Contract++ 0.4.1

Lorenzo Caminiti <lorcaminiti@gmail.com>

Table of Contents

Introduction 2
Full Table of Contents 5
Getting Started 9
Contract Programming Overview 11
Tutorial 19
Advanced Topics 34
Virtual Specifiers 65
Concepts 69
Named Parameters 78
Examples 89
Grammar 164
No Variadic Macros 179
Reference 183
Release Notes 210
Bibliography 213
Acknowledgments 215

> “*Our field needs more formality, but the profession has not realized it yet.*”
>
> --*Meyer (see [Meyer97] page 400)*

This library implements Contract Programming for the C++ programming language. In addition, the library implements virtual specifiers (`final`, `override`, and `new`, see C++11), concept checking, and named parameters.

Consult this documentation in HTML or PDF format.

Introduction

Contract Programming (CP) allows to specify preconditions, postconditions, and class invariants that are automatically checked when functions are executed at run-time. These conditions assert program specifications within the source code itself allowing to find bugs more quickly during testing, making the code self-documenting, and increasing overall software quality.

Contract Programming is also known as Design by Contract (DbC) [1] and it was first introduced by the Eiffel programming language (see [Meyer97]). All Contract Programming features of the Eiffel programming language are supported by this library, among others (see also the Features section):

1. Support for preconditions, postconditions, class invariants, block invariants, and loop variants.
2. Subcontract derived classes (with support for pure virtual functions and multiple inheritance).
3. Access expression old values and function return value in postconditions.
4. Optional compilation and checking of preconditions, postconditions, class invariants, block invariants, and loop variants.
5. Customizable actions on contract assertion failure (terminate by default but it can throw, exit, etc).

In addition, this library supports virtual specifiers, concept checking, and named parameters which together with contracts specify requirements of the program interface. This library is implemented for the C++03 standard and it does not require C++11.

An Example

The example below shows how to use this library to program contracts for the C++ Standard Template Library (STL) member function `std::vector::push_back` (in order to illustrate subcontracting, the `vector` class inherits from the somewhat arbitrary `pushable` base class). The syntax used by this library is compared side-by-side with the syntax proposed by [N1962] for adding Contract Programming to C++ (see also `push_back.cpp` and `pushable.hpp`):

[1] Design by Contract is a registered trademark of Eiffel Software.

#	This Library (C++03)	N1962 and N2081 Proposals (not adopted by C++11) [a]

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
```

```
#include <contract.hpp> // This library.
#include <boost/concept_check.hpp>
#include <vector>
#include "pushable.hpp" // Some base class.

CONTRACT_CLASS(
    template( typename T ) requires( boost::CopyConstruct↵
ible<T> ) // Concepts.
   class (vector) extends( public pushable<T> ) // Subcontracting.
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        empty() == (size() == 0) // More class invariants here...
    )

    public: typedef typename std::vector<T>::size_type size_type;
    public: typedef typename std::vector<T>::const_refer↵
ence const_reference;

    CONTRACT_FUNCTION_TPL(
        public void (push_back) ( (T const&) value ) override
            precondition(
                size() < max_size() // More preconditions here...
            )
            postcondition(
                auto old_size = CONTRACT_OLDOF size(), // Old-
of values.
                size() == old_size + 1 // More postconditions ↵
here...
            )
    ) {
        vector_.push_back(value); // Original function body.
    }

    // Rest of class here (possibly with more contracts)...
    public: bool empty ( void ) const { return vector_.empty(); }
    public: size_type size ( void ) const { return vec↵
tor_.size(); }
    public: size_type max_size ( void ) const { return vec↵
tor_.max_size(); }
    public: const_reference back ( void ) const { return vec↵
tor_.back(); }

    private: std::vector<T> vector_;
};
```

```
// Extra spaces, newlines, etc used to align text with this lib↵
rary code.
#include <concept>
#include <vector>
#include "pushable.hpp" // Some base class.


template< typename T > requires CopyConstructible<T> // Concepts.
class vector : public pushable<T> // Subcontracting.
{
    invariant {
        empty() == (size() == 0); // More class invariants here...
    }

    public: typedef typename std::vector<T>::size_type size_type;
    public: typedef typename std::vector<T>::const_refer↵
ence const_reference;


    public: void push_back ( T const& value ) override
        precondition {
            size() < max_size(); // More preconditions here...
        }
        postcondition {
            // Old-of values use `oldof` operator.
            size() == oldof size() + 1; // More postconditions ↵
here...
        }
    {
        vector_.push_back(value); // Original function body.
    }

    // Rest of class here (with possibly more contracts)...
    public: bool empty ( void ) const { return vector_.empty(); }
    public: size_type size ( void ) const { return vec↵
tor_.size(); }
    public: size_type max_size ( void ) const { return vec↵
tor_.max_size(); }
    public: const_reference back ( void ) const { return vec↵
tor_.back(); }

    private: std::vector<T> vector_;
};
```

[a] Unfortunately, the Contract Programming proposal [N1962] and the concept proposal [N2081] were never adopted by the C++ standard so the example on the right hand side will not compile.

Classes and functions are declared using the `CONTRACT_CLASS` and `CONTRACT_FUNCTION` macros respectively. Class invariants must appear at the very beginning of the class definition and they are specified using the `CONTRACT_CLASS_INVARIANT` macro. The `TPL` postfixes indicate versions of the macros that need to be used within templates.

Note the following differences between the syntax used by this library macros and the usual C++ declaration syntax (see the Differences with C++ Syntax section for a complete list):

- Round parenthesis `template( ... )` are used instead of angular parenthesis `template< ... >` to declare templates (line 7).
- Similarly to [N2081], the specifier `requires( ... )` is used to specify concepts to check (line 7).
- The class and function names are wrapped within round parenthesis `(vector)` and `(push_back)` (lines 8 and 18).

- The specifier `extends( ... )` is used instead of the column symbol `:` to inherit from base classes (line 8). This automatically subcontracts `vector` from `pushable` (when subcontracting, derived and base preconditions are checked in logic-or, derived and base postconditions are checked in logic-and, derived and base class invariants are checked logic-and).
- The access level `public`, `protected`, or `private` must always be specified for member functions but without the trailing column symbol `:` (line 18).
- The function result and parameter types are wrapped within round parenthesis `(T const&)` (line 18). The parenthesis are allowed but not required for fundamental types that contain no symbol (e.g., the function result type `void` at line 18, but also `bool`, `unsigned int const`, etc).
- Similarly to C++11, the specifier `override` is used to check that a member function is indeed overriding a base member function (line 18).
- The specifiers `precondition( ... )` and `postcondition( ... )` are used to program function preconditions and postconditions (lines 19 and 22). Furthermore, the `CONTRACT_OLDOF` macro can be used within postconditions to declare variables and initialize them to old values that the specified expressions had before executing the function body (line 23).
- Class invariants, preconditions, and postconditions assert boolean expressions separated by commas `,` (lines 11, 20, and 24).

Finally, note that the class and function definitions are programmed outside the macros so they retain their usual C++ syntax (e.g., the function body at line 27).

The library executes the following steps when the `push_back` function is called at run-time (see also the Contract Programming Overview section):

1. First, the class invariants and the function preconditions are checked.
2. Then, the function body is executed.
3. Last, the class invariants and the function postconditions are checked.

For example, if there is a bug in the function caller for which `push_back` is called when `size` is equal to `max_size` then the execution of the program will be interrupted reporting a failure of the first assertion in the preconditions and it will be evident that the bug is in the caller:

```
precondition number 1 "size() < max_size()" failed: file "push_back.cpp", line 26
```

Instead, if there is a bug in the `push_back` implementation for which `size` is not increased by one after `value` is added to `vector` by the function body then the execution will be interrupted reporting a failure of the first assertion in the postconditions and it will be evident that the bug is in `push_back` body:

```
postcondition number 1 "size() == old_size + 1" failed: file "push_back.cpp", line 26
```

By default, when assertions fail this library prints the above error messages to the standard error `std::cerr` and it terminates the program calling `std::terminate` (this behaviour can be customized to throw an exception, exit the program, etc, see the Broken Contracts section). Note that the library error messages contain enough information to uniquely identify the contract failure: Assertion type (class invariant, precondition, postcondition, etc), assertion number, asserted expression, file name, and line number. (The line number refers to the single line on which each macro expands so it will be the same for class invariants, preconditions, and postconditions of a given class or function but it can still be used together with the assertion number to uniquely identity which assertion failed.)

Language Support

This library suffers of two limitations:

1. The unusual syntax used to declare classes and functions within the macros which causes cryptic compiler errors when not used correctly (syntax error checking and reporting could be somewhat improved in future revisions of the library but there are fundamental limitations on what can be done using the preprocessor, see also the Grammar section).
2. High compilation times (the authors will try to reduce compilation times in future revisions of the library, see also the Cost section).

With the addition of contracts, concepts, and named parameters, C++ could introduce formal program specification into main-stream programming. The authors wish the work done in developing this library will persuade the C++ community and the C++ standard committee to add these features to the core language so to support formal program specification without the unusual macro syntax and avoiding high compilation times (unfortunately, this has not been the direction taken for C++11 with the rejection of the concept proposal [N2081] and the lack of consideration for the Contract Programming proposal [N1962], but there might still be hope for C++1x with x > 1).

Full Table of Contents

Introduction

Full Table Of Contents

Getting Started

- This Documentation
- Compilers and Platforms
- Installation
- Disable Contract Compilation

Contract Programming Overview

- Assertions
- Benefits
- Costs
- Free Function Calls
- Member Function Calls
- Constructor Calls
- Destructor Calls
- Constant-Correctness
- Specification vs. Implementation
- Broken Contracts
- Features

Tutorial

- Free Functions
- Preconditions
- Postconditions (Result and Old Values)
- Classes and Class Invariants
- Constructors
- Destructors
- Member Functions
- Inheritance and Subcontracting
- Class Templates
- Function Templates
- Forward Declarations and Body Definitions

Advanced Topics

- Commas and Leading Symbols in Macros
- Static Assertions
- Constant Assertions
- Select Assertions
- Assertion Statements
- Assertion Requirements
- Old-Of Requirements
- Old and Result Value Copies
- Pure Virtual Functions
- Subcontracting Preconditions
- Static Member Functions
- Volatile Member Functions
- Operators
- Nested Classes
- Friends

- Template Specializations
- Exception Specifications and Function-Try Blocks
- Specifying Types (no Boost.Typeof)
- Block Invariants and Loop Variants
- Contract Broken Handlers (Throw on Failure)

Virtual Specifiers

- Final Classes
- Final Member Functions
- Overriding Member Functions
- New Member Functions

Concepts

- Class Templates
- Function Templates
- Concept Definitions (Not Implemented)

Named Parameters

- Overview
- Named Function Parameters
- Deduced Function Parameters
- Member Function Parameters
- Constructor Parameters
- Class Template Parameters
- Concepts
- Parameter Identifiers

Examples

- [N1962] Vector: Comparison with C++ proposed syntax
- [N1962] Circle: Subcontracting
- [N1962] Factorial: Recursion and assertion computational complexity
- [N1962] Equal: Operators
- [N1962] Sum: Separated body definitions
- [N1962] Square Root: Default parameters and comparison with D syntax
- [N1962] Block: Block invariants
- [N2081] Add: Generic addition algorithm
- [N2081] Advance: Concept-based iterator overloading (emulated using tags)
- [N2081] Find: Generic find algorithm
- [N2081] Apply: Overloaded invocation of functors
- [N2081] For Each: Generic for-each algorithm
- [N2081] Transform: Generic binary transformation algorithm
- [N2081] Count: Generic counting algorithm
- [N2081] Convert: Conversion between two types
- [N2081] Equal: Generic equality comparison
- [N2081] Less Equal: Generic less-than or equal-to comparison
- [N2081] De-Ref: Generic iterator dereferencing
- [N2081] Min: Generic minimum algorithm
- [Meyer97] Stack4: Comparison with Eiffel Syntax
- [Meyer97] Stack3: Error codes instead of preconditions
- [Meyer97] GCD: Loop variants and invariants plus comparison with Eiffel syntax
- [Meyer97] Max-Array: Nested loop variants and invariants
- [Mitchell02] Name List: Relaxed subcontracts
- [Mitchell02] Dictionary: Simple key-value map
- [Mitchell02] Courier: Subcontracting and static class invariants
- [Mitchell02] Stack: Simple stack dispenser

[Mitchell02] Simple Queue: Simple queue dispenser
[Mitchell02] Customer Manager: Contracts instead of Defensive Programming
[Mitchell02] Observer: Contracts for pure virtual functions
[Mitchell02] Counter: Subcontracting and virtual specifiers (final, override, new, and pure virtual)
[Stroustrup97] String: Throw when contract is broken
[Cline90] Vector: Comparison with A++ proposed syntax
[Cline90] Stack: Function-Try blocks and exception specifications
[Cline90] Vector-Stack: Subcontracting from Abstract Data Type (ADT)
[Cline90] Calendar: A very simple calendar

Grammar

Preprocessor DSEL
Differences with C++ Syntax
Macro Interface
Lexical Conventions
Class Declarations
Base Classes
Template Specializations
Template Parameters
Concepts
Types
Function Declarations
Result Type
Function and Operator Names
Exception Specifications
Member Initializers
Function Parameters
Result and Old Values
Class Invariants
Assertions
Loop Variants
Named Parameter Declarations
Terminals
Alternative Assertion Syntax (Not Implemented)

No Variadic Macros

Sequence Syntax
Commas and Leading Symbols in Macros

Reference

contract::block_invariant_broken
contract::broken
contract::broken_contract_handler
contract::class_invariant_broken_on_entry
contract::class_invariant_broken_on_exit
contract::class_invariant_broken_on_throw
contract::copy
contract::from
contract::has_oldof
contract::loop_variant_broken
contract::postcondition_broken
contract::precondition_broken
contract::set_block_invariant_broken
contract::set_class_invariant_broken
contract::set_class_invariant_broken_on_entry
contract::set_class_invariant_broken_on_exit

contract::set_class_invariant_broken_on_throw
contract::set_loop_variant_broken
contract::set_postcondition_broken
contract::set_precondition_broken
CONTRACT_BLOCK_INVARIANT
CONTRACT_BLOCK_INVARIANT_TPL
CONTRACT_CLASS
CONTRACT_CLASS_INVARIANT
CONTRACT_CLASS_INVARIANT_TPL
CONTRACT_CLASS_TPL
CONTRACT_CONFIG_ARRAY_DIMENSION_MAX
CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS
CONTRACT_CONFIG_FUNCTION_ARITY_MAX
CONTRACT_CONFIG_INHERITANCE_MAX
CONTRACT_CONFIG_NO_BLOCK_INVARIANTS
CONTRACT_CONFIG_NO_CLASS_INVARIANTS
CONTRACT_CONFIG_NO_LOOP_VARIANTS
CONTRACT_CONFIG_NO_POSTCONDITIONS
CONTRACT_CONFIG_NO_PRECONDITIONS
CONTRACT_CONFIG_OLDOF_MAX
CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION
CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE
CONTRACT_CONFIG_THREAD_SAFE
CONTRACT_CONSTRUCTOR
CONTRACT_CONSTRUCTOR_ARG
CONTRACT_CONSTRUCTOR_BODY
CONTRACT_CONSTRUCTOR_TPL
CONTRACT_DESTRUCTOR
CONTRACT_DESTRUCTOR_BODY
CONTRACT_DESTRUCTOR_TPL
CONTRACT_FREE_BODY
CONTRACT_FUNCTION
CONTRACT_FUNCTION_TPL
CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES
CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS
CONTRACT_LIMIT_OLDOFS
CONTRACT_LOOP
CONTRACT_LOOP_VARIANT
CONTRACT_LOOP_VARIANT_TPL
CONTRACT_MEMBER_BODY
CONTRACT_OLDOF
CONTRACT_PARAMETER
CONTRACT_PARAMETER_BODY
CONTRACT_PARAMETER_TYPEOF
CONTRACT_TEMPLATE_PARAMETER

Release Notes

Bibliography

Acknowledgments

Getting Started

This section explains how to setup a system to use this library.

This Documentation

Programmers should have enough knowledge to use this library after reading the Introduction, Getting Started, and Tutorial sections. The other sections can be consulted to gain a more in depth knowledge of the library.

Some footnotes are marked by the word "**Rationale**". They explain reasons behind decisions made during the design and implementation of this library.

In most of the examples presented in this documentation the Boost.Detail/LightweightTest macro `BOOST_TEST` is used to assert test conditions (see also `boost/detail/lightweight_test.hpp`). The `BOOST_TEST` macro is conceptually similar to C++ `assert` but a failure of the checked condition does not abort the program, instead it makes `boost::report_errors` return a non-zero program exit code. [2]

Compilers and Platforms

The implementation of this library uses preprocessor and template meta-programming (as supported by Boost.Preprocessor and Boost.MPL respectively), templates with partial specializations and function pointers (similarly to Boost.Function), and local functions (as supported by Boost.LocalFunction). The authors originally developed and tested the library on:

1. GCC 4.5.3 on Cygwin (with and without C++11 features enabled `-std=c++0x`). [3]
2. Microsoft Visual C++ (MSVC) 8.0 on Windows XP and Windows 7.

At present, the library has not been tested on other compilers or platforms.

Installation

This library is composed of header files only. Therefore there is no pre-compiled object file which needs to be installed or linked. Programmers can simply instruct the C++ compiler where to find the library header files and they can start compiling code using this library.

> **Important**
>
> This library extensively uses Boost libraries. Boost version 1.50 must be properly installed in order for this library to compile.

Let *`ROOT`* be the root directory of this library installation then the directory structure is as follow:

```
ROOT/
    doc/
        html/   # This documentation.
    example/    # Examples using this library.
    include/    # This library source files (headers only).
```

For example, the following commands can be used to compile code using this library: [4]

```
$ g++ -I ROOT/include ...    # For GCC.
> cl  /I ROOT\include ...    # For MSVC.
```

All necessary library headers are included in the source code by the following instruction (it is not necessary to include single headers separately):

```
#include <contract.hpp> // Include this library headers.
```

[2] **Rationale.** Using Boost.Detail/LightweightTest allows to add the examples to the library regression tests so to make sure that they always compile and run correctly.
[3] When using GCC to compile large projects that use this library, it might be necessary to appropriately set the `--param gcc-min-expand` option to avoid internal compiler errors due to excessive virtual memory usage.
[4] For convenience, a `Jamfile.v2` file is provided in the example directory that can be used to compile and test all the examples using Boost.Jam. However, it is not necessary to use Boost.Jam to compile code that uses this library.

The following symbols are part of the library private interface, they are not documented, and they should not be directly used by programmers: [5]

- Any symbol defined by files within the `contract/aux_/` or `contract/detail/` directories (these header files should not be directly included by programmers).
- Any symbol within the `contract::aux` or `contract::detail` namespace.
- Any symbol prefixed by `contract_aux_...` or `contract_detail_...` (regardless of its namespace).
- Any symbol prefixed by `CONTRACT_AUX_...` or `CONTRACT_DETAIL_...` (regardless of its namespace).

Symbols starting with `ERROR_...` are used to report compile-time errors via static assertions and programmers should not use these symbols to define macros or other constructs in the global namespace.

Disable Contract Compilation

Some of the library behaviour can be customized at compile-time by defining special *configuration macros* (see `contract/config.hpp`). In particular, the following configuration macros can be used to selectively turn on or off contract compilation and the related run-time checks:

- Defining the `CONTRACT_CONFIG_NO_PRECONDITIONS` macro turns off compilation and run-time checking of all preconditions.
- Defining the `CONTRACT_CONFIG_NO_POSTCONDITIONS` macro turns off compilation and run-time checking of all postconditions.
- Defining the `CONTRACT_CONFIG_NO_CLASS_INVARIANTS` macro turns off compilation and run-time checking of all class invariants.
- Defining the `CONTRACT_CONFIG_NO_BLOCK_INVARIANTS` macro turns off compilation and run-time checking of all block invariants.
- Defining the `CONTRACT_CONFIG_NO_LOOP_VARIANTS` macro turns off compilation and run-time checking of all loop variants.

By default, all contracts are compiled and checked at run-time (i.e., all the macros above are not defined).

> **Important**
>
> In Contract Programming, it is usually important to selectively turn off contract compilation to reduce run-time, binary size, and compilation-time overhead associated with the contracts (see [Meyer97]). This library guarantees zero run-time and binary size overhead when all contracts are all turned off (however, even when contracts are all turned off there is a limited compile-time overhead associated with expanding the contract macros to generate the original class and function declarations). Note that when contracts are turned off their assertions are completely ignored by the compiler so the assertion code might not even be syntactically correct.

For example, the following commands compile and check preconditions and class invariants, but they do not compile and check postconditions, block invariants, and loop variants:

```
$ g++ -DCONTRACT_CONFIG_NO_POSTCONDITONS -DCONTRACT_CONFIG_NO_BLOCK_INVARIANTS -DCONTRACT_CONFIG_NO_LOOP_VARIANTS ...   # For GCC.
> cl  /DCONTRACT_CONFIG_NO_POSTCONDITONS /DCONTRACT_CONFIG_NO_BLOCK_INVARIANTS /DCONTRACT_CONFIG_NO_LOOP_VARIANTS ...   # For MSVC.
```

Other configuration macros are provided to customize other aspects of the library. For example, the `CONTRACT_CONFIG_FUNCTION_ARITY_MAX` macro is used to specify the maximum number of function parameters and the `CONTRACT_CONFIG_INHERITANCE_MAX` macro is used to specify the maxim number of base classes. All configuration macros have appropriate default values when they are left undefined by programmers.

[5] **Rationale.** This library concatenates symbols specified by programmers (e.g., the function name) with other symbols (e.g., special prefixes and line numbers) to make internal symbols with unique names to avoid name clashes. These symbols are separated by the letter `X` when they are concatenated so they read more easily during debugging (unfortunately, the underscore character `_` could not be used instead of the letter `X` because if the original symbol already contained a leading or trailing underscore, the concatenation could result in a symbol with double underscores `___` which is reserved by the C++ standard). The “aux” symbols are internal to the implementation of this library. The “detail” symbols are not officially part of the library public interface and they are not documented however they constitute a separate set of standalone libraries that could be added to the library public interface in the future.

Contract Programming Overview

> *"It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?"*
>
> *--Hoare (see [Hoare73])*

This section gives an overview of Contract Programming (see [Meyer97], [Mitchell02], and [N1613] for a detailed introduction to Contract Programming).

Note

The objective of this library is *not* to convince programmers to use Contract Programming. It is assumed that programmes understand the benefits and trade-offs associated with Contract Programming and they have already decided to use this methodology to formally program specifications. Then, this library aims to be the best Contract Programming library for C++.

Assertions

Contract Programming is characterized by the following type of assertion mechanisms.

Assertion	Purpose
Preconditions	These are logical conditions that programmers expect to be true when the function is called (e.g., to check constraints on the function arguments).
Postconditions	These are logical conditions that programmers expect to be true when the function has ended normally (e.g., to check the result and any side effect that a function might have). Postconditions can usually access the function result value (for non-void functions) and *old values* that expressions had before the body execution.
Class Invariants	These are logical conditions that programmers expect to be true after the constructor has been executed successfully, before and after the execution of every non-static public member function, and before the destructor is executed (e.g, class invariants can define valid states of all objects of a class). It is possible to describe a different set of class invariants for volatile member functions but *volatile class invariants* are assumed to be same as class invariants unless differently specified. It is also possible to describe *static class invariants* which are expected to be true before and after the execution of any public member function (even if static), including constructor entry and destructor exit. [a]
Subcontracting	Subcontracting is defined using the substitution principle and it predicates that: Preconditions cannot be strengthen, postconditions and class invariants cannot be weaken.
Block Invariants	These are logical conditions that programmers except to be true every time the execution reaches the point where the condition is asserted. When used within a loop (i.e., a block of code that can be executed in iteration), block invariants are also called *loop invariants* and they assert conditions that are expected to be true for every loop iteration.
Loop Variants	For a given loop, a loop variant is a non-negative integer expression (`>= 0`) with a value that is expected to decrease at every subsequent loop iteration. Loop variants are used to ensure that loops terminate avoiding infinite iterations.

[a] **Rationale.** Static and volatile class invariants were first introduced by this library to reflect the fact that C++ support both static and volatile member functions. Static and volatile class invariants are not part of [N1962].

It is a common Contract Programming requirement to disable other contracts while a contract assertions is being evaluated (in order to avoid infinite recursive calls). This library implement this feature however it should be noted that in order to globally disable assertions while checking another assertion, some type of global variable needs to be used. In multi-threaded environments, programmers can define the `CONTRACT_CONFIG_THREAD_SAFE` configuration macro to protect such a global variable from racing conditions (but that will effectively introduce a global lock in the program).

A limited form of Contract Programming is the use of the C++ `assert` macro. Using `assert` is common practice for many programmers but unfortunately it suffers from the following problems (that are instead resolved by using Contract Programming):

1. `assert` is used within the implementation therefore the asserted conditions are not easily visible to the caller which is only familiar with the class and function declarations.
2. `assert` does not distinguish between preconditions and postconditions. In well-tested production code, postconditions can be disabled trusting the correctness of the implementation while preconditions might still need to remain enabled because of the evolution of the calling code. Using `assert` it is not possible to selectively disable only postconditions and all assertions must be disabled at once.

3. `assert` requires to manually program extra code to check class invariants (e.g., extra member functions and try blocks).

4. `assert` does not support subcontracting.

Benefits

The main use of Contract Programming is to improve software quality. [Meyer97] discusses how Contract Programming can be used as the basic tool to write "correct" software. Furhtermore, [Stroustrup97] discusses the key importance of class invariants plus advantages and disadvantages of preconditions and postconditions. The following is a short summary of the benefits associated with Contract Programming mainly taken from [N1613].

#	Topic	Benefit
1.	Preconditions and Postconditions	Using function preconditions and postconditions, programmers can give a precise semantic description of what a function requires at its entry and what it ensures under its (normal) exit. In particular, using postcondition old values, Contract Programming provides a mechanism that allows programmers to compare values of an expression before and after the function body execution. This mechanism is powerful enough to enable programmers to express many constraints within the code, constraints that would otherwise have to be captured at best only informally by the code documentation.
2.	Class Invariants	Using class invariants, programmers can describe what to expect from a class and the logic dependencies between the class members. It is the job of the constructor to ensure that the class invariants are satisfied when the object is first created. Then the implementation of the member functions can be largely simplified as they can be written knowing that the class invariants are satisfied because Contract Programing checks them before and after the execution of every member function. Finally, the destructor makes sure that the class invariants hold for the entire life of the object, checking the class invariants one last time before the object is destructed.
3.	Self-Documenting Code	Contracts are part of the source code, they are executed and verified at run-time so they are always up to date with the code itself. Therefore the specifications, as documented by the contracts, can be trusted to always be up to date with the implementation code.
4.	Easier Debugging	Contract Programming can provide a powerful debugging facility because, if contracts are well written, bugs will cause contract assertions to fail exactly where the problem first occurs instead than at some later stage of the program execution in an apparently unrelated manner. Note that a precondition failure points to a bug in the function caller, a postcondition failure points instead to a bug in the function implementation. Furthermore, in case of contract failure, this library provides detailed error messages that greatly helps debugging. [a]
5.	Easier Testing	Contract Programming facilitates testing because a contract naturally specifies what a test should check. For example, preconditions of a function state which inputs cause the function to fail and postconditions state which outputs are produced by the function on normal exit.
6.	Formal Design	Contract Programming can serve to reduce the gap between designers and programmers by providing a precise and unambiguous specification language. Moreover, contracts can make code reviews easier.
7.	Formal Inheritance	Contract Programming formalizes the virtual function overriding mechanism using subcontracting as justified by the substitution principle. This keeps the base class programmers in control as overriding functions still have to fully satisfy the base class contracts.
8.	Replace Defensive Programming	Contract Programming assertions can replace Defensive Programming checks localizing these checks within the contract and making the code more readable.

[a] Of course, if the contract is ill written then Contract Programming is of little use. However, it is less likely to have a bug in both the function body and the contract than in the function body only. For example, consider the validation of a result in postconditions. Validating the return value might seem redundant, but in this case we actually want that redundancy. When programmers write a function, there is a certain probability that they make a mistake in implementing the function body. When programmers specify the result of the function in the postconditions, there is also a certain probability that they make a mistake in writing the contract. However, the probability that programmers make a mistake twice (in both the body *and* the contract) is lower than the probability that the mistake is made just once (in either the body or the contract).

Costs

Contract Programming benefits come to the cost of performance as discussed in detail by both [Stroustrup97] and [Meyer97]. However, while performance trade-offs should be carefully considered depending on the specific application domain, software quality cannot be sacrificed (it is difficult to see the value of software that quickly and efficiently provides an incorrect result).

The run-time performances are negatively impacted by Contract Programming mainly because of the following:

1. The extra processing required to check the assertions.
2. The extra processing required by the additional function calls (additional functions are invoked to check preconditions, postconditions, class invariants, etc).
3. The extra processing required to copy expression results when old values that are used in postconditions.

To mitigate the run-time performance impact, programmers can selectively turn off some of the contract compilation and the related run-time checking. Programmers will have to decide based on the performance trade-offs required by their applications but a reasonable approach usually is to:

- Always write contracts to clarify the semantics of the design embedding the specifications directly into the code and making the code self-documenting.
- Enable preconditions, postconditions, and class invariants compilation and checking during initial testing.
- Enable only preconditions (and possibly class invariants) during release testing and for the final release (see also `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, and `CONTRACT_CONFIG_NO_CLASS_INVARIANTS`).

This approach is usually reasonable because in well-tested production code, validating the function body implementation using postconditions and class invariants is rarely needed since the function has shown itself to be “correct” during testing. On the other hand, checking function arguments using postconditions is always needed because of the evolution of the calling code. Furthermore, postconditions are usually computationally more expensive to check (see the Assertion Requirements section for a mechanism to selectively enable assertions also based on their computational complexity).

Compile-time performances are also impacted by this library mainly because:

1. The contracts appear in class and function declarations (usually header files) so they have to be re-compiled for each translation unit.
2. The library implementation extensively uses preprocessor and template meta-programming which significantly stress compiler performances.

Warning

Unfortunately, the current implementation of this library significantly slows down compilation. For example, for a project with 122 files and 15,471 lines of code, adding contracts to a total of 50 classes and 302 functions increased compilation time from 1 minute to 26 minutes when compilation was enabled for all contracts, and from 1 minute to 2 minutes when contracts were left in the code but contract compilation was disabled using this library configuration macros `CONTRACT_CONFIG_NO_...`. [6]

On compilers that support pre-compiled headers (GCC, MSVC, etc), these can be used to reduce re-compilation time. For example, re-compilation time of `vector.cpp` from the Examples section is reduced from 44 seconds to 24 seconds (55% faster) when its header `vector.hpp` is pre-compiled (with all contracts enabled). [7]

The authors have not done a detailed preprocessing-time and compile-time analysis of the performances of this library. Therefore, the authors have not yet tried to optimize this library compilation-time. Reducing compilation-time will be a major focus of future releases (see also Ticket 48).

Finally, Contract Programming should be seen as a tool to complement (and not to substitute) testing.

Free Function Calls

A *free function* [8] call executes the following steps:

1. Check the function preconditions.
2. Execute the function body.
3. Check the function postconditions.

[6] This library macros always need to expand to generate the class and function declarations even when contract compilation is disabled. That is why there is a compile-time overhead, even if significantly smaller, also when contracts are all disabled (in this case however there is zero run-time overhead). The compilation time overhead when all contracts are turned off could be further reduced by optimizing the library implementation to not include internal headers that are not required when contracts are off (this type of optimizations will be a major focus on future releases).
[7] There is essentially no gain in pre-compiling this library headers because most of the compilation time is taken by expanding and compiling this library macros as they appear in the user code (and not by the library implementation code itself). Furthermore, pre-compiled headers should be used with the usual care because they can complicate the build system and they introduce extra dependencies that might limit the amount of targets that can be built simultaneously in a parallel build. However, if properly used, pre-compiled headers can significantly reduce re-compilation time when using this library.
[8] In C++, a free function is any function that is not a member function.

Member Function Calls

A member function call executes the following steps:

1. Check the static class invariants.
2. Check the non-static class invariants (in logic-and with the base class invariants when subcontracting). These are checked only for non-static member functions. [9] Volatile member functions check volatile class invariants instead.
3. Check the function preconditions (in logic-or with the overridden function preconditions when subcontracting).
4. Execute the function body.
5. Check the static class invariants (even if the body throws an exception).
6. Check the non-static class invariants (in logic-and with the base class invariants when subcontracting). These are checked only for non-static member functions and even if the body throws an exception. Volatile member functions check volatile class invariants instead.
7. Check the function postconditions (in logic-and with the overridden function postconditions when subcontracting). These are checked only if the body does not throw an exception.

In this documentation logic-and and logic-or are the logic *and* and *or* operations evaluated in short-circuit:

- p logic-and q is true if and only if both p and q are true but q is evaluated only when p is true.
- p logic-or q is true if and only if either p or q is true but q is evaluated only when p is false.

Class invariants are checked before preconditions and postconditions so that preconditions and postconditions assertions can be simplified by being programmed under the assumption that class invariants are satisfied (e.g., if class invariants assert that a pointer cannot be null then preconditions and postconditions can safety dereference the pointer without additional checking). Similarly, subcontracting checks contracts of base classes before checking the derived class contracts so that the base class contract can be programmed under the assumption that the base class contracts are satisfied. When a member function overrides more than one virtual base function because of multiple inheritance:

- Class invariants are checked in logic-and with the class invariants of all base classes following the inheritance order.
- Preconditions are checked in logic-or with the preconditions of all overridden functions following the inheritance order.
- Postconditions are checked in logic-and with the postconditions of all overridden functions following the inheritance order.

Note that:

- Preconditions and postconditions of static member functions cannot access the object.
- Preconditions, postconditions, and volatile class invariants of volatile member functions access the object as (constant) volatile.

Constructor Calls

A constructor call executes the following steps:

1. Check the constructor preconditions.
2. Initialize base classes and members executing the constructor member initialization list if present.
3. Check the static class invariants (but not the non-static class invariants).
4. Execute the constructor body.
5. Check the static class invariants (even if the body throws an exception).
6. Check the non-static class invariants, but only if the body did not throw an exception.
7. Check the constructor postconditions, but only if the body did not throw an exception.

[9] Static member functions cannot be virtual so they cannot be overridden and they do not subcontract.

Preconditions are checked before initializing base classes and members so that these initializations can relay on preconditions to be true (for example to validate constructor arguments before they are used to initialize member variables). C++ object construction mechanism will automatically check base class contracts when subcontracting.

Note that:

- Non-static class invariants are not checked at constructor entry (because there is no object before the constructor body is executed).
- Preconditions cannot access the object (because there is no object before the constructor body is executed).
- Postconditions cannot access the object old value (because there was no object before the constructor body was executed).
- The object is never volatile within constructors so constructors do not check volatile class invariants.

Destructor Calls

A destructor call executes the following steps:

1. Check the static class invariants.
2. Check the non-static class invariants.
3. Execute the destructor body.
4. Check the static class invariants (even if the body throws an exception). [10]
5. Check the non-static class invariants, but only if the body threw an exception.

C++ object destruction mechanism will automatically check base class contracts when subcontracting.

Note that:

- Destructors have no parameter and they can be called at any time after object construction so they have no preconditions.
- After the destructor body is executed, the object no longer exists so non-static class invariants do not have to be true and they are not checked at destructor exit.
- Destructors have no postconditions because they have no parameter and after the body execution there is no object. [11]
- The object is never volatile within destructors so destructors do not check volatile class invariants.

Constant-Correctness

Contracts are only responsible to check the program state in oder to ensure its compliance with the specifications. Therefore, contracts should not be able to modify the program state and exclusively "read-only" operations (or *queries*) should be used to program contracts.

This library enforces this constraint at compile-time using C++ `const` qualifier. [12] Contracts only have access to the object, function arguments, and function return value via constant references `const&` or pointers `const*`. Other variables (static data members, global variables, etc) can be explicitly made constant using *constant assertions* (see the Advanced Topics section).

Specification vs. Implementation

Contracts are part of the program specification and not of its implementation. Therefore, contracts should appear within C++ declarations and not within definitions. [13]

Contracts are most useful when they assert conditions only using public members. For example, the caller of a member function cannot in general make sure that the member function preconditions are satisfied if these preconditions use private members that are not accessible by the caller. Therefore, a failure in the preconditions will not necessarily indicate a bug in the caller given that the caller was not able to fully check the preconditions before calling the member function. In most cases, the need of using non-public members to check contracts indicates

[10] For generality, this library does not require the destructor body to not throw exceptions. However, in order to comply with the STL exception safety requirements, destructors should never throw.
[11] In theory, destructors could have static postconditions (i.e., postconditions that are not allowed to access the object which no longer exists after destruction). Still destructors shall never have preconditions because a destructor can be called at any point after the object is constructed as long the class invariants hold. None of the Contract Programming references that the authors have studied propose static postconditions for destructor (neither [N1962] nor [Meyer97], but Eiffel has no static data member). Future revisions of this library might implement destructor static postconditions (e.g., a destructor postconditions of a class that counts object instances could assert that the instance counter stored in a static data member should be decreased of one because the object has been destructed, see also Ticket 41).
[12] As usual in C++, constant-correctness can be enforced at compile-time only as long as programmers do not use `const_cast` and `mutable`.
[13] This is a major conceptual difference with respect to Defensive Programming and using `assert` because they program assertions within the function body instead that with the function declaration.

an error in the design of the class. However, given that C++ provides programmers ways around access level restrictions (e.g., `friend` and function pointers), this library leaves it up to the programmers to make sure that only public members are used in contract assertions ([N1962] follows the same approach not forcing contracts to only use public members, Eiffel instead generates a compiler error if preconditions use non-public members). [14]

Finally, only public member functions shall check class invariants. Private and protected member functions are allowed to brake class invariants because private and protected member are part of the class implementation and not of its specification.

Broken Contracts

After programmers specify contracts, this library automatically checks preconditions, postconditions, class invariants, block invariants, and loop variants at run-time. If a precondition, postcondition, class invariant, block invariants, or loop variant is checked to be false or throws an exception then the library invokes the `contract::precondition_broken`, `contract::postcondition_broken`, `contract::class_invariant_broken_on_entry` or `contract::class_invariant_broken_on_exit` or `contract::class_invariant_broken_on_throw`, `contract::block_invariant_broken`, and `contract::loop_variant_broken` function respectively.

These function invoke `std::terminate()` by default but programmers can redefine them to take a different action (throw an exception, exit the program, etc) using `contract::set_precondition_broken`, `contract::set_postcondition_broken`, `contract::set_class_invariant_broken`, `contract::set_block_invariant_broken`, and `contract::set_loop_variant_broken` (this mechanism is similar to the one of C++'s `std::terminate()`). See the Contract Broken Handlers section for an example.

Features

The design of this library was largely based on [N1962] and on the Eiffel programming language as specified by [Meyer97]. The following table compares features between this library, the proposal for adding Contract Programming to the C++ standard [N1962], the Eiffel programming language [Meyer97], and the D programming language [Bright04].

[14] **Rationale.** In theory, if C++ defect 45 were not fixed, this library could have generated a compile-time error for preconditions that use non-public members.

Feature	**This Library (C++03)**	**[N1962] Proposal (not part of C++)**	**ISE Eiffel 5.4**	**D**
Keywords and Specifiers	`precondition`, `postcondition`, `extends`, `initialize`, `requires`, `in`, `out`, `deduce`, `comma` (these are specifiers and not keywords, they have special meaning only in specific positions within the declarations passed to this library macros)	`invariant`, `precondition`, `postcondition`, `oldof`	`invariant`, `require`, `ensure`, `do`, `require else`, `ensure then`, `old`, `result`, `variant`	`invariant`, `in`, `out`, `assert`, `static`
On contract failure	Default to `std::terminate()` (but can be customized to exit, abort, throw exceptions, etc).	Default to `std::terminate()` (but can be customized to exit, abort, throw exceptions, etc).	Throw exceptions	Throw exceptions
Result value in postconditions	Yes, `auto` *`result-variable-name`* `= return`.	Yes, `postcondition (`*`result-variable-name`*`)`.	Yes, `result` keyword.	No.
Old values in postconditions	Yes, `auto` *`old-variable-name`* `= CONTRACT_OLDOF` *`oldof-expression`*.	Yes, `oldof` *`oldof-expression`*.	Yes, `old` *`old-expression`*.	No.
Subcontracting	Yes, also support multiple base contracts for multiple inheritance (can force preconditions to be specified only by base classes using `CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS`).	Yes, also support multiple base contracts but only base classes can specify preconditions.	Yes.	Yes.
Contracts for pure virtual functions	Yes.	Yes.	Yes (contracts for abstract functions).	No (but planned).
Arbitrary code in contracts	No, assertions only.	No, assertions only.	No, assertions only plus preconditions can only access public members.	Yes.
Constant-correct	Yes.	Yes.	Yes.	No.
Function code ordering	Preconditions, postconditions, body.	Preconditions, postconditions, body.	Preconditions, body, postconditions.	Preconditions, postconditions, body.
Static assertions	Yes, can use `static_assert` (internally implemented using `BOOST_MPL_ASSERT_MSG`, no C++11 required).	Yes, can use C++11 `static_assert`.	No.	Yes.
Block invariants	Yes, `CONTRACT_BLOCK_INVARIANT`.	Yes, `invariant`.	Yes (`check` instruction and loop invariants).	No (just `assert`).
Loop variants	Yes, `CONTRACT_LOOP_VARIANT`.	No.	Yes.	No.
Disable assertion checking within assertions checking	Yes (use `CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION` for preconditions to disable no assertion). Use `CONTRACT_CONFIG_THREAD_SAFE` to make the implementation of this feature thread-safe in multi-threaded programs (but it will introduce a global lock).	Yes for class invariants and postconditions but preconditions disable no assertion.	Yes.	No.
Assertion requirements	Yes (compilation and run-time checking of single assertions can be disabled when specific assertion requirements are not met).	No.	No (but all types are `CopyConstructible`, `Assignable`, and `EqualityComparable` in Eiffel so there is not a real need for this).	No.
Nested member function calls	Disable nothing. [a]	Disable nothing.	Disable all checks.	Disable nothing.
Non-static class invariants checking	At constructor exit, at destructor entry, and at public member function entry, exit, and on throw (but only if programmers declare these functions using this library macros). Same for volatile class invariants.	At constructor exit, at destructor entry, and at public member function entry, exit, and on throw (volatile class invariants not supported).	At constructor exit, and around public member functions.	At constructor exit, at destructor entry, an around public member functions.
Static class invariants checking	At entry and exit of any (also static) member function, constructor, and destructor.	No.	No (but Eiffel does not have static members).	No.

Feature	This Library (C++03)	[N1962] Proposal (not part of C++)	ISE Eiffel 5.4	D
Removable from object code	Yes, using any combination of `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, `CONTRACT_CONFIG_NO_CLASS_INVARIANTS`, `CONTRACT_CONFIG_NO_BLOCK_INVARIANTS`, and `CONTRACT_CONFIG_NO_LOOP_VARIANTS`.	Yes.	Yes (but predefined combinations only).	Yes.

[a] **Rationale.** Older versions of this library automatically defined a data member used to disable checking of class invariants within member function calls. However, this feature, which was required by older revisions of [N1962], is no longer required by [N1962], it complicates the implementation, and in multi-thread programs would introduce a lock that synchronizes all member functions calls for a given object so it was removed in the current revision of the library.

Contract Programming is also provided by the following references.

Reference	Language	Notes
[Bright04b]	C++	The Digital Mars C++ compiler extends C++ adding Contract Programming language support (among many other features).
[Lindrud04]	C++	This supports class invariants and old values but it does not support subcontracting, contracts are specified within definitions instead of declarations, assertions are not constant-correct (unfortunately, these missing features are all essential to Contract Programming).
[Tandin04]	C++	Interestingly, these contract macros automatically generate Doxygen documentation [a] but old values, class invariants, and subcontracting are not supported plus contracts are specified within definitions instead of declarations (unfortunately, these missing features are all essential to Contract Programming).
[Maley99]	C++	This supports Contract Programming including subcontracting but with some limitations (e.g., programmers need to manually build an inheritance tree using artificial template parameters), it does not use macros so programmers are required to write by hand a significant amount of boiler-plate code. (The authors have found this work very inspiring when they started to develop this library.)
[C2]	C++	This uses an external preprocessing tool (the authors could no longer find this project codebase to evaluate it).
[iContract]	Java	This uses an external preprocessing tool.
[Jcontract]	Java	This uses an external preprocessing tool.
[CodeContracts]	.NET	Microsoft Contract Programming for .NET programming languages.
[SpecSharp]	C#	This is a C# extension with Contract Programming language support.
[Chrome02]	Object Pascal	This is the .NET version of Object Pascal and it has language support for Contract Programming.
[SPARKAda]	Ada	This is an Ada-like programming language with Contract Programming support.

[a] **Rationale.** Older versions of this library used to automatically generate Doxygen documentation from the contract macros. This functionality was abandoned for a number of reasons: this library macros became too complex and the Doxygen preprocessor is no longer able to expand them; the Doxygen documentation was just a repeat of the contract code (so programmers can directly look at contracts in the header files), Doxygen might not necessarily be the documentation tool chosen by all programmers.

Typically, preprocessing tools external to the language work by transforming specially formatted code comments into contract code that is then checked at run-time. One of this library primary goals was to support Contract Programming entirely within C++ and without using any tool external to the standard language (C++ macros were used instead of external preprocessing tools).

To the authors' knowledge, this the only library that fully support Contract Programming for C++ (see the Bibliography section for the complete list of Contract Programming references studied by the authors).

Tutorial

This section explains how to program contracts using this library. See the Grammar section for a complete guide on this library syntax.

Free Functions

Consider the following free function `postinc` which performs a post-increment on its parameter (see also `no_contract_postinc.cpp`):

```
int const postinc ( int& value )
    // precondition:    value < std::numeric_limits<int>::max()
    // postcondition:   value == oldof value + 1
    //                  result == oldof valule
{
    return value++;
}
```

Let's now program the function declaration using the `CONTRACT_FUNCTION` macro but without programming the contract yet (see also `no_pre_post_postinc.cpp`):

```
#include <contract.hpp>

CONTRACT_FUNCTION( // Declare free function (for programming contracts).
    int const (postinc) ( (int&) value ) // Non-fundamental types in `()`.
        // precondition:    value < std::numeric_limits<int>::max()
        // postcondition:   value == oldof value + 1
        //                  result == oldof value
) {
    return value++;
}
```

All necessary header files for this library are included by `#include <contract.hpp>` (see also the Getting Started section).

The function body is programmed outside the library macro. Therefore, while this library alters the function declaration syntax, it does not change the syntax used to implement the function.

The function name `(postinc)` must always be wrapped within parenthesis so it can be passed to this library macros.

The function result and parameter types must be wrapped within parenthesis unless they are fundamental types containing no symbol (symbols are tokens different from the alphanumeric tokens `a-z`, `A-Z`, `0-9`). In this example, the result type `int const` is a fundamental type and it contains no symbol so the parenthesis around it `(int const)` are allowed but not required. [15] Instead, the parameter type `int&` is a fundamental type but it contains the reference symbol `&` so it must be wrapped within parenthesis `(int&)`.

Important

In general, every token which is not a known keyword (`int` is a known keyword but the function name is not) or that contains a non-alphanumeric symbol (e.g., `int&`) must be wrapped within round parenthesis `(...)` unless it is the very last token of a given syntactic element (e.g., the function parameter name `value` is the last token of the parameter declaration). [16]

See the Grammar section for more information.

Each function parameter must always specify both its type and its name (parameter names can instead by omitted in usual C++ declarations). [17] As usual, additional parameters can be specified separated by commas `,` on compilers that support variadic macros (these include most modern compilers, MSVC, GCC, and all C++11 compilers, see the No Variadic Macros section for compilers that do not support variadic macros). The configuration macro `CONTRACT_CONFIG_FUNCTION_ARITY_MAX` indicates the maximum number of function parameters that can be specified. An empty parameter list must be specified using `void` (this is similar to the usual C++ syntax that allows to declare a function with no parameter using *`result-type function-name`* `( void )`). [18] Default parameter values can be specified using `, default`

[15] In the examples presented in this documentation, extra parenthesis are in general avoided unless strictly required (for example, extra parenthesis around fundamental types containing no symbol `void`, `bool`, `unsigned int const`, etc are always omitted). In the authors' opinion, the syntax is more readable with lesser parenthesis. However, other programmers might find it more readable to always specify extra parenthesis around result and parameter types for consistency even when they are not strictly required.

[16] **Rationale.** This library uses preprocessor meta-programming to parse declarations and contracts of classes and functions. The preprocessor cannot parse a token if it is not known a priori or if it contains symbols, unless such a token is wrapped within round parenthesis `()`. For example, the function name is arbitrary, it cannot be known a priori, so it must always be wrapped within parenthesis. If a type is a fundamental type then it is known a priori (because it is composed of known C++ keywords `int`, `const`, `unsigned`, etc), and if the fundamental type also contains no symbols (`&`, `*`, etc) then the parenthesis around such a type are optional.

[17] **Rationale.** It would be possible to modify the library syntax to make parameter names optional but that will complicate the library implementation without adding any feature for programmers.

[18] **Rationale.** Unfortunately, it is not possible to specify an empty parameter list simply as *`result-type function-name`* `( )` because the preprocessor can only parse `( )` if empty macro parameters are supported. Empty macro parameters together with variadic macros where added to C99 and the preferred syntax of this library uses variadic macros. However, not all compilers (notably MSVC) that support variadic macros also correctly support empty macro parameters so `( void )` is always used instead of `( )` to increase portability.

default-value immediately following the parameter name. The storage classifiers `auto` and `register` can be specified as usual as part of the parameter type. [19] Functions can be overloaded by specifying different parameter types and they can recursively call themselves as usual.

For example, the following function `postinc` is overloaded to work on `long` instead of `int` numbers, it is implemented using recursion, it increments the specified number by `offset` which is stored into a register variable and it has a default value of `1` (see also `params_postinc.cpp`):

```
CONTRACT_FUNCTION( // Overload, storage classifiers, and default parameters.
    long const (postinc) ( (long&) value, register unsigned offset, default 1 )
        // precondition:    value < std::numeric_limits<long>::max()
        // postcondition:   value == oldof value + 1
        //                  result == oldof value
) {
    long result = value;
    if(offset > 0) {
        value += 1;
        postinc(value, offset - 1); // Recursive calls.
    }
    return result;
}
```

```
long value = 1;
BOOST_TEST(postinc(value) == 1);    // Increment of 1 (it is 1).
BOOST_TEST(postinc(value, 4) == 2); // Increment of 4 (it is 2).
BOOST_TEST(value == 6);             // Incremented of 4 (it was 2).
```

Function and array types cannot be directly used as function parameter types within the contract macros but extra `typedef` declarations can be used to workaround this limitation (for multi-dimensional arrays, the maximum number of supported array dimensions is specified by the `CONTRACT_CONFIG_ARRAY_DIMENSION_MAX` macro). For example (see also `params_funcptr_array_apply.cpp`):

```
// Contracts cannot directly use array type syntax, use via `typedef` instead.
typedef int array2x3[2][3];
typedef int (*funcptr) ( int );

CONTRACT_FUNCTION(
    void (apply) ( (funcptr) f, (array2x3) m )
        // precondition:    f != NULL
        //                  m != NULL
) {
    for ( size_t i = 0; i < 2; ++i)
        for ( size_t j = 0; j < 3; ++j)
            m[i][j] = f(m[i][j]);
}

int offset ( int i ) { return i + 10; }
```

```
int x[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
apply(offset, x);
```

Spaces and new-lines make no syntactical difference and they can be used freely within this library macros. [20] The following aesthetic conventions are followed in this documentation (because the authors think they improve readability):

1. Round parenthesis are spaced when a list of tokens is expected (e.g., the function parameter list): `( token1 )`, `( token1, token2 )`, `( token1, token2, token3 )`, etc.

2. Round parenthesis are not spaced when only a single token is expected (e.g., the function name): `(token)`.

[19] Note that the `auto` storage classifier in deprecated by C++11 so it should be used with the usual care when writing programs that should be portable from C++03 to C++11.
[20] An MSVC preprocessor bug requires to use at least one space or newline to separate a parameter name from its type even when the parameter type is wrapped within parenthesis.

Preconditions

Let's program the preconditions of `postinc` (see also `pre_only_postinc.cpp`):

```
CONTRACT_FUNCTION(
    int const (postinc) ( (int&) value )
        precondition( value < std::numeric_limits<int>::max() ) // Precondition.
        // postcondition:   value == oldof value + 1
        //                  result == oldof value
) {
    return value++;
}
```

In this example, the preconditions assert only one boolean condition but additional boolean conditions can be listed separated by commas `,`. Preconditions are automatically checked by the library when the `postinc` function is called, immediately before the body is executed (i.e., at function *entry*). See also the Free Function Calls section.

Contracts are supposed to check but not to modify the state of the program (see the Constant-Correctness section). Therefore, this library evaluates all contract conditions in constant-correct context. Specifically, in this example the type of `value` type is automatically promoted from `int&` to `int const&` within the preconditions and an attempt to modify `value` within the preconditions will correctly generate a compile-time error.

Postconditions (Result and Old Values)

Let's program the postconditions of `postinc` completing the contract (see also `post_also_postinc.cpp`):

```
CONTRACT_FUNCTION(
    int const (postinc) ( (int&) value )
        precondition( value < std::numeric_limits<int>::max() )
        postcondition(                              // Postconditions.
            auto result = return,                   // Result value.
            auto old_value = CONTRACT_OLDOF value,  // Old value(s).
            value == old_value + 1,                 // Assertion(s)...
            result == old_value
        )
) {
    return value++;
}
```

Postconditions are automatically checked by the library when the `postinc` function is called, immediately after the body is executed but only if the body does not throw an exception (i.e., at function *normal exit*). See also the Free Function Calls section.

Postconditions can access the function return value by declaring a variable of type `auto` [21] and assigning it to the `return` keyword (the variable name is arbitrary but `result` is often used).

Postconditions can also access the *old value* that an expression had just before the body was executed. This is done by declaring a variable of type `auto` and assigning it to `CONTRACT_OLDOF` *`expression`* (the variable name is arbitrary but the `old_` prefix is often used). Before executing the function body, the library automatically copies (once) the value of the specified expression into a variable with the specified name which is then made available to the postconditions. Therefore, the type of an old value expression must have a copy constructor (or, more precisely, it must be be `ConstantCopyConstructible`), otherwise the library will generate a compile-time error. The parenthesis around the expression passed to the `CONTRACT_OLDOF` macro are allowed but not required (in this example, `CONTRACT_OLDOF(value)` could have been equivalently used instead of `CONTRACT_OLDOF value` but the latter was preferred because of lesser parenthesis). In this sense, the syntax of the `CONTRACT_OLDOF` macro resembles the syntax of the `sizeof` operator which also allows but does not require parenthesis when applied to value expressions (i.e., `sizeof value` and `sizeof(value)` are both valid). [22] (The maximum number of supported and possible old value declarations is specified by the `CONTRACT_CONFIG_OLDOF_MAX` and `CONTRACT_LIMIT_OLDOFS` macros respectively.)

Postconditions always access result and old values in constant-correct context so that contracts cannot modify the state of the program (see the Constant-Correctness section).

In general, it is recommended to specify multiple assertions separated by commas and not group them together into a single condition using the operator `and` (or `&&`). This is because if assertion conditions are programmed together using `and` then it will not be clear which assertion condition actually failed in case the contract is broken. For example, if in the code above we programmed one single postcondition `(value == old_value + 1) and (result == old_value)` then in case the postcondition failed we would not know which condition failed `value == old_value + 1` or `result == old_value`.

[21] **Rationale.** The C++11 `auto` declaration syntax was adopted for postcondition return value declarations however this library knows the result type because it is specified in the function declaration within the macros so no type deduction is actually used to implement auto declarations of return values. Similarly, the C++11 `auto` declaration syntax was adopted for postcondition old value declarations however this library deduces the type of old value expressions using Boost.Typeof so no C++11 feature is actually needed (in this case programmers can optionally specify the old value type so to not use Boost.Typeof as explained in the Advanced Topics section).
[22] **Rationale.** This is also the syntax specified by [N1962] for the `oldof` operator which is the equivalent of the `CONTRACT_OLDOF` macro.

Note

All tokens must be specified in the fixed order listed in the Grammar section (it is not possible to specify postconditions before preconditions, `volatile` before `const` for member functions, etc). [23]

Classes and Class Invariants

Consider the following class `ivector` which is a vector of integers with a constructor, destructor, and the `push_back` member function (see also `no_contract_ivector.cpp`):

```
class ivector
{
    // invariant:   empty() == (size() == 0)
    //              size() <= capacity()
    //              capacity() <= max_size()
    //              std::distance(begin(), end()) == int(size())

    public: typedef std::vector<int>::size_type size_type;

    public: explicit ivector ( size_type count )
        // precondition:    count >= 0
        // postcondition:   size() == count
        //                  boost::algorithm::all_of_equal(begin(), end(), 0)
        : vector_(count)
    {}

    public: virtual ~ivector ( void ) {}

    public: void push_back ( int const value )
        // precondition:    size() < max_size()
        // postcondition:   size() == oldof size() + 1
        //                  capacity() >= oldof capacity()
        //                  back() == value
    {
        vector_.push_back(value);
    }

    private: std::vector<int> vector_;
};
```

Let's program the class declaration and the class invariants using the `CONTRACT_CLASS` and `CONTRACT_CLASS_INVARIANT` macros respectively (see also `class_ivector.cpp`):

```
CONTRACT_CLASS( // Declare the class.
    class (ivector)
) {
    CONTRACT_CLASS_INVARIANT( // Specify the class invariants.
        empty() == (size() == 0),
        size() <= capacity(),
        capacity() <= max_size(),
        std::distance(begin(), end()) == int(size())
    )

    public: typedef std::vector<int>::size_type size_type;
    // ...
```

A class must always be declared using the `CONTRACT_CLASS` macro in oder to later be able to program contracts for its constructors, destructor, and member functions. Similarly to function names, the class name must always be wrapped within parenthesis `(ivector)` (because the class name is not a known keyword and it is not the last element of the class declaration syntax as base classes and other elements can be specified). Only constructors, destructors, and member functions that are public check the class invariants while the ones that are either protected or private do not (see the Contract Programming Overview section). The class invariant must always be specified for a class declared using the `CONTRACT_CLASS` macro and it must appear at the very beginning of the class definition (no statement can precede the `CONTRACT_CLASS_INVARIANT` macro, not even a `typedef` or a `friend` declaration). If a class has no class invariant, the class invariants must be specified `void`:

[23] **Rationale.** This library macros could have been implemented to allow to mix the order of some tokens (preconditions and postconditions, `volatile` and `const`). However, that would have complicated the macro implementation without any added functionality for the user.

```
CONTRACT_CLASS(
    class (ivector)
) {
    CONTRACT_CLASS_INVARIANT( void ) // No class invariant.

    // ...
```

Using the same macro, this library also allows to specify contracts for `struct`: [24]

```
CONTRACT_CLASS(
    struct (ivector) // Structs with contracts.
) {
    // ...
```

The usual differences between `struct` and `class` also apply when this library is used (i.e., default member and inheritance access levels are `public` for `struct` and `private` for `class`).

Note

Unfortunately, this library does not allow to specify contracts for `union`. [25]

No contracts are checked (not event the class invariants) when a data member is accessed directly so it might be best for both classes and structs to have no non-constant public data member (but access all data members via appropriate member functions that can check the class and sturct contracts).

Constructors

Let's program the `ivector` class constructor and its contract using the `CONTRACT_CONSTRUCTOR` macro (see also `class_ivector.cpp`):

```
CONTRACT_CONSTRUCTOR( // Declare the constructor and its contract.
    public explicit (ivector) ( (size_type) count )
        precondition( count >= 0 )
        postcondition(
            size() == count,
            boost::algorithm::all_of_equal(begin(), end(), 0)
        )
        initialize( vector_(count) )
) {}
```

Constructor, destructor, and member function declarations must always repeat their access level `public`, `protected`, or `private` [26] which is specified without the usual trailing column symbol `:`. (This use of `public`, `protected`, and `private` resembles Java's syntax.)

The constructor name `(ivector)` and the non-fundamental parameter type `(size_type)` must be wrapped within parenthesis (because they are not known keywords and they are not at the end of a syntactic element).

Member initializers are programed as a comma-separated list but using the specifier `initialize( expression1, expression2, ... )` instead of the usual column symbol `: expression1, expression2, ...`. When specified, they must appear at the very end of constructor declarations (after both preconditions and postcondition if present). [27] Unfortunately, when member initializers are specified, the constructor body must be defined together with its declaration and contract (see the Forward Declarations and Body Definitions section).

There is no object before the constructor body is executed, therefore this library will generate a compile-time error if either preconditions or postcondition old value expressions try to access the object `this`. For the same reason, only static class invariants are checked at entry of public constructors (non-static class invariants are checked at public constructor normal exit but not at entry). Finally, preconditions are checked before member initializers are executed (so the initializers can rely on preconditions for validating constructor arguments). See also the Constructor Calls section.

[24] **Rationale.** There is no need for a `CONTRACT_STRUCT` macro because this library macros can parse the class declaration and distinguish between the `struct` and `class` specifiers.
[25] **Rationale.** The authors have not fully investigated if this library could be extended to specify contracts for unions. It is possible that future revisions of this library will support contracts for unions (see also Ticket 50).
[26] **Rationale** This library needs to know each member access level because in Contract Programming only public members are supposed to check class invariants while protected and private members only check preconditions and postconditions but not the class invariants. Unfortunately, in C++ it is not possible for a member to introspect its access level using template meta-programming. Therefore, this library requires the access level to be specified within the macros and it uses preprocessor meta-programming to detect it. Furthermore, given that the access level is has to be specified, the `CONTRACT_FUNCTION` macro also uses it to differentiate between free and member functions so no additional macro `CONTRACT_MEMBER` is needed.
[27] **Rationale.** Member initializers are specified after the preconditions and postconditions because they are part of the constructor definition while preconditions and postconditions are part of the constructor declaration.

Destructors

Let's program the `ivector` class destructor using the `CONTRACT_DESTRUCTOR` macro (see also `class_ivector.cpp`):

```
CONTRACT_DESTRUCTOR( // Declare the destructor (so it checks invariants).
    public virtual (~ivector) ( void )
) {}
```

The destructor access level `public`, `protected`, or `private` must always be specified. The destructor name `(~ivector)` must be wrapped within parenthesis. The destructor empty parameter list must be specified using `void` (same as for constructors, member functions, and free functions with no parameters).

Destructors have no parameter and there is no object after the destructor body is executed. Therefore, destructors cannot specify neither preconditions nor postconditions. [28] However, it is still necessary to ontract (at least public) destructors so they check static and non-static class invariants at entry, and non-static class invariants at exit. See also the Destructor Calls section.

Important

Only members declared using this library macros check the class invariants. Therefore, at least all public members of a class with non-empty class invariants should be declared using this library macros even when they have no preconditions and no postconditions so that they check the class invariants (that is the case for public destructors).

In general, there is no need to use this library macros to declare free functions, constructors, destructors, and member function only when:

1. A free function has no preconditions and no postconditions.
2. A private or protected constructor, destructor, or member function has no preconditions and no postconditions (regardless of class invariants because private and protected members never check class invariants).
3. A public constructor, destructor, or member function has no preconditions, no postconditions, and its class has no class invariants.

Member Functions

Let's program the member function `ivector::push_back` and its contract using the `CONTRACT_FUNCTION` macro (see also `class_ivector.cpp`):

```
CONTRACT_FUNCTION( // Declare the member function with its contract.
    public void (push_back) ( int const value )
        precondition( size() < max_size() )
        postcondition(
            auto old_size = CONTRACT_OLDOF size(),
            auto old_capacity = CONTRACT_OLDOF capacity(),
            size() == old_size + 1,
            capacity() >= old_capacity,
            back() == value
        )
) {
    vector_.push_back(value);
}
```

The member function access level `public`, `protected`, or `private` must always be specified. The member function name `(push_back)` must be wrapped within parenthesis. The member function result type `void` and parameter type `int const` can but do not have to be wrapped within parenthesis (because they are both fundamental types containing no symbol).

Public non-static member functions check static and non-static class invariants at both entry and exit. Preconditions are checked at entry and postconditions are checked at normal exit. Both preconditions and postconditions can access the object and the member function parameters in constant-correct context. Postconditions can declare return and old value variables (again, in contract-correct context). See also the Member Function Calls section.

[28] Future revisions of this library might implement static postconditions for destructors (see the Destructor Calls section).

Note

Contracts are most useful when the assertions only use public members that are accessible from the caller so the caller can properly verify and use the contract. In particular, preconditions of a member function or constructor that use non-public members are essentially incorrect because they cannot be verified by the caller before calling the function (in fact, Eiffel generates a compile-time error in this case).

This library leave it up to programmers to only use public members when programming contracts and especially when programming preconditions (see the Specification vs. Implementation section).

Inheritance and Subcontracting

Consider the following class `unique_identifiers` which is a collection of unique integral identifiers (see also `subcontract_identifiers.cpp`):

```
CONTRACT_CLASS(
    class (unique_identifiers)
) {
    CONTRACT_CLASS_INVARIANT( size() >= 0 )

    CONTRACT_CONSTRUCTOR(
        public (unique_identifiers) ( void )
            postcondition( size() == 0 )
    ) {}

    CONTRACT_DESTRUCTOR(
        public virtual (~unique_identifiers) ( void )
    ) {}

    CONTRACT_FUNCTION(
        public virtual void (add) ( int id )
            precondition(
                // Id not already present.
                std::find(begin(), end(), id) == end()
            )
            postcondition(
                auto old_size = CONTRACT_OLDOF size(),
                auto old_found = CONTRACT_OLDOF
                        std::find(begin(), end(), id) != end(),
                // If id was not already present, it was added now...
                old_found ? true : std::find(begin(), end(), id) != end(),
                // ...and size was increased of 1.
                old_found ? true : size() == old_size + 1
            )
    ) {
        identifiers_.push_back(id);
    }

    // ...
```

Reading the contracts we can understand the semantics of the class operation (even if we do not consult the class implementation):

1. The collection is constructed with zero size as specified by the constructor postconditions. Consistently, the class invariants assert that the class size must never be negative (but it can be zero).

2. The member function `unique_identifiers::add` allows to add an integral identifier to the collection. The preconditions assert that the specified identifier must not already be part of the collection making sure that duplicate identifiers are never added.

3. If the specified identifier was not already part of the collection (`old_found` is `false`) then the postconditions assert that the identifier is added and consequently that the size of the collection increased by one. Note that both postconditions are trivially `true` if instead the identifier was already part of the collection (`old_found` is `true`). We call these type of assertions that are "guarded" by an activation condition *select assertions* (here we have used the C++ ternary operator `?:` to program select assertions, the Advanced Topics section will introduce an alternative construct for select assertions).

Let's assume that we want to derive from `unique_identifiers` another type of collection `duplicate_identifiers` which instead allows to add duplicate identifiers (see also `subcontract_identifiers.cpp`):

```
CONTRACT_CLASS(
    class (duplicate_identifiers)
        extends( public unique_identifiers ) // Automatically subcontract.
) {
    CONTRACT_CLASS_INVARIANT(
        size() >= 1 // Strengthened inherited class invariants (in `and`).
    )

    CONTRACT_CONSTRUCTOR(
        public (duplicate_identifiers) ( int id )
            postcondition( size() == 1 )
    ) {
        // As usual, constructor should not call virtual member `add`.
        identifiers_.push_back(id);
    }

    CONTRACT_DESTRUCTOR(
        public virtual (~duplicate_identifiers) ( void )
    ) {}

    CONTRACT_FUNCTION(
        public virtual void (add) ( int id )
            precondition( // Wakened inherited preconditions (in `or`).
                // OK even if id is already present.
                std::find(begin(), end(), id) != end()
            )
            postcondition( // Strengthened inherited postconditions (in `and`).
                auto old_size = CONTRACT_OLDOF size(),
                auto old_found = CONTRACT_OLDOF
                        std::find(begin(), end(), id) != end(),
                // Inherited postconditions not checked because of
                // select assertions, plus size unchanged if already present.
                old_found ? size() == old_size : true
            )
    ) {
        if(std::find(begin(), end(), id) == end()) { // Not already present.
            // Unfortunately, must invoke base function via `BODY` macro.
            unique_identifiers::CONTRACT_MEMBER_BODY(add)(id);
        }
    }
};
```

The derived class `duplicate_identifiers` publicly inherits from the base class `unique_identifiers` using the specifier `extends( base1, base2, ... )` instead of the usual column symbol `: base1, base2, ...`. (This use of `extends` resembles Java's syntax.) As usual, the inheritance access level `public`, `protected`, or `private` is optional and it defaults to `public` for `struct` and to `private` for `class`. Virtual base classes can also be specified but `virtual` must always be specified after the inheritance access level `public`, `protected`, or `private` if present. Multiple base classes (`base1`, `base2`, etc) can be specified for multiple-inheritance (the configuration macro `CONTRACT_CONFIG_INHERITANCE_MAX` indicates the maximum number of base classes that can be specified).

Note

This library automatically subcontracts the derived class when one or more base classes are specified (see the Contract Programming Overview section).

Let's assume that a `duplicate_identifers` object is constructed with identifier `123` and that we try to add the identifier `123` again:

```
duplicate_identifiers ids(123);
ids.add(123);
```

Reading both the derived and base class contracts, we can understand the semantics of the derived class operation (even if we do not consult the base and derived class implementations):

1. The `duplicate_identifiers` constructor creates a collection with size one as specified by the constructor postconditions. Therefore, the `duplicate_identifiers` class invariants assert that size is always positive. Subcontracting automatically checks the `unique_identifiers` base class invariants in logic-and with the `duplicate_identifiers` derived class invariants thus class invariants can only be strengthened by derived classes (this is necessary because the derived class can be used whenever the base class is used therefore the derived class must satisfy the base class invariants at all times, see also the substitution principle and the Contract Programming Overview section).

2. Subcontracting automatically checks the `unique_identifiers::add` overridden preconditions in logic-or with the `duplicate_identifiers::add` overriding preconditions thus preconditions can only be weakened by overriding member functions (this is necessary because the overriding member function can be called under the conditions for which the overridden function is called, see also the substitution principle and the Member Function Calls section). Indeed, in this example the `unique_identifiers::add` preconditions fail (because `123` is already in the collection) but the `derived_identifiers::add` preconditions pass so the logic-or of the two sets of preconditions passes and the call `ids.add(123)` is a valid call.

3. Subcontracting automatically checks the `unique_identifiers::add` overridden postconditions in logic-and with the `duplicate_identifiers::add` overriding postconditions thus postconditions can only be strengthen by overriding member functions (this is necessary because the overriding function can be called in any context in which the overridden function is called, see also the substitution principle and the Member Function Calls section). Indeed, in this example the `unique_identifiers::add` postconditions pass because the identifier `123` was already added (`old_found` is `true`) and `unique_identifiers::add` select assertions trivially return `true`. The `duplicate_identifiers::add` postconditions are then checked in logic-and and they correctly assert that the collection size did not change given that `123` was already added. Therefore, the call `ids.add(123)` is valid but adds no identifier.

Finally, note that within the body of overriding functions it is necessary to use the `CONTRACT_MEMBER_BODY` macro to call the overridden function as in `unique_identifiers::CONTRACT_MEMBER_BODY(add)(id)`.

Warning

Unfortunately, invoking the overridden function from within the overriding function body without using `CONTRACT_MEMBER_BODY` results in infinite recursion. [29] This limitation might be removed in future revisions of this library (see also Ticket 46).

Class Templates

Let's program a class template `vector` similar to the class `ivector` we have seen before but that works on a generic value type `T` and not just on `int` (see also `class_template_vector.cpp` and `pushable.hpp`):

[29] **Rationale**. In this case, the library implementation will recursively check contracts of the overriding function forever (in all other cases, this library is able to avoid infinite recursions due to contract checking).

```
#include "pushable.hpp"

CONTRACT_CLASS(
    template( typename T )
    class (vector) extends( public pushable<T> ) // Subcontract.
) {
    // Within templates, use contract macros postfixed by `_TPL`.

    CONTRACT_CLASS_INVARIANT_TPL(
        empty() == (size() == 0),
        size() <= capacity(),
        capacity() <= max_size(),
        std::distance(begin(), end()) == int(size())
    )

    public: typedef typename std::vector<T>::size_type size_type;
    public: typedef typename std::vector<T>::const_reference const_reference;
    public: typedef typename std::vector<T>::const_iterator const_iterator;

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( (size_type) count )
            precondition( count >= 0 )
            postcondition(
                size() == count,
                boost::algorithm::all_of_equal(begin(), end(), T())
            )
            initialize( vector_(count) )
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~vector) ( void )
    ) {}

    CONTRACT_FUNCTION_TPL(
        public void (push_back) ( (T const&) value )
            precondition( size() < max_size() )
            postcondition(
                auto old_size = CONTRACT_OLDOF size(),
                auto old_capacity = CONTRACT_OLDOF capacity(),
                size() == old_size + 1,
                capacity() >= old_capacity
                // Overridden postconditions also check `back() == value`.
            )
    ) {
        vector_.push_back(value);
    }

    // ...
```

(The specifications of the code in this example are not fully programmed, in particular concepts, see the Concepts section, and assertion requirements, see the Advanced Topics section, have been omitted for simplicity.)

Note that templates are declared using the specifier `template( typename T, ... )` which uses round parenthesis instead of the usual angular parenthesis `template< typename T, ... >`. However, after a template is declared using round parenthesis, angular parenthesis are used as usual to instantiate the template (e.g., the base class `pushable<T>`).

Important

Within a type-dependent scope (e.g., within the class template of the above example) the special macros with the `_TPL` postfix must be used instead of the macros we have seen so far. [30] (The same applies to function templates, see below.)

[30] **Rationale.** The library must know if the enclosing scope is a template so it knows when to prefix nested type expressions with `typename` (because C++03 does not allow to use `typename` outside templates). This constraints could be relaxed on future revisions of this library for C++11 compilers (because C++11 allows to use `typename` more freely). Earlier versions of this library did not require to use the special `_TPL` macros within templates because the library internally implemented every contracted function using a template function, possibly with dummy template parameters, even if the original function was not a template so `typename` could always be used by the library. The dummy template parameters were hidden to the user so this approach did not change the user API and had the benefit of not requiring the `_TPL` macros. However, this approach increased compilation time because of the extra templates that it introduced so the current revision of the library uses the `_TPL` macros.

In the example above, the enclosing class template `vector` is not declared using the `CONTRACT_CLASS_TPL` macro because its enclosing scope is not type-dependent (it is the global namespace). Within such a class template however, all contract macros must use the `_TPL` postfix so `CONTRACT_CLASS_INVARIANT_TPL`, `CONTRACT_CONSTRUCTOR_TPL`, `CONTRACT_DESTRUCTOR_TPL`, and `CONTRACT_FUNCTION_TPL` must be used to specify class invariants, declare constructors, declare the destructor, and declare member functions respectively. In general, the `CONTRACT_CLASS_TPL` macro is used when declaring a class nested within a class template (see the Advanced Topics section).

This libraries supports all the different kinds of template parameters that C++ supports: type template parameters, value template parameters, and template template parameters.

- Type template parameters can be prefixed by either `typename` or `class` as usual, `typename A` or `class C`.
- The type of value template parameters must be wrapped within parenthesis `(A) D` but the parenthesis are optional for fundamental types that contain no symbol `int B`.
- Template template parameters use round parenthesis for their template specifier but then their inner template parameters are specified using the usual C++ syntax `template( typename X, template< ... > class Y, ... ) class E`.

Default template parameters are specified using `, default` *`default-value`* right after the template parameter declaration. For example (see also `template_params.cpp`):

```
CONTRACT_CLASS( // Class template parameters.
    template(
          typename A            // Type template parameter.
        , int B                 // Value template parameter.
        , class C, default A    // Optional type template parameter.
        , (A) D, default B      // Optional value template parameter.
        , template(             // Template template parameter: Outer template
              typename X        // uses `()` but rest uses usual syntax.
            , template< typename S, typename T = S > class Y
            , class Z = int
            , Z V = 0
          ) class E, default x  // Optional template template parameter.
    )
    struct (a)
) {
    // ...
```

Finally, the syntax `template( void )` is used instead of the usual `template< >` to specify a template with no template parameters (this is typically used for template specializations, see the Advanced Topics section).

Function Templates

Let's program a function template `postinc` similar to the function `postinc` that we have seen before but that works on a generic type `T` and not just on `int` (see also `function_template_postinc.cpp`):

```
CONTRACT_FUNCTION(
    template( typename T ) // Template parameter(s).
    (T const) (postinc) ( (T&) value )
        precondition( value < std::numeric_limits<T>::max() )
        postcondition(
            auto result = return,
            auto old_value = CONTRACT_OLDOF value,
            value == old_value + 1,
            result == old_value
        )
) {
    return value++;
}
```

(The specifications of the code in this example are not fully programmed, in particular concepts, see the Concepts section, and assertion requirements, see the Advanced Topics section, have been omitted for simplicity.)

Type, value, and template template parameters are specified for function templates using the same syntax used for class template parameters, for example (see also `template_params.cpp`):

```
CONTRACT_FUNCTION( // Function template parameters.
    template( // As usual, no default template parameters allowed in functions.
          typename A            // Type template parameter.
        , int B                 // Value template parameter.
        , class C               // Type template parameter.
        , (A) D                 // Value template parameter.
        , template(             // Template template parameter: Outer template
              typename X        // uses `()` but rest uses usual syntax.
            , template< typename S, typename T = S > class Y
            , class Z = int
            , Z V = 0
          ) class E
    )
    void (f) ( void )
) {
    // ...
```

Note that as usual, function templates cannot have default template parameters in C++03.

Forward Declarations and Body Definitions

This library supports contracts for classes and functions that have forward declarations. Furthermore, the function body definition can be deferred and separated from the function declaration and its contracts. For example (see also `body_natural.hpp`, `body_natural_impl.hpp`, and `body_natural.cpp`):

```
template< typename T, T Default = T() > // Class forward declaration.
class natural ;

template< typename T > // Function forward declaration.
bool less ( natural<T> const& left, natural<T> const& right ) ;

CONTRACT_FUNCTION(
    template( typename T )
    bool (greater) ( (natural<T> const&) left, (natural<T> const&) right )
        postcondition(
            auto result = return,
            result ? not less(left, right) : true
        )
) ; // Deferred free function body definition, use `;`.

CONTRACT_CLASS( // Class declaration (with contracts).
    template( typename T, (T) Default )
    class (natural)
) {
    CONTRACT_CLASS_INVARIANT_TPL( get() >= 0 )

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (natural) ( (T const&) value, default Default )
            precondition( value >= 0 )
            postcondition( get() == value )
            // Unfortunately, no member initializers when body deferred.
    ) ; // Deferred constructor body definition, use `;`.

    CONTRACT_DESTRUCTOR_TPL(
        public (~natural) ( void )
    ) ; // Deferred destructor body definition, use `;`.

    CONTRACT_FUNCTION_TPL(
        public bool (equal) ( (natural const&) right ) const
            postcondition(
                auto result = return,
                result == not less(*this, right) && not greater(*this, right)
            )
    ) ; // Deferred member function body definition, use `;`.

    CONTRACT_FUNCTION_TPL(
        public (T) (get) ( void ) const
    ) {
        return value_;
    }

    private: T value_;
};

CONTRACT_FUNCTION( // Function declaration (with contracts).
    template( typename T )
    bool (less) ( (natural<T> const&) left, (natural<T> const&) right )
        postcondition(
            auto result = return,
            result ? not greater(left, right) : true
        )
) {
    return left.get() < right.get();
}
```

In this example, the class `natural` and the function `less` are first forward declared and later declared. Note that the contracts can be specified only once and together with the class or function declaration (not with the forward declarations that can be many).

Important

This library forces to specify contracts with the function and class declarations and not with their definitions. This is correct because contracts are part of the program specifications (declarations) and not of the program implementation (definitions), see also the Specification vs. Implementation section.

Note that as usual in C++, not all functions need to be defined when they are first declared. In this example, `less` and `natural::get` are defined together with their declarations and contracts while the other function bodies are defined separately.

The definitions of the free function `greater`, the constructor `natural::natural`, the destructor `natural::~natural`, and the member function `natural::equal` are deferred and separated from their declarations and contracts. In fact, a semicolon `;` is specified right after the contract declaration macros instead of programming the function bodies. As usual in C++, the semicolon `;` indicates that the function body definition is deferred and it will appear at some other point in the program (often in a `.cpp` file that can be compiled separately). In this example, the deferred functions are templates or template members therefore the definitions where programmed in a header file that must be made available when the template declarations are compiled (see also `body_natural_impl.hpp`): [31]

```
// Deferred body definitions, separated from their declarations and contracts.

template< typename T >
bool CONTRACT_FREE_BODY(greater) (
        natural<T> const& left, natural<T> const& right )
{
    return left.get() > right.get();
}

template< typename T, T Default >
CONTRACT_CONSTRUCTOR_BODY((natural<T, Default>), natural) ( T const& value )
{
    value_ = value;
}

template< typename T, T Default >
CONTRACT_DESTRUCTOR_BODY((natural<T, Default>), ~natural) ( void )
{
    // Do nothing.
}

template< typename T, T Default >
bool natural<T, Default>::CONTRACT_MEMBER_BODY(equal) ( natural const& right )
        const
{
    return not less(*this, right) && not greater(*this, right);
}
```

The usual C++ syntax is used to define the bodies (in fact this library only alters the syntax of class and function declarations but not the syntax of their definitions). However, free function, constructor, destructor, and member function names must be specified using the macros `CONTRACT_FREE_BODY`, `CONTRACT_CONSTRUCTOR_BODY`, `CONTRACT_DESTRUCTOR_BODY`, and `CONTRACT_MEMBER_BODY` respectively. [32] The free and member body macros take one parameter which is the function name. The constructor and destructor body macros take two parameters that are the fully qualified class type (including eventual template parameters as usual) and the class name which must be prefixed by the tilde symbol ~ for destructors as usual. [33] The class type parameter must be wrapped within extra parenthesis like in this example but only if it contains multiple template parameters (because these use unwrapped commas which cannot be passed within macro parameters, see the beginning of the Advanced Topics section).

In this example, it was possible to separate the constructor body definition because the constructor did not specify member initializers.

[31] In principle, this library supports `export` templates that can be used to program template definitions separately from their declarations and in files that can be pre-compiled. However, `export` templates are not supported by most C++03 compilers (in fact, Comeau might be the only compiler that supports them), they were removed from C++11, and they are an untested feature of this library. Instead of using `export` templates, it is common practise in C++ to program both template declarations and definitions together so they can be compiled correctly. In this example, the authors have used a separate header file `..._impl.hpp` to logically separate template declarations and definitions but the header is included at the bottom of the declaration header file so the definitions are always available together with their declarations.

[32] **Rationale.** Four different body macros are needed because contracts are disabled differently for the different type of functions. For example, disabling preconditions and postconditions turns off contracts for free functions but not for member functions, disabling class invariants turns off contracts for destructors but not for constructors, etc.

[33] Some compilers accept to repeat template parameter names in the constructor and destructor names. However, this is not C++03 compliant (and, for example, it was fixed in more recent versions of GCC) therefore programmers are advised to specify the template parameter names only for the class type and not for the constructor and destructor names (this is also why the macros must take a second parameter with the class name instead of just the class type parameter).

Warning

Unfortunately, when member initializers are specified, the macro `CONTRACT_CONSTRUCTOR_BODY` cannot be used, the constructor body definition cannot be deferred, and the constructor body must be defined together with the constructor declaration and its contract inside the class definition. [34]

[34] **Rationale.** This limitation comes from the fact that C++03 does not support delegating constructors. If member initializers are specified within the contract, the deferred body will not compile when contract compilation is turned off by the configuration macros. If instead member initializers are specified with the deferred body definition, the deferred body will not compile when contract compilation is turned on by the configuration macros. There is no way to reconcile these two conditions without delegating constructors, so definitions cannot be deferred for constructors that specify member initializers. This limitation could be removed in future revisions of this library for C++11 compilers that support delegating constructors (see also Ticket 51.

Advanced Topics

This section explains advanced usages of this library in programming contracts. See the Grammar section for a complete guide on this library syntax.

Commas and Leading Symbols in Macros

C++ macros cannot correctly parse a macro parameter if it contains a comma `,` that is not wrapped within round parenthesis `()` (because the preprocessor interprets such a comma as separation between two distinct macro parameters instead that as part of one single macro parameter, see also Boost.Utility/IdentityType). Furthermore, syntactic elements specified to this library macros cannot start with a non-alphanumeric symbol (`-1`, `1.23`, `"abc"`, `::`, etc). [35] For example, the parameter type `::std::pair<OtherKey, OtherT> const&` cannot be passed to this library for two reasons: It starts with the non-alphanumeric symbol `::` and it contains the comma `OtherKey, OtherT` which is not wrapped by round parenthesis.

The Boost.Utility/IdentityType macro `BOOST_IDENTITY_TYPE` can be used as usual to overcome this issue. However, using the `BOOST_IDENTITY_TYPE` macro presents two limitations: It makes the syntax of this library macros more cumbersome and, more importantly, it does not allow C++ to automatically deduce function template parameters (see Boost.Utility/IdentityType for more information). Therefore, the syntax of this library provides an alternative to `BOOST_IDENTITY_TYPE` to handle commas and leading symbols within macro parameters:

1. Commas and leading symbols can be used freely within elements of the syntax that already require wrapping parenthesis (e.g., non-fundamental parameter types (`::std::pair<OtherKey, OtherT> const&`)).
2. Extra parenthesis can always be used to wrap elements of the syntax that might contain commas and leading symbols (e.g, the base class type `public (::sizeable<Key, T>)`).
3. Extra parenthesis can always be used to wrap value expressions so they can contain commas and leading symbols (e.g., a class invariants assertion (`::sizeable<Key, T>::max_size >= size()`)).

For example (see also `macro_commas_symbols_integral_map.cpp`):

[35] Note that for the preprocessor a number with decimal period `1.23`, `0.12`, `.34` is considered a symbol (because its concatenation will not result in a valid macro identifier).

```
CONTRACT_CLASS(
    template(
        typename Key,
        typename T, // Commas in following template params.
        class Allocator,
                default (::std::allocator<std::pair<Key const, T> >),
        (typename ::std::map<int, T>::key_type) default_key,
                default (-1)
    ) requires( (boost::Convertible<Key, int>) ) // Commas in concepts.
    class (integral_map)
        extends( public (::sizeable<Key, T>) ) // Commas in bases.
) {
    CONTRACT_CLASS_INVARIANT_TPL( // Commas in class invariants.
        (::sizeable<Key, T>::max_size) >= size()
    )

    public: typedef typename std::map<Key, T, std::less<Key>, Allocator>::
            iterator iterator;

    CONTRACT_FUNCTION_TPL(
        public template( typename OtherKey, typename OtherT )
            requires(
                (::boost::Convertible<OtherKey, Key>),
                (::boost::Convertible<OtherT, T>)
            )
        (::std::pair<iterator, bool>) (insert) ( // Commas in result and params.
                (::std::pair<OtherKey, OtherT> const&) other_value,
                        default (::std::pair<Key, T>(default_key, T()))
            ) throw( (::std::pair<Key, T>) ) // Commas in exception specs.
            precondition( // Leading symbols in preconditions (same for commas).
                ((!full())) // LIMITATION: Two sets `((...))` (otherwise seq.).
            )
            postcondition( // Commas in postconditions.
                auto result = return,
                (typename sizeable<OtherKey, OtherT>::size_type)
                        old_other_size = CONTRACT_OLDOF
                                (size<OtherKey, OtherT>()),
                (::sizeable<Key, T>::max_size) >= size(),
                size() == old_other_size + (result.second ? 1 : 0)
            )
    ) {
        // ...
```

(Concepts and other advanced constructs used in this example are explained in the rest of this section and in later sections of this documentation.)

All these extra wrapping parenthesis are optional when there is no unwrapped comma and no leading symbol. Programmers could chose to always program the extra parenthesis for consistency but it is the authors' opinion that the syntax is harder to read with the extra parenthesis so it is recommended to use them only when strictly necessary.

It is also recommended to avoid using commas and leading symbols whenever possible so to limit the cases when it is necessary to use the extra wrapping parenthesis. For example, in many cases the leading symbol `::` might not be necessary and leading symbols like `!` can be replaced by the alternative `not` keyword. Furthermore, in some cases `typedef` can be programmed just before class and function declarations to avoid passing types within multiple template parameters separated by commas. Declarations of class templates and function templates are the most common cases were commas cannot be avoided and extra wrapping parenthesis are necessarily used.

Warning

Preconditions, postconditions, and class invariants composed of one single assertion that needs to be wrapped within extra parenthesis need to use double parenthesis (this should be a relatively rare case). [36] This is the case in the above example for `precondition( ((!full())) )` (but note that these extra parenthesis could have been avoided all together simply using `not` instead of `!` as in the more readable `precondition( not full() )`).

[36] **Rationale.** This limitation is introduced by the need to support also the sequence syntax for preprocessor without variadic macros (see the No Variadic Macros section).

Static Assertions

This library allows to use *static assertions* to program preconditions, postconditions, class invariants, and block invariants so that they are checked at compile-time instead of at run-time. The C++11 syntax is used (but Boost.MPL `BOOST_MPL_ASSERT_MSG` is used to implement static assertions so no C++11 feature is required):

```
static_assert(constant-boolean-condition, constant-string-literal)
```

For example (see also `static_assertion_memcopy.cpp`):

```
CONTRACT_FUNCTION(
    template( typename To, typename From )
    (To*) (memcopy) ( (To*) to, (From*) from )
        precondition( // Static assertions.
            static_assert(sizeof(To) >= sizeof(From), "destination too small"),
            // Wrapping parenthesis for asserted condition with commas.
            static_assert((boost::is_convertible<From, To>::value),
                    "incompatible types"),
            to, // pointer not null
            from // pointer not null
        )
) ;
```

Static assertions are always checked at compile-time (regardless of them appearing in preconditions, postconditions, etc). However, static assertions can be selectively disabled depending on where they are specified using `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, etc so it is still important to logically associate them with preconditions, postconditions, etc.

The string message passed as the second parameter of `static_assert` is ignored by the current implementation of this library (but it could be used by future revisions of this library that take advantage of C++11 features). It is recommended to always specify meaningful messages for static assertions to increase the readability and documentation of contracts. Similarly, it might be useful to put a short code comment following each assertions (`// pointer not null`, etc) to increase contract readability and documentation. [37]

In case of a static assertion failure, this library will generated a compile-time error containing text similar to the following:

```
static_assertion_memcopy.cpp:18 ... ERROR_statically_checked_precondition_number_1_failed_at_line_18 ...
```

This message is similar to the run-time assertion errors generated by this library, note how it contains all the information to uniquely identify the assertion that failed.

Constant Assertions

As explained in the Constant-Correctness section, contract assertions shall not change the state of the program because contracts are only supposed to check (and not alter) the state of the program. This library automatically makes member functions, function parameters, function result, and old values constant so the compiler will correctly generate an error if assertions mistakenly try to change the object, the function parameters, the function result, or the old values (this should be sufficient in most cases when programming contracts).

Important

This library cannot automatically make constant other variables that might be accessible by the contract assertions (e.g., global and static variables). [38]

This library provides *constant assertions* that can be used by programmers to explicitly make constant the variables used by the asserted condition:

```
const( variable1, variable2, ...) boolean-expression-using-variable1-variable2-...
```

The following example calculates the next even and odd numbers that are stored (for some reason) in a global variable and static data member respectively (see also `const_assertion_number.cpp`):

[37] **Rationale.** Eiffel supports *assertion labeling* to further documenting assertions. However, when designing this library for C++, the authors judged that it is sufficient to add a short code comment after each assertion to achieve a similar effect.

[38] **Rationale.** The library needs to know at least the variable name in order to make it constant. There is no way for this library to know the name of a global variable that is implicitly accessible from the contract assertion scope so such a variable cannot be automatically made constant. Non-static data members are automatically made constant by making constant the member function that checks the contract, but that does not apply to static data members. Not even C++11 lambda implicit captures could be used in this context because they make variables constant only when captured by value and that introduces a `CopyConstructible` requirement of the captured variable types.

```
unsigned even = 0;

CONTRACT_CLASS(
    class (number)
) {
    CONTRACT_CLASS_INVARIANT( void )

    public: static unsigned odd;

    CONTRACT_FUNCTION(
        public void (next) ( void )
            postcondition(
                auto old_even = CONTRACT_OLDOF even,
                auto old_odd = CONTRACT_OLDOF odd,
                // `[old_]even` and `[old_]odd` all `const&` within assertions.
                const( even, old_even ) even == old_even + 2,
                const( odd, old_odd ) odd == old_odd + 2
            )
    ) {
        even += 2;
        odd += 2;
    }
};

unsigned number::odd = 1;
```

Note know the postconditions use constant assertions to force the `even` and `odd` variables to be constant within each boolean expression that evaluates the assertion. If for example the assignment operator `=` were mistakenly used instead of the equality operator `==` in the above postconditions, the compiler would correctly generate an error.

Select Assertions

In the Tutorial section we have used the ternary operator `:?` to program assertions that are guarded by a boolean condition:

```
old_found ? true : std::find(begin(), end(), id) != end(),
old_found ? true : size() == old_size + 1
```

However, in cases like this one when the same boolean condition guards multiple assertions, it might be more efficient to evaluate the guard condition only once using one *select assertion* instead of multiple ternary operators `:?`. In addition, some programmers might find the select assertion syntax more readable than the ternary operator `:?`. For example, the above guarded assertions could have been programmed using select assertions as:

```
if(not old_found) ( // Guard condition evaluated only once.
    std::find(begin(), end(), id) != end(),
    size() == old_size + 1
)
```

Select assertion allow to specify an optional else-block:

```
if(boolean-condition) (   // Round parenthesis `()`.
    ...                   // Comma-separated assertions.
) else (                  // Else optional as usual.
    ...                   // Comma-separated assertions.
)
```

Note that round parenthesis `( ... )` are used to program the then-block and else-block instead of the usual C++ curly parenthesis `{ ... }`: Select assertions can be nested into one another (the `CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS` macro specifies the maximum nesting level for select assertions).

For example, consider the postconditions of the following function which calculates the factorial of a natural number (see also `select_assertion_factorial.cpp`):

```
CONTRACT_FUNCTION(
    int (factorial) ( int n )
        precondition( n >= 0 )
        postcondition(
            auto result = return,
            result >= 1,
            if(n == 0 || n == 1) ( // Select assertion, if-then statement.
                result == 1
            ) else ( // Select assertion, else statement (optional).
                // Assertions disabled within assertion so recursion OK.
                result == n * factorial(n - 1)
            )
        )
) {
    if(n == 0 || n == 1) return 1;
    else return n * factorial(n - 1);
}
```

Using the same syntax used for constant assertions, it is possible to force all variables (global, static, etc) used to evaluate the if-condition of the select assertion to be constant (see also `const_select_assertion_factorial.cpp`):

```
int n = 0;

CONTRACT_FUNCTION(
    int (factorial) ( void )
        precondition( const( n ) n >= 0 )
        postcondition(
            auto result = return,
            if(const( n ) n == -1 || n == 0) ( // Constant-correct if-condition.
                result == 1
            ) else (
                result >= 1
            )
        )
) {
    int m = n;
    if(m == 0 || m == 1) {
        return 1;
    } else {
        --n;
        return m * factorial();
    }
}
```

Assertion Statements

Only assertions can be programmed within the contracts while normal C++ statements are not allowed. This is keeps contracts simple making programming errors within the contracts less likely and therefore increasing the probably that error-free contracts can properly check the correctness of the implementation.

However, `using` directives, `namespace` aliases, and `typedef` statements are allowed within the contracts because they only affect compilation (not altering the state of the program at run-time) and they can be used to write more readable contracts (for example, shortening namespaces within contract assertions). When used, these statements have affect only locally within the preconditions, postconditions, class invariants, etc where they appear. [39]

For example (see also `assertion_statement_ialloc.cpp`):

[39] Assertion statements might be especially useful because contracts appear with the class and function declarations usually in header files where `using` directives and `namespace` aliases should only be used with extreme care and, for example, at local scope as assertion statements allow to do.

```
CONTRACT_FUNCTION(
    template( typename T )
    (T*) (ialloc) ( void )
        precondition(
            namespace mpl = boost::mpl, // Assertion's namespace aliasing.
            using namespace std, // Assertion's using directive.
            typedef typename allocator<T>::pointer ptr, // And type definition.
            static_assert(
                  (mpl::or_<
                      mpl::bool_<sizeof(T) <= sizeof(int)>
                    , mpl::bool_<sizeof(ptr) <= sizeof(int*)>
                  >::value)
                , "within max size"
            )
        )
) {
    return static_cast<T*>(malloc(sizeof(T)));
}
```

These statements follow their usual C++ syntax but they are terminated with a comma `,` instead of a semicolon `;`. (The `BOOST_IDENTITY_TYPE` macro can be used to wrap eventual commas within the `typedef` statement.)

Assertion Requirements

In general, programming contract assertions can introduce a new set of requirements on the types used by the program. Some of these type requirements might be necessary only to program the assertions and they might not be required by the implementation itself. In such cases, if the code is compiled with contracts disabled (`CONTRACT_CONFIG_NO_PRECONDITIONS`, etc), the program might compile but it might no longer compile when contracts are enabled because some of the types do not provide all the operations necessary to check the contract assertions.

1. More in general, in some cases it might be acceptable or even desirable to cause a compile-time error when a program uses types that do not provide all the operations needed to check the contracts (because it is not possible to fully check the correctness of the program). In these cases, programmers specify contract assertions as we have seen so far (and maybe even use concepts to explicitly specify the contract type requirements, see the Concepts section).
2. However, in other cases it might be desirable that adding contracts to a program does not change its type requirements and that assertions are simply not checked when the types do not provide all the operations necessary to evaluate the asserted conditions. In these cases, programmers can use *assertion requirements* to disable compilation and run-time checking of specific assertions: [40]

```
assertion, requires constant-boolean-expression
```

(The constant boolean expression of an assertion requirement can be wrapped within round parenthesis `()` if it contains unwrapped commas.)

Let's consider the STL `std::vector<T>` class template. This template normally does not require the value type `T` to be `EqualityComparable` (i.e., to have an equality operator ==), it only requires `T` to be `CopyConstructible` (i.e., to have a copy constructor). However, in order to check the following `std::vector<T>::push_back` postcondition:

```
back() == value // Compiler error if not `EqualityComparable<T>`.
```

The type `T` must to be `EqualityComparable`. Therefore, the `EqualityComparable` requirement on `T` is introduced only by the contract assertions and it is not required by the `std::vector<T>` implementation.

In some cases, programmers might want compilation to fail when `T` is not `EqualityComparable` because in these cases it is not possible to fully check the `std::vector<T>` contract and therefore its correctness (in these cases programmers do not specify assertion requirements and let the compilation fail, or even better programmers can explicitly specify the assertion type requirements using concepts which will also fail compilation but hopefully with more informative error messages, see the Concepts section).

However, in other cases programmers might want to use the contracted version of `std::vector<T>` exactly as they use the non-contracted version of the template and therefore without failing compilation if `T` is not `EqualityComparable`. This can be achieved specifying assertion requirements for the `std::vector<T>::push_back` postcondition:

```
back() == value, requires boost::has_equal_to<T>::value // No error if not `EqualityComparable<T>`.
```

This postcondition will be checked only when `T` is `EqualityComparable`, otherwise the postcondition will be ignored otherwise causing no compilation error.

[40] **Rationale.** The assertion requirement syntax takes a constant boolean expression instead of a nullary boolean meta-function because the authors have found no reason to use a meta-function in this context. Furthermore, constant boolean expressions can be manipulated via the usual operators `not`, `and`, etc, therefore more naturally than boolean meta-functions which need to use `boost::mpl::not_`, `boost::mpl::and_`, etc instead.

For example (see also assertion_requirements_push_back.cpp): [41]

```
#include <boost/type_traits/has_equal_to.hpp>

CONTRACT_CLASS(
    template( typename T )
    class (vector)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        size() <= capacity(),
        capacity() <= max_size()
        // ...
    )

    CONTRACT_FUNCTION_TPL(
        public void (push_back) ( (T const&) value )
            precondition( size() < max_size() )
            postcondition(
                auto old_size = CONTRACT_OLDOF size(),
                auto old_capacity = CONTRACT_OLDOF capacity(),
                size() == old_size + 1,
                capacity() >= old_capacity,
                // Requirements disable assertion if `T` has no `operator==`.
                back() == value, requires boost::has_equal_to<T>::value
            )
    ) {
        vector_.push_back(value);
    }

    // ...
```

```
struct num // Not equality comparable.
{
    int value;
    explicit num ( int a_value ) : value(a_value) {}
};

int main ( void )
{
    vector<int> i;
    i.push_back(123);

    vector<num> n;
    n.push_back(num(123));
    BOOST_TEST(not boost::has_equal_to<num>::value);

    // ...
```

The `i.push_back(123)` call will check the postcondition `back() == value` because `int` is `EqualityComparable` and the assertion requirement `boost::has_equal_to<int>::value` is true. However, the `n.push_back(num(123))` call will not check, and in fact not even compile, the postconditions `back() == value` because `num` is not `EqualityComparable` and the assertion requirement `boost::has_equal_to<num>::value` is false. [42]

The Boost.TypeTraits library provides a set of meta-functions that are very useful to program assertion requirements. [43] However, C++ does not allow to inspect every trait of a type so there might be some assertion requirements that unfortunately cannot be programmed within the language.

[41] Future revisions of this library might provide wrappers that program contracts for the STL in appropriate header files `contract/std/vector.hpp`, `contract/std/algorithm.hpp`, etc (see also SGI STL and Ticket 47). However, given that STL implementations are usually considered to be "correct", it is not clear if STL class invariants and postconditions would add any value, maybe programmers would only find STL preconditions useful.

[42] Assertion requirements where first introduced by this library and they are not supported by [N1962], D, or Eiffel (even if they all allow to write contracts for templates). Based on the authors' experience, assertion requirements are necessary for a language that make extensive use of templates like C++. Furthermore, C++ does not automatically define equality operators `==` while it automatically defines copy constructors and that make the use of the assertion requirements for `EqualityComparable` a rather common practise (in Eiffel instead types can be both copied and compared for equality by default). It has been argued that it is not possible to check a program for correctness if types that are copyable cannot also be compared for equality and the authors experience with programming contracts confirms such an argument.

[43] As of Boost 1.50, `boost::has_equal_to` and similar traits always return true for an STL container even if the value type of the container does not have an operator `==`. This is arguably a defect of the STL that always defines an operator `==` for its containers even when a container template is instantiated with a value type that has no operator `==`, in which case the container operator `==` will produce a compiler error (the STL should instead use SFINAE to disable the declaration of the container operator `==` when the value type has no operator `==`). Future versions of Boost.TypeTraits will probably specialize `boost::has_equal_to` and similar traits to work around this behaviour of the STL. In the meanwhile, programmers can specialize these traits themselves if needed:

Another interesting use of assertion requirements is to model assertion computational complexity. In some cases, checking assertions can be as computationally expensive as executing the function body or even more. While preconditions, postconditions, etc can be disabled in groups at compile-time to improve performance (using `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, etc), it is also useful to be able to disable only specific assertions that are very computationally expensive while keeping all other preconditions, postconditions, etc enabled. For example (see also `assertion_complexity_factorial.cpp`):

```
// Header: std_has_equal_to.hpp
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/type_traits/is_convertible.hpp>
#include <boost/mpl/and.hpp>
#include <vector>

// STL defines `operator==` only between two identical containers (same value
// type, allocator, etc) and returning a type convertible to `bool`.

namespace boost {

// Specializations for `std::vector`.

template< typename T, class Alloc >
struct has_equal_to < std::vector<T, Alloc>, std::vector<T, Alloc> > :
    has_equal_to<T, T, bool>
{};

template< typename T, class Alloc, typename Ret >
struct has_equal_to < std::vector<T, Alloc>, std::vector<T, Alloc>, Ret > :
    mpl::and_<
          has_equal_to<T, T, bool>
        , is_convertible<Ret, bool>
    >
{};

// ... specialize `has_equal_to` for more STL containers.

} // namespace
```

```
// Assertion requirements used to model assertion computational complexity.
#define O_1      0 // O(1) constant (default).
#define O_N      1 // O(n) linear.
#define O_NN     2 // O(n * n) quadratic.
#define O_NX     3 // O(n^x) polynomial.
#define O_FACTN  4 // O(n!) factorial.
#define O_EXPN   5 // O(e^n) exponential.
#define O_ALL   10
#define COMPLEXITY_MAX O_ALL

CONTRACT_FUNCTION(
    int (factorial) ( int n )
        precondition( n >= 0 )
        postcondition(
            auto result = return,
            result >= 1,
            if(n < 2) (
                result == 1
            ) else (
                // Assertion compiled and checked only if within complexity.
                result == n * factorial(n - 1),
                        requires O_FACTN <= COMPLEXITY_MAX
            )
        )
) {
    if(n < 2) return 1;
    else return n * factorial(n - 1);
}
```

In this case the postcondition `result = n * factorial(n - 1)` has factorial computation complexity and it is compiled and checked at run-time only if the macro `COMPLEXITY_MAX` is defined to be `O_FACTN` or greater. Similarly, macros like `O_SMALLER_BODY`, `O_BODY`, and `O_GREATER_BODY` could have been defined to express computational complexity in relative terms with respect to the body computational complexity (see the Examples section). [44]

Assertion requirements can be used with assertions, constant assertions, and static assertions but they cannot be used with assertion statements and with the if-then-else statements of select assertions (however, if select assertions are programmed using the ternary operator `:?` then assertion requirements will disable the entire ternary operator `:?` expression including its if-condition).

Old-Of Requirements

Assertion requirements can be specified also for postconditions that use old values. However, old values have the additional requirement that the type of the expression passed to the `CONTRACT_OLDOF` macro must be `ConstantCopyConstructible`. If such a requirement is not met:

1. Either, programmers want the compiler to error because the postconditions using the old value cannot be evaluated (in which case programmers will not specify assertion requirements and they might explicitly specify type requirements using concepts so to get more informative error messages, see the Concepts section).

2. Or, programmers want the old value declaration that uses the `CONTRACT_OLDOF` macro to have no effect (because the old value cannot be copied) and the postconditions using such an old value to not be evaluated.

Unfortunately, C++ cannot automatically detect if a type has a copy constructor therefore a special trait `contract::has_oldof` had to be introduced to support this second case. The `contract::has_oldof<T>` trait is a unary boolean meta-function which is `boost::mpl::true_` by default for any type `T` and it must be specialized by programmers for types which old values cannot be copied.

For example, consider the following increment function (see also `has_oldof_inc.cpp`):

[44] [N1866], an earlier version of [N1962], introduced the notion of *importance ordering* that could be used to selectively disable assertions that where too expensive computationally. Important ordering was then dropped by [N1962]. Among other things, assertion requirements can be used to achieve the importance ordering functionality.

```
CONTRACT_FUNCTION(
    template( typename T )
    (T&) (inc) ( (T&) value, (T const&) offset )
        postcondition(
            auto old_value = CONTRACT_OLDOF value,      // Skip if no old-of.
            auto result = return,                       // Never skipped.
            value == old_value + offset, requires
                    contract::has_oldof<T>::value and   // Requires old-of.
                    boost::has_plus<T>::value and
                    boost::has_equal_to<T>::value,
            result == value, requires
                    boost::has_equal_to<T>::value
        )
) {
    return value += offset;
}

class num : boost::noncopyable // Non-copyable (for some reason...).
{
    public: explicit num ( int value ) : ptr_(new int(value)) {}
    private: num ( num const& other ) : ptr_(other.ptr_) {} // Private copy.
    public: ~num ( void ) { delete ptr_; }

    public: num operator+ ( num const& right ) const
        { return num(*ptr_ + *right.ptr_); }
    public: num& operator+= ( num const& right )
        { *ptr_ += *right.ptr_; return *this; }
    public: bool operator== ( num const& right ) const
        { return *ptr_ == *right.ptr_; }

    private: int* ptr_;
};

// Specialization disables old-of for non-copyable `num` (no C++ type trait can
// automatically detect copy constructors).
namespace contract
{
    template< > struct has_oldof < num > : boost::mpl::false_ {};
} // namespace
```

```
int i = 1, j = 2;
BOOST_TEST(inc(i, j) == 3);

num n(1), m(2);
BOOST_TEST(inc(n, m) == num(3));
```

In this example, the `num` type is non-copyable so the `contract::has_oldof<num>` specialization inherits from `boost::mpl::false_`. As a consequence, the call `inc(n, m)` will automatically skip the old value declaration:

```
auto old_value = CONTRACT_OLDOF value
```

The following postcondition will also be skipped because its assertion requirement lists `contract::has_oldof<T>` (where `T` is indeed the type of `value` passed to the `CONTRACT_OLDOF` macro):

```
value == old_value + offset, requires contract::has_oldof<T>::value && ...
```

Note how this postcondition actually requires the type `T` to have an old value, a plus operator `+`, and an equality operator `==` thus all of these requirements are programmed in the assertion requirements.

Finally, return value declarations are never skipped so there is no `has_result` trait. [45]

[45] **Rationale.** The result type needs to be copyable in order for the function itself to return a value so this library can always evaluate return value declarations.

Old and Result Value Copies

This library uses the class template `contract::copy` to copy old values and the result value for postconditions.

```
namespace contract
{
    template< typename T >
    class copy
    {
        public: copy ( T const& object ) ;
        public: T const& value ( void ) const ;
    };
} // namespace
```

The default implementation of the `contract::copy` class template requires that the type of the old value expression passed to the `CONTRACT_OLDOF` macro and the result type are `ConstantCopyConstructible` (i.e., these types provide a copy constructor that copies the object from a constant reference, the reference has to be constant so to ensure the constant-correctness of the copy operations that are executed within the contracts, see also the Constant-Correctness section):

```
// `ConstantCopyConstructible<T>` (which implies `CopyConstructible<T>`):
T::T ( T const& source ) ;  // OK: constant-correct copy constructor.

// `CopyConstructible<T>` (but not `ConstantCopyConstructible<T>`):
T::T ( T& source ) ;        // Error, copy constructor is not constant-correct.
```

Programmers can specialize the `contract::copy` class template to allow old value and result value copies even for types that are not `ConstantCopyConstructible` (in this case it is the programmers' responsibility to make sure that the copy operations that they define are constant-correct, otherwise the contracts will no longer be constant-correct). [46]

In the following example the old and result value declarations use the specialization `contract::copy<num>` so it is possible to access the old and result value of variables of type `num` even if `num` is non-copyable (see also `copy_inc.cpp`):

[46] Currently, it is not possible to force constant-correctness of the expression passed to the `CONTRACT_OLDOF` macro. Future revisions of this library will support `CONTRACT_OLDOF const( ... ) ...` to allow to make constant all variables (including global, static, etc) used by old value expressions (see also Ticket 52).

```
CONTRACT_FUNCTION(
    template( typename T )
    (T&) (inc) ( (T&) value, (T const&) offset )
        postcondition(
            auto old_value = CONTRACT_OLDOF value,  // Use `copy`.
            auto result = return, // Would use `copy` but reference so no copy.
            value == old_value + offset, requires
                    boost::has_plus<T>::value and
                    boost::has_equal_to<T>::value,
            result == value, requires
                    boost::has_equal_to<T>::value
        )
) {
    return value += offset;
}

class num : boost::noncopyable // Non-copyable (for some reason...).
{
    friend class contract::copy<num>; // Contract copy is friend.

    public: explicit num ( int value ) : ptr_(new int(value)) {}
    private: num ( num const& other ) : ptr_(other.ptr_) {}
    public: ~num ( void ) { delete ptr_; }

    public: num operator+ ( num const& right ) const
        { return num(*ptr_ + *right.ptr_); }
    public: num& operator+= ( num const& right )
        { *ptr_ += *right.ptr_; return *this; }
    public: bool operator== ( num const& right ) const
        { return *ptr_ == *right.ptr_; }

    private: int* ptr_;
};

// Specialization disables old-of for non-copyable `num` (no C++ type trait can
// automatically detect copy constructors).
namespace contract
{
    template< >
    class copy < num >
    {
        public: explicit copy ( num const& n ) : num_(*n.ptr_) {}
        public: num const& value ( void ) const { return num_; }

        private: num num_;
    };
} // namespace
```

```
int i = 1, j = 2;
BOOST_TEST(inc(i, j) == 3);

num n(1), m(2);
BOOST_TEST(inc(n, m) == num(3));
```

Both calls `inc(i, j)` and `inc(n, m)` check all postconditions. The first call copies the old and result values using `int`'s default copy constructor, the second call copies the old and result values using the copy operation implemented by the `contract::copy<num>` specialization.

Pure Virtual Functions

It is possible to program contracts for pure virtual functions. Pure virtual functions are specified with the usual `= 0;` symbol replacing the function definition just after the contract macros, for example (see also `pushable.hpp`):

```
CONTRACT_CLASS(
    template( typename T )
    class (pushable)
) {
    // Subcontracting: In `and` with derived class invariants.
    CONTRACT_CLASS_INVARIANT_TPL( void )

    public: typedef typename std::vector<T>::const_reference const_reference;

    CONTRACT_FUNCTION_TPL(
        public virtual void (push_back) ( (T const&) value ) new
            // Subcontracting: In `or` with overriding function preconditions.
            precondition( false ) // Force check to overriding preconditions.
            // Subcontracting: In `and` with overriding function postconditions.
            postcondition(
                back() == value, requires boost::has_equal_to<T>::value
            )
    ) = 0; // Contract for pure virtual function.

    public: virtual const_reference back ( void ) const = 0;
};
```

Furthermore, using the `CONTRACT_MEMBER_BODY` macro, a derived class can be programmed without using the `CONTRACT_CLASS` and `CONTRACT_FUNCTION` macros even when the base class has pure virtual functions (see also `default_subcontracting_base.cpp`):

```
CONTRACT_CLASS(
    class (base) // Declared with contract macros.
) {
    CONTRACT_CLASS_INVARIANT( std::clog << "base::inv" << std::endl )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void )
            precondition( std::clog << "base::f::pre" << std::endl )
            postcondition( std::clog << "base::f::post" << std::endl )
    ) = 0;
};

class deriv : public base // Declared without contract macros...
{
    // ...but using `BODY` instead of `CLASS` and `FUNCTION` macros, overriding
    // functions have exact same contracts as base (not real subcontracting).
    public: virtual void CONTRACT_MEMBER_BODY(f) ( void )
    {
        std::clog << "deriv::f::body" << std::endl;
    }
};
```

This use of the `CONTRACT_MEMBER_BODY` macro does not force programmers of derived classes to use the library macros `CONTRACT_CLASS` and `CONTRACT_FUNCTION` which would otherwise introduce the unusual library syntax in the derived class declaration. However, in this case the derived class is forced to inherit the base class invariants, preconditions, and postconditions exactly as they are without the possibility to truly subcontract the base class (which is instead possible when the `CONTRACT_CLASS` and `CONTRACT_FUNCTION` macros are used in declaring the derived class). [47] Correctly, the derived class can never avoid checking the base class contracts neither when declared using the `CONTRACT_CLASS` and `CONTRACT_FUNCTION` macros, nor when declared using the `CONTRACT_MEMBER_BODY` macro.

Subcontracting Preconditions

While subcontracting is theoretically sound and justified by the substitution principle, in practise subcontracted preconditions might be confusing for programmers because of the implications of evaluating overriding preconditions in logic-or with overridden preconditions (this is not the case for subcontracted postconditions and class invariants which usually behave as programmers expect because they are evaluated in logic-and).

For example, consider the following base class `integer` which holds an integral value and its derived class `natural` which holds a natural (i.e., non-negative integer) value (see also `subcontract_pre_natural_failure.cpp`):

[47] **Rationale.** Older revisions of this library defined the macro `CONTRACT_MEMBER_BODY(class_type, function_name)` taking two parameters so this macro could not be used in declaring derived classes and the `CONTRACT_CLASS` and `CONTRACT_FUNCTION` macros where required when programming a class inheriting from a base class with contracts. However, the authors prefer to leave the choice of programming full contracts for derived classes to the derived class programmers because of the unusual declaration syntax introduced by this library might not be necessary when derived classes do not need to alter the base's class contract.

```
CONTRACT_CLASS(
    class (integer)
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (set) ( int value )
            // No preconditions so this and all overrides can always be called.
            postcondition( get() == value )
    ) {
        value_ = value;
    }

    CONTRACT_FUNCTION(
        public virtual int (get) ( void ) const
    ) {
        return value_;
    }

    private: int value_;
};

CONTRACT_CLASS(
    class (natural) extends( public integer ) // Subcontract.
) {
    CONTRACT_CLASS_INVARIANT( get() >= 0 )

    CONTRACT_FUNCTION(
        public virtual void (set) ( int value )
            precondition( value >= 0 ) // NOTE: Will not fail because of base.
    ) {
        integer::CONTRACT_MEMBER_BODY(set)(value);
    }
};
```

From reading the contracts we conclude that:

- `integer::get` returns the latest value set (as specified by `integer::set` postconditions).
- There is no constraint on the integer value that is passed to `integer::set` (because `integer::set` has no precondition).
- `natural::get` always returns a non-negative value (as specified by the `natural` class invariants).
- Only non-negative values can be passed to `natural::set` (as specified by `natural::set` preconditions).

This last conclusion is incorrect! Negative values can be passed to `natural::set` because they can be passed to its base virtual function `integral::set` (preconditions cannot only be weakened). The complete set of `natural::set` preconditions is given by its base virtual function `integer::set` preconditions (which are always `true` because they are not specified) evaluated in logic-or with the preconditions explicitly specified by `natural::set` (i.e., `value >= 0`):

```
(true) or (value >= 0)
```

Obviously, this always evaluates to `true` regardless of `value` being non-negative. This is correct in accordance with the substitution principle for which `natural::set` can be called in any context where `integer::set` is called because `natural` inherits from `integer`. Given that `integer::set` can be called regardless of `value` being non-negative (because it has no precondition) there is no precondition that we can later specify for `natural::set` that will change that and in fact `natural::set` can also be called with negative values without failing its subcontracted preconditions. For example, the call:

```
natural n;
n.set(-123); // Error: Fails call invariants instead of preconditions.
```

Fails the class invariants checked while exiting `natural::set`:

```
class invariant (on exit) number 1 "get() >= 0" failed: file "natural_subcontractpre.cpp", line 30
```

Ideally, this call would have failed much earlier at `natural::set` preconditions (in fact, the `natural::set` body is executed with the logically invalid negative value `-123` which could in general lead to catastrophic errors and mysterious bugs).

The issue here is in the design. A natural number is not an integer number because while it is valid to use an integer number in a context where it is assigned to a negative number, it is not valid to use a natural number in such a context so a natural number should not inherit from an integer number. Inheritance models the *is-a* relationship which should not be used in this case as the contracts and the substitution principle are telling us.

Note

Note that if a virtual member function has no preconditions, it means that it is always valid to call it, and (in accordance with the substitution principle) this semantic cannot be changed by the contracts of any overriding function no matter what preconditions we specify for it. Similarly, if an overriding member function has no preconditions, it means that is is always valid to call it regardless of possible preconditions specified by any function that it overrides.

An overriding function can specify `precondition( false )` if it wants to keep the same preconditions of the functions that is overriding. A pure virtual function can specify `precondition( false )` to indicate that overriding functions will specify preconditions (this only makes sense for pure virtual functions because a function with `precondition( false )` can never be called successfully unless it is overridden and its preconditions are weakened, that might be acceptable for pure virtual because they must always be overridden).

If programmers find subcontracted preconditions confusing, this library allows to forbid them by defining the configuration macro `CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS`. When this macro if defined, the library will generate a compile-time error if a derived class tries to override preconditions of member functions of any of its base classes (this is also the subcontracting behaviour specified by [N1962]). However, note that in case of multiple-inheritance preconditions from the overridden function from all bases classes will always be checked in logic-or with each other so preconditions from a base class could still be weaken by the preconditions of another base class even when the `CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS` macro is defined. By default the `CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS` macro is not defined and this library allows to override preconditions (as specified by for Eiffel by [Meyer97] and in accordance with the substitution principle).

Another difference between subcontracted preconditions and subcontracted postconditions or class invariants is that subcontracted preconditions will always report a failure of the overridden preconditions. For example, consider a set of base classes `bN` and a derived class `d`:

```
CONTRACT_CLASS(
    struct (a)
) {
    ... // No member function `f`.
};

CONTRACT_CLASS(
    struct (bN)
) {
    CONTRACT_CLASS_INVARIANT( ... )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void ) // Overridden function.
            precondition( ... )
            postcondition( ... )
    ) {}

    ...
};

CONTRACT_CLASS(
    struct (c)
) {
    ... // No member function `f`.
};

CONTRACT_CLASS(
    struct (d) extends( b1, a, b2, c, b3 )
) {
    CONTRACT_CLASS_INVARIANT( ... )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void ) // Overriding function.
            precondition( ... )
            postcondition( ... )
    ) {}

    ...
};
```

When `d::f` is called, its subcontracted class invariants, postconditions, and preconditions are evaluated using the following expressions (the base classes `a` and `c` do not declare a virtual function `f` so they are automatically excluded from `d::f` subcontracting):

```
b1::class-invariants  and b2::class-invariants  and b3::class-invariants  and d::class-invariants  // `d::f` subcontracted class invariants.
b1::f::postconditions and b2::f::postconditions and b3::f::postconditions and d::f::postconditions // `d::f` subcontracted postconditions.
b1::f::preconditions  or  b2::f::preconditions  or  b3::f::preconditions  or  d::f::preconditions  // `d::f` subcontracted preconditions.
```

When subcontracted class invariants or subcontracted postconditions fail, this library reports the first failed condition which can in general be in the base class contracts because they are checked first in the logic-and chain (this can report a failure from any subcontracted class invariants `bN::class-invariants` or `d::class-invariants`, and subcontracted postconditions `bN::f::postconditions` or `d::f::postconditions`). However, when subcontracted preconditions fail it means that all overridden preconditions as well as the overriding preconditions have failed (because subcontracted preconditions are evaluated in logic-or). In this case, this library will report the last evaluated failure which will always be in the overriding preconditions (always report a failure from `d::f::preconditions`). If programmers want instead the library to report the failure from the first overridden precondition that failed, they can define the configuration macro `CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE` (always report a failure from `b1::f::preconditions`).

Static Member Functions

It is possible to program contracts for static member functions. Static member functions cannot access the object therefore their preconditions and postconditions also cannot access the object and they can only access other static members. This library allows to specify a subset of class invariants called *static class invariants* which do not access the object and that are checked at entry and exit of every constructor, destructor, and member functions even if static. (Non-static class invariants are instead not checked at constructor entry, at destructor exit, and at entry and exit of static member functions because they require accessing the object, see also the Contract Programming Overview section.)

Static class invariants are empty (`void`) by default unless they are explicitly specified within the `CONTRACT_CLASS_INVARIANT` macro using the following syntax: [48]

[48] Static class invariants are not part of [N1962] and they were first introduced by this library.

```
static class( assertion1, assertion2, ... )
```

For example (see also `static_contract_instance_counter.cpp`):

```
CONTRACT_CLASS(
    template( typename T )
    class (instance_counter)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        object(), // Non-static class invariants.
        static class( // Static class invariants.
            count() >= 0
            // ...
        )
    )

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (instance_counter) ( (T*) the_object )
            precondition( the_object )
            postcondition(
                auto old_count = CONTRACT_OLDOF count(),
                count() == old_count + 1,
                object() == the_object
            )
            initialize( object_(the_object) )
    ) {
        count_++;
    }

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~instance_counter) ( void )
            // FUTURE: Destructors could have static postconditions.
            // postcondition: count() = oldof count() - 1
    ) {
        delete object_;
        count_--;
    }

    CONTRACT_FUNCTION_TPL(
        public (T const* const) (object) ( void ) const
    ) {
        return object_;
    }

    CONTRACT_FUNCTION_TPL( // Contract static member function.
        public static int (count) ( void )
            // No preconditions nor postconditions for this example but when
            // present no object can be accessed by assertions (i.e., `static`).
    ) {
        return count_;
    }

    private: static int count_;
    private: T* object_;
};

template< typename T >
int instance_counter<T>::count_ = 0;
```

In this example there is only one static class invariant assertion `count() >= 0` and it is checked at entry and exit of every constructor, destructor, and member function including the static member function `count`. If the static member function `count` had preconditions or postconditions, they would not be able to access the object (i.e., preconditions and postconditions will also be executed in static context).

Volatile Member Functions

It is possible to program contracts for volatile member functions. Volatile member functions access the object as volatile therefore their preconditions and postconditions also access the object as volatile and can only access other volatile members. This library allows to specify a different set of class invariants called *volatile class invariants* which can only access the object as volatile and that are checked at entry and exit of every volatile member function. (Constructors and destructors never access the object as volatile so volatile class invariants are not checked by constructors and destructors, see also the Contract Programming Overview section.)

Volatile class invariants are assumed to be the same as non-volatile class invariants unless they are explicitly specified within the `CONTRACT_CLASS_INVARIANT` macro using the following syntax: [49]

```
volatile class( assertion1, assertion2, ... )
```

In most cases, it will be necessary to explicitly specify volatile class invariants when using volatile member functions (unless volatile overloads are provided for every member function that is used by the non-volatile class invariants).

For example (see also `volatile_contract_shared_instance.cpp`):

```
CONTRACT_CLASS(
    template( typename T )
    class (shared_instance)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        queries() >= 0, // Non-volatile class invariants.
        volatile class( // Volatile class invariants.
            object()
            // ...
        )
    )

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (shared_instance) ( (T*) the_object )
            precondition( the_object )
            postcondition( object() == the_object )
            initialize( object_(the_object), queries_(0) )
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~shared_instance) ( void )
    ) {
        delete object_;
    }

    CONTRACT_FUNCTION_TPL( // Contracts for volatile member function.
        public (T const volatile* const) (object) ( void ) const volatile
            // No preconditions nor postconditions for this example but when
            // present object is `const volatile` within assertions.
    ) {
        queries_++;
        return object_;
    }

    CONTRACT_FUNCTION_TPL(
        public int (queries) ( void ) const
    ) {
        return queries_;
    }

    private: T volatile* object_;
    private: mutable int queries_;
};
```

[49] Volatile class invariants are not part of [N1962] and they were first introduced by this library. It is not clear if there are real applications for volatile class invariants mainly because real applications of `volatile` itself are not clear in C++. One very interesting use of `volatile` has been to avoid race conditions in concurrent programming (but contracts are not useful for such an application of `volatile`).

As usual, contracts are checked in constant-correct context therefore only `const volatile` members can be accessed from volatile class invariants, preconditions, and postconditions.

Operators

It is possible to program contracts for operators (both member function operators and free function operators). The operator name must be specified using the following syntax when passed to the `CONTRACT_FUNCTION`, `CONTRACT_FREE_BODY`, and `CONTRACT_MEMBER_BODY` macros:

```
operator(symbol)(arbitrary-alphanumeric-name)
```

The first parameter is the usual operator symbol (`==`, `+`, etc) the second parameter is an arbitrary but alphanumeric name (`equal`, `plus`, etc). The operator `new`, the operator `delete`, and implicit type conversion operators for fundamental types containing no symbol can also be specified simply as:

```
operator new
operator delete
operator fundamental-type-without-symbols
```

The comma operator must be specified as: [50]

```
operator comma
```

For example (see also `member_operator_string.cpp`):

[50] **Rationale.** The comma operator cannot be specified using the comma symbol as in `operator(,)(my_comma_operator)` because the preprocessor uses commas to separate macro parameters so it is not possible to distinguish that from an implicit type conversion operator like `operator(std::vector<T, Allocator>)(std_vector)` (note that the operator name is arbitrary so it also cannot be used to distinguish these two cases).

```
CONTRACT_CLASS(
    class (string)
) {
    CONTRACT_CLASS_INVARIANT(
        static class( pointers >= 0, arrays >= 0 )
    )

    public: static int pointers;
    public: static int arrays;

    CONTRACT_CONSTRUCTOR(
        public explicit (string) ( (char const*) c_str, default("") )
            initialize( string_(c_str) )
    ) {}

    CONTRACT_DESTRUCTOR(
        public (~string) ( void )
    ) {}

    CONTRACT_FUNCTION( // Symbolic operators: `(==)(equal)`, `(())(call)`, etc.
        public bool operator(==)(equal) ( (string const&) right ) const
    ) {
        return string_ == right.string_;
    }

    CONTRACT_FUNCTION( // Implicit type conversion operator (keyword type).
        public operator char const ( void ) const
    ) {
        return string_[0];
    }

    CONTRACT_FUNCTION( // Implicit type conversion operator (symbolic type).
        public operator(char const*)(char_const_ptr) ( void ) const
    ) {
        return string_.c_str();
    }

    CONTRACT_FUNCTION( // Implicit type conversion operator (type with commas).
        public template( typename T, class Allocator )
        operator(std::vector<T, Allocator>)(std_vector) ( void ) const
    ) {
        std::vector<T, Allocator> v(string_.size());
        for(size_t i = 0; i < string_.size(); ++i) v[i] = string_[i];
        return v;
    }

    CONTRACT_FUNCTION( // Comma operator (use `comma` to diff. from above).
        public (string&) operator comma ( (string const&) right )
    ) {
        string_ += right.string_;
        return *this;
    }

    // All memory operators (new, delete, new[], and delete[]) must be
    // explicitly `static` (because pp can't inspect new[] and delete[] that
    // could be any symbolic operator like ==, +, etc).

    CONTRACT_FUNCTION(
        public static (void*) operator new ( size_t size )
    ) {
        void* p = malloc(size);
        if(p == 0) throw std::bad_alloc();
        pointers++;
```

```
        return p;
    }

    CONTRACT_FUNCTION(
        public static void operator delete ( (void*) pointer )
    ) {
        if(pointer) {
            free(pointer);
            pointers--;
        }
    }

    CONTRACT_FUNCTION(
        public static (void*) operator(new[])(new_array) ( size_t size )
    ) {
        void* a = malloc(size);
        if(a == 0) throw std::bad_alloc();
        arrays++;
        return a;
    }

    CONTRACT_FUNCTION(
        public static void operator(delete[])(delete_array) ( (void*) array )
    ) {
        if(array) {
            free(array);
            arrays--;
        }
    }

    private: std::string string_;
};

int string::pointers = 0;
int string::arrays = 0;
```

The memory operators `new`, `delete`, `new[]`, and `delete[]` must always be specified `static` when they are programmed as member function operators. [51]

Free function operators are programmed using the same syntax for the operator names.

Nested Classes

It is possible to program contracts for nested classes noting that:

1. The access level `public`, `protected`, or `private` must be specified for contracted members and therefore also for nested classes.

2. The `_TPL` macros must be used within templates so the `CONTRACT_CLASS_TPL` macro needs to be used when programming a class nested within a class template.

For example (see also `nested_class_bitset.cpp` and the Concepts section):

[51] **Rationale.** C++ automatically promotes memory member operators to static members so the `static` keyword is optional for these member operators. However, this library cannot inspect the operator symbol using the preprocessor to check if the operator is a memory member operator or not because the symbol contains non-alphanumeric tokens (which cannot be concatenated by the preprocessor). Therefore, this library always requires memory member operators to be explicitly declared as static member functions.

```
CONTRACT_CLASS( // Enclosing class.
    template( size_t N )
    class (bitset)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        static class( size() == N )
    )

    CONTRACT_CLASS_TPL( // Nested class.
        public class (reference)
    ) {
        CONTRACT_CLASS_INVARIANT_TPL( bitptr_ )

        friend class bitset;

        CONTRACT_CLASS_TPL( // Nested (twice) class template with concepts.
            private template( typename T )
                requires( boost::DefaultConstructible<T> )
            class (bit)
        ) {
            CONTRACT_CLASS_INVARIANT_TPL( void )

            CONTRACT_CONSTRUCTOR_TPL(
                public (bit) ( void )
                    initialize( value_() )
            ) {}

            CONTRACT_FUNCTION_TPL(
                public void (from_bool) ( bool value )
                    postcondition( to_bool() == value )
            ) {
                value_ = value;
            }

            CONTRACT_FUNCTION_TPL(
                public bool (to_bool) ( void ) const
            ) {
                return value_;
            }

            private: T value_;
        };

        CONTRACT_CONSTRUCTOR_TPL(
            private (reference) ( void )
                postcondition( bitptr_->to_bool() == int() )
                initialize(
                    bitptr_(boost::shared_ptr<bit<int> >(new bit<int>()))
                )
        ) {}

        CONTRACT_DESTRUCTOR_TPL(
            public (~reference) ( void )
        ) {}

        CONTRACT_FUNCTION_TPL(
            public operator bool ( void ) const
        ) {
            return bitptr_->to_bool();
        }

        CONTRACT_FUNCTION_TPL(
            public (reference&) operator(=)(assign) ( bool const bit_value )
```

```
                postcondition( auto result = return, result == *this )
        ) {
            bitptr_->from_bool(bit_value);
            return *this;
        }

        private: boost::shared_ptr<bit<int> > bitptr_;
    };

    CONTRACT_FUNCTION_TPL(
        public (reference) operator([])(at) ( size_t index )
            precondition( index < N )
    ) {
        return bitset_[index];
    }

    CONTRACT_FUNCTION_TPL(
        public static size_t (size) ( void )
    ) {
        return N;
    }

    private: reference bitset_[N];
};
```

Friends

It is possible to program contracts for friend classes and friend functions. Classes can be declared friends and later contracted using the `CONTRACT_CLASS` macro as usual:

```
CONTRACT_CLASS( // Not even necessary to contract this class.
    class (x)
) {
    CONTRACT_CLASS_INVARIANT( void )

    friend class y ;

    // ...
};

CONTRACT_CLASS(
    class (y)
) {
    CONTRACT_CLASS_INVARIANT( void )

    // ...
};
```

Friend functions that are declared and defined within the enclosing class use the `friend` keyword within the `CONTRACT_FUNCTION` macro (note that the access level `public`, `protected`, or `private` is optional in this case because these friend functions are not member functions). Friend functions that are either forward declared friends or that are defined friends within the enclosing class, must use the `CONTRACT_FREE_BODY` macro. For example (see also `friend_counter.cpp`):

```
CONTRACT_CLASS( // Not even necessary to contract this class.
    template( typename T ) requires( boost::CopyConstructible<T> )
    class (counter)
) {
    CONTRACT_CLASS_INVARIANT_TPL( value() >= 0 );

    CONTRACT_FUNCTION_TPL( // Friend (contracted declaration and definition).
        public friend (counter) operator(/)(div) ( (counter const&) left,
                (T const&) right )
            precondition( right > 0 ) // Strictly positive, cannot be zero.
            postcondition(
                auto result = return,
                result.value() * right == left.value()
            )
    ) {
        return counter(left.value() / right);
    }

    // Friend forward declaration (contracted declaration and definition below).
    public: template< typename U >
    friend bool CONTRACT_FREE_BODY(operator(==)(equal)) ( // Use BODY.
            counter<U> const& left, U const& right ) ;

// NOTE: Forward friend template instantiations give internal MSVC error.
#if defined(BOOST_MSVC) && !defined(CONTRACT_CONFIG_NO_POSTCONDITIONS)
    // Friend definition (contracted declaration below).
    public: friend counter CONTRACT_FREE_BODY(operator(*)(mult)) ( // Use BODY.
            counter const& left, T const& right )
    {
        return counter(left.value() * right); // Contract checked `right >=0` .
    }
#endif

    CONTRACT_CONSTRUCTOR_TPL( // Public constructor gets next counter value.
        public explicit (counter) ( void )
            initialize( value_(next_value_++) )
    ) {}

    CONTRACT_CONSTRUCTOR_TPL( // Private constructor.
        private explicit (counter) ( (T const&) a_value )
            precondition( a_value >= 0 )
            postcondition( value() == a_value )
            initialize( value_(a_value) )
    ) {}

    CONTRACT_FUNCTION_TPL(
        public (T) (value) ( void ) const
            postcondition( auto result = return,  result >= 0 )
    ) {
        return value_;
    }

    private: T value_;
    private: static T next_value_;
};

template<typename T>
T counter<T>::next_value_ = T();

CONTRACT_FUNCTION(
    template( typename U )
    bool operator(==)(equal) ( (counter<U> const&) left, (U const&) right )
        precondition( right >= 0 )
```

```
        postcondition(
            auto result = return,
            result == (left.value() == right)
        )
) {
    return left.value() == right;
}

#if defined(BOOST_MSVC) && !defined(CONTRACT_CONFIG_NO_POSTCONDITIONS)
CONTRACT_FUNCTION(
    (counter<int>) operator(*)(mult) ( (counter<int> const&) left,
            (int const&) right )
        precondition( right >= 0 )
        postcondition(
            auto result = return,
            result.value() == left.value() * right
        )
) ;
#endif
```

The class enclosing the friend declarations might or might not be contracted.

Template Specializations

It is possible to program contracts for class template specializations. The generic class template might or might not be contracted (see also `template_specializations_vector.cpp`):

```
CONTRACT_CLASS( // Not even necessary to contract this template.
    template( typename T, class Allocator, default std::allocator<T> )
    class (vector) extends( public pushable<T> )
) {
    // ...
```

The template specialization types follow the class name as usual but they are wrapped within round parenthesis `( bool, Allocator )` instead than angular parenthesis `< bool, Allocator >` as in the following partial specialization:

```
CONTRACT_CLASS( // Template specialization (one template parameter).
    template( class Allocator )
    class (vector) ( bool, Allocator ) extends( public pushable<bool> )
) {
    // ...
```

The syntax `template( void )` must be used instead of `template< >` to indicate explicit specializations:

```
CONTRACT_CLASS( // Template specialization (no template parameter).
    template( void )
    class (vector) ( bool ) extends( public pushable<bool> )
) {
    // ...
```

Exception Specifications and Function-Try Blocks

It is possible to program exception specifications and function-try blocks for constructors, destructors, member functions, and free functions with contracts. Exception specifications are part of the function declarations therefore they are programmed within the contract macros but the special syntax `throw( void )` must be used instead of `throw( )` to specify that no exception is thrown:

```
throw( exception-type1, exception-type2, ... ) // As usual.
throw( void ) // Instead of `throw( )`.
```

Function-try blocks are part of the function definition so they are normally programmed outside the contract macros. However, for constructors with member initializers, the member initializers must be programmed within the `CONTRACT_CONSTRUCTOR` macro and therefore also the function-try blocks must be programmed within the macro using the following syntax:

```
try initialize( initializer1, initializer2, ... )
catch(exception-declaration1) ( instruction1; instruction2; ... )
catch(exception-declaration2) ( instruction1; instruction2; ... )
catch(...) ( instruction1; instruction2; ... )
```

As usual, only one catch statement must be specified and the other catch statements are optional plus `catch(...)` can be used to catch all exceptions. Note however that round parenthesis `( ... )` are used instead of curly parenthesis `{ ... }` to wrap the catch statement instructions (then the catch instructions are programmed with the usual syntax and separated by semicolons `;`). (The maximum number of catch statements that can be programmed for constructor-try blocks is specified by the `CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES` macro.)

In the following example the constructor uses a function-try block to throw only `out_of_memory` and `error` exceptions while the destructor uses exception specifications to throw no exception (see also `exception_array.cpp`):

```
CONTRACT_CLASS(
    template( typename T )
    class (array)
) {
    CONTRACT_CLASS_INVARIANT_TPL( size() >= 0 )

    public: struct out_of_memory {};
    public: struct error {};

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (array) ( size_t count )
            precondition( count >= 0 )
            postcondition( size() == count )
            // Function try-blocks are programmed within the macros only for
            // constructors with member initializers otherwise they are
            // programmed with the body definitions and outside the macros.
            try initialize( // Try block for constructor initializers and body.
                data_(new T[count]),
                size_(count)
            // Use `BOOST_IDENTITY_TYPE` if the exception type has unwrapped
            // commas or leading symbols.
            ) catch(std::bad_alloc&) (
                throw out_of_memory();
            ) catch(...) (
                throw error();
            )
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~array) ( void ) throw( void ) // Throw nothing.
    ) {
        delete[] data_;
    }

    // ...
```

(The `BOOST_IDENTITY_TYPE` macro can be used to wrap the catch statement exception declaration types if they contain unwrapped commas.)

Exception specifications and function-try blocks apply only to exceptions thrown by the function body and not to exceptions thrown by the contracts themselves (if any) and by the contract checking code generated by this library macros. [52]

Specifying Types (no Boost.Typeof)

This library uses Boost.Typeof to automatically deduces postcondition old value types and constant assertion variable types. If programmers do not want the library to use Boost.Typeof they can explicitly specify these types.

The types of postcondition old values are specified instead of using `auto` and they must be wrapped within round parenthesis unless they are fundamental types containing no symbol (these must match the type of the specified old-of expressions):

[52] **Rationale.** [N1962] specifies that function-try blocks should only apply to the body code and not to the contract code. No explicit requirement is stated in [N1962] for exception specifications but the authors have decided to adopt the same requirement that applies to function-try blocks (because it seemed of little use for the contract to throw an exception just so it is handled by the exception specification).

```
(type) variable = CONTRACT_OLDOF oldof-expression
```

The types of constant assertion variables are specified just before each variable name and they also must be wrapped within round parenthesis unless they are fundamental types containing no symbol:

```
const( (type1) variable1, (type2) variable2, ... ) boolean-expression
```

For example (see also `typed_counter.cpp`):

```
CONTRACT_CLASS(
    class (counter)
) {
    CONTRACT_CLASS_INVARIANT( void )

    public: static unsigned value;

    CONTRACT_FUNCTION(
        public static int (next) ( void )
            postcondition( // Explicit types so no Boost.Typeof.
                auto result = return,
                unsigned const old_value = CONTRACT_OLDOF value,
                const( unsigned value, unsigned const old_value )
                        value == old_value + 1,
                result == int(value)
            )
    ) {
        return ++value;
    }
};

unsigned counter::value = 0;
```

Note that postcondition result values are always specified using `auto`:

```
auto result-variable = return
```

This is because the function return type is know by the library as specified in the function declaration within the `CONTRACT_FUNCTION` macro. Therefore, the postcondition result value type is never explicitly specified by programmers (but Boost.Typeof is never used to deduce it).

Note

It is recommended to not specify these types explicitly and let the library internally use Boost.Typeof to deduce them because the library syntax is more readable without the explicit types. However, all types must be properly registered with Boost.Typeof as usual in order to use type-of emulation mode on C++03 compilers that do not support native type-of (see Boost.Typeof for more information).

Block Invariants and Loop Variants

Block invariants can be used anywhere within the function body and they are used to assert correctness conditions of the implementation (very much like `assert`). They are programmed specifying a list of assertions to the `CONTRACT_BLOCK_INVARIANT` macro (including static, constant, and select assertions):

```
CONTRACT_BLOCK_INVARIANT( assertion1, assertion2, ... )
```

When block invariants are used within a loop, they are also called *loop invariants* (e.g., Eiffel uses this terminology).

Furthermore, this library allows to specify *loop variants*. A loop variant is a non-negative monotonically decreasing number that is updated at each iteration of the loop. The specified loop variant expression is calculated by this library at each iteration of the loop and it is automatically asserted to be non-negative (`>= 0`) and to decrease monotonically from the previous loop iteration. Because the loop variant monotonically decreases and it cannot be smaller than zero, either the loop terminates or one of the two library assertions will eventually fail in which case the library will call the `contract::loop_variant_broken` handler therefore detecting and stopping infinite loops. Each given loop can only have one variant which is specified using the `CONTRACT_LOOP_VARIANT` macro. The enclosing loop (`while`, `for`, `do`, etc) must be declared using the `CONTRACT_LOOP` macro:

```
CONTRACT_LOOP( loop-declaration ) {
    CONTRACT_LOOP_VARIANT( non-negative-monotonically-decreasing-expression )
    ...
}
```

Note that the library cannot automatically make constant any of the variables within the function body therefore constant block invariant assertions and constant loop variant expressions should be used by programmers if they want to enforce constant-correctness for block invariants and loop variants.

The following example uses a loop to calculate the Greatest Common Divisor (GCD) of two integral numbers (see also `blockinv_loopvar_gcd.cpp`):

```
CONTRACT_FUNCTION(
    template( typename T )
    (T) (gcd) ( (T const&) a, (T const&) b )
        precondition(
            static_assert(boost::is_integral<T>::value, "integral type"),
            a != 0,
            b != 0
        )
        postcondition(
            auto result = return,
            result <= a,
            result <= b
        )
) {
    T x = a, y = b;
    // Block invariant assert conditions within body,
    CONTRACT_BLOCK_INVARIANT_TPL( const( x, a ) x == a, const( y, b ) y == b )

    CONTRACT_LOOP( while(x != y) ) { // Define a loop with variants (below).
        // Block invariant for loops (i.e., loop invariants).
        CONTRACT_BLOCK_INVARIANT_TPL( const( x ) x > 0, const( y ) y > 0 )
        // Loop variant checked to be non-negative and monotonically decreasing.
        CONTRACT_LOOP_VARIANT_TPL( const( x, y ) std::max(x, y) )

        if(x > y) x = x - y;
        else y = y - x;
    }

    return x;
}
```

Eiffel supports loop variants but [N1962] does not. Loop variants might not be very useful especially if Boost.Foreach or similar constructs are used to ensure loop termination.

Contract Broken Handlers (Throw on Failure)

When a contract assertion fails, this library prints a message on the standard error `std::cerr` and it terminates the program calling `std::terminate`. Programmers can change this behavior customizing the actions that the library takes on contract assertion failure by setting the contract broken handler functions. By default the library terminates the program because a contract failure indicates that the program is in an invalid state that programmers specified it should never happen (so the only sensible assumption is that the program execution should not continue). However, in some cases programmers might need to handle even such catastrophic failures by executing some fail-safe code instead of terminating the program and that can be done customizing the contract broken handlers.

The library functions `contract::set_precondition_broken`, `contract::set_postcondition_broken`, `contract::set_class_invariant_broken`, `contract::set_block_invariant_broken`, and `contract::set_loop_variant_broken` can be used to customize the action to taken in case of precondition, postcondition, class invariant, block invariant, and loop variant failure respectively. Furthermore, class invariants are checked at entry, at normal exit, and at exit but when exceptions are thrown so more granular handlers can be set for each of these cases using `contract::set_class_invariant_broken_on_entry`, `contract::set_class_invariant_broken_on_exit`, and `contract::set_class_invariant_broken_on_throw` (using `contract::set_class_invariant_broken` is equivalent to setting all these class invariant broken handles to the same handler). [53]

This library passes a parameter of type `contract::from` to the contract broken handler functions indicating the context that failed the contract assertion (e.g., this parameter will be set to `contract::FROM_DESTRUCTOR` if the contract assertion failed from a destructor): [54]

[53] [N1962] does not allow to configure class invariant broken handlers differently for entry, exit, and throw.
[54] The `contract::from` parameter was not part of [N1962] contract broken handlers but the proposal hinted that it might be needed (need that was confirmed by the implementation of this library).

```
void contract-broken-handler ( contract::from const& context )
```

Important

In order to comply with the STL exception safety requirements, destructors should never throw. Therefore, even if programmers customize the contract broken handlers to throw exceptions instead of terminating, the handlers should never throw when contract assertions fail from a destructor (and the `contract::from` parameter can be used to discriminate this case).

The contract broken handlers are always invoked with an active exception that refers to the exception that failed the contract: [55]

1. Either an exception that was explicitly thrown by the user from the contract (e.g. `not_a_number` in the example blow).
2. Or, an exception that was thrown "behind the scene" while evaluating a contract assertion (e.g., the assertion calls an STL algorithm and that throws `std::bad_alloc`).
3. Or, an exception automatically thrown by this library in case a contract assertion is evaluated to be false (these exceptions are always `contract::broken` objects and they contained detailed information about the contract assertion that was evaluated to be false: file name, line number, assertion number, and assertion code). [56]

In all these cases the contract assertion is considered failed because it was not evaluated to be true (and that is the case not only for assertions that are evaluated to be false but also for assertions that throw exceptions while being evaluated). [57] The customized contract broken handlers can re-throw the active exception and catch it so to inspect it for logging or for any other customized behaviour that might need to be implemented by the programmers.

The following example customizes the contract broken handlers to throw exceptions (both user-defined exceptions like `not_a_number` and the `contract::broken` exception that the library automatically throws in case of a contract assertion failure). However, the customized handlers never throw from within a destructor to comply with STL exception safety requirements (see also `broken_handler_sqrt.cpp`): [58]

[55] **Rationale.** Exceptions are used to signal a contract assertion failure because it is not possible to directly call the contract broken handler (`contract::precondition_broken`, etc) instead of throwing the exception in oder to properly implement subcontracting. For example, if an overriding precondition fails but the overridden precondition is true then the library will not call the broken handler even if the overriding precondition threw an exception (as required by subcontracting). Contract assertion failures are critical error conditions so it is actually natural that the library uses exceptions to signal them (note that the library will always handle the exceptions by calling the contract broken handlers which terminate the program by default so the fact that the library signals contract assertion failures using exceptions does not imply that an exception will be thrown if a contract assertion fails, that is entirely decided by the implementation of the broken handlers).
[56] The assertion number is meaningful only within a specific handler. For example, if assertion number `2` failed within the class invariant broken handler that mean that the class invariant number `2` failed. Therefore, the assertion number is not useful if a single handler is programmed for all types of contracts because such an handler can no longer distinguish between class invariants, preconditions, postconditions, etc.
[57] **Rationale.** Threating an exception thrown while evaluating an assertion as a failure of the contract assertion is a clear policy established by this library under the principle that a contract assertion fails unless it is evaluated to be true. [N1962] does not completely clarify this point.
[58] In this example, contract failures from destructors are logged and then simply ignored. That might be acceptable in this simple example but it is probably a very bad idea in general.

```
struct not_a_number {}; // User defined exception.

CONTRACT_FUNCTION(
    double (sqrt) ( double x )
        precondition(
            x >= 0.0 ? true : throw not_a_number() // Throw user exception.
        )
        postcondition(
            auto root = return,
            root * root == x // Failure throws `contract::broken` exception.
        )
) {
    return 0.0; // Intentionally incorrect to fail postcondition.
}

void throwing_handler ( contract::from const& context )
{
    // Failure handlers always called with active exception that failed the
    // contract, so re-throw it to catch it below for logging.
    try {
        throw;
    } catch(contract::broken& failure) {
        std::cerr << failure.file_name() << "(" << failure.line_number() <<
                "): contract assertion \"" << failure.assertion_code() <<
                "\" failed " << std::endl;
    } catch(std::exception& failure) {
        std::cerr << "contract failed: " << failure.what() << std::endl;
    } catch(...) {
        std::cerr << "contract failed with unknown error" << std::endl;
    }

    // Cannot throw from destructors to comply with STL exception safety,
    // otherwise re-throw active exception that failed the contract.
    if(context == contract::FROM_DESTRUCTOR)
        std::cerr << "Ignoring destructor contract failure (probably "
            << "something bad has happened)" << std::endl;
    else throw; // Never terminates.
}

int main ( void )
{
    // Customize contract broken handlers to throw exceptions instead of
    // terminating the program (default).
    contract::set_precondition_broken(&throwing_handler);
    contract::set_postcondition_broken(&throwing_handler);

#ifndef CONTRACT_CONFIG_NO_PRECONDITIONS
    bool pre = false;
    try {
        sqrt(-1.0); // Fails precondition.
    } catch(not_a_number&) {
        pre = true;
        std::clog << "Ignoring not-a-number exception" << std::endl;
    }
    BOOST_TEST(pre);
#endif

#ifndef CONTRACT_CONFIG_NO_POSTCONDITIONS
    bool post = false;
    try {
        sqrt(4.0); // Fails postcondition.
    } catch(...) {
        post = true;
```

```
        std::clog << "Unable to calculate square root" << std::endl;
    }
    BOOST_TEST(post);
#endif

    return boost::report_errors();
}
```

Note how the ternary operator `:?` can be used to program assertions that throw exceptions on failure:

```
x >= 0.0 ? true : throw not_a_number
```

Then the contract broken handlers are customized to propagate the exceptions instead of handling them by calling `std::terminate` (default behaviour).

See also `contract_failure.cpp` for a more complex example that redefines the contract broken handlers to throw exceptions in order to automatically check that the correct contract broken handlers are called when assertions fail in accordance with the call semantics explained in the Contract Programming Overview section.

Virtual Specifiers

This section explains how to use this library to program virtual specifiers `final`, `override`, and `new`. Virtual specifiers are part of the program specifications because they enforce inheritance constraints at compile-time and therefore they are within the scope of this library.

Important

This library implements virtual specifiers for C++03 and without using any C++11 feature. [59] Obviously, virtual specifiers are supported only if both the base and derived classes and member functions in question are declared using this library macros (otherwise this library has no control over usual C++ declarations).

The examples presented in this section are rather simple (and they do not define virtual destructors for brevity). These examples only aim to illustrate the syntax of this library virtual specifiers and not to make a case about the utility of these specifiers. Virtual specifiers were adopted by C++11 and in some form are part of other languages like Java, programmers can refer to the web and to the Examples section for more interesting examples.

Final Classes

Final classes cannot be derived, otherwise the compiler will generated an error. This library allows to declare a class final by specifying `final` within the `CONTRACT_CLASS` macro after the class name, after template specialization types but before base classes if present (see also the Grammar section).

For example (see also `final_class.hpp` and `final_class.cpp`):

```
CONTRACT_CLASS(
    class (x) final // A final class.
) {
    CONTRACT_CLASS_INVARIANT( void )
};
```

If a derived class declared using `CONTRACT_CLASS` attempts to inherit from a final base class, the compiler will generate a compile-time error (this is true only if both the base and derived classes are declared using this library macro `CONTRACT_CLASS`). For example, consider the following derived class (see also `final_class_error.cpp`):

```
CONTRACT_CLASS(
    class (y) extends( public x ) // Correctly errors because `x` is final.
) {
    CONTRACT_CLASS_INVARIANT( void )
};
```

This will generate a compile-time error similar to the following because `x` is final (note that the number of the base class that violates the `final` specifier is reported so it is clear which base class is causing the error also in case of multiple-inheritance):

```
final_class_error.cpp:8 ... ERROR_cannot_extend_final_base_class_number_1 ...
```

Final class checks are performed at compile-time even when contracts are disabled (using `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, `CONTRACT_CONFIG_NO_CLASS_INVARIANTS`, etc).

Final Member Functions

Final member functions cannot be overridden by derived classes, otherwise the compiler will generate an error. This library allows to declare a member function final by specifying `final` within the `CONTRACT_FUNCTION` macro, after the cv-qualifier but before exception specifications if present (see also the Grammar section).

For example (see also `final_member.hpp` and `final_member.cpp`):

[59] This library declares special member functions to "tag" virtual traits of a given member function in a base class (if it is `virtual`, `final`, etc). Then template meta-programming introspection is used by the derived class to inspect virtual trait in its base classes and generate compiler errors if the virtual specifiers are not satisfied. These techniques do not use any C++11 feature however, future revisions of this library might use C++11 native support for virtual specifiers so to eliminate the extra compilation time required by template meta-programming introspection and correct a number of bugs associated with the current implementation of virtual specifiers in this library (see also Ticket 53, Ticket 55, and Ticket 56).

```
CONTRACT_CLASS(
    class (x)
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void )
    ) {}
};

CONTRACT_CLASS(
    class (y) extends( public x )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public void (f) ( void ) final // Final member function.
    ) {}
};
```

If a member function of a derived class is declared using `CONTRACT_FUNCTION` and attempts to override a final member function from one of the base classes, the compiler will generate a compile-time error (this is true only if both the overriding and overridden functions are declared using this library macro `CONTRACT_FUNCTION`). For example, consider the following derived class (see also `final_member_error.cpp`):

```
CONTRACT_CLASS(
    class (z) extends( public y )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public void (f) ( void ) // Correctly errors because `y::f` is final.
    ) {}
};
```

This will generate a compile-time error similar to the following because `y::f` is final (note that the number of the base class that violates the `final` specifier is reported so it is clear which overridden member function is causing the error also in case of multiple-inheritance):

```
final_member_error.cpp:13 ... ERROR_cannot_override_final_function_from_base_class_number_1 ...
```

In order to correctly handle overloaded functions, the overriding function and the final function must have the same name, parameter types, and cv-qualifier to generate this error.

Final member function checks are performed at compile-time but only when class invariants are enabled (`CONTRACT_CONFIG_NO_CLASS_INVARIANTS` is left not defined).

Overriding Member Functions

Overriding member functions must override a virtual member function from one or more of the base classes, otherwise the compiler will generate an error. This library allows to declare an overriding member function by specifying `override` within the `CONTRACT_FUNCTION` macro, after the cv-qualifier but before the `final` specifier if present (`override` and `final` can be used together, see also the Grammar section).

For example (see also `override_member.hpp` and `override_member.cpp`):

```
CONTRACT_CLASS(
    class (x)
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void ) // Virtual so it can be overridden.
    ) {}

    CONTRACT_FUNCTION(
        public void (g) ( void ) // Not virtual so it cannot be overridden.
    ) {}
};

CONTRACT_CLASS(
    class (y) extends( public x )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public void (f) ( void ) override // OK, overriding virtual `x::f`.
    ) {}
};
```

If a member function of a derived class is declared using `CONTRACT_FUNCTION` and attempts to override an non-virtual or non-existing member function from one of the base classes, the compiler will generate a compile-time error (this is true only if both the overriding and overridden functions are declared using this library macro `CONTRACT_FUNCTION`). For example, consider the following derived class (see also `override_member_error.cpp`):

```
CONTRACT_CLASS(
    class (z) extends( public x )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public void (g) ( void ) override // Correctly errors, cannot override.
    ) {}
};
```

This will generate a compile-time error similar to the following because `x::g` is not virtual:

```
override_member_error.cpp:12 ... ERROR_no_base_class_declares_matching_virtual_function_to_override_at_line_12 ...
```

In order to correctly handle overloaded functions, the overriding function and the overridden virtual function must have the same name, parameter types, and cv-qualifier to *not* generate this error.

Overriding member function checks are performed at compile-time but only if at least one between preconditions, postconditions, and class invariants is enabled (`CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS` and `CONTRACT_CONFIG_NO_CLASS_INVARIANTS` are not all defined at the same time).

New Member Functions

New member functions shall not override member functions from any of the base classes, otherwise the compiler will generate an error. This library allows to declare a new member function by specifying `new` within the `CONTRACT_FUNCTION` macro, after the cv-qualifier but before the `final` specifier if present (`new` and `final` can be used together, see also the Grammar section).

For example (see also `new_member.hpp` and `new_member.cpp`):

```
CONTRACT_CLASS(
    class (x)
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (f) ( void ) new // OK, no base so no `f` override.
    ) {}
};

CONTRACT_CLASS(
    class (y) extends( public x )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (g) ( void ) new // OK, there is no `x::g`.
    ) {}
};
```

If a new member function of a derived class is declared using `CONTRACT_FUNCTION` but it overrides an existing member function from one of the base classes, the compiler will generate a compile-time error (this is true only if both the overriding and overridden functions are declared using this library macro `CONTRACT_FUNCTION`). For example, consider the following derived class (see also `new_member_error.cpp`):

```
CONTRACT_CLASS(
    class (z) extends( public y )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION(
        public virtual void (g) ( void ) new // Correctly errors because `y::g`.
    ) {}
};
```

This will generate a compile-time error similar to the following because `f` was already declared in `x` (note that the number of the base class that violates the `new` specifier is reported so it is clear which overridden member function is causing the error also in case of multiple-inheritance):

```
new_member_error.cpp:12 ... ERROR_matching_virtual_function_already_declared_by_base_class_number_1 ...
```

In order to correctly handle overloaded functions, the new function and the overridden function must have the same name, parameter types, and cv-qualifier to generate this error.

New member function checks are performed at compile-time but only if at least one between preconditions, postconditions, and class invariants is enabled (`CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS` and `CONTRACT_CONFIG_NO_CLASS_INVARIANTS` are not all defined at the same time).

Concepts

This section explains how to use this library to check concepts. Concepts are part of the program specifications because they enforce requirements on generic types at compile-time and therefore they are within the scope of this library.

Concepts were proposed for addition to C++11 but they were unfortunately never adopted (see [N2081]). The concepts that are checked using this library need to be defined using the Boost.ConceptCheck library. This library does not add any extra functionality with respect to the concept checking features already provided by Boost.ConceptCheck but it allows to specify both concepts and contracts together within a unified syntax.

Class Templates

It is possible to list the concepts to check for class templates using `requires` within the `CONTRACT_CLASS` macro, after the template parameter declarations (see also the Grammar section):

```
template( template-parameter1, template-parameter2, ... ) requires( concept1, concept2, ... )
```

The following example requires that the generic type `T` specified for the class template `vector` is always `CopyConstructible` and it uses the `boost::CopyConstructible` concept defined by Boost.ConceptCheck (see also `class_member_concept_vector.hpp`, `class_member_concept_vector_error.cpp`, and `class_member_concept_vector.cpp`):

```
#include <boost/concept_check.hpp> // Boost.ConceptCheck predefined concepts.

CONTRACT_CLASS(
    template( typename T )
        requires( boost::CopyConstructible<T> ) // Check concepts.
    class (vector)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        size() <= capacity(),
        capacity() <= max_size()
    )

    // ...
```

If the class template `vector` is instantiated on a type `T` that is not `CopyConstructible`, the compiler will generate an error (with the usual format of Boost.ConceptCheck errors).

Function Templates

It is possible to list concepts to check for function templates using `requires` within the `CONTRACT_FUNCTION` macro, after the template parameter declarations (see also the Grammar section).

For example, for a constructor function template (see also `class_member_concept_vector.hpp`, `class_member_concept_vector_constructor_error.cpp`, and `class_member_concept_vector.cpp`):

```
CONTRACT_CONSTRUCTOR_TPL(
    public template( class Iterator )
        requires( boost::InputIterator<Iterator> ) // Check concepts.
    (vector) ( (Iterator) first, (Iterator) last )
        postcondition( std::distance(first, last) == int(size()) )
        initialize( vector_(first, last) )
) {}
```

And, for a member function template (see also `class_member_concept_vector.hpp`, `class_member_concept_vector_member_error.cpp`, and `class_member_concept_vector.cpp`):

```
CONTRACT_FUNCTION_TPL(
    public template( class Iterator )
        requires( boost::InputIterator<Iterator> ) // Check concepts.
    void (insert) ( (Iterator) where, (Iterator) first, (Iterator) last )
        precondition( size() + std::distance(first, last) < max_size() )
        postcondition(
            auto old_increased_size = CONTRACT_OLDOF
                    size() + std::distance(first, last),
            auto old_capacity = CONTRACT_OLDOF capacity(),
            size() == old_increased_size,
            capacity() >= old_capacity
        )
) {
    vector_.insert(where, first, last);
}
```

If the `vector` constructor template or the `vector::insert` member function template are instantiated on a type `Iterator` that is not an `InputIterator`, the compiler will generate an error (with the usual format of Boost.ConceptCheck errors).

So far we have used concepts that are predefined by Boost.ConceptCheck but it is also possible to check user-defined concepts as long as they are defined using Boost.ConceptCheck. For example, consider the following user-defined `Addable` concept (see also `free_concept_operator_preinc.hpp`, `free_concept_operator_preinc_error.cpp`, and `free_concept_operator_preinc.cpp`):

```
template< typename T >
struct Addable // New concept defined and implemented using Boost.ConceptCheck.
{
    BOOST_CONCEPT_USAGE(Addable)
    {
        return_type(x + y);
    }

private:
    void return_type ( T ) {}
    static T const& x;
    static T const& y;
};

CONTRACT_FUNCTION(
    template( typename T )
        requires(
            boost::Assignable<T>,   // Check both predefined and
            Addable<T>              // newly defined concepts.
        )
    (T&) operator(++)(preinc) ( (T&) value )
        postcondition(
            auto result = return,
            auto old_value = CONTRACT_OLDOF value,
            value == old_value + T(1), requires boost::has_equal_to<T>::value
            result == value, requires boost::has_equal_to<T>::value
        )
) {
    return value = value + T(1);
}
```

If this templated operator `++` is instantiated on a type `T` that is not `Assignable` or not `Addable`, the compiler will generate an error (with the usual format of Boost.ConceptCheck error).

Note that concepts can also be specified for free function templates, for free operator templates, and for member operator templates (as illustrated in some of the above examples).

Concept Definitions (Not Implemented)

Using the preprocessor parsing techniques introduced by the library, it should be possible to implement the following syntax to define concepts. Note how this syntax resembles the syntax proposed by [N2081] for adding concepts to C++11 ([N2081] and concepts were unfortunately not adopted by the standard committee).

```
CONTRACT_CONCEPT( // A concept definition.
    auto concept (LessThanComparable) ( typename T )
    (
        bool operator(<)(less) ( T , T )
    )
)

CONTRACT_CONCEPT( // Concepts with multiple parameters.
    auto concept (Convertible) ( typename T, typename U )
    (
        operator(U)(U_type) ( T const& )
    )
)

CONTRACT_CONCEPT( // Concept composition.
    concept (InputIterator) ( typename Iterator, typename Value )
    (
        requires Regular<Iterator>,

        (Value) operator(*)(deref) ( Iterator const& ),
        (Iterator&) operator(++)(preinc) ( Iterator& ),
        (Iterator) operator(++)(postinc) ( Iterator&, int )
    )
)

CONTRACT_CONCEPT( // Derived concepts.
    concept (ForwardIterator) ( typename Iterator, typename Value )
        extends( InputIterator<Iterator, Value> )
    (
        // Other requirements here...
    )
)

CONTRACT_CONCEPT( // Typenames within concepts.
    concept (InputIterator) ( typename Iterator )
    (
        typename value_type,
        typename reference,
        typename pointer,
        typename difference_type,

        requires Regular<Iterator>,
        requires (Convertible<reference, value_type>),

        (reference) operator(*)(deref) ( const Iterator& ),
        (Iterator&) operator(++)(preinc) ( Iterator& ),
        (Iterator) operator(++)(postinc) ( Iterator&, int )
    )
)

CONTRACT_CONCEPT( // Concept maps.
    concept_map (InputIterator<char*>)
    (
        typedef char value_type,
        typedef (char&) reference,
        typedef (char*) pointer,
        typedef ptrdiff_t difference_type
    )
)

CONTRACT_CONCEPT( // Concept maps can be templated.
    template( typename T )
    concept_map (InputIterator) ( T* )
```

```
    (
        typedef (T&) value_type,
        typedef (T&) reference,
        typedef (T*) pointer,
        typedef ptrdiff_t difference_type
    )
)

CONTRACT_CONCEPT( // Concept maps as mini-types.
    concept (Stack) ( typename X )
    (
        typename value_type,

        void (push) ( X&, value_type const& ),
        void (pop) ( X& ),
        (value_type) (top) ( X const& ),
        bool (empty) ( X const& )
    )
)

CONTRACT_CONCEPT( // Concept maps as mini-types (specializations).
    template( typename T )
    concept_map (Stack) ( std::vector<T> )
    (
        typedef (T) value_type,

        void (push) ( (std::vector<T>&) v, (T const&) x )
        (
            v.push_back(x);
        ),

        void (pop) ( (std::vector<T>&) v )
        (
            v.pop_back();
        ),

        (T) (top) ( (std::vector<T> const&) v )
        (
            return v.back();
        ),

        bool (empty) ( (std::vector<T> const&) v )
        (
            return v.emtpy();
        )
    )
)

CONTRACT_CONCEPT( // Axioms.
    concept (Semigroup) ( typename Op, typename T )
        extends( CopyConstructible<T> )
    (
        (T) operator(())(call) ( Op, T, T ),

        axiom (Associativity) (  (Op) op, (T) x, (T) y, (T) z )
        (
            op(x, op(y, z)) == op(op(x, y), z);
        )
    )
)
```

Note that:

1. Parenthesis around function parameter types can be always allowed but they should be required only when the parameter name is also specified.

2. The function bodies need to be specified within the macros and that will make compiler errors very hard to use (because the errors will all refer to the same line number, the line number on which the `CONTRACT_CONCEPT` macro expanded). However, concept definitions, including possible function bodies, might be simple enough for this not to be a major issue at least in the most common cases (the authors do not have enough experience with programming concept definitions to truly assess this).

The authors *think* this syntax can be implemented and parsed using the preprocessor however the authors have not tried to implement this yet. If this syntax can indeed be implemented, it should then be investigated if the actual concept definitions can be programmed from the parsed concept traits using C++ (this seems possible at least for C++11 based on some work done for the Generic library).

The authors recognize that implementing concept definitions would be a nice addition to this library (again, concepts are parts of the program specifications, they are contracts on the type system that are checked at compile-time, so both concept checking and concept definition are within the scope of this library). However, at the moment there are no concrete plans for extending this library with the concept definitions (see also Ticket 49).

The following is a side-by-side comparison of this possible concept definition syntax with the syntax proposed by [N2081]:

Possible Extension of This Library (not implemented)

```
CONTRACT_CONCEPT(
    concept (ArithmeticLike) ( typename T )
        extends(
            Regular<T>,
            LessThanComparable<T>,
            HasUnaryPlus<T>,
            HasNegate<T>,
            (HasPlus<T, T>),
            (HasMinus<T, T>)
            // ...
        )
    (
        explicit constructor(T, T) ( intmax_t),
        explicit constructor(T, T) ( uintmax_t ),
        explicit constructor(T, T) ( long double ),

        requires (Convertible<typename HasUnaryPlus<T>::result_type, T>)
        // ...
    )
)

CONTRACT_CONCEPT(
    auto concept (HasFind) ( typename T )
    (
        typename key_type, as typename T::key_type,
        typename mapped_type,

        (std::pair<key_type, mapped_type>) (find) ( T const&, key_type const& )
    )
)

CONTRACT_CONCEPT(
    auto concept (HasDereference) ( typename T )
    (
        typename result_type,

        (result_type) operator(*)(deref) ( T& ),
        (result_type) operator(*)(deref) ( T&& )
    )
)

CONTRACT_CONCEPT(
    auto concept (IdentityOf) ( typename T )
    (
        typename type, as T,
        requires (SameType<type, T>)
    )
)

CONTRACT_CONCEPT(
    auto concept (MemberFunctionRequirements) ( typename T )
    (
        constructor(T, T) ( int a, float b ),
        destructor(T, ~T) ( void ),
        void member(T, f) ( void ) const
    )
)

CONTRACT_CONCEPT(
    auto concept (HasEqualTo) ( typename T, typename U )
```

[N2081] Proposal (not part of C++)

```
// Extra spaces, newlines, etc used to align text with this library code.
concept ArithmeticLike < typename T >
    :
        Regular<T>,
        LessThanComparable<T>,
        HasUnaryPlus<T>,
        HasNegate<T>,
        HasPlus<T, T>,
        HasMinus<T, T>
        // ...

{
    explicit T::T ( intmax_t ) ;
    explicit T::T ( uintmax_t ) ;
    explicit T::T ( long double ) ;

    requires Convertible<typename HasUnaryPlus<T>::result_type, T> ;
    // ...
}



auto concept HasFind < typename T >
{
    typename key_type = typename T::key_type ;
    typename mapped_type ;

    std::pair<key_type, mapped_type> find ( T const&, key_type const& ) ;
}



auto concept HasDereference < typename T >
{
    typename result_type ;

    result_type operator* ( T& ) ;
    result_type operator* ( T&& ) ;
}



auto concept IdentityOf < typename T >
{
    typename type = T ;
    requires SameType<type, T> ;
}



auto concept MemberFunctionRequirements < typename T >
{
    T::T ( int a, float b ) ;
    T::~T ( void ) ;
    void T::f ( void ) const ;
}



auto concept HasEqualTo < typename T, typename U >
```

Possible Extension of This Library (not implemented)

```
    (
        bool operator(==)(equal) ( T const&, U const& )
    )
)

CONTRACT_CONCEPT(
    auto concept (ExplicitlyConvertible) ( typename T, typename U )
    (
        explicit operator(U)(U_type) ( T const& )
    )
)

CONTRACT_CONCEPT(
    auto concept (Convertible) ( typename T, typename U )
        extends( (ExplicitlyConvetible<T, U>) )
    (
        operator(U)(U_type) ( T const& )
    )
)

CONTRACT_CONCEPT(
    concept (True) ( bool Value )
    (
        void // Empty concept definition.
    )
)

CONTRACT_CONCEPT(
    concept_map (True) ( true )
    (
        void // Empty concept definition.
    )
)

CONTRACT_CONCEPT(
    auto concept (IsEven) ( intmax_t Value )
    (
        requires True<V % 2 == 0>
    )
)

CONTRACT_CONCEPT(
    auto concept (CopyConstructible) ( typename T )
        extends( MoveConstructible<T>, (Constructible<T, T const&>) )
    (
        axiom (CopyPreservation) ( (T) x )
        (
            T(x) == x;
        )
    )
)

CONTRACT_CONCEPT(
    concept (Iterator) ( typename X ) extends( Semiregular<X> )
    (
        typename(MoveConstructible) reference, as typename X::reference,
        typename(MoveConstructible) postinc_result,
        requires HasDereference<postinc_result>,

        (reference) operator(*)(deref) ( X& ),
```

[N2081] Proposal (not part of C++)

```
{
    bool operator== ( T const&, U const& ) ;
}



auto concept ExplicitlyConvertible < typename T, typename U >
{
    explicit operator U ( T const& ) ;
}



auto concept Convertible < typename T, typename U >
    : ExplicitlyConvetible<T, U>
{
    operator U ( T const& ) ;
}



concept True < bool Value >
{
    // No operation.
}



concept_map True < true >
{
    // No operation.
}



auto concept IsEven < intmax_t Value >
{
    requires True<V % 2 == 0> ;
}



auto concept CopyConstructible < typename T >
    : MoveConstructible<T>, Constructible<T, T const&>
{
    axiom CopyPreservation ( T x )
    {
        T(x) == x;
    }
}



concept Iterator < typename X > : Semiregular<X>
{
    MoveConstructible reference = typename X::reference ;
    MoveConstructible postinc_result ;
    requires HasDereference<postinc_result> ;

    reference operator* ( X& ) ;
```

Possible Extension of This Library (not implemented)

```
        (reference) operator(*)(deref) ( X&& ),
        (X&) operator(++)(preinc) ( X& ),
        (postinc_result) operator(++)(postinc) ( X&, int )
    )
)

CONTRACT_CONCEPT(
    concept (RandomAccessIterator) ( typename X )
        extends( BidirectionalIterator<X>, LessThanComparable<X> )
    (
        typename(MoveConstructible) reference,
        requires (Convertible<reference, value_type const&>),

        (X&) operator(+=)(inc) ( X&, difference_type ),

        (X) operator(+)(add) ( (X const&) x, (difference_type) n )
        (
            X y(x);
            y += n;
            return y;
        ),

        (X) operator(+)(add) ( (difference_type) n, (X const&) x )
        (
            X y(x);
            y += n;
            return y;
        ),

        (X) operator(-=)(dec) ( X&, difference_type ),

        (X) operator(-)(sub) ( (X const&) x, (difference_type) n )
        (
            X y(x);
            y -= n;
            return y;
        ),

        (difference_type) operator(-)(sub) ( X const&, X const& ),

        (reference) operator([])(at) ( X const&, difference_type )
    )
)

CONTRACT_CONCEPT(
    template( typename(ObjectType) T )
    concept_map (RandomAccessIterator) ( T* )
    (
        typedef (T) value_type,
        typedef ptrdiff_t difference_type,
        typedef (T&) reference,
        typedef (T*) pointer
    )
)
```

[N2081] Proposal (not part of C++)

```
    reference operator* ( X&& ) ;
    X& operator++ ( X& ) ;
    postinc_result operator++ ( X&, int ) ;
}



concept RandomAccessIterator < typename X >
    : BidirectionalIterator<X>, LessThanComparable<X>
{
    MoveConstructible reference ;
    requires Convertible<reference, value_type const&> ;

    X& operator+= ( X&, difference_type ) ;

    X operator+ ( X const& x, difference_type n )
    {
        X y(x);
        y += n;
        return y;
    }

    X operator+ ( difference_type n, X const& x )
    {
        X y(x);
        x += n;
        return y;
    }

    X operator-= ( X&, difference_type ) ;

    X operator- ( X const& x, difference_type n )
    {
        X y(x);
        y -= n;
        return y;
    }

    difference_type operator- ( X const&, X const& ) ;

    reference operator[] ( X const&, difference_type ) ;
}



template< ObjectType T >
concept_map RandomAccessIterator < T* >
{
    typedef T value_type ;
    typedef ptrdiff_t difference_type ;
    typedef T& reference ;
    typedef T* pointer ;
}
```

Note that:

1. Extra wrapping parenthesis are used when expressions contain unwrapped commas `,` or leading symbols.

2. The specifiers `constructor`, `destructor`, and `member` follow the same syntax as the `CONTRACT_CONSTRUCTOR_BODY` and `CONTRACT_DESTRUCTOR_BODY`, and they serve a purpose similar to these macros in naming the constructors, destructors, and member functions outside class declarations.

If concept definitions were added to this library, concepts would still be checked using the `requires` specifier in class and function declarations as we have seen so far (it might even be possible for concepts defines using Boost.ConceptCheck to be still specified using some type tagging of this library concept types to internally distinguish between Boost.ConceptCheck concepts and concepts define using this library). Furthermore, this library could provide all the standard concepts defined in [N2914] in an header file `contract/std/concept.hpp`.

Named Parameters

This section explains how to use this library to program named parameters and deduced named parameters which are used instead of C++ usual positional parameters to specify arguments to function calls. Parameter names are part of the program specifications and therefore named parameters are within the scope of this library.

This library internally uses Boost.Parameter to implement named parameters. This library only offers an easier syntax for constructor and class template named parameters with respect the functionality already provided by Boost.Parameter, but this library allows to specify named parameters, concepts, and contracts all together within a unified syntax.

Overview

In C++, parameters are identified with respect to their declaration position within the parameter list: The first call *argument* (i.e., the specific value assigned to the formal parameter at the call site) maps to the first declared parameter, the second call argument maps to the second declared parameter, etc. For example:

```
window* make_window (
        char const& name,
        size_t width = 10,
        bool moveable = true,
        bool visible = true
    )
;
```

This protocol is sufficient when there is at most one parameter with a useful default value but when there are even a few useful default parameters the positional interface becomes difficult to use at the calling site. For example, in the following call we need to repeat the default parameter value `10`:

```
bool const unmoveable = false;
make_window("warning", 10, unmoveable); // Positional (usual).
```

Furthermore, it would be easier to understand the meaning of the arguments at the calling site if the parameters could be referred by their names. For example, in order to understand if the following window is moveable and invisible, or unmoveable and visible, programmers at the calling site need to remember the order of the parameter declarations:

```
make_window("note", 10, true, false); // Positional (usual).
```

These issues would be solved if the parameter names could be referred at the calling site and that is what *named parameters* allow to do:

```
make_window("warning", moveable = false); // Named.
make_widnow("note", visible = false);
```

Named parameters were proposed for addition to early versions of C++ but the proposal was never accepted (see “keyword arguments” in [Stroustrup94]). Other programming languages support named parameters (Ada, Python, etc).

Furthermore, *deduced named parameters* are named parameters that can be passed in any position and without supplying their names. These are useful when functions have parameters that can be uniquely identified (or *deduced*) based on the argument types of the function call. For example, the `name` parameter is such a parameter, it can be deduced because no other argument, if valid, can be reasonably converted into a `char const*`. Therefore, with a deduced parameter interface, we could pass the window name in any position without causing any ambiguity (with an appropriately designed deduced parameter interface, the caller might not even have to remember the actual parameter names):

```
make_window(moveable = false, "warning"); // Deduced.
make_window("note", visible = false);
```

The same reasoning applies to named template parameters and deduced named template parameters:

```
template< typename ValueType, bool Shared = true >
class smart_ptr ;

smart_ptr<std::string, false> p1;                       // Positional (usual).
smart_ptr<false, std::string> p2;                       // Deduced.
smart_ptr<Shared = false, ValueType = std::string> p3;  // Named.
```

Named Function Parameters

To show how to program named parameters, we will program a named parameter interface for the Depth First Search (DFS) algorithm of the Boost.Graph library:

```
template<
    class Graph,
    class Visitor,
    class IndexMap,
    class ColorMap
> requires
    is_incidence_and_vertex_list_graph<Graph>,
    is_integral_property_map_of_key<IndexMap, vertext_descriptor<Graph> >,
    is_property_map_of_key<ColorMap, vertex_descriptor<Graph> >
void depth_first_search (
        Graph const& graph,
        Visitor const& visitor = dfs_visitor<>(),
        vertex_descriptor<Graph> const& root_vertex =
                *vertices(graph).first,
        IndexMap const& index_map = get(vertex_index, graph),
        ColorMap& color_map =
                default_color_map(num_vertices(graph), index_map)
    )
;
```

This is non-valid C++ code but it can be considered pseudo-code to illustrate the function interface that we need to program. The concept syntax `requires` was "borrowed" here to express parameter type requirements but a failure to satisfy such requirements is intended to fail the function call resolution and not to cause a concept error as usual.

`graph` This is an input parameter (`const&`), its type must match a model of both `IndicenceGraph` and `VertexListGraph`, and it has no default value (required parameter).

`visitor` This is an input parameter (`const&`), its type has no requirement, and its value defaults to a DFS visitor (optional parameter).

`root_vertex` This is an input parameter (`const&`), its type must be `vertex_descriptor<Graph>`, and its value defaults to the first vertex of the specified graph (optional parameter).

`index_map` This is an input parameter (`const&`), its type must match a model of `IntegralPropertyMap` with key type `vertex_descriptor<Graph>`, its value defaults to the vertex index of the specified graph (optional parameter).

`color_map` This is an input and output parameter (non-constant reference `&`), its type must match a model of `PropertyMap` with key type `vertex_descriptor<Graph>`, its value defaults to a map with size equal to the number of vertices of the specified graph and created from the specified `index_map` (optional parameter).

Parameter Identifiers

First of all, it is necessary to declare special elements that will serve as the parameter and argument identifiers (see also `named_params_dfs.cpp`):

```
namespace graphs {

CONTRACT_PARAMETER(graph)
CONTRACT_PARAMETER(visitor)
CONTRACT_PARAMETER(root_vertex)
CONTRACT_PARAMETER(index_map)
CONTRACT_PARAMETER(color_map)

} // namespace
```

For example, the first `CONTRACT_PARAMETER` macro declares an identifier `graph` that is used as the parameter name in the function declaration and also an identifier `graph_` (note the trailing underscore _) that is used to name the argument at the call site.

Note

By default, argument identifiers are differentiated from parameter identifiers using a trailing underscore _ to avoid name clashes (see also the Parameter Identifiers section). Furthermore, it is recommended to always declare these identifiers within a namespace to avoid name clashes (because different functions in different sections of the code might very well use the same parameter names so namespaces can be used to control overlapping names).

For example, for a library named Graphs, all named parameter identifiers could be declared under the `graphs` namespace and maybe in a single header `graphs_params.hpp`. Many different functions within `graphs` (for example `graphs::depth_first_search` and `graphs::breadth_first_search`) will share a number of common parameter names (for example a parameter named `graph` with parameter identifier `graphs::graph` and argument identifier `graphs::graph_`). All these named parameters will be defined in one places in `graphs_params.hpp` and within the `graphs` namespace so programmers can conveniently control and avoid named clashes.

Function Declaration

The function is declared using the following syntax (see also `named_params_dfs.cpp`):

```
namespace graphs {

CONTRACT_FUNCTION(
    void (depth_first_search) (
            // Required parameter (no default value) with requirement predicate.
            in requires(is_incidence_and_vertex_list_graph<boost::mpl::_>)
                    graph,
            // Optional (default value) of any type (auto).
            in auto visitor, default boost::dfs_visitor<>(),
            // Exact type specified.
            in (vertex_descriptor<CONTRACT_PARAMETER_TYPEOF(graph)>)
                    root_vertex, default (*boost::vertices(graph).first),
            // Any type matching requirement predicate (unary meta-function).
            in requires(
                is_integral_property_map_of_key<
                      boost::mpl::_
                    , vertex_descriptor<CONTRACT_PARAMETER_TYPEOF(graph)>
                >
            ) index_map, default boost::get(boost::vertex_index, graph),
            // In-out parameter.
            in out requires(
                is_property_map_of_key<
                      boost::mpl::_
                    , vertex_descriptor<CONTRACT_PARAMETER_TYPEOF(graph)>
                >
            ) color_map, default default_color_map(boost::num_vertices(graph),
                    index_map)
        )
        // No `precondition(...)` or `postcondition(...)` in this example.
) {
    // Also function definition can use `PARAMETER_TYPEOF`.
    typedef CONTRACT_PARAMETER_TYPEOF(graph)& graph_ref; // Unused by example.
    boost::depth_first_search(graph, boost::visitor(visitor).
            color_map(color_map).root_vertex(root_vertex).
            vertex_index_map(index_map));
}

} // namespace
```

This example required no preconditions and no postconditions.

In, In Out, and Out Parameters

Input parameters (i.e., constant references `const&`) are prefixed by the specifier `in` (see also the Grammar section):

```
in ... graph // Input parameter.
```

Input and output parameters (i.e., non-constant reference `&`) are prefixed by both specifiers `in out` (in this order, see also the Grammar section):

```
in out ... color_map // Input-output parameter.
```

Output parameters can be prefixed just by the specifier `out` but their implementation is equivalent to using the specifiers `in out` (i.e., non-constant reference `&`). [60] Note that a named parameter must always be specified either `in`, `in out`, or `out` (and that is what distinguish syntactically a named parameter from a positional parameter; this use of `in`, `in out`, and `out` resembles Ada's syntax). Named and positional parameters cannot be used together in the same function declaration.

Parameter Types

Exact parameter types are specified just before the parameter name and within parenthesis (unless they are fundamental types containing no symbol, see also the Grammar section):

```
in (vertex_descriptor<CONTRACT_PARAMETER_TYPEOF(graph)>) root_vertex // Exact parameter type requirement.
```

Generic parameter types (i.e., parameter types with no requirement, same as usual type template parameters) are specified using `auto` instead of the parameter type (see also the Grammar section):

```
in auto visitor // No parameter type requirement.
```

Predicate parameter type requirements are specified using `requires(`*`unary-boolean-metafunction`*`)` instead of the parameter type, the type of the argument specified by the function call will be required to match the specified unary boolean meta-function in order for the call to be resolved (see also the Grammar section):

```
in requires(is_incidence_and_vertex_list_graph<boost::mpl::_>) graph // Predicate parameter type requirement.
```

The placeholder `boost::mpl::_` makes this expression a unary meta-function (see Boost.MPL) and the library will substitute such a placeholder with the argument type specified at the call site when resolving the function call.

Default Parameter Values

Default parameter values are specified using `, default` *`default-parameter-value`* right after the parameter declaration (see also the Grammar section):

```
in auto visitor, default boost::dfs_visitor<>()
```

Default parameter values are not evaluated and their types are not even instantiated if an actual argument is specified by the function call.

Note

The type and value of a parameters can be used within the declaration of other parameters. The type of a named parameter can be accessed using the `CONTRACT_PARAMETER_TYPEOF` macro (this is especially useful for generic parameter types and for parameter types with predicate requirements because these types are not known until the function call is resolved).

Function Call

The `graphs::depth_first_search` function can be called using argument identifiers to name its parameters (see also `named_params_dfs.cpp`):

```
graphs::depth_first_search(graphs::visitor_ = vis, graphs::graph_ = g);
```

Deduced Function Parameters

To show how to program deduced named parameters, we program the parameter interface for the `def` function of the Boost.Python library (see also `deduced_params_pydef.cpp`):

[60] **Rationale.** C++ cannot express the semantics that an output parameter should be written and never read within a function because references can always be both written and read.

```
namespace py {

CONTRACT_PARAMETER(name)
CONTRACT_PARAMETER(func)
CONTRACT_PARAMETER(docstring)
CONTRACT_PARAMETER(keywords)
CONTRACT_PARAMETER(policies)

CONTRACT_FUNCTION(
    void (def) (
            // Non-deduced (and also required) named parameters.
            in (char const*) name,
            in auto func,
            // Deduced (and also optional) named parameters.
            deduce in (char const*) docstring, default (""),
            deduce in requires(is_keyword_expression<boost::mpl::_>) keywords,
                    default no_keywords(),
            // Type requirements also used to uniquely deduce parameters.
            deduce in requires(
                boost::mpl::not_<
                    boost::mpl::or_<
                          boost::is_convertible<boost::mpl::_, char const*>
                        , is_keyword_expression<boost::mpl::_>
                    >
                >
            ) policies, default boost::python::default_call_policies()
        // Concepts for named parameters.
        ) requires( CallPolicies<CONTRACT_PARAMETER_TYPEOF(policies)> )
        precondition( is_identifier(name) ) // Contracts.
) {
    boost::python::def(name, func, docstring, keywords, policies);
}

} // namespace
```

Also a precondition was specified in this example. (Concepts for named parameters are explained later in this section.)

Deduced parameters are named parameter that are prefixed by the `deduce` specifier (see also the Grammar section):

```
deduce in (char const*) docstring // Deduced input parameter.
```

When calling the function `py::def` only two arguments are required `name` and `func`. The association between any addition argument and its parameter can be deduced based on the types of the arguments specified by the function call (so the caller is neither required to remember the parameter positions nor to explicitly specify the parameter names for these arguments).

For example, the first two calls below are equivalent and if programmers need to pass a `policy` argument that is also, for some reason, convertible to `char const*`, they can always specify the parameter name as in the third call below (see also `deduced_params_pydef.cpp`):

```
py::def("f", print<'f'>, nonstring_policies, "Documentation for f");
py::def("g", print<'g'>, "Documentation for g", nonstring_policies);
py::def("h", print<'h'>, py::policies_ = string_policies,
        "Documentation for h");
```

Member Function Parameters

The same syntax is used to specify named and deduced parameters for member functions (see also `member_named_params_callable2.cpp`):

```
namespace calls {

CONTRACT_PARAMETER(arg1)
CONTRACT_PARAMETER(arg2)

CONTRACT_CLASS(
    struct (callable2)
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_FUNCTION( // Member function with named parameters.
        public void (call) ( in int arg1, in int arg2 )
    ) ; // Body definition can be separated...

    CONTRACT_FUNCTION( // Constant member with named parameters.
        public void (call) ( in int arg1, in int arg2 ) const
    ) {
        total = arg1 + arg2;
    }

    CONTRACT_FUNCTION( // Static member with named parameters.
        public static void (static_call) ( in int arg1, in int arg2 )
    ) {
        total = arg1 + arg2;
    }

    static int total;
};

// ... but body definition uses template parameters and special `BODY` macro.
template< typename Arg1, typename Arg2 >
void callable2::CONTRACT_PARAMETER_BODY(call)( Arg1 arg1, Arg2 arg2 )
{
    total = arg1 + arg2;
}

int callable2::total = 0;

} // namespace
```

No contracts were specified in this example.

Note

When the body definition of a function with named parameters is deferred from the function declaration, the body must be declared as a template function when it is defined and the special macro `CONTRACT_PARAMETER_BODY` must be used to name the function.

The same macro is used to name both free and member body functions because the class type is specified outside this macro. The constructor body definition cannot be deferred from the constructor declaration (because of the lack of delegating constructors in C++03). Destructors have no named parameters (because they have no parameter). Named parameters are not supported for operators (because of a Boost.Parameter bug). [61] Therefore the `CONTRACT_PARAMETER_BODY` macro is used only with free and member functions that are not operators. [62]

In this example, named parameters were specified for all `callable2::call` overloads and for the `static` member function `callable2:static_call`:

[61] **Rationale.** This library syntax supports named and deduced parameters for operators. However, Boost.Parameter does not compile when used with operators because of a bug (Boost.Parameter documentation claims support for operators but later revisions of Boost.Parameter no longer compile when operators are used). If Boost.Parameter were to be fixed to work with operators then this library named and deduced parameters should also work with operators.

[62] **Rationale.** A macro `CONTRACT_PARAMETER_BODY` different from `CONTRACT_MEMBER_BODY` and `CONTRACT_FREE_BODY` is necessary because named parameters will still be enabled even when contracts are turned off using `CONTRACT_CONFIG_NO_PRECONDITIONS`, etc.

```
calls::callable2::static_call(calls::arg2_ = 2, calls::arg1_ = 1);
BOOST_TEST(calls::callable2::total == 3);

calls::callable2 c;
c.call(calls::arg2_ = 3, calls::arg1_ = 4);
BOOST_TEST(calls::callable2::total == 7);

calls::callable2 const& cc = c;
cc.call(calls::arg2_ = 5, calls::arg1_ = 6);
BOOST_TEST(calls::callable2::total == 11);
```

Constructor Parameters

The same syntax is used to specify named and deduced parameters for constructors (see also `constructor_named_params_family.cpp`): [63]

```
namespace family {

CONTRACT_PARAMETER(name)
CONTRACT_PARAMETER(age)
CONTRACT_PARAMETER(relatives)
CONTRACT_PARAMETER(weight)

CONTRACT_CLASS(
    struct (member) extends( public person )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_CONSTRUCTOR(
        public (member) (
                in out auto name,
                in out auto age, //default 32,
                deduce out (std::vector<person>) relatives,
                deduce in requires(
                    boost::is_convertible<boost::mpl::_, double>
                ) weight //, default(75.0)
            ) // Preconditions, body, etc access arguments as usual...
            precondition( std::string(name) != "", age >= 0, weight >= 0.0 )
            postcondition(
                auto old_relatives_size = CONTRACT_OLDOF relatives.size(),
                relatives.size() == old_relatives_size + 1
            )
            initialize( // ... but initializers must use the `..._ARG` macro.
                person(CONTRACT_CONSTRUCTOR_ARG(name),
                       CONTRACT_CONSTRUCTOR_ARG(age),
                       CONTRACT_CONSTRUCTOR_ARG(weight)),
                family_(CONTRACT_CONSTRUCTOR_ARG(relatives))
            )
    ) {
        relatives.push_back(*this);
    }

    private: std::vector<person>& family_;
};

} // namespace
```

Both preconditions and postconditions were specified for the constructor.

[63] Arguably, this library offers a better syntax than Boost.Parameter for constructor named and deduced parameters (because Boost.Parameter requires boiler-plate code and an extra base class in order to handle constructors).

Important

Member initializers must access named and deduced parameters using the special macro `CONTRACT_CONSTRUCTOR_ARG` (otherwise the compiler will generate an error). [64]

Named and deduced parameters can be used to specify the arguments when constructing the object:

```
std::vector<family::person> r;/*
family::member m1(family::name_ = "Mary", family::relatives_ = r);
```

Class Template Parameters

Only class templates are discussed in this section because function templates are effectively always declared when a function uses named or deduced parameters. In fact, a function with named or deduced parameters is always a function template (possibly with fully generic templated types when the named and deduced parameters types are specified `auto`).

To show how to program named and deduced template parameters, we will program a named parameter interface for the `class_` class template of the Boost.Python library:

```
template<
    class ClassType,
    class Bases = bases<>,
    typename HeldType = not_sepcified,
    typename Copyable = not_specified
> requires
    is_class<ClassType>
class class_ ;
```

ClassType — This type parameter must be a class and it has no default value (required parameter).

Bases — This type parameter must be a specialization of `boost::python::bases` specifying base classes if any (see `boost::python::detail::specifies_bases`) and its value defaults to `boost::python::bases<>`.

HeldType — This type parameter has no constraint and it is not specified by default (see `boost::python::detail::not_specified`).

Copyable — This type parameter must be `boost::noncopyable` if specified and it is not specified by default (see `boost::python::detail::not_specified`).

Important

Unfortunately, named and deduced template parameters can only handle type template parameters (i.e., value template parameters and template template parameters are not supported). [65]

First of all, it is necessary to declare elements that will serve as template parameter and argument identifiers (see also `named_template_params_pyclass.cpp`):

```
namespace py {

CONTRACT_TEMPLATE_PARAMETER(ClassType)
CONTRACT_TEMPLATE_PARAMETER(Bases)
CONTRACT_TEMPLATE_PARAMETER(HeldType)
CONTRACT_TEMPLATE_PARAMETER(Copyable)
CONTRACT_PARAMETER(name)

} // namespace
```

For example, the first `CONTRACT_TEMPLATE_PARAMETER` macro declares a specifier `ClassType` that can be used as the parameter name in the class template declaration and a specifier `ClassType_` (note the trailing underscore _) that can be used for the template argument at the template instantiation site.

[64] **Rationale.** This limitation comes from the lacks of delegating constructors in C++03 (as with all other limitations of this library related to member initializers).

[65] **Rationale.** This library only supports named and deduced type template parameter because it internally uses Boost.Parameter and Boost.Parameter does not support non-type template parameters. However, this library syntax would support named and deduced value template parameters and template template parameters if these were ever supported by Boost.Parameter (see template parameters within the Grammar section). Named and deduced value template parameters can be emulated by wrapping their values into types (using `boost::mpl::int_`, `boos::mpl::bool_`, etc) at the cost of a somewhat cumbersome syntax.

Note

By default, template argument identifiers are differentiated from template parameter identifiers using a trailing underscore _ to avoid name clashes (see also the Parameter Identifiers section). Furthermore, it is recommended to always declare these identifiers within a namespace to avoid name clashes (because different class templates in different sections of the code might very well use the same template parameter names so namespaces can be used to control overlapping names).

The class template is declared using a syntax similar to the one we have seen so far for named and deduced function parameters (see also `named_template_params_pyclass.cpp`): [66]

```
namespace py {

CONTRACT_CLASS(
    template(
        // Required named template parameter.
        in typename requires(boost::is_class<boost::mpl::_>) ClassType,
        // Optional deduced named template parameters with requirements.
        deduce in typename requires(
            boost::python::detail::specifies_bases<boost::mpl::_>
        ) Bases, default boost::python::bases<>,
        deduce in typename requires(
            boost::mpl::not_<
                boost::mpl::or_<
                      boost::python::detail::specifies_bases<boost::mpl::_>
                    , boost::is_same<boost::noncopyable, boost::mpl::_>
                >
            >
        ) HeldType, default boost::python::detail::not_specified,
        deduce in typename requires(
            boost::is_same<boost::noncopyable, boost::mpl::_>
        ) Copyable, default boost::python::detail::not_specified
        // Unfortunately, non-type named template parameters are not supported.
    // Concepts named template parameters.
    ) requires( boost::DefaultConstructible<ClassType> )
    class (class_)
        extends( (boost::python::class_<ClassType, Bases, HeldType, Copyable>) )
) {
    CONTRACT_CLASS_INVARIANT_TPL( void ) // Contracts.

    public: typedef ClassType class_type;
    public: typedef Bases bases;
    public: typedef HeldType held_type;
    public: typedef Copyable copyable;

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (class_) ( in (char const*) name )
            precondition( is_identifier(name) )
            initialize(
                (boost::python::class_<ClassType, Bases, HeldType, Copyable>(
                        CONTRACT_CONSTRUCTOR_ARG(name)))
            )
    ) {}
};

} // namespace
```

Note that template parameters are always specified using `in` because they are always input parameters (they are static types). Furthermore, `typename` or `class` must always follow the `in` specifier for type template parameters. [67] Predicate type requirements `requires(`*`unary-boolean-metafunction`*`)` are optional and they are specified right after `typename` or `class` if present. Default template parameter values are specified using `, default` *`default-parameter-value`* as usual. (See also the Grammar section.)

[66] Arguably, this library offers a better syntax than Boost.Parameter for named and deduced template parameters (because Boost.Parameter requires boiler-plate code and the additional argument packing types to be programmed manually).

[67] **Rationale.** This syntax was designed so it can support non-type template parameters if this library and Boost.Parameter were ever extended to support value template parameters and template template parameters (see template parameters in the Grammar section).

The class template `py::class_` can be instantiated using the template parameter identifiers to name the arguments (see also `named_template_params_pyclass.cpp`): [68]

```
// Named parameters.
py::class_<py::ClassType_<bx>, py::Copyable_<boost::noncopyable> >("bx");
py::class_<x, py::HeldType_<boost::shared_ptr<x> >,
        py::Bases_<boost::python::bases<bx> > >("x");
// Deduced parameters.
py::class_<by, boost::noncopyable>("by");
py::class_<y, boost::shared_ptr<y>, boost::python::bases<by> >(
        py::name_ = "y");
```

Concepts

As shown in the `py::class_` example above, it is possible to specify concepts for named and deduced template parameters. Concepts are specified using `requires` after the class template parameter declaration:

```
template( ... ) requires( concept1, concept2, ... )
class (class_) ...
```

As shown in the `py::def` example earlier in this section, it is also possible to specify concepts for named and deduced function parameters (because named and deduced function parameter can be generic, as function template parameters are, when their types are specified `auto` or via a predicate type requirement). However, in this case concepts are specified using `requires` after the function parameter list (because there is not `template` declaration for functions with named and deduced parameters):

```
void (def) ( ... ) requires( concept1, concept2, ... )
```

Note the following differences between specifying predicate type requirements and concepts for named and deduced parameters:

- If predicate type requirements are not met, the function or class template will be taken out from the set of possible function overloads or template specializations in the function call or template instantiation resolution (this might ultimately generate a compiler error but only if there is no other function overload or template specialization that can be used).
- If a concept check fails, the compiler will always generate a compiler error.

Parameter Identifiers

When using the `CONTRACT_PARAMETER` and `CONTRACT_TEMPLATE_PARAMETER` macros (see also the Grammar section):

1. It is possible to specify a namespace that will contain the parameter identifiers using `namespace(`*`parameter-identifier-namespace`*`)`.

2. It is also possible to specify the name of the argument identifier instead of using the automatic underscore _ postfix using `(`*`argument-identifier`*`)`.

The following example generates parameter identifiers within the `params` namespace, and argument identifiers `NumberArg` and `value_arg` instead of the default `Number_` and `value_` for the `Number` and `value` parameters respectively (see also `named_param_identifiers_positive.cpp`):

```
CONTRACT_TEMPLATE_PARAMETER( namespace(params) (NumberArg) Number )
CONTRACT_PARAMETER( namespace(params) (value_arg) value )
```

When the parameter identifier namespace is specified, it must be repeated in the named and deduced parameter declaration list (see also the Grammar section):

```
using namespace paraemeter-identifier-namespace, ... // named and deduced parameters follow
```

The authors find this syntax confusing so they recommend to never specify the parameter identifier namespace when using the `CONTRACT_PARAMETER` and `CONTRACT_TEMPLATE_PARAMETER` macros. [69] macro.] Instead programmers are encouraged to use these macros within an enclosing namespace as done by the rest of the examples in this documentation.

For example (see also `named_param_identifiers_positive.cpp`):

---

[68] This named and deduced parameter syntax is not entirely ideal because it uses angular parenthesis `py::ClassType_<bx>` instead of the more readable assignment operator `py::ClassType_ = bx` to name the arguments (but that is how Boost.Parameter is implemented and also this library authors see no way to implement the assignment operator syntax for named and deduced template parameters).

[69] **Rationale.** The ability to specify the parameter identifier (or tag) namespace is offered because it is offered by Boost.Parameter and it gives programmers greater control. However, it should be rarely if ever needed because this library provides the [macro CONTRACT_PARAMETER_TYPEOF

```
CONTRACT_CLASS(
    // Named template parameters must specify `params` namespace.
    template( using namespace params, in typename Number )
    struct (positive)
) {
    CONTRACT_CLASS_INVARIANT_TPL( get() > 0 )

    CONTRACT_CONSTRUCTOR_TPL(
        // Named parameters must specify `params` namespace.
        public (positive) ( using namespace params, in (Number) value )
            precondition( value > 0 )
            postcondition( get() == value )
            initialize( value_(CONTRACT_CONSTRUCTOR_ARG(value)) )
    ) { }

    CONTRACT_FUNCTION_TPL(
        public (Number const&) (get) ( void ) const
    ) {
        return value_;
    }

    private: Number value_;
};
```

Class invariants, preconditions, and postconditions were also specified in this example.

At the template instantiation and function call site, the specified argument identifiers `NumberArg` and `value_arg` can be used instead of the usual trailing underscore (see also `named_param_identifiers_positive.cpp`):

```
positive<NumberArg<int> > n(value_arg = 123); // Use ...Arg and ..._arg.
```

It is not necessary to specify both the parameter identifier namespace and the argument identifier as they are both and independently optional (see also the Grammar section):

```
CONTRACT_TEMPLATE_PARAMETER( (NumberArg) Number )   // Specify only argument identifier.
CONTRACT_PARAMETER( namespace(params) value )       // Specify only parameter identifier namespace.
```

It is recommended to never specify the argument identifiers to have the same names as the parameter identifiers in order to avoid the following usually silent bug (see also `named_param_identifiers_failure.cpp`):

```
CONTRACT_PARAMETER( (name) name )   // Both parameter and argument
CONTRACT_PARAMETER( (age) age )     // have same name, but...

CONTRACT_FUNCTION(
    void (print) ( in auto name, default("bob"), in auto age, default 42 )
        precondition( age > 0 )
) {
    if(age >= 40) throw too_old(); // Will throw because age defaulted to 42.
}

void f ( int age ) // Another parameter with the same name.
{
    print(age = 3); // So incorrect call but no compiler-error.
}
```

The call `print(age = 3);` will assign `3` to `f`'s parameter `age` instead than passing `3` as the argument of `print`'s parameter `age`. The trailing underscore _ convention and enclosing the named parameter declaration macros `CONTRACT_PARAMETER` and `CONTRACT_TEMPLATE_PARAMETER` within a namespace make this type of name clashes unlikely.

Examples

This section lists non-trivial examples programmed using this library. The listed examples are taken from the foloowing sources (which are referenced in the title of each example).

Sources	Notes
[N1962]	Examples from the proposal to add Contract Programming to C++11 submitted to the standard committee (unfortunately, this proposal was never accepted into the standard).
[N2081]	Examples from the proposal to add concepts to C++11 submitted to the standard committee (unfortunately, this proposal was never accepted into the standard). These examples have been extended to specify both concepts and contracts.
[Meyer97] [Mitchell02]	Examples using the Eiffel programming language and reprogrammed using this library for C++.
[Cline90]	Examples from a very early proposal called Annotated C++ (A++) to add Contract Programming to C++ (A++ was never implemented or proposed for addition to the standard).
[Stroustrup97]	One example that shows the importance of class invariants and how to configure this library to throw exceptions instead of terminating the program when contracts are broken.

The followings are among the most interesting examples:

Key Example	Topic
[N1962] Vector	A comparison between this library syntax and the syntax for contract and concept checking proposed by [N1962] and [N2081] respectively.
[N2081] Add	Contracts and user-defined concepts.
[Mitchell02] Counter	Subcontracting and C++11-like virtual specifiers `final`, `override`, `new`, and pure virtual.
[Meyer97] Stack4	A comparison between this library and Eiffel syntax for preconditions, postconditions, and class invariants.
[Meyer97] GCD	A comparison between this library and Eiffel syntax for loop variants and block invariants.
[Cline90] Vector	A comparison between this library and A++ syntax.

[N1962] Vector: Comparison with C++ proposed syntax

This Library (C++03)

```
// File: vector.hpp
#include <contract.hpp>
#include <boost/utility.hpp>
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/algorithm/cxx11/all_of.hpp>
#include <boost/concept_check.hpp>
#include <iterator>
#include <vector>

CONTRACT_CLASS( // Vector wrapper with contracts.
    template( class T, class Allocator, default std::allocator<T> )
    class (vector) // No base classes so no subcontracting for this example.
) {
    CONTRACT_CLASS_INVARIANT_TPL( // At very beginning of the class declaration.
        empty() == (size() == 0),
        std::distance(begin(), end()) == int(size()),
        std::distance(rbegin(), rend()) == int(size()),
        size() <= capacity(),
        capacity() <= max_size()
    ) // No static or volatile class invariants for this example.

    public: typedef typename std::vector<T, Allocator>::allocator_type
            allocator_type;
    public: typedef typename std::vector<T, Allocator>::pointer pointer;
    public: typedef typename std::vector<T, Allocator>::const_pointer
            const_pointer;
    public: typedef typename std::vector<T, Allocator>::reference reference;
    public: typedef typename std::vector<T, Allocator>::const_reference
            const_reference;
    public: typedef typename std::vector<T, Allocator>::value_type value_type;
    public: typedef typename std::vector<T, Allocator>::iterator iterator;
    public: typedef typename std::vector<T, Allocator>::const_iterator
            const_iterator;
    public: typedef typename std::vector<T, Allocator>::size_type size_type;
    public: typedef typename std::vector<T, Allocator>::difference_type
            difference_type;
    public: typedef typename std::vector<T, Allocator>::reverse_iterator
            reverse_iterator;
    public: typedef typename std::vector<T, Allocator>::const_reverse_iterator
            const_reverse_iterator;

    CONTRACT_CONSTRUCTOR_TPL(
        public (vector) ( void ) // Always specify access level `public`, etc.
            postcondition( not empty() )
            initialize( vector_() ) // Member initializers.
    ) {}

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( (const Allocator&) allocator )
            postcondition(
                empty(),
                allocator == get_allocator()
            )
            initialize( vector_(allocator) )
    ) {}

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( (size_type) count )
            postcondition(
                size() == count,
```

[N1962] and [N2081] Proposals (not part of C++)

```
// File: vector_npaper.cpp
// Extra spaces, newlines, etc used to align text with this library code.

#include <boost/utility.hpp>
#include <boost/algorithm/cxx11/all_of.hpp>
#include <concepts>
#include <iterator>
#include <vector>


template< class T, class Allocator = std::allocator<T> >
class vector
{
    invariant {
        empty() == (size() == 0);
        std::distance(begin(), end()) == int(size());
        std::distance(rbegin(), rend()) == int(size());
        size() <= capacity();
        capacity() <= max_size();
    }

    public: typedef typename std::vector<T, Allocator>::allocator_type
            allocator_type;
    public: typedef typename std::vector<T, Allocator>::pointer pointer;
    public: typedef typename std::vector<T, Allocator>::const_pointer
            const_pointer;
    public: typedef typename std::vector<T, Allocator>::reference reference;
    public: typedef typename std::vector<T, Allocator>::const_reference
            const_reference;
    public: typedef typename std::vector<T, Allocator>::value_type value_type;
    public: typedef typename std::vector<T, Allocator>::iterator iterator;
    public: typedef typename std::vector<T, Allocator>::const_iterator
            const_iterator;
    public: typedef typename std::vector<T, Allocator>::size_type size_type;
    public: typedef typename std::vector<T, Allocator>::difference_type
            difference_type;
    public: typedef typename std::vector<T, Allocator>::reverse_iterator
            reverse_iterator;
    public: typedef typename std::vector<T, Allocator>::const_reverse_iterator
            const_reverse_iterator;


    public: vector ( void )
        postcondition { not empty(); }
        : vector_()
    {}


    public: explicit vector ( const Allocator& alllocator )
        postcondition {
            empty();
            allocator == get_allocator();
        }
        : vector_(allocator)
    {}


    public: explicit vector ( size_type count )
        postcondition {
            size() == count;
```

This Library (C++03)

```
            boost::algorithm::all_of_equal(begin(), end(), T()),
                    requires boost::has_equal_to<T>::value // Requirements.
        )
        initialize( vector_(count) )
) {}

CONTRACT_CONSTRUCTOR_TPL(
    public (vector) ( (size_type) count, (const T&) value )
        postcondition(
            size() == count,
            boost::algorithm::all_of_equal(begin(), end(), value),
                    requires boost::has_equal_to<T>::value
        )
        initialize( vector_(count, value) )
) {}

CONTRACT_CONSTRUCTOR_TPL(
    public (vector) ( (size_type) count, (const T&) value,
            (const Allocator&) allocator )
        postcondition(
            size() == count,
            boost::algorithm::all_of_equal(begin(), end(), value),
                    requires boost::has_equal_to<T>::value,
            allocator == get_allocator()
        )
        initialize( vector_(count, value, allocator) )
) {}

CONTRACT_CONSTRUCTOR_TPL( // Contract with concepts.
    public template( class Iterator )
        requires( boost::InputIterator<Iterator> )
    (vector) ( (Iterator) first, (Iterator) last )
        postcondition( std::distance(first, last) == int(size()) )
        initialize( vector_(first, last) )
) {}

CONTRACT_CONSTRUCTOR_TPL(
    public template( class Iterator )
            requires( boost::InputIterator<Iterator> )
    (vector) ( (Iterator) first, (Iterator) last,
            (const Allocator&) allocator )
        postcondition(
            std::distance(first, last) == int(size()),
            allocator == get_allocator()
        )
        initialize( vector_(first, last, allocator) )
) {}

CONTRACT_CONSTRUCTOR_TPL(
    public (vector) ( (const vector&) right )
        postcondition(
            right == *this, requires boost::has_equal_to<T>::value
        )
        initialize( vector_(right.vector_) )
) {}

CONTRACT_FUNCTION_TPL( // Operator symbol and (arbitrary) name `copy`.
    public (vector&) operator(=)(copy) ( (const vector&) right )
        postcondition(
            auto result = return, // Return value.
```

[N1962] and [N2081] Proposals (not part of C++)

```
            boost::algorithm::all_of_equal(begin(), end(), T());
                    // Assertion requirements not supported by [N1962].
        }
        : vector_(count)
    {}


    public: vector ( size_type count, const T& value )
        postcondition {
            size() == count;
            boost::algorithm::all_of_equal(begin(), end(), value);

        }
        : vector_(count, value)
    {}


    public: vector ( size_type count, const T& value,
            const Allocator& allocator )
        postcondition {
            size() == count;
            boost::algorithm::all_of_equal(begin(), end(), value);

            allocator == get_allocator();
        }
        : vector_(count, value, allocator)
    {}


    public: template< class Iterator >
        requires std::InputIterator<Iterator>
    vector ( Iterator first, Iterator last )
        postcondition { std::distance(first, last) == int(size()); }
        : vector_(first, last)
    {}


    public: template< class Iterator >
        requires std::InputIterator<Iterator>
    vector ( Iterator first, Iterator last,
            const Allocator& allocator )
        postcondition {
            std::distance(first, last) == int(size());
            allocator == get_allocator();
        }
        : vector_(first, last, allocator)
    {}


    public: vector ( const vector& right )
        postcondition {
            right == *this;
        }
        : vector_(right.vector_)
    {}


    public: vector& operator= ( const vector& right )
        postcondition(result) {
```

This Library (C++03)

```
            result == *this, requires boost::has_equal_to<T>::value,
            result == right, requires boost::has_equal_to<T>::value
        )
) {
    return vector_ = right.vector_;
}

CONTRACT_DESTRUCTOR_TPL( // Destruct contract (check class invariant).
    public virtual (~vector) ( void )
) {}

CONTRACT_FUNCTION_TPL(
    // Wrapping parenthesis optional for keyword types `void`, `bool`, etc.
    public void (reserve) ( (size_type) count )
        precondition( count < max_size() )
        postcondition( capacity() >= count )
) {
    vector_.reserve(count);
}

CONTRACT_FUNCTION_TPL(
    public (size_type) (capacity) ( void ) const // Constant member.
        postcondition( auto result = return, result >= size() )
) {
    return vector_.capacity();
}

CONTRACT_FUNCTION_TPL(
    public (iterator) (begin) ( void )
        postcondition(
            auto result = return,
            if(empty()) ( // Select assertions `if(...) ( ...)`.
                result == end()
            ) // Omitted optional `else( ... )`.
        )
) {
    return vector_.begin();
}

CONTRACT_FUNCTION_TPL( // Contract overloaded function (see above).
    public (const_iterator) (begin) ( void ) const
        postcondition(
            auto result = return,
            if(empty()) ( result == end() )
        )
) {
    return vector_.begin();
}

CONTRACT_FUNCTION_TPL( // Contract with no pre/post checks class invariants.
    public (iterator) (end) ( void )
) {
    return vector_.end();
}

CONTRACT_FUNCTION_TPL(
    public (const_iterator) (end) ( void ) const
) {
    return vector_.end();
}
```

[N1962] and [N2081] Proposals (not part of C++)

```
            right == *this;
            result == *this;
        }
    {
        return vect_ = right.vect_;
    }


    public: virtual ~vector ( void )
    {}



    public: void reserve ( size_type count )
        precondition { count < max_size(); }
        postcondition { capacity() >= count; }
    {
        vector_.reserve(count);
    }


    public: size_type capacity ( void ) const
        postcondition(result) { result >= size(); }
    {
        return vector_.capacity();
    }


    public: iterator begin ( void )
        postcondition {

            if(empty()) {
                result == end();
            }
        }
    {
        return vector_.begin();
    }


    public: const_iterator begin ( void ) const
        postcondition(result) {

            if(empty()) { result == end(); }
        }
    {
        return vector_.begin();
    }


    public: iterator end ( void )
    {
        return vector_.end();
    }


    public: const_iterator end ( void ) const
    {
        return vector_.end();
    }
```

This Library (C++03)

```
CONTRACT_FUNCTION_TPL(
    public (reverse_iterator) (rbegin) ( void )
        postcondition(
            auto result = return,
            if(empty()) ( result == rend() )
        )
) {
    return vector_.rbegin();
}

CONTRACT_FUNCTION_TPL(
    public (const_reverse_iterator) (rbegin) ( void ) const
        postcondition(
            auto result = return,
            if(empty()) ( result == rend() )
        )
) {
    return vector_.rbegin();
}

CONTRACT_FUNCTION_TPL(
    public (reverse_iterator) (rend) ( void )
) {
    return vector_.rend();
}

CONTRACT_FUNCTION_TPL(
    public (const_reverse_iterator) (rend) ( void ) const
) {
    return vector_.rend();
}

CONTRACT_FUNCTION_TPL(
    public void (resize) ( (size_type) count )
        postcondition(
            auto old_size = CONTRACT_OLDOF size(), // Old value (at entry).
            size() == count,
            if(count > old_size) (
                boost::algorithm::all_of_equal(begin() + old_size, end(),
                        T()), requires boost::has_equal_to<T>::value
            )
        )
) {
    vector_.resize(count);
}

CONTRACT_FUNCTION_TPL(
    public void (resize) ( (size_type) count, (T) value )
        postcondition(
            (size_type) old_size = CONTRACT_OLDOF size(), // With old type.
            size() == count,
            count > old_size ? // Ternary operator.
                boost::algorithm::all_of_equal(begin() + old_size, end(),
                        value)
            :
                true
            , requires boost::has_equal_to<T>::value
        )
) {
```

[N1962] and [N2081] Proposals (not part of C++)

```
public: reverse_iterator rbegin ( void )
    postcondition(result) {

        if(empty()) { result == rend(); }
    }
{
    return vector_.rbegin();
}


public: const_reverse_iterator rbegin ( void ) const
    postcondition(result) {

        if(empty()) { result == rend(); }
    }
{
    return vector_.rbegin();
}


public: reverse_iterator rend ( void )
{
    return vector_.rend();
}


public: const_reverse_iterator rend ( void ) const
{
    return vector_.rend();
}


public: void resize ( size_type count )
    postcondition {

        size() == count;
        if(count > oldof size()) {
            boost::algorithm::all_of_equal(begin() + oldof size(), end(),
                    T());
        }
    }
{
    vectcor_.resize(count);
}


public: void resize ( size_type count, T value )
    postcondition {

        size() == count;
        count > oldof size() ?
            boost::algorithm::all_of_equal(begin() + oldof size(), end(),
                    value)
            :
                true
            ;
    }
{
```

This Library (C++03)

```
        vector_.resize(count, value);
    }

    CONTRACT_FUNCTION_TPL(
        public (size_type) (size) ( void ) const
            postcondition( auto result = return, result <= capacity() )
    ) {
        return vector_.size();
    }

    CONTRACT_FUNCTION_TPL(
        public (size_type) (max_size) ( void ) const
            postcondition( auto result = return, result >= capacity() )
    ) {
        return vector_.max_size();
    }

    CONTRACT_FUNCTION_TPL(
        public bool (empty) ( void ) const
            postcondition(
                auto result = return,
                result == (size() == 0)
            )
    ) {
        return vector_.empty();
    }

    CONTRACT_FUNCTION_TPL(
        public (Allocator) (get_allocator) ( void ) const
    ) {
        return vector_.get_allocator();
    }

    CONTRACT_FUNCTION_TPL(
        public (reference) (at) ( (size_type) index )
            precondition( index < size() )
    ) {
        return vector_.at(index);
    }

    CONTRACT_FUNCTION_TPL(
        public (const_reference) (at) ( (size_type) index ) const
            precondition( index < size() )
    ) {
        return vector_.at(index);
    }

    CONTRACT_FUNCTION_TPL(
        public (reference) operator([])(at) ( (size_type) index )
            precondition( index < size() )
    ) {
        return vector_[index];
    }

    CONTRACT_FUNCTION_TPL(
        public (const_reference) operator([])(at) ( (size_type) index ) const
            precondition( index < size() )
    ) {
        return vector_[index];
    }
```

[N1962] and [N2081] Proposals (not part of C++)

```
        vector_.resize(count, value);
    }


    public: size_type size ( void ) const
        postcondition(result) { result <= capacity(); }
    {
        return vector_.size();
    }


    public: size_type max_size ( void ) const
        postcondition(result) { result >= capacity(); }
    {
        return vector_.max_size();
    }


    public: bool empty ( void ) const
        postcondition(result) {

            result == (size() == 0);
        }
    {
        return vector_.empty();
    }


    public: Alloctor get_allocator ( void ) const
    {
        return vector_.get_allocator();
    }


    public: reference at ( size_type index )
        precondition { index < size(); }
    {
        return vectcor_.at(index);
    }


    public: const_reference at ( size_type index ) const
        precondition { index < size(); }
    {
        return vector_.at(index);
    }


    public: reference operator[] ( size_type index )
        precondition { index < size(); }
    {
        return vector_[index];
    }


    public: const_reference operator[] ( size_type index ) const
        precondition { index < size(); }
    {
        return vectcor_[index];
    }
```

This Library (C++03)

```
CONTRACT_FUNCTION_TPL(
    public (reference) (front) ( void )
        precondition( not empty() ) // `not` instead of symbol `!`.
) {
    return vector_.front();
}

CONTRACT_FUNCTION_TPL(
    public (const_reference) (front) ( void ) const
        precondition( bool(!empty()) ) // Or, `bool` to wrap symbol `!`.
) {
    return vector_.front();
}

CONTRACT_FUNCTION_TPL(
    public (reference) (back) ( void )
        precondition( not empty() )
) {
    return vector_.back();
}

CONTRACT_FUNCTION_TPL(
    public (const_reference) (back) ( void ) const
        precondition( not empty() )
) {
    return vector_.back();
}

CONTRACT_FUNCTION_TPL(
    public void (push_back) ( (const T&) value )
        precondition( size() < max_size() )
        postcondition(
            auto old_size = CONTRACT_OLDOF size(),
            auto old_capacity = CONTRACT_OLDOF capacity(),
            back() == value, requires boost::has_equal_to<T>::value,
            size() == old_size + 1,
            capacity() >= old_capacity
        )
) {
    vector_.push_back(value);
}

CONTRACT_FUNCTION_TPL(
    public void (pop_back) ( void )
        precondition( not empty() )
        postcondition(
            auto old_size = CONTRACT_OLDOF size(),
            size() == old_size - 1
        )
) {
    vector_.pop_back();
}

CONTRACT_FUNCTION_TPL(
    public template( class Iterator )
        requires( boost::InputIterator<Iterator> )
    void (assign) ( (Iterator) first, (Iterator) last )
        // precondition: [begin(), end()) does not contain [first, last)
        postcondition( std::distance(first, last) == int(size()) )
```

[N1962] and [N2081] Proposals (not part of C++)

```
public: reference front ( void )
    precondition { not empty(); }
{
    return vectcor_.front();
}


public: const_reference front ( void ) const
    precondition { bool(!empty()); }
{
    return vector_.front();
}


public: reference back ( void )
    precondition { not empty(); }
{
    return vector_.back();
}


public: const_reference back ( void ) const
    precondition { not empty(); }
{
    return vector_.back();
}


public void push_back ( const T& value )
    precondition { size() < max_size() }
    postcondition {


        back() == vallue;
        size() == oldof size() + 1;
        capacity() >= oldof capacity()
    }
{
    vector_.push_back(value);
}


public: void pop_back ( void )
    precondition { not empty(); }
    postcondition {

        size() == oldof size() - 1;
    }
{
    vector_.pop_back();
}


public: template< class Iterator >
    requires std::InputIterator<Iterator>
void assign ( Iterator first, Iterator last )
    // precondition: [begin(), end()) does not contain [first, last)
    postcondition { std::distance(first, last) == int(size()); }
```

This Library (C++03)

```
) {
    vector_.assign(first, last);
}

CONTRACT_FUNCTION_TPL(
    public void (assign) ( (size_type) count, (const T&) value )
        precondition( count <= max_size() )
        postcondition(
            boost::algorithm::all_of_equal(begin(), end(), value),
                    requires boost::has_equal_to<T>::value
        )
) {
    vector_.assign(count, value);
}

CONTRACT_FUNCTION_TPL(
    public (iterator) (insert) ( (iterator) where, (const T&) value )
        precondition( size() < max_size() )
        postcondition(
            auto result = return,
            auto old_size = CONTRACT_OLDOF size(),
            value == *result, requires boost::has_equal_to<T>::value,
            size() == old_size + 1
            //  if(capacity() > oldof capacity())
            //     [begin(), end()) is invalid
            //  else
            //     [where, end()) is invalid
        )
) {
    return vector_.insert(where, value);
}

CONTRACT_FUNCTION_TPL(
    public void (insert) ( (iterator) where, (size_type) count,
            (const T&) value )
        precondition( size() + count < max_size() )
        postcondition(
            using boost::prior,
            auto old_size = CONTRACT_OLDOF size(),
            auto old_capacity = CONTRACT_OLDOF capacity(),
            auto old_where = CONTRACT_OLDOF where,
            size() == old_size + count,
            capacity() >= old_capacity,
            if(capacity() == old_capacity) (
                boost::algorithm::all_of_equal(prior(old_where),
                        prior(old_where) + count, value),
                        requires boost::has_equal_to<T>::value
                // [where, end()) is invalid
            ) // else [begin(), end()) is invalid
        )
) {
    vector_.insert(where, count, value);
}

CONTRACT_FUNCTION_TPL(
    public template( class Iterator )
        requires( boost::InputIterator<Iterator> )
    void (insert) ( (iterator) where, (Iterator) first, (Iterator) last )
        precondition(
            // [first, last) is not contained in [begin(), end())
```

[N1962] and [N2081] Proposals (not part of C++)

```
{
    vector_.assign(first, last);
}


public: void assign ( size_type count, const T& vallue )
    precondition { count <= max_size(); }
    postcondition {
        boost::algorithm::all_of_equal(begin(), end(), value);

    }
{
    vector_.assign(count, value);
}


public: iterator insert ( iterator where, const T& value )
    precondition { size() < max_size(); }
    postcondition(result) {


        value == *result;
        size() == oldof size() + 1;
        //  if(capacity() > oldof capacity())
        //     [begin(), end()) is invalid
        //  else
        //     [where, end()) is invalid
    }
{
    return vector_.insert(where, value);
}


public: void insert ( iterator where, size_type count,
        const T& value )
    precondition { size() + count < max_size(); }
    postcondition {




        size() == oldof size() + count;
        capacity() >= oldof capacity();
        if(capacity() == oldof capacity()) {
            boost::algorithm::all_of_equal(boost::prior(oldof where),
                    boost::prior(oldof where) + count, value);

            // [where, end()) is invalid
        } // else [begin(), end()) is invalid
    }
{
    vector_.insert(where, count, value);
}


public: template< class Iterator >
    requires std::InputIterator<Iterator>
void insert ( iterator where, Iterator first, Iterator last )
    precondition {
        // [first, last) is not contained in [begin(), end())
```

This Library (C++03)

```
            size() + std::distance(first, last) < max_size()
        )
        postcondition(
            auto old_size = CONTRACT_OLDOF size(),
            auto old_capacity = CONTRACT_OLDOF capacity(),
            size() == old_size + std::distance(first, last),
            capacity() >= old_capacity
        )
) {
    vector_.insert(where, first, last);
}

CONTRACT_FUNCTION_TPL(
    public (iterator) (erase) ( (iterator) where )
        precondition(
            not empty(),
            where != end()
        )
        postcondition(
            auto result = return,
            auto old_size = CONTRACT_OLDOF size(),
            size() == old_size - 1,
            if(empty()) ( result == end() )
            // [where, end()) is invalid
        )
) {
    return vector_.erase(where);
}

CONTRACT_FUNCTION_TPL(
    public (iterator) (erase) ( (iterator) first, (iterator) last )
        precondition( size() >= std::distance(first, last) )
        postcondition(
            auto result = return,
            auto old_size = CONTRACT_OLDOF size(),
            size() == old_size - std::distance(first, last),
            if(empty()) ( result == end() )
            // [first, last) is invalid
        )
) {
    return vector_.erase(first, last);
}

CONTRACT_FUNCTION_TPL(
    public void (clear) ( void )
        postcondition( empty() )
) {
    vector_.clear();
}

CONTRACT_FUNCTION_TPL(
    public void (swap) ( (vector&) right )
        postcondition(
            auto old_self = CONTRACT_OLDOF *this,
            auto old_right = CONTRACT_OLDOF right,
            right == old_self, requires boost::has_equal_to<T>::value,
            old_right == *this, requires boost::has_equal_to<T>::value
        )
) {
    vector_.swap(right);
```

[N1962] and [N2081] Proposals (not part of C++)

```
            size() + std::distance(first, last) < max_size();
        }
        postcondition {


            size() == oldof size() + std::distance(first, last);
            capacity() >= oldof capacity();
        }
    {
        vector_.insert(where, first, last);
    }


    public: iterator erase ( iterator where )
        precondition {
            not empty();
            where != end();
        }
        postcondition(result) {


            size() == oldod size() - 1;
            if(empty()) { result == end(); }
            // [where, end()) is invalid
        }
    {
        return vector_.erase(where);
    }


    public: iterator erase ( iterator first, iterator last )
        precondition { size() >= std::distance(first, lasst); }
        postcondition(result) {


            size() == oldof size() - std::distance(first, last);
            if(empty()) { result == end(); }
            // [first, last) is invalid
        }
    {
        return vector_.erase(first, last);
    }


    public: void clear ( void )
        postcondition( empty() )
    {
        vector_.clear();
    }


    public: void swap ( vector& right )
        postcondition {


            right == oldof *this;
            oldof right == *this;
        }
    {
        vector_.swap(right);
```

This Library (C++03)

```
    }

    friend bool operator== ( vector const& left, vector const& right )
    {
        return left.vector_ == right.vector_;
    }

    private: std::vector<T, Allocator> vector_;
};
```

[N1962] and [N2081] Proposals (not part of C++)

```
    }

    friend bool operator== ( vector const& left, vector const& right )
    {
        return left.vector_ == right.vector_;
    }

    private: std::vector<T, Allocator> vector_;
};
```

```
// File: vector.cpp
#include "vector.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    // Run a few of the operations (could test more of them...).
    vector<char> v(3);
    BOOST_TEST(v.size() == 3);
    BOOST_TEST(boost::algorithm::all_of_equal(v, '\0'));
    const vector<char>& cv = v; // A reference so no copy.

    vector<char> w(v);
    BOOST_TEST(w == v);

    vector<char>::iterator b = v.begin();
    BOOST_TEST(*b == '\0');

    vector<char>::const_iterator cb = cv.begin();
    BOOST_TEST(*cb == '\0');

    v.insert(b, 2, 'a');
    BOOST_TEST(v[0] == 'a');
    BOOST_TEST(v[1] == 'a');

    v.push_back('b');
    BOOST_TEST(v.back() == 'b');
    return boost::report_errors();
}
```

[N1962] Circle: Subcontracting

```
// File: circle.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>

CONTRACT_CLASS(
    class (shape)
) {
    CONTRACT_CLASS_INVARIANT( void ) // Must always specify class invariants.

    CONTRACT_DESTRUCTOR(
        public virtual (~shape) ( void )
    ) {}

    CONTRACT_FUNCTION(
        public virtual int (compute_area) ( void ) const
            // No base preconditions so all derived preconditions true.
            postcondition( auto result = return, result > 0 )
    ) = 0; // Contract for pure virtual function.
};

CONTRACT_CLASS(
    class (circle) extends( public shape ) // Subcontract from base `shape`.
) {
    CONTRACT_CLASS_INVARIANT( void ) // In AND with base class invariants.

    public: int radius() const { return 2; }

    CONTRACT_FUNCTION( // Also using `override` specifier (optional).
        public virtual int (compute_area) ( void ) const override
            // Eventual preconditions in OR with base function preconditions.
            postcondition( // In AND with base function postconditions.
                auto result = return,
                result == pi * radius() * radius()
            )
    ) {
        return pi * radius() * radius();
    }

    private: static const int pi = 3; // Truncated from 3.14...
};

int main ( void )
{
    circle c;
    BOOST_TEST(c.compute_area() == 12);
    return boost::report_errors();
}
```

[N1962] Factorial: Recursion and assertion computational complexity

```
// File: factorial.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>

// Assertion requirements used to model assertion computational complexity.
#define O_1      0 // O(1) constant (default).
#define O_N      1 // O(n) linear.
#define O_NN     2 // O(n * n) quadratic.
#define O_NX     3 // O(n^x) polynomial.
#define O_FACTN  4 // O(n!) factorial.
#define O_EXPN   5 // O(e^n) exponential.
#define O_ALL   10
#define COMPLEXITY_MAX O_ALL

CONTRACT_FUNCTION(
    int (factorial) ( int n )
        precondition(
            n >= 0, // Non-negative natural number.
            n <= 12 // Max function input.
        )
        postcondition(
            auto result = return,
            result >= 1,
            if(n < 2) ( // Select assertion.
                result == 1
            ) else (
                // Assertions automatically disabled within other assertions.
                // Therefore, this postcondition can recursively call the
                // function without causing infinite recursion.
                result == n * factorial(n - 1),
                        // This assertion introduces significant run-time
                        // overhead (same as the function body) so assertion
                        // requirements are used to selectively disable it.
                        requires O_FACTN <= COMPLEXITY_MAX
            )
        )
) {
    if(n < 2) return 1;
    else return n * factorial(n - 1);
}

int main ( void )
{
    BOOST_TEST(factorial(4) == 24);
    return boost::report_errors();
}
```

[N1962] Equal: Operators

```
// File: equal.hpp
#ifndef EQUAL_HPP_
#define EQUAL_HPP_

#include "equal_not.hpp"
#include <contract.hpp>

CONTRACT_FUNCTION_TPL(
    template( typename T )
    bool operator(==)(equal) ( (T const&) left, (T const&) right )
        postcondition(
            auto result = return,
            result == !(left != right)
        )
) {
    return left.value == right.value;
}

#endif // #include guard
```

```
// File: equal_not.hpp
#ifndef EQUAL_NOT_HPP_
#define EQUAL_NOT_HPP_

#include "equal.hpp"
#include <contract.hpp>

CONTRACT_FUNCTION_TPL(
    template( typename T )
    bool operator(!=)(not_equal) ( (T const&) left, (T const&) right )
        postcondition(
            auto result = return,
            result == !(left == right)
        )
) {
    return left.value != right.value;
}

#endif // #include guard
```

```
// File: equal_main.cpp
#include "equal.hpp"
#include "equal_not.hpp"
#include <boost/detail/lightweight_test.hpp>

struct number { int value; };

int main ( void )
{
    number n;
    n.value = 123;

    BOOST_TEST((n == n) == true);
    BOOST_TEST((n != n) == false);
    return boost::report_errors();
}
```

[N1962] Sum: Separated body definitions

```
// File: sum.hpp
#ifndef SUM_HPP_
#define SUM_HPP_

#include <contract.hpp>

CONTRACT_FUNCTION(
    int (sum)( int count, (int*) array )
        precondition( count % 4 == 0 )
) ; // Separate body definition from function declaration (see ".cpp" file).

#endif // #include guard
```

```
// File: sum.cpp
#include "sum.hpp"

int CONTRACT_FREE_BODY(sum) ( int count, int* array ) {
    int total = 0;
    for(int i = 0; i < count; ++i) total += array[i];
    return total;
}
```

```
// File: sum_main.cpp
#include "sum.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    int a[8] = {1, 1, 1, 1, 1, 1, 1, 1};
    BOOST_TEST(sum(8, a) == 8);
    return boost::report_errors();
}
```

[N1962] Square Root: Default parameters and comparison with D syntax

This Library (C++03)

```
// File: sqrt.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <cmath>

CONTRACT_FUNCTION( // Default value for `precision` parameter.
    double (mysqrt) ( double x, double precision, default(1e-6) )
        precondition( x >= 0.0 )
        postcondition(
            auto root = return, // Result value named `root`.
            fabs(root * root - x) <= precision // Equal within precision.
        )
) {
    return sqrt(x);
}

int main ( void )
{
    BOOST_TEST(fabs(mysqrt(4.0) - 2.0) <= 1e-6);
    return boost::report_errors();
}
```

The D Programming Language

```
// File: sqrt.d
// Extra spaces, newlines, etc used to align text with this library code.



real mysqrt ( real x ) // No default parameters in D.
    in { assert(x >= 0); }
    out(root) {

        assert(std.math.fabs(root * root - x) <= 1e6);
    }
body {
    return std.math.sqrt(x);
}
```

[N1962] Block: Block invariants

```
// File: block_invariant.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <iostream>

int main ( void )
{
    int i = 0;
    for(i = 0; i < 100; ++i)
    {
        CONTRACT_BLOCK_INVARIANT( const( i ) i >= 0 )
    }
    BOOST_TEST(i == 100);
    return boost::report_errors();
}
```

[N2081] Add: Generic addition algorithm

```
// File: add.hpp
#ifndef ADD_HPP
#define ADD_HPP

#include <contract.hpp>
#include <boost/concept_check.hpp>
#include <boost/type_traits/has_equal_to.hpp>

template< typename T >
struct Addable // User-defined concept.
{
    BOOST_CONCEPT_USAGE(Addable)
    {
        return_type(x + y); // Check addition `T operator+(T x, T y)`.
    }

private:
    void return_type ( T const& ) {} // Used to check addition returns type `T`.

    T const& x;
    T const& y;
};

CONTRACT_FUNCTION(
    template( typename T ) requires( Addable<T> )
    (T) (add) ( (T const&) x, (T const&) y )
        postcondition(
            auto result = return,
            result == x + y, requires boost::has_equal_to<T>::value
        )
) {
    return x + y;
}

#endif // #include guard
```

```
// File: add.cpp
#include "add.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    int x = 10, y = 20;
    BOOST_TEST(add(x, y) == 30);
    return boost::report_errors();
}
```

```
// File: add_error.cpp
#include "add.hpp"

struct num
{
    int value;
    num ( int v = 0 ) : value(v) {};
};

int main ( void )
{
    num n(10), m(20);
    add(n, m); // Error: Correctly fails `add` concept requirements.
    return 0;
}
```

[N2081] Advance: Concept-based iterator overloading (emulated using tags)

```
// File: advance.cpp
#include <contract.hpp>
#include <boost/mpl/bool.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>
#include <list>
#include <iterator>

namespace aux { // Tag dispatch helpers (in a private namespace).

template< typename InputIterator, typename Distance >
void myadvance_dispatch ( InputIterator& i, Distance n,
        std::input_iterator_tag )
{
    CONTRACT_LOOP( while(n--) ) {
        CONTRACT_LOOP_VARIANT_TPL( const( n ) n )
        ++i;
    }
}

template< typename BidirectionalIterator, typename Distance >
void myadvance_dispatch ( BidirectionalIterator& i, Distance n,
        std::bidirectional_iterator_tag )
{
    if(n >= 0) {
        CONTRACT_LOOP( while(n--) ) {
            CONTRACT_LOOP_VARIANT_TPL( const( n ) n )
            ++i;
        }
    } else {
#ifndef CONTRACT_CONFIG_NO_LOOP_VARIANTS
        // Avoid unused variable warning when loop invariants are disabled.
        Distance n_max = n;
#endif
        CONTRACT_LOOP( while(n++) ) {
            CONTRACT_LOOP_VARIANT_TPL( const( n_max, n ) n_max - n )
            --i;
        }
    }
}

template< typename RandomAccessIterator, typename Distance >
void myadvance_dispatch ( RandomAccessIterator& i, Distance n,
```

```
        std::random_access_iterator_tag )
{
    i += n;
}

} // namespace aux

// Contract helper meta-function.
template< typename T > struct is_input_iterator : boost::mpl::false_ {};
template< > struct is_input_iterator<std::input_iterator_tag> :
        boost::mpl::true_ {};

CONTRACT_FUNCTION(
    template( typename InputIterator, typename Distance )
    void (myadvance) ( (InputIterator&) i, (Distance) n )
        precondition(
            // range [i, i + n] is non-singular.
            if(is_input_iterator<typename std::iterator_traits<InputIterator>::
                    iterator_category>::value) (
                n > 0 // if input iterator, n positive
            )
        )
        postcondition(
            auto old_i = CONTRACT_OLDOF i,
            std::distance(old_i, i) == n // iterator advanced of n
        )
) {
    aux::myadvance_dispatch(i, n, typename
            std::iterator_traits<InputIterator>:: iterator_category());
}

int main ( void )
{
    std::vector<int> v(10);
    v[2] = 123; v[3] = -123;
    std::vector<int>::iterator r = v.begin(); // Random iterator.
    myadvance(r, 3);
    BOOST_TEST(*r == -123);

    std::list<int> l(v.begin(), v.end());
    std::list<int>::iterator b = l.begin(); // Bidirectional and not random.
    myadvance(b, 2);
    BOOST_TEST(*b == 123);
    return boost::report_errors();
}
```

[N2081] Find: Generic find algorithm

```
// File: find.hpp
#ifndef FIND_HPP_
#define FIND_HPP_

#include <contract.hpp>
#include <boost/concept_check.hpp>

CONTRACT_FUNCTION(
    template( typename Iterator )
        requires(
            boost::InputIterator<Iterator>,
            boost::EqualityComparable< // Equality needed to find value.
                    typename boost::InputIterator<Iterator>::value_type>
        )
    (Iterator) (myfind) ( (Iterator) first, (Iterator) last,
            (typename boost::InputIterator<Iterator>::value_type const&) value )
        // precondition: range [first, last) is valid
        postcondition(
            auto result = return,
            if(result != last) (
                value == *result // if result not last, value found
            )
        )
) {
    CONTRACT_LOOP( while(first != last && *first != value) ) {
        CONTRACT_LOOP_VARIANT_TPL(
                const( first, last ) std::distance(first, last) )
        ++first;
    }
    return first;
}

#endif // #include guard
```

```
// File: find.cpp
#include "find.hpp"
#include <boost/detail/lightweight_test.hpp>
#include <vector>

int main ( void )
{
    std::vector<int> ints(3);
    ints[0] = 1; ints[1] = 2; ints[2] = 3;
    BOOST_TEST(*myfind(ints.begin(), ints.end(), 2) == 2);
    return boost::report_errors();
}
```

```
// File: find_error.cpp
#include "find.hpp"
#include <vector>

struct num
{
    int value;
    explicit num ( int v = 0 ) : value(v) {}
};

int main ( void )
{
    std::vector<num> nums(3);
    nums[0] = num(1); nums[1] = num(2); nums[2] = num(3);
    // Error: Correctly, num does not meet concept requirements.
    myfind(nums.begin(), nums.end(), num(2));
    return 0;
}
```

[N2081] Apply: Overloaded invocation of functors

```
// File: apply.cpp
#include <contract.hpp>
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>

// Use assertion requirements to model assertion computational complexity.
#define O_1     0   // constant computational complexity O(1)
#define O_BODY  10  // same computation complexity of the body
#define COMPLEXITY_MAX O_1 // only check assertions with complexity within O(1)

CONTRACT_FUNCTION( // Invoke unary functors.
    template( typename Result, typename Argument0, typename Function )
        requires( (boost::UnaryFunction<Function, Result, Argument0>) )
    (Result) (apply) ( (Function) function, (Argument0) argument0,
            (Result&) result )
        postcondition(
            auto return_value = return,
            result == function(argument0),
                    requires O_BODY <= COMPLEXITY_MAX &&
                            boost::has_equal_to<Result>::value,
            return_value == result,
                    requires boost::has_equal_to<Result>::value
        )
) {
    return result = function(argument0);
}

CONTRACT_FUNCTION( // Overload to invoke binary functors.
    template( typename Result, typename Argument0, typename Argument1,
            typename Function )
        requires( (boost::BinaryFunction<Function, Result, Argument0,
                Argument1>) )
    (Result) (apply) ( (Function) function, (Argument0) argument0,
            (Argument1) argument1, (Result&) result )
        postcondition(
            auto return_value = return,
            result == function(argument0, argument1),
                    requires O_BODY <= COMPLEXITY_MAX &&
                            boost::has_equal_to<Result>::value,
```

```
            return_value == result,
                    requires boost::has_equal_to<Result>::value
        )
) {
    return result = function(argument0, argument1);
}

int complement ( int x ) { return -x; }
int add ( int x, int y ) { return x + y; }

int main ( void )
{
    int r = 0;
    apply(complement, 1, r);
    BOOST_TEST(r == -1);

    apply(add, 1, 2, r);
    BOOST_TEST(r == 3);
    return boost::report_errors();
}
```

[N2081] For Each: Generic for-each algorithm

```
// File: for_each.cpp
#include <contract.hpp>
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>

CONTRACT_FUNCTION(
    template( typename Iterator, typename Function )
        requires(
            boost::InputIterator<Iterator>,
            (boost::UnaryFunction<Function, void, typename
                    boost::InputIterator<Iterator>::value_type>)
        )
    (Function) (my_for_each) ( (Iterator) first, (Iterator) last,
            (Function) function )
        // precondition: range [first, last) is valid
        postcondition(
            auto result = return,
            result == function, requires boost::has_equal_to<Function>::value
        )
) {
    CONTRACT_LOOP( while(first < last) ) { // OK: Iterator is InputIterator.
        CONTRACT_LOOP_VARIANT_TPL(
                const( first, last ) std::distance(first, last) )
        function(*first); // OK: Function is UnaryFunction.
        ++first; // OK: Iterator is InputIterator.
    }
    return function;
}

int total = 0;
void add ( int i ) { total += i; }

int main ( void )
{
    std::vector<int> v(3);
    v[0] = 1; v[1] = 2; v[2] = 3;
    my_for_each(v.begin(), v.end(), add);
    BOOST_TEST(total == 6);
    return boost::report_errors();
}
```

[N2081] Transform: Generic binary transformation algorithm

```
// File: transform.cpp

#define CONTRACT_CONFIG_FUNCTION_ARITY_MAX 5 // Support for 5 parameters.
#include <contract.hpp>

#include <boost/type_traits/has_equal_to.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>
#include <algorithm>

CONTRACT_FUNCTION(
    template(
        typename Iterator1,
        typename Iterator2,
        typename ResultIterator,
        typename Function
    )
        requires(
            boost::InputIterator<Iterator1>,
            boost::InputIterator<Iterator2>,
            (boost::OutputIterator<ResultIterator, typename
                    boost::InputIterator<Iterator1>::value_type>),
            (boost::BinaryFunction<Function,
                    typename boost::InputIterator<Iterator1>::value_type,
                    typename boost::InputIterator<Iterator1>::value_type,
                    typename boost::InputIterator<Iterator2>::value_type>)
        )
    (ResultIterator) (mytransform) (
            (Iterator1) first1,
            (Iterator1) last1,
            (Iterator2) first2,
            (ResultIterator) result,
            (Function) function
        )
        // precondition: [first1, last1) is valid
        // precondition: [first2, first2 + (last1 - first1)) is valid
        // precondition: result is not an iterator within [first1 + 1, last1)
        //               or [first2 + 1, first2 + (last1 - first1))
        // precondition: [result, result + (last1 - first1)) is valid
) {
    return std::transform(first1, last1, first2, result, function);
}

int main ( void )
{
    std::vector<int> v(3);
    v[0] = 1; v[1] = 2; v[2] = 3;
    std::vector<int> w(3);
    w[0] = 10; w[1] = 20; w[2] = 30;

    mytransform(v.begin(), v.end(), w.begin(), v.begin(), std::plus<int>());

    BOOST_TEST(v[0] == 11);
    BOOST_TEST(v[1] == 22);
    BOOST_TEST(v[2] == 33);
    return boost::report_errors();
}
```

[N2081] Count: Generic counting algorithm

```
// File: count.cpp
#include <contract.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>

CONTRACT_FUNCTION(
    template( typename Iterator )
        requires(
            boost::InputIterator<Iterator>,
            boost::EqualityComparable<Iterator>,
            boost::EqualityComparable<
                    typename boost::InputIterator<Iterator>::value_type>
        )
    (typename boost::InputIterator<Iterator>::difference_type) (mycount) (
            (Iterator) first, (Iterator) last,
            (typename boost::InputIterator<Iterator>::value_type const&) value )
        // precondition: range [first, last) is valid
        postcondition( auto result = return, result >= 0 )
) {
    typename boost::InputIterator<Iterator>::difference_type result = 0;
    CONTRACT_LOOP( while(first != last) ) { // OK: Iterator has `==`.
        CONTRACT_LOOP_VARIANT_TPL(
                const( first, last ) std::distance(first, last) )
        if(*first == value) // OK: Value is equality comparable.
            ++result;
        ++first; // OK: Iterator is input iterator.
    }
    return result; // OK: Input iterator difference is copy constructible.
}

int main ( void )
{
    std::vector<int> v(3);
    v[0] = -1; v[1] = 0; v[2] = -1;
    BOOST_TEST(mycount(v.begin(), v.end(), -1) == 2);
    return boost::report_errors();
}
```

[N2081] Convert: Conversion between two types

```
// File: convert.hpp
#ifndef CONVERT_HPP_
#define CONVERT_HPP_

#include <contract.hpp>
#include <boost/concept_check.hpp>

CONTRACT_FUNCTION(
    template( typename To, typename From )
        requires( (boost::Convertible<From, To>) )
    (To) (convert) ( (From const&) from )
) {
    return from;
}

#endif // #include guard
```

```cpp
// File: convert.cpp
#include "convert.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    BOOST_TEST(convert<int>('\0') == 0);
    return boost::report_errors();
}
```

```cpp
// File: convert_error.cpp
#include "convert.hpp"

int main ( void )
{
    convert<int*>(10); // Error: Correctly cannot convert type.
    return 0;
}
```

[N2081] Equal: Generic equality comparison

```cpp
// File: equal.hpp
#ifndef EQUAL_HPP_
#define EQUAL_HPP_

#include <contract.hpp>
#include <boost/concept_check.hpp>

CONTRACT_FUNCTION(
    template( typename T )
        requires( boost::EqualityComparable<T> )
    bool (equal) ( (T const&) left, (T const&) right )
        postcondition( auto result = return, result == (left == right) )
) {
    return left == right;
}

#endif // #include guard
```

```cpp
// File: equal.cpp
#include "equal.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void)
{
    BOOST_TEST(equal(1, 1) == true);
    BOOST_TEST(equal(1, 2) == false);
    return boost::report_errors();
}
```

```
// File: equal_error.cpp
#include "equal.hpp"

struct num
{
    int value;
    explicit num ( int v = 0 ) : value(v) {}
};

int main ( void )
{
    equal(num(1), num(1)); // Compiler error.
    return 0;
}
```

[N2081] Less Equal: Generic less-than or equal-to comparison

```
// File: less_eq.cpp
#include <contract.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>

CONTRACT_FUNCTION(
    template( typename T )
        requires(
            boost::EqualityComparable<T>,
            boost::LessThanComparable<T>
        )
    bool (less_eq) ( (T) left, (T) right )
        postcondition(
            auto result = return,
            result == (left < right || left == right)
        )
) {
    return left < right || left == right;
}

int main ( void )
{
    BOOST_TEST(less_eq(1, 2) == true);
    BOOST_TEST(less_eq(2, 1) == false);
    return boost::report_errors();
}
```

[N2081] De-Ref: Generic iterator dereferencing

```
// File: deref.cpp
#include <contract.hpp>
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/concept_check.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>

CONTRACT_FUNCTION(
    template( typename Iterator )
        requires(
            boost::InputIterator<Iterator>,
            boost::CopyConstructible<
                    typename boost::InputIterator<Iterator>::value_type>
        )
    (typename boost::InputIterator<Iterator>::value_type) (deref) (
            (Iterator) iterator )
        // precondition: iterator is non-singular
        postcondition(
            auto result = return,
            result == *iterator, requires boost::has_equal_to<
                    typename boost::InputIterator<Iterator>::value_type>::value
        )
) {
    return *iterator;
}

int main ( void )
{
    std::vector<int> v(1);
    v[0] = 123;
    BOOST_TEST(deref(v.begin()) == 123);
    return boost::report_errors();
}
```

[N2081] Min: Generic minimum algorithm

```
// File: min.hpp
#ifndef MIN_HPP_
#define MIN_HPP_

#include <contract.hpp>
#include <boost/type_traits/has_equal_to.hpp>
#include <boost/concept_check.hpp>

CONTRACT_FUNCTION(
    template( typename T ) requires( boost::LessThanComparable<T> )
    (T const&) (min) ( (T const&) x, (T const&) y )
        postcondition(
            auto result = return,
            x < y ? result == x : result == y,
                    requires boost::has_equal_to<T>::value
        )
) {
    return x < y ? x : y; // OK: T is less than comparable `<`.
}

#endif // #include guard
```

```cpp
// File: min.cpp
#include "min.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    BOOST_TEST(min(1, 2) == 1);
    BOOST_TEST(min(-1, -2) == -2);
    return boost::report_errors();
}
```

```cpp
// File: min_error.cpp
#include "min.hpp"

struct num
{
    int value;
    explicit num ( int v = 0 ) : value(v) {}
};

int main ( void )
{
    min(num(1), num(2)); // Compiler error: Fail concept requirements.
    return 0;
}
```

[Meyer97] Stack4: Comparison with Eiffel syntax

This Library (C++03)

```
// File: stack4.hpp
#ifndef STACK4_HPP_
#define STACK4_HPP_

#include <contract.hpp>

// Specification.

CONTRACT_CLASS( // Dispenser with LIFO access policy and a fixed max capacity.
    template( typename T )
    class (stack4)
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        count() >= 0, // count non negative
        count() <= capacity(), // count bounded
        empty() == (count() == 0) // empty if no elements
    )

    // Initialization.

    CONTRACT_CONSTRUCTOR_TPL( // Allocate stack for a maximum of n elements.
        public explicit (stack4) ( (const int&) n )
            precondition(
                n >= 0 // non negative capacity
            )
            postcondition(
                capacity() == n // capacity set
            )
    ) ; // Deferred body definition (see bottom).

    CONTRACT_CONSTRUCTOR_TPL( // Deep copy.
        public (stack4) ( (const stack4&) right )
            postcondition(
                capacity() == right.capacity(),
                count() == right.count()
                // all items equal right's items one by one
            )
    ) ;

    CONTRACT_FUNCTION_TPL( // Deep assignment.
        public (stack4&) operator(=)(assign) ( (const stack4&) right )
            postcondition(
                capacity() == right.capacity(),
                count() == right.count()
                // all items equal right's items one by one
            )
    ) ;

    CONTRACT_DESTRUCTOR_TPL( // Destroy this stack.
        public virtual (~stack4) ( void )
    ) ;

    // Access.

    CONTRACT_FUNCTION_TPL( // Max number of stack elements.
        public int (capacity) ( void ) const
    ) ;

    CONTRACT_FUNCTION_TPL( // Number of stack elements.
        public int (count) ( void ) const
```

The Eiffel Programming Language

```
-- File: stack4.e
-- Extra spaces, newlines, etc used to align text with this library code.




indexing
    destription: "Dispenser with LIFO access policy and a fixed max capacity."
class interface STACK4[G] creation make

invariant
    count_non_negative: count >= 0
    count_bounded: count <= capacity
    empty_if_no_elements: empty = (count = 0)


feature -- Initialization.

    -- Allocate stack for a maximum of n elements.
    make ( n : INTEGER ) is
        require
            non_negative_capacity: n >= 0

        ensure
            capacity_set: capacity = n

        end
```

```
feature -- Access.

    -- Max number of stack elements.
    capacity : INTEGER


    -- Number of stack elements.
    count : INTEGER
```

This Library (C++03)

```
    ) ;

    CONTRACT_FUNCTION_TPL( // Top element.
        public (const T&) (item) ( void ) const
            precondition(
                not empty() // not empty (i.e., count > 0)
            )
    ) ;

    // Status report.

    CONTRACT_FUNCTION_TPL( // Is stack empty?
        public bool (empty) ( void ) const
            postcondition(
                auto result = return,
                result == (count() == 0) // empty definition
            )
    ) ;

    CONTRACT_FUNCTION_TPL( // Is stack full?
        public bool (full) ( void ) const
            postcondition(
                auto result = return,
                result == (count() == capacity()) // full definition
            )
    ) ;

    // Element change.

    CONTRACT_FUNCTION_TPL( // Add x on top.
        public void (put) ( (const T&) x )
            precondition(
                not full() // not full
            )
            postcondition(
                auto old_count = CONTRACT_OLDOF count(),
                not empty(), // not empty
                item() == x, // added to top
                count() == old_count + 1 // one more item
            )
    ) ;

    CONTRACT_FUNCTION_TPL( // Remove top item.
        public void (remove) ( void )
            precondition(
                not empty() // not empty (i.e., count > 0)
            )
            postcondition(
                auto old_count = CONTRACT_OLDOF count(),
                not full(), // not full
                count() == old_count - 1 // one fewer item
            )
    ) ;

    private: int capacity_;
    private: int count_;
    private: T* representation_; // Using C-style array.
};

// Implementation.
```

The Eiffel Programming Language

```
    -- Top element
    item : G is
        require
            not_empty: not empty -- i.e., count > 0

        end

feature -- Status report.

    -- Is stack empty?
    empty : BOOLEAN is
        ensure

            empty_definition: result = (count = 0)

        end

    -- Is stack full?
    full : BOOLEAN is
        ensure

            full_definition: result = (count = capacity)

        end

feature -- Element change.

    -- Add x on top.
    put ( x : G ) is
        require
            not_full: not full

        ensure

            not_empty: not empty
            added_to_top: item = x
            one_more_item: count = old count + 1

        end

    -- Remove top element.
    remove is
        require
            not_empty: not empty -- i.e., count > 0

        ensure

            not_full: not full
            one_fewer_item: count = old count - 1

        end




end -- class interface STACK4
```

This Library (C++03)	The Eiffel Programming Language

```
template< typename T >
CONTRACT_CONSTRUCTOR_BODY(stack4<T>, stack4) ( const int& n )
{
    capacity_ = n;
    count_ = 0;
    representation_ = new T[n];
}

template< typename T >
CONTRACT_CONSTRUCTOR_BODY(stack4<T>, stack4) ( const stack4& right )
{
    capacity_ = right.capacity_;
    count_ = right.count_;
    representation_ = new T[right.capacity_];
    for(int i = 0; i < right.count_; ++i)
        representation_[i] = right.representation_[i];
}

template< typename T >
stack4<T>& stack4<T>::CONTRACT_MEMBER_BODY(operator(=)(assign)) (
        const stack4& right )
{
    delete[] representation_;
    capacity_ = right.capacity_;
    count_ = right.count_;
    representation_ = new T[right.capacity_];
    for(int i = 0; i < right.count_; ++i)
        representation_[i] = right.representation_[i];
    return *this;
}

template< typename T >
CONTRACT_DESTRUCTOR_BODY(stack4<T>, ~stack4) ( void )
{
    delete[] representation_;
}

template< typename T >
int stack4<T>::CONTRACT_MEMBER_BODY(capacity) ( void ) const
{
    return capacity_;
}

template< typename T >
int stack4<T>::CONTRACT_MEMBER_BODY(count) ( void ) const
{
    return count_;
}

template< typename T >
const T& stack4<T>::CONTRACT_MEMBER_BODY(item) ( void ) const
{
    return representation_[count() - 1];
}

template< typename T >
bool stack4<T>::CONTRACT_MEMBER_BODY(empty) ( void ) const
{
    return count() == 0;
```

This Library (C++03)	The Eiffel Programming Language

```
}

template< typename T >
bool stack4<T>::CONTRACT_MEMBER_BODY(full) ( void ) const
{
    return count() == capacity();
}

template< typename T >
void stack4<T>::CONTRACT_MEMBER_BODY(put) ( const T& x )
{
    representation_[count_++] = x;
}

template< typename T >
void stack4<T>::CONTRACT_MEMBER_BODY(remove) ( void )
{
    --count_;
}

#endif // #include guard
```

```
// File: stack4_main.cpp
#include "stack4.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    stack4<int> s(3);
    BOOST_TEST(s.capacity() == 3);
    BOOST_TEST(s.count() == 0);
    s.put(123);
    BOOST_TEST(not s.empty());
    BOOST_TEST(not s.full());
    BOOST_TEST(s.item() == 123);
    s.remove();
    BOOST_TEST(s.empty());
    return boost::report_errors();
}
```

[Meyer97] Stack3: Error codes instead of preconditions

```
// File: stack3.cpp
#include "stack4.hpp"
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>

CONTRACT_CLASS( // Dispenser LIFO and max capacity using error codes.
    template( typename T )
    class (stack3)
) {
    CONTRACT_CLASS_INVARIANT_TPL( void ) // no class invariant

    public: enum Error { // Error codes.
        NO_ERROR = 0,
        OVERFLOW_ERROR,
        UNDERFLOW_ERROR,
        SIZE_ERROR
    };

    // Initialization.

    // Create stack for max of n elements, if n < 0 set error (no precondition).
    CONTRACT_CONSTRUCTOR_TPL(
        public (stack3) ( (const int&) n, (const T&) none, default T() )
            postcondition(
                bool(n < 0) == (error() == SIZE_ERROR), // error if impossible
                bool(n >= 0) == !error(), // no error if possible
                if(bool(!error())) ( capacity() == n ) // created if no error
            )
            initialize( none_(none), representation_(0), error_(NO_ERROR) )
    ) {
        if(n >= 0) representation_ = stack4<T>(n);
        else error_ = SIZE_ERROR;
    }

    CONTRACT_DESTRUCTOR_TPL( // Destroy stack.
        public virtual (~stack3) ( void )
    ) {}

    // Access.

    CONTRACT_FUNCTION_TPL( // Max number of stack elements.
        public int (capacity) ( void ) const
    ) {
        return representation_.capacity();
    }

    CONTRACT_FUNCTION_TPL( // Number of stack elements.
        public int (count) ( void ) const
    ) {
        return representation_.count();
    }

    // Top element if present, otherwise set error (no precondition).
    CONTRACT_FUNCTION_TPL(
        public (const T&) (item) ( void ) const
            postcondition(
                empty() == (error() == UNDERFLOW_ERROR), // error if impossible
                bool(!empty() == !error()) // no error if possible
            )
    ) {
```

```
    if(!empty()) {
        error_ = NO_ERROR;
        return representation_.item();
    } else {
        error_ = UNDERFLOW_ERROR;
        return none_;
    }
}

// Status report.

CONTRACT_FUNCTION_TPL( // Error indicator set by various operations.
    public (Error) (error) ( void ) const
) {
    return error_;
}

CONTRACT_FUNCTION_TPL( // Is stack empty?
    public bool (empty) ( void ) const
) {
    return representation_.empty();
}

CONTRACT_FUNCTION_TPL( // Is stack full?
    public bool (full) ( void ) const
) {
    return representation_.full();
}

// Element change.

// Add x to top if capacity left, otherwise set error (no precondition).
CONTRACT_FUNCTION_TPL(
    public void (put) ( (const T&) x )
        postcondition(
            auto old_full = CONTRACT_OLDOF full(),
            auto old_count = CONTRACT_OLDOF count(),
            old_full == (error() == OVERFLOW_ERROR), // error if impossible
            not old_full == not error(), // no error if possible
            if(not error()) (
                not empty(), // not empty if no error
                item() == x, // added to top is no error
                count() == old_count + 1 // one more item if no error
            )
        )
) {
    if(full()) {
        error_ = OVERFLOW_ERROR;
    } else {
        representation_.put(x);
        error_ = NO_ERROR;
    }
}

// Remove top item if possible, otherwise set error to (no precondition).
CONTRACT_FUNCTION_TPL(
    public void (remove) ( void )
        postcondition(
            auto old_empty = CONTRACT_OLDOF empty(),
            auto old_count = CONTRACT_OLDOF count(),
            old_empty == (error() == UNDERFLOW_ERROR), // if impossible
            not old_empty == not error(), // no error if possible
            if(not error()) (
```

```
                    not full(), // not full is no error
                    count() == old_count - 1 // one fewer item if no error
                )
            )
    ) {
        if(empty()) {
            error_ = UNDERFLOW_ERROR;
        } else {
            representation_.remove();
            error_ = NO_ERROR;
        }
    }

    private: const T none_;
    private: stack4<T> representation_;
    private: mutable Error error_; // Mutable for logic constantness.
};

int main ( void )
{
    stack3<int> s(3);
    BOOST_TEST(s.capacity() == 3);
    BOOST_TEST(s.count() == 0);
    BOOST_TEST(s.empty());
    BOOST_TEST(not s.full());
    s.put(123);
    BOOST_TEST(s.item() == 123);
    s.remove();
    BOOST_TEST(s.empty());
    return boost::report_errors();
}
```

[Meyer97] GCD: Loop variants and invariants plus comparison with Eiffel syntax

This Library (C++03)

```
// File: gcd.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <algorithm>

CONTRACT_FUNCTION( // Great common divisor of given positive integers.
    int (gcd) ( int a, int b )
        precondition(
            a > 0, b > 0
        )
) {
    int x = a, y = b;

    CONTRACT_LOOP( while(x != y) ) { // Loop contracted with a variant.
        // Block invariant assert loop invariant.
        CONTRACT_BLOCK_INVARIANT( const( x ) x > 0, const( y ) y > 0 )
        // Loop variant is non-negative and monotonically decreasing.
        CONTRACT_LOOP_VARIANT( const( x, y ) std::max(x, y) )

        // `x` and `y` have the same GCD as `a` and `b`.

        if(x > y) x = x - y; else y = y - x;
    }
    return x;
}

int main ( void )
{
    BOOST_TEST(gcd(12, 18) == 6);
    BOOST_TEST(gcd(4, 14) == 2);
    return boost::report_errors();
}
```

The Eiffel Programming Language

```
-- File: gcd.e
-- Extra spaces, newlines, etc used to align text with this library code.



-- Greatest common divisor of a and b.
gcd ( a, b : INTEGER ) : INTEGER is
    require
        a > 0; b > 0
    local
        x, y : INTEGER
    do
        from
            x := a; y := b
        invariant
            x > 0; y > 0
        variant
            x.max(y)
        until
            x = y
        loop
            if x > y then x := x - y else y := y - x end
        end
        Result := x
    end
```

[Meyer97] Max-Array: Nested loop variants and invariants

```
// File: maxarray.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <algorithm>

CONTRACT_FUNCTION(
    int (maxarray) ( (int const* const) array, (size_t const&) size )
        precondition(
            array, // array allocated
            size >= 1 // size in range
        )
) {
    int maxnow = array[0];
    CONTRACT_LOOP( for(size_t i = 0; i < (size - 1); ++i) ) {
        // Nested loop (with variant) used to assert enclosing loop invariants.
        CONTRACT_LOOP( for(size_t j = 0; j < i; ++j) ) {
            CONTRACT_LOOP_VARIANT( const( i, j ) i - j )
            CONTRACT_BLOCK_INVARIANT( const( maxnow, array, j )
                    maxnow >= array[j] )
        }
        // -2 because starts from 0 (not 1) and already done element at 0.
        CONTRACT_LOOP_VARIANT( const( size, i ) size - i - 2 )

        maxnow = std::max(maxnow, array[i]);
    }
    return maxnow;
}

int main ( void )
{
    int a[] = {1, 5, 3};
    BOOST_TEST(maxarray(a, 3) == 5);
    return boost::report_errors();
}
```

[Mitchell02] Name List: Relaxed subcontracts

```
// File: name_list.hpp
#ifndef NAME_LIST_HPP_
#define NAME_LIST_HPP_

#include <contract.hpp>
#include <string>
#include <list>

CONTRACT_CLASS( // List of names.
    class (name_list)
) {
    CONTRACT_CLASS_INVARIANT( count() >= 0 ) // non-negative count

    CONTRACT_CONSTRUCTOR( // Create an empty list.
        public (name_list) ( void )
            postcondition( count() == 0 ) // empty list
    ) ; // Deferred body definition.

    CONTRACT_DESTRUCTOR( // Destroy list.
        public virtual (~name_list) ( void )
    ) ;

    CONTRACT_FUNCTION( // Number of names in list.
        public int (count) ( void ) const
            // postcondition: non-negative count already in class invariants
    ) ;

    CONTRACT_FUNCTION( // Is name in list?
        public bool (has) ( (std::string const&) name ) const
            postcondition(
                auto result = return,
                if(count() == 0) ( not result ) // does not have is empty
            )
    ) ;

    CONTRACT_FUNCTION( // Add name to list.
        public virtual void (put) ( (std::string const&) name )
            precondition(
                not has(name) // not already in list
            )
            postcondition(
                auto old_has_name = CONTRACT_OLDOF has(name),
                auto old_count = CONTRACT_OLDOF count(),
                // Following if-guard allows to relax subcontracts.
                if(not old_has_name) (
                    has(name), // name on list
                    count() == old_count + 1 // number of names increased
                )
            )
    ) ;

    private: std::list<std::string> names_;
};

CONTRACT_CLASS( // List of names that allows for duplicates.
    class (relaxed_name_list) extends( public name_list ) // Subcontracting.
) {
    CONTRACT_CLASS_INVARIANT( void ) // no subcontracted invariants

    // Creation.
```

```
    CONTRACT_CONSTRUCTOR( // Create an empty list.
        public (relaxed_name_list) ( void )
            postcondition( count() == 0 )
    ) ;

    CONTRACT_DESTRUCTOR( // Destroy list.
        public virtual (~relaxed_name_list) ( void )
    ) ;

    // Commands.

    CONTRACT_FUNCTION( // Add name to list.
        public void (put) ( (std::string const&) name )
            precondition( // Relax inherited precondition.
                has(name) // already in list
            )
            postcondition(
                auto old_has_name = CONTRACT_OLDOF has(name),
                auto old_count = CONTRACT_OLDOF count(),
                // Inherited postconditions not checked because of its if-guard.
                if(old_has_name) (
                    count() == old_count // unchanged if name already in list
                )
            )
    ) ;
};

#endif // #include guard
```

```
// File: name_list.cpp
#include "name_list.hpp"
#include <algorithm>

// name_list

CONTRACT_CONSTRUCTOR_BODY(name_list, name_list) ( void ) {}

CONTRACT_DESTRUCTOR_BODY(name_list, ~name_list) ( void ) {}

int name_list::CONTRACT_MEMBER_BODY(count) ( void ) const
    { return names_.size(); }

bool name_list::CONTRACT_MEMBER_BODY(has) ( std::string const& name ) const
    { return names_.end() != std::find(names_.begin(), names_.end(), name); }

void name_list::CONTRACT_MEMBER_BODY(put) ( std::string const& name )
    { names_.push_back(name); }

// relaxed_name_list

CONTRACT_CONSTRUCTOR_BODY(relaxed_name_list, relaxed_name_list) ( void ) {}

CONTRACT_DESTRUCTOR_BODY(relaxed_name_list, ~relaxed_name_list) ( void ) {}

void relaxed_name_list::CONTRACT_MEMBER_BODY(put) ( std::string const& name )
{
    // Must use `BODY` to call base functions (to avoid infinite recursion).
    if(!has(name)) name_list::CONTRACT_MEMBER_BODY(put)(name);
}
```

```
// File: name_list_main.cpp
#include "name_list.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    relaxed_name_list rl;
    rl.put("abc");
    BOOST_TEST(rl.has("abc"));
    rl.put("abc"); // Note: Can call `put("abc")` this gain.

    name_list nl;
    nl.put("abc");
    BOOST_TEST(nl.has("abc"));
    // Note: Cannot call `put("abc")` again.
    return boost::report_errors();
}
```

[Mitchell02] Dictionary: Simple key-value map

```
// File: dictionary.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <utility>
#include <map>

CONTRACT_CLASS( // Simple dictionary.
    template( typename Key, typename Value )
    class (dictionary)
) {
    CONTRACT_CLASS_INVARIANT_TPL( count() >= 0 ) // non-negative count

    // Creation.

    CONTRACT_CONSTRUCTOR_TPL( // Create empty dictionary.
        public (dictionary) ( void )
            postcondition( count() == 0 ) // empty
    ) {}

    CONTRACT_DESTRUCTOR_TPL( // Destroy dictionary.
        public virtual (~dictionary) ( void )
    ) {}

    // Basic queries.

    CONTRACT_FUNCTION_TPL( // Number of key entires.
        public int (count) ( void ) const
            // postcondition: non-negative count asserted by class invariants
    ) {
        return items_.size();
    }

    CONTRACT_FUNCTION_TPL( // Has entry for key?
        public bool (has) ( (Key const&) key ) const
            postcondition(
                auto result = return,
                if(count() == 0) ( not result ) // empty has no key
            )
    ) {
        return items_.find(key) != items_.end();
    }
```

```
CONTRACT_FUNCTION_TPL( // Value for given key.
    public (Value const&) (value_for) ( (Key const&) key ) const
        precondition( has(key) ) // has key
) {
    return items_.find(key)->second;
}

// Commands.

CONTRACT_FUNCTION_TPL( // Put value for given key.
    public void (put) ( (Key const&) key, (Value const&) value )
        postcondition(
            auto old_count = CONTRACT_OLDOF count(),
            count() == old_count + 1, // count increased
            has(key), // has key
            value_for(key) == value // value for key set
        )
) {
    items_.insert(std::make_pair(key, value));
}

CONTRACT_FUNCTION_TPL( // Remove value for given key.
    public void (remove) ( (Key const&) key )
        precondition( has(key) ) // has key
        postcondition(
            auto old_count = CONTRACT_OLDOF count(),
            count() == old_count - 1, // count decreased
            not has(key) // does not have key
        )
) {
    items_.erase(key);
    }

    private: std::map<Key, Value> items_;
};

int main ( void )
{
    dictionary<std::string, int> ages;
    BOOST_TEST(not ages.has("john"));
    ages.put("john", 23);
    BOOST_TEST(ages.value_for("john") == 23);
    ages.remove("john");
    BOOST_TEST(ages.count() == 0);
    return boost::report_errors();
}
```

[Mitchell02] Courier: Subcontracting and static class invariants

```
// File: courier.hpp
#ifndef COURIER_HPP_
#define COURIER_HPP_

#include <contract.hpp>
#include <string>

struct package // Basic package information.
{
    public: double weight_kg; // Weight in kilograms.
    public: std::string location; // Current location.
    public: double accepted_hour; // Hour when it was accepted for delivery.
    public: double delivered_hour; // Hour when it was delivered.

    public: explicit package (
            double const a_weight_kg,
            std::string const a_location = "",
            double const an_accepted_hour = 0.0,
            double const a_delivered_hour = 0.0
        )
        :   weight_kg(a_weight_kg),
            location(a_location),
            accepted_hour(an_accepted_hour),
            delivered_hour(a_delivered_hour)
    {}
};

CONTRACT_CLASS( // Basic courier for package delivery.
    class (courier)
) {
    CONTRACT_CLASS_INVARIANT(
        insurance_cover_dollar() >= min_insurance_dollar, // above min insur.
        static class( // Static class invariants.
            min_insurance_dollar > 0.0 // positive min insurance
        )
    )

    public: static double min_insurance_dollar;

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create courier with specified insurance value.
        public explicit (courier) (
                double const an_insurance_cover_dollar,
                        default min_insurance_dollar
            )
            precondition( an_insurance_cover_dollar > 0.0 ) // positive insur.
    ) ; // Deferred body definition.

    CONTRACT_DESTRUCTOR( // Destroy courier.
        public virtual (~courier) ( void )
    ) ;

    // Queries.

    CONTRACT_FUNCTION( // Return insurance cover.
        public double (insurance_cover_dollar) ( void ) const
    ) ;

    // Commands.
```

```
    CONTRACT_FUNCTION( // Deliver package to destination.
        public virtual void (deliver) ( (package&) the_package,
                (std::string const) destination )
            precondition(
                the_package.weight_kg < 5.0 // within max wight
            )
            postcondition(
                double(the_package.delivered_hour - the_package.accepted_hour)
                        <= 3.0, // within max delivery time
                the_package.location == destination // delivered at destination
            )
    ) ;

    private: double insurance_cover_dollar_;
};

CONTRACT_CLASS( // Different courier for package delivery.
    class (different_courier) extends( public courier )
) {
    CONTRACT_CLASS_INVARIANT(
        insurance_cover_dollar() >= different_insurance_dollar,
        static class(
            different_insurance_dollar >= courier::min_insurance_dollar
        )
    )

    public: static double different_insurance_dollar;

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create currier with specified insurance value.
        public explicit (different_courier) (
                double const insurance_cover_dollar,
                        default different_insurance_dollar
            )
            precondition( insurance_cover_dollar > 0.0 )
            initialize( courier(insurance_cover_dollar) )
    ) {} // Cannot separated body definition because has member initializers.

    CONTRACT_DESTRUCTOR( // Destroy courier.
        public virtual (~different_courier) ( void )
    ) ;

    // Commands.

    CONTRACT_FUNCTION(
        public virtual void (deliver) ( (package&) the_package,
                (std::string const) destination ) override final
            precondition(
                // Weaker precondition on weight (it can weight more).
                the_package.weight_kg <= 8.0
            )
            postcondition(
                // Stronger postcondition on delivery time (faster delivery).
                double(the_package.delivered_hour - the_package.accepted_hour)
                        <= 2.0
                // Inherits "delivered at destination" postcondition.
            )
    ) ;
};

#endif // #include guard
```

```
// File: courier.cpp
#include "courier.hpp"

// Courier.

double courier::min_insurance_dollar = 10.0e+6;

CONTRACT_CONSTRUCTOR_BODY(courier, courier) (
        double const an_insurance_cover_dollar )
{
    insurance_cover_dollar_ = an_insurance_cover_dollar;
}

CONTRACT_DESTRUCTOR_BODY(courier, ~courier) ( void ) {}

double courier::CONTRACT_MEMBER_BODY(insurance_cover_dollar) ( void ) const
    { return insurance_cover_dollar_; }

void courier::CONTRACT_MEMBER_BODY(deliver) ( package& the_package,
        std::string const destination )
{
    the_package.location = destination;
    // Delivery takes 2.5 hours.
    the_package.delivered_hour = the_package.accepted_hour + 2.5;
}

// Different courier.

double different_courier::different_insurance_dollar = 20.0e+6;

CONTRACT_DESTRUCTOR_BODY(different_courier, ~different_courier) ( void ) {}

void different_courier::CONTRACT_MEMBER_BODY(deliver) ( package& the_package,
        std::string const destination )
{
    the_package.location = destination;
    // Delivery takes only 0.5 hours.
    the_package.delivered_hour = the_package.accepted_hour + 0.5;
}
```

```
// File: courier_main.cpp
#include "courier.hpp"

int main ( void )
{
    courier c;
    different_courier dc;
    package cups(3.6, "store");
    package desk(7.2, "store");
    c.deliver(cups, "home");
    dc.deliver(desk, "office");
    return 0;
}
```

[Mitchell02] Stack: Simple stack dispenser

```
// File: stack.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>

CONTRACT_CLASS( // Simple stack.
    template( typename T )
    class (stack)
) {
    CONTRACT_CLASS_INVARIANT_TPL( count() >= 0 ) // non-negative count

    // Creation.

    CONTRACT_CONSTRUCTOR_TPL( // Create empty stack.
        public (stack) ( void )
            postcondition( count() == 0 ) // empty
    ) {}

    CONTRACT_DESTRUCTOR_TPL( // Destroy stack.
        public virtual (~stack) ( void )
    ) {}

    // Basic queries.

    CONTRACT_FUNCTION_TPL( // Number of items.
        public int (count) ( void ) const
    ) {
        return items_.size();
    }

    CONTRACT_FUNCTION_TPL( // Item at index in [1, count()] (as in Eiffel).
        public (T const&) (item_at) ( int const index ) const
            precondition(
                index > 0, // positive index
                index <= count() // index within count
            )
    ) {
        return items_.at(index - 1);
    }

    // Derived queries.

    CONTRACT_FUNCTION_TPL( // If no items.
        public bool (is_empty) ( void ) const
            postcondition(
                auto result = return,
                result == (count() == 0) // consistent with count
            )
    ) {
        return count() == 0;
    }

    CONTRACT_FUNCTION_TPL( // Top item.
        public (T const&) (item) ( void ) const
            precondition( count() > 0 ) // not empty
            postcondition(
                auto result = return,
                result == item_at(count()) // item on top
            )
    ) {
```

```
        return item_at(count());
    }

    // Commands.

    CONTRACT_FUNCTION_TPL( // Push item to the top.
        public void (put) ( (T const&) new_item )
            postcondition(
                auto old_count = CONTRACT_OLDOF count(),
                count() == old_count + 1, // count increased
                item() == new_item // item set
            )
    ) {
        items_.push_back(new_item);
    }

    CONTRACT_FUNCTION_TPL( // Pop top item.
        public void (remove) ( void )
            precondition( count() > 0 ) // not empty
            postcondition(
                auto old_count = CONTRACT_OLDOF count(),
                count() == old_count - 1 // count decreased
            )
    ) {
        items_.resize(items_.size() - 1);
    }

    private: std::vector<T> items_;
};

int main ( void )
{
    stack<int> s;
    BOOST_TEST(s.count() == 0);
    s.put(123);
    BOOST_TEST(s.item_at(1) == 123);
    s.remove();
    BOOST_TEST(s.is_empty());
    return boost::report_errors();
}
```

[Mitchell02] Simple Queue: Simple queue dispenser

```
// File: simple_queue.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <vector>

// Assertion requirements used to model assertion computational complexity.
#define O_1 0   // O(1) constant (default)
#define O_N 1   // O(n) linear
#define COMPLEXITY_MAX O_1

CONTRACT_CLASS(
    template( typename T )
    class (simple_queue)
) {
    CONTRACT_CLASS_INVARIANT( count() >= 0 ) // non-negative count

    // Creation.

    CONTRACT_CONSTRUCTOR_TPL( // Create empty queue.
        public explicit (simple_queue) ( int const the_capacity )
            precondition( the_capacity > 0 ) // positive capacity
            postcondition(
                capacity() == the_capacity, // capacity set
                is_empty() // empty
            )
    ) {
        items_.reserve(the_capacity);
    }

    CONTRACT_DESTRUCTOR_TPL( // Destroy queue.
        public virtual (~simple_queue) ( void )
    ) {}

    // Basic queries.

    CONTRACT_FUNCTION_TPL( // Items in the queue (in their order).
        public (std::vector<T> const&) (items) ( void ) const
    ) {
        return items_;
    }

    CONTRACT_FUNCTION_TPL( // Max number of items queue can hold.
        public int (capacity) ( void ) const
    ) {
        return items_.capacity();
    }

    // Derived queries.

    CONTRACT_FUNCTION_TPL( // Number of items.
        public int (count) ( void ) const
            postcondition(
                auto result = return,
                result == int(items().size()) // return items count
            )
    ) {
        return items_.size();
    }

    CONTRACT_FUNCTION_TPL( // Item at head.
```

```
    public (T const&) (head) ( void ) const
        precondition( not is_empty() )
        postcondition(
            auto result = return,
            result == items().at(0) // return item on top
        )
) {
    return items_.at(0);
}

CONTRACT_FUNCTION_TPL( // If queue contains no item.
    public bool (is_empty) ( void ) const
        postcondition(
            auto result = return,
            result == (count() == 0) // consistent with count
        )
) {
    return items_.size() == 0;
}

CONTRACT_FUNCTION_TPL( // If queue has no room for another item.
    public bool (is_full) ( void ) const
        postcondition(
            auto result = return,
            // consistent with size and capacity
            result == (capacity() == int(items().size()))
        )
) {
    return items_.size() == items_.capacity();
}

// Commands.

CONTRACT_FUNCTION_TPL( // Remove head item and shift all other items.
    public void (remove) ( void )
        precondition( not is_empty() )
        postcondition(
            auto old_count = CONTRACT_OLDOF count(),
            auto old_items = CONTRACT_OLDOF items(),
            count() == old_count - 1, // count decreased
            // Expensive assertion to check so marked with its complexity.
            all_equal(items(), old_items, 1 /* shifted */),
                    requires O_N <= COMPLEXITY_MAX
        )
) {
    items_.erase(items_.begin());
}

CONTRACT_FUNCTION_TPL( // Add item to tail.
    public void (put) ( (T const&) item )
        precondition( count() < capacity() )
        postcondition(
            auto old_count = CONTRACT_OLDOF count(),
            auto old_items = CONTRACT_OLDOF items(),
            count() == old_count + 1, // count increased
            items().at(count() - 1) == item, // second last item
            if(count() >= 2) (
                // Computationally expensive assertion to check.
                all_equal(items(), old_items),
                        requires O_N <= COMPLEXITY_MAX
            )
        )
) {
```

```
        items_.push_back(item);
    }

    CONTRACT_FUNCTION_TPL( // Contract helper.
        private static bool (all_equal) (
                (std::vector<T> const&) left,
                (std::vector<T> const&) right,
                size_t offset, default 0
            )
            precondition(
                right.size() == left.size() + offset // correct offset
            )
    ) {
        for(size_t i = offset; i < right.size(); ++i)
            if(left.at(i - offset) != right.at(i)) return false;
        return true;
    }

    private: std::vector<T> items_;
};

int main ( void )
{
    simple_queue<int> q(10);
    q.put(123);
    q.put(456);
    std::vector<int> const& items = q.items();
    BOOST_TEST(items[0] == 123);
    BOOST_TEST(items[1] == 456);
    BOOST_TEST(q.capacity() == 10);
    BOOST_TEST(q.head() == 123);
    BOOST_TEST(not q.is_empty());
    BOOST_TEST(not q.is_full());
    q.remove();
    BOOST_TEST(q.count() == 1);
    return boost::report_errors();
}
```

[Mitchell02] Customer Manager: Contracts instead of Defensive Programming

```
// File: customer_manager.hpp
#ifndef CUSTOMER_MANAGER_HPP_
#define CUSTOMER_MANAGER_HPP_

#include <contract.hpp>
#include <string>
#include <map>

class basic_customer_details // Basic customer information.
{
    friend class customer_manager;

    public: typedef std::string identifier;

    public: explicit basic_customer_details ( identifier const& an_id )
        : id(an_id), name(), address(), birthday()
    {}

    protected: identifier id; // Customer identifier.
    protected: std::string name; // Customer name.
    protected: std::string address; // Customer address.
    protected: std::string birthday; // Customer date of birth.
};

CONTRACT_CLASS( // Manage customers.
    class (customer_manager)
) {
    CONTRACT_CLASS_INVARIANT( count() >= 0 ) // non-negative count

    // Creation.

    CONTRACT_CONSTRUCTOR(
        public (customer_manager) ( void )
            // LIMITATION: Cannot use member initializes because deferring
            // body definition.
    ) ; // Deferred body definition.

    CONTRACT_DESTRUCTOR(
        public virtual (~customer_manager) ( void )
    ) ;

    // Basic queries.

    CONTRACT_FUNCTION(
        public int (count) ( void ) const
            // postcondition: non-negative count asserted by class invariants
    ) ;

    CONTRACT_FUNCTION( // There is a customer with given identifier.
        public bool (id_active) (
                (basic_customer_details::identifier const&) id ) const
    ) ;

    // Derived queries.

    CONTRACT_FUNCTION( // Name of customer with given identifier.
        public (std::string const&) (name_for) (
                (basic_customer_details::identifier const&) id ) const
            precondition( id_active(id) ) // id active
    ) ;
```

```
// Commands.

CONTRACT_FUNCTION( // Add given customer.
    public void (add) ( (basic_customer_details const&) details )
        precondition( not id_active(details.id) ) // id not active
        postcondition(
            auto old_count = CONTRACT_OLDOF count(),
            count() == old_count + 1, // count increased
            id_active(details.id) // id active
        )
) ;

CONTRACT_FUNCTION( // Set name of customer with given identifier.
    public void (set_name) (
            (basic_customer_details::identifier const&) id,
            (std::string const&) name
        )
        precondition( id_active(id) ) // id active
        postcondition( name_for(id) == name ) // name set
) ;

private: class agent {}; // Customer agent.

private: struct customer : basic_customer_details // Basic customer.
{
    public: agent managing_agent; // Customer agent.
    public: std::string last_contact; // Customer last contacted.

    public: explicit customer ( basic_customer_details const& details )
        : basic_customer_details(details), managing_agent(),
                last_contact()
        {}
    };

    private: std::map<basic_customer_details::identifier, customer> customers_;
};

#endif // #include guard
```

```
// File: customer_manager.cpp
#include "customer_manager.hpp"
#include <utility>

CONTRACT_CONSTRUCTOR_BODY(customer_manager, customer_manager) ( void ) {}

CONTRACT_DESTRUCTOR_BODY(customer_manager, ~customer_manager) ( void ) {}

int customer_manager::CONTRACT_MEMBER_BODY(count) ( void ) const
    { return customers_.size(); }

bool customer_manager::CONTRACT_MEMBER_BODY(id_active) (
        basic_customer_details::identifier const& id) const
{
    return customers_.find(id) != customers_.end();
}

std::string const& customer_manager::CONTRACT_MEMBER_BODY(name_for) (
        basic_customer_details::identifier const& id) const
{
    // Find != end because of `id_active()` pre so no defensive programming.
    return customers_.find(id)->second.name;
}

void customer_manager::CONTRACT_MEMBER_BODY(add) (
        basic_customer_details const& details )
    { customers_.insert(std::make_pair(details.id, customer(details))); }

void customer_manager::CONTRACT_MEMBER_BODY(set_name) (
        basic_customer_details::identifier const& id,
        std::string const& name )
{
    // Find != end because of `id_active()` pre so no defensive programming.
    customers_.find(id)->second.name = name;
}
```

```
// File: customer_manager_main.cpp
#include "customer_manager.hpp"
#include <boost/detail/lightweight_test.hpp>

int main ( void )
{
    customer_manager mgr;
    basic_customer_details d("id1");

    mgr.add(d);
    mgr.set_name("id1", "abc");
    BOOST_TEST(mgr.name_for("id1") == "abc");
    BOOST_TEST(mgr.count() == 1);
    BOOST_TEST(mgr.id_active("id1"));
    return boost::report_errors();
}
```

[Mitchell02] Observer: Contracts for pure virtual functions

```
// File: observer/observer.hpp
#ifndef OBSERVER_HPP_
#define OBSERVER_HPP_

#include <contract.hpp>

CONTRACT_CLASS( // Observer.
    class (observer)
) {
    CONTRACT_CLASS_INVARIANT( void ) // no invariant

    friend class subject;

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create observer.
        public (observer) ( void )
    ) {}

    CONTRACT_DESTRUCTOR( // Destroy observer.
        public virtual (~observer) ( void )
    ) {}

    // Commands.

    CONTRACT_FUNCTION( // If up to date with its subject.
        protected virtual bool (up_to_date_with_subject) ( void ) const new
    ) = 0;

    CONTRACT_FUNCTION( // Update and inform its subject.
        protected virtual void (update) ( void ) new
            postcondition( up_to_date_with_subject() ) // up-to-date
    ) = 0;
};

#endif // #include guard
```

```
// File: observer/subject.hpp
#ifndef SUBJECT_HPP_
#define SUBJECT_HPP_

#include "observer.hpp"
#include <contract.hpp>
#include <list>
#include <algorithm>

// Assertion requirements used to model assertion computational complexity.
#define O_1 0   // O(1) constant (default)
#define O_N 1   // O(n) linear
#define COMPLEXITY_MAX O_1

CONTRACT_CLASS( // Subject for observer design pattern.
    class (subject)
) {
    CONTRACT_CLASS_INVARIANT(
        all_observers_valid(observers()), // observes valid
                requires O_N <= COMPLEXITY_MAX
    )

    // Creation.

    CONTRACT_CONSTRUCTOR( // Construct subject with no observer.
        public (subject) ( void )
    ) {}

    CONTRACT_DESTRUCTOR( // Destroy subject.
        public virtual (~subject) ( void )
    ) {}

    // Queries.

    CONTRACT_FUNCTION( // If given observer is attached.
        public bool (attached) ( (observer const* const) obs ) const
            precondition( obs ) // not null
    ) {
        return std::find(observers_.begin(), observers_.end(), obs) !=
                observers_.end();
    }

    // Commands.

    CONTRACT_FUNCTION( // Remember given object as an observer.
        public void (attach) ( (observer* const) obs )
            precondition(
                obs, // not null
                not attached(obs) // not already attached
            )
            postcondition(
                auto old_observers = CONTRACT_OLDOF observers(),
                attached(obs), // attached
                other_observers_unchanged(old_observers, observers(), obs),
                        // others not changed (frame rule)
                        requires O_N <= COMPLEXITY_MAX
            )
    ) {
        observers_.push_back(obs);
    }

    // Queries.
```

```
CONTRACT_FUNCTION( // All observers attached to this subject.
    protected (std::list<observer const*>) (observers) ( void ) const
) {
    // Create list of pointers to const observers.
    std::list<const observer*> obs;
    for(std::list<observer*>::const_iterator
            i = observers_.begin(); i != observers_.end(); ++i) {
        obs.push_back(*i);
    }
    return obs;
}

// Commands.

CONTRACT_FUNCTION( // Update all attached observers.
    protected void (notify) ( void )
        postcondition( all_observers_updated(observers()) ) // all updated
) {
    for(std::list<observer*>::iterator
            i = observers_.begin(); i != observers_.end(); ++i) {
        // Class invariant ensures no null pointers in observers but
        // class invariants not checked for non-public members so check.
        CONTRACT_BLOCK_INVARIANT( const( i ) 0 != *i ) // pointer not null
        (*i)->update();
    }
}

// Contract helpers.

CONTRACT_FUNCTION(
    private static bool (all_observers_valid) (
            (std::list<observer const*> const&) observers )
) {
    for(std::list<observer const*>::const_iterator
            i = observers.begin(); i != observers.end(); ++i) {
        if(!(*i)) return false;
    }
    return true;
}

CONTRACT_FUNCTION(
    private bool (other_observers_unchanged) (
            (std::list<observer const*> const&) old,
            (std::list<observer const*> const&) now,
            (observer const*) obs
        ) const
        precondition( obs ) // not null
) {
    std::list<observer const*> remaining = now;
    std::remove(remaining.begin(), remaining.end(), obs);

    std::list<observer const*>::const_iterator
            remaining_it = remaining.begin();
    std::list<observer const*>::const_iterator old_it = old.begin();
    while(remaining.end() != remaining_it && old.end() != old_it) {
        if(*remaining_it != *old_it) return false;
        ++remaining_it;
        ++old_it;
    }
    return true;
}

CONTRACT_FUNCTION(
```

```
        private bool (all_observers_updated) (
                (std::list<observer const*> const&) observers ) const
    ) {
        for(std::list<observer const*>::const_iterator
                i = observers.begin(); i != observers.end(); ++i) {
            if(!(*i)) return false;
            if(!(*i)->up_to_date_with_subject()) return false;
        }
        return true;
    }

    private: std::list<observer*> observers_;
};

#endif // #include guard
```

```
// File: observer_main.cpp
#include "observer/observer.hpp"
#include "observer/subject.hpp"
#include <boost/detail/lightweight_test.hpp>
#include <contract.hpp>

int state_check; // For unit testing.

class concrete_subject : public subject // Implement an actual subject.
{
    public: typedef int state; // Some state being observed.

    public: concrete_subject ( void ) : state_() {}

    public: void set_state (state const& the_state) {
        state_ = the_state;
        BOOST_TEST(state_ == state_check);
        notify(); // Notify observers.
    }

    public: state get_state ( void ) const { return state_; }

    private: state state_;
};

CONTRACT_CLASS( // Implement of actual observer.
    class (concrete_observer) extends( public observer )
) {
    CONTRACT_CLASS_INVARIANT( void )

    CONTRACT_CONSTRUCTOR( // Create concrete observer.
        public explicit (concrete_observer) (
                (concrete_subject const&) the_subject )
            initialize( subject_(the_subject), observed_state_() )
    ) {}

    // Implement base virtual functions.

    CONTRACT_FUNCTION(
        private bool (up_to_date_with_subject) ( void ) const override final
    ) {
        return true; // For simplicity, always up-to-date.
    }

    CONTRACT_FUNCTION(
        private void (update) ( void ) override final
```

```
    ) {
        observed_state_ = subject_.get_state();
        BOOST_TEST(observed_state_ == state_check);
    }

    private: concrete_subject const& subject_;
    private: concrete_subject::state observed_state_;
};

int main ( void )
{
    concrete_subject sbj;
    concrete_observer ob(sbj);
    sbj.attach(&ob);
    sbj.set_state(state_check = 123);
    sbj.set_state(state_check = 456);
    return boost::report_errors();
}
```

[Mitchell02] Counter: Subcontracting and virtual specifiers (final, override, new, and pure virtual)

```
// File: counter/push_button.hpp
#ifndef PUSH_BUTTON_HPP_
#define PUSH_BUTTON_HPP_

#include <contract.hpp>

CONTRACT_CLASS( // Basic button.
    class (push_button)
) {
    CONTRACT_CLASS_INVARIANT( void ) // no invariant

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create an enabled button.
        public (push_button) ( void )
            postcondition( enabled() ) // enabled
            initialize( enabled_(true) )
    ) {}

    CONTRACT_DESTRUCTOR( // Destroy button.
        public virtual (~push_button) ( void )
    ) {}

    // Queries.

    CONTRACT_FUNCTION( // If button enabled.
        public bool (enabled) ( void ) const
    ) {
        return enabled_;
    }

    // Commands.

    CONTRACT_FUNCTION( // Enable this button.
        public void (enable) ( void )
            postcondition( enabled() ) // enabled
    ) {
        enabled_ = true;
    }
```

```
    CONTRACT_FUNCTION( // Disable this button.
        public void (disable) ( void )
            postcondition( not enabled() ) // disabled
    ) {
        enabled_ = false;
    }

    CONTRACT_FUNCTION( // Invoked externally when this button is clicked.
        public virtual void (on_bn_clicked) ( void ) new
            precondition( enabled() ) // enabled
    ) = 0; // Contract for pure virtual function.

    private: bool enabled_;
};

#endif // #include guard
```

```
// File: counter/decrement_button.hpp
#ifndef DECREMENT_BUTTON_HPP_
#define DECREMENT_BUTTON_HPP_

#include "push_button.hpp"
#include "counter.hpp"
#include "../observer/observer.hpp"
#include <contract.hpp>
#include <boost/utility.hpp>

CONTRACT_CLASS( // Button that decrements counter.
    class (decrement_button) final // Contract for final class.
        extends( public push_button, protected observer, boost::noncopyable )
) {
    CONTRACT_CLASS_INVARIANT( void ) // no invariant

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create button associated with given counter.
        public explicit (decrement_button) ( (counter&) the_counter )
            postcondition(
                // enabled iff positive value
                enabled() == (the_counter.value() > 0)
            )
            initialize( counter_ref_(the_counter) )
    ) {
        counter_ref_.attach(this);
    }

    CONTRACT_DESTRUCTOR( // Destroy button.
        public (~decrement_button) ( void )
    ) {}

    // Commands.

    CONTRACT_FUNCTION(
        public void (on_bn_clicked) ( void ) override
            postcondition(
                old_value = CONTRACT_OLDOF counter_ref_.value(),
                counter_ref_.value() == old_value - 1 // counter decremented
            )
    ) {
        counter_ref_.decrement();
    }
```

```
    CONTRACT_FUNCTION(
        private bool (up_to_date_with_subject) ( void ) const override
    ) {
        return true; // For simplicity, always up-to-date.
    }

    CONTRACT_FUNCTION(
        private void (update) ( void ) override
            postcondition(
                // enabled if positive value
                enabled() == (counter_ref_.value() > 0)
            )
    ) {
        if(counter_ref_.value() == 0) disable();
        else enable();
    }

    private: counter& counter_ref_;
};

#endif // #include guard
```

```
// File: counter/counter.hpp
#ifndef COUNTER_HPP_
#define COUNTER_HPP_

#include "../observer/subject.hpp"
#include <contract.hpp>

CONTRACT_CLASS( // Positive integer counter.
    class (counter) extends( public subject )
) {
    CONTRACT_CLASS_INVARIANT( void ) // no invariants

    // Creation.

    CONTRACT_CONSTRUCTOR( // Construct counter with specified value.
        public explicit (counter) ( int const a_value, default 10 )
            postcondition( value() == a_value ) // value set
            initialize( value_(a_value) )
    ) {}

    CONTRACT_DESTRUCTOR( // Destroy counter.
        public virtual (~counter) ( void )
    ) {}

    // Queries.

    CONTRACT_FUNCTION( // Current counter value.
        public int (value) ( void ) const
    ) {
        return value_;
    }

    // Commands.

    CONTRACT_FUNCTION( // Decrement counter value.
        public void (decrement) ( void )
            postcondition(
                auto old_value = CONTRACT_OLDOF value(),
                value() == old_value - 1 // decremented
            )
```

```
    ) {
        --value_;
        notify(); // Notifies all attached observers.
    }

    private: int value_;
};

#endif // #include guard
```

```
// File: counter_main.cpp
#include "counter/counter.hpp"
#include "counter/decrement_button.hpp"
#include "observer/observer.hpp"
#include <boost/detail/lightweight_test.hpp>

int counter_check;

CONTRACT_CLASS( // Show current value of associated counter.
    class (view_of_counter) extends( private observer )
) {
    CONTRACT_CLASS_INVARIANT( void ) // no invariant

    // Creation.

    CONTRACT_CONSTRUCTOR( // Create viewer associated with given counter.
        public explicit (view_of_counter) ( (counter&) the_counter )
            initialize( counter_ref_(the_counter) )
    ) {
        counter_ref_.attach(this);
        BOOST_TEST(counter_ref_.value() == counter_check);
    }

    CONTRACT_DESTRUCTOR( // Destroy viewer.
        public virtual (~view_of_counter) ( void )
    ) {}

    CONTRACT_FUNCTION(
        private bool (up_to_date_with_subject) ( void ) const override final
    ) {
        return true; // For simplicity, always up-to-date.
    }

    CONTRACT_FUNCTION(
        private void (update) ( void ) override final // Contract final func.
    ) {
        BOOST_TEST(counter_ref_.value() == counter_check);
    }

    private: counter& counter_ref_;
};

int main ( void )
{
    counter count(counter_check = 1);
    view_of_counter view(count);
    decrement_button decrement(count);
    BOOST_TEST(decrement.enabled());
```

```
    counter_check--;
    decrement.on_bn_clicked();
    BOOST_TEST(not decrement.enabled());
    return boost::report_errors();
}
```

[Stroustrup97] String: Throw when contract is broken

```
// File: string.hpp
#ifndef STRING_HPP_
#define STRING_HPP_

#include <contract.hpp>
#include <cstring>

// Adapted from an example presented in [Stroustrup1997] to illustrate
// importance of class invariants. Simple preconditions were added where it
// made sense. This should be compiled with postconditions checking turned off
// (define the `CONTRACT_CONFIG_NO_POSTCONDITIONS` macro) because
// postconditions are deliberately not used.
// See [Stroustrup1997] for a discussion on the importance of class invariants,
// and on pros and cons of using pre and post conditions.
CONTRACT_CLASS(
    class (string)
) {
    CONTRACT_CLASS_INVARIANT(
        // It would be better to assert conditions separately so to generate
        // more informative error in case they fail.
        chars_ ? true : throw invariant_error(),
        size_ >= 0 ? true : throw invariant_error(),
        too_large >= size_ ? true : throw invariant_error(),
        chars_[size_] == '\0' ? true : throw invariant_error()
    )

    // Broken contracts throw user defined exceptions.
    public: class range_error {};
    public: class invariant_error {};
    public: class null_error {};
    public: class too_large_error {};

    public: enum { too_large = 16000 }; // Length limit.

    CONTRACT_CONSTRUCTOR(
        public (string) ( (const char*) chars )
            precondition(
                chars ? true : throw null_error(),
                strlen(chars) <= too_large ? true : throw too_large_error()
            )
    ) ; // Deferred body definition.

    CONTRACT_CONSTRUCTOR(
        public (string) ( (const string&) other )
    ) ;

    CONTRACT_DESTRUCTOR(
        public (~string) ( void )
    ) ;

    CONTRACT_FUNCTION(
        public (char&) operator([])(at) ( int index )
```

```
            precondition(
                index >= 0 ? true : throw range_error(),
                size_ > index ? true : throw range_error()
            )
    ) ;

    CONTRACT_FUNCTION(
        public int (size) ( void ) const
    ) ;

    CONTRACT_FUNCTION( // Not public so it does not check class invariants.
        private void (init) ( (const char*) q )
    ) ;

    private: int size_;
    private: char* chars_;
};

#endif // #include guard
```

```
// File: string.cpp
#include "string.hpp"

CONTRACT_CONSTRUCTOR_BODY(string, string) ( const char* chars )
    { init(chars); }

CONTRACT_CONSTRUCTOR_BODY(string, string) ( const string& other )
    { init(other.chars_); }

CONTRACT_DESTRUCTOR_BODY(string, ~string) ( void )
    { delete[] chars_; }

char& string::CONTRACT_MEMBER_BODY(operator([])(at)) ( int index )
    { return chars_[index]; }

int string::CONTRACT_MEMBER_BODY(size) ( void ) const { return size_; }

void string::CONTRACT_MEMBER_BODY(init) ( const char* chars ) {
    size_ = strlen(chars);
    chars_ = new char[size_ + 1];
    for(int i = 0; i < size_; ++i) chars_[i] = chars[i];
    chars_[size_] = '\0';
}
```

```
// File: string_main.cpp
#include "string.hpp"
#include <boost/detail/lightweight_test.hpp>

// Handler that re-throws contract broken exceptions instead of terminating.
void throwing_handler ( contract::from const& context )
{
    if(context == contract::FROM_DESTRUCTOR) {
        // Destructor cannot throw for STL exception safety (ignore error).
        std::clog << "Ignored destructor contract failure" << std::endl;
    } else {
        // Failure handlers always called with active an exception.
        throw; // Re-throw active exception thrown by precondition.
    }
}

int main ( void )
{
    // Setup all contract failure handlers to throw (instead of terminate).
    contract::set_precondition_broken(&throwing_handler);
    contract::set_postcondition_broken(&throwing_handler);
    contract::set_class_invariant_broken(&throwing_handler);
    contract::set_block_invariant_broken(&throwing_handler);
    contract::set_loop_variant_broken(&throwing_handler);

    string s("abc");
    BOOST_TEST(s[0] == 'a');

#ifndef CONTRACT_CONFIG_NO_PRECONDITIONS
    bool pass = false;
    try { s[3]; } // Out of range.
    catch(string::range_error) { pass = true; }
    BOOST_TEST(pass);
#endif

    return boost::report_errors();
}
```

[Cline90] Vector: Comparison with A++ proposed syntax

This Library (C++03)

```
// File: vector.hpp
#ifndef VECTOR_HPP_
#define VECTOR_HPP_

#include <contract.hpp>

CONTRACT_CLASS(
    template( typename T )
    class (vector)
) {
    // NOTE: Incomplete set of assertions addressing only `size`.

    CONTRACT_CLASS_INVARIANT_TPL( size() >= 0 )

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( int count, default 10 )
            precondition( count >= 0 )
            postcondition( size() == count )
            initialize( data_(new T[count]), size_(count) )
    ) {
        for(int i = 0; i < size_; ++i) data_[i] = T();
    }

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~vector) ( void )
    ) {
        delete[] data_;
    }

    CONTRACT_FUNCTION_TPL(
        public int (size) ( void ) const
            // postcondition: Result non-negative checked by class invariant
    ) {
        return size_;
    }

    CONTRACT_FUNCTION_TPL(
        public void (resize) ( int count )
            precondition( count >= 0 )
            postcondition( size() == count )
    ) {
        T* slice = new T[count];
        for(int i = 0; i < count && i < size_; ++i) slice[i] = data_[i];
        delete[] data_;
        data_ = slice;
        size_ = count;
    }

    CONTRACT_FUNCTION_TPL(
        public (T&) operator([])(at) ( int index )
            precondition(
                index >= 0,
                index < size()
            )
    ) {
        return data_[index];
    }

    private: T* data_;
    private: int size_;
```

A++ Proposal (not part of C++)

```
// File: vector_app.hpp
// Extra spaces, newlines, etc used to align text with this library code.




template< typename T>
class vector
{
    legal: size() >= 0; // Class invariants (legal).

    public: explicit vector ( int count = 10 )
        : data_(new T[count]), size_(count)
    {
        for(int i = 0; i < size_; ++i) data_[i] = T();
    }

    public: virtual ~vector ( void )
    {
        delete[] data_;
    }

    public: int size ( void ) const
    {
        return size_;
    }

    public: void resize ( int count )
    {
        T* slice = new T[count];
        for(int i = 0; i < count && i < size_; ++i) slice[i] = data_[i];
        delete[] data_;
        data_ = slice;
        size_ = count;
    }

    public: T& operator[] ( int index )
    {
        return data_[index];
    }

    // Preconditions (require) and postconditions (promise) for each function.
    axioms: [ int count; require count >= 0; promise size() == count ]
            vector(count);
    axioms: [ int count; require count >= 0; promise size() == count ]
            resize(count);
    axioms: [ int index; require index >= 0 && index < size() ]
            (*this)[x];

    private: T* data_;
    private: int size_;
};
```

This Library (C++03)	A++ Proposal (not part of C++)

`};`
 	
`#endif // #include guard`	

`// File: vector_main.cpp`
`#include "vector.hpp"`
`#include <boost/detail/lightweight_test.hpp>`
 	
`int main ( void )`
`{`
`  vector<int> v(3); // Also set all elements to 0.0.`
`  BOOST_TEST(v.size() == 3);`
`  v[0] = 123;`
`  v.resize(2);`
`  BOOST_TEST(v[0] == 123);`
`  BOOST_TEST(v.size() == 2);`
`  return boost::report_errors();`
`}`	

[Cline90] Stack: Function-Try blocks and exception specifications

```
// File: stack.cpp
#include <contract.hpp>
#include <boost/config.hpp>
#include <boost/utility/identity_type.hpp>
#include <boost/detail/lightweight_test.hpp>
#include <exception>
#include <new>
#include <utility>
#include <iostream>

CONTRACT_CLASS(
    template( typename T )
    class (stack)
) {
    // NOTE: Incomplete set of assertions addressing only `empty` and `full`.

    CONTRACT_CLASS_INVARIANT( void )

    public: struct out_of_memory {};
    public: struct error {};

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (stack) ( int capacity )
            precondition( capacity >= 0 )
            postcondition(
                empty(),
                full() == (capacity == 0)
            )
            // Function-Try blocks are programmed within the macros only for
            // constructors with member initializers otherwise they are
            // programmed with the body definition and outside the macros.
            try initialize( // Try-block for constructor initializers and body.
                data_(new T[capacity]),
                capacity_(capacity),
                size_(0)
            ) catch(std::bad_alloc& e) (
                std::cerr << "out of memory for " << capacity << "-stack: " <<
                        e.what() << std::endl;
                throw out_of_memory();
            // Unfortunately, cannot wrap exception type commas with extra
            // parenthesis (because symbol `...` used to catch-all) but
            // `BOOST_IDENTITY_TYPE` can be used.
            ) catch(BOOST_IDENTITY_TYPE((std::pair<int, char const*>&)) e) (
                std::cerr << "error number " << e.first << " for " <<
                        capacity << "-stack: " << e.second << std::endl;
                throw error();
            ) catch(...) (
                std::cerr << "unknown error for " << capacity << "-stack" <<
                        std::endl;
                throw; // Re-throw exception.
            )
    ) {
        for(int i = 0; i < capacity_; ++i) data_[i] = T();
    }

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~stack) ( void )
    )
#if !defined(BOOST_MSVC) // MSVC supports only constructor-try blocks.
    try { // Function-try block (outside the macro).
```

```
        delete[] data_;
    } catch(...) {
        std::cerr << "error for stack destruction, terminating" << std::endl;
        std::terminate(); // Destructor should never throw.
    }
#else // MSVC
    {
        delete[] data_;
    }
#endif // MSVC

    CONTRACT_FUNCTION_TPL(
        public bool (empty) ( void ) const
    ) {
        return size_ == 0;
    }

    CONTRACT_FUNCTION_TPL(
        public bool (full) ( void ) const
    ) {
        return size_ == capacity_;
    }

    CONTRACT_FUNCTION_TPL(
        public void (push) ( (T) value )
#if !defined(BOOST_MSVC) // MSVC only supports throw( void ) exception spec.
            throw( std::exception, error ) // Ex spec.
#endif // MSVC
            precondition( not full() )
            postcondition( not empty() )
    )
#if !defined(BOOST_MSVC) // MSVC supports only constructor-try blocks.
    try
#endif // MSVC
    { // Function-Try block (outside the macro).
        data_[size_++] = value;
    }
#if !defined(BOOST_MSVC) // MSVC supports only constructor-try blocks.
    catch(std::exception& e) {
        std::cerr << "error for " << capacity_ << "-stack: " << e.what() <<
                std::endl;
        throw; // Re-throw STL exception.
    } catch(...) {
        std::cerr << "unknown error for " << capacity_ << "-stack" <<
                std::endl;
        throw error();
    }
#endif // MSVC

    CONTRACT_FUNCTION_TPL(
        public (T) (pop) ( void )
            precondition( not empty() )
            postcondition( not full() )
    ) {
        return data_[--size_];
    }

    private: T* data_;
    private: int capacity_;
    private: int size_;
};

int main ( void )
```

```
{
    stack<int> s(3);
    s.push(123);
    BOOST_TEST(s.pop() == 123);
    return boost::report_errors();
}
```

[Cline90] Vector-Stack: Subcontracting from Abstract Data Type (ADT)

```
// File: vstack.cpp
#include "vector.hpp"
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>

CONTRACT_CLASS( // Stack Abstract Data Type (ADT).
    template( typename T )
    class (stack_adt)
) {
    // NOTE: Incomplete set of assertions addressing only empty/full issues.

    CONTRACT_CLASS_INVARIANT_TPL( void ) // no invariants

    CONTRACT_CONSTRUCTOR_TPL(
        public (stack_adt) ( void )
            // postcondition:
            //  empty (cannot be checked because empty's postcondition uses
            //  length which is pure virtual during construction)
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public (~stack_adt) ( void )
    ) {}

    CONTRACT_FUNCTION_TPL(
        public bool (full) ( void ) const
            postcondition(
                auto result = return,
                result == (length() == capacity())
            )
    ) {
        return length() == capacity();
    }

    CONTRACT_FUNCTION_TPL(
        public bool (empty) ( void ) const
            postcondition( auto result = return, result == (length() == 0) )
    ) {
        return length() == 0;
    }

    CONTRACT_FUNCTION_TPL(
        public virtual int (length) ( void ) const
            postcondition( auto result = return, result >= 0 )
    ) = 0;

    CONTRACT_FUNCTION_TPL(
        public virtual int (capacity) ( void ) const
            postcondition( auto result = return, result >= 0 )
    ) = 0;

    CONTRACT_FUNCTION_TPL(
```

```
        public virtual void (push) ( (T) value )
            precondition( not full() )
            postcondition( not empty() )
    ) = 0; // Contract for pure virtual function.

    CONTRACT_FUNCTION_TPL(
        public virtual (T) (pop) ( void )
            precondition( not empty() )
            postcondition( not full() )
    ) = 0;

    CONTRACT_FUNCTION_TPL(
        public virtual void (clear) ( void )
            postcondition( empty() )
    ) = 0;
};

CONTRACT_CLASS( // Vector-based stack.
    template( typename T )
    class (vstack) extends( public stack_adt<T> )
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        length() >= 0,
        length() < capacity()
    )

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vstack) ( int count, default 10 )
            precondition( count >= 0 )
            postcondition( length() == 0, capacity() == count )
            initialize( vect_(count), length_(0) ) // OK, after preconditions.
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~vstack) ( void )
    ) {}

    // NOTE: All following inherit contracts from `stack_adt`.

    CONTRACT_FUNCTION_TPL(
        public int (length) ( void ) const override final
    ) {
        return length_;
    }

    CONTRACT_FUNCTION_TPL(
        public int (capacity) ( void ) const
    ) {
        return vect_.size();
    }

    CONTRACT_FUNCTION_TPL(
        public void (push) ( (T) value ) override final
    ) {
        vect_[length_++] = value;
    }

    CONTRACT_FUNCTION_TPL(
        public (T) (pop) ( void ) override final
    ) {
        return vect_[--length_];
    }
```

```
    CONTRACT_FUNCTION_TPL(
        public void (clear) ( void ) override final
    ) {
        length_ = 0;
    }

    private: vector<T> vect_;
    private: int length_;
};

int main ( void )
{
    vstack<int> s(3);
    BOOST_TEST(s.capacity() == 3);
    s.push(123);
    BOOST_TEST(s.length() == 1);
    BOOST_TEST(s.pop() == 123);
    return boost::report_errors();
}
```

[Cline90] Calendar: A very simple calendar

```
// File: calendar.cpp
#include <contract.hpp>
#include <boost/detail/lightweight_test.hpp>

CONTRACT_CLASS(
    class (calendar)
) {
    CONTRACT_CLASS_INVARIANT(
        month() >= 1,
        month() <= 12,
        date() >= 1,
        date() <= days_in(month())
    )

    CONTRACT_CONSTRUCTOR(
        public (calendar) ( void )
            postcondition( month() == 1, date() == 31 )
            initialize( month_(1), date_(31) )
    ) {}

    CONTRACT_DESTRUCTOR(
        public virtual (~calendar) ( void )
    ) {}

    CONTRACT_FUNCTION(
        public int (month) ( void ) const
    ) {
        return month_;
    }

    CONTRACT_FUNCTION(
        public int (date) ( void ) const
    ) {
        return date_;
    }

    CONTRACT_FUNCTION(
        public void (reset) ( int new_month )
            precondition( new_month >= 1, new_month <= 12 )
            postcondition( month() == new_month )
    ) {
        month_ = new_month;
    }

    CONTRACT_FUNCTION(
        private static int (days_in) ( int month )
            precondition( month >= 1, month <= 12 )
            postcondition( auto result = return, result >= 1, result <= 31 )
    ) {
        return 31; // For simplicity, assume all months have 31 days.
    }

    private: int month_, date_;
};

int main ( void )
{
    calendar c;
```

```
    BOOST_TEST(c.date() == 31);
    BOOST_TEST(c.month() == 1);
    c.reset(8); // Set month to August.
    BOOST_TEST(c.month() == 8);
    return boost::report_errors();
}
```

Grammar

> *"Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer."*
>
> *--Stroustrup (see [Stroustrup97] page 160)*

This library uses macros to overcome a limitation of C++, namely the fact that the core language does not support preconditions, postconditions, class invariants, old values, and subcontracting. This section lists the complete grammar of the syntax used by this library macros.

Warning

In general, an error in programming this library syntax will generate cryptic compiler errors (often exposing internal code from this library and from Boost.Preprocessor). [70]

There are intrinsic limitations on the amount of error checking that can be implemented by this library because it uses the preprocessor to parse its syntax (e.g., there is no way the preprocessor can gracefully detect and report unbalanced round parenthesis `( ... /* missing closing parenthesis here */` or an invalid concatenation symbol `BOOST_PP_CAT(xyz, ::std::vector)`). In addition, for a given macro all compiler error messages report the same line number (because macros always expand on a single line) so line numbers are not very useful in identifying syntactic errors.

While the preprocessor imposes limits on the error checking that can be implemented, the current version of this library does not focus on providing the best possible syntax error messages (this will be the focus of future releases, see also Ticket 44).

The best way to resolve syntactic errors is for programmers to inspect the code "by eye" instead of trying to make sense of the compiler errors. This section is very useful for programmers to make sure that they are using the syntax correctly.

Preprocessor DSEL

The syntax used by this library macros effectively defies a new language within C++. More precisely, this library macros define a Domain-Specific Embedded Language (DSEL) that replaces the usual C++ syntax for class and function declarations. This is the *Language of Contract++* (or *LC++* for short).

In contrast with other DSEL hosted by C++ which are parsed using template meta-programming (e.g., Boost.Phoenix), this library DSEL is parsed "one meta-programming level higher" using preprocessor meta-programming. Using both processor meta-programming and template meta-programming allows this library to implement rules like this:

```
if a member function is not public then it does not check the class invariants
```

This rule cannot be implemented using only template meta-programming because it is not possible to check if a function is public using template meta-programming introspection techniques. For example, it is not possible to implement a boolean meta-function like the following: [71]

```
template< ... >
struct is_public { ... }; // Unfortunately, this cannot be implemented.
```

Instead a macro can be programmed to parse the following function declarations and expand to `1` if and only if the function is public:

```
#define IS_PUBLIC(function_declaration) ... // This can be implemented.

IS_PUBLIC( public  void (f) ( int x ) )     // Expand to 1.
IS_PUBLIC( private void (g) ( int y ) )     // Expand to 0.
IS_PUBLIC(         void (h) ( int z ) )     // Expand to 0.
```

There are more examples of class and function declaration traits (`virtual`, etc) that need to be known to correctly implement Contract Programming but that cannot be inspected using template meta-programming. This library macros can instead parse the specified class and function declarations extracting *any* declaration trait (if a constructor is `explicit`, if a member function is `virtual`, if a base class is `protected`, if a parameter has a default value, etc).

It should be noted that while the syntax of the DSEL defined by this library macros is rather extensive and complex, macros always define a DSEL that is intrinsically different from the core C++ language. For example, consider the following function-like macro:

[70] Usually, the complexity of C++ definitions is greater than the complexity of C++ declaration and the fact that this library macros only affect declarations would help by preserving the usefulness of the compiler error messages for the definition code. However, this library aims to make declarations more complex by adding program specifications to them (preconditions, postconditions, etc). Therefore, it can no longer be argued that declarations are significantly simpler than definitions and it would be rather helpful to have sensible compiler error messages at least for the declarations with contracts.

[71] Using C++11, it *might* be possible to implement the `is_public` template because SFINAE was extended to support access level (but to the authors' knowledge such a meta-function has not been implemented yet so the authors cannot be sure that `is_public` can be properly implemented even using C++11 SFINAE). Even if that were possible in C++11, this library still needs declaration traits other than `public` in oder to properly implement Contract Programming (e.g., if a function is `virtual` or not in order to implement subcontracting, and there are examples like that). Therefore, the arguments made here for the need to use a preprocessor DSEL in oder to properly implement Contract Programming in C++ hold even if `is_public` could be implemented in C++11.

```
#define F(x, y) (int(x) - int(y))

template< typename X, typename Y >
int f ( X const& x, Y const& y ) { return (int(x) - int(y)); }

int main ( void )
{
    std::cout << F( , 2) << std::endl; // OK, no syntax error, it prints `-2`.
    std::cout << f( , 2) << std::endl; // Compiler generates a syntax error.
    return 0;
}
```

Note how it is valid to invoke the macro with an empty parameter `F( , 2)` while it is syntactically invalid to invoke a function with an empty parameter `f( , 2)`. This very simple macro already shows fundamental differences between the syntax of macros and the syntax of the core language.

Differences with C++ Syntax

The following is a summary of all the differences between the syntax of this library macros and the usual C++ class and function declaration syntax.

Rule	Syntactic Element	Syntax Differences
1	Template Declarations	Use round parenthesis `template( `*`template-parameters`*` )` instead of angular parenthesis `template< `*`template-parameters`*` >` to declare templates (note that template instantiations are not template declarations and they use angular parenthesis as usual).
2	Template Specializations	Use round parenthesis `( `*`template-specializations`*` )` instead of angular parenthesis `< `*`template-specializations`*` >` after a class template name to specify template specialization arguments.
3	Class, Function, and Operator Names	Wrap class and function declaration names within round parenthesis `(`*`class-name`*`)` and `(`*`function-name`*`)`. Use `operator(`*`symbol`*`)(`*`arbitrary-name`*`)` for operators (allowed but not required for `operator new`, `operator delete`, and implicit type conversion operators for fundamental types with no symbol). Always use `operator comma` for comma operator. Memory member operators must always be explicitly declared `static`.
4	Base Classes	Use `extends( `*`base-classes`*` )` instead of the column symbol `: `*`base-classes`* to inherit from base classes.
5	Default Parameters	Use `default `*`parameter-default`* instead of the assignment symbol `= `*`parameter-default`* to specify template and function parameter default values.
6	Member Access Levels	Always specify the access level `public`, `protected`, or `private` (note no trailing comma `:`) for every constructor, destructor, member function, and nested class declaration.
7	Result and Parameter Types	Wrap function result and parameter types within round parenthesis `(`*`type`*`)`. The wrapping parenthesis are allowed but not required for fundamental types containing only alphanumeric tokens (e.g., both `(const unsigned int)` and `const unsigned int` are allowed, only `(int&)` and not `int&` is allowed because of the non-alphanumeric symbol `&`, only `(mytype)` is allowed because `mytype` is not a fundamental type).
8	Member Initializers	Use `initialize( `*`member-initializers`*` )` instead of the column symbol `: `*`member-initializers`* to specify constructor member initializers.
9	Empty Lists	Specify empty lists (parameters, exception specifications, template specializations, etc) using `( void )` instead of `( )` or `< >`.
10	Commas and Leading Symbols	Syntactic elements containing commas and leading non-alphanumeric symbols must be wrapped within extra round parenthesis `(...)`. (Note that `'a'`, `"abc"`, `1.23`, etc are not alphanumeric so they need to be wrapped as `('a')`, `("abc")`, `(1.23)`, etc when specified as default parameters or similar.)

Important

In general, every token which is not a known keyword (`int` is a known keyword but the function name is not) or that contains a non-alphanumeric symbol (e.g., `int&`) must be wrapped within round parenthesis `(...)` unless it is the very last token of a syntactic element (e.g., the function parameter name).

Sometimes the C++ syntax allows to equivalently specify elements in different orders (e.g., `const volatile` and `volatile const` are both allowed for member functions and they have the same meaning). The syntax of this library requires instead to specify the elements exactly in the order listed by this grammar (e.g., only `const volatile` is allowed for member functions). [72]

Macro Interface

This library uses the following macros.

[72] **Rationale.** This library macros could be implemented to allow to specify syntactic elements in different orders but that would complicate the macro implementation and this grammar with no additional feature for the user.

```
CONTRACT_CLASS(class-declaration)
CONTRACT_CLASS_TPL(class-declaration)

CONTRACT_CLASS_INVARIANT(class-invariants)
CONTRACT_CLASS_INVARIANT_TPL(class-invariants)

CONTRACT_FUNCTION(function-declaration)
CONTRACT_FUNCTION_TPL(function-declaration)
CONTRACT_CONSTRUCTOR(function-declaration)
CONTRACT_CONSTRUCTOR_TPL(function-declaration)
CONTRACT_DESTRUCTOR(function-declaration)
CONTRACT_DESTRUCTOR_TPL(function-declaration)

CONTRACT_FREE_BODY(function-name)
CONTRACT_MEMBER_BODY(function-name)
CONTRACT_CONSTRUCTOR_BODY(class-type, class-name)
CONTRACT_DESTRUCTOR_BODY(class-type, ~class-name)

CONTRACT_BLOCK_INVARIANT(assertions)
CONTRACT_BLOCK_INVARIANT_TPL(assertions)

CONTRACT_LOOP(loop-declaration)
CONTRACT_LOOP_VARIANT(loop-variant)
CONTRACT_LOOP_VARIANT_TPL(loop-variant)

CONTRACT_CONSTRUCTOR_ARG(parameter-name)
CONTRACT_PARAMETER_TYPEOF(parameter-name)
CONTRACT_PARAMETER(named-parameter-declaration)
CONTRACT_TEMPLATE_PARAMETER(named-parameter-declaration)
CONTRACT_PARAMETER_BODY(function-name)
```

The macros with the trailing `_TPL` must be used when the enclosing scope is type-dependent (e.g., within templates).

Lexical Conventions

The following conventions are used to express this grammar.

Lexical Expression	**Meaning**
`[tokens]`	Either `tokens` or nothing (optional tokens).
`{expression}`	The result of the enclosed expression `expression` (evaluation order).
`tokens1 | tokens2`	Either `tokens1` or `tokens2` (“or” operation).
`tokens*`	`tokens` repeated zero or more times (repetition starting from zero).
`tokens+`	`tokens` repeated one or more times (repetition starting from one).
`token, ...`	A comma separated list of tokens that could also have one single element (i.e., `token` or `token1, token2` or `token1, token2, token3`). See the No Variadic Macros section for compilers that do not support variadic macros.
`tokens`	Terminal symbols (in bold font).
`tokens`	Non-terminal symbols (in italic font).

Class Declarations

```
class-declaration:
        [[export] template( template-parameters ) [requires( concepts )]]
        [friend] {class | struct} (class-name)[( template-specializations )] [final]
        [extends( base-classes )]
```

Base Classes

```
base-classes:
        [public | protected | private] [virtual] class-type, ...
```

Note that when specified, `virtual` must appear after the inheritance access level for a base class.

Template Specializations

```
template-specializations:
        template-specialization, ...
```

Template Parameters

```
template-parameters:
        void |
        positional-template-parameters |
        named-template-parameters

positional-template-parameters:
        positional-template-parameter, ...
positional-template-parameter:
        positional-type-template-parameter |
        positional-value-template-parameter |
        positional-template-template-parameter
positional-type-template-parameter:
        {class | typename} parameter-name
        [, default parameter-default]
positional-value-template-parameter:
        wrapped-type parameter-name
        [, default parameter-default]
positional-template-template-parameter:
        template( template-parameter, ... ) class parameter-name
        [, default parameter-default]

named-template-parameters:
        [using namespace named-parameter-identifier-namespace,]
        named-type-template-parameter, ...
named-type-template-parameter:
        [deduce] in {class | typename}
        [requires(unary-boolean-metafunction)] parameter-name
        [, default parameter-default]
```

Note that if `typename` appears within *`wrapped-type`* for a value template parameter then it will be wrapped within parenthesis `(typename ...)` so it is syntactically distinguishable from the `typename` leading a type template parameter.

Unfortunately, named template parameters only support type template parameters. Named value template parameters and named template template parameters are not supported because they are not supported by Boost.Parameter but if they were ever supported, they could follow this syntax:

```
named-template-parameters:
        named-template-parameter, ...
named-template-parameter:
        named-type-template-parameter |
        named-value-template-parameter |
        named-template-template-parameter
named-value-template-parameter: // Not supported.
        [deduce] in
        {wrapped-type | auto | requires(unary-boolean-metafunction)} parameter-name
        [, default parameter-default]
named-template-template-parameter: // Not supported.
        [deduce] in template( positional-template-parameter, ... ) class
        [requires(unary-boolean-metafunction)] parameter-name
        [, default parameter-default]

template( // For example...
    // Named type template parameter (supported).
      in typename requires(is_const<_>) T, default int
    // Named value template parameter (not supported).
    , in requires(is_converible<_, T>) val, default 0
    // Named template template parameter (not supported).
    , in template(
          typename A
        , template(
              typename X
            , typename Y, default int
          ) class B, default b
      ) class requires(pred1< _<T> >) Tpl, default tpl
)
```

Concepts

```
concepts:
        boost-concept, ...
```

Types

```
wrapped-type:
        fundamental-type | (fundamental-type) | (type)

fundamental-type:
        type-qualifier* type-keyword type-qualifier*
type-qualifier:
        const | volatile | long | short | signed | unsigned
type-keyword:
        void | bool | char | double | float | int | wchar_t | size_t | ptrdiff_t
```

This is the syntax used to specify the function result type, the function parameter types, the types of value template parameters, etc. As indicated by the syntax, extra parenthesis around the specified type are always allowed but they are required only for user-defined types (`mytype`) and types containing non-alphanumeric symbols (`int&`) while the parenthesis are optional for fundamental types containing no symbol `unsigned long int const` (or equivalently (`unsigned long int const`)).

Function Declarations

```
function-declaration:
        [public | protected | private]
        [[export] [template( template-parameters ) [requires( concepts )]]
        [explicit] [inline] [extern] [static] [virtual] [friend]
        [result-type] function-name ( function-parameters )
        [requires( concepts )]
        [const] [volatile] [override | new] [final]
        [throw( exception-specifications )]
        [precondition( assertions )]
        [postcondition( result-oldof-assertions )]
        [member-initializers]
```

This is the syntax used to declare all functions: Free functions, member functions, constructors, destructors, and operators. The usual constraints of C++ function declarations apply: It is not possible to declare a static virtual member function, only constructors can use the class name as the function name, constructors and destructors have no result type, etc. The `static` specifier can only be used for member functions (because it was deprecated for free functions from C to C++03). The `volatile` specifier must always appear after `const` when they are both specified.

Result Type

```
result-type:
        wrapped-type
```

Note that fundamental types containing no symbol can be specified without extra parenthesis: `void`, `bool`, `int`, `unsigned long const`, etc.

Function and Operator Names

```
function-name:
        (function-identifier) | (class-name) | (~class-name) | operator-name

operator-name:
        operator(operator-symbol)(operator-identifier) | operator fundamental-type |
        operator new | operator delete | operator comma
```

Names for free functions, member functions, constructors, and destructors are specified as usual but wrapped within parenthesis.

Operator names are specified wrapping within parenthesis the usual operator symbol followed by an arbitrary but alphanumeric identifier:

```
operator([])(at)
operator(+)(plus)
operator(())(call)
operator(new[])(new_array)
operator(delete[])(delete_array)
```

Implicit type conversion operators use the same syntax:

```
operator(int*)(int_ptr)
operator(mytype const&)(mytype_const_ref)
operator(std::map<char, int>)(std_map)
```

However, if the type is a fundamental type containing no symbol, the identifier is optional:

```
operator(const int)(const_int)  // Allowed but...
operator const int              // ... more readable.
```

Similarly, the parenthesis and identifier are optional for the `new` and `delete` operators:

```
operator(new)(new_ptr)          // Allowed but...
operator new                    // ... more readable.

operator(delete)(delete_ptr)    // Allowed but...
operator delete                 // ... more readable.
```

Finally, the comma symbol `,` cannot be used to specify the comma operator [73] so the specifier `comma` must always be used to name the comma operator:

```
operator comma      // OK.
operator(,)(comma)  // Error.
```

C++ automatically promotes the memory member operators `operator new`, `operator delete`, `operator new[]`, and `operator delete[]` to be static members so the `static` specifier is allowed but optional for these member operators. This library cannot automatically perform such a promotion so the `static` specifier is always required by this library for the memory member operators.

Exception Specifications

```
exception-specifications:
        void | exception-type, ...
```

Note that the syntax `throw( void )` is used instead of `throw( )` for no-throw specifications.

Exception specifications apply only to exceptions thrown by the function body and not to exceptions thrown by the contracts themselves (if any) and by the contract checking code generated by this library macros.

Member Initializers

```
member-initializers:
        [try] initialize( member-initializer, ... )
        {catch(catch-declaration) ( catch-instructions )}*
```

As indicated by this syntax, it is possible to specify function-try blocks for constructor member initializers:

```
struct vector_error { ... };

CONTRACT_CONSTRUCTOR( // Constructor with member initializers.
explicit (vector) ( int count )
    precondition( count >= 0 )
    postcondition( size() == count )

    try initialize( ptr_(new T[count]) )
    catch(std::bad_alloc& ex) (
        std::cerr << "not enough memory for " << count << " elements vector: " << ex.what() << std::endl;
        throw vector_error(1);
    ) catch(std::exception& ex) (
        std::cerr << "error for " << count << " elements vector: " << ex.what() << std::endl;
        throw vector_error(2);
    ) catch(...) (
        std::cerr << "unknown error for " << count << " elements vector" << std::endl;
        throw vector_error(3);
    )
) { // Above function-try block only applies to exceptions thrown by the body (and not by the contracts).
    ...
}
```

[73] **Rationale.** Within macros a comma `,` has the special meaning of separating the macro parameters so it cannot be used to indicate the comma operator otherwise `operator(,)(mycomma)` and `operator(std::map<char, int>)(stdmap)` could not be distinguished from one another by the preprocessor.

For functions other than constructors and for constructors without member initializers, function-try blocks are programmed outside the macros and around the body definition as usual. As specified by [N1962], function-try blocks apply only to exceptions thrown by the function body and not to exceptions thrown by the contracts themselves (if any) and by the contract checking code generated by the library macros.

Member initializers (and their function-try blocks) are part of the constructor definition and not of the constructor declaration. However, C++03 lack of delegating constructors requires member initializers (and their function-try blocks) to be specified within this library macros and constructors must be defined together with their declarations when they use member initializers.

Function Parameters

```
function-parameters:
        void |
        positional-function-parameters |
        named-function-parameters

positional-function-parameters:
        positional-function-parameter, ...
positional-function-parameter:
        [auto | register] wrapped-type parameter-name
        [, default parameter-default]

named-function-parameters:
        [using namespace named-parameter-identifier-namespace,]
        named-function-parameter, ...
named-function-parameter:
        [deduce] {in | out | in out}
        {wrapped-type | auto | requires(unary-boolean-metafunction)} parameter-name
        [, default parameter-default]
```

Note that the positional parameter storage classifier `auto` is supported by this library because it is part of C++03 (but the `auto` keyword changed meaning in C++11 so use it with the usual care when writing code portable to C++11).

Result and Old Values

```
result-oldof-assertions:
        [auto variable-name = return,]
        [oldof-declaration, ...]
        assertion, ...

oldof-declaration:
        {auto | wrapped-type} variable-name = CONTRACT_OLDOF oldof-expression
```

If present, result and old-of declarations should appear at the very beginning of the postconditions because they will always be visible to all assertions plus these declarations cannot be nested (within select-assertions, etc).

The result declaration type is always `auto` because the function result type is know and already specified by the function declaration (so no result type deduction is ever needed). The old-of declaration type is automatically deduced (using Boost.Typeof) when its type is specified `auto` instead of *wrapped-type*. The macro `CONTRACT_OLDOF` does not require but allows parenthesis around the value expression **oldof-expression** (this is as specified for the `oldof` operator in [N1962] and similar to the `sizeof` operator which requires parenthesis when applied to a type expression `sizeof(size_type)` but not when applied to a value expression `sizeof size()`). For example, all the followings are valid:

```
auto old_size = CONTRACT_OLDOF size()
auto old_size = CONTRACT_OLDOF(size())          // Extra parenthesis around expression (optional).
(size_type) old_size = CONTRACT_OLDOF size()    // Do not use Boost.Typeof.
(size_type) old_size = CONTRACT_OLDOF(size())   // No Boost.Typeof and extra parenthesis.
```

If an old-of copy is performed on a type that is not `ConstantCopyConstructible`, the old-of declaration itself will not fail compilation but it will produce an old variable that will cause a compiler-error as soon as it is used in an assertion (unless the assertion specifies a requirement using a properly specialized `contract::has_oldof` trait).

Class Invariants

```
class-invariants:
        void | class-invariant, ...

class-invariant:
        assertion |
        static class( void | assertion, ... ) |
        volatile class( void | assertion, ... )
```

Volatile class invariants are assumed to have the same assertions as non-volatile class invariants unless they are explicitly specified. Static class invariants are assumed to assert nothing unless they are explicitly specified.

Assertions

```
assertions:
        assertion, ...
assertion:
        using using-directive |
        namespace namespace-alias |
        typedef typedef-type new-type-name |
        assertion-condition

assertion-condition:
        assertion-expression |
        select-assertion
assertion-expression:
        boolean-condition |
        static_assert(constant-boolean-expression, constant-string-literal)
        [, requires constant-boolean-expression]
select-assertion:
        if(boolean-condition) ( assertion-condition, ... )
        [else ( assertion-condition, ... )]

boolean-condition:
        boolean-expression |
        const( inscope-variables ) boolean-expression-using-inscope-variables
inscope-variables:
        [wrapped-type] inscope-variable, ...
```

Some basic name manipulations are allowed at the local scope where the assertions are being declared in case they are ever needed to simplify the assertion expressions. Specifically, using-directives, namespace-aliases, and type-definitions are allowed (these will always affect all assertions within the given preconditions, postconditions, etc so it is recommended to always use these statement at the very beginning before the actual assertion conditions). Note that these name manipulations have no effect on the program run-time state and therefore they do not compromise the contract constant-correctness requirement.

As indicated by the grammar above, it is not possible to specify assertion requirements (using `requires`) for the entire select assertion if-then-else expression. Eventual assertion requirements must be specified for the single assertions within the select assertion if-then-else statement and they will never disable compilation and checking of the select assertion if-condition. Programmers can use the ternary operator `?:` instead of a select assertion if they need to specify a guarded assertion with a condition that is also disabled by the assertion requirements:

```
boolean-guard ? boolean-expression : true, requires constant-boolean-expression
```

Constant expressions `const( ... )` *`expression`* can be used to assert conditions and to check the select assertion if-condition so to fully enforce the assertion constant-correctness requirement. However, function arguments, result value, old-of values, and the object `this` are automatically made constant by this library so constant expressions only need to be used to assert conditions on global variables, static variables, etc. The type of the in-scope variable **`inscope-variable`** is optional and it is automatically deduced using Boost.Typeof when it is not specified.

Loop Variants

```
loop-variant:
        natural-expression |
        const( inscope-variables ) natural-expression-using-inscope-variables
```

A loop variant must specify a non-negative integral expression that monotonically decreases at each subsequent loop iteration (the library will automatically check these two conditions at each loop iteration and terminate the loop if they are not met by calling the loop variant broken handler). The loop variant can be specified using a constant-expression `const( ... )` *`expression`* so to fully enforce the contract constant-correctness requirement.

Named Parameter Declarations

```
named-parameter-declaration:
        [namespace(named-parameter-identifier-namespace)]
        [(named-argument-identifier)] parameter-name
```

Terminals

Terminal	Description	If terminal contains unwrapped commas or leading symbols
boolean-expression	A boolean expression: `x == 1`.	Wrap value within parenthesis: `(vey_sizeof<Key, T>::value)`.
boolean-expression-using-inscope-variables	A boolean expression that only uses in-scope variables captured as constants by a constant expression `const( ... )` *expression*.	Wrap value within parenthesis: `(key_sizeof<Key, T>::value + x)`.
boost-concept	A concept class defined using Boost.ConceptCheck: `boost::CopyConstructible`.	Wrap type within parenthesis: `(boost::Convertible<T, int>)`.
catch-declaration	The declaration of an exception for a `catch` statement: `std::runtime_error& error`.	Wrap type using Boost.Utility/IdentityType: `BOOST_IDENTITY_TYPE((map<char, int>::exception&)) error`.
catch-instructions	The instructions of a `catch` statement terminated by semicolons `;`: `std::cout << "error" << std::endl; exit(255);`.	Wrap types using Boost.Utility/IdentityType and values within parenthesis: `typedef BOOST_UTILITY_TYPE(std::map<char, int>) mtype; (mtype m, m['a'] = 1);`.
class-name	The class name: `myclass`. For class templates this must not include the template instantiation parameters: `vector`. (For non-template classes, the class type and name are the same.)	Never the case.
class-type	The class type, for class templates this must include the template instantiation parameters: `vector<T>`.	Wrap type within parenthesis: `(map<Key, T>)`.
constant-boolean-expression	A compile-time constant boolean expression: `sizeof(T) >= sizeof(int)`.	Wrap value within parenthesis: `(boost::is_convertible<T, int>::value)`.
constant-string-literal	A compile-time constant string literal: `"abc"`.	Do nothing: `"abc"`.
exception-type	A type: `std::exception`, `int`, `mytype`.	Wrap type within parenthesis: `(map<char, int>::exception)`.
function-identifier	A valid function name identifier (C++ requires it to be alphanumeric): `f`, `push_back`, `my-func`.	Never the case.
inscope-variable	A variable in-scope.	Never the case.
loop-declaration	A loop declaration: `for(int i = 0; i < 10; ++i)`, `while(i < 10)`.	Never the case.
member-initializer	A member initialization expression: `vector_(count)`.	Wrap object initializations within parenthesis: `(base_map<char, int>())`.
named-argument-identifier	The argument name to use at the calling site to pass named and deduced parameter values: `value_arg`, `NumberArg`.	Never the case.
named-parameter-identifier-namespace	The internal namespace for named and deduced parameter identifiers: `params`.	Never the case.
namespace-alias	The argument to pass to namespace aliasing: `mpl = boost::mpl`.	Never the case.
natural-expression	A natural (i.e., non-negative integral) expression: `2 - 1`.	Wrap value within parenthesis: `(key_sizeof<Key, T>::value)`.
natural-expression-using-inscope-variables	A natural (i.e., non-negative integral) expression that only uses in-scope variables captured as constant by a constant-expression: `x + 10`.	Wrap value within parenthesis: `(key_sizeof<Key, T>::value + x)`.
new-type-name	A new type name for `typedef` statement: `myint`.	Never the case.
oldof-expression	A expression of type `ConstantCopyConstructible` to pass to the `CONTRACT_OLDOF` macro: `value`, `size()`.	Wrap value within parenthesis: `(x, y)`.
operator-identifier	An arbitrary but alphanumeric identifier: `equal`, `less`, `call`.	Never the case.
operator-symbol	The usual operator symbols: `==`, `<=`, `()`.	Do nothing: `std::map<char, int>`.

Terminal	Description	If terminal contains unwrapped commas or leading symbols
`parameter-default`	A function or template parameter default value (either a value, a type, or a template depending on the kind of parameter): `123`.	Wrap value within parenthesis: `("abc")`, `('a')`, `(-123)`, `(1.23)`.
`parameter-name`	A function or template parameter name: `value`, `T`.	Never the case.
`template-parameter`	A usual C++ type template parameter, value template parameter, or template template parameter): `typename T`, `class U`, `int Number`, `T Value`, `template< typename X, class Y > class Template`.	Do nothing: `std::map<Key, T> Default`, `template< typename X, class Y > class Template`.
`template-specialization`	A template specialization argument (type, value, etc) that follow the class name in the declaration to specialize a template: `void (int, T)`.	Wrap types within parenthesis: `(std::map<char, int>)`.
`type`	A type: `int`, `int const&`, `mytype`.	Do nothing: `std::map<char, int>`.
`typedef-type`	A type: `int`, `mytype`.	Wrap type using Boost.Utility/IdentityType: `BOOST_IDENTITY_TYPE((std::map<char, int>))`.
`unary-boolean-metafunction`	A boolean meta-function that takes one parameter: `boost::is_class<boost::mpl::_>`.	Do nothing: `boost:is_convertible<boost::mpl::_, int>`.
`using-directive`	The argument to pass to a using directive: `namespace std`, `std::vector`.	Never the case.
`variable-name`	A valid name to use to declare a variable: `result`, `old_size`.	Never the case.

If terminals contain commas not already wrapped by round parenthesis or if they start with a non-alphanumeric symbol (including tokens like `'a'`, `"abc"`, `-123`, and `1.23`), [74] they need to be wrapped by extra round parenthesis `(...)` or by the Boost.Utility/IdentityType `BOOST_IDENTITY_TYPE((...))` macro. Value expressions can always be wrapped within extra around parenthesis in C++. Type expressions can always be wrapped using Boost.Utility/IdentityType but that will make the syntax less readable (and it prevents C++ from automatically deducing function template parameters) so this syntax allows to wrap type expressions within extra round parenthesis `(...)` for most terminals, including types, as indicated by the table above.

Alternative Assertion Syntax (Not Implemented)

The following alternative syntax could have been implemented to program the contract assertions:

[74] The C++ preprocessor cannot concatenate `1.23` because it contains the `.` symbol (even if that symbol is technically not the leading symbol). The precise preprocessor requirement is that the concatenated symbol must be a valid macro identifier and concatenating `1.23` with any token will never give a valid macro identifier because of the presence of the dot symbol `.` (e.g., `BOOST_PP_CAT(XYZ, 1.23)` gives `XYZ1.23` which is not a valid macro identifier).

This Library Syntax

```
CONTRACT_CLASS(
    template( typename T )
    class (vector)
) {
    ...

    CONTRACT_FUNCTION_TPL(
        public (iterator) (erase) ( (iterator) where )
            precondition(
                not empty(),
                where != end(),
                static_assert(sizeof(T) >= sizeof(int), "large enough")
            )
            postcondition(
                auto result = return,
                auto old_size = CONTRACT_OLDOF size(),
                size() == old_size - 1,
                if(const( this ) this->empty()) (
                    result == end()
                )
            )
    ) {
        return vector_.erase(where);
    }
};
```

Alternative Syntax (not implemented)

```
CONTRACT_CLASS(
    template( typename T )
    class (vector)
) {
    ...

    CONTRACT_FUNCTION_TPL(
        public (iterator) (erase) ( (iterator) where )
            precondition(
                assert(not empty())
                assert(where != end())
                static_assert(sizeof(T) >= sizeof(int), "large enough")
            )
            postcondition(
                decl(auto result = return)
                decl(auto old_size = CONTRACT_OLDOF size())
                assert(size() == old_size - 1)
                if(const(this, this->empty())) (
                    assert(result == end())
                )
            )
    ) {
        return vector_.erase(where);
    }
};
```

An advantage of this alternative syntax is that it does not require commas at the end of each assertion. However, when compared with this library syntax, the alternative syntax is overall more verbose, it uses more parenthesis, it deviates more from [N1962] and Eiffel (even if it is more similar to D), and it looks less readable at least because of `decl(...)` (in many ways this alternative syntax is equivalent to the already supported sequencing syntax from the No Variadic Macros section with the addition of “decoration identifiers” like `assert` which might make the code more readable but are not needed syntactically). Therefore, the authors opted for implementing the syntax on the left hand side.

No Variadic Macros

This section illustrates an alternative syntax, the *sequence syntax*, that can be used on compilers that do not support variadic macros. Most modern compilers support variadic macros (notably, these include GCC, MSVC, and all C++11 compilers).

Warning

The sequence syntax presented in this section has not been fully tested yet. Future revisions of the library are expected to test and support the sequence syntax more thoroughly (see also Ticket 58).

The sequence syntax uses many extra parenthesis and it is significantly less readable than the comma-separated syntax that we have seen so far. Therefore, it is strongly recommended to not use the sequence syntax unless it is absolutely necessary to program contracts that are portable to compilers that do not support variadic macros.

Sequence Syntax

In the rare case that programmers need to use this library on compliers without variadic macros, this library also allows to specify its macro parameters using a Boost.Preprocessor sequence in which tokens are separated using round parenthesis `()`:

```
(token1) (token2) ... // All compilers.
```

Instead of the comma-separated lists that we have seen so far which require variadic macros:

```
token1, token2, ... // Only compilers with variadic macros (preferred).
```

This library detects preprocessor support for variadic macros using the Boost.Config macro `BOOST_NO_VARIADIC_MACROS`. Boost.Config defines the `BOOST_NO_VARIADIC_MACROS` only on compilers that do not support variadic macros furthermore programmers can forcefully define this macro also on compilers that support variadic macros. When this macro is not defined, this library macros support both the camma-separated and sequence syntax, otherwise only the sequence syntax is supported.

Note

The same macros accept both syntaxes on compilers with variadic macros and only the sequence syntax on compilers without variadic macros.

For example, the syntax on the left hand side works on all compilers with and without variaidic macros (see also `class_template_vector_seq.cpp`, `pushable_seq.hpp`, `class_template_vector.cpp`, and `pushable.hpp`):

Sequence Syntax (all compilers)

```
CONTRACT_CLASS(
    template( (typename T) )
    class (pushable)
) {
    // Subcontracting: In `and` with derived class invariants.
    CONTRACT_CLASS_INVARIANT_TPL( (void) )

    public: typedef typename std::vector<T>::const_reference const_reference;

    CONTRACT_FUNCTION_TPL(
        public virtual void (push_back) ( ((T const&) value) ) new
            // Subcontracting: In `or` with overriding function preconditions.
            precondition( (false) ) // Force check to overriding preconditions.
            // Subcontracting: In `and` with overriding function postconditions.
            postcondition(
                (back() == value)(requires boost::has_equal_to<T>::value)
            )
    ) = 0; // Contract for pure virtual function.

    public: virtual const_reference back ( void ) const = 0;
};
```

Comma-Separated Syntax (variadic macros only, preferred)

```
CONTRACT_CLASS(
    template( typename T )
    class (pushable)
) {
    // Subcontracting: In `and` with derived class invariants.
    CONTRACT_CLASS_INVARIANT_TPL( void )

    public: typedef typename std::vector<T>::const_reference const_reference;

    CONTRACT_FUNCTION_TPL(
        public virtual void (push_back) ( (T const&) value ) new
            // Subcontracting: In `or` with overriding function preconditions.
            precondition( false ) // Force check to overriding preconditions.
            // Subcontracting: In `and` with overriding function postconditions.
            postcondition(
                back() == value, requires boost::has_equal_to<T>::value
            )
    ) = 0; // Contract for pure virtual function.

    public: virtual const_reference back ( void ) const = 0;
};
```

Sequence Syntax (all compilers)

```
#include "pushable_seq.hpp"

CONTRACT_CLASS(
    template( (typename T) )
    class (vector) extends( (public pushable<T>) )
) {
    // Use preprocessor sequences instead of variadic comma-separated lists.

    CONTRACT_CLASS_INVARIANT_TPL(
        (empty() == (size() == 0))
        (size() <= capacity())
        (capacity() <= max_size())
        (std::distance(begin(), end()) == int(size()))
    )

    public: typedef typename std::vector<T>::size_type size_type;
    public: typedef typename std::vector<T>::const_reference const_reference;
    public: typedef typename std::vector<T>::const_iterator const_iterator;

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( ((size_type) count) )
            precondition( (count >= 0) )
            postcondition(
                (size() == count)
                (boost::algorithm::all_of_equal(begin(), end(), T()))
            )
            initialize( (vector_(count)) )
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~vector) ( void )
    ) {}

    CONTRACT_FUNCTION_TPL(
        public void (push_back) ( ((T const&) value) )
            precondition( (size() < max_size()) )
            postcondition(
                (auto old_size = CONTRACT_OLDOF size())
                (auto old_capacity = CONTRACT_OLDOF capacity())
                (size() == old_size + 1)
                (capacity() >= old_capacity)
                // Overridden postconditions also check `back() == value`.
            )
    ) {
        vector_.push_back(value);
    }

    // ...
```

Comma-Separated Syntax (variadic macros only, preferred)

```
#include "pushable.hpp"

CONTRACT_CLASS(
    template( typename T )
    class (vector) extends( public pushable<T> ) // Subcontract.
) {
    // Within templates, use contract macros postfixed by `_TPL`.

    CONTRACT_CLASS_INVARIANT_TPL(
        empty() == (size() == 0),
        size() <= capacity(),
        capacity() <= max_size(),
        std::distance(begin(), end()) == int(size())
    )

    public: typedef typename std::vector<T>::size_type size_type;
    public: typedef typename std::vector<T>::const_reference const_reference;
    public: typedef typename std::vector<T>::const_iterator const_iterator;

    CONTRACT_CONSTRUCTOR_TPL(
        public explicit (vector) ( (size_type) count )
            precondition( count >= 0 )
            postcondition(
                size() == count,
                boost::algorithm::all_of_equal(begin(), end(), T())
            )
            initialize( vector_(count) )
    ) {}

    CONTRACT_DESTRUCTOR_TPL(
        public virtual (~vector) ( void )
    ) {}

    CONTRACT_FUNCTION_TPL(
        public void (push_back) ( (T const&) value )
            precondition( size() < max_size() )
            postcondition(
                auto old_size = CONTRACT_OLDOF size(),
                auto old_capacity = CONTRACT_OLDOF capacity(),
                size() == old_size + 1,
                capacity() >= old_capacity
                // Overridden postconditions also check `back() == value`.
            )
    ) {
        vector_.push_back(value);
    }

    // ...
```

Note the many extra parenthesis around all tokens within the lists: template parameters `(typename T)`, base classes `(public pushable<T>)`, function parameters `(size_type) count`, all assertions, etc. Furthermore, empty lists need to be specified using `(void)` instead of just `void`.

When using the sequence syntax, the macro `CONTRACT_LIMIT_OLDOFS` specifies the maximum number of postconditions sequence elements (instead of the maximum possible number of old value declarations as for variadic macros).

Commas and Leading Symbols in Macros

As we have seen in the Advanced Topics section, syntactic elements containing unwrapped commas and leading symbols need to be wrapped within extra round parenthesis:

```
(::std::map<char, int, std::less<char> >)          // With variadic macros.
```

However, without variadic macros this is no longer sufficient and the number of commas needs to be explicitly specified using the following syntax: [75]

```
comma(2)(::std::map<char, int, std::less<char> >)   // Without variadic macros.
```

For example (see also `macro_commas_symbols_integral_map_seq.cpp` and `macro_commas_symbols_integral_map.cpp`):

Sequence Syntax (all compilers)

```
CONTRACT_CLASS(
    template(
        (typename Key)
        (typename T)
        (class Allocator)
                (default comma(1)(::std::allocator<std::pair<Key const, T> >))
        (comma(1)(typename ::std::map<int, T>::key_type) default_key)
                (default (-1))
    ) requires( (comma(1)(boost::Convertible<Key, int>)) )
    class (integral_map)
        extends( (public comma(1)(::sizeable<Key, T>)) )
) {
    CONTRACT_CLASS_INVARIANT_TPL(
        ((((::sizeable<Key, T>::max_size)) >= size())
    )

    public: typedef typename std::map<Key, T, std::less<Key>, Allocator>::
            iterator iterator;

    CONTRACT_FUNCTION_TPL(
        public template( (typename OtherKey) (typename OtherT) )
            requires(
                (comma(1)(::boost::Convertible<OtherKey, Key>))
                (comma(1)(::boost::Convertible<OtherT, T>))
            )
        comma(1)(::std::pair<iterator, bool>) (insert) (
                (comma(1)(::std::pair<OtherKey, OtherT> const&) other_value)
                        (default comma(1)(::std::pair<Key,T>(default_key, T())))
            ) throw( (comma(1)(::std::pair<Key, T>)) )
            precondition(
                ((!full()))
            )
            postcondition(
                (auto result = return)
                (comma(1)(typename sizeable<OtherKey, OtherT>::size_type)
                        old_other_size = CONTRACT_OLDOF
                                (size<OtherKey, OtherT>()))
                ((((::sizeable<Key, T>::max_size)) >= size())
                (size() == old_other_size + (result.second ? 1 : 0))
            )
    ) {
        // ...
```

Comma-Separated Syntax (variadic macros only, preferred)

```
CONTRACT_CLASS(
    template(
        typename Key,
        typename T, // Commas in following template params.
        class Allocator,
                default (::std::allocator<std::pair<Key const, T> >),
        (typename ::std::map<int, T>::key_type) default_key,
                default (-1)
    ) requires( (boost::Convertible<Key, int>) ) // Commas in concepts.
    class (integral_map)
        extends( public (::sizeable<Key, T>) ) // Commas in bases.
) {
    CONTRACT_CLASS_INVARIANT_TPL( // Commas in class invariants.
        (::sizeable<Key, T>::max_size) >= size()
    )

    public: typedef typename std::map<Key, T, std::less<Key>, Allocator>::
            iterator iterator;

    CONTRACT_FUNCTION_TPL(
        public template( typename OtherKey, typename OtherT )
            requires(
                (::boost::Convertible<OtherKey, Key>),
                (::boost::Convertible<OtherT, T>)
            )
        (::std::pair<iterator, bool>) (insert) ( // Commas in result and params.
                (::std::pair<OtherKey, OtherT> const&) other_value,
                        default (::std::pair<Key, T>(default_key, T()))
            ) throw( (::std::pair<Key, T>) ) // Commas in exception specs.
            precondition( // Leading symbols in preconditions (same for commas).
                ((!full())) // LIMITATION: Two sets `((...))` (otherwise seq.).
            )
            postcondition( // Commas in postconditions.
                auto result = return,
                (typename sizeable<OtherKey, OtherT>::size_type)
                        old_other_size = CONTRACT_OLDOF
                                (size<OtherKey, OtherT>()),
                (::sizeable<Key, T>::max_size) >= size(),
                size() == old_other_size + (result.second ? 1 : 0)
            )
    ) {
        // ...
```

[75] **Rationale.** Using variadic macros, the preprocessor can automatically determine the number of commas within a tuple but without variadic macros that is no longer possible so programmers must manually specify the number of commas.

Reference

Header <contract/block_invariant.hpp>

Macros used to specify block invariants (this header is automatically included by `contract.hpp`).

```
CONTRACT_BLOCK_INVARIANT(assertions)
CONTRACT_BLOCK_INVARIANT_TPL(assertions)
```

Macro CONTRACT_BLOCK_INVARIANT

CONTRACT_BLOCK_INVARIANT — Macro used to specify block invariants.

Synopsis

```
// In header: <contract/block_invariant.hpp>

CONTRACT_BLOCK_INVARIANT(assertions)
```

Description

This macro can appear at any point in a code block within a function definition and it is used to assert correctness condition of the implementation (similarly to C++ `assert`).

Parameters:

`assertions`	The syntax for the assertions is explained in the Grammar section. Static assertions, constant assertions, and select assertions can be used.

Within a type-dependent scope (e.g., within templates), `CONTRACT_BLOCK_INVARIANT_TPL` must be used instead of this macro.

See also: Advanced Topics section.

Macro CONTRACT_BLOCK_INVARIANT_TPL

CONTRACT_BLOCK_INVARIANT_TPL — Macro used to specify block invariants within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/block_invariant.hpp>

CONTRACT_BLOCK_INVARIANT_TPL(assertions)
```

Description

This macro is the exact same as `CONTRACT_BLOCK_INVARIANT` but is must be used when specifying block invariants within a type-dependent scope.

See also: Advanced Topics section.

Header <contract/body.hpp>

Macros used to program body definitions separately from the contract declarations (this header is automatically include by `contract.hpp`).

```
CONTRACT_FREE_BODY(function_name)
CONTRACT_MEMBER_BODY(function_name)
CONTRACT_CONSTRUCTOR_BODY(class_type, constructor_name)
CONTRACT_DESTRUCTOR_BODY(class_type, destructor_name)
```

Macro CONTRACT_FREE_BODY

CONTRACT_FREE_BODY — Macro used to name the body of free functions and free function operators.

Synopsis

```
// In header: <contract/body.hpp>

CONTRACT_FREE_BODY(function_name)
```

Description

This macro is used to name the body of free functions and free function operators when the body is defined separately from the contract declaration. Free functions and free function operators with contracts are declared using the `CONTRACT_FUNCTION` macro.

Parameters:

`function_name`	The syntax for free function and free function operator names is explained in the Grammar section.

See also: Tutorial section.

Macro CONTRACT_MEMBER_BODY

CONTRACT_MEMBER_BODY — Macro used to name the body of member functions and member function operators.

Synopsis

```
// In header: <contract/body.hpp>

CONTRACT_MEMBER_BODY(function_name)
```

Description

This macro is used to name the body of member functions and member function operators when the body is defined separately from the contract declaration. Member functions and member function operators with contracts are declared using the `CONTRACT_FUNCTION` macro.

Parameters:

`function_name`	The syntax for function and operator names is explained in the Grammar section.

Warning: This macro must also be used when a virtual function invokes the overridden function from one of its base classes (see the Tutorial section).

See also: Tutorial section.

Macro CONTRACT_CONSTRUCTOR_BODY

CONTRACT_CONSTRUCTOR_BODY — Macro used to name the body of constructors.

Synopsis

```
// In header: <contract/body.hpp>

CONTRACT_CONSTRUCTOR_BODY(class_type, constructor_name)
```

Description

This macro is used to name the body of constructors when the body is defined separately from the contract declaration. Constructors with contracts are declared using the `CONTRACT_CONSTRUCTOR` macro.

Parameters:

`class_type`	The syntax for the class type is explained in the Grammar section (for class templates, this type is qualified with the template parameters).
`constructor_name`	This is the class name and its syntax is explained in the Grammar section (for class templates, this name is *not* qualified with the template parameters).

Warning: The body of constructors with member initializers should always be defined together with the constructor declaration and its contract.

See also: Tutorial section.

Macro CONTRACT_DESTRUCTOR_BODY

CONTRACT_DESTRUCTOR_BODY — Macro used to name the body of destructors.

Synopsis

```
// In header: <contract/body.hpp>

CONTRACT_DESTRUCTOR_BODY(class_type, destructor_name)
```

Description

This macro is used to name the body of destructors when the body is defined separately from the contract declaration. Destructors with contracts are declared using the `CONTRACT_DESTRUCTOR` macro.

Parameters:

`class_type`	The syntax for the class type is explained in the Grammar section (for class templates, this type is qualified with the template parameters).
`destructor_name`	This is the class name prefixed by the tilde symbol ~ and its syntax is explained in the Grammar section (for class templates, this name is *not* qualified with the template parameters).

See also: Tutorial section.

Header <contract/broken.hpp>

Contract broken handlers (this header is automatically included by `contract.hpp`).

```
namespace contract {
  class broken;

  enum from;
  typedef handler_function_pointer broken_contract_handler;

  // Set precondition broken handler to specified handler returning replaced handler.
  broken_contract_handler
  set_precondition_broken(broken_contract_handler handler);
  void precondition_broken(from const &);

  // Set postcondition broken handler to specified handler returning replaced handler.
  broken_contract_handler
  set_postcondition_broken(broken_contract_handler handler);
  void postcondition_broken(from const &);

  // Set handler for class invariant broken on entry to specified handler returning replaced handler.
  broken_contract_handler
  set_class_invariant_broken_on_entry(broken_contract_handler handler);
  void class_invariant_broken_on_entry(from const &);

  // Set handler for class invariant broken on exit to specified handler returning replaced handler.
  broken_contract_handler
  set_class_invariant_broken_on_exit(broken_contract_handler handler);
  void class_invariant_broken_on_exit(from const &);

  // Set handler for class invariant broken on throw to specified handler returning replaced handler.
  broken_contract_handler
  set_class_invariant_broken_on_throw(broken_contract_handler handler);
  void class_invariant_broken_on_throw(from const &);

  // For convenience, set all class invariant broken handlers (on entry, on exit, and on throw) to specified handler.
  void set_class_invariant_broken(broken_contract_handler handler);

  // Set block invariant broken handler to specified handler returning replaced handler.
  broken_contract_handler
  set_block_invariant_broken(broken_contract_handler handler);
  void block_invariant_broken(from const &);

  // Set the loop variant broken handler to specified handler returning replaced handler.
  broken_contract_handler
  set_loop_variant_broken(broken_contract_handler handler);
  void loop_variant_broken(from const &);
}
```

Class broken

contract::broken — Exception automatically thrown by the library to signal a contract assertion failure.

Synopsis

```
// In header: <contract/broken.hpp>


class broken {
public:
  // construct/copy/destruct
  explicit broken(char const *, unsigned long const &, char const *,
                  unsigned int const & = 0);
  broken(broken const &);
  broken& operator=(broken const &);
  ~broken(void);

  // public member functions
  char const * what(void) const;
  char const * file_name(void) const;
  unsigned long const line_number(void) const;
  char const * assertion_code(void) const;
  unsigned int const assertion_number(void) const;
};
```

Description

This exception contains detailed information about the failed contract assertion (file name, line number, etc).

A contract assertion is considered failed if it cannot be evaluated to be true (so if it is evaluated to be false but also if an exception is thrown by the code that evaluates the assertion condition). In case a contract assertion fails, the library automatically calls the appropriate contract broken handler with this exception as the active exception (the fact that assertion failures are signaled by this library by throwing this exception does not necessarily mean that the exception will be ultimately thrown by the broken contract, that is entirely up to the implementation of the contract broken handlers). By default, contract broken handlers print an error message to `std::cerr` and terminate the program calling `std::terminate`, but programmers can redefine this behaviour by customizing the contract broken handlers (for example to have the handlers throw exceptions instead of terminating the program). Within customized contract broken handlers, programmers can re-throw and catch this exception to obtained information about the failed asserted condition (file name, line number, etc).

Note: This exception is guaranteed to publicly inherit from `std::logic_error` (because `std::logic_error` models programming errors and those are the type of errors that contract assertions detect).

See also: Advanced Topics section.

`broken` public construct/copy/destruct

1.
```
explicit broken(char const * file_name, unsigned long const & line_number,
                char const * assertion_code,
                unsigned int const & assertion_number = 0);
```

Construct this exception specifying the failed assertion file name, line number, assertion code, and assertion number (no-throw).

The assertion number is optional and it defaults to zero (an assertion number equal to zero is ignored and not shown in the error description).

2.
```
broken(broken const & source);
```

Copy constructor (no-throw).

3.
```
broken& operator=(broken const & source);
```

Copy operator (no-throw).

4.
```
~broken(void);
```

Virtual destructor (no-throw).

`broken` public member functions

1. ```
   char const * what(void) const;
   ```

   Return a description of the assertion failure (no-throw).

2. ```
   char const * file_name(void) const;
   ```

   Return the name of the file containing the failed assertion (no-throw).

3. ```
   unsigned long const line_number(void) const;
   ```

   Return the number of the line containing the failed assertion (no-throw).

4. ```
   char const * assertion_code(void) const;
   ```

   Return the text of the asserted code that failed (no-throw).

5. ```
   unsigned int const assertion_number(void) const;
   ```

   Return the number of the failed assertion (no-throw).

   The assertion number is useful only within the context of a specific contract broken handler that can differentiates between broken preconditions, postconditions, etc because different preconditions, postconditions, etc will in general have the same assertion number (the first precondition is assertion number 1, the first postcondition is also assertion number 1, etc).

   If this number is zero then no sensible assertion number was specified and it should be ignored (e.g., loop variant assertions have no assertion number because there can only be one single variant for a loop).

Type from

contract::from — Specify the context from which the contract assertion failed.

Synopsis

```
// In header: <contract/broken.hpp>


enum from { FROM_CONSTRUCTOR, FROM_DESTRUCTOR, FROM_NONSTATIC_MEMBER_FUNCTION,
            FROM_STATIC_MEMBER_FUNCTION, FROM_FREE_FUNCTION, FROM_BODY };
```

Description

This information is important because in order to comply with STL exception safety requirements, destructors shall never throw an exception. Therefore, even if programmers customize the contract broken handlers to throw an exception instead of terminating the program, it is still important to know at least if the assertion failed from a destructor's contract so programmers can avoid throwing an exception from the broken handlers in such a case.

Note: All the different contexts identified by this enumeration have different contract checking semantics (see also Contract Programming Overview) so it might be relevant to differentiate between them in the contract broken handlers.

See also: Advanced Topics section.

`FROM_CONSTRUCTOR`	Assertion failed from within constructor contracts.
`FROM_DESTRUCTOR`	Assertion failed from within destructor contracts.
`FROM_NONSTATIC_MEMBER_FUNC-TION`	Assertion failed from within non-static member function contracts.
`FROM_STATIC_MEMBER_FUNCTION`	Assertion failed from within static member function contracts.
`FROM_FREE_FUNCTION`	Assertion failed from within free function contracts.
```

`FROM_BODY`	Assertion failed from within body contracts (for both block invariants and loop variants).

Type definition broken_contract_handler

broken_contract_handler — Contract broken handler function pointer.

Synopsis

```
// In header: <contract/broken.hpp>

typedef handler_function_pointer broken_contract_handler;
```

Description

A contract broken handler is a function returning `void` and taking only one parameter of type `from` indicating the context from which the contract assertion failed:

```
typedef void (*handler_function_pointer) ( from const& context ) ;
```

Note: This function prototype is not non-throw to allow programmers to customize the contract broken handlers to throw exceptions if they wish to do so.

See also: Advanced Topics section.

Function precondition_broken

contract::precondition_broken — Broken handler called when a precondition assertion fails.

Synopsis

```
// In header: <contract/broken.hpp>

void precondition_broken(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_precondition_broken` (even to throw exceptions).

Parameters:

`context`	Context from which the precondition assertion failed.

This handler is automatically called by the library in case of a precondition assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function postcondition_broken

contract::postcondition_broken — Broken handler called when a postcondition assertion fails.

Synopsis

```
// In header: <contract/broken.hpp>

void postcondition_broken(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_postcondition_broken` (even to throw exceptions).

Parameters:

`context`	Context from which the postcondition assertion failed.

This handler is automatically called by the library in case of a postcondition assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function class_invariant_broken_on_entry

contract::class_invariant_broken_on_entry — Broken handler called when a class invariant assertion fails on function entry.

Synopsis

```
// In header: <contract/broken.hpp>

void class_invariant_broken_on_entry(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_class_invariant_broken_on_entry` (even to throw exceptions, but programmers should be careful to never throw from destructors).

Parameters:

`context`	Context from which the class invariants assertion failed (destructors, etc).

This handler is automatically called by the library in case of a class invariant assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function class_invariant_broken_on_exit

contract::class_invariant_broken_on_exit — Broken handler called when a class invariant assertion fails on function exit.

Synopsis

```
// In header: <contract/broken.hpp>

void class_invariant_broken_on_exit(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_class_invariant_broken_on_exit` (even to throw exceptions, but programmers should be careful to never throw from destructors).

Parameters:

`context`	Context from which the class invariants assertion failed (destructors, etc).

This handler is automatically called by the library in case of a class invariant assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function class_invariant_broken_on_throw

contract::class_invariant_broken_on_throw — Broken handler called when a class invariant assertion fails on function exit after the function body threw an exception.

Synopsis

```
// In header: <contract/broken.hpp>

void class_invariant_broken_on_throw(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_class_invariant_broken_on_throw` (even to throw exceptions, but programmers should be careful to never throw from destructors).

Parameters:

`context`	Context from which the class invariants assertion failed (destructors, etc).

This handler is automatically called by the library in case of a class invariant assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function block_invariant_broken

contract::block_invariant_broken — Broken handler called when a block invariant assertion fails.

Synopsis

```
// In header: <contract/broken.hpp>

void block_invariant_broken(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_block_invariant_broken` (even to throw exceptions, but programmers should be careful to never throw from destructors).

Parameters:

`context`	Context from which the block invariants assertion failed (i.e., the body).

This handler is automatically called by the library in case of a block invariant assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Function loop_variant_broken

contract::loop_variant_broken — Broken handler called when a loop variant assertion fails.

Synopsis

```
// In header: <contract/broken.hpp>

void loop_variant_broken(from const & context);
```

Description

By default, it prints information about the failed assertion to `std::cerr` and it calls `std::terminate`. However, it can be customized using `contract::set_loop_variant_broken` (even to throw exceptions, but programmers should be careful to never throw from destructors).

Parameters:

`context`	Context from which the loop variants assertion failed (i.e., the body).

This handler is automatically called by the library in case of a loop invariant assertion failure (an assertion fails if it is not evaluated to be true, so if it is evaluated to be false but also if it cannot be evaluated because an exception is thrown by the asserted condition).

See also: Advanced Topics section.

Header <contract/class.hpp>

Macros used to declare classes with contracts (this header is automatically included by `contract.hpp`).

```


CONTRACT_CLASS(class_declaration)
CONTRACT_CLASS_TPL(class_declaration)
```

Macro CONTRACT_CLASS

CONTRACT_CLASS — Macro used to declare classes with contracts.

Synopsis

```
// In header: <contract/class.hpp>

CONTRACT_CLASS(class_declaration)
```

Description

This macro is used to declare a class with class invariants and member functions with preconditions and postconditions. A class declared with this macro must always specify (possibly empty) class invariants using the `CONTRACT_CLASS_INVARIANT` macro.

Constructors, destructors, and member functions with contracts are declared using the `CONTRACT_CONSTRUCTOR`, `CONTRACT_DESTRUCTOR`, and `CONTRACT_FUNCTION` macros respectively. The semantics of calls to constructors, destructors, and member functions with contracts are explained in the Contract Programming Overview section.

Parameters:

`class_declaration`	The class declaration syntax is explained in the Grammar section. If present, base classes must be specified using `extends( ... )`.

Nested classes with contracts are also declared using this macro but they must always repeat their access level `public`, `protected`, or `private`. Within a type-dependent scope, nested classes with contracts must be declared using the `CONTRACT_CLASS_TPL` macro.

See also: Tutorial section.

Macro CONTRACT_CLASS_TPL

CONTRACT_CLASS_TPL — Macro used to declare nested classes with contracts within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/class.hpp>

CONTRACT_CLASS_TPL(class_declaration)
```

Description

This macro is the exact same as `CONTRACT_CLASS` but it must be used when declaring nested classes with contracts within a type-dependent scope.

See also: Tutorial section.

Header <contract/class_invariant.hpp>

Macros used to specify class invariants (this header is automatically included by `contract.hpp`).

```


CONTRACT_CLASS_INVARIANT(class_invariants)
CONTRACT_CLASS_INVARIANT_TPL(class_invariants)
```

Macro CONTRACT_CLASS_INVARIANT

CONTRACT_CLASS_INVARIANT — Macro used to specify class invariants.

Synopsis

```
// In header: <contract/class_invariant.hpp>

CONTRACT_CLASS_INVARIANT(class_invariants)
```

Description

This macro must be used to specify class invariant at the very beginning of the definition of a class with contracts declared using the `CONTRACT_CLASS` macro (no other statement can appear before this macro in the class definition, not even a `typedef` or a `friend` declaration).

Constructors, destructors, and member functions with contracts are declared using the `CONTRACT_CONSTRUCTOR`, `CONTRACT_DESTRUCTOR`, and `CONTRACT_FUNCTION` macros respectively. The semantics of calls to constructors, destructors, and member functions of a class with contracts are explained in the Contract Programming Overview section.

Parameters:

`class_invariants`	The syntax for class invariants is explained in the Grammar section. Use `void` to specify empty class invariant.

Within a type-dependent scope (e.g., within templates), the `CONTRACT_CLASS_INVARIANT_TPL` macro must be used instead of this macro.

See also: Tutorial section.

Macro CONTRACT_CLASS_INVARIANT_TPL

CONTRACT_CLASS_INVARIANT_TPL — Macro used to specify class invariants within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/class_invariant.hpp>

CONTRACT_CLASS_INVARIANT_TPL(class_invariants)
```

Description

This macro is the exact same as `CONTRACT_CLASS_INVARIANT` but it must be used when specifying class invariants within a type-dependent scope.

See also: Tutorial section.

Header <contract/config.hpp>

Macros used to configure the library behaviour at compile-time (this header is automatically included by `contract.hpp`).

These configuration macros have appropriate default values if left undefined. Programmers can define these macros before including any of the library headers (using compiler options like `-D` for GCC, `/D` for MSVC, etc) in order to change the library behaviour.

The macros `CONTRACT_CONFIG_NO_PRECONDITIONS`, `CONTRACT_CONFIG_NO_POSTCONDITIONS`, etc can be defined to selectively disable compilation and run-time checking of all preconditions, postconditions, etc. This is a common practice in Contract Programming to generate debug and release builds with less correctness checks but faster run-times. Note that all contracts are compiled and checked at run-time by default unless specified otherwise using these configuration macros.

See also: Getting Started section, Contract Programming Overview section.

```
CONTRACT_CONFIG_NO_PRECONDITIONS
CONTRACT_CONFIG_NO_POSTCONDITIONS
CONTRACT_CONFIG_NO_CLASS_INVARIANTS
CONTRACT_CONFIG_NO_BLOCK_INVARIANTS
CONTRACT_CONFIG_NO_LOOP_VARIANTS
CONTRACT_CONFIG_FUNCTION_ARITY_MAX
CONTRACT_CONFIG_INHERITANCE_MAX
CONTRACT_CONFIG_OLDOF_MAX
CONTRACT_CONFIG_ARRAY_DIMENSION_MAX
CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS
CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION
CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE
CONTRACT_CONFIG_THREAD_SAFE
```

Macro CONTRACT_CONFIG_NO_PRECONDITIONS

CONTRACT_CONFIG_NO_PRECONDITIONS — Disable compilation and run-time checking of all preconditions.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_NO_PRECONDITIONS
```

Description

Preconditions are not checked at run-time and not even compiled when programmers define this macro. This can be used to speed up execution and compilation of debug and release builds at different stages of development and testing as it is common practise in Contract Programming.

Note: Assertion requirements can be used to selectively disable single assertions (see the Advanced Topics section).

See also: Getting Started section.

Macro CONTRACT_CONFIG_NO_POSTCONDITIONS

CONTRACT_CONFIG_NO_POSTCONDITIONS — Disable compilation and run-time checking of all postconditions.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_NO_POSTCONDITIONS
```

Description

Postconditions are not checked at run-time and not even compiled when programmers define this macro. This can be used to speed up execution and compilation of debug and release builds at different stages of development and testing as it is common practise in Contract Programming.

Note: Assertion requirements can be used to selectively disable single assertions (see the Advanced Topics section).

See also: Getting Started section.

Macro CONTRACT_CONFIG_NO_CLASS_INVARIANTS

CONTRACT_CONFIG_NO_CLASS_INVARIANTS — Disable compilation and run-time checking of all class invariants.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_NO_CLASS_INVARIANTS
```

Description

Class invariants are not checked at run-time and not even compiled when programmers define this macro. This can be used to speed up execution and compilation of debug and release builds at different stages of development and testing as it is common practise in Contract Programming.

Note: Assertion requirements can be used to selectively disable single assertions (see the Advanced Topics section).

See also: Getting Started section.

Macro CONTRACT_CONFIG_NO_BLOCK_INVARIANTS

CONTRACT_CONFIG_NO_BLOCK_INVARIANTS — Disable compilation and run-time checking of all block invariants.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_NO_BLOCK_INVARIANTS
```

Description

Block invariants are not checked at run-time and not even compiled when programmers define this macro. This can be used to speed up execution and compilation of debug and release builds at different stages of development and testing as it is common practise in Contract Programming.

Note: Assertion requirements can be used to selectively disable single assertions (see the Advanced Topics section).

See also: Getting Started section.

Macro CONTRACT_CONFIG_NO_LOOP_VARIANTS

CONTRACT_CONFIG_NO_LOOP_VARIANTS — Disable compilation and run-time checking of all loop variants.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_NO_LOOP_VARIANTS
```

Description

Loop variants are not checked at run-time and not even compiled when programmers define this macro. This can be used to speed up execution and compilation of debug and release builds at different stages of development and testing as it is common practise in Contract Programming.

Note: Assertion requirements can be used to selectively disable single assertions (see the Advanced Topics section).

See also: Getting Started section.

Macro CONTRACT_CONFIG_FUNCTION_ARITY_MAX

CONTRACT_CONFIG_FUNCTION_ARITY_MAX — Specify the maximum number of supported function parameters.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_FUNCTION_ARITY_MAX
```

Description

This macro automatically defaults to `5` if left undefined by programmers. This macro must be a non-negative integral number. Increasing the value specified by this macro might increase compilation time.

See also: Tutorial section.

Macro CONTRACT_CONFIG_INHERITANCE_MAX

CONTRACT_CONFIG_INHERITANCE_MAX — Specify the maximum number of base classes supported for multiple-inheritance.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_INHERITANCE_MAX
```

Description

This macro automatically defaults to `4` if left undefined by programmers. This macro must be a non-negative integral number. Increasing the value specified by this macro might increase compilation time.

See also: Tutorial section.

Macro CONTRACT_CONFIG_OLDOF_MAX

CONTRACT_CONFIG_OLDOF_MAX — Specify the maximum number of supported old value declarations in postconditions.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_OLDOF_MAX
```

Description

This macro automatically defaults to `5` if left undefined by programmers. This macro must be a non-negative integral number smaller or equal to `CONTRACT_LIMIT_OLDOFS`. Increasing the value specified by this macro might increase compilation time.

See also: Tutorial section.

Macro CONTRACT_CONFIG_ARRAY_DIMENSION_MAX

CONTRACT_CONFIG_ARRAY_DIMENSION_MAX — Specify the maximum supported array dimension for multi-dimensional arrays.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_ARRAY_DIMENSION_MAX
```

Description

This macro automatically defaults to `3` if left undefined by programmers (therefore, by default up to 3-dimensional arrays `x[]`, `x[][]`, and `x[][][]` can be passed to functions declared using this library, but always using an extra `typedef` as specified by the Tutorial section). This macro must be a non-negative integral number. Increasing the value specified by this macro might increase compilation time.

See also: Tutorial section.

Macro CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS

CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS — Do not allow overriding functions to specify preconditions.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_DO_NOT_SUBCONTRACT_PRECONDITIONS
```

Description

If programmers define this macro, the library will generate a compile-time error if overriding functions specify preconditions. Therefore, preconditions can only be specified by functions at the very root of the inheritance tree and they cannot be subcontracted (this is as specified by [N1962], however note that in case of multiple-inheritance the preconditions of all base functions will still be checked in logic-or with each other effectively still allowing a base class to weaken the preconditions from another base class).

By default, this library allows to subcontract preconditions.

See also: Advanced Topics section.

Macro CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION

CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION — Specify that no assertion should be disabled while checking preconditions.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_PRECONDITIONS_DISABLE_NO_ASSERTION
```

Description

Assertion checking is disabled within assertions in order to avoid infinite recursion and that is a standard requirement of Contract Programming. However, if programmers define this macro then no assertion checking is disabled within preconditions (this is as specified by [N1962]). Assertion checking within assertion checking will still be disabled for postconditions, class invariants, etc.

By default, this library disables assertion checking within assertions for all contracts, including preconditions.

See also: Contract Programming Overview section.

Macro CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE

CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE — Report precondition that failed in the overridden function (instead of the precondition failed in the overriding function).

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_REPORT_BASE_PRECONDITION_FAILURE
```

Description

Subcontracted preconditions fail only when the preconditions of the overriding function fail together with the overridden preconditions from all the base classes. By default the assertion that failed in the overriding function is reported. If programmes define this macro then the library will instead the report the assertion that failed in the overridden function from within one of the base classes.

See also: Advanced Topics section.

Macro CONTRACT_CONFIG_THREAD_SAFE

CONTRACT_CONFIG_THREAD_SAFE — Make the implementation of this library thread-safe.

Synopsis

```
// In header: <contract/config.hpp>

CONTRACT_CONFIG_THREAD_SAFE
```

Description

In order to globally disable assertion checking within assertions, this library internally has to use a global variable. If programmers defined this macro, such a variable will be automatically locked to avoid race conditions (this effectively introduces a global lock in the program).

See also: Contract Programming Overview section.

Header <contract/constructor.hpp>

Macros used to declare constructors with contracts (this header is automatically included by `contract.hpp`).

```


CONTRACT_CONSTRUCTOR(function_declaration)
CONTRACT_CONSTRUCTOR_TPL(function_declaration)
```

Macro CONTRACT_CONSTRUCTOR

CONTRACT_CONSTRUCTOR — Macro used to declare constructors with contracts.

Synopsis

```
// In header: <contract/constructor.hpp>

CONTRACT_CONSTRUCTOR(function_declaration)
```

Description

This macro is used to declare a constructor with possible preconditions and postconditions. At least all public constructors of a class with non-empty class invariants should be declared using this macro so to check the class invariants (even if the constructors have no precondition and no postcondition).

The semantics of a call to a constructor with contracts are explained in the Contract Programming Overview section. Destructors and member functions are declared using the `CONTRACT_DESTRUCTOR` and `CONTRACT_FUNCTION` macros respectively.

Parameters:

`function_declaration`	The constructor declaration syntax is explained in the Grammar section. Constructors must always repeat their access level `public`, `protected`, or `private`. If present, member initializers must be specified using `initialize( ... )` and within this macro (even if they are technically part of the constructor definition and not of its declaration).

The `CONTRACT_CLASS` macro must be used to declare the class enclosing a constructor declared using this macro.

Within a type-dependent scope (e.g., within templates), the `CONTRACT_CONSTRUCTOR_TPL` macro must be used instead of this macro.

The `CONTRACT_CONSTRUCTOR_BODY` macro must be used when separating the constructor body definition form the constructor declaration programmed using this macro. Unfortunately, it is not possible to use this macro so separate body definitions for constructors that have member initializers.

See also: Tutorial section.

Macro CONTRACT_CONSTRUCTOR_TPL

CONTRACT_CONSTRUCTOR_TPL — Macro used to declare constructors with contracts within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/constructor.hpp>

CONTRACT_CONSTRUCTOR_TPL(function_declaration)
```

Description

This macro is the exact same as `CONTRACT_CONSTRUCTOR` but it must be used when declaring constructors with contracts within a type-dependent scope.

See also: Tutorial section.

Header <contract/copy.hpp>

Copy result value and old values for postconditions (this header is automatically included by `contract.hpp`).

```
namespace contract {
  template<typename T> class copy;
}
```

Class template copy

contract::copy — Copy result value and old values for postconditions.

Synopsis

```
// In header: <contract/copy.hpp>

template<typename T>
class copy {
public:
  // types
  typedef boost::add_reference< typename boost::add_const< T >::type >::type const_reference_type;

  // construct/copy/destruct
  explicit copy(const_reference_type);

  // public member functions
  const_reference_type value(void) const;
};
```

Description

The default implementation of his class template can copy values of any type that is `ConstantCopyConstructible` which is of any type `T` that has a constant-correct copy constructor of the form:

```
    T::T( T const& source ) ;
```

Parameters:

T	The type of the value to copy (it is either the result type of the type of an old-of expression).

Programmers can specialize this class template to copy a specific type differently (and even for types that are not `ConstantCopyConstructible`, or not `CopyConstructible` at all).

Warning: When specializing this class template, programmers must make sure that their code is constant-correct otherwise the contracts will no longer be constant-correct.

See also: Advanced Topics section.

copy public construct/copy/destruct

1. ```
   explicit copy(const_reference_type value);
   ```

   Construct this object copying the specified value assuming `T` has a constant-correct copy constructor.

copy public member functions

1. ```
   const_reference_type value(void) const;
   ```

   Return a constant reference to the copied value.

Header <contract/destructor.hpp>

Macros used to declare destructors with contracts (this header is automatically included by `contract.hpp`).

```
CONTRACT_DESTRUCTOR(function_declaration)
CONTRACT_DESTRUCTOR_TPL(function_declaration)
```

Macro CONTRACT_DESTRUCTOR

CONTRACT_DESTRUCTOR — Macro used to declare destructors with contracts.

Synopsis

```
// In header: <contract/destructor.hpp>

CONTRACT_DESTRUCTOR(function_declaration)
```

Description

This macro is used to declare a destructor. At least the public destructor of a class with non-empty class invariants should be declared using this macro in order to check the class invariants (even if destructors never have preconditions or postconditions).

The semantics of a call to a destructor with contracts are explained in the Contract Programming Overview section. Constructor and member functions are declared using the `CONTRACT_CONSTRUCTOR` and `CONTRACT_FUNCTION` macros respectively.

Parameters:

function_declaration	The destructor declaration syntax is explained in the Grammar section. Destructors must always repeat their access level `public`, `protected`, or `private`. The keyword `void` must be used to indicate that the destructor has an empty parameter list.

The `CONTRACT_CLASS` macro must be used to declare the class enclosing a destructor declared using this macro.

Within a type-dependent scope (e.g., within templates), the `CONTRACT_DESTRUCTOR_TPL` macro must be used instead of this macro.

The `CONTRACT_DESTRUCTOR_BODY` macro must be used when separating the destructor body definition from the destruction declaration programmed using this macro.

See also: Tutorial section.

Macro CONTRACT_DESTRUCTOR_TPL

CONTRACT_DESTRUCTOR_TPL — Macro used to declare destructors with contracts within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/destructor.hpp>

CONTRACT_DESTRUCTOR_TPL(function_declaration)
```

Description

This macro is the exact same as `CONTRACT_DESTRUCTOR` but it must be used when declaring destructors with contracts within a type-dependent scope.

See also: Tutorial section.

Header <contract/function.hpp>

Macros used to declare free functions, member functions, and operators with contracts (this header is automatically included by `contract.hpp`).

```


CONTRACT_FUNCTION(function_declaration)
CONTRACT_FUNCTION_TPL(function_declaration)
```

Macro CONTRACT_FUNCTION

CONTRACT_FUNCTION — Macro used to declare free functions, free function operators, member functions, and member function operators with contracts.

Synopsis

```
// In header: <contract/function.hpp>

CONTRACT_FUNCTION(function_declaration)
```

Description

This macro is used to declare a function with possible preconditions and postconditions. At least all public member functions and public member function operators of a class with non-empty class invariants should be declared using this macro in order to check the class invariants (even if the functions and operators have no precondition and no postcondition).

The semantics of a call to a function with contracts are explained in the Contract Programming Overview section. Constructors and destructors are declared using the `CONTRACT_CONSTRUCTOR` and `CONTRACT_DESTRUCTOR` macros respectively.

Parameters:

`function_declaration`	The function declaration syntax is explained in the Grammar section. Member functions must always repeat their access level `public`, `protected`, or `private`. Operators use a special syntax to specify their symbols and names.

The `CONTRACT_CLASS` macro must be used to declare classes enclosing member functions and member function operators declared using this macro.

Within a type-dependent scope (e.g., within templates), the `CONTRACT_FUNCTION_TPL` macro must be used instead of this macro.

The `CONTRACT_FREE_BODY` and `CONTRACT_MEMBER_BODY` macros must be used when separating free and member function body definitions from the function declarations programmed using this macro.

See also: Tutorial section.

Macro CONTRACT_FUNCTION_TPL

CONTRACT_FUNCTION_TPL — Macro used to declared free functions, free function operators, member functions, and member function operators with contracts within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/function.hpp>

CONTRACT_FUNCTION_TPL(function_declaration)
```

Description

This macro is the exact same as `CONTRACT_FUNCTION` but it must be used when declaring functions with contracts within a type-dependent scope.

See also: Tutorial section.

Header <contract/limits.hpp>

Macros reporting bounds of some library constructs (this header is automatically included by `contract.hpp`).

These are *not* configuration macros so programmers cannot change these values. These macros are used to inform programmers of bounds on some of this library constructs.

```
CONTRACT_LIMIT_OLDOFS
CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS
CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES
```

Macro CONTRACT_LIMIT_OLDOFS

CONTRACT_LIMIT_OLDOFS — Upper bound on possible maximum number of postcondition old variables.

Synopsis

```
// In header: <contract/limits.hpp>

CONTRACT_LIMIT_OLDOFS
```

Description

The maximum possible number of postcondition old variables is `15` (the actual maximum value is controlled by `CONTRACT_CONFIG_OLDOF_MAX`).

For compilers that do not support variadic macros, this is the maximum possible total number of postcondition statements (total of old variable declarations, plus return value declaration, plus assertions, etc).

Note: This is not a configuration macro. The value of this macro is fixed and programmers cannot change it.

See also: Tutorial section, No Variadic Macros section.

Macro CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS

CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS — Maximum number of select assertions that can be nested into one another.

Synopsis

```
// In header: <contract/limits.hpp>

CONTRACT_LIMIT_NESTED_SELECT_ASSERTIONS
```

Description

The maximum number of select assertions that can be nested into one another is `5`.

Note: This is not a configuration macro. The value of this macro is fixed and programmers cannot change it.

See also: Advanced Topics section.

Macro CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES

CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES — Maximum number of catch statements for a constructor-try block with member initializers.

Synopsis

```
// In header: <contract/limits.hpp>

CONTRACT_LIMIT_CONSTRUCTOR_TRY_BLOCK_CATCHES
```

Description

The maximum number of catch statements for a constructor-try block with member initializers is `10`. (Constructor-try blocks are specified outside the `CONTRACT_CONSTRUCTOR` macro when the is no member initializers so this limit does not apply to that case.)

Note: This is not a configuration macro. The value of this macro is fixed and programmers cannot change it.

See also: Advanced Topics section.

Header <contract/loop_variant.hpp>

Macros used to specify loop variants (this header is automatically included by `contract.hpp`).

```


CONTRACT_LOOP(loop_declaration)
CONTRACT_LOOP_VARIANT(loop_variant)
CONTRACT_LOOP_VARIANT_TPL(loop_variant)
```

Macro CONTRACT_LOOP

CONTRACT_LOOP — Macro used to declare a loop that will specify a loop variant.

Synopsis

```
// In header: <contract/loop_variant.hpp>

CONTRACT_LOOP(loop_declaration)
```

Description

This macro must be used to declare a loop that will later specify a loop variant using the `CONTRACT_LOOP_VARIANT` macro.

Parameters:

`loop_declaration`	This is a loop declaration (`for`, `while`, etc) that follows the usual C++ syntax (see also the Grammar section).

See also: Advanced Topics section.

Macro CONTRACT_LOOP_VARIANT

CONTRACT_LOOP_VARIANT — Macro used to specify a loop variant.

Synopsis

```
// In header: <contract/loop_variant.hpp>

CONTRACT_LOOP_VARIANT(loop_variant)
```

Description

This macro is used to specify loop variants which ensure that loops terminate. This macro must be used within the body `{ ... }` of a loop declared using the `CONTRACT_LOOP` macro. Each loop can have at most one loop variant.

Parameters:

`loop_variant`	The loop variant must be a non-negative integral expression that monotonically decreases when calculated at each subsequent loop iteration (constant-expressions can also be used, see the Grammar section).

Within type-dependent scope (e.g., within templates), the `CONTRACT_LOOP_VARIANT_TPL` macro must be used instead of this macro.

See also: Advanced Topics section.

Macro CONTRACT_LOOP_VARIANT_TPL

CONTRACT_LOOP_VARIANT_TPL — Macro used to specify a loop variant within a type-dependent scope (e.g., within templates).

Synopsis

```
// In header: <contract/loop_variant.hpp>

CONTRACT_LOOP_VARIANT_TPL(loop_variant)
```

Description

This macro is the exact same as `CONTRACT_LOOP_VARIANT` but it must be used when specifying loop variants within a type-dependent scope.

See also: Advanced Topics section.

Header <contract/oldof.hpp>

Constructs to declare postcondition old values (this header is automatically included by `contract.hpp`).

```
CONTRACT_OLDOF
```

```
namespace contract {
  template<typename OldofExpressionType> struct has_oldof;
}
```

Struct template has_oldof

contract::has_oldof — Trait used to determine if a type can be copied for an old value or not.

Synopsis

```
// In header: <contract/oldof.hpp>

template<typename OldofExpressionType>
struct has_oldof {
};
```

Description

A type can be used for a postcondition old value if and only if it can be copied using the `contract::copy` template. By default, all types that have a constant-correct copy constructor (i.e., that are `ConstantCopyConstructible`) can be copied by `contract::copy`.

If a type cannot be copied, an old value declaration using such a type will not fail but it will generate an old value that will cause a compile-time error as soon as it is used in a contract assertion. Therefore, this trait can be used to program assertion requirements for those assertions that use old values on generic types that are not always known to be copyable. This way, the assertions will be disabled by the requirement instead of generating a compile-time error when the old values they use cannot be copied.

Parameters:

`OldofExpressionType`	The type of the expression specified to the `CONTRACT_OLDOF` macro for a given postcondition old value declaration.

Unfortunately, in C++ it is not possible to portably implement a template meta-function that checks if a generic type has a constant-correct copy constructor (or even just a copy constructor). Therefore, the default implementation of this trait is `boost::mpl::true_` for any type `OdlofExpressionType`. This will cause compile-time errors for old value types that cannot be copied in which case users can specialize this unary boolean meta-function to inherit from `boost::mpl::false_` for such types.

See also: Advanced Topics section.

Macro CONTRACT_OLDOF

CONTRACT_OLDOF — Macro used to refer to the old value of an expression.

Synopsis

```
// In header: <contract/oldof.hpp>

CONTRACT_OLDOF
```

Description

This macro refers the old value of the expression that follows the macro (i.e., the value the expression had after function entry but before body execution). This macro must be used to initialize old value declarations within postconditions (see also the Grammar section):

```
auto old_variable = CONTRACT_OLDOF odlof_expression
```

The specified expression that follows the macro might or not be wrapped within parenthesis.

The old-of expression type can be explicitly specified instead of `auto` but it must be wrapped within parenthesis unless it is a fundamental type containing no symbol. If not explicitly specified, the library will automatically deduce the type using Boost.Typeof (as always, types must be properly registered with Boost.Typeof for type-of emulation mode on compilers that do not support native type-of).

See also: Tutorial section, Advanced Topics section.

Header <contract/parameter.hpp>

Macros used to program named and deduced parameters (this header is automatically included by `contract.hpp`).

```
CONTRACT_CONSTRUCTOR_ARG(parameter_name)
CONTRACT_PARAMETER_TYPEOF(parameter_name)
CONTRACT_PARAMETER(named_parameter_declaration)
CONTRACT_TEMPLATE_PARAMETER(named_parameter_declaration)
CONTRACT_PARAMETER_BODY(function_name)
```

Macro CONTRACT_CONSTRUCTOR_ARG

CONTRACT_CONSTRUCTOR_ARG — Macro used to access constructor named or deduced arguments within member initializers.

Synopsis

```
// In header: <contract/parameter.hpp>

CONTRACT_CONSTRUCTOR_ARG(parameter_name)
```

Description

This macro must be used to access the constructor arguments within the member initializers. Outside of the member initializers, the parameter names are used directly as usual and without using this macro.

Parameters:

`parameter_name`	The name of a constructor named or deduced parameter previously declared using the `CONTRACT_PARAMETER` macro.

See also: Named Parameters section.

Macro CONTRACT_PARAMETER_TYPEOF

CONTRACT_PARAMETER_TYPEOF — Macro used to access the actual type of a named or deduced parameter.

Synopsis

```
// In header: <contract/parameter.hpp>

CONTRACT_PARAMETER_TYPEOF(parameter_name)
```

Description

Named and deduced parameters can have generic types (possibly matching predicate type requirements) so the actual parameter types are known at compile-time but only after the function call has been resolved. This macro is used to refer to the actual parameter type as it was determined by the function call resolution.

Parameters:

`parameter_name`	The name of a named or deduced parameter previously declared using the `CONTRACT_PARAMETER` macro.

This macro can be used within both the function declaration and the body definition.

See also: Named Parameters section.

Macro CONTRACT_PARAMETER

CONTRACT_PARAMETER — Macro used to declare a named or deduced function parameter.

Synopsis

```
// In header: <contract/parameter.hpp>

CONTRACT_PARAMETER(named_parameter_declaration)
```

Description

This macro is used to declare a named or deduced parameter that will later be used within a function declared using the `CONTRACT_FUNCTION` or the `CONTRACT_CONSTRUCTOR` macros. This macro should be used at namespace scope.

Parameters:

`named_parameter_declaration`	The syntax of named and deduced parameter declarations is explained in the Grammar section. At lest the parameter name must be specified.

It is recommended to always use this macro within an enclosing namespace different from the global namespace so to control and avoid clashing declarations of named and deduced parameters that have the same name.

See also: Named Parameters section.

Macro CONTRACT_TEMPLATE_PARAMETER

CONTRACT_TEMPLATE_PARAMETER — Macro used to declare a named or deduced template parameter.

Synopsis

```
// In header: <contract/parameter.hpp>

CONTRACT_TEMPLATE_PARAMETER(named_parameter_declaration)
```

Description

This macro is used to declare a named or deduced parameter that will later be used within a class template declared using the `CONTRACT_CLASS` macro. This macro should be used at namespace scope.

Parameters:

`named_parameter_declaration`	The syntax of named and deduced parameter declarations is explained in the Grammar section. At lest the template parameter name must be specified.

It is recommended to always use this macro within an enclosing namespace different from the global namespace so to control and avoid clashing declarations of named and deduced parameters that have the same name.

See also: Named Parameters section.

Macro CONTRACT_PARAMETER_BODY

CONTRACT_PARAMETER_BODY — Macro used to name the body of free and member functions with named parameters.

Synopsis

```
// In header: <contract/parameter.hpp>

CONTRACT_PARAMETER_BODY(function_name)
```

Description

This macro is used to name the body of a function with named parameters when the body is defined separately from the function declaration.

Parameters:

`function_name`	The name of the function with named parameters.

For member functions, the class type must precede this macro (this allows to use this same macro for both free and member functions):

```
class_type::CONTRACT_PARAMETER_BODY(function_name)
```

Note: Named parameters are currently not supported for operators so this function name cannot be the name of an operator (because of a Boost.Parameter bug). Constructors with named parameters cannot defer the definition of their body (because of lack of delegating constructors in C++03). Finally, destructors have no parameter so named parameters do not apply to destructors. Therefore, this macro is never used for operators, constructors, and destructors.

See also: Named Parameters section.

Release Notes

This section contains notes on the current and on all previous library releases (in chronological order).

Release 0.4.1

August 20, 2012

Notes:

1. Using non-fix-spaced font in Full Table of Contents section.
2. Added a couple of notes to the documentation.
3. Changed `CONTRACT_MEMBER_BODY(class_type, function_name)` to `class_type::CONTRACT_MEMBER_BODY(function_name)` so the macro can also be used to declare derived classes avoiding using the library syntax even when the base class has contracts.

Release files and documentation.

Release 0.4.0

June 4, 2012

Notes:

1. Simplified syntax by reducing extra parenthesis to the bare necessary minimum (using some of the preprocessor parsing techniques originally introduced by Boost.LocalFunction).
2. Postcondition old values only copy the old-of expression (e.g., copy just vector size instead of entire vector). This improves performance and introduces the `ConstantCopyConstructible` requirement just for the old value expression type (e.g., a vector might not be copyable while its size always is because it is an integral type). Removed the `copyable` tag.
3. Body defined outside the macros (so compiler-errors for definitions retain their usual meaning).
4. Added `CONTRACT_CLASS` macro and removed the need to duplicate declaration elements (do not repeat function declaration, do not repeat class name in function declaration, etc).
5. Using `_TPL` macros so to reduce compile-time (instead of internally making all templates contract functions so to use `typename` freely).
6. Overloading no longer requires unique parameter names.
7. Added C++11-like virtual specifiers.
8. Added constant assertions plus constant-expressions for select assertion if-conditions and for loop variants.
9. Added named and deduced parameters.
10. Added concept checking.
11. Removed the interface to use the library without the macro (programmers were required to write too much boiler-plate code for the non-macro interface to be actually usable, plus supporting both the macro and non-macro interfaces limited what the macros could do).

Release files and documentation.

Release 0.3.490

March 7, 2010

Notes:

1. Added support and examples for `volatile`, `auto`, `explicit`, `export`, `extern`, `friend`, `inline`, `struct`, and `throw` (for exception specifications).
2. Documented that `union` cannot be contracted.

Release files and documentation.

Release 0.3.469

February 21, 2010

Notes:

1. Removed use of `self`, `variable.now`, and `variable.old` in writing contracts. Object `this` and variables are now accessed as usual in member functions. `CONTRACT_OLDOF(variable)` is used to access old values in postconditions.
2. Added `(precondition)`, `(postcondition)`, and `(body)` to specify contracts within the function signature sequence. If no preconditions then `(precondition) ({...})` is simply omitted from the sequence (same for postconditions, body is mandatory instead). For non-void functions, users can name the result argument with `(postcondition) (result-name) ({...})`.
3. Changed contract class template to use same syntax as Boost.Function (i.e., `F` function type).
4. Added support for free functions and static member functions.
5. Added support for subcontracting with multiple inheritance.
6. Added static class invariants which are always checked (also at constructors entry, destructor exit, and by static member functions).
7. Added block invariants and Eiffel-like loop variants.
8. Added handlers to customize action on contract failure (default to `std::terminate()`).
9. Removed feature for automatic contract documentation using Doxygen (this is not compatible with added `(precondition)`, `(postcondition)`, and `(body)` because Doxygen preprocessor is not capable to handle Boost.Preprocessor sequences).
10. Rewritten entire documentation (now using Boost.QuickBook instead of Doxygen).

Release files and documentation.

Release 0.2.190

November 21, 2009

Notes:

1. Compiled using both GCC (Linux and Cygwin) and MSVC (Windows XP).
2. Required to use void to specify empty function argument list. This is to comply with C++03 standard that does not allow to pass empty macro parameters so it does not support empty preprocessor sequences `()`.

Release files and documentation.

Release 0.1.126

June 17, 2009

Notes:

1. Completed first documentation draft.

Release files and documentation.

Release 0.1.55

April 19, 2009

Notes:

1. Reorganized files to cleanup root directory.
2. Added installation program.

Release files and documentation.

Release 0.1.50

April 19, 2009

Notes:

1. First public release.

Release files and documentation.

Bibliography

[Bright04] W. Bright. *Contract Programming for the D Programming Language*. 2004.

[Bright04b] W. Bright. *Contract Programming for the Digital Mars C++ Compiler*. 2004.

[C2] Aechmea. *C^2 Contract Programming add-on for C++*. 2005.

[Chrome02] RemObjects. *Chrome: Contract Programming for Object Pascal in .NET*. 2002.

[Clarke06] L. A. Clarke and D. S. Rosenblum. *A Historical Perspective on Runtime Assertion Checking in Software Development*. Newsletter ACM SIGSOFT Software Engineering Notes, 2006.

[Cline90] M. Cline and D. Lea. *The Behaviour of C++ Classes* and *Using Annotated C++*. Proc. of the Symposium on Object Oriented Programming Emphasizing Practical Applications, Maris College, 1990.

[Ellis90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. ANSI Base Document, Addison Wesley, 1990.

[Gautron92] P. Gautron. *An Assertion Mechanism Based on Exceptions*. Fourth C++ Technical Conference, 1992.

[Hoare73] C. A. R. Hoare. *Hints on Programming Language Design*. Stanford University Artificial Intelligence memo AIM-224/STAN-CS-73-403, pages 193-216, 1973.

[CodeContracts] Microsoft Research. *Code Contracts: Design-By-Contract Programming for All .NET Programming Languages*. 2012.

[iContract] O. Enseling. *iContract: Contract Programming for Java*. 2001.

[Jcontract] Parasoft. *Jcontract: Contract Programming for Java*.

[Lindrud04] J. Lindrud. *Design by Contract in C++*. 2004.

[Maley99] D. Maley and I. Spence. *Emulating Design by Contract in C++*. Proceedings of TOOLS, IEEE Computer Society, 1999.

[Meyer97] B. Meyer. *Object Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[Mitchell02] R. Mitchell and J. McKim. *Design by Contract, by Example*. Addison-Wesley, 2002.

[N1613] T. Ottosen. *Proposal to add Design by Contract to C++*. The C++ Standards Committee, N1613, 2004.

[N1653] C. Nelson. *Working draft changes for C99 preprocessor synchronization*. C++ Standards Committee, N1653, 2004.

[N1669] T. Ottosen. *Proposal to add Contract Programming to C++ (revision 1)*. The C++ Standards Committee, N1669, 2004.

[N1773] D. Abrahams, L. Crowl, T. Ottosen, and J. Widman. *Proposal to add Contract Programming to C++ (revision 2)*. The C++ Standards Committee, N1773, 2005.

[N1866] L. Crowl and T. Ottosen. *Proposal to add Contract Programming to C++ (revision 3)*. The C++ Standards Committee, N1866, 2005.

[N1895] H. Sutter and F. Glassborow. *Delegating Constructors (revision 2)*. C++ Standards Committee, N1895, 2005.

[N1962] L. Crowl and T. Ottosen. *Proposal to add Contract Programming to C++ (revision 4)*. The C++ Standards Committee, N1962, 2006.

[N2081] D. Gregor, B. Stroustrup. *Concepts (revision 1)*. The C++ Standards Committee, N2081, 2006.

[N2914] P. Becker. *Working Draft, Standard for Programming Language C++*. The C++ Standards Committee, N2914, 2009.

[N2906] B. Stroustrup. *Simplifying the sue of concepts*. The C++ Standards Committee, N2906, 2009.

[Rosenblum95] D. S. Rosenblum. *A practical Approach to Programming With Assertions*. IEEE Transactions on Software Engineering, 1995.

[SPARKAda] Praxis. *SPARKAda (Ada-like Language with Contract Programming)*.

[SpecSharp] Microsoft. *Spec# (C# Extension)*.

[Stroustrup94] B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[Stroustrup97] B. Stroustrup. *The C++ Programming Language*. Prentice-Hall, 2nd Edition, 1997.

[Tandin04] A. Tandin. *Design by Contract macros for C++ and link to Doxygen*. 2004.

[Wilson06] M. Wilson. *Contract Programming 101 - The Nuclear Reactor and the Deep Space Probe*. The C++ Source, 2006.

Acknowledgments

This section aims to recognize the contributions of *all* the different people that participated directly or indirectly to the design and development of this library.

Sincere thanks to my parents for their support with my education and my studies in computer science.

Sincere thanks to Marina for her kindness and continuous support.

Many thanks to Bertrand Meyer for his pioneering and thorough work on Contract Programming in [Meyer97].

Many thanks to Thorsten Ottosen for his work with [N1962] (and previous revisions) and for clarifying the [N1962] requirements directly with the library authors when needed.

Many thanks to Andrzej Krzemienski for reviewing earlier versions of this library providing valuable insights, for exchanging ideas on implementing assertion requirements, and for suggesting to support named parameters.

Many thanks to Vicente J. Botet Escriba for reviewing earlier versions of this library providing valuable insights and for suggesting to sue `contract::copy`.

Thanks to Steven Watanabe for providing valuable insights on C++ and for hinting to use template meta-programming introspection to detect if a base class has a given member function (technique which turned out to be essential to fully automate subcontracting).

Thanks to Dave Abrahams for providing valuable comments on the library syntax and especially on the syntax to support named parameters.

Thanks to David Maley for having shared source code form his inspiring work in [Maley99] on emulating Contract Programming in C++.

Finally, many thanks to the entire Boost community and mailing list for providing valuable comments about this library and great insights on the C++ programming language.