Contracts: What we are doing here

P3343R0

Joshua Berne - jberne4@bloomberg.net

2024-06-25

What are Contracts?

What are Contracts?

- Agreements between multiple parties

What are Contracts?

- Agreements between multiple parties
  - Implementers and Users of a function or library

What are Contracts?

- Agreements between multiple parties
  - Implementers and Users of a function or library
  - Programmers and the platform they are working on

What are Contracts?

- Agreements between multiple parties
  - Implementers and Users of a function or library
  - Programmers and the platform they are working on
  - Users and the programs they run

What are Contracts?

- Agreements between multiple parties
  - Implementers and Users of a function or library
  - Programmers and the platform they are working on
  - Users and the programs they run
- Written (or implicit) in plain language

What are Contracts?

- Agreements between multiple parties
  - Implementers and Users of a function or library
  - Programmers and the platform they are working on
  - Users and the programs they run
- Written (or implicit) in plain language
- Contracts define what is and is not correct behavior

What is a Correct program?

What is a Correct program?

- One which violates no contracts on any input

What is a Correct program?

- One which violates no contracts on any input
- Has no behavior not defined by the platform on any input

What is a Correct program?

- One which violates no contracts on any input
- Has no behavior not defined by the platform on any input
- Must be well-formed

What is a Correct program evaluation?

What is a Correct program evaluation?

- An evaluation of a program (with specific inputs) that violates no contracts

What is a Correct program evaluation?

- An evaluation of a program (with specific inputs) that violates no contracts
- Has no behavior not defined by the platform

What is an Incorrect program?

What is an Incorrect program?

- One which will violate a contract on certain inputs

What is an Incorrect program?

- One which will violate a contract on certain inputs
- Still potentially a well-formed program

What is a Contract Check?

What is a Contract Check?

- An algorithm to identify when a contract has been violated

What is a Contract Check?

- An algorithm to identify when a contract has been violated
  - `x > 0`

What is a Contract Check?

- An algorithm to identify when a contract has been violated
  - `x > 0`
  - Call `917-555-5555` to verify you have a license to use this software

What is a Contract Check?

- An algorithm to identify when a contract has been violated
  - `x > 0`
  - Call `917-555-5555` to verify you have a license to use this software
- A part of the contract

What is a Contract-Checking Facility?

What is a Contract-Checking Facility?

- A tool to describe contract checks

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things
    - *documentation* — Informing readers what will and won't constitute correct behavior

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things
  - *documentation* — Informing readers what will and won't constitute correct behavior
  - *runtime checking* — Identifying at runtime when a program evaluation is incorrect

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things
  - *documentation* — Informing readers what will and won't constitute correct behavior
  - *runtime checking* — Identifying at runtime when a program evaluation is incorrect
  - *runtime mitigation* — Mitigating the downsides of an incorrect program

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things
  - *documentation* — Informing readers what will and won't constitute correct behavior
  - *runtime checking* — Identifying at runtime when a program evaluation is incorrect
  - *runtime mitigation* — Mitigating the downsides of an incorrect program
  - *static analysis* — Identifying at compile time that a program will be or might be incorrect

What is a Contract-Checking Facility?

- A tool to describe contract checks
- Any functionality that leverages those descriptions to do things
  - *documentation* — Informing readers what will and won't constitute correct behavior
  - *runtime checking* — Identifying at runtime when a program evaluation is incorrect
  - *runtime mitigation* — Mitigating the downsides of an incorrect program
  - *static analysis* — Identifying at compile time that a program will be or might be incorrect
  - *optimization* — Optimizing based on the presumption that a program is correct

What isn't a Contract-Checking facility?

What isn't a Contract-Checking facility?

- A tool to add to what a Contract says a program will do

What isn't a Contract-Checking facility?

- A tool to add to what a Contract says a program will do
- A tool to add to the correct behaviors of a program

What isn't a Contract-Checking facility?

- A tool to add to what a Contract says a program will do
- A tool to add to the correct behaviors of a program
- A new form of flow control

What isn't a Contract-Checking facility?

- A tool to add to what a Contract says a program will do
- A tool to add to the correct behaviors of a program
- A new form of flow control
- A tool to do aspect-oriented programming

Principles History

Principles History

- Many papers have attempted to identify and motivate the central principles of our design
  - P2834R1 - Semantic Stability Across Contract-Checking Build Modes
  - P2932R3 - A Principled Approach to Open Design Questions for Contracts
  - P2900R7 - Contracts for C++

Principle: Prime Directive

The use of a Contract-Checking facility should not change the correctness of a program.

Principle: Prime Directive

The use of a Contract-Checking facility should not change the correctness of a program.

- If it does, it is now part of the program and not checking the contract

Principle: Prime Directive

The use of a Contract-Checking facility should not change the correctness of a program.

- If it does, it is now part of the program and not checking the contract
- When possible we aim to prevent this at compile time

Principle: Prime Directive

The use of a Contract-Checking facility should not change the correctness of a program.

- If it does, it is now part of the program and not checking the contract
- When possible we aim to prevent this at compile time
- When possible we aim to make it harder to do this accidentally

Violating the prime directive...

Violating the prime directive...

- The program with checks evaluated tells you nothing about the program with checks unevaluated

Violating the prime directive...

- The program with checks evaluated tells you nothing about the program with checks unevaluated
- Heisenbugs — bugs appear and disappear when you try to observe them

Violating the prime directive...

- The program with checks evaluated tells you nothing about the program with checks unevaluated
- Heisenbugs — bugs appear and disappear when you try to observe them
- Cannot reason (as a reader or a static analyzer) about the program state locally without considering all previous contract checks — and thus 2^n program states

Following the prime directive...

Following the prime directive...

- Makes ignoring contract checks useful — don't pay to check what you are confident is true, program will remain correct

Following the prime directive...

- Makes ignoring contract checks useful — don't pay to check what you are confident is true, program will remain correct
- Allows static analysis of one program state instead of 2^N program states

Following the prime directive...

- Makes ignoring contract checks useful — don't pay to check what you are confident is true, program will remain correct
- Allows static analysis of one program state instead of 2^N program states
- Prevents Heisenbugs

Following the prime directive...

- Makes ignoring contract checks useful — don't pay to check what you are confident is true, program will remain correct
- Allows static analysis of one program state instead of 2^N program states
- Prevents Heisenbugs
-

Existing contract-checking facilities

Existing contract-checking facilities

- Comments

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked
  - No support for any behavior in the standard

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked
  - No support for any behavior in the standard
    - no runtime checking, minimal static analysis

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked
  - No support for any behavior in the standard
    - no runtime checking, minimal static analysis
  - No structure

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked
  - No support for any behavior in the standard
    - no runtime checking, minimal static analysis
  - No structure
  - Never violates the prime directive

Existing contract-checking facilities

- Comments
  - Documentation of a contract can tell you how it can be checked
  - No support for any behavior in the standard
    - no runtime checking, minimal static analysis
  - No structure
  - Never violates the prime directive

Existing contract-checking facilities

Existing contract-checking facilities

- `<cassert>`

Existing contract-checking facilities

- `<cassert>`
  - Almost complete freedom

Existing contract-checking facilities

- `<cassert>`
  - Almost complete freedom
  - No protection from violating the prime directive

SG21 MVP

SG21 MVP

- P2900 introduces *contract assertions*

SG21 MVP

- P2900 introduces *contract assertions*
  - Each `pre`, `post`, or `contract_assert` is a contract assertion

SG21 MVP

- P2900 introduces *contract assertions*
  - Each `pre`, `post`, or `contract_assert` is a contract assertion
  - Each contract assertion is expected to follow the prime directive

Principle: Prime Directive (Contract Assertions)

Neither the presence of a contract assertion nor the evaluation of a contract predicate should alter the correctness of a program's evaluation.

Principle: Prime Directive (Contract Assertions)

Neither the presence of a contract assertion nor the evaluation of a contract predicate should alter the correctness of a program's evaluation.

- The presences alone violating the prime directive would prevent users from *not* violating the prime directive

Principle: Prime Directive (Contract Assertions)

Neither the presence of a contract assertion nor the evaluation of a contract predicate should alter the correctness of a program's evaluation.

- The presences alone violating the prime directive would prevent users from *not* violating the prime directive
- We cannot prevent all predicates from violating, but we can discourage common cases where they would

Prevent violating the prime directive at compile time

Principle: Concepts do not see Contracts

The presence of a contract assertion shall not be observable through the use of concepts.

Prevent violating the prime directive at compile time

Principle: Concepts do not see Contracts

The presence of a contract assertion shall not be observable through the use of concepts.

- Guides our decisions on a number of design aspects

Prevent violating the prime directive at compile time

Principle: Concepts do not see Contracts

The presence of a contract assertion shall not be observable through the use of concepts.

- Guides our decisions on a number of design aspects
  - Compile-time evaluation behavior

Prevent violating the prime directive at compile time

Principle: Concepts do not see Contracts

The presence of a contract assertion shall not be observable through the use of concepts.

- Guides our decisions on a number of design aspects
  - Compile-time evaluation behavior
  - Implicit lambda captures

Prevent violating the prime directive at compile time

Principle: Concepts do not see Contracts

The presence of a contract assertion shall not be observable through the use of concepts.

- Guides our decisions on a number of design aspects
  - Compile-time evaluation behavior
  - Implicit lambda captures
  - Function contract assertions are not part of the immediate context (no SFINAE)

Prevent violating the prime directive at runtime

Prevent violating the prime directive at runtime

- A predicate whose evaluation would change the correctness of a program is a *destructive predicate*

Prevent violating the prime directive at runtime

- A predicate whose evaluation would change the correctness of a program is a *destructive predicate*
- We cannot determine systematically if a predicate is destructive

Is this destructive i?

```
void f() pre(true);
```

Is this destructive i?

```
void f() pre(true);
```

- It can be:

Is this destructive i?

```
void f() pre(true);
```

- It can be:
    - Contract: This program will not use C++ contract checking

Is this destructive i?

```
void f() pre(true);
```

- It can be:
  - Contract: This program will not use C++ contract checking
  - Contract: No identifiers will be used that are macros in C

Is this destructive i?

```
void f() pre(true);
```

- It can be:
  - Contract: This program will not use C++ contract checking
  - Contract: No identifiers will be used that are macros in C
- In most other cases, not destructive

Is this destructive i?

```
void f() pre(true);
```

- It can be:
  - Contract: This program will not use C++ contract checking
  - Contract: No identifiers will be used that are macros in C
- In most other cases, not destructive
  - Evaluates entirely at compile time

Is this destructive ii?

```
int *binary_search(int* begin, int* end, int v)
  pre(std::is_sorted(begin,end));
```

Is this destructive ii?

```
int *binary_search(int* begin, int* end, int v)
  pre(std::is_sorted(begin,end));
```

- Yes if evaluated, complexity is no longer logarithmic

Is this destructive iii?

```
bool test(int x)
{
   x = x & 1;
   return x > 0;
}
void f(int x)
  pre(test(x));
```

Is this destructive iii?

```
bool test(int x)
{
   x = x & 1;
   return x > 0;
}
void f(int x)
  pre(test(x));
```

- Probably not

Is this destructive iii?

```
bool test(int x)
{
   x = x & 1;
   return x > 0;
}
void f(int x)
  pre(test(x));
```

- Probably not
- Has core-language side effects

Is this destructive iii?

```
bool test(int x)
{
   x = x & 1;
   return x > 0;
}
void f(int x)
  pre(test(x));
```

- Probably not
- Has core-language side effects
    - Modifies a variable whose lifetime is within the evaluation

Is this destructive iii?

```
bool test(int x)
{
   x = x & 1;
   return x > 0;
}
void f(int x)
  pre(test(x));
```

- Probably not
- Has core-language side effects
  - Modifies a variable whose lifetime is within the evaluation
  - Called "Inside the cone of evaluation"

Is this destructive iv?

```
template<typename T, typename U>
void f(const std::map<T,int>& m, const U& k)
  pre(m.contains(k));
```

Is this destructive iv?

```
template<typename T, typename U>
void f(const std::map<T,int>& m, const U& k)
  pre(m.contains(k));
```

- Probably not

Is this destructive iv?

```
template<typename T, typename U>
void f(const std::map<T,int>& m, const U& k)
  pre(m.contains(k));
```

- Probably not
- Might have side effects outside cone of evaluation

Is this destructive iv?

```
template<typename T, typename U>
void f(const std::map<T,int>& m, const U& k)
  pre(m.contains(k));
```

- Probably not
- Might have side effects outside cone of evaluation
  - If `T` is `std::string` and `U` is `const char*`.

Is this destructive iv?

```
template<typename T, typename U>
void f(const std::map<T,int>& m, const U& k)
  pre(m.contains(k));
```

- Probably not
- Might have side effects outside cone of evaluation
  - If `T` is `std::string` and `U` is `const char*`.
  - State change (allocation and deallocation) is reverted after expression

Is this destructive v?

```
template<typename T>
void f(std::map<T,int>& m, const T& k)
  pre(m[k] == 0);
```

Is this destructive v?

```
template<typename T>
void f(std::map<T,int>& m, const T& k)
  pre(m[k] == 0);
```

- If `k` is not definitely in the map this modifies state

Is this destructive v?

```
template<typename T>
void f(std::map<T,int>& m, const T& k)
  pre(m[k] == 0);
```

- If `k` is not definitely in the map this modifies state
- If anything depends on the contents of the map, this is destructive

Is this destructive vi?

```
bool test() {
  printf("Test was called");
  return true;
}
void f()
  pre(test());
```

Is this destructive vi?

```
bool test() {
  printf("Test was called");
  return true;
}
void f()
  pre(test());
```

- Destructive if output to standard output is guaranteed by contract

Is this destructive vi?

```
bool test() {
  printf("Test was called");
  return true;
}
void f()
  pre(test());
```

- Destructive if output to standard output is guaranteed by contract
- Fine if standard output is used for logging and tracing

Is this destructive vii?

```
int testCalls = 0;
bool test() {
  ++testCalls;
  return true;
}
void f()
  pre(test());
```

Is this destructive vii?

```
int testCalls = 0;
bool test() {
  ++testCalls;
  return true;
}
void f()
  pre(test());
```

- If correctness depends on the values of `testCalls`, no

Is this destructive vii?

```
int testCalls = 0;
bool test() {
  ++testCalls;
  return true;
}
void f()
  pre(test());
```

- If correctness depends on the values of `testCalls`, no
- Otherwise, fine

Is this destructive viii?

```
struct List { int d_data; List * d_next; };
void f(List *lp)
{
  //#ifndef NDEBUG
  int index = 0;
  //#endif
  while (lp) {
    contract_assert(++index < 5);
    lp = lp->d_next;
  }
}
```

Is this destructive viii?

```
struct List { int d_data; List * d_next; };
void f(List *lp)
{
  //#ifndef NDEBUG
  int index = 0;
  //#endif
  while (lp) {
    contract_assert(++index < 5);
    lp = lp->d_next;
  }
}
```

- Always destructive — correctness of future evaluations changes each time `++index` is evaluated

Is this destructive viii?

```
struct List { int d_data; List * d_next; };
void f(List *lp)
{
  //#ifndef NDEBUG
  int index = 0;
  //#endif
  while (lp) {
    contract_assert(++index < 5);
    lp = lp->d_next;
  }
}
```

- Always destructive — correctness of future evaluations changes each time `++index` is evaluated
- No protection from using `index` and depending on it for correctness

Takeaways about Destructive Predicates

Takeaways about Destructive Predicates

- No predicate is non-destructive in all contexts

Takeaways about Destructive Predicates

- No predicate is non-destructive in all contexts
- Changes to local objects are likely to be destructive

Takeaways about Destructive Predicates

- No predicate is non-destructive in all contexts
- Changes to local objects are likely to be destructive
- Side effects within the cone of evaluation are likely to not be destructive

Takeaways about Destructive Predicates

- No predicate is non-destructive in all contexts
- Changes to local objects are likely to be destructive
- Side effects within the cone of evaluation are likely to not be destructive
- Side effects outside the cone of evaluation are not always destructive

Prevent violating the prime directive at runtime

Prevent violating the prime directive at runtime

- Discourage any dependance on evaluation

Prevent violating the prime directive at runtime

- Discourage any dependance on evaluation
- Minimize the chance of non-encapsulated modifications of existing objects

Prevent violating the prime directive at runtime

- Discourage any dependance on evaluation
- Minimize the chance of non-encapsulated modifications of existing objects
- Trust that `const` means state does not change

Elision

- A non-destructive predicate is always fine to elide

Elision

- A non-destructive predicate is always fine to elide
- Ignoring a contract assertion gives you the same program state as elision

Elision

- A non-destructive predicate is always fine to elide
- Ignoring a contract assertion gives you the same program state as elision
- A platform could provide elision of non-violated contract assertions already

Elision

- A non-destructive predicate is always fine to elide
- Ignoring a contract assertion gives you the same program state as elision
- A platform could provide elision of non-violated contract assertions already
  - Define the semantic of any check that can be proven as *ignore*

Repetition

- A non-destructive predicate is usually fine to evaluate again

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive
  - Those same contracts might make a single evaluation destructive

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive
  - Those same contracts might make a single evaluation destructive
- Repetition gives implementation freedom and user choice as to where code is generated for checks

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive
  - Those same contracts might make a single evaluation destructive
- Repetition gives implementation freedom and user choice as to where code is generated for checks
- Repetition allows detecting many destructive side effects

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive
  - Those same contracts might make a single evaluation destructive
- Repetition gives implementation freedom and user choice as to where code is generated for checks
- Repetition allows detecting many destructive side effects
- Experience reports

Repetition

- A non-destructive predicate is usually fine to evaluate again
  - Overly-specific contracts that limit the number of operations might make this destructive
  - Those same contracts might make a single evaluation destructive
- Repetition gives implementation freedom and user choice as to where code is generated for checks
- Repetition allows detecting many destructive side effects
- Experience reports
  - P3336R0 — only issues were pedantic testing

`const`-ification

- Prevents accidental modification of state in a contract assertion

`const`-ification

- Prevents accidental modification of state in a contract assertion
- Allows encapsulated changes that say they are **`const`**

`const`-ification

- Prevents accidental modification of state in a contract assertion
- Allows encapsulated changes that say they are `const`
- Experience reports

`const`-ification

- Prevents accidental modification of state in a contract assertion
- Allows encapsulated changes that say they are `const`
- Experience reports
  - P3268R0 — manual analysis of one large codebase

`const`-ification

- Prevents accidental modification of state in a contract assertion
- Allows encapsulated changes that say they are **`const`**
- Experience reports
  - P3268R0 — manual analysis of one large codebase
  - P3336R0 — uses current implementation in gcc

Throwing Violation Handlers

Throwing Violation Handlers

- Throwing is the primary mitigation strategy available without terminating

Throwing Violation Handlers

- Throwing is the primary mitigation strategy available without terminating
- Termination for many C++ users is never an option (P2698R0)

The *observe* semantic

The *observe* semantic

- Introducing a contract check into existing programs requires observing

The *observe* semantic

- Introducing a contract check into existing programs requires observing
  - Crashing users depending on Hyrum's law is often unacceptable

The *observe* semantic

- Introducing a contract check into existing programs requires observing
  - Crashing users depending on Hyrum's law is often unacceptable
  - Narrowing contracts is often needed for evolution

Compile Time Semantics

Compile Time Semantics

- *ignore* is needed as an option

Compile Time Semantics

- *ignore* is needed as an option
    - Algorithmically expensive checks can make a program un-compilable

Compile Time Semantics

- *ignore* is needed as an option
  - Algorithmically expensive checks can make a program un-compilable
  - **`constexpr`** evaluations tuned to the limit of operations will fail if contract assertions are checked

Compile Time Semantics

- *ignore* is needed as an option
  - Algorithmically expensive checks can make a program un-compilable
  - **`constexpr`** evaluations tuned to the limit of operations will fail if contract assertions are checked
- *observe* is needed as an option

Compile Time Semantics

- *ignore* is needed as an option
  - Algorithmically expensive checks can make a program un-compilable
  - **`constexpr`** evaluations tuned to the limit of operations will fail if contract assertions are checked
- *observe* is needed as an option
  - For any library used at compile time code must still compile with new releases

Compile Time Semantics

- *ignore* is needed as an option
  - Algorithmically expensive checks can make a program un-compilable
  - **`constexpr`** evaluations tuned to the limit of operations will fail if contract assertions are checked
- *observe* is needed as an option
  - For any library used at compile time code must still compile with new releases
  - Just like runtime libraries require observe so code still runs at runtime with new releases

Undefined Behavior in Contract Predicates

Undefined Behavior in Contract Predicates

- If semantics change we have a hard time talking about what a predicate will do

Undefined Behavior in Contract Predicates

- If semantics change we have a hard time talking about what a predicate will do
- Spreading UB to the context around a contract predicate can be bad

Undefined Behavior in Contract Predicates

- If semantics change we have a hard time talking about what a predicate will do
- Spreading UB to the context around a contract predicate can be bad
  - P1494R3 gives us a mechanism to prevent this

Undefined Behavior in Contract Predicates

- If semantics change we have a hard time talking about what a predicate will do
- Spreading UB to the context around a contract predicate can be bad
  - P1494R3 gives us a mechanism to prevent this
  - P3328R0 applies that mechanism to P2900

Too much implementation-defined behavior

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination
  - Selection of number of repetitions

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination
  - Selection of number of repetitions
  - Replaceability of the contract-violation handler

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination
  - Selection of number of repetitions
  - Replaceability of the contract-violation handler
  - When elision might happen

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination
  - Selection of number of repetitions
  - Replaceability of the contract-violation handler
  - When elision might happen
- Upcoming paper P3321R0

Too much implementation-defined behavior

- Only 5 points of implementation-defined behavior:
  - Selection of contract semantic
  - Methods of termination
  - Selection of number of repetitions
  - Replaceability of the contract-violation handler
  - When elision might happen
- Upcoming paper P3321R0
- All of these are for different

Principle: General Order One (Starfleet)

No starship may interfere with the normal development of any alien life or society.

Principle: General Order One (Contracts)

No contract check may interfere with the correctness of a program.

- The contract-checking facility is Starfleet

Principle: General Order One (Contracts)

No contract check may interfere with the correctness of a program.

- The contract-checking facility is Starfleet
- Each individual contract check is the starship

Principle: General Order One (Contracts)

No contract check may interfere with the correctness of a program.

- The contract-checking facility is Starfleet
- Each individual contract check is the starship
- The program is the non-warp-capable alien life or society