVEL\_ASSERT  
-DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

BSLS\_ASSERT\_OPT(...)  
BSLS\_ASSERT(...)  
BSLS\_ASSERT\_SAFE(...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

- DBDE\_BUILD\_TARGET\_OPT
- DBDE\_BUILD\_TARGET\_SAFE
- DBDE\_BUILD\_TARGET\_SAFE\_2

### BSLS Assertion Overrides

- DBSLS\_ASSERT\_LEVEL\_NONE
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT
- DBSLS\_ASSERT\_LEVEL\_ASSERT
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|                      |                  |                       |
|----------------------|------------------|-----------------------|
| BSLS_ASSERT_OPT(...) | BSLS_ASSERT(...) | BSLS_ASSERT_SAFE(...) |
| @                    | @                |                       |
| @                    |                  |                       |
| @                    | @                | @                     |
| @                    | @                | @                     |

|   |   |   |
|---|---|---|
|   |   |   |
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

- DBDE\_BUILD\_TARGET\_OPT
- DBDE\_BUILD\_TARGET\_SAFE
- DBDE\_BUILD\_TARGET\_SAFE\_2

### BSLS Assertion Overrides

- DBSLS\_ASSERT\_LEVEL\_NONE
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT
- DBSLS\_ASSERT\_LEVEL\_ASSERT
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|                      |                  |                       |
|----------------------|------------------|-----------------------|
| BSLS_ASSERT_OPT(...) | BSLS_ASSERT(...) | BSLS_ASSERT_SAFE(...) |
| @                    | @                |                       |
| @                    |                  |                       |
| @                    | @                | @                     |
| @                    | @                | @                     |

|   |   |   |
|---|---|---|
|   |   |   |
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the `bsls_assert` Component

## Build-Mode Summary

### BDE Build Targets

- DEFAULT
- DBDE\_BUILD\_TARGET\_OPT
- DBDE\_BUILD\_TARGET\_SAFE
- DBDE\_BUILD\_TARGET\_SAFE\_2

BSLS ASSERT OPT(...)

BSLS ASSERT (...)

BSLS ASSERT\_SAFE (...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

### BSLS Assertion Overrides

- DBSLS\_ASSERT\_LEVEL\_NONE
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT
- DBSLS\_ASSERT\_LEVEL\_ASSERT**
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

# Using the **bsls\_assert** Component

## Build-Mode Summary

### BDE Build Targets

DEFAULT

- DBDE\_BUILD\_TARGET\_OPT
- DBDE\_BUILD\_TARGET\_SAFE
- DBDE\_BUILD\_TARGET\_SAFE\_2

### BSLS Assertion Overrides

- DBSLS\_ASSERT\_LEVEL\_NONE
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_OPT
- DBSLS\_ASSERT\_LEVEL\_ASSERT
- DBSLS\_ASSERT\_LEVEL\_ASSERT\_SAFE

BSLS\_ASSERT\_OPT(...)

BSLS\_ASSERT(...)

BSLS\_ASSERT\_SAFE(...)

|   |   |   |
|---|---|---|
| @ | @ |   |
| @ |   |   |
| @ | @ | @ |
| @ | @ | @ |

|   |   |   |
|---|---|---|
|   |   |   |
| @ |   |   |
| @ | @ |   |
| @ | @ | @ |

## 4. Implementing Defensive Checks

End of Section

# Questions?

## 4. Implementing Defensive Checks

# What Questions are we Answering?

- Who should decide how much CPU time to spend checking library preconditions and what to do if a violation is detected?
- Why don't library developers specify the defensive checks a component performs in its function-level contracts?
- When should library developers use `BSLS_ASSERT_SAFE` vs. `BSLS_ASSERT` vs. `BSLS_ASSERT_OPT`?
- What is the difference between hard vs. soft undefined behavior?
- What are the two primary causes for `bsls_assert` macros to violate ODR, and why is that acceptable?
- Why do we need `BDE_BUILD_TARGET_SAFE_2`, and how is it related (and not) to `BDE_BUILD_TARGET_SAFE`?

# Outline

1. Brief Review of Physical Design

2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

4. Implementing Defensive Checks

Using the **bsls\_assert** Component

5. Negative Testing

Using the **bsls\_asserttest** Component

## 5. Negative Testing

# The Component-Level Test Driver

## 5. Negative Testing

# The Component-Level Test Driver

What is a Test Driver?

## 5. Negative Testing

# The Component-Level Test Driver

What is a Test Driver?

- It's a **file** that defines `main`.

## 5. Negative Testing

# The Component-Level Test Driver

### Test Driver

```
// component.t.cpp
#include <component.h>
// ...
int main(...)
{
    //...
}
//-- END OF FILE --
```

**component.t.cpp**

```
// component.h
```

```
// ...
```

```
//-- END OF FILE --
```

**component.h**

```
// component.cpp
```

```
#include <component.h>
// ...
```

```
//-- END OF FILE --
```

**component.cpp**

**component**

## 5. Negative Testing

# The Component-Level Test Driver

## What is a Test Driver?

- It's a **file** that defines `main`.
- It's a **tool** for developers
  - used during the initial development process.

## 5. Negative Testing

# The Component-Level Test Driver

## What is a Test Driver?

- It's a **file** that defines `main`.
- It's a **tool** for developers
  - used during the initial development process.
- It's a **"cartridge"** for an automated regression-testing system
  - used throughout the lifetime of the component.

## 5. Negative Testing

# The Component-Level Test Driver

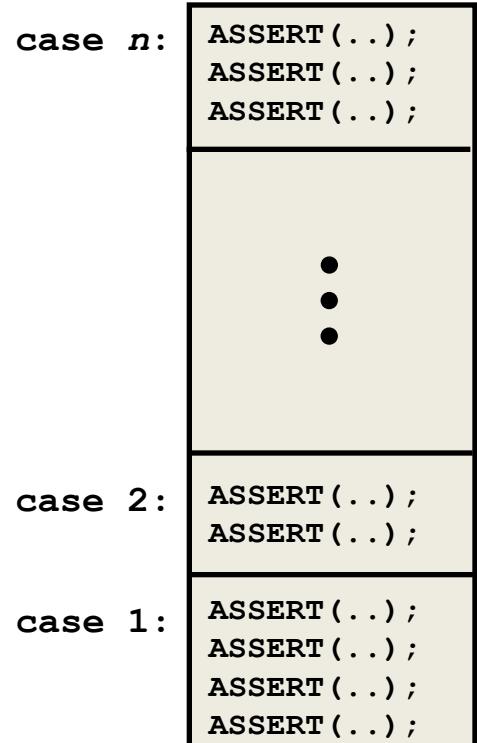
What does a Test Driver comprise?

## 5. Negative Testing

# The Component-Level Test Driver

What does a Test Driver comprise?

- Set of consecutively numbered **test cases**.

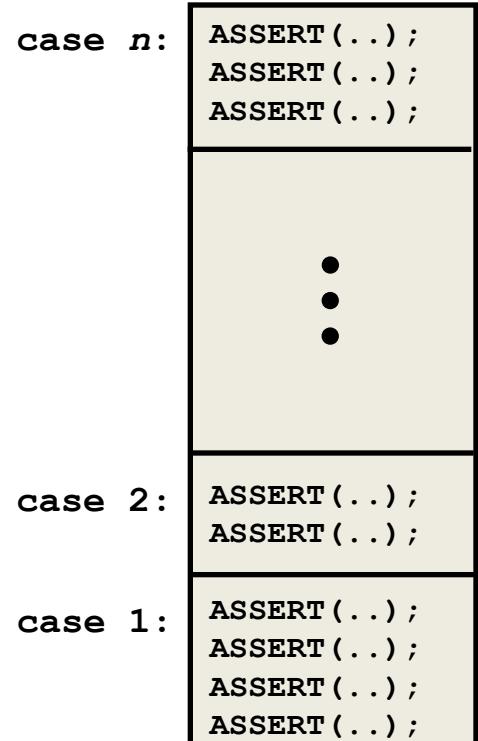


## 5. Negative Testing

# The Component-Level Test Driver

What does a Test Driver comprise?

- Set of consecutively numbered ***test cases***.
- Each *test case* performs some number of individual ***ASSERTIONS***.



## 5. Negative Testing

# The Component-Level Test Driver

What is the Command-Line Interface/Contract?

## 5. Negative Testing

# The Component-Level Test Driver

What is the Command-Line Interface/Contract?

```
testDriver [ testCase# [ addlArgs ... ] ]  
// Run the optionally specified 'testCase#', and  
// the most recently added one otherwise.
```

## 5. Negative Testing

# The Component-Level Test Driver

What is the Command-Line Interface/Contract?

```
testDriver [ testCase# [ addlArgs ... ] ]  
// Run the optionally specified 'testCase#', and  
// the most recently added one otherwise.
```

Return:

|          |                                 |
|----------|---------------------------------|
| 0        | // on success                   |
| positive | // number of assertion failures |
| negative | // test case not found          |

## 5. Negative Testing

# The Component-Level Test Driver

What is the *User Experience*?

## 5. Negative Testing

# The Component-Level Test Driver

What is the *User Experience*?

- A test driver should succeed quietly in production.

## 5. Negative Testing

# The Component-Level Test Driver

What is the *User Experience*?

- A test driver should succeed quietly in production.
- When an error occurs, the test driver should report the offending expression along with the line number:

```
filename(line #): 2 == sqrt(4) (failed)
```

## 5. Negative Testing

# The Component-Level Test Driver

Verbose Mode:

Specifying a (i.e., any) second argument should enable a coarse trace of the activity in the test driver, to facilitate development and debugging:

```
Testing length 0
    without aliasing
    with aliasing
Testing length 1
    without aliasing
    with aliasing
Testing length 2
    ...
    ...
```

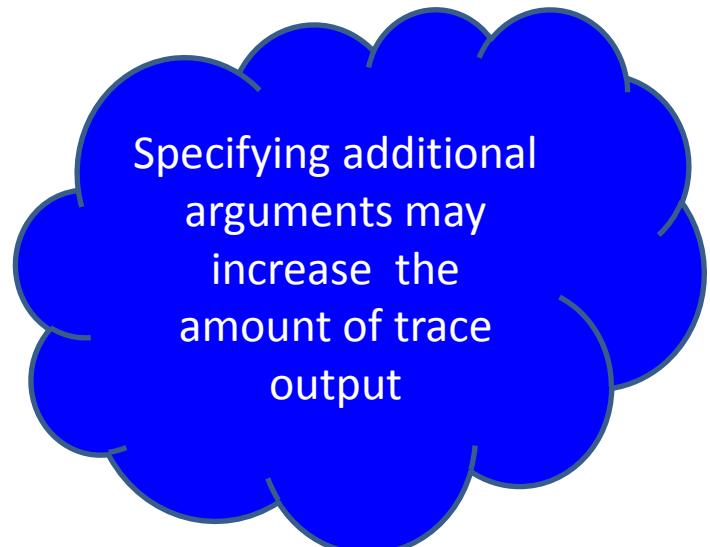
## 5. Negative Testing

# The Component-Level Test Driver

### Verbose Mode:

Specifying a (i.e., any) second argument should enable a coarse trace of the activity in the test driver, to facilitate development and debugging:

```
Testing length 0
    without aliasing
    with aliasing
Testing length 1
    without aliasing
    with aliasing
Testing length 2
    ...
    ...
```



Specifying additional arguments may increase the amount of trace output

# 5. Negative Testing

# Test Driver Layout

## 5. Negative Testing

# Test Driver Layout

```
#include
TEST PLAN
// [ 2] Point(int x, int y)
// [ 1] void setX(int x)
// [ 1] int y() const
// [ 4] void moveBy(int dx, int dy)
// [ 3] void moveTo(int x, int y)
TEST APPARATUS
main(int argc, char argv[]) {
TEST SETUP
    switch (testCase) { case 0:
        case 3: {
            // ...
        }
        case 2: {
            // ...
        }
        case 1: {
            // ...
        }
        default: status = -1;
TEST SHUTDOWN
}
```

- include directives
- test plan identifying case in which each public function has been tested
- ASSERT macro definition, supporting functions, etc.
- common setup for all test cases
- switch on test case number (actual test code goes here)
- any common cleanup code (rare)

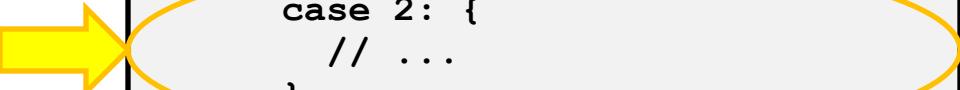
## 5. Negative Testing

# Test Case

```
#include
TEST PLAN
// [ 2] Point(int x, int y)
// [ 1] void setX(int x)
// [ 1] int y() const
// [ 4] void moveBy(int dx, int dy)
// [ 3] void moveTo(int x, int y)
TEST APPARATUS
main(int argc, char argv[])
TEST SETUP

switch (testCase) { case 0:
    case 3: {
        // ...
    }
    case 2: {
        // ...
    }
    case 1: {
        // ...
    }
    default: status = -1;
}
TEST SHUTDOWN
```

- include directives
- test plan identifying case in which each public function has been tested
- ASSERT macro definition, supporting functions, etc.
- common setup for all test cases
- switch on test case number (actual test code goes here)
- any common cleanup code (rare)



## 5. Negative Testing

# Test Case

- **TITLE**
  - Short Label (printed in verbose mode) + optional intro.
- **CONCERNS**
  - Precise (and concise) description of “what could go wrong”  
with this particular implementation.
- **PLAN**
  - How this test case will address each of our concerns.
- **TESTING**
  - Copy-and-paste cross-reference from the overall test plan.

## 5. Negative Testing

### Test Case

- **TITLE**
  - Short Label (printed in verbose mode) + optional intro.
- **CONCERNS**
  - Precise (and concise) description of “what could go wrong”  
with this particular implementation.
- **PLAN**
  - How this test case will address each of our concerns.
- **TESTING**
  - Copy-and-paste cross-reference from the overall test plan.

## 5. Negative Testing

```
    } break;
case 2: {
    //-----
    // UNIQUE BIRTHDAY
    // The value returned for an input of 365 is small.

    // Concerns:
    //: 1 That it can represent the result as a 'double'.
    //: 2 ...
    //:
    //: 6 That the special-case input of 0 returns 1.
    //: 7 ...

    // Plan:
    // Test for explicit values near 0, 365, and 'INT_MAX'.

    // Testing:
    // double uniqueBirthday(int value);
    //-----

    if (verbose) cout << endl << "UNIQUE BIRTHDAY" << endl
        << "======" << endl;

    // ... test code goes here

} break;
case 1: {
```

## 5. Negative Testing

```
    } break;
case 2: {
    //-----
    // UNIQUE BIRTHDAY
    // The value returned for an input of 365 is small.

    // Concerns:
    //: 1 That it can represent the result as a 'double'.
    //: 2 ...
    //:
    //: ...
    //: 6 That the special-case input of 0 returns 1.
    //: 7 ...

    // Plan:
    // Test for explicit values near 0, 365, and 'INT_MAX'.

    // Testing:
    double uniqueBirthday(int value);
    //-----

if (verbose) cout << endl << "UNIQUE BIRTHDAY" << endl
    << "======" << endl;

ASSERT(1 == uniqueBirthday( 0));
ASSERT(1 == uniqueBirthday( 1));
ASSERT(1 >  uniqueBirthday( 2));
// ...
ASSERT(0 <  uniqueBirthday(365));
ASSERT(0 == uniqueBirthday(366));
```

## Test Case

## 5. Negative Testing

# What is Negative Testing?

## 5. Negative Testing

# What is Negative Testing?

*Negative Testing is...*

## 5. Negative Testing

# What is Negative Testing?

*Negative Testing is...*

testing that verifies that unadvertised defensive checks, inserted into library software to guard against client misuse, are functioning *as intended* in the assertion-level build modes in which they *are intended* to be active.

## 5. Negative Testing

# What is Negative Testing?

*Negative Testing is...*

testing that verifies that unadvertised defensive checks, inserted into library software to guard against client misuse, are functioning *as intended* in the assertion-level build modes in which they *are intended* to be active.

## 5. Negative Testing

# What is Negative Testing?

*Negative Testing is...*

testing that verifies that unadvertised defensive checks, inserted into library software to guard against client misuse, are functioning ***as intended*** in the assertion-level build modes in which they ***are intended*** to be active.

## 5. Negative Testing

# What is Negative Testing?

*Negative Testing is...*

testing that verifies that unadvertised defensive checks, inserted into library software to guard against client misuse, are functioning *as intended* in the assertion-level build modes in which they *are intended* to be active.

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is
    - i. Among the last to be proven in practice.

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?

- ❖ Error detection/handling software is
  - i. Among the last to be proven in practice.
  - ii. Among the most critical, when finally needed.

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is
    - i. Among the last to be proven in practice.
    - ii. Among the most critical, when finally needed.
  - 2. If behavior is undefined, how can we test it?

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is
    - i. Among the last to be proven in practice.
    - ii. Among the most critical, when finally needed.
  - 2. If behavior is undefined, how can we test it?
    - a. Recall **hard** vs. **soft** *undefined behavior*: **Soft** undefined behavior isn't necessarily truly undefined.

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is
    - i. Among the last to be proven in practice.
    - ii. Among the most critical, when finally needed.
  - 2. If behavior is undefined, how can we test it?
    - a. Recall **hard** vs. **soft** *undefined behavior*: **Soft** undefined behavior isn't necessarily truly undefined.
    - b. A test driver has white-box knowledge of what and when unadvertised defensive checks are active.

## 5. Negative Testing

# Why Do We need Negative Testing?

Some “silly” questions:

1. Why should we test undefined behavior?
  - ❖ Error detection/handling software is
    - i. Among the last to be proven in practice.
    - ii. Among the most critical, when finally needed.
  - 2. If behavior is undefined, how can we test it?
    - a. Recall **hard** vs. **soft** *undefined behavior*: **Soft** undefined behavior isn't necessarily truly undefined.
    - b. A test driver has white-box knowledge of what and when unadvertised defensive checks are active.
    - c. Precondition checks catch misuse before the problem manifests as **hard** undefined behavior.

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.
  - In doing so we must also avoid **hard** undefined behavior.

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.
  - In doing so we must also avoid **hard** undefined behavior.
2. Negative testing must be especially easy to implement in any testing environment (e.g., ours).

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.
  - In doing so we must also avoid **hard** undefined behavior.
2. Negative testing must be especially easy to implement in any testing environment (e.g., ours).
  - Most developers will not want to invest a lot of time trying to test code that is not supposed to be executed.

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.
  - In doing so we must also avoid **hard** undefined behavior.
2. Negative testing must be especially easy to implement in any testing environment (e.g., ours).
  - Most developers will not want to invest a lot of time trying to test code that is not supposed to be executed.
3. **Negative tests must be easy and efficient to run.**

## 5. Negative Testing

# Negative Testing Requirements

For negative testing to be successful:

1. We need to be able to invoke a function out of contract and observe that the misuse was caught.
  - In doing so we must also avoid **hard** undefined behavior.
2. Negative testing must be especially easy to implement in any testing environment (e.g., ours).
  - Most developers will not want to invest a lot of time trying to test code that is not supposed to be executed.
3. **Negative tests must be easy and efficient to run.**
  - E.g., it is not reasonable to require having a different process for executing each individual negative test.

## 5. Negative Testing

# Negative Testing Requirements

Additional negative-testing concerns:

## 5. Negative Testing

# Negative Testing Requirements

Additional negative-testing concerns:

1. We want to ensure that a contract violation is caught directly by the component under test,

## 5. Negative Testing

# Negative Testing Requirements

Additional negative-testing concerns:

1. We want to ensure that a contract violation is caught directly by the component under test, and not (by chance) by an unadvertised implementation detail of some other component.

## 5. Negative Testing

# Negative Testing Requirements

## Additional negative-testing concerns:

1. We want to ensure that a contract violation is caught directly by the component under test, and not (by chance) by an unadvertised implementation detail of some other component.
2. We must avoid invoking any function out of contract unless the corresponding defensive check is active in the current build mode.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:
  - a. Component filename.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:
  - a. Component filename.
  - b. Source line number.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:
  - a. Component filename.
  - b. Source line number.
  - c. Text representation of expression that failed.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:
  - a. Component filename.
  - b. Source line number.
  - c. Text representation of expression that failed.
  - d. **The assertion level of the defensive check.**

Recent addition, suggested during  
the standardization process.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (`bsls_asserttest`):

1. Create a special-purpose “test” exception (to be thrown by a custom test-assertion-failure handler) that captures:
  - a. Component filename.
  - b. Source line number.
  - c. Text representation of expression that failed.
2. Create a test utility component defining a suite of static methods and supporting macros to facilitate negative testing.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (FOR EACH TEST DRIVER):

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.
- II. At the end of each test case for a function that has a *narrow* contract:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

## Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.
- II. At the end of each test case for a function that has a *narrow* contract:
  1. Install our custom *test* failure handler.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

## Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.
- II. At the end of each test case for a function that has a *narrow* contract:
  1. Install our custom *test* failure handler.
  2. Call the function on either side of the boundaries of defined behavior.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

## Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.
- II. At the end of each test case for a function that has a *narrow* contract:
  1. Install our custom *test* failure handler.
  2. Call the function on either side of the boundaries of defined behavior.
  3. Observe that the assertions fire (or not) as expected when active in the current build mode.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

## Plan (FOR EACH TEST DRIVER):

- I. Assume that our standard ASSERT test macro is available in the test driver to process any negative testing errors.
- II. At the end of each test case for a function that has a *narrow* contract:
  1. Install our custom *test* failure handler.
  2. Call the function on either side of the boundaries of defined behavior.
  3. Observe that the assertions fire (or not) as expected when active in the current build mode.
  4. Don't worry about expected results (covered earlier).

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n)` ;  $0 \leq n \leq 100$

## 5. Negative Testing

# Using the bsls\_asserttest Component

Example: factorial (n) ; 0 <= n <= 100

Add the following at the end of the test case:

## 5. Negative Testing

# Using the bsls\_asserttest Component

Example: factorial (n) ; 0 <= n <= 100

Add the following at the end of the test case:

```
bsls_Assert::setFailureHandler(  
    &bsls_AssertTest::failTestDriver);
```

## 5. Negative Testing

# Using the bsls\_asserttest Component

Example: factorial (n) ; 0 <= n <= 100

Add the following at the end of the test case:

```
bsls Assert::setFailureHandler  
        (&bsls AssertTest::failTestDriver);
```

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial( 0));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial(100));  
BSLS_ASSERTTEST_ASSERT_FAIL(factorial(101));
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

Add the following at the end of the test case:

```
bsls Assert::setFailureHandler  
        (&bsls AssertTest::failTestDriver);
```

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial( 0));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial(100));  
BSLS_ASSERTTEST_ASSERT_FAIL(factorial(101));
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

Add the following at the end of the test case:

```
bsls Assert::setFailureHandler  
        (&bsls AssertTest::failTestDriver);
```

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial( 0));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial(100));  
BSLS_ASSERTTEST_ASSERT_FAIL(factorial(101));
```

## 5. Negative Testing

# Using the bsls\_asserttest Component

Example: factorial (n) ; 0 <= n <= 100

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));
```

Which effectively expands to the following:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));
```

Which effectively expands to the following:

```
#if BSLS_ASSERT_IS_ACTIVE
try {
    factorial( -1);
    ASSERT(false); // should not get here
}
catch(const bsls AssertTestException& e) {
    ASSERT(0 == strcmp("our_mathutil.cpp",
                       e.filename()));
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

Add the following at the end of the test case:

```
bsls Assert::setFailureHandler  
        (&bsls AssertTest::failTestDriver);
```

```
BSLS_ASSERTTEST_ASSERT_FAIL(factorial( -1));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial(  0));  
BSLS_ASSERTTEST_ASSERT_PASS(factorial(100));  
BSLS_ASSERTTEST_ASSERT_FAIL(factorial(101));
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

Which happens to expand to the following C++ code:

```
bsls_assert::setFailureHandler(
    &bslsAssertTest::failTestDriver);

#ifndef BSLS_ASSERT_IS_ACTIVE
try {
    factorial(-1);
    ASSERT(false); // should not get here
}
catch(const bsls AssertTestException& e) {
    ASSERT(0 == strcmp("our_mathutil.cpp", e.filename()));
}
#endif
factorial(0);
factorial(100);
#ifndef BSLS_ASSERT_IS_ACTIVE
try {
    factorial(101);
    ASSERT(false); // should not get here
}
catch(const bsls AssertTestException& e) {
    ASSERT(0 == strcmp("our_mathutil.cpp", e.filename()));
}
#endif
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example: `factorial(n); 0 <= n <= 100`

Which happens to expand to the following C++ code:

```
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
  
#if BSLS_ASSERT_IS_ACTIVE  
try {  
    factorial(-1);  
    ASSERT(false); // should not get here  
}  
catch(const bsls AssertTestException& e) {  
    ASSERT(0 == strcmp("our_mathutil.cpp", e.filename()));  
}  
#endif  
factorial(0);  
factorial(100);  
  
#if BSLS_ASSERT_IS_ACTIVE  
try {  
    factorial(101);  
    ASSERT(false); // should not get here  
}  
catch(const bsls AssertTestException& e) {  
    ASSERT(0 == strcmp("our_mathutil.cpp", e.filename()));  
}  
#endif
```

Implementation Experience Observation

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Implementation Experience Observation:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Implementation Experience Observation:

If the “positive” test changes object state (e.g., allocates memory), it typically must run in all build modes or clean-up code becomes too complicated.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Implementation Experience Observation:

If the “positive” test changes object state (e.g., allocates memory), it typically must run in all build modes or clean-up code becomes too complicated.

For example,

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Implementation Experience Observation:

If the “positive” test changes object state (e.g., allocates memory), it typically must run in all build modes or clean-up code becomes too complicated.

For example,

`BSLS_ASSERTTEST_ASSERT_SAFE_FAIL`

is disabled unless we are in “safe” mode.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Implementation Experience Observation:

If the “positive” test changes object state (e.g., allocates memory), it typically must run in all build modes or clean-up code becomes too complicated.

For example,

`BSLS_ASSERTTEST_ASSERT_SAFE_FAIL`

is disabled unless we are in “safe” mode.

`BSLS_ASSERTTEST_ASSERT_SAFE_PASS` always runs.

## 5. Negative Testing

# Using the `bsl::asserttest` Component

Example: `bsl::vector<int> v;`

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example:

```
bsl::vector<int> v;  
// ...
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example:

```
bsl::vector<int> v;
// ...
bsls_assert::setFailureHandler(
    &bsls AssertTest::failTestDriver);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Example:

```
bsl::vector<int> v;
// ...
bsls_assert::setFailureHandler(
    &bslsAssertTest::failTestDriver);
vector<int> v;
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);  
v.push_back(42);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);  
v.push_back(42);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[1]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[2]);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);  
v.push_back(42);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[1]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[2]);  
v.pop_back();
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
Example: bsl::vector<int> v;  
// ...  
bsls_assert::setFailureHandler(  
    &bslsAssertTest::failTestDriver);  
vector<int> v;  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[0]);  
v.push_back(9);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);  
v.push_back(42);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[1]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[2]);  
v.pop_back();  
BSLS_ASSERTTEST_ASSERT_SAFE_PASS(v[0]);  
BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(v[1]);
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first
  - i. Correctly detecting misuse, but

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first
  - i. Correctly detecting misuse, but
  - ii. Returning the wrong filename.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first
  - i. Correctly detecting misuse, but
  - ii. Returning the wrong filename.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d));
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first
  - i. Correctly detecting misuse, but
  - ii. Returning the wrong filename.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late!
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first

- i. Correctly detecting misuse, but
- ii. Returning the wrong filename.

This **contract violation** could lead to **hard undefined behavior**.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late!
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

### 1. An initializer-list may assert first

- i. Correctly detecting misuse, but
- ii. Returning the wrong filename.

This **contract violation** could lead to **hard undefined behavior**.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late?
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

### 1. An initializer-list may assert first

- i. Correctly detecting misuse, but
- ii. Returning the wrong filename.

This **contract violation** could lead to **hard undefined behavior**.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late?
}
```

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

### 1. An initializer-list may assert first

- i. Correctly detecting misuse, but
- ii. Returning the wrong filename.

This **contract violation** could lead to **hard undefined behavior**.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late?
}
```

Suppose

MyDateImpUtil::toSerial  
throws our “test” exception?

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

### 1. An initializer-list may assert first

- i. Correctly detecting misuse, but
- ii. Returning the wrong filename.

This **contract violation** could lead to **hard undefined behavior**.

```
MyDate::MyDate(int y, int m, int d)
: d_serialDate(MyDateImpUtil::toSerial(y, m, d))
{
    BSLS_ASSERT(MyDate::isValid(y, m, d)); // <- Too late?
}
```

But the component filename  
is not what we expected.

Suppose  
MyDateImpUtil::toSerial  
throws our “test” exception?

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Constructors present a unique problem:

1. An initializer-list may assert first
  - i. Correctly detecting misuse, but
  - ii. Returning the wrong filename.
2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
bsls_assert::setFailureHandler(&bsls_asserttest::failTest);
```

2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
bsls_assert::setFailureHandler(&bsls_asserttest::failTest);  
BSLS_ASSERTTEST_ASSERT_PASS_RAW(MyDate( 2000, 7, 15));
```

2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the bsls\_asserttest Component

```
bsls_assert::setFailureHandler(&bsls_asserttest::failTest);  
  
BSLS_ASSERTTEST_ASSERT_PASS_RAW(MyDate( 2000, 7, 15));  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 0, 7, 15));  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate(10000, 7, 15));
```

2. We address this issue by providing a special version of each macro (each ending in \_RAW) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
bsls_assert::setFailureHandler(&bsls_asserttest::failTest);  
  
BSLS_ASSERTTEST_ASSERT_PASS_RAW(MyDate( 2000, 7, 15);  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 0, 7, 15);  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate(10000, 7, 15);  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 0, 15);  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 13, 15);
```

2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

```
bsls_assert::setFailureHandler(&bsls_asserttest::failTest);  
  
BSLS_ASSERTTEST_ASSERT_PASS_RAW(MyDate( 2000, 7, 15);  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 0, 7, 15);  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate(10000, 7, 15);  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 0, 15);  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 13, 15);  
  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 7, 0);  
BSLS_ASSERTTEST_ASSERT_FAIL_RAW(MyDate( 2000, 7, 32));
```

2. We address this issue by providing a special version of each macro (each ending in `_RAW`) that relaxes only this one aspect of the check.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Negative Testing of `SAFE_2`-Mode-Only  
defensive checks.

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Negative Testing of `SAFE_2`-Mode-Only  
defensive checks.

Just as in the source code itself, test code must wrap any negative tests that should be active in only “Safe 2” mode:

## 5. Negative Testing

# Using the `bsls_asserttest` Component

Negative Testing of `SAFE_2`-Mode-Only  
defensive checks.

Just as in the source code itself, test code must wrap any negative tests that should be active in only “Safe 2” mode:

```
#if defined(BSLS_BUILD_TARGET_SAFE_2)
    BSLS_ASSERTTEST_ASSERT_SAFE_FAIL(...);
    BSLS_ASSERTTEST_ASSERT_SAFE_PASS(...);
#endif
```

5. Negative Testing

End of Section

Questions?

## 5. Negative Testing

# What Questions are we Answering?

- What is **negative testing**, and why is it important?
- Why is it that we can **reliably test** our own (**library**) **undefined behavior**?
- Why are **assertion-level** build modes important for negative testing?
- Why is it that we **always execute** the "positive" tests, but not the "negative" ones?
- Why is it that we **don't care about** expected values/results during the negative testing phase?
- What is it about **constructors** that makes them **uniquely problematic**, and how do we deal with them?
- How do we negatively test Safe-2-Mode-Only checks?

# Late Breaking News...

# Late Breaking News...

The C++ Standards Committee has adopted a new exception-specification construct: noexcept.

# Late Breaking News...

The C++ Standards Committee has adopted a new exception-specification construct: `noexcept`.

1. It was added to support move semantic operations, all of which have *wide contracts*.

# Late Breaking News...

The C++ Standards Committee has adopted a new exception-specification construct: `noexcept`.

1. It was added to support move semantic operations, all of which have *wide* contracts.
2. Decorating arbitrary functions with `noexcept` may yield speed/space optimizations.

# Late Breaking News...

The C++ Standards Committee has adopted a new exception-specification construct: `noexcept`.

1. It was added to support move semantic operations, all of which have *wide* contracts.
2. Decorating arbitrary functions with `noexcept` may yield speed/space optimizations.
3. Attempting to throw through a function decorated with `noexcept` terminates the program!

# Late Breaking News...

The C++ Standards Committee has adopted a new exception-specification construct: `noexcept`.

1. It was added to support move semantic operations, all of which have *wide* contracts.
2. Decorating arbitrary functions with `noexcept` may yield speed/space optimizations.
3. Attempting to throw through a function decorated with `noexcept` terminates the program!

How is all this relevant to *defensive programming* in general, and *negative testing* in particular?

# Later Breaking News...

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow contracts*, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

Does anyone  
see a problem?

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow contracts*, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow contracts*, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).

# Later Breaking News...

The C++ Standards Committee was eager to apply `noexcept` liberally throughout the library, including for functions having *narrow contracts*, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide contract*, and the operation must never throw.

# Later Breaking News...

The C++ Standards Committee was eager to apply `noexcept` liberally throughout the library, including for functions having *narrow contracts*, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide contract*, and the operation must never throw.
- c. Use wherever the operation must never throw in contract.

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow* contracts, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide* contract, and the operation must never throw.
- c. Use wherever the operation must never throw in contract. **(Over my dead body!)**

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow* contracts, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide* contract, and the operation must never throw.
- c. Use wherever the operation must never throw in contract. **(Over my dead body!)**

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow* contracts, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide* contract, and the operation must never throw.
- c. Use wherever the operation must never throw in contract. **(Over my dead body!)**

# Later Breaking News...

The C++ Standards Committee was eager to apply noexcept liberally throughout the library, including for functions having *narrow* contracts, such as `vector<T>::front()` and `vector<T>::operator[](size_t)`.

On 2011-Mar-21, I proposed three sets of guidelines:

- a. Use only for move constructors and move assignment (i.e., for semantics only).
- b. Use wherever there is a *wide* contract, and the operation must never throw. **(75% consensus)**
- c. Use wherever the operation must never throw in contract. **(Over my dead body!)**

# Latest Breaking News...

# Latest Breaking News...

The entire C++ Standards Committee spent the remainder of the week carefully applying these criteria for the use of noexcept to literally every function in the C++ Standard Library.

# Latest Breaking News...

The entire C++ Standards Committee spent the remainder of the week carefully applying these criteria for the use of noexcept to literally every function in the C++ Standard Library.

We finished a day early:

# Latest Breaking News...

The entire C++ Standards Committee spent the remainder of the week carefully applying these criteria for the use of noexcept to literally every function in the C++ Standard Library.

We finished a day early:

The new standard was approved by the full committee in Madrid on Friday, March 25, 2011!

# Latest Breaking News...

The entire C++ Standards Committee spent the remainder of the week carefully applying these criteria for the user-defined casts to literally every function in the C++ Standard Library.

# IT'S DONE!

We finished a day early:

And C++14 Conforms To  
These Guidelines As Well!

# Conclusion

1. Brief Review of Physical Design

2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

3. ‘Good’ Contracts

Defensive Programming (*Narrow* versus *Wide* Contracts)

4. Implementing Defensive Checks

Using the **bsls\_assert** Component

5. Negative Testing

Using the **bsls\_asserttest** Component

# Conclusion

## 1. Brief Review of Physical Design

Components, Logical Relationships, & Physical Dependencies

# Conclusion

## 1. Brief Review of Physical Design

Components, Logical Relationships, & Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.

# Conclusion

## 1. Brief Review of Physical Design

Components, Logical Relationships, & Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- It comprises a .h and .cpp file, along with its associated component-level test driver (.t.cpp).

# Conclusion

## 1. Brief Review of Physical Design

Components, Logical Relationships, & Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- It comprises a .h and .cpp file, along with its associated component-level test driver (.t.cpp).
- Logical relationships among classes imply physical ones between components.

# Conclusion

## 1. Brief Review of Physical Design

Components, Logical Relationships, & Physical Dependencies

- A *Component* is the fundamental unit of both *logical* and *physical* software design.
- It comprises a `.h` and `.cpp` file, along with its associated component-level test driver (`.t.cpp`).
- Logical relationships among classes imply physical ones between components.
- *Level numbers* characterize acyclic physical designs.

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

- An *interface* is *syntactic*; a *contract* is *semantic*.

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined* Behavior

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined Behavior*

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.
- *Undefined Behavior* if a precondition not met.

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined Behavior*

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.
- *Undefined Behavior* if a precondition not met.
- What undefined behavior does is undefined!

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined Behavior*

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.
- *Undefined Behavior* if a precondition not met.
- What undefined behavior does **is undefined!**
- Documented *Essential Behavior* must not change!

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined Behavior*

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.
- *Undefined Behavior* if a precondition not met.
- What undefined behavior does **is undefined!**
- Documented *Essential Behavior* must not change!
- *Test drivers* verify essential behavior.

# Conclusion

## 2. Interfaces and Contracts

Syntax versus Semantics: *Essential/Undefined Behavior*

- An *interface* is *syntactic*; a *contract* is *semantic*.
- The *contract* defines the *pre/postconditions*.
- *Undefined Behavior* if a precondition not met.
- What undefined behavior does **is undefined!**
- Documented *Essential Behavior* must not change!
- *Test drivers* verify essential behavior.
- *Assertions in destructors* help verify *invariants*.

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (*Narrow versus Wide Contracts*)

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*
- *Undefined Behavior is GOOD:*

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*
- *Undefined Behavior is GOOD:*
  - Reduces costs associated with development/testing.

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*
- *Undefined Behavior is GOOD:*
  - Reduces costs associated with development/testing.
  - Improves performance and reduces object-code size.

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*
- *Undefined Behavior is GOOD:*
  - Reduces costs associated with development/testing.
  - Improves performance and reduces object-code size.
  - Enables practical/effective Defensive Programming.

# Conclusion

## 3. ‘Good’ Contracts

Defensive Programming (Narrow versus Wide Contracts)

- *Defensive programming means fault intolerance!*
- *Narrow contracts imply undefined behavior.*
- *Undefined Behavior is GOOD:*
  - Reduces costs associated with development/testing.
  - Improves performance and reduces object-code size.
  - Enables practical/effective Defensive Programming.
  - Allows useful behavior to be added as needed.

# Conclusion

## 4. Implementing Defensive Checks Using the `bsls_assert` Component

# Conclusion

## 4. Implementing Defensive Checks

Using the `bsls_assert` Component

- The application owner knows best:

# Conclusion

## 4. Implementing Defensive Checks

Using the **bsls\_assert** Component

- The application owner knows best:
  - How much time to spend checking preconditions.

# Conclusion

## 4. Implementing Defensive Checks

Using the `bsls_assert` Component

- The application owner knows best:
  - How much time to spend checking preconditions.
  - What to do if a violation is detected.

# Conclusion

## 4. Implementing Defensive Checks

Using the **bsls\_assert** Component

- The application owner knows best:
  - How much time to spend checking preconditions.
  - What to do if a violation is detected.
- The **bsls\_assert** component enables application and library developers to collaborate:

# Conclusion

## 4. Implementing Defensive Checks

Using the **bsls\_assert** Component

- The application owner knows best:
  - How much time to spend checking preconditions.
  - What to do if a violation is detected.
- The **bsls\_assert** component enables application and library developers to collaborate:
  - Enjoying the benefits of DP, without classical drawbacks.

# Conclusion

## 5. Negative Testing

Using the **bsls\_asserttest** Component

# Conclusion

## 5. Negative Testing

Using the **bsls\_asserttest** Component

- Testing defensive checks is part of getting it right!

# Conclusion

## 5. Negative Testing

Using the **bsls\_asserttest** Component

- Testing defensive checks is part of getting it right!
- The **bsls\_asserttest** component makes negative testing practical, easy, and efficient.

# Conclusion

## 5. Negative Testing

Using the **bsls\_asserttest** Component

- Testing defensive checks is part of getting it right!
- The **bsls\_asserttest** component makes negative testing practical, easy, and efficient.
- Care must be taken to ensure we test defensive checks only when they are supposed to be active.

# Conclusion

## 5. Negative Testing

Using the **bsls\_asserttest** Component

- Testing defensive checks is part of getting it right!
- The **bsls\_asserttest** component makes negative testing practical, easy, and efficient.
- Care must be taken to ensure we test defensive checks only when they are supposed to be active.
- Testing constructors is uniquely problematic.

## Conclusion

noexcept

**WE GOT IT RIGHT!**

# Conclusion

- See **N4075**: Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)

# Conclusion

- See **N4075**: Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)
- Find our open-source distribution at:  
<http://www.openbloombg.com/bsl>
- Moderator: [kpfleming@bloomberg.net](mailto:kpfleming@bloomberg.net)
- How to contribute? *See our site.*
- All comments and criticisms welcome...

# Conclusion

- See **N4075**: Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)
- Find our open-source distribution at:  
<http://www.openbloombg.com/bsl>
- Moderator: [kpfleming@bloomberg.net](mailto:kpfleming@bloomberg.net)
- How to contribute? *See our site.*
- All comments and criticisms welcome...

# The End