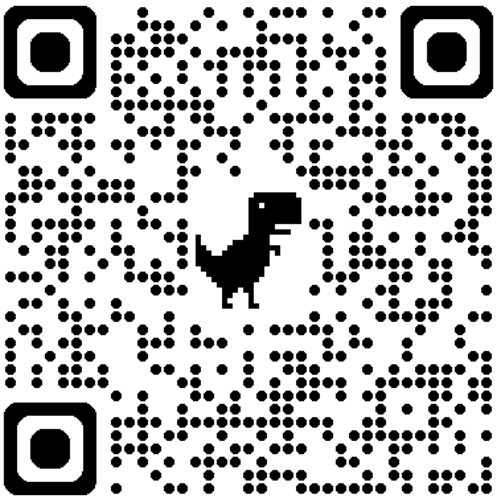
The Lens of Localized Contracts in Examples



Daniel Mall
dmall97@gmail.com
https://gist.github.com/cpp-source/f5255583f5cd660e758072547c5dca58

New Contract Syntax Idea #1

The first idea is local contract or if-then contract or inline contract.

```
constexpr double NaN = std::numeric_limits<double>::quiet_NaN();

double sqrt_vN(double x)
	handle_pre(x >= 0), fail{ assert(false); return NaN; }
{
	return std::sqrt(x);
}

double sqrt_vT(double x)
	handle_pre(x >= 0), fail{ throw std::invalid_argument("x is negative"); }
{
	return std::sqrt(x);
}
```

New Contract Syntax Idea #2

The second idea is named contract. Each contract is given a unique name.

```
std::string extract(const char* pB, const char* pE)
	pre(pB != nullptr) cpre1
	pre(pE != nullptr) cpre2
	pre(pB <= pE) cpre3
{
	if (!cpre1) { assert(false); return std:string(); }
	if (!cpre2) { assert(false); return std:string(); }
	if (!cpre3) { assert(false); return std:string(); }
	const int length = pE - pB;
	const std::string ns = std::string(pB, length);
	return ns;
}
```

Aim of Ideas is Local Control, Return Values, and Input Validation

These contract ideas allow for return values and input validation while still documenting the preconditions and postconditions. These contract ideas allow for optional use of the global contract violation handler and optional use of the compiler project setting for contract evaluation semantic.

Third Alternative – Contract Labels

```
// p3400r1 Controlling Contract-Assertion Properties
// choose a contract label that does not enforce
// the precondition is duplicated in pre and body

double sqrt_lai(const double x)
       pre<always_ignore>(x >= 0)
{
       if (x<0) { return NaN; }
       return std::sqrt(x);
}

double sqrt_lao(const double x)
       pre<always_observe>(x >= 0)
{
       if (x<0) { throw std::invalid_argument("x is negative"); }
       return std::sqrt(x);
}

double sqrt_lsym(const double x)
       pre<symbolic>(x >= 0)
{
       if (x < 0) { assert(false); }
       return std::sqrt(x);
}
```

Localized Contracts

A localized contract is always evaluated at runtime. Each localized contract defines it's own fail action. The fail action may respond to a contract violation by: throwing an exception, returning from the contracted function with a return value, terminating the program, adjusting an input value into an acceptable input range, or any other response that the software developer sees fit including logging. A localized contract leaves the decisions at the local level: the decision to evaluate the contract and the decision how to respond to a contract violation.

Named Contracts

A named contract is always assigned a unique name like assigning a variable name. The contract name is utilized to separate the definition of a precondition, the evaluation of a precondition, and the handling of a contract violation. The contract name must appear in the function body to initiate the evaluation of the contract. A contract name can invoke the "requires" action for a precondition or "ensures" action for a postcondition. Both "requires" and "ensures" action will use the default contract scheme (follows p2900r14 Contracts for C++). The default contract scheme is implementation defined behavior, a compiler setting for contract evaluation semantic, and a global contract violation handler. A named contract can localize the contract violation response.

Benefits of Contracts

- Documentation: Express the contracts of functions and other software components in code, as opposed to comments or separate documentation, thus making them consumable by human readers, IDEs, and other tooling and facilitating correct use of interfaces.
- Runtime checking: Evaluate contract predicates during program execution to portably and scalably identify defects.
- Static analysis: Provide contract assertions as additional input for static analysis tools to identify defects from the source code, e.g., through symbolic evaluation.

Benefits of Localized Contracts

- Return Values: A localized contract works for a function which needs to return an error code value for a contract violation.
- Local Control: A localized contract allows a software developer to take local control or definitive control of a contract violation response.
- Default is Optional: A localized contract bypasses the default contract scheme. A localized contract does not use the global contract violation handler.
- Implementation Defined is Optional: A localized contract does not use implementation defined behavior or compiler settings or project settings.
- Better than Labels: A localized contract contract could be an additional contract option like contract labels (p3400r1 Controlling Contract-Assertion Properties). The p2900r14 Contracts for C++ is advertised as a minimal viable product (MVP).
- Robustness: A localized contract allows robustness to be a priority over non-redundancy. A robust program wants to handle unexpected inputs, errors, and unusual conditions without crashing or producing incorrect results. A robust program attempts to gracefully recover from errors, maintain functionality, and provide a consistent user experience.

First Example - Existing Source Code

```
// precondition fails then return an error code
double sqrt_v1(double x)
{
	if (x < 0) { return NaN; }
	return std::sqrt(x);
}

// precondition fails then throw an exception
double sqrt_v2(double x)
{
	if (x < 0) { throw std::invalid_argument("x is negative"); }
	return std::sqrt(x);
}

// precondition is evaluated only in debug mode
double sqrt_v3(double x)
{
	if (x < 0) { assert(false); }
	return std::sqrt(x);
}
```

First Example - Now Add Contracts

```
// p2900r14 Contracts for C++
double sqrt_v4(double x)
	pre(x >= 0)
{
	return std::sqrt(x);
}
```

Is the function sqrt_v4 the same as sqrt_v1 or sqrt_v2 or sqrt_v3?

First Example - Localized Contract Idea

```
// precondition fails then return an error code
double sqrt_v5(double x)
	handle_pre(x >= 0), fail{ return NaN; }
{
	return std::sqrt(x);
}

// precondition fails then throw an exception
double sqrt_v6(double x)
	handle_pre(x >= 0), fail{ throw std::invalid_argument("x is negative"); }
{
	return std::sqrt(x);
}

// precondition is evaluated only in debug mode
double sqrt_v7(double x)
	handle_pre(x >= 0), fail{ assert(false); }
{
	return std::sqrt(x);
}
```

Localized Contracts on a Function

```
constexpr int r_good = 0;
constexpr int r_error = 1;
constexpr int r_ultimate_answer = 42;

int f1(const int x, int& result)
       // document contract only
       handle_pre(x != 1), fail{}
       // debug halt contract only
       handle_pre(x != 2), fail{ assert(false); }
       // exception throwing contract
       handle_pre(x != 3), fail{ throw std::exception(); }
       // error code contract return
       handle_pre(x != 4), fail{ return r_error; }
       // observe and continue contract
       handle_pre(x != 5), fail{ log_houston(); }
       // log and terminate program contract
       handle_pre(x != 6), fail{ log_houston(); std::terminate(); }
       // post conditions on an output parameter
       handle_post(result == x), fail{ assert(false); return r_error; }
       handle_post(result != r_ultimate_answer), fail{ assert(false); return r_error; }
       // post conditions on return value
       handle_post(r : r == r_good || r == r_error), fail{ assert(false); }
{
       handle_contract(x != 7), fail{ return r_error; };
       result = x;
       return r_good;
}

constexpr int minimum_psi = 40;
constexpr int maximum_psi = 80;

double set_psi(int psi)
       // corrective action on input value
       handle_pre(psi >= minimum_psi), fail{ log_houston(); psi = minimum_psi; }
       handle_pre(psi <= maximum_psi), fail{ log_houston(); psi = maximum_psi; }
{
       return (psi * 6.89476);
}

std::string extract(const char* pB, const char* pE)
       handle_pre(pB != nullptr), fails{ log_houston(); return std:string(); }
       handle_pre(pE != nullptr), fails{ log_houston(); return std:string(); }
       handle_pre(pB <= pE), fails{ log_houston(); return std:string(); }
{
       const int length = pE - pB;
       const std::string ns = std::string(pB, length);
       return ns;
}
```

Localized Contracts on a Class Member Function

```
// starting point
class account
{
public:
virtual bool open(const double x)
	handle_pre(m_minimum >= 100.0), fail{ err("minimum is too low"); return false; }
	handle_pre(m_balance == 0.0), fail{ err("account is already open"); return false; }
	handle_pre(x >= m_minimum), fail{ err("deposit is too small"); return false; }
{
	m_balance = x;
	return true;
}

virtual bool withdraw(const double x)
	handle_pre(x > 0.0), fail{ err("withdraw is too small"); return false; }
	handle_pre(x <= available()), fail{ err("withdraw is too large"); return false; }
{
	m_balance = (m_balance - x);
	return true;
}

double available() const { return (m_balance - m_minimum); }
std::vector<std::string> get_errors() const { return m_errors; }

private:
	void err(std::string error) { m_errors.push_back(error); }

	double m_balance = 0.0;
	double m_minimum = 1000.0;
	std::vector<std::string> m_errors;
};

inline bool is_friday() const
{
	const auto now = std::chrono::system_clock::now();
	const auto now_local = std::chrono::current_zone()->to_local(now);
	const std::chrono::weekday wd{ floor<std::chrono::days>(now_local) };
	const bool bFriday = (wd == std::chrono::Friday);
	return bFriday;
}

// an account restricted to withdraws on friday
class account_friday_withdraw : public account
{
public:
virtual bool withdraw(const double x) override
	handle_pre(x <= available()), fail{ err("funds are insufficient"); return false; }
	handle_pre(is_friday()), fail{ err("withdraw day is not friday"); return false; }
{
	return account::withdraw(x);
}
};
```

First Example - Named Contracts

```
// precondition default follows p2900r14 Contracts for C++
double sqrt_cn1(double x)
       pre(x >= 0) cpre1
{
       cpre1.require();
       return std::sqrt(x);
}

// precondition fails then return an error code
double sqrt_cn2(double x)
       pre(x >= 0) cpre1
{
       if (!cpre1) { return NaN; }
       return std::sqrt(x);
}

// precondition fails then throw an exception
double sqrt_cn3(double x)
       pre(x >= 0) cpre1
{
       if (!cpre1) { throw std::invalid_argument("x is negative"); }
       return std::sqrt(x);
}

// precondition is evaluated only in debug mode
double sqrt_cn4(double x)
       pre(x >= 0) cpre1
{
       assert(cpre1);
       return std::sqrt(x);
}

double set_psi(int psi)
       pre(psi >= minimum_psi) cpre1
       pre(psi <= maximum_psi) cpre2
{
       if (!cpre1) { log_houston(); psi = minimum_psi; }
       if (!cpre2) { log_houston(); psi = maximum_psi; }
       return (psi * 6.89476);
}

std::string extract(const char* pB, const char* pE)
       pre(pB != nullptr) cpre1
       pre(pE != nullptr) cpre2
       pre(pB <= pE) cpre3
{
       if (!cpre1) { log_houston(); return std:string(); }
       if (!cpre2) { log_houston(); return std:string(); }
       if (!cpre3) { log_houston(); return std:string(); }
       const int length = pE - pB;
       const std::string ns = std::string(pB, length);
       return ns;
}
```

Named Contracts on a Function

```
constexpr int r_good = 0;
constexpr int r_error = 1;
constexpr int r_ultimate_answer = 42;

int f1(const int x, int& result)
       // named preconditions
       pre(x != 1) cpre1
       pre(x != 2) cpre2
       pre(x != 3) cpre3
       pre(x != 4) cpre4
       pre(x != 5) cpre5
       pre(x != 6) cpre6
       pre(x != 7) cpre7
       // named post conditions on output parameter
       post(result == x) out_post1
       post(result != r_ultimate_answer) out_post2
       post(result == x) out_post3
       post(result != r_ultimate_answer) out_post4
       // named post conditions on return value
       post(r : r == r_good || r == r_error) r_post1
       post(r : r == r_good || r == r_error) r_post2
{
       // follow the rules of p2900r14 Contracts for C++
       cpre1.require();
       // document contract only
       cpre2.document();
       // debug halt contract only
       assert(cpre3);
       // exception throwing contract
       if (!cpre4) { throw std::exception(); }
       // error code contract return
       if (!cpre5) { return r_error; }
       // observe and continue contract
       if (!cpre6) { log_houston(); }
       // log and terminate program contract
       if (!cpre7) { log_houston(); std::terminate(); }

       result = x;
       return r_good;

       // list post condition evaluations at function end
out_post1:
       out_post1.ensure(); // rules of p2900r14
out_post2:
       out_post2.ensure(); // rules of p2900r14
r_post1:
       r_post1.ensure(); // rules of p2900r14
out_post3:
       if (!out_post3) { assert(false); return r_error; }
out_post4:
       if (!out_post4) { assert(false); return r_error; }
r_post2:
       if (!r_post2) { assert(false); }
}
```

Named Contracts on a Class Member Function

```
// starting point
class account
{
public:
	virtual bool open(const double x)
		pre(m_minimum >= 100.0) cpre1
		pre(m_balance == 0.0) cpre2
		pre(x >= m_minimum) cpre3
	{
		if (!cpre1) { err("minimum balance is too low"); return false; }
		if (!cpre2) { err("account is already open"); return false; }
		if (!cpre3) { err("opening deposit is too small"); return false; }
		m_balance = x;
		return true;
	}

	virtual bool withdraw(const double x)
		pre(x > 0.0) cpre1
		pre(x <= available()) cpre2
	{
		if (!cpre1) { err("withdraw is too small"); return false; }
		if (!cpre2) { err("withdraw is too large"); return false; }
		m_balance = (m_balance - x);
		return true;
	}

	double available() const { return (m_balance - m_minimum); }
	std::vector<std::string> get_errors() const { return m_errors; }

private:
	void err(std::string error) { m_errors.push_back(error); }

	double m_balance = 0.0;
	double m_minimum = 1000.0;
	std::vector<std::string> m_errors;
};

// an account restricted to withdraws on friday
class account_friday_withdraw : public account
{
public:
	virtual bool withdraw(const double x) override
		pre(is_friday()) cpre3
	{
		if (!cpre1) { err("withdraw is too small"); return false; }
		if (!cpre2) { err("funds are insufficient"); return false; }
		if (!cpre3) { err("withdraw day is not friday"); return false; }
		return account::withdraw(x);
	}
};
```

Named Contracts with Configuration

```
#include <bsls_assert.h>
#include <bsls_review.h>
int f_configure_bloomberg_bde_bsls(const int x)
      pre(x != 1) cpre1
      pre(x != 2) cpre2
      pre(x != 3) cpre3
      pre(x != 4) cpre4
      pre(x != 5) cpre5
      pre(x != 6) cpre6
{
      BSLS_ASSERT(cpre1); // enabled in non-opt build modes
      BSLS_REVIEW(cpre2); // enabled in non-opt build modes
      BSLS_ASSERT_SAFE(cpre3); // enabled in safe build modes
      BSLS_REVIEW_SAFE(cpre4); // enabled in safe build modes
      BSLS_ASSERT_OPT(cpre5); // enabled in all build modes
      BSLS_REVIEW_OPT(cpre6); // enabled in all build modes
      return x;
}

#include <boost/assert.hpp>
int f_configure_boost_assert(const int x)
      pre(x != 1) cpre1
      pre(x != 2) cpre2
      pre(x != 3) cpre3
      pre(x != 4) cpre4
{
      BOOST_ASSERT(cpre1);
      BOOST_ASSERT_MSG(cpre2, "precondition2");
      BOOST_VERIFY(cpre3);
      BOOST_VERIFY_MSG(cpre4, "precondition4");
      return x;
}

#include "base/logging.h"
int f_configure_google_check(const int x)
      pre(x != 1) cpre1
      pre(x != 2) cpre2
{
      DCHECK(cpre1);
      CHECK(cpre2);
      return x;
}

#include <libassert/assert.hpp>
int f_configure_libassert(const int x)
      pre(x != 1) cpre1
      pre(x != 2) cpre2
      pre(x != 3) cpre3
{
      // checked in debug but does nothing in release
      DEBUG_ASSERT(cpre1, x);
      // checked in both debug and release
      ASSERT(cpre2, x);
      // checked in debug and serves as an optimization hint in release
      ASSUME(cpre3, x);
      return x;
}
```

Named Contracts as Input Validation

```
// don't do this per ACCU 2025 Contracts for C++
int string_to_int_v01(const std::string& text)
	pre(text.size() > 0)
{
	// otherwise errors turn into bugs here per ACCU 2025 Contracts for C++
	int sign = 1; size_t i = 0;
	if (text[i] == '-' || text[i] == '+')
	{
		sign = (text[i] == '-') ? -1 : 1;
		i++;
	}
	// continues...
}

// do this instead per ACCU 2025 Contracts for C++
int string_to_int_v02(const std::string& text)
{
	if (text.size() == 0) { throw std::invalid_argument("empty text"); }

	int sign = 1; size_t i = 0;
	if (text[i] == '-' || text[i] == '+')
	{
		sign = (text[i] == '-') ? -1 : 1;
		i++;
	}
	// continues...
}

// named contract as input validation with exception throw
int string_to_int_v03(const std::string& text)
	pre(text.size() > 0) cpre1
{
	if (!cpre1) { throw std::invalid_argument("empty text"); }

	int sign = 1; size_t i = 0;
	if (text[i] == '-' || text[i] == '+')
	{
		sign = (text[i] == '-') ? -1 : 1;
		i++;
	}
	// continues...
}

// named contract as input validation with return value
int string_to_int_v04(const std::string& text)
	pre(text.size() > 0) cpre1
{
	if (!cpre1) { assert(false); return 0; }

	int sign = 1; size_t i = 0;
	if (text[i] == '-' || text[i] == '+')
	{
		sign = (text[i] == '-') ? -1 : 1;
		i++;
	}
	// continues...
}
```