Doc. no. N1613=04-0053
Date: March 29th, 2004
Reply-To: Thorsten Ottosen, nesotto@cs.auc.dk or tottosen@dezide.com

Proposal to add Design by Contract to C++

> **Fact 31:** Error removal is the most time-consuming phase of the [software] life cycle.
> **Fact 36:** Programmer-created built-in debug code, preferably optionally included in the object code based on compiler parameters, is an important supplement to testing tools.

Excerpted from Robert L. Glass' fascinating book *Facts and Fallacies of Software Engineering* [Gla03].

Contents

1 Motivation

This proposal is about making it easier to produce correct software. It is about making code self-documenting and about minimizing separation of code and documentation. And it is about providing stronger support for debugging. I do not claim that Design by Contract (DbC) is a panacea, but rather that it is a generally useful concept. (Remark: some do see it as a panacea [JM04], but that is an exaggeration [Gar98] [Gla03, 137f].)

During his discussion of Fact 36, Robert Glass writes [Gla03, 90]:

> But what is really important about this phase is that it takes longer to do error removal, for most software products, than it does to gather the requirements or do the design or code the solution—just about twice as long, as a matter of fact.

Since error removal is so time-consuming, we should ask ourselves what the programming language can do to improve the situation. The answer is that it can provide programmers with consistent tools and methodologies that enables and encourages them to write code with fewer errors.

As the second quote suggests, we are not interested in replacing testing—we are interested in *supplementing* it. As Kevlin Henney has said, DbC is used to specify functional contracts and not operational contracts [Hen03]. Sometimes it can be practically impossible to write a unit test, and other times it can be impractical or impossible to write a contract. Therefore the D programming language comes with builtin support for both DbC and unit testing [Bri04a].

The Design and Evolution of C++ mentions previous suggestions to include similar facilities. The conclusion is [Str94, 398]:

> The ease with which assertions and invariants can be defined and used within the existing C++ language has minimized the clamor for extensions that specifically support program verification features.

My personal experience is (1) that many C++ programmers like the idea of DbC, (2) that it is possible to get some of the benefits (but far from all) without language support, and (3) that most C++ programmers abandon DbC again because it is too inelegant, relies too much on macros, and is too weak without language support.

DbC is built into the Eiffel and the D programming language (see eg. [Bez99], [Mey97] and [Bri04a]) and the Digital Mars C++ compiler has it as an extension [Bri04b]. There are tools for different languages that try to emulate DbC (see eg. [Ens01], [Par04], [T99] and [Eve04]).

2 What is Design by Contract?

DbC is characterized by at least four simple assertion mechanisms. While I introduce each mechanism, I will give an example written in the syntax that I propose for C++. Later I will explain in depth why that design is suggested (see section 5 on page 11).

1. It is possible to describe **function preconditions**, that is, logical conditions that the implementer of the function expects to be true when the function is called. (The typical use is to check constraints on function parameters.) Example:

```
double sqrt( double r )
in // start a block with preconditions
{
    r > 0.0: throw bad_input();
}
do // normal function body
{ ... }
```

   The precondition is read as "if $r > 0.0$, continue; otherwise throw an exception".

2. It is possible to describe **function postconditions**, that is, logical conditions that the implementer of the function expects to be true when the function has ended normally. (The typical use is to validate the return value and any side-effects that the function has.) Example:

```
int foo( int& i )
out // start block with postconditions
{
    i == in i + 1;
    return % 2 == 0;
}
do // normal function body
{
   i++;
   return 4;
}
```

   The idea is that `in i` means the value of the expression `i` *before* the function body is evaluated. `return` is a special variable that equals the result of the function if it exits normally. (Remark: in Eiffel `i == in i + 1` would be written `i = old i + 1`.)

3. It is possible to describe **class invariants**, that is, logical conditions that the implementer of the class expects to be true after the constructor has executed successfully and before and after any execution of a public member function. (The typical use is to define a valid state for all objects of a class.) Example:

```
class container
{
    // ...
    invariant
    {
        size() >= 0;
        size() <= max_size();
        is_empty() implies size() == 0;
        is_full() implies size() == max_size();
        distance( begin(), end() ) == size();
     }
};
```

The last assertion is an example of a constraint that is infeasible even in debug code if random-access iterators are not supported (linear complexity or worse is unsuitable in contracts [Hen03]). Invariants are inherited and can never be weaker in a derived class.

4. It is possible to formalize the notion of overriding a virtual function; the idea can be justified by substitution principles and is referred to as **subcontracting**. Subcontracting consists of two simple rules that the overriding function must obey [Mey97]. (1) the precondition cannot be stronger, and (2) the postcondition cannot be weaker. Example:

```
void Base::foo( int i ) in { i > 5; } do { ... }
void Derived::foo( int i )            do { ... }
Base& b = *new Derived;
b.foo( 5 ); // will trigger assertion
```

Thus the compiler will automatically OR preconditions from the base class function with the preconditions of the overriding function and it will automatically AND postconditions from the base class function with the postconditions of the overriding function.

Once a programmer has specified contracts, the language provides a mechanism for checking whether the conditions hold at run-time and a mechanism to turn off this run-time check for the sake of efficiency. (Remark: Eiffel also has the notion of loop invariants and monitoring of expression that must change within each iterations of a loop [Eng01a].)

Notice that I distinguish between pre- and postconditions and invariants. This is important when talking about member functions since eg. the effective preconditions consist of both the assertions in the `in`-block *and* the invariant. So when I mention pre- and postconditions, I never include the invariant.

3 What problems can Design by Contract address?

3.1 It can provide a consistent documentation framework

Let's face it: writing good documentation is hard work. So much work that in fact many programmers consider the problems bigger than the benefits and hence decide not to write it [Gla03, 120ff]. The problems that a programmer faces with documentation are:

1. it is hard to give a precise description of what a function requires and delivers,
2. it is hard to give a precise description of the class invariant (including dependencies between public member functions), and
3. it is hard to keep the documentation synchronized with the code and therefore it is likely that inconsistency will emerge.

The fact that writing documentation involves a considerable amount of redundancy means it becomes a boring and repetitive task. At least one methodology therefore suggest not to write documentation (XP). Writing no documentation can actually be a reasonable choice; as Bertrand Meyer said: "if there is any situation worse than having no documentation, it must be having wrong documentation" [Mey97]. Indeed, the documentation is often discarded and one starts from scratch by reading the code itself [Gla03, 120ff].

But how can DbC improve this situation? (Remark: I intentionally do not say "how can DbC *solve*...". Many programming issues are inherently complex and no single scheme can solve them entirely.)

The first problem is addressed by writing pre- and postconditions. In particular the postconditions are hard to express in C++. DbC provides constructs that enable the programmer to compare values of an expression before and after a function is called, and it provides a way to compare the function return value with the expected return value. These constructs are powerful enough to enable the programmer to express many constraints in the code—constraints he would otherwise have to document informally.

The second problem is addressed by embedding run-time checked invariants in the code. The invariant mechanism makes it largely possible to describe what to expect from a class.

Because these facilities allow us to embed constraints directly in the code and execute and verify them at runtime, they are always up to date and can be trusted. Tools can then extract functions and classes with their contracts to serve as documentation.

3.2 It can provide a consistent basis for debugging and testing

DbC can provide a powerful debugging facility because of its ability to instrument code:

1. In a language with multiple return statements, it is convenient to have a way to make an assertion about the return value. In particular, this is convenient if a unit test is impractical.

2. Virtual functions automatically inherit contracts so the programmer does not need to specify them once again when overriding a function.

3. Class invariants will be automatically called last in the constructor, before and after each call to a public function and first in the destructor (see section 5.3 on page 14 for details).

This is a way of calling assertions that is not possible today. Moreover, there is also the importance of consistency. Quite a few different assertions mechanisms exist, see eg. the Gnu Nana library [Mak04], the Smart Assert mechanism [Ale03] [AT03], Enforcements [AM03], a new assertion class [Gue01], and EventHelix' DbC library [Eve04].

DbC complements testing because a contract will specify what the test should check. For example, the preconditions of a function will state which input that should cause the function to throw exceptions (or simply fail) and which input that will cause the function to exit normally. (Remark: in the future it might even be possible to use the contracts to automate test generation [Mad02].)

3.3 It can provide a consistent specification language

Some software projects (especially safety-critical projects) tend to have a strict labour division between design and coding. The problem is that it is not the same person that does both tasks and therefore misunderstandings can occur [Gla03, 84ff]. To reduce the gap between designer and programmer, it might help to have a precise and unambiguous specification language. Moreover, contracts could also make it easier to do code-reviews.

3.4 It can enable the compiler to generate better code

Compiler writers should be allowed to assume that the precondition of a function holds even when the run-time check is left out. The precondition can therefore be used as a basis for generating more efficient code. As an example, consider a switch statement:

```
void foo( int i )
in
{
    i == 1 || i == 2;
}
do
{
    switch( i )
    { ... }
}
```

Another interesting possibility is to describe the *absence* of aliasing. C99 introduces a new keyword `restrict` which can be used to declare pointers and dynamic arrays that do not overlap [C9903, 77ff]. This can be crucial to performance. Even though C++ has an advantage compared to C when it comes to aliasing analysis [Mit00], it might still be worth adding `restrict` to the language. However, it would be even better to avoid a new keyword and use assertions to specify aliasing properties.

To state that two objects cannot overlap can be done as simply as

```
void bar( Foo* l, Foo* r )
in
{
   l != r;
}
```

And to state that two arrays do not overlap might be done as

```
template< typename T >
inline bool not_overlapping( const T* l, size_t l_length,
                             const T* r, size_t r_length )
{
    return l + l_length < r || r + r_length < l;
}
// ...
void bar( Foo* l, size_t l_length, Foo* r, size_t r_length )
in
{
    not_overlapping( l, l_length, r, r_length );
}
do
{
    for( size_t i = 0; i < l_length; ++i )
      // access arrays somehow
}
```

Of course, it requires extra work from the optimizer to recognize constructs such as `not_overlapping()` and verify that the function body obeys that requirement. (Remark: paragraph 5.9 of the standard could prohibit portable use of this feature unless the constraint can be expressed differently [ISO98].)

3.5 It can make inheritance easier to use

DbC and inheritance it not just about formalizing the overriding mechanism; it is also about making it *easier* to use correctly and to keep the base class programmer in control.

C++ already supports an idiom that gives us some of the benefits of subcontracting: the Non-Virtual Interface Idiom as described by Herb Sutter [Sut04]. James Kanze has been advocating the idiom on newsgroups for some time (see eg. [NG01]):

```
class X
{
   virtual int doFoo( int i ) = 0;

public:

  int foo( int i )
  {
    assert( i > 0 );
    int result = doFoo( i );
    assert( result > 0 );
    return result;
  }
};
```

The purpose of the public function `foo()` is to enforce a contract upon the use of `doFoo()`. Since all calls to `doFoo()` must pass through `foo()`, it can also be used to manually instrument the code during debug sessions. What is particular good about it is that the base class stays in control; the derived classes cannot escape the contract enforced by `foo()`. What is not so good is that it is a kind of a hack to have two functions each time one virtual function is needed.

DbC would allow programmers to only have the virtual function and to make it public while still leaving the base class in control. Moreover, to support updating of contracts using the Non-Virtual Interface idiom, one would have to override `foo()` and let the client use a pointer to this derived class. Whether this is a good idea is doubtful [Mey98, Item 37].

Has subcontracting actually any use in practice? According to one professional Eiffel programmer I spoke with, Berend de Boer, he has not used the possibility to loosen preconditions, but he has used the ability to provide stronger postconditions regularly. As an example, the Gobo Eiffel Project consists of approximately 140k lines of code (including comments) and contains 219 stronger postconditions and 109 weaker preconditions [Bez03].

3.6 It can make it harder for errors to escape

As an example consider the validation of the return value in the postconditions. Validating the return value might seem redundant, but in this case

we actually want that redundancy. When a programmer writes a function, there is a certain probability that he makes an error in a function. When he specifies the result of the function in the postconditions, there is also a certain probability that he makes the error again. However, the probability that he makes the same error twice is *lower* than the probability that he makes it once.

A more rigid explanation follows. Let us consider two events:

- $A =$ the programmer makes error X in the body of function `foo()`.
- $B =$ the programmer makes error X in the postconditions of function `foo()`.

And let us further assume $0 < P(A) = P(B) = p < 1$. The question is now which probability is smaller: $P(A)$ or $P(A \text{ and } B)$? We can use the following formulas to reach a conclusion [SSS00, 7f]:

1. $P(A \text{ and } B) = P(A)P(B)$ if A and B are independent,
2. $P(A \text{ and } B) = P(B|A)P(A)$ if A and B are dependent.

If we assume A and B are independent, then $P(A \text{ and } B) = p^2 < p$ since $p < 1$. If we assume A and B are *totally* dependent (A always implies B), then $P(B|A) = 1$ and we conclude $P(A \text{ and } B) = P(A)$. However, it would be unrealistic to assume that the two events are totally independent, and it would also be unrealistic to assume that they are totally dependent. Therefore $P(A \text{ and } B) < P(A)$ in general and more errors will be caught.

4 How is Design by Contract done in other languages?

To study how other languages has implemented DbC can be informative and might reveal weak as well as strong design decisions. A comparison between Eiffel, D and C++ can be found in table 1 on page 16.

4.1 Design by Contract in Eiffel

Eiffel was one of the first general purpose languages to incorporate DbC more than a decade ago [Mey97]. The keywords involved are `ensure`, `require`, `do`, `invariant`, `result`, `old`, `require else`, and `ensure then`. When a contract is broken, an exception is thrown. (Remark: an exception in Eiffel behaves largely as a C++ exception.) If some of the contracts are left in the final release version of the program, exceptions will still be thrown.

Pre- and postconditions look like this [Eng01b] (note = means comparison):

```
put( x: ELEMENT; key: STRING ) is
    require
           count <= capacity
           not key.empty
    do
           ... Some insertion algorithm ...
    ensure
           has( x )
           item( key ) = x
           count = old count + 1
   end
```

If the function has a return value, then it may be checked in postconditions using the `result` keyword. Invariants are expressed as [Eng01b]:

```
class ACCOUNT
invariant
    consistent_balance: balance = all_deposits.total
    ... rest of class ...
end
```

This also illustrates how an assertion might be given a label: `consistent_balance` is the comment of that particular assertion.

Subcontracting is supported by `require else` and `ensure then` and it is an error if an overriding function has contracts without using these two statements.

4.2 Design by Contract in D

Walter Bright has incorporated DbC in his C++ style language D (and DbC in his C++ compiler adheres to the same approach [Mar03]). He has taken a bit different approach than Eiffel. When a contract is broken in D, an exception is thrown to report the bug. (Remark: in D exceptions behave like in C++.)

The feature makes use of the keywords `in`, `out`, `body`, `assert`, and `invariant`. Consider a function with contracts:

```
int add_one( int i )
in                  // start precondition block
{
    assert( i > 0 );
}
out( result )       // start postcondition block
{
    assert( result == i + 1 );
}
body                // start function body
{
    return i + 1;
}
```

The semantics are intuitive and simple: `in` and `out` group all preconditions and postconditions, respectively. In postconditions the syntax indicates that `result` should be an immutable, scoped variable initialized with the return value; note that any identifier name can be used instead of `result`.

A class invariant looks like this:

```
class Date
{
    int day;
    int hour;

    invariant      // start class invariant
    {
       assert( 1 <= day && day <= 31 );
       assert( 0 <= hour && hour < 24 );
    }
}
```

The use of `assert` is neccesary because D allows any statement to appear in the contracts. D also allows static assertions to be specified:

```
static assert( <a compile-time expression> );
```

As an interesting property, the invariant can be checked when a class object is the argument to an `assert()` expression:

```
Date mydate;
...
assert( mydate ); // check that class Date invariant holds
```

It is worth noticing that the contract blocks can contain any statement and/or expression; hence the compiler can therefore not prohibit obvious side-effects.

5 How should Design by Contract be in C++?

In this section I will describe in detail how the syntax and semantics of DbC should be. (Remark: many alternatives are discussed in section 7 on page 18.)

There is one feature which makes DbC more powerful, but which really should be part of the core language anyway and that is an implication operator. Implication is a natural boolean operator and it could be spelled as `=>` or `==>` (to avoid confusion with `>=`), and like we may say `and` instead of `&&`, we should allow `implies` instead of `==>`. The operator should have low precedence and be left-associative. (see also section 6 on page 15 and section 8 on page 21).

There is one global rule about assertions and it is that assertions must be disabled in assertions. This removes the possibility of infinite recursion and allows a reasonable performance even with assertions enabled. Without disabling assertions, infinite recursion will happen when a public function appears in the invariant.

5.1 Preconditions

Preconditions on a function are optional, but when they are used the function body starts with a `do`-block. In D there has been added a new keyword, `body`, which serves the same purpose as `do`. However, `do` seems like a reasonable choice compared to introducing a new keyword. The preconditions can be specified like this:

```
void foo( int i )
in
{
    i > 0;                      // call 'terminate()' on failure
    i > 0: exit( 1 );           // call 'exit()' on failure
    i > 0: throw range_error(); // throw an exception on failure
}
do { ... }
```

The first precondition could be called a **default precondition**. It should be possible to remove such preconditions from object code. However, the second and the third precondition must always be part of the object code. I consider it essential that the programmer can choose which preconditions that are always part of the program flow.

There are some things that are not allowed in the precondition block:

```
int not_ok( int& );
int ok( int );
struct Foo { int foo(); int bar() const; };
Foo  f;
Foo* f_ptr;
...
void foo( int i )
in
{
   not_ok( i );  // error: 'not_ok()' takes a reference argument
   ok( i );      // ok: 'ok()' takes a value argument
   f.foo();      // error: cannot call a non-const member
   f_ptr->foo(); // error: not even through a pointer
   f_ptr->bar(); // ok: bar is a const member function
   f_ptr;        // ok: conversion to bool
   "a comment";  // ok: conversion to bool
   if( ... );    // error: statements not allowed
   i = 2;        // error: assignment not possible
   i > 0; return FAILURE_CODE; // error: 'return' not allowed
}
```

While const member functions can certainly contain side-effects or change mutable data, calling a non-const member function would certainly be wrong; the fact that a C++ implementation can actually make this check is certainly an advantage.

There are several choices for specifying a precondition; `in` is the choice made in D and it is preferable to `precondition` or `require` because of its terseness. It also turns out that `in` may be reused in different contexts (see the next section and section 8 on page 21 for details).

5.2 Postconditions

Postconditions are much like preconditions: (1) they are optional, (2) can include throw clauses (which are never compiled away), and (3) has the same rules regarding const-correctness. Note that postconditions are only checked when the function exits normally.

What is also new in postconditions is that side-effects can be described:

```
int foo( int& i )
out
{
    i == in i + 1;             // keep track of changes to 'i'
    return == 5:  terminate(); // call 'terminate()' on failure
}
do
{
  ++i;
  if( i % 2 == 0 )
      return 5;
  else
      return 4;
}
```

Whenever `in <expression>` occurs in a postcondition it means the value that expression had before the function body was executed. This requires that the result of the expression can be copied. (Remark: an alternative syntax could be to say `new i == i + 1` and thereby reverse the way expressions are evaluated in postconditions. However, this would require that every expression not preceded by `new` to be copied before entering the function.) The `return` acts like an immutable variable with the same type as the return-type of the function. To use `return` instead of a new keyword `result` is also done in iContract [Ens01].

There are also things that cannot be done in postconditions:

```
struct X : noncopyable { int foo(); };
void foo( int& i, X& x )
out
{
    i = 5;                     // error: cannot assign to const object
```

```
    return = 5;                 // error: ditto
    in x == x                   // error: X is not copy-constructible
}
```

For projects that do not use C++ exceptions, but rather relies on error codes, the postconditions can also be used to document that:

```
int insert_element( const Element* e )
out
{
     e == 0 || in full() implies return == EXIT_FAILURE;
     in !full() && e != 0 implies return == EXIT_SUCCESS;
}
do { ... }
```

Arguably this could be rewritten into one postcondition, but such a big postcondition would be harder to comprehend.

5.3 Invariants

The invariant of a class can only be called from within the class itself. It should preferable document how public functions interact and therefore not refer to data members of the class, although that should not be enforced by the compiler. If the programmer wants to check the private state of the class, he can either add assertions about the variables directly or call a private function from the invariant:

```
class X
{
    bool debug_invariant();

    invariant
    {
        // normal public invariant
        debug_invariant();
    }
};
```

Since C++ has the notion of const-member functions, it might be tempting to let the compiler exclude calls to the invariant in these functions. However, the use of logical constness and mutable data suggest that it would be best to call the invariant anyway. Therefore we should call the invariant as the last statement in a constructor and together with preconditions and postconditions of public functions. (Remark: an alternative would be to let the invariant be explicitly callable by the programmer.) So if the invariant throws exceptions, it means that the constructor and any public member function can throw the same exceptions.

Should the invariant be called in the destructor? Preferably "yes" since it can help track down bugs earlier. It does open up the possibility for a destructor to throw exceptions—in those cases the call of the invariant in the destructor should be implemented like this:

```
struct X
{
    ~X()
    {
        try
        {
            invariant();
        }
        catch( ... )
        {
            std::broken_destructor_invariant();
        }
        // normal code
     }
};
```

The function `broken_destructor_invariant()` should be a customizable callback that defaults to calling `terminate()`.

When calls to public member functions are nested, the invariant is not checked before and after the inner call. This is because the invariant is allowed to be temporarily broken within a call to a public function. The other contracts of the functions must still be checked though (see also section 7 on page 18).

6 Could Design by Contract be provided as a library?

Andrei Alexandrescu mentioned the interesting idea that DbC should not be part of the core language [NG03]—we should rather seek to enhance the language such that a strong library implementation was possible. I therefore discuss each feature in turn and explains what would be necessary to emulate it.

1. We cannot detect side-effects in assertions. This could probably be done by introducing some kind of const-block:

   ```
   { // some block
       const { /* no side-effects here */ }
   ```

2. We cannot make assertions about the return value without manually copying the result to a variable. I see no alternative to this mechanism.

Feature	*ISE Eiffel 5.4*	*D*	*C++ proposal*
keywords	`require`, `ensure`, `do` `require else`, `ensure then` `old`, `invariant`, and `result`	`in`, `out`, `body`, `invariant`, and `assert`	`in`, `out`, `do`, `invariant`, and `return`
on failure	throws exception	throws exception	defaults to `terminate()` might throw
return value evaluation	yes, `result` keyword	yes, `result` keyword	yes, `return` keyword
expression copying in postconditions	yes, `old` keyword	no	yes, `in` keyword
subcontracting	yes	yes	yes
assertion naming	yes	no	no
arbitrary code in contracts	yes	yes	no
contracts on abstract func.	yes	no	yes (*)
code ordering	pre -> body -> post	pre -> post -> body	pre -> post -> body
compile-time assertions	no	yes	yes
loop invariants	yes	no	no
const-correct	no	no	yes
removable from object code	not preconditions	yes	only default assertions
invariant calls	end of "constructor", around public functions	end of constructor, around public functions, start of destructor	end of constructor, around public functions start of destructor
disabling of assertions during assertions	yes	no	yes
when public func. call public func.	disable all assertions	disable nothing	disable invariant only

Table 1: Comparison of features in Design by Contract in different languages. (*) Pure virtual functions are already allowed to have a function body.

3. We cannot make assertions about the side-effects of a function without copying the expression(s) manually first. I see no alternative to this mechanism.

4. We cannot inherit assertions in public virtual functions. Here we might add another way to inherit code in virtual functions, for example

```
virtual void foo( int i )
{
    // call this code implicitly in overridden functions
    implicit assert( i > 0 );
```

5. We cannot make an effective `implies` implementation. The first way we could simulate `implies` is via a macro:

```
#define implies != true ? true :
assert( foo() implies bar() );                  // ok
assert( !foo() && !bar() implies something() ); // bad
```

   The scheme fails because `!=` has higher precedence rules than eg. `&&`. We were probably better off if we tried to overload `operator,()`, but we could still not do short-circuit evaluation and hence avoid costly calls to the right hand side if the left argument is false.

6. We cannot call functions implicitly before and/or after each call to a public function. This would be necessary to define an invariant. Interestingly, such a feature has been part of the language before standardization [Str94, 57]:

   > Curiously enough, the initial implementation of C with Classes contained a feature that is not provided by C++, but is often requested. One could define a function that would implicitly be called before every call of every member function (except the constructor) and another that would be implicitly called before every return from functions (except the destructor). They were called `call()` and `return()` functions.

I do not consider my treatment of this complete, however, I hope it can be seen that it is a non-trivial task to come up with a set of features which can support DbC while preserving general usefulness.

7 Discussion and open issues

7.1 Where to put the contracts?

Since C++ distinguishes between the declaration and the definition of a function, it opens up the question of where to put the contract code. Should contracts cling to the declaration, the definition or perhaps both?

Some argue that the contract is part of the interface and therefore must always be part of the declaration. If we ignore implementation issues, there are three alternative solutions:

1. only allow contracts in function declarations,
2. only allow contracts in function definitions, and
3. allow both, but each function chooses individually between 1 and 2.

The last is flexible, but messy. The first would make a header almost self-documenting and remove the need for `do`-blocks, but would remove some of the overview a short declarative style offers and it might trigger more recompilation than is desirable. The second alternative seems to remove some documentation value (but not all), but it preserves the overview and minimizes recompilation. It might also be practical to have the contracts right at the cursor when trying to write the function body. Documentation tools can easily be extended to pick up assertions and it is worth to notice that even Eiffel relies on a tool to compose the final documentation.

7.2 Can we call the invariant less often?

It could also be argued that the invariant should only be checked as a function exits. The invariant in the precondition would automatically hold because of an earlier postcondition. The programmer could be malicious and by-pass the mechanism by using public data, friends, function pointers and templated member functions, but why protect against that? Thus we could reduce the calls to the invariant to a half which improves debug performance. (Remark: checking the invariant more often would detect memory corruption errors earlier, but it is debatable whether this is any better since the error would probably not be due to an error in the class where the invariant failed.)

7.3 Should arbitrary expressions and statements be allowed in contracts?

The feature has probably been allowed in D to make more complex assertions possible:

```
in
{
   if( something() )
       assert( false );
   for( i = 0; ... )
       assert( x[i] > 0 );
}
```

However, the need is not really there: (1) the first can be done using `implies`, and (2) the second can be refactored into a small function that verifies the same and then returns `false` on error. The difference is that we cannot detect the precise spot where the error occurred (at least not without a debugger). The good thing is that the assertion is much more informative to read because its level of abstraction is higher. (Remark: two reviewers disagree with this restriction.) Functions can therefore be used to emulated mathematical quantifiers like *for all* and *exists*.

7.4 When should assertions be turned off?

When a public function calls another public function in the same class (directly or indirectly), an interesting issue arises: the invariant might not hold at that point, but the function might still produce a meaningful result. For example:

```
struct X
{
    void foo( int i ) in { i > 0; } do { ... }
    void bar() { ...; foo( 0 ); ...; }                    // #1
    void call_back() { Y* p = ...; p->call_back( this ); } // #2
    // ...
};

struct Y
{
    void call_back( X* x ) { x->foo( 0 ); }
};
...
X x;
x.bar();       // #1
x.call_back(); // #2
```

If all assertions are disabled inside `bar()`, the error made when calling `foo()` will not be caught. If only the invariant was disabled, it would be caught. If `foo()` relies on a correct invariant, it will still produce wrong results, but detecting that property could be left as a burden on the programmer. In the second situation we get the same behavior.

It seems reasonable to always enforce pre- and postconditions on public functions because

1. if the precondition is broken, it means that the function will never produce correct results (by definition),
2. if the invariant is not broken, the function is guaranteed to produce correct results, and
3. if the invariant is broken, the function might still produce correct results.

(Remark: the new ISE 5.4 implementation of Eiffel only disables invariants. This is a change in policy from version 5.3 where both invariants and precondition were disabled. Unfortunately I do not know what the undergoing ECMA standard for Eiffel will decide.)

7.5 Exception-safety

Certain resource holding objects could be troublesome to use with contracts. Imagine a container that stores dynamically allocated objects:

```
void replace( T* p, size_t where )
in
{
  p != 0:       throw bad_ptr();   // ok, no leak since pointer is 0
  where < size: throw bad_index(); // oops, 'p' might leak
}
do
{
  auto_ptr<T> ptr( p );
}
```

The simplest solution would be to use the comma operator:

```
void replace( T* p, size_t where )
in
{
  p != 0:       throw bad_ptr();
  where < size: delete p, throw bad_index(); // no leak anymore.
}
```

What is particularly good about this is that it documents clearly that the function is exception-safe. A reviewer suggested allowing bracketed code:

```
  where < size: { delete p; throw bad_index(); } // no leak anymore.
```

Some reviewers were against throwing exceptions in contracts while others liked it.

7.6 Is the reuse of `return` confusing?

One reviewer mentioned that he would rather see the result of the function described differently:

```
int foo()
out(r)
{
    r == 1;
} do { ... }
```

This is how D does it.

7.7 Is both an `in` and `out` block necessary?

Having a designated block for postconditions is strictly not necessary. That an assertion is a postcondition could be deduces from the presence of `in` and `return`. For example,

```
int foo( int i )
contract
{
    i > 0;      // precondition
    in i > 0;   // postcondition
    return > 0; // postcondition
} do { ... }
```

I find this confusing.

7.8 Should the order of the function body and postconditions be fixed?

Some programmers might prefer to write a function like this:

```
int foo()
in  { ... }
do  { ... }
out { ... } // this one after function body
```

This block layout could be allowed as well.

8 Can Design by Contract be integrated with other C++ extensions?

8.1 Concepts

It has recently been discussed how concepts might be added to the language. One suggestion is the usage-pattern approach [Str03]:

```
concept Add
{    // We can copy and add Adds
     constraints( Add x )
     { Add y = x; x = y; x = x+y; Add xx = x+y; }
};
```

It might be possible to exchange `constraints` with `implies`:

```
concept Add
{
     implies( Add x )
     { Add y = x; x = y; x = x+y; Add xx = x+y; }
};
```

Moreover, another concept paper suggest this syntax for describing invariants for an equivalence relation `Op` [SR03]:

```
{ Op(x, x) } == { true}
{ Op(x, y) == true } -> { Op(y, x) == true }
{ Op(x, y) == true && Op(y, z) == true } -> { Op(x, z) == true
```

Maybe this could be described as

```
concept EquivalenceRelation
{
    invariant( EquivalenceRelation Op )
    {
      Op(x, x);
      Op(x, y) implies Op(y, x);
      Op(x, y) && Op(y, z) implies Op(x, z);
    }
};
```

To use similar keywords for concepts and assertions could make the language more consistent.

8.2 Static assertions facility

It has recently been proposed to add a static assertion facility to the core language [KMDH04]. The proposal involves adding a new keyword `static_assert` to the language and it is mainly introduced to produce readable compiler error messages from within template code.

However, if the assertions in DbC were required to evaluate expressions at compile-time whenever it is possible, we could achieve the same within the DbC framework. For example:

```
template< typename T >
void foo( T& t )
{
    static_assert( sizeof( T ) > sizeof( long ), "error msg" );
```

could become

```
template< typename T >
void foo( T& t )
in
{
    sizeof( T ) > sizeof( long ) && "error msg";
```

or even

```
template< typename T >
void foo( T& t )
in
{
    static sizeof( T ) > sizeof( long ) && "error msg";
```

That could be seen as a unification of compile-time and run-time assertions.

9 Implementability

This is really not a question I can answer. Instead I have asked Walter Bright, who has implemented partial support for DbC in his C++ compiler [Bri04b] and full support in his D compiler [Bri04a]. He assesses that it will take about one man-month to complete the feature.

10 Summary

Section 1 explained that error removal was a big problem for programmers and suggested that Design by Contract (DbC) could be a help. Section 2 explained what the defining properties of DbC are and gave a short look at the syntax of this proposal.

In section 3 I gave five reasons that could justify DbC: (1) enhanced documentation, (2) better debugging support, (3) design specifications, (4) optimizations, and (5) easier inheritance maintenance.

Section 4 gave an overview of how DbC is implemented in Eiffel and D. The differences between this proposal and the other languages are summarized in table 1 on page 16. Finally I was able to review the details of how DbC should look according to this proposal (see section 5). Contrary to DbC other languages, DbC in C++ has the potential to be (1) const-correct and (2) more flexible with respect to the error-handling.

The next section (see section 6) described what it would take to provide a strong library version of DbC. The conclusion is that it would be difficult. The section also explains why it would be a good idea to introduce the boolean `implies` operator.

In particular, section 7 described design tradeoffs in DbC. The issues are mainly (1) where to put the contract, (2) whether to allow arbitrary statements in contracts, (3) whether to turn contract-checking completely off when a public function calls other public function in the same class (directly or indirectly), and (4) alternatives to the syntax.

Finally, section 8 discussed how keywords may be reused in the next C++ standard. We may also merge the static assertion facility into the DbC scheme to unify run-time and compile-time assertions.

11 Acknowledgements

Thanks goes to Daniel Watkins, Darren Cook, Berend de Boer, Reece Dunn, Kevlin Henney, Walter Bright and Matthew Wilson. Special thanks goes to Walter Bright for implementing Design by Contract in his C++ compiler.

References

[Ale03] Andrei Alexandrescu. Assertions. http://www.moderncpp-design.com/publications/cuj-04-2003.html, 2003. 6

[AM03] Andrei Alexandrescu and Petru Marginean. Enforcements. http://www.moderncppdesign.com/publications/cuj-06-2003.html, 2003. 6

[AT03] Andrei Alexandrescu and John Torjo. Enhancing Assertions. http://www.moderncppdesign.com/publications/cuj-08-2003.html, 2003. 6

[Bez99] Eric Bezault. Eiffel: The Syntax. http://www.gobosoft.com/-eiffel/syntax/, 1999. 2

[Bez03] Eric Bezault. Gobo eiffel project. http://www.gobosoft.com/-eiffel/gobo/, 2003. 8

[Bri04a] Walter Bright. D Programming Language. http://www.digitalmars.com/d/, 2004. 2, 23

[Bri04b] Walter Bright. Digital Mars C++ Compiler. http://www.-digitalmars.com, 2004. 2, 23

[C9903] Rationale for International Standard—Programming Languages—C. http://anubis.dkuug.dk/jtc1/sc22/wg14/www/-C99RationaleV5.10.pdf, 2003. 7

[Eng01a] Interactive Software Engineering. Chapter 10, OTHER MECHANISMS. http://docs.eiffel.com/general/guided_tour/-language/tutorial-11.html#38021, 2001. 4

[Eng01b] Interactive Software Engineering. Chapter 8, DESIGN BY CONTRACT, ASSERTIONS, EXCEPTIONS. http://docs.-eiffel.com/general/guided_tour/language/tutorial-09.html, 2001. 9, 10

[Ens01] Oliver Enseling. iContract: Design by Contract in Java. http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html, 2001. 2, 13

[Eve04] EventHelix.com. Design by Contract Programming in C++. http://www.eventhelix.com/RealtimeMantra/Object_Oriented/design_by_contract.htm, 2004. 2, 6

[Gar98] Ken Garlington. Critique of "put it in the contract: The lessons of Ariane". http://home.flash.net/ kennieg/ariane.html, 1998. 2

[Gla03] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003. 1, 2, 5, 6

[Gue01] Pedro Guerreiro. Simple Support for Design by Contract in C++. Proceedings TOOLS 39, 2001. 6

[Hen03] Kevlin Henney. Sorted. http://www.two-sdg.demon.co.uk/-curbralan/papers/Sorted.pdf, 2003. 2, 4

[ISO98] ISO/IEC. *International Standard, Programming languages — C++*, 1st edition, 1998. 7

[JM04] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. http://archive.eiffel.com/doc/-manuals/technology/contract/ariane/page.html, 2004. 2

[JS04] J. Järvi and B. Stroustrup. Decltype and auto (revision 3), 2004.

[KMDH04] R. Klarer, J. Maddock, B. Dawes, and H. Hinnant. Proposal to Add Static Assertions to the Core Language (Revision 1), 2004. 22

[Mad02] Per Madsen. Testing By Contract—Combining Unit Testing and Design by Contract. http://www.cs.auc.dk/m̃adsen/-Homepage/Research/Publications/madsen_nwper2002.pdf, 2002. 6

[Mak04] Phil Maker. GNU Nana: improved support for assertion checking and logging in GNU C/C++. http://www.gnu.org/-software/nana/nana.html, 2004. 6

[Mar03] Digital Mars. Design by Contract. www.digitalmars.com/-ctg/designbycontract.html, 2003. 10

[Mey97] Bertrand Meyer. *Object Orienteret Software Construction, 2nd edition*. Prentice Hall, 1997. 2, 4, 5, 9

[Mey98] Scott Douglas Meyers. Effective C++ CD, 1998. 8

[Mit00] Mark Mitchell. Optimization that makes C++ faster than C. *Dr. Dobb's Journal*, October 2000. 7

[NG01] Virtual methods should only be private or protected? comp.lang.c++.moderated, 2001. 8

[NG03] Design by Contract in D and C++. comp.lang.c++.moderated, 2003. 15

[Par04] Parasoft. Jcontract, Commercial Design by Contract Software. http://www.parasoft.com/, 2004. 2

[SR03] Bjarne Stroustrup and Gabriel Dos Reis. Concepts—Design choices for template argument checking, 2003. 22

[SSS00] Murray R. Spiegel, John Schiller, and R. Alu Srinivasan. Probability and Statistics, 2000. 9

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994. 2, 17

[Str03] Bjarne Stroustrup. Concept checking—a more abstract complement to type checking, 2003. 21

[Sut03] Herb Sutter. Private mailings, 2003.

[Sut04] Herb Sutter. Virtuality. http://www.gotw.ca/publications/-mill18.htm, 2004. 8

[T99] AT & T. R++. http://www.research.att.com/sw/tools/r++/, 1999. 2