Doc. no. P0380
Date: 2016-05-28
Project: Programming Language C++
Audience: Evolution Working Group
Reply to: Bjarne Stroustrup (bs@ms.com)

A Contract Design

G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup

Abstract

This paper describes a design of a powerful, minimal facility for specifying contracts (preconditions, postconditions, and assertions) for C++. It covers a wide range of needs and is implementable using conventional techniques. It is a refinement of practical experience with several systems for specifying requirements for code.

The design has been influenced by numerous papers (see the reference list) and discussions. It is essentially what J. Daniel Garcia presented at the February 2016 Jacksonville WG21 meeting.

This paper describes the contract design primarily from a user's point of view with a few hints of potential implementation techniques. The proposed standards wording is in a companion paper (TBD).

1. Introduction

Specifying contracts (in particular, preconditions, postconditions, and assertions) has proven a valuable tool for improving software quality. The literature and industrial practice are littered with contract systems, demonstrating their utility. C++ needs a standard to allow contracts to be used across organizations; in particular, to allow libraries using contracts to be widely used.

The design presented here is the result of joint work by many (see the author list) and relies on inputs from many more (see the reference list). It is complete and minimal in the sense that it includes all facilities that the authors consider essential, but leaves out many that we consider merely useful or potentially useful. We can always add features later if need be.

In this paper, we aim for clarity, picking one solution to each "bike shed issue", such as "what is the word used to denote the requirements for a set of argument values?" See the "Naming" section for a very brief discussion of alternatives.

A programmer can express

- A *precondition*: a predicate that is supposed to hold upon entry into a function – placed outside a function body. This expresses a function's expectation of its arguments and/or the state of objects used by the function.
- An *assertion*: a predicate that is supposed to hold at its point in the computation – placed anywhere a statement can appear in a function body.
- A *postcondition*: a predicate that is supposed to hold upon exit from the function – placed outside a function body. This expresses what the function is supposed to ensure for the return value and/or the state of objects used by the function.

```
void push(queue & q)
        [[ expects: !q. full () ]]      // there must be room for another element
        [[ ensures: !q.empty() ]]       // q can't be empty after adding an element
{
        // ...
        [[ assert: q.is_ok() ]];        // q's invariant is (re)established at this point
}
```

Preconditions ("expects"), postconditions ("ensures"), and assertions ("assert") are collectively called contracts. A contract has no observable effect in a correct program (i.e., a program without contract violations) beyond performance differences.

A contract is checked (or not) at run time depending on a build system setting. The default action upon a contract violation is program termination.

We use the attribute syntax to indicate that contracts are not always checked at run-time (see section 3) and that they are not part of a function's type. Support for contract attributes is *not* optional in the sense that they can be ignored by a conforming compiler; contracts are enforced as described in sections 2 and 3.

Preconditions and postconditions are expressed outside the function body, so they cannot access local variables in the function (except the arguments) or private or protected members of the class of a member function. They operate on the external view of a function; they are considered part of a function's interface. Assertions are internal to the body of a function and can access everything the function can access from their point in the function; they are considered part of the implementation of a function.

```
class X {
public:
        void f(int n)
                [[ expects: n<m ]]      // error: precondition cannot access private member m
        {
                [[ assert: n<m ]];      // OK: we are inside a member function
                // …
        }
private:
        int m;
};
```

No contract expression is permitted to modify any object, but this constraint is not required to be checked by the compiler.

A contract on a **constexpr** function cannot refer to non-local *variables*. It can refer to a non-local *constant* with a value known at compile time:

```
int min = -27;
constexpr int max = 27;

constexpr int nonsense(int x)
        [[expects: min<=x]]     // error
        [[expects: x<max]]      // OK
{
        // …
}
```

We consider preconditions, postconditions, and assertions essential (that is, in the union of what was deemed essential by the participants in the discussions). Further elaboration may be desirable (e.g., class invariants and loop invariants), but more can be added later if widespread need is demonstrated.

2. Enforcement

By default, a contract violation causes program termination. However, a programmer can control what is checked and also what is done in response to a violation (section 3).

There are three "levels" of contracts:

- *Default*: the default; the cost of run-time checking is assumed to be small (or at least not expensive) compared to the cost of executing the function
- *Audit*: a check that is more comprehensive than the default; the cost of run-time checking is assumed to be large (or at least significant) compared to executing the function
- *Axiom*: a formal comment for humans and static analyzers; no run-time cost because the predicate is not evaluated at run time

For example:

```
[[expects: … ]]          // implicitly default

[[expects default: … ]]  // explicitly default

[[expects axiom: … ]]    // no run-time checking

[[expects audit: … ]]    // this check is expensive
```

Why three levels? We could have more (for example, add a **min** level for checks that are even cheaper than the default for contracts where run-time checking is considered essential even in extremely performance sensitive code) or less (do we really need **axiom**?). These three levels are the union of what people consider essential. More levels would simply be "nice or convenient". We can add more if a widespread need is demonstrated. The **axiom** level was deemed essential by people wanting more precise comments and/or using contract systems augmented by static analysis.

Note that the predicate of an axiom is not evaluated and may contain calls to undefined functions. For example:

```
template<class InputIterator>
Iter algo(InputIterator fst, InputIterator lst)
  [[expects axiom: fst!=lst && reachable(fst,lst)]]
{
  // ...
}
```

Here, the contract is used to state that [**fst**:**lst**) is a sequence, rather than two unrelated values. We chose **InputIterator** because that makes **reachable(fst,lst)** impossible to execute as a side-effect-free contract predicate.

3. Handling contract violations

A compilation has a *buid level*. If the build level is

- **off**, no contracts are checked
- **default**, default contracts are checked
- **audit**, default and audit contracts are checked

Exactly how the build level is set is implementation specific (and not specified by the standard). You could think of it as a compiler option, but a build system will most likely ensure consistent use of such an option. By default, the build level is **default**.

A compilation has a *violation handler*. If a contract violation is detected, the violation handler is invoked.

A user can set a violation handler. Exactly how the violation handler is set is implementation specific (and not specified by the standard). You could think of it as a compiler option, but a build system will most likely ensure consistent use of such an option

The type of the violation handler is

```
void(const violation_info&)
```

where a **violation_info** has at least the following members

- **int line_number;** **//** *-1 means unknown*

- **const char* file_name;** **//** *nullptr means unknown*
- **const char* function_name;** **//** *nullptr means unknown*
- **const char* comment;** **//** *the text of the contract*

So **[[assert default: x<0]]** might **pass { 10234, "foo.cpp", "foo", "assert default: x<0"}**.

By default, the violation handler immediately invokes **std::abort**.

Setting a violation handler can be a very sensitive issue so some implementations will put severe restrictions on the handlers that can be installed and how. For example, a violation handler could be used to inject a Trojan horse into a system, so security-sensitive systems will prevent that (e.g., by allowing installation of approved violation handlers only.

There are programs that need to resume computation after executing a violations handler. This seems counterintuitive ("continue after a precondition violation!!!?"), but there are two important use cases:

- Gradual introduction of contracts: Experience shows that once you start introducing contracts into a large old code base, violations are found in "correctly working code." In other words, contracts are violated in ways that did not cause crashes for the actual use of the code for the actual data used. There are examples where the number of such "currently harmless violations" is massive. The way to cope is to install a violation handler that logs the problem and continues. This allows gradual adoption.
- Test harnesses: Contracts are code, so they can contain bugs. To test contracts, we need to execute examples of violations. A convenient way of organizing such a test suite is to have a violation handler throw an exception and have the main testing loop catch exceptions and proceed running the next test.

A compilation has a *continuation option*, set to **on** or **off**. If the continuation option is set to **on**, the execution will resume after the violation handler exits.

Exactly how the continuation option is set is implementation specific (and not specified by the standard). You could think of it as a compiler option, but a build system will most likely ensure consistent use of such an option. By default, the continuation option is **off**.

Note that continuing after a violation is technically undefined behavior. For example, after **[[assert: p!=nullptr]]** a compiler may assume that **p** is not the **nullptr** and may proceed to eliminate subsequent **p!=nullptr** tests. This is not hypothetical: modern aggressive optimizers perform such elisions. Fortunately, all optimizers that do such optimizations also have an option for suppressing them for critical code, such as kernel code. Setting the continuation option will imply the use of such (implementation specific) compiler/optimizer options.

To sum up, a build system will offer three implementation specific settings:

- Build level (to be matched against contracts' checking levels)
- Violation handler (to be invoke in case of contract violation)

- Continuation option (to indicate whether a computation can be resumed after a return from a violation handler)

The details of these options are beyond the C++ standard.

There is (deliberately) no way of setting these options in the source code. If there were, complexity and opportunities for errors and security violations would arise.

You cannot exit a **noexcept** function by throwing an exception from a violation handler; attempting to do so causes termination exactly as if an exception had been thrown from inside the **noexcept** function. For example:

```
int f(int x) noexcept [[expects: 0<x]];

int y = f(-1);      // terminate
```

This is true even if a violation handler that throws has been installed. That is, you cannot get to a **catch**-clause outside a **noexcept** function by throwing from a violation handler.

4. Declarations and contracts

A contract can be placed on declarations of functions and pointers to functions. This includes declarations of virtual functions, template functions, and lambdas.

The contracts every declaration of a function must be (ODR) identical. For example:

```
struct B {
        virtual void f(int x);
        virtual void g(int x) [[expects: x<0]];
        virtual void h(int x) [[expects: x>0]];
        virtual void k(int x) [[expects: 0<=x]];
};

Struct D : B {
        void f(int x) override [[expects: x<0]];    // error: no contract on B::f
        void g(int x) override [[expects: x<1]];    // error: contract differ from B::g's
        void h(int x) override;                     // error: there  was a contract on B::h
        void k(int x) override [[expects: 0<=x]]; // OK: identical contracts
};
```

This rule ensures that for a given function the same contract is enforced for every invocation.

Obviously, we could relax this rule. For example, we could allow a contract to be on only one declaration (and check the contract only if that declaration was used for a class or for all calls) or allow weakening preconditions in an overriding function. We decided that the added complexity wasn't worth it. If serious need is demonstrated, we can relax this rule later.

Note that you can achieve the effect of a stronger contract on a virtual function than on its overrider by using an assertion:

```
class A {
        // ...
        void f () [[expects: p1()]]
        {
                [[assert: p2()]];
                // ...
        }
};

class B : public A {
        // ...
        void f () [[expects: p1()]];
};
```

This requires a bit of foresight (to use a suitably weak precondition).

Another example:

```
void f(int x) [[expects: x>0]];
void (*pf) ( int ) = &f;                        // error: would remove contract
void (*pf) ( int ) [[expects: x>0]] = &f;       // OK
void (*pf) ( int ) [[expects: x>=0] = &f;       // error: different contract

void g(int x) ;
void (*pg)(int ) [[expects: x>0]] = &g;         // error: would add contract
```

Enforcing identical contracts for pointers to functions might prove to be overly constraining, so this design choice may have to be reviewed in light of future experience. It is, however, the simplest design that doesn't imply the possibility of an undetected contract violation.

A declaration can have several contracts. For example:

```
template<RandomAccessIterator Iter>
void bsearch(Iter p, Iter q)
        [[expects default: p<=q]]
        [[expects audit: is_sorted(p,q)]]
{
        // …
}
```

Importantly, a compiler may not use information from a contract that is not checked at run time to suppress run-time checks of otherwise checked contracts. In this case, a compiler "understanding" **is_sorted(p,q)** may ***not*** use than knowledge to suppress the (weaker, but cheaper) **p<=q** check when the checking level is **default**. The programmer asked for that check and should get it.

Checking that a sequence is sorted is typically much more expensive (**O(N)**) than doing a binary search (**O(log N)**). This kind of check is often better done using an **assert**:

```
template<RandomAccessIterator Iter>
void bsearch(Iter p, Iter q)
        [[expects: p<=q]]
{
        If (p==q) return;
        Iter mid;
        // …
        [[assert: *mid<=*(mid+1)]];      // where we pick a new mid
        // …
}
```

5. Notation and naming

As usual, naming is difficult and potentially controversial. In the sections above we have used our preferred choices. This section very briefly outlines alternatives.

Contract names:

- Precondition: **expects**, precondition, assumes, pre, expect, require
- Postcondition: **ensures**, postcondition, post, commitment, promise, ensure
- Assertion: **assert**, assertion, invariant

People will have to learn what the exact meaning is independently of what names we use, so “more natural” or “more informative” names (according some individual’s experience) could even be misleading (“false friends”). We consider “perfect naming” impossible.

Checking level names:

- No runtime checking: **axiom**, not-runtime, static, compile time, comment, annotation
- Cheap/standard checking: **default**, cheap, normal
- Expensive/comprehensive checking: **audit**, expensive, max, comprehensive, exhaustive

We choose something short and again we assume that people will have to learn the exact meaning whatever names we use, so that “better names” could become “false friends” when people more often thought they understood them without looking up their meaning.

We considered having fewer alternatives and more alternatives. We might have eliminated **axiom**, but there were strong support (insistence) for that from people who wanted more formal comments and for people with contract systems relying of static analysis support. We might have added **min** (or **critical**) for run-time checking that was cheaper than **default** and/or **medium** for run-time checking that was more expensive than **default** and cheaper than **audit**, but every new checking level opens opportunities for confusion (incompatible use in different places) and we can always add a level later if need be.

We use the **[[expects: …]]** notation rather than **[[expects(…)]]**. The latter became quite tedious to write, is harder to read, and **expects**, **ensures**, and **assert** really are not functions. When combined with checking levels, it became uglier still. We know that aesthetics is not objective, but this is our consensus. For **assert**, **[[assert(p)]]** would clash with the assert macro.

We consider the kind of contract the most significant part so we place the checking level after the kind of contract:

```
[[assert audit: x.ok()&&y<99]]
```

We could equally well have chosen the opposite order:

```
[[audit assert: x.ok()&&y<99]]
```

We could even had left the order unspecified, but that would have left opportunities for gratuitous style differences and confusion.

It is not uncommon to want to refer to the return value in a postcondition. For example:

```
int area(int h, int w) [[ensures: 0<=return_value]]      // How do we say something like this?
{
        // …
}
```

What would be proper notation for **return_value**? Suggestions include **return**, the function name, **return_value**, **`return**, **-> ret**, and various other special-purpose syntactic constructs. Consider:

- We don't want to have to write a whole new expression parser for postconditions (and the reference to the returned value will occur in expressions, potentially deep in complicated expressions)
- We don't want to use a common identifier that might clash with the user's choice of identifiers
- The function's name can be use with its ordinary meaning (e.g., in a recursive call)

We were tempted to leave out the ability to refer to the returned value, but the need is common in contract systems and some consider it essential.

To refer to the return value, we introduce a name for it after **ensures.** For example:

```
int area(int h, int w) [[ensures ret_val: 0<=ret_val]]
{
        // …
}
```

If we refer to an argument in a postcondition, the value used (if the postcondition is evaluated) is the value of the argument at the point of return. For example:

```
void incr(int&r)
        [[expects: 0<r]]
```

```
        [[ensures: 1<r]]
{
        ++r;
}
```

There is no way of referring to the old/original value of an argument in a postcondition. A suitable notation for that can be added later if a widespread need is demonstrated. Naturally, we know of examples where this would be useful, but deemed it "not essential for a minimal facility." For now, we can always compensate by using an **assert**; for example:

```
void incr(int&r)
        [[expects: 0<r]]
{
        int old = r;
        ++r;
        [[assert: r==old+1]];      // "faking" a postcondition
}
```

Changing the value of an argument passed by value and used in a postcondition is ill-formed. For example:

```
int* find(int* first, int* last, int x)
    [[ensures: first<=last]]
{
    while (first!=last && *first!=x)
        ++first;    // error
    return *first;
}
```

Note that we have to use a semicolon after **[[assert: …]]** in function bodies. That semicolon is logically redundant, but is currently required by the grammar and helps "smart" editors indent properly.

6. Acknowledgements

Many people contributed; see the reference list. Also Pablo Halpern.

7. References

In roughly reverse chronological order:

- Gabriel Dos Reis, J. Daniel García, Francesco Logozzo, Manuel Fähndrich, Shuvendu Lahiri: Simple Contracts for C++ (R1). P0287r0.
- Nathan Myers: Criteria for Contract Support. P0247R0.
- John Lakos, Alisdair Meredith, Nathan Myers: Contract Support Merged Proposal. P0246R0.
- J. Daniel García: Three interesting questions about contracts. P0166r0.
- Lawrence Crowl: The Use and Implementation of Contracts. P0147r0.

- Walter Brown: Proposing Contract Attributes. N4435.
- Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, Manuel Fahndrich, Shuvendu Lahri: Simple Contracts for C++. N4415.
- John Lakos, Nathan Myers, Alexei Zakharov, Alexander Beels: Language Support for Contract Assertions (Revision 10). N4378.
- Gabriel Dos Reis, Shuvendu Lahiri, Francesco Logozzo, Thomas Ball, Jared Parsons: Contracts for C++: What Are the Choices? N4319.
- John Lakos, Nathan Myers: FAQ about Contract Assertions. N4379.
- Nathan Myers: Library support for runtime contract violations. N4289.
- Nathan Myers: Language support for optional contract violations. N4290.
- Nathan Myers: Language support for block scope assertions. N4291.
- Nathan Myers: Language support for call-site assertions. N4292.
- J. Daniel Garcia: C++ language support for contract programming. N4293.
- J. Lakos, A. Zakharov, A. Beels, N. Myers: Language Support for Runtime Contract Validation (Revision 8). N4135.
- J. Daniel Garcia: Exploring the design space of contract specifications for C++. N4110.
- J. Lakos, A. Zakharov, A. Beels: Centralized Defensive-Programming Support for Narrow Contracts (Revision 5). N4075.
- J. Lakos, A. Zakharov: Centralized Defensive-Programming Support for Narrow Contracts (Revision 4). N3963.
- J. Lakos, A. Zakharov: Centralized Defensive-Programming Support for Narrow Contracts (Revision 3). N3877.
- J. Lakos, A. Zakharov: Centralized Defensive-Programming Support for Narrow Contracts. N3604.

We have left a few intermediate versions of proposals out.