Document Number: P1289R0
Date: 2018-10-08
Reply to: J. Daniel Garcia
e-mail: josedaniel.garcia@uc3m.es
Audience: EWG

Access control in contract conditions

J. Daniel Garcia
Computer Science and Engineering Department
University Carlos III of Madrid

Ville Voutilainen
Qt Company

Abstract

Access controls in contract conditions (preconditions and postconditions) enforce that only members with the same or better access level are used in the corresponding predicates. This paper proposes to remove those controls and allow that every member can be used in a contract conditions. Allowing such access makes contract checking (completely) orthogonal to programmatic APIs, which has many benefits and only dubious disadvantages. Lifting the access restrictions would make contracts more applicable, allowing contracts to be used in a wider spectrum of API design choices, and allowing zero-overhead APIs to be designed without losing the ability to diagnose misuses.

1 Goals

The proposed change has the following goals:

1. Apply contracts to any existing API without changing the (programmatic) API; if the facilities via which a contract can be checked are not exposed via the accessible API but are available or injectable via or into the private or protected API, we still want to be able to apply contracts on such an API, including preconditions and postconditions.

2. Design contract-enforced APIs from the scratch without having to introduce API entry points that expose the correctness checks; such exposure may be contradictory to the programmatic API's design.

The current access restrictions prevent achieving either of those goals. Failing to achieve the first goal is a limitation on the applicability of contracts; some might say there's no harm done and it's a pure opportunity cost. Failing to achieve the second goal could be said to be even more an opportunity cost. Those opportunity costs seem to us too high to bear. It seems plausible that it would be unfortunate indeed if the applicability of contracts to existing APIs is limited this way, and it seems perhaps even more unfortunate to bear the cost of not being able to design new contract-enforced APIs without exposing checking facilities in a programmatic API, because that seems like a very good use of contracts, and a language capability that wasn't available before.

Additionally, having different access rules for contracts conditions and all other code in classes makes the language irregular and more difficult to learn and teach.

2 Introduction

This paper proposes removing constraints on which members may be used in preconditions and postconditions of member functions. As an example, we propose that public member functions may use in their preconditions and postconditions protected and private members. This would lead to less boilerplate code.

As an example, consider the following class.

```
class X {
public:
  //...
  // ptr_ must be non-null
  double get_value() const { return *ptr_; }
```

```
private:
  double * ptr_;
};
```

If we want to express the precondition with contracts (and make it available to callers) we would not be able to write that contract without adding an additional function to the class to get access to the pointer.

```
class X {
public:
  //...
  double * get_ptr() const { return ptr_; }

  // ptr_ must be non-null
  double get_value() const [[expects: get_ptr()!=nullptr]] { return *ptr_; }
private:
  double * ptr_;
};
```

By doing so we are exposing now the `ptr_` member to the caller, which is what we wanted to avoid by encapsulating the pointer in class `X`.

One could argue that a better solution would be to put the predicate in a public member function and avoiding to to expose the pointer to the callers.

```
class X {
public:
  //...
  bool valid_ptr() const { return ptr_!=nullptr; }

  // ptr_ must be non-null
  double get_value() const [[expects: valid_ptr()]] { return *ptr_; }
private:
  double * ptr_;
};
```

Even if this is slightly better, we still have the problem that the interface needs to be enlarged just to be able to express the contracts.

In contrast, we propose that the private member function can be used in public contracts.

```
class X {
public:
  //...

  // ptr_ must be non-null
  double get_value() [[expects: ptr_!=nullptr]] { return *ptr_; }
private:
  double * ptr_;
};
```

3 Problems with the current status quo

Currently the rules defining what can be used in a precondition and a postcondition are governed by clause [**dcl.attr.contract.cond**]. In summary those rules are:

- The predicate has the same semantic restrictions as if it appeared as the first statement in the body of the function.
- A contract condition of a public member function can only use public members or members of base classes that can be used as public members.
- A contract condition of a protected member function can only use public or protected members or members of base classes that can be used as public or protected members.

This leads to a number of problematic situations.

3.1 Exposing implementation details in the interface

One motivating case discussed at WG21 mailing list was:

```
class X {
  private:
    internal_store  s;
  public:
    void open();
    void insert_data(const D & data) [[expects: s.is_open()]];
};

//...

void insert_my_data(X & x, const D & d1, const D & d2) {
  x.insert_data(d1);
  x.insert_data(d2);
}
```

The obvious precondition for function `insert_my_data()` is that `x.open()` must be called before. However, we cannot write that if we cannot use the private data member `s`.

Peter Dimov suggested, three choices for the precondition of `insert_data()` that we study here:

1. `X::insert_data()` can state the preconditions in the form of `[[expects]]` being able to access private members. In that case, the author of `insert_my_data()` cannot express its own precondition.
2. `X::insert_data()` cannot use private members in its `[[expects]]` preconditions but can use an `assert]]` in its body. In that case, the author of `insert_my_data()` cannot express its own preconditions. Consequently, this case is equivalent to the previous one.
3. `X` could make its private state to become public, but this would lead to making the internal state of `X` visible to its clients.

From these three scenarios, the worst case is scenario 3 because it goes against abstraction and would favor a style where many implementation details are visible to clients. The key difference between options 1 and 2 are the ability (in option 1) to use private members in public contracts. However, a drawback of option 2 is that the precondition is no longer visible to the compiler at the caller site which reduces its usefulness.

3.2 Ensuring single initialization

Another example (provided by Loïc Joly) where access control in contracts leads excessive implementation leakage to clients is single initialization of global state.

```
class System {
public:
  static void globalInit()
    [[expects: !alreadyCalled]]
    [[ensures: alreadyCalled]];
private:
  static inline bool alreadyCalled = false;
};
```

In such case class designer may well not want to give access to `alreadyCalled`, since it is only there to help finding bugs. If `alreadyCalled` was exposed through a public `isAlreadyCalled()` function, users might use it in client code (which seems undesirable for the original designer). In fact, they should use `std::call_once()`.

It is also interesting to note, that if the precondition and the postcondition where moved into assertions in the body of the function, information would be lost for some forms of static analysis as well as possible optimizations.

3.3 Subverting the rules with inheritance

If a given class has a private member it will not be possible to use that member in the preconditions of that class. However, a derived class that exposes the member through `using` would be able to use it in its own preconditions.

Additionally, a base class with a private virtual member function may use private members in its contract conditions. If a derived class overrides the member function as public we face the paradox that the overridden function needs to have the same contract conditions but that those conditions cannot be used because they cannot make use of private members.

3.4 Violation of information hiding principle

Some have argued that using private members in the contract condition of a public function would be leaking implementation details in the interface. However, it should be noted that we usually have a number of implementation details in header files today (inline function bodies, private members of classes or whole implementation of templates).

The rationale behind information hiding principle is to limit the number of operations that have access to implementation details and to guarantee that invariants are kept. It should be noted that by allowing the use of private members in preconditions and postconditions we are not changing anything about this.

Firstly, the invariants will be still held. This is a consequence from the fact that any precondition or postcondition that performs and observable modification of the state of the class would lead to undefined behavior.

Secondly, the client code cannot gain access to private members through the preconditions or postconditions. The only thing that client code may know is whether contract conditions are satisfied or not.

3.5 Expressing semantics accurately

If a precondition involves private members and we do not want to expose details by enlarging the class interface, one alternative could be to add an assert to the function body:

```
class X {
public:
  //...

  // ptr_ must be non-null
  double get_value() const;
private:
  double * ptr_;
};

double X::get_value() const {
  [[ assert : ptr_!=nullptr ]];
  return ptr_;
}
```

However, now it is not so clear that the intent of the developer is that the client has the responsibility to make sure that the precondition is satisfied before calling the member function.

```
class X {
public:
  //...

  // ptr_ must be non-null
  double get_value() const [[expects: ptr_!=nullptr]] { return ptr_; }
private:
  double * ptr_;
};
```

The use of preconditions (instead of assertions) also makes much more visible that the function does not have an empty contract and that there are preconditions to be satisfied.

3.6 Examples of multi-state interfaces

Mathias Stearn pointed out that `std::promise` has many member functions that should only be called if it is in certain states, but the current state isnt exposed. For example, you can only call `get_future()` once, and it must not be in a *moved-from* state. The same thing happens for the completion member functions, as calling any of them disallows calling any other as well. Now, technically they are defined to throw if they are misused.

Consequently, technically speaking they are not preconditions and the corresponding member functions have an empty contract (a contract with now preconditions and no postconditions).

However, many have argued that those kinds of APIs are a bad idea. In this case it was because of a potential usage pattern that allows multiple threads to race to complete the promise with a first-in-wins resolution, but that imposes a huge cost on all users, when many dont need that ability, and those that do could achieve it easily with external synchronization.

Consequently, there is precedent in the standard library for having preconditions that arent queryable and rely on users doing the right thing.

3.7 Refactoring an utility type

This idea came up fairly recently in a practical project. The ingredients we have are:

- there's a third-party type that looks a lot like a standard type.
- there's a desire to migrate the implementation and the uses of that third-party type to the standard type
- there are subtle differences in the API semantics of the third-party type and the standard type. What is desired is a way to check that those differences don't cause breakage during migration.

Interestingly, we can now say "contracts to the rescue!". So, we have something along the lines of:

```
template <class T>
class MyOptional {
private:
  MyOptional_impl<T> m_impl;
public:
  template <class T>
  MyOptional(T&& val); // constraints omitted

  template <class T>
  MyOptional(const MyOptional<T>&& val);
};
```

Now we want to first turn `MyOptional_impl` into `std::optional`, and check that there was no practical impact. What we'll do is this:

```
template <class T>
class MyOptional {
private:
  MyOptional_impl<T> m_impl;
public:
  template <class U>
  MyOptional(U&& val) // constraints omitted
    [[ensures: different_optionals_initialized_the_same_way ()]];

  template <class U>
  MyOptional(const MyOptional<U>& val);
    [[ensures: different_optionals_initialized_the_same_way ()]];

  // other ctor overloads omitted

  /* extremely */ private:
  bool different_optionals_initialized_the_same_way ();
};
```

The idea here is that the check in the postcondition of a constructor verifies that the initialization of `MyOptional` did the same thing regardless of whether `m_impl` is `MyOptional_impl<T>` or `std::optional<T>`. The way it does is it initializes the member as usual, and then in the check, initializes another object of the different type and verifies that the engagedness and the value are the same. Same goes for assignments and other operations.

Now, whether other people find such contract uses handy is an interesting question. To me, this is an astounding capability to have. It makes NO sense to expose this checking functionality for users of the type, but it's a really valuable thing to be able to do.

3.8 Avoiding repetition

In the case of overridden functions, preconditions in the form of `[[expects]]` will be present in all overridden functions.

```
class A {
public:
  //...
  virtual void f()  [[expects: ptr_!=nullptr]];
private:
  double * ptr_;
};

class B : public A {
public:
  //...
  void f()  [[expects: ptr_!=nullptr]];
};
```

When the precondition needs to use a a private member, two options are possible.

First option is to add a public member function encapsulating the predicate. This option has already been criticized in previous sections.

```
class A {
public:
  //...
  bool is_valid() const { return ptr_!=nullptr; }
  virtual void f()  [[expects: is_valid ()]];
private:
  double * ptr_;
};

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

A second option would be to move assertions to the function bodies in the form of `assert`.

```
class A {
public:
  //...
  virtual void f() {
    [[ assert : ptr_!=nullptr ]];
    //...
  }
private:
  double * ptr_;
};

class B : public A {
public:
  //...
  virtual void f() {
    [[ assert : ptr_!=nullptr ]];
    //...
  }
};
```

This option leads to the need of repeating the assertion in each overridden function and the possibility for forgetting it in some function.

3.9 Simplicity and regularity

Having special rules for accessibility in preconditions and postconditions are a complication in the language. Programmers need to remember different sets of rules when they are writing function bodies or when they are stating preconditions and postconditions.

A simpler rule is to mandate that the access control performed in preconditions and postconditions are exactly the same than in function bodies.

3.10 Supporting multiple styles

One key characteristic of C++ has been its ability to support multiple styles of programming and not imposing a single style over the others.

Allowing access to private members while allowed might be rare. Still, some styles or programming might decide not use private members, while other styles might need that.

3.11 How do we handle the problems?

It is true that in most of the cases preconditions and postconditions should and can be expressed in terms of the public interface. However there cases where making this an absolute rule would introduce excessive complications. On the other hand allowing access to private members is a simple solution for those cases.

4 Reasons to simplify control access

4.1 Zero-overhead principle

Private access can help allow contracts to promote zero-overhead dont-pay-for-what-you-dont-use code, while giving users a debugging aid to find out why they ran into undefined behavior when they do. Some APIs require operations to be done in certain order, and contracts could be used to check for that.

Consider a `FastFileChannel` where the caller is required to call open before calling `insert_data`:

```
class FastFileChannel {
public:
  bool open(size_t howbig);  // can fail

  void insert_data(Data data) [[expects: f.is_open() && data.size() <= f.available_size ()]];

private:
  File f;
};
```

Or a `FastBuffer` where the caller is required to reserve sufficient capacity before `push_back` is called. In this example we assume that vector has `reserve_nothrow` and `push_back_unchecked` as proposed in P0132 [1]

```
class FastBuffer {
public:
  bool reserve( size_t howmany);

  void push_back() [[expects: buf.size() < buf.capacity() ]];

private:
  vector<char> buf;
};
```

Calling `FastBuffer::push_back` and `FastFileChannel::insert_data` will perform no capacity checks, and the `insert_data` operation will perform no checks on whether the file is open for writing or whether the data fits. The intent is that the API makes it impossible to write slow code: `file.is_open()` is deliberately not exposed, because exposing it has a time and space cost that is not otherwise incurred.

If the concern is about *"exposing implementation details to the client"*, then allowing private member access from preconditions and postconditions of public member functions does not do that, whereas the suggestion to just write public getters for those privates would expose them.

In summary, the desire is that:

- Callers are required/assumed to be correct.

- If the contract is violated, undefined behavior is okay.
- The programmer can guarantee zero checking overhead of the contract in release builds. `File::available_size()` is assumed to be expensive and the desire it that it not be callable in non-checked modes, or alternatively it does nothing. (This further means that such queries are ill-suited to be exposed in an API.)

There are different ways to design such APIs (have a `File` create a `FastFileChannel` that is guaranteed to be open and sized), but its not obvious that such approaches are better, and they might not be feasible considering existing code and existing uses.

4.2 Interface design

Tony Van Eerd provided an interesting example with *iterators*:

Consider the `erase()` member function from a container. A precondition would be that the passed iterator refers to an element in that container.

```
void vector::erase( iterator  it )  [[ expects: is_my_iterator( it )]];
```

Function `is_my_iterator()` is only mildly useful outside a contract. Yet, with current control access it would be necessary to expose it in the public interfaces. Usually, such public member function could be considered a code-smell in normal code.

Similarly, consider the sort algorithm where `last` iterator must be reachable from `first`.

```
template <typename I>
void sort(I  first , I  last )  [[ expects: reachable( first , last )]];
```

Function `reachable()` might only be an approximation (e.g. are both iterators in the same container?). Such approximation would not be suitable as a public member function unless naming has been performed very very carefully.

4.3 Information hiding

Access to non exposed members is reasonable because there may already be suitable private members and they are private for a reason, and exposing them conflicts with that reason. Additionally, some such private members may be conditionally available, and are thus ill-suited for exposure to users.

There are additional observations to be considered. Those observations happen to match some age-old principles rather well:

1. If the concern is abuse (*"unmaintainable mini-programs"*), that should not prevent allowing programmers who know what they are doing from doing so. This is, *"trust the programmer"*.
2. If a programmatic API does not expose certain members, that is for a reason. Having to expose those members to use them in declaration-level contract predicates violates *"do not pay for what you don't use"*, not just on the performance level, but on that level as well.
3. Finally, being able to keep members private and being able to use them in declaration-level contract predicates supports programming with the "zero-overhead principle".

The members may be conditionally available, or may have do-nothing semantics in release mode, which makes them (in some designs, not all) ill-suited for exposure to public clients. In addition, using such members is not necessary to write correct programs, and using them introduces branches, conditions and checks that, in some cases, the API designer does not want to encourage.

5 Alternate solutions

It has been suggested by John Lakos to allow using access to private members in function declarations different from the first one.

```
class X {
private:
  data_store  s;
public:
  void insert_data(const D & d);
};

// ### Implementation only below this line
void X::insert_data(const D & d) [[expects: s.is_open ()]];
```

In such case, access to private members would be allowed only in subsequent declarations of a function, while the first declaration would be reserved only for use of public members.

However, this (as pointed out by Ville Voutilainen) is a change that modifies the current requirement that a *"contract predicate must appear in the first declaration of a function"* into the new requirement that a *"contract predicate can appear in a non-first declaration of a function"*. This change would allow to have multiple non-first declarations with conflicting contract predicates.

If declarations with conflicting contract predicates are allowed two options are possible:

1. Allow conflicting declarations as long as they appear in different translation units. In that case it would be possible that the same function is checked in different ways in different translation units. This approach presents a number of implementation problems. For example, it would be problematic to use the double entry point technique.

2. Consider that conflicting contract predicates are *ill-formed with no diagnostic required*. This option is more aligned with the approach already taken for inline functions.

Even if option 2 is more desirable than option 1, still this would be another introduction of *ill-formed/no diagnostic required* which seems an unnecessary complication.

6 Contracts in other programming languages

Other programming languages have considered this issue. In this section we review how access to private members is handled in some languages supporting contracts.

6.1 D

The D programming language (see `https://dlang.org/spec/spec.html`) allows *"arbitrary D code in the **in** and **out** contract blocks"*. Consequently, it does not impose any constraint accessibility. In particular use of private members is allowed.

The following example (provided by Thrasten Ottosen) is valid in D:

```
class Foo
{
    private int i = 0;

    public void foo( int x )
    in
    {
        assert( x > 0 );
        assert( i > 0 );
    }
    body
    {
        writeln("doh!");
    }
}
```

6.2 Eiffel

Eiffel [2, 3] offers support for checking preconditions and postconditions as well as type invariants. Preconditions can be expressed using only what the caller can see. However, postconditions my access internal representation as they belong to the implementation. Finally, invariants may also access internal representation.

6.3 SPARK

SPARK (see `https://www.adacore.com/sparkpro`) is a subset of Ada 2012 [4]. It does not include runtime checking although it offers preconditions, postconditions and invariants as a way to support static analysis. Accessibility in SPARK is not considered at type level but at the package level.

7 Proposed workding

In section **[dcl.attr.contract.cond]/4**, edit as follows:

The predicate of a contract condition has the same semantic restrictions as if it appeared as the first expression-statement in the body of the function it applies to. Additional access restrictions apply to names appearing in a contract condition of a member function of class C:

- Friendship is not considered.
- ~~For a contract condition of a public member function, no member of C or of an enclosing class of C is accessible unless it is a public member of C, or a member of a base class accessible as a public member of C (10.8.2).~~
- ~~For a contract condition of a protected member function, no member of C or of an enclosing class of C is accessible unless it is a public or protected member of C, or a member of a base class accessible as a public or protected member of C.~~

For names appearing in a contract condition of a non-member function, friendship is not considered. *[Example:*

```
class X {
public:
  int v() const;
  void f()  [[expects: x > 0]];              // OK~~error: x is private~~
  void g()  [[expects: v() > 0]];            // OK
  friend void r(int z)  [[expects: z > 0]];  // OK
  friend void s(int z)  [[expects: z > x]];  // OK~~error: x is private~~
protected:
  int w();
  void h()  [[expects: x > 0]];              // OK~~error: x is private~~
  void i()  [[ensures: y > 0]];              // OK
  void j()  [[ensures: w() > 0]];            // OK
  int y;
private:
  void k()  [[expects: x > 0]];              // OK
  int x;
};

class Y : public X {
public:
  void a()  [[expects: v() > 0]];            // OK
  void b()  [[ensures: w() > 0]];            // OK~~error: w is protected~~
protected:
  void c()  [[expects: w() > 0]];            // OK
};
```

end example]

Acknowledgements

Many people participated through the WG21 mailing lists, among others Matt Calabrese, Peter Dimov, Gabriel Dos Reis, Ion Gaztañaga, Kevlin Henney, Loïc Joly, John Lakos, Nevin Liber, Thomas Köppe, Andrzej Krzemienski, Nathan Myers, Roger Orr, Thorsten Ottosen, Sam Saariste, Oleg Smolsky, Mathias Stearn, Bjarne Stroustrup, Herb Sutter, Tony Van Eerd. We apologize if ommitted unintentionally somebody.

The work from J. Daniel Garcia is partially funded by the Spanish Ministry of Economy and Competetiveness through project grant TIN2016-79637-P (BigHPC – Towards Unification of HPC and Big Data Paradigms) and the Europena Commission thro grant No. 801091 (ASPIDE – Exascale programIng models for extreme data processing).

References

[1] Ville Voutilainen. Non-throwing container operations. Working paper p0132r1, ISO/IEC JTC1/SC22/WG21, May 2018.

[2] ISO/IEC JTC1/SC22. Information technology – Eiffel: Analysis, Design and Programming Language. International Standard ISO/IEC 25436:2006, ISO, December 2006.

[3] ECMA. Eiffel: Analysis, Design and Programming Language. ECMA Standard ECMA-367, ECMA, June 2006.

[4] ISO/IEC JTC1/SC22. Information technology – Programming languages – Ada. International Standard ISO/IEC 8652:2012, ISO, 2012.