Checking Contracts with BDE

Joshua Berne - jberne4@bloomberg.net

2020-06-10

Copyright Notice

- Let's say you want to write a function.

- Let's say you want to write a function.

```
int foo(int x, int y);
```

- Let's say you want to write a function.
- With a contract.

```
int foo(int x, int y);
```

- Let's say you want to write a function.
- With a contract.

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you want to write a function.
- With a contract.
- That is a narrow contract.

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you want to write a function.
- With a contract.
- That is a narrow contract.

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were. The behavior is undefined
  // unless 'x <= y'.
```

- Assume `foo` is in a new library.

```
#include <foo.h>


int foo(int x, int y)
{

  return fooability(x) * fooability(y);
}
```

- Assume `foo` is in a new library.
- We can assert our preconditions.

```
#include <foo.h>
#include <bsls_assert.h>

int foo(int x, int y)
{
  BSLS_ASSERT(x <= y);
  return fooability(x) * fooability(y);
}
```

- Assume `foo` is in a new library.
- We can assert our preconditions.
- We can invoke our function.

- Assume `foo` is in a new library.
- We can assert our preconditions.
- We can invoke our function.

```
#include <foo.h>
#include <bsl_iostream.h>

int main()
{
  int fooishness = foo(3,5);
  bsl::cout << "My Fooishness is:" << fooishness << bsl::endl;
  return 0;
}
```

- Assume `foo` is in a new library.
- We can assert our preconditions.
- We can invoke our function.
- ... or have a very bad bug.

```
#include <foo.h>

int main()
{
  int fooishness = foo(5,3);
  bsl::cout << "My Fooishness is:" << fooishness << bsl::endl;
  return 0;
}
```

- We can build our code with assertions enabled.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o badmain.tsk foo.cpp badmain.cpp
```

- We can build our code with assertions enabled.
- Then run it.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o badmain.tsk foo.cpp badmain.cpp
$ ./badmain.tsk
```

- We can build our code with assertions enabled.
- Then run it.
- And kabooom.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o badmain.tsk foo.cpp badmain.cpp
$ ./badmain.tsk
FATAL foo.cpp:6 Assertion failed: x <= y
Aborted (core dumped)
```

What Can a violation handler do?

- Violations can be configured to do a number of things

What Can a violation handler do?

- Violations can be configured to do a number of things

```
typedef void (*ViolationHandler)(const AssertViolation&);
bsls::Assert:setViolationHandler(Assert::ViolationHandler function);
```

What Can a violation handler do?

- Violations can be configured to do a number of things

```
typedef void (*ViolationHandler)(const AssertViolation&);
bsls::Assert:setViolationHandler(Assert::ViolationHandler function);
```

- Handlers exist to abort, sleep, log, throw, or write your own.

What Can a violation handler do?

- Violations can be configured to do a number of things

```
typedef void (*ViolationHandler)(const AssertViolation&);
bsls::Assert:setViolationHandler(Assert::ViolationHandler function);
```

- Handlers exist to abort, sleep, log, throw, or write your own.
- The default handler aborts. Use that.

What *SHOULD* a violation handler do?

What *SHOULD* a violation handler do?

- In production code: Abort.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.
  - Best case is a quick failure with bad values

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.
  - Best case is a quick failure with bad values
  - Worst case is silent failure and spreading corruption.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.
  - Best case is a quick failure with bad values
  - Worst case is silent failure and spreading corruption.
- Fast failure has well known cost.

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.
  - Best case is a quick failure with bad values
  - Worst case is silent failure and spreading corruption.
- Fast failure has well known cost.
  - Failures will be caught and escalated

What *SHOULD* a violation handler do?

- In production code: Abort.
- In development code: Abort.
- In unit tests: Abort. or throw when explicitly tested. (see `BSLS_ASSERTTEST`)
- The source of a violation may be close or far.
- The risk when continuing is unbounded.
  - Best case is a quick failure with bad values
  - Worst case is silent failure and spreading corruption.
- Fast failure has well known cost.
  - Failures will be caught and escalated
  - Software will not attempt to execute in a corrupt state

Writing A New Library

Writing A New Library

- In a new library, use `BSLS_ASSERT`

Writing A New Library

- In a new library, use `BSLS_ASSERT`
- In a new function, use `BSLS_ASSERT`

Writing A New Library

- In a new library, use `BSLS_ASSERT`
- In a new function, use `BSLS_ASSERT`
- When deploying a new application, use `BSLS_ASSERT`

Writing A New Library

- In a new library, use `BSLS_ASSERT`
- In a new function, use `BSLS_ASSERT`
- When deploying a new application, use `BSLS_ASSERT`
- Catch errors fast, run safer systems.

Checks In Old Code

Checks In Old Code

- New checks in old code? use `BSLS_REVIEW`

Checks In Old Code

- New checks in old code? use `BSLS_REVIEW`
- Enabling dormant checks in old code? use `BSLS_REVIEW_LEVEL...`

Checks In Old Code

- New checks in old code? use `BSLS_REVIEW`
- Enabling dormant checks in old code? use `BSLS_REVIEW_LEVEL...`
- Safely roll out checks before enforcing them.

- Let's say you wrote a function long ago.

- Let's say you wrote a function long ago.

```
int foo(int x, int y);
```

- Let's say you wrote a function long ago.
- With a contract.

```
int foo(int x, int y);
```

- Let's say you wrote a function long ago.
- With a contract.

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.
  - Returns a value out of range

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.
  - Returns a value out of range
  - Writes and doesn't delete a large temporary file on disk

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.
  - Returns a value out of range
  - Writes and doesn't delete a large temporary file on disk
  - Takes seconds to complete instead of microseconds

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.
  - Returns a value out of range
  - Writes and doesn't delete a large temporary file on disk
  - Takes seconds to complete instead of microseconds
- All problems that could be going unnoticed in production

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were.
```

- Let's say you wrote a function long ago.
- With a contract.
- But it fails badly and subtly if `x <= y`, so you want to narrow the contract.
  - Returns a value out of range
  - Writes and doesn't delete a large temporary file on disk
  - Takes seconds to complete instead of microseconds
- All problems that could be going unnoticed in production
- So you want to narrow the contract

```
int foo(int x, int y);
  // Do some foo with the specified 'x' and 'y'.  Return
  // how fooable they were. The behavior is undefined
  // unless 'x <= y'.
```

- Assume `foo` is old as dirt.

```
#include <foo.h>


int foo(int x, int y)
{

  return fooability(x) * fooability(y);
}
```

- Assume `foo` is old as dirt.
- We can review our new preconditions.

```
#include <foo.h>
#include <bsls_review.h>

int foo(int x, int y)
{
  BSLS_REVIEW(x <= y);
  return fooability(x) * fooability(y);
}
```

- Assume `foo` is old as dirt.
- We can review our new preconditions.
- Eventually, we migrate on to `BSLS_ASSERT`.

```
#include <foo.h>
#include <bsls_assert.h>

int foo(int x, int y)
{
  BSLS_ASSERT(x <= y);
  return fooability(x) * fooability(y);
}
```

- Assume `foo` is old as dirt.
- We can review our new preconditions.
- Eventually, we migrate on to `BSLS_ASSERT`.
- But that can't be released safely.

KABOOM

- We can build our code with assertions enabled

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o questionablemain.tsk
  oldfoo.cpp questionablemain.cpp
```

- We can build our code with assertions enabled
- This will enable reviews of the same type too.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o questionablemain.tsk
  oldfoo.cpp questionablemain.cpp
```

- We can build our code with assertions enabled
- This will enable reviews of the same type too.
- Then run it.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o questionablemain.tsk
  oldfoo.cpp questionablemain.cpp
$ ./questionablemain.tsk
```

- We can build our code with assertions enabled
- This will enable reviews of the same type too.
- Then run it.
- And oopsie.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o questionablemain.tsk
  oldfoo.cpp questionablemain.cpp
$ ./questionablemain.tsk
ERROR oldfoo.cpp:6 BSLS_REVIEW failure: (level:R-DBG) 'x <= y'
Please run "/bb/bin/showfunc.tsk ./questionablemain.tsk 8048B28
8048A07 8048A26" to see the stack trace.
My Fooishness is: 171717
```

- We can build our code with assertions enabled
- This will enable reviews of the same type too.
- Then run it.
- And oopsie.
- ...with a stack trace!.

```
$ /bb/bin/showfunc.tsk ./questionablemain.tsk 8048B28 8048A07
  8048A26
0x8048b28 _ZN11BloombergLP4bsls6Review9failByLogERKNS0
  _15ReviewViolationE + 88
0x8048a07 _Z3fooii + 81
0x8048a26 main + 26
```

- We can build our code with assertions enabled
- This will enable reviews of the same type too.
- Then run it.
- And oopsie.
- ...with a stack trace!.
- ... or a more readable stack trace!.

```
$ /bb/bin/showfunc.tsk ./questionablemain.tsk 8048B28 8048A07
  8048A26 | c++filt
0x8048b28 BloombergLP::bsls::Review::failByLog(
  BloombergLP::bsls::ReviewViolation const&) + 88
0x8048a07 foo(int, int) + 81
0x8048a26 main + 26
```

Tickets

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.
- Most will result in a DRQS created for the owner of the code that failed.

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.
- Most will result in a DRQS created for the owner of the code that failed.
- Tickets are throttled, only `act.log` is monitored, and GUTS routing can be inaccurate.

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.
- Most will result in a DRQS created for the owner of the code that failed.
- Tickets are throttled, only `act.log` is monitored, and GUTS routing can be inaccurate.
  - Keep your PWHO entries and procmgr ids set up correctly

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.
- Most will result in a DRQS created for the owner of the code that failed.
- Tickets are throttled, only `act.log` is monitored, and GUTS routing can be inaccurate.
  - Keep your PWHO entries and procmgr ids set up correctly
  - Monitor your own logs too.

Tickets

- Every `BSLS_REVIEW` failure logged to `act.log` will be processed by GUTS.
- Most will result in a DRQS created for the owner of the code that failed.
- Tickets are throttled, only `act.log` is monitored, and GUTS routing can be inaccurate.
  - Keep your PWHO entries and procmgr ids set up correctly
  - Monitor your own logs too.

- You might be running with asserts off

```
$ g++ -DBSLS_ASSERT_LEVEL_NONE -o badmain.tsk foo.cpp
  badmain.cpp
```

- You might be running with asserts off
- If you switch to this, things might go boom

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o badmain.tsk foo.cpp
  badmain.cpp
```

- You might be running with asserts off
- If you switch to this, things might go boom
- This will switch asserts to reviews first

```
$ g++ -DBSLS_ASSERT_LEVEL_NONE -DBSLS_REVIEW_LEVEL_REVIEW
  -o badmain.tsk foo.cpp badmain.cpp
```

- You might be running with asserts off
- If you switch to this, things might go boom
- This will switch asserts to reviews first
- Now deploy, monitor, fix bugs

```
$ g++ -DBSLS_ASSERT_LEVEL_NONE -DBSLS_REVIEW_LEVEL_REVIEW
  -o badmain.tsk foo.cpp badmain.cpp
```

- You might be running with asserts off
- If you switch to this, things might go boom
- This will switch asserts to reviews first
- Now deploy, monitor, fix bugs
- Then switch to this.

```
$ g++ -DBSLS_ASSERT_LEVEL_ASSERT -o badmain.tsk foo.cpp
  badmain.cpp
```

Documentation

- BDE Starting point: `http://bde.bloomberg.com/`

Documentation

- BDE Starting point: `http://bde.bloomberg.com/`
- `BSLS_ASSERT`: `https://bde.bloomberg.com/bde-resources/doxygen/bde_api_prod/group__bsls__assert.html`

Documentation

- BDE Starting point: `http://bde.bloomberg.com/`
- `BSLS_ASSERT`: `https://bde.bloomberg.com/bde-resources/doxygen/bde_api_prod/group__bsls__assert.html`
- `BSLS_REVIEW`: `https://bde.bloomberg.com/bde-resources/doxygen/bde_api_prod/group__bsls__review.html`

Source Code

- BDE Starting point: `http://bbgithub.dev.bloomberg.com/bde/bde/`

Source Code

- BDE Starting point: `http://bbgithub.dev.bloomberg.com/bde/bde/`
- Or Open Source: `https://github.com/bloomberg/bde`

Source Code

- BDE Starting point: `http://bbgithub.dev.bloomberg.com/bde/bde/`
- Or Open Source: `https://github.com/bloomberg/bde`
- `BSLS_ASSERT`: `https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_assert.h`

Source Code

- BDE Starting point: `http://bbgithub.dev.bloomberg.com/bde/bde/`
- Or Open Source: `https://github.com/bloomberg/bde`
- `BSLS_ASSERT`: `https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_assert.h`
- `BSLS_REVIEW`: `https://github.com/bloomberg/bde/blob/master/groups/bsl/bsls/bsls_review.h`

Things to Learn

- Build Targets: `BDE_BUILD_TARGET_OPT`, `BDE_BUILD_TARGET_DBG`, `BDE_BUILD_TARGET_SAFE`.

Things to Learn

- Build Targets: `BDE_BUILD_TARGET_OPT`, `BDE_BUILD_TARGET_DBG`, `BDE_BUILD_TARGET_SAFE`.
- Build Levels, other macros: `BSLS_ASSERT_LEVEL_*`, `BSLS_REVIEW_LEVEL_*`, `BSLS_ASSERT_*`, `BSLS_REVIEW_*`.

Things to Learn

- Build Targets: `BDE_BUILD_TARGET_OPT`, `BDE_BUILD_TARGET_DBG`, `BDE_BUILD_TARGET_SAFE`.
- Build Levels, other macros: `BSLS_ASSERT_LEVEL_*`, `BSLS_REVIEW_LEVEL_*`, `BSLS_ASSERT_*`, `BSLS_REVIEW_*`.
- Testing: `BSLS_ASSERTTEST`