Контракты для C++

версия 0.4

Тимур Думлер

C++ Russia
Москва, 3 июня 2024 г.

Contracts for C++

Document #: P2900R7
Date: 2024-05-22
Project: Programming Language C++
Audience: EWG, LEWG
Reply-to: Joshua Berne <jberne4@bloomberg.net>
Timur Doumler <papers@timur.audio>
Andrzej Krzemieński <akrzemi1@gmail.com>
— with —
Gašper Ažman <gasper.azman@gmail.com>
Louis Dionne <ldionne@apple.com>
Tom Honermann <tom@honermann.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Ville Voutilainen <ville.voutilainen@gmail.com>

Abstract

In this paper, we propose a Contracts facility for C++ that has been carefully considered by SG21 with the highest bar possible for consensus. The proposal includes syntax for specifying three kinds of contract assertions: precondition assertions, postcondition assertions, and assertion statements. In addition, we specify four evaluation semantics for these assertions — one non-checking semantic, *ignore*, and three checking semantics, *observe*, *enforce*, and *quick_enforce* — as well as the ability to specify a user-defined handler for contract violations. The features proposed in this paper allow C++ users to leverage contract assertions in their ecosystems in numerous ways.

Contracts for C++

Document #: P2900R7
Date: 2024-05-22
Project: Programming Language C++
Audience: EWG, LEWG
Reply-to: Joshua Berne <jberne4@bloomberg.net>
Timur Doumler <papers@timur.audio>
Andrzej Krzemieński <akrzemi1@gmail.com>
— with —
Gašper Ažman <gasper.azman@gmail.com>
Louis Dionne <ldionne@apple.com>
Tom Honermann <tom@honermann.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Ville Voutilainen <ville.voutilainen@gmail.com>

https://wg21.link/p2900

Abstract

In this paper, we propose a Contracts facility for C++ that has been carefully considered by SG21 with the highest bar possible for consensus. The proposal includes syntax for specifying three kinds of contract assertions: precondition assertions, postcondition assertions, and assertion statements. In addition, we specify four evaluation semantics for these assertions — one non-checking semantic, *ignore*, and three checking semantics, *observe*, *enforce*, and *quick_enforce* — as well as the ability to specify a user-defined handler for contract violations. The features proposed in this paper allow C++ users to leverage contract assertions in their ecosystems in numerous ways.

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Контрактное программирование (Design by Contract)

- Метод проектирования программного обеспечения
- Предполагает определение формальных, точных и верифицируемых спецификаций интерфейсов для компонентов системы:
  - предусловия
  - постусловия
  - инварианты
- Данные спецификации называются «контрактами» в соответствии с концептуальной метафорой условий и ответственности в гражданско-правовых договорах

Контрактное программирование (Design by Contract)

- Метод проектирования программного обеспечения
- Предполагает определение формальных, точных и верифицируемых спецификаций интерфейсов для компонентов системы:
  - предусловия
  - постусловия
  - инварианты
- Данные спецификации называются «контрактами» в соответствии с концептуальной метафорой условий и ответственности в гражданско-правовых договорах

What are Contracts?

- A **contract** is a set of conditions that expresses expectations on a correct program.
- A **function contract** is a contract that is part of the specification of a function.
  - A **precondition** is a part of a function contract where the responsibility for satisfying it is on the caller of the function. Generally, these are requirements placed on the arguments passed to a function and/or the global state of the program upon entry into the function.
  - A **postcondition** is a part of a function contract where the responsibility for satisfying the condition is on the callee, i.e. the implementer of the function itself. These are generally conditions that will hold true regarding the return value of the function or the state of objects modified by the function when it completes execution normally.

Contract violations

- A contract violation is not an error.
- A contract violation is a **bug in the program.**

Contract violations

- A contract violation is not an error.
- A contract violation is a **bug in the program.**
- **Who is responsible** for the contract violation?
  - Precondition: the caller of the function
  - Postcondition: the callee, i.e. the implementation of the function

Contract violations

- A contract violation is not an error.
- A contract violation is a **bug in the program.**
- **Who is responsible** for the contract violation?
  - Precondition: the caller of the function
  - Postcondition: the callee, i.e. the implementation of the function
- **What happens** when there is a contract violation?
  - It depends...
  - ...in C++, it often results in **undefined behaviour**

Contract violations

```
// Returns a reference to the element at position `index`.
// The behaviour is undefined unless `index < size()`.
T& Vector::operator[] (size_t index);
```

Contract violations

```
  // Returns a reference to the element at position `index`.
  // The behaviour is undefined unless `index < size()`.
  T& Vector::operator[] (size_t index) {
    return _data[index];
  }

private:
  T* _data;
```

Contract violations

```
  // Returns a reference to the element at position `index`.
  // The behaviour is undefined unless `index < size()`.
  T& Vector::operator[] (size_t index) {
    return _data[index]; // language UB if `index < size`!
  }

private:
  T* _data;
```

3 Descriptions of function semantics contain the following elements (as appropriate):[143]

(3.1) — *Constraints*: the conditions for the function's participation in overload resolution ([over.match]).

[*Note 1*: Failure to meet such a condition results in the function's silent non-viability. — *end note*]

[*Example 1*: An implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]). — *end example*]

(3.2) — *Mandates*: the conditions that, if not met, render the program ill-formed.

[*Example 2*: An implementation can express such a condition via the *constant-expression* in a *static_assert-declaration* ([dcl.pre]). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]) and also define the function as deleted. — *end example*]

(3.3) — *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

(3.4) — *Effects*: the actions performed by the function.

(3.5) — *Synchronization*: the synchronization operations ([intro.multithread]) applicable to the function.

(3.6) — *Postconditions*: the conditions (sometimes termed observable results) established by the function.

(3.7) — *Result*: for a *typename-specifier*, a description of the named type; for an *expression*, a description of the type of the expression; the expression is an lvalue if the type is an lvalue reference type, an xvalue if the type is an rvalue reference type, and a prvalue otherwise.

(3.8) — *Returns*: a description of the value(s) returned by the function.

(3.9) — *Throws*: any exceptions thrown by the function, and the conditions that would cause the exception.

(3.10) — *Complexity*: the time and/or space complexity of the function.

(3.11) — *Remarks*: additional semantic constraints on the function.

(3.12) — *Error conditions*: the error conditions for error codes reported by the function.

Terminology

- A function with no preconditions has a **wide contract.**
- A function with preconditions has a **narrow contract.**

`std::vector::size`

```
std::vector::size  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end  // wide contract
std::vector::push_back
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end  // wide contract
std::vector::push_back  // wide contract
```

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end  // wide contract
std::vector::push_back  // wide contract
std::vector::swap
```

[65] The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall meet the *Cpp17Swappable* requirements and shall be exchanged by calling `swap` as described in [swappable.requirements]. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then `allocator_type` shall meet the *Cpp17Swappable* requirements and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in [swappable.requirements]. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `a.get_allocator() == b.get_allocator()`. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

```
std::vector::size  // wide contract
std::vector::empty  // wide contract
std::vector::operator[]  // narrow contract
std::vector::at  // wide contract
std::vector::front  // narrow contract
std::vector::back  // narrow contract
std::vector::reserve  // wide contract
std::vector::begin  // wide contract
std::vector::end  // wide contract
std::vector::push_back  // wide contract
std::vector::swap  // narrow or wide contract (depending on type)
```

How do we specify a contract?

- In the documentation: **plain language contract**

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments

```
// Returns a reference to the element at position `index`.
// The behaviour is undefined unless `index < size()`.
T& Vector::operator[] (size_t index);
```

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)

```
constexpr const_reference operator[](size_type pos) const;
```

1 *Preconditions*: `pos < size()`.

2 *Returns*: `data_[pos]`.

3 *Throws*: Nothing.

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon convention)

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon convention)
- In code: **contract assertions**

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon convention)
- In code: **contract assertions**
  - A language feature that provides support for contract assertions is a **Contracts facility**
  - Can be a core language feature (D, Eiffel, Ada...) or a library feature
  - P2900R7 proposes a Contracts facility for C++ as a core language feature

Contract assertions

```
T& operator[] (size_t pos) const
  pre (index < size());
```

Contract assertions

```
T& operator[] (size_t pos) const
  pre (index < size());
```

- A contract assertion typically expresses a particular **provision** of the plain-language contract rather than the entire contract
- A contract assertion specifies a C++ algorithm that allows to either:
  - Verify compliance with the provision, or
  - Identify violations of the provision.

Contract assertions

```
T& operator[] (size_t pos) const
  pre (index < size());
```

- A contract assertion typically expresses a particular **provision** of the plain-language contract rather than the entire contract
- A contract assertion specifies a C++ algorithm that allows to either:
  - Verify compliance with the provision, or
  - Identify violations of the provision.
- In P2900R7, this algorithm is a C++ expression contextually convertible to `bool` called a **contract predicate**.

Checking contracts with contact assertions

- Sometimes straightforward

```
T& operator[] (size_t index) const
  pre (index < size());
```

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees

```
void binary_search(Iter begin, Iter end)  // O(log N)
  pre (is_sorted(begin, end));             // O(N)
```

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an existing object")

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an existing object")
- Or outright impossible ("passed-in function `f` returns a value")

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an existing object")
- Or outright impossible ("passed-in function `f` returns a value")
- Contract assertions in general specify only **a subset of the plain-language contract** of the function rather than the entire contract

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

History of Contracts

- Term "Design by Contract" coined by Bertrand Meyer in the 80s in the context of designing *Eiffel*
- Languages with a Contracts facility as a language feature: *Eiffel, D, Ada, Dafny...*
- Languages with a Contracts facility as a library facility: *Java, C#, C++...*

C++ has a Contracts facility!

C++ has a Contracts facility!

```
#include <cassert>
T& operator[] (size_t index) {
  // Index needs to be within range!
  assert(index < size());
}
```

C++ has a Contracts facility!

```
#include <cassert>
T& operator[] (size_t index) {
  // Index needs to be within range!
  assert(index < size());
}
```

- Behaviour not customisable (token-ignore or `std::abort`)
- Cannot go on function declarations, only in function bodies
- Information about contract violation not programmatically accessible
- It's a macro (ODR violations, other macro-related problems)

Why do we need a Contracts facility in C++ <u>as a language feature</u>

Why do we need a Contracts facility in C++

- To prevent UB and increase the safety and correctness of our code

Why do we need a Contracts facility in C++ as a language feature

- Precondition and postcondition assertions on **declarations**
- Portably usable across different libraries and codebases
- A central place for the entire program to handle contract violations
- Fully customisable behaviour
- Information about the contract violation programmatically available
- Predicate expressions parsed even if not evaluated, no ODR issues
- Accessible for tooling (IDEs, static analysis tools, ...)

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)

```
double sqrt(double x)
precondition
{
  x > 0.0;
}
postcondition(r)
{
  approx_equal(r * r, x);
}
```

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)

- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)

```
double sqrt(double x)
{
  contract_assert(x > 0.0);
}
```

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)
- 2014-2019: C++20 Contracts – P0542
  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962

  (Thorsten Ottosen, Lawrence Crowl, et al)

- 2013-2014: BDE-like Macro Contracts – N4378

  (John Lakos et al)

- 2014-2019: C++20 Contracts – P0542

  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)

```
double sqrt(double x)
  [[expects: x > 0.0]]
  [[ensures r: approx_equal(r * r, x)]]
{ [[assert: i != x ]]; }
```

Contract assertions in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)
- 2014-2019: C++20 Contracts – P0542
  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)
- 2019-today: Contracts MVP – P2900
  (Joshua Berne, Timur Doumler, Andrzej Krzemieński, et al.)

ISO/IEC JTC 1 (IT) — (F)DIS Approval

SC 22 (Pgmg Langs) — CD & PDTS Approval

WG21 – C++ Committee (full plenary) — Internal Approval

Core Wording | Lib Wording — **3 - Wording & Consistency**

Core Evolution | Lib Evolution — **2 - Design & Target (IS/TS)**

Admin Group

Direction Group

ABI Review Group

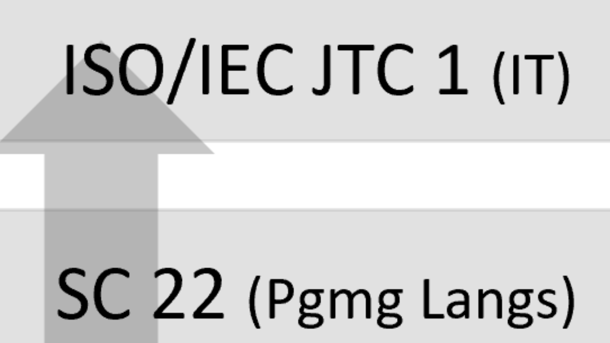
SG1 Concurrency	SG2 Modules	SG4 Networking	SG5 Tx. Memory	SG6 Numerics
SG7 Reflection	SG9 Ranges	SG10 Feature Test	SG12 U. Behavior	SG14 Game, Embedded, Low Latency
SG15 Tooling	SG16 Text	SG17 EWG Incubator	SG18 LEWG Incubator	SG19 Machine Learning
SG20 Education	SG21 Contracts	SG22 C/C++ Liaison	SG23 Safety & Security	

1 - Domain Specific Investigation & Incubation

SG3 Filesystem	SG8 Concepts	SG11 Databases	SG13 HMI, I/O

Completed, inactive

3-stage pipeline



(F)DIS Approval

CD & PDTS Approval

WG21 – C++ Committee (full plenary)

Internal Approval

Admin Group

Direction Group

ABI Review Group

Core Wording

Lib Wording

3 - Wording & Consistency

Core Evolution

Lib Evolution

2 - Design & Target (IS/TS)

1 - Domain Specific Investigation & Incubation

Completed, inactive

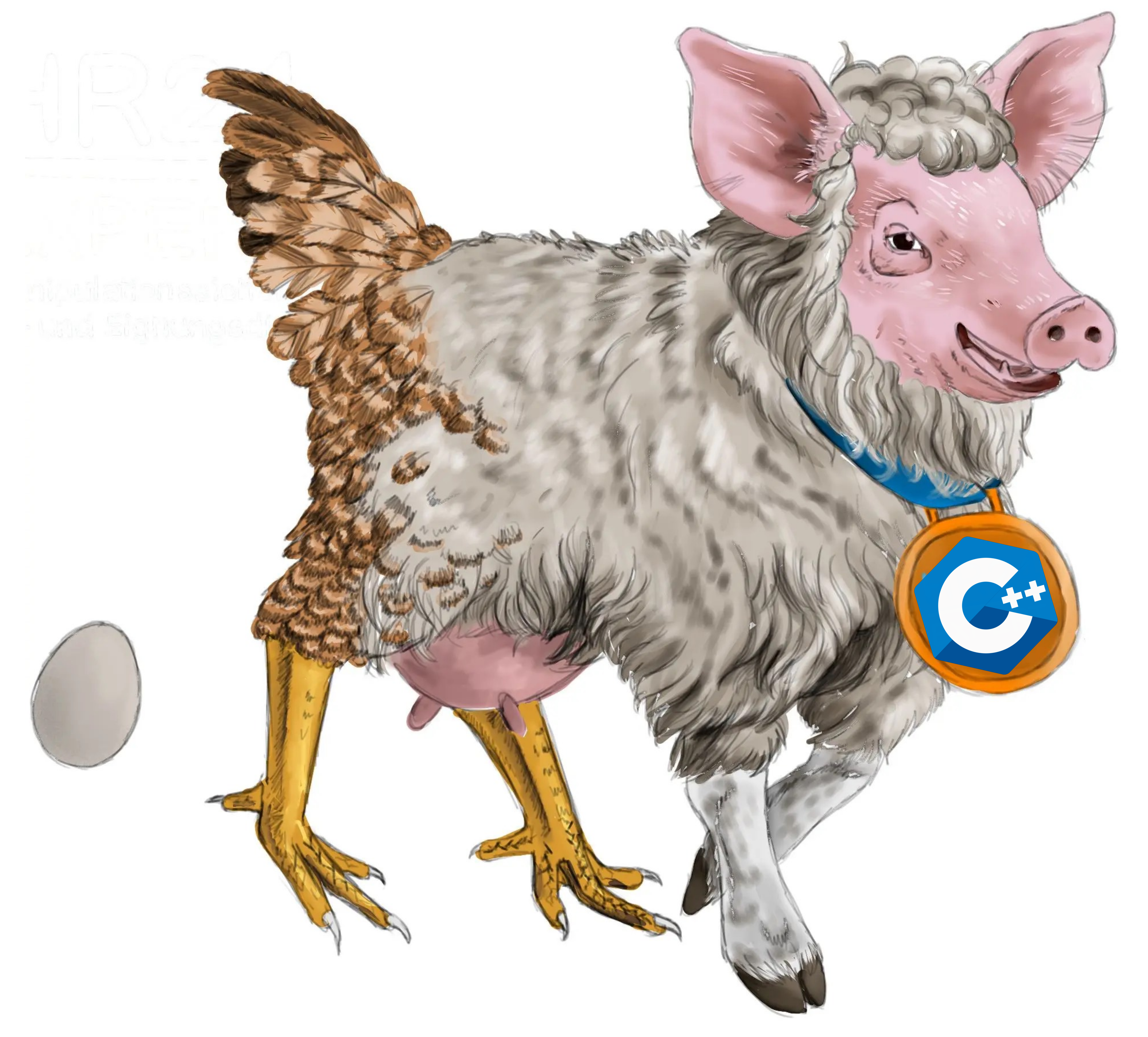
Document no: P1995R1
Date: 2020-03-02
Authors: Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, Herb Sutter
Reply-to: jberne4@bloomberg.net
Audience: SG21

Contracts — Use Cases

Introduction

SG21 has gathered a large number of use cases for contracts between teh WG21 Cologne and Belfast meetings. This paper presents those use cases, along with some initial results from polling done of SG21 members to identify some level of important to the community for each individual use case.

Each use case has been assigned an identifier that can be used to reference these use cases in other papers, which will hopefully be stable. We expect this content to evolve in a number of ways:



Copyright (c) Timur Doumler

The Contracts MVP

- Minimal viable product
  - Does not yet support all use cases!
    - However, explicitly designed for extensibility
  - Provides immediate value for a significant fraction of C++ users
    - But not everyone, and that's OK

The Contracts MVP

- Self-documenting code
- Run-time contract checks
- Static analysis
- Formal verification
- Optimisation

The Contracts MVP

- Self-documenting code ✅
- Run-time contract checks
- Static analysis
- Formal verification
- Optimisation

The Contracts MVP

- Self-documenting code ✅
- Run-time contract checks ✅
- Static analysis
- Formal verification
- Optimisation

The Contracts MVP

- Self-documenting code ✅
- Run-time contract checks ✅ →
- Static analysis
- Formal verification
- Optimisation

- replacement for <cassert> and custom assertion macros
- customisable global contract-violation handler
- can go on declarations
- information about violation available programmatically
- no macros :)

The Contracts MVP

- Self-documenting code ✅
- Run-time contract checks ✅
- Static analysis 🤷‍♂️
- Formal verification 🤷‍♂️
- Optimisation 🤷‍♂️

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

```
int f(int x)
  pre (x != 1);        // precondition assertion
```

```
int f(int x)
  pre (x != 1)        // precondition assertion
  post (r: r != 2);   // postcondition assertion; `r` names return value

// return value name is optional
// `pre` and `post` are contextual keywords
// pre(...) and post(...) appear at the end of the declaration
// - can also be a definition
// - functions and function templates (including member functions)
// - lambda expressions
```

```
int f(int x)
  pre (x != 1)       // precondition assertion
  post (r: r != 2)   // postcondition assertion; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

// `contract_assert` is full keyword
// we did not use `assert` because of clash with assert macro
```

contract assertion
function-contract
assertion
precondition
assertion
pre(expr)
postcondition
assertion
post(expr)
assertion statement
contract_assert(expr)

Where you can place a contract assertion

- `pre`, `post`:
  - on declarations of functions and function templates
    - obligatory on first declarations*, optional on redeclarations
    - if deduced (auto) return type, first declaration has to be a definition
  - on lambda expressions
- `contract_assert`:
  - Anywhere you can place a statement

---

*first declaration = declaration from which no other declaration is reachable

Where you *cannot* place a contract assertion

- `pre`, `post`:
  - not on `=deleted` functions
  - not on functions `=defaulted` on their first declaration
  - not on virtual functions (coming soon → P3097R0, P3165R0, D3169R0)
  - not on function pointers
    (`pre`, `post` are still evaluated when calling through a function pointer!)
  - not on coroutines (`contract_assert` is allowed inside a coroutine)

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Copyright (c) Timur Doumler | @timur_audio | https://timur.audio

Name lookup and access control

```
struct X {
  void f(int j)
    pre (j != i);  // name lookup & access as-if first statement in body
private:
  int i = 0;
};
```

Referring to non-reference parameters in `post`

```
int clamp(int v, int min, int max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in post

```
int clamp(const int v, const int min, const int max) // on all declarations
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in `post`

```
int clamp(int v, int min, int max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val)
{
  min = max = value = 0;
  return 0;
}
```

Referring to non-reference parameters in post

```
int clamp(const int v, const int min, const int max) // on all declarations
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to result value in post

```
int f()
  post(r: r > 0);

// r is an lvalue of type `const T` referring to result object
```

Referring to result value in post

```
int f()
  post(r: r > 0);

// r is an lvalue of type `const T` referring to result object


int f2()
  post(r: ++r);  // error

int f3()
  post(r: ++const_cast<int&>(r));  // OK (but evil)
```

Local variables are implicitly const

```
int global = 0;

void f(int x, int& ref, char* p)
  pre((x = 0) == 0)               // #1: error: assignment to const lvalue
  pre((ref = 5))                  // #2: error: assignment to const lvalue
  pre((*p = 5))                   // #3: OK (but evil)
  pre((global = 2));              // #4: OK (but evil)
```

Local variables are implicitly const

```
int global = 0;

void f(int x, int& ref, char* p)
  pre((x = 0) == 0)            // #1: error: assignment to const lvalue
  pre((ref = 5))               // #2: error: assignment to const lvalue
  pre((*p = 5))                // #3: OK (but evil)
  pre((global = 2))            // #4: OK (but evil)
{
  int var = 42;
  contract_assert((var = 42)); // #5: error: assignment to const lvalue
  contract_assert((const_cast<int&>(var) = 42));  // #6: OK (but evil)
  static int svar = 1;
  contract_assert((svar = 1)); // #7: OK (but evil)
}
```

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Point of evaluation

- **Precondition assertions (pre):**
  after the initialisation of function parameters,
  before the evaluation of the function body
- **Postcondition assertions (post):**
  after the result object value has been initialised and local automatic variables have been destroyed, but prior to the destruction of function parameters
- **Assertion statements (contract_assert):**
  when the statement is executed

```
int f(int x)
  pre (x != 1)        // precondition assertion
  post (r: r != 2)    // postcondition assertion; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

void g() {
  f(0); // no contract violation
  f(1); // violates precondition assertion of f
  f(2); // violates postcondition assertion of f
  f(3); // violates assertion statement within f
  f(4); // no contract violation
}
```

A contract assertion can be evaluated with one of the following four evaluation semantics:

- ***ignore***: do not check the predicate (but still parse and ODR-use it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

A contract assertion can be evaluated with one of the following four evaluation semantics:

- ***ignore***: do not check the predicate (but still parse and ODR-use it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then terminate the program*
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program*

*in some implementation-defined way

A contract assertion can be evaluated with one of the following four evaluation semantics:

"non-checking semantic"

- ***ignore***: do not check the predicate (but still parse and ODR-use it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

A contract assertion can be evaluated with one of the following four evaluation semantics:

- ***ignore***: do not check the predicate (but still parse and ODR-use it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

"checking semantics"

A contract assertion can be evaluated with one of the following four evaluation semantics:

- ***ignore***: do not check the predicate (but still parse and ODR-use it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

"enforcing semantics"

Evaluation semantics

- The mechanism of choosing a contract semantic (***ignore***, ***observe, enforce, quick_enforce)*** is **implementation-defined**
- Contract semantic can be different for each contract assertion, or even for each evaluation of the same contract assertion
- Contract semantic can be chosen at compile time, link time, or runtime

Evaluation semantics

- The mechanism of choosing a contract semantic (***ignore***, ***observe, enforce, quick_enforce)*** is **implementation-defined**
- Contract semantic can be different for each contract assertion, or even for each evaluation of the same contract assertion
- Contract semantic can be chosen at compile time, link time, or runtime
- Recommended practice:
  - provide a mode where all contract assertions have the *ignore* semantic;
  - provide a mode where all contract assertions have the *enforce* semantic;
  - When nothing else has been specified by the user, *enforce* is the default.

Checking the contract predicate

- The predicate evaluates to true → no contract violation, execution continues
- The predicate evaluates to false → contract violation
- Evaluation of the predicate does not finish, but control remains in the purview of the contract-checking process → "contract violation"
  - Evaluation exits via an exception
  - Evaluation occurs during constant evaluation, and predicate is not a core constant expression
- Evaluation of the predicate does not finish, control never returns to the purview of the contract-checking process → "you get what you get"
  - longjmp, terminate, infinite loop, suspend current thread forever, etc.

Calling the contract-violation handler

- When a contract violation occurs with the *observe* or *enforce* evaluation semantic:
  - An object of type `std::contracts::contract_violation` will be produced through implementation-defined means,
  - the **contract-violation handler** will be called,
  - the `std::contracts::contract_violation` object will be passed to the contract-violation handler (by `const&`)
  - If the contract violation occurred because evaluation of the predicate exited via an exception, the contract-violation handler acts as a handler for that exception (i..e the exception can be acccessed from within the contract-violation handler via `std::current_exception()`).

The contract-violation handler

- Function named `::handle_contract_violation`
  - Attached to the global module
  - Takes a single argument `const std::contracts::contract_violation&`
  - Returns `void`
  - May be `noexcept(true)` or `noexcept(false)`
- No declaration of `::handle_contract_violation` provided in any standard library header
- Implementation provides a default definition: **default contract-violation handler**
  - semantics implementation-defined, recommendation: print info about contract violation
- Implementation-defined whether it is **replaceable** (at link time, like `operator new/delete`)
  - You can provide your own **user-defined contract-violation handler** by implementing a function with a matching name and signature, and linking it into your program

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  LOG(std::format("Contract violated at: {}\n", violation.location()));
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::breakpoint();
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  while (!std::is_debugger_present())
    /* spin */;

  std::breakpoint();
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::cout << std::stacktrace::current(1);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::cout << std::stacktrace::current(1);
  std::contracts::invoke_default_contract_violation_handler(violation);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
}
```

Recursive contract violations

Recursive contract violations

- Two options:
  - Prevent recursion: disable contract checking while inside contract-violation handler

Recursive contract violations

- Two options:
  - Prevent recursion: disable contract checking while inside contract-violation handler
  - Do nothing: "you get what you get" → chosen for Contracts MVP

Contract assertions during constant evaluation ("at compile time")

```
constexpr int f(int i) // only odd values for i are allowed
  pre ( i % 2 == 0) {
  return ++i;
}
```

Contract assertions during constant evaluation ("at compile time")

```
constexpr int f(int i) // only odd values for i are allowed
  pre ( i % 2 == 0) {
  return ++i;
}

int main() {
  std::array<int, f(5)> arr; // compile-time contract violation!
  std::cout << f(5);          // run-time contract violation!
}
```

Contract assertions during constant evaluation ("at compile time")

- contract violation:
  - predicate evaluates to `false`
  - predicate is not a core constant expression
- behaviour of evaluation semantics (mechanism of choice is implementation-defined, can be different from runtime evaluation semantics):
  - ***ignore***: do not constant-evaluate the predicate
  - ***observe***: constant-evaluate the predicate
    - contract-violation → issue a diagnostic ("warning")
  - ***enforce / quick_enforce***: constant-evaluate the predicate
    - contract-violation → program is ill-formed ("compile error")

Not covered in this talk - read P2900:

- Trial constant evaluation:

```
const int i = maybe_constexpr_f(5);
```

Not covered in this talk - read P2900:

- Trial constant evaluation:

```
const int i = maybe_constexpr_f(5);
```

- Interaction with templates and friend declarations
- Interaction with implicit lambda captures
- `pre`/`post` on constructors and destructors
- other technical details

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Standard Library API

- Only needed to implement a user-defined violation handler, not needed to add contract assertions to your code!
- Everything is in header `<contracts>`
- Everything is in namespace `std::contracts`
- One class `contract_violation` (passed into the contract-violation handler)
- Three enums to express the return values of some of its member functions
- One free function
- That's it!

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class detection_mode : int {
    predicate_false = 1,
    evaluation_exception = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class evaluation_semantic : int {
    enforce = 1,
    observe = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class assertion_kind : int {
    pre = 1,
    post = 2,
    assert = 3,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

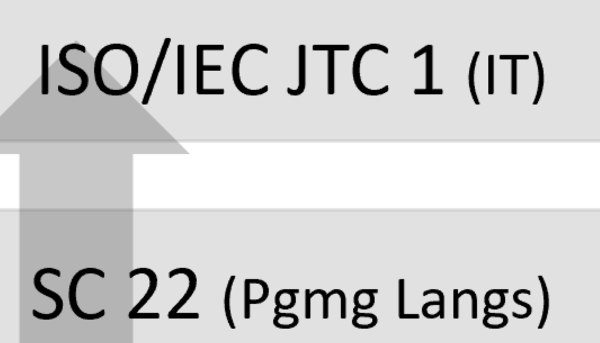
```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Impact on existing library facilities

Unless specified otherwise, an implementation is **allowed but not required** to check a subset of the preconditions and postconditions specified in the C++ standard library using contract assertions.

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions



ISO/IEC JTC 1 (IT) — (F)DIS Approval

SC 22 (Pgmg Langs) — CD & PDTS Approval

WG21 – C++ Committee (full plenary) — Internal Approval

3 - Wording & Consistency: Core Wording, Lib Wording

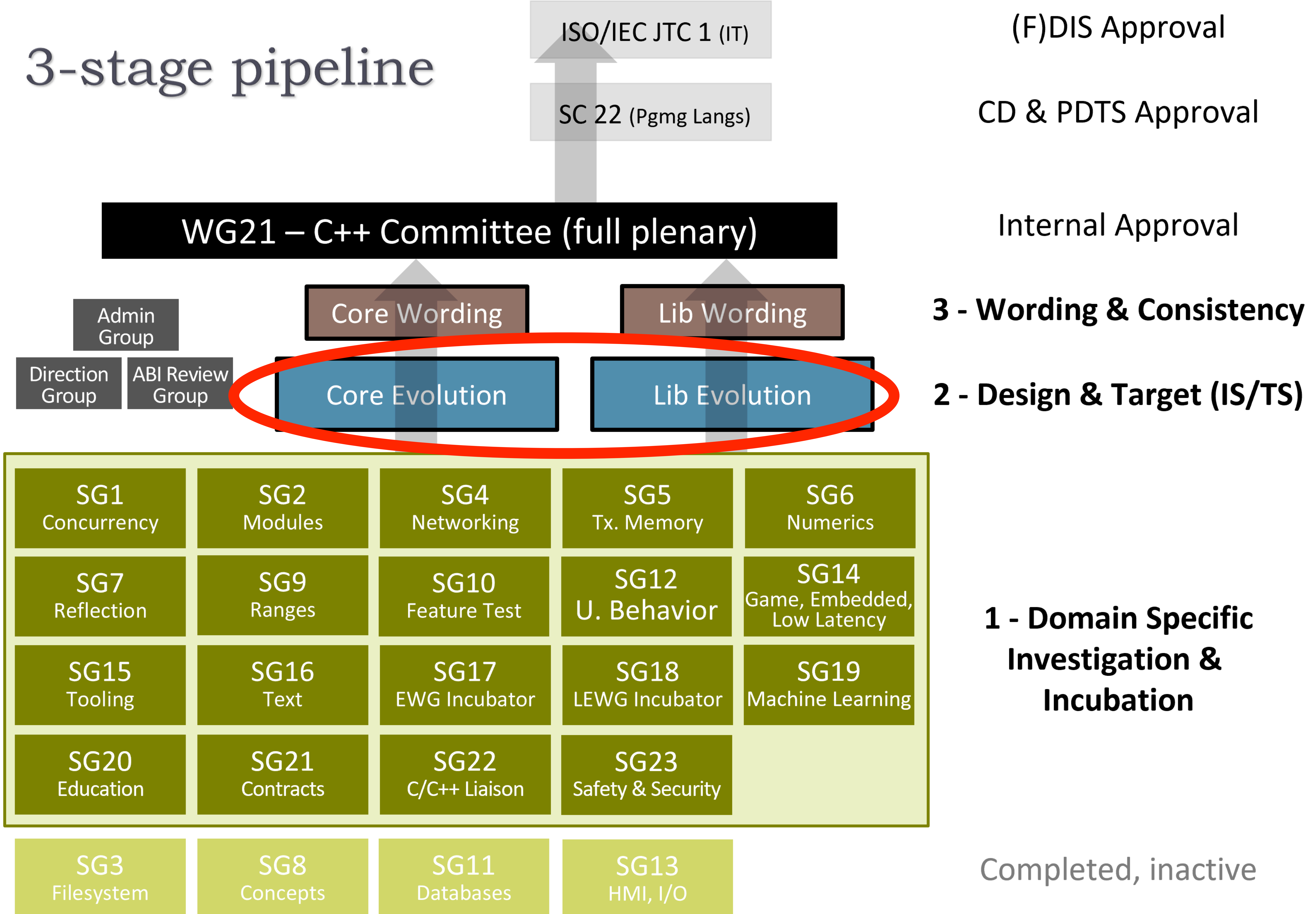
2 - Design & Target (IS/TS): Core Evolution, Lib Evolution

Admin Group, Direction Group, ABI Review Group

1 - Domain Specific Investigation & Incubation

SG1 Concurrency	SG2 Modules	SG4 Networking	SG5 Tx. Memory	SG6 Numerics
SG7 Reflection	SG9 Ranges	SG10 Feature Test	SG12 U. Behavior	SG14 Game, Embedded, Low Latency
SG15 Tooling	SG16 Text	SG17 EWG Incubator	SG18 LEWG Incubator	SG19 Machine Learning
SG20 Education	SG21 Contracts	SG22 C/C++ Liaison	SG23 Safety & Security	

Completed, inactive: SG3 Filesystem, SG8 Concepts, SG11 Databases, SG13 HMI, I/O


3-stage pipeline
ISO/IEC JTC 1 (IT)
SC 22 (Pgmg Langs)
(F)DIS Approval
CD & PDTS Approval
WG21 – C++ Committee (full plenary)
Internal Approval
Admin Group
Direction Group
ABI Review Group
Core Wording
Lib Wording
3 - Wording & Consistency
Core Evolution
Lib Evolution
2 - Design & Target (IS/TS)
SG1 Concurrency
SG2 Modules
SG4 Networking
SG5 Tx. Memory
SG6 Numerics
SG7 Reflection
SG9 Ranges
SG10 Feature Test
SG12 U. Behavior
SG14 Game, Embedded, Low Latency
SG15 Tooling
SG16 Text
SG17 EWG Incubator
SG18 LEWG Incubator
SG19 Machine Learning
SG20 Education
SG21 Contracts
SG22 C/C++ Liaison
SG23 Safety & Security
1 - Domain Specific Investigation & Incubation
SG3 Filesystem
SG8 Concepts
SG11 Databases
SG13 HMI, I/O
Completed, inactive

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers

Undefined behaviour, case 1

```
int f(int* p)  {
  std::cout << *p;          // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour, case 1

```
int f(int* p)
  pre ( p != nullptr ) {  // precondition assertion
  std::cout << *p;
}

int main() {
  f(nullptr);
}
```

Undefined behaviour, case 1

```
int f(int* p)
  pre ( p != nullptr ) {  // ignore: precondition not checked
  std::cout << *p;        // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour, case 1

```
int f(int* p)
  pre ( p != nullptr ) {  // enforce/quick_enforce: terminate here
  std::cout << *p;        // cannot get here!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour, case 1

```
int f(int* p)
  pre ( p != nullptr ) {  // observe: compiler can elide check
  std::cout << *p;        // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour, case 2

```
int f(int a) {
  return a + 100;
}

int g(int a)
  pre (f(a) > a);
```

Undefined behaviour, case 2

```
int f(int a) {
  return a + 100;  // compiler can assume this never overflows
}

int g(int a)
  pre (f(a) > a);
```

Undefined behaviour, case 2

```
int f(int a) {
  return a + 100;  // compiler can assume this never overflows
}

int g(int a)
  pre (f(a) > a);  // compiler can replace this with `pre (true)`
```

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers

Predicate side effects

- Contract pedicates with side effects are useful
  (use cases: alloc, lock/unlock mutex...)
- We cannot make them ill-formed
- We cannot make them undefined behaviour
- So we allow them
  - Side effects can be elided if the compiler can prove that the predicate would evaluate to true or false (~ copy elision)
  - Side effects can occur multiple times

Predicate side effects

```
int i = 0;
void f()
  pre ((++i, true));

void g() {
  f();  // `i` may be 0, 1, 17, etc.
}
```

Consequences of allowing elision/duplication

```
#ifndef NDEBUG
  unsigned nIter = 0;
#endif

while (keepIterating()) {
  assert(++nIter < 6); // it is a bug if we end up iterating more than 6 times
  // ...
}
```

Why allow multiple evaluation?

- Multiple reasons (see P3228, P3270)
- Main reason (see P3264):
  - `pre` and `post` checks can be performed caller-side or callee-side
  - `pre` and `post` checks can be enabled or disabled in dynamic library
  - `pre` and `post` checks can be enabled or disabled in application
  - application doesn't know if checks are enabled in library
    (`pre` and `post` are not part of ABI)
  - can lead to checks being performed twice

Why allow elision?

- Multiple reasons (see P3228, P3270)
- Main reason:
  - optimise away checks while guaranteeing program correctness

Why allow elision?

```
int f(int i)
  pre (i > 0);    // opaque function

int g(int i)
  pre (i > 0) {   // same precondition as f(i)
  return f(i) - 1;
}

int main() {
  int i;
  std::cin >> i;
  return g(i);
}
```

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers

Pros and cons of allowing contract-violation handlers to throw an exception

- Pro:
  - Provides a way to handle a contract violation such that we *neither* continue executing buggy code *nor* terminate the app
- Con:
  - any assertion can throw from anywhere, including in code that is not exception-safe (and cannot be made exception-safe)
  - there is no way to tell at compile time whether that might happen (the contract-violation handler is replaceable at link time)
  - cannot put `noexcept` on functions with narrow contract ("Lakos Rule")

The Lakos Rule is foundational for Contracts

```
int f(int i) noexcept
  pre(i > 0);  // `pre` and `post` cannot throw through noexcept!
               // instead, you get std::terminate
```

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers

Open design questions

- Should contract assertions have special provisions for mitigating UB?
- What guarantees should we give for side effects of contract predicates?
- Should the contract-violation handler be allowed to throw?
- Support for `pre` and `post` on virtual functions and function pointers
  - Work in progress (P3097R0, P3165R0, P3169R0)

Overview

- What are Contracts?
- History and motivation
- Syntax
- Semantic rules and restrictions
- Evaluation and contract-violation handling
- Library API
- Open design questions
- Future extensions

Copyright (c) Timur Doumler | @timur_audio | https://timur.audio

Future extensions

- Ability to refer to "old" values (at the time of call) inside a postcondition predicate:
  ```
  void push_back(T& item) post [oldSize = size()] (size() == oldSize + 1);
  ```
- Optimise based on assumption that predicate evaluates to true; otherwise, the behaviour is undefined (*assume* semantic)
- Contract levels ("audit", etc), explicit contract semantics, or other labels or meta-assertions that control the meaning of a contract assertion
- Precondition and postcondition assertions on coroutines
- Expressing postconditions expected to hold when a function exits via an exception
- Contract assertions that cannot be expressed by boolean predicates (procedural interfaces)
- Predicates that cannot be evaluated at runtime
- Invariants (class invariants, loop invariants...)

Спасибо за внимание!