Document Number: N4110
Date: 2014-07-06
Reply to: josedaniel.garcia@uc3m.es

Exploring the design space of contract specifications for C++

J. Daniel Garcia
Computer Architecture Group
University Carlos III of Madrid

Abstract

This paper explores the design space for introducing contracts as part of the interface of functions and types. The goal is to address the problem of specifying an interface (with special attention to standard library interfaces) making a clear distinction between a contract violation and an explicitly thrown exception. Preconditions and postconditions are attached to function declarations (i.e. to interfaces) rather than to definitions, and are, in principle, checked before and after a function's definition is entered. No macros are needed.

1 Introduction

In the design of any library there is a tension between two issues that can be seen as related: correctness and robustness. *Correctness* can be seen as the degree to which a software component matches its specification. *Robustness* is the ability of a software component to react appropriately to abnormal conditions.

Currently the C++ language and many libraries use a single feature for managing both properties: exception handling.

When a failure happens we use exceptions as an error reporting mechanism, to notify that an error has occurred and needs to be handled somewhere else. In this way, we allow decoupling the error identification from the error handling.

Besides, when a library detects that some assumed condition has not been met, it needs a mechanism to react properly. We call this situation a *contract violation*. Some libraries just decide not to check contract violations and just document those options. In that case (e.g. the C++ Standard Library), a contract violation leads to undefined behavior.

In this paper, I claim that correctness and robustness are orthogonal properties of a program and that they should be managed independently. This idea is not new, and the approach of independent management of correctness and robustness can be found in other somehow similar languages (e.g. Ada, Eiffel, D).

1.1 Robustness in the C++ standard library

Robustness is related to the identification and handling of abnormal situations. Normally, these errors occur even if a program is completely correct. A typical example is a program that cannot read from a file. Such a failure is treated by throwing an exception an catching it in a different context. Another example is the failure to allocate dynamic memory.

Currently, the C++ standard library identifies the cases when an abnormal situation may happen, by specifying i) the condition that may cause that situation and ii) the exception that will be thrown to notify of that situation.

For example, the standard allocator *std::allocator* has the following *Throws:* clause in the specification of its *allocate* member function:

Throws: `bad_alloc` if the storage cannot be obtained.

Even if a program is correct, it may fail in allocating memory.

1.2 Correctness in the C++ standard library

Correctness is related to finding programming errors. Those situations happen due to incorrectly programmed software. A correctly stated precondition violation happens because the caller is not fulfilling that precondition before performing the call. A correctly stated postcondition happens because the callee is not fulfilling the postcondition after execution. A program failure (a robustness error) cannot be avoided in a program, as it usually depends on some external condition. In contrast, a contract violation (a correctness error), should never happen in a correctly written program.

In its current form, the C++ standard library documents the preconditions of a function. Section 17.4.6.11 states:

> *Violation of the preconditions specified in a functions Requires: paragraph results in undefined behavior unless the functions Throws: paragraph specifies throwing an exception when the precondition is violated.*

Thus, in practice, there are two approaches for contract violations in the standard library: either a precondition violation may result in undefined behavior or it may be notified by throwing an exception.

1.3 The narrow and wide contracts

During C++11 standardization final stage there were arguments against aggressive use of **noexcept** [1]. The key argument was that the fundamental implication was that such approach would make *"functions marked* `noexcept` *difficult to test"*. To overcome this difficulty two approaches were outlined: either to lift the requirement that a `noexcept` violation would result in a call to `terminate` or to impose a severe criteria on when a standard library function can be marked `noexcept`.

The route taken by C++11 was the definition of those criteria [2] based on the definitions of *wide contracts* (does not specify any undefined behavior and has no precondition) and *narrow contracts* (results in undefined behavior when a precondition is violated). With those definitions the adopted guidelines were:

- No library destructor should throw.
- Each library function having a wide contract, that LWG agrees cannot throw, should be marked unconditionally `noexcept`.
- If a library swap function, move-constructor, or move-assignment operator is conditionally-wide, then it should be marked conditionally `noexcept`.
- No other function should be marked conditionally `noexcept`.
- Library functions designed for compatibility with "C" code, may be marked as unconditionally noexcept.

There are two opposed views about the use of `noexcept`. On one hand, there is the view that `noexcept` was designed to solve a very specific problem to correctly handle throwing move constructors. This view tries to limit `noexcept` to move operations. On the other hand, there is a movement of marking `noexcept` as much functions as possible just because they me lead to a performance benefit.

However, `noexcept` is a specification of an interface to indicate that a a certain function does not throw exceptions at all. No less and no more than that. Thus, those functions that do not throw to notify that an abnormal situation has happened should be `noexcept`.

The current library guidelines make that functions that are conceptually `noexcept` are not marked so. This may lead to some performance loses. In fact, this may be seen to be against the principle of zero-overhead. However, the key point is a different issue. Functions are not marked `noexcept` even if they should not be throwing at all.

2 Definitions

To ensure a common understanding I provide a set of definitions that will be used in the rest of the paper.

- **Error condition**: A condition that when met will lead to an error from which a program may or may not recover.
- **Precondition**: A condition that needs to be satisfied prior to the execution of a function.

- **Postcondition**: A condition that is ensured to be held after the execution of a function.
- **Invariant**: A condition that needs to be satisfied prior to the execution of a function and is ensured to be held after the execution of a function.
- **Contract**: Set of preconditions, postconditions and invariants associated to a function.
- **Contract violation**: The execution of a function when a precondition or an invariant is not satisfied before the function or when a postcondition or invariant is not held after the function execution.

3 Alternatives

Mechanisms have been proposed in the past to handle correctness both in the library and the language level.

3.1 Library solutions

Many solutions have been proposed for supporting contracts in C++.

3.1.1 The `assert` macro

The most classical solution has been extensive use of `assert` macro from C. The macro expands to a call to `abort()` when macro `NDEBUG` is not defined. Otherwise it expands to nothing. This leads to a set of issues:

- Assertions are evaluated in the callee site. This means that decision about checking is taken when the callee is compiled. Thus, one can have either a fully checked library with a performance penalty or a faster but unchecked library.
- Assertions cannot be distinguished from regular code. This makes impossible for a compiler to take advantage from semantic information provided in the form of a contract. This information could eventually be used for optimizations or for supporting formal or semi-formal verifications.
- Assertions do not allow to distinguish between different kinds of conditions. Making these distinctions is important if they are going to be activated/deactivated separately.

3.1.2 Centralized defensive-programming library

The most recent proposal [3] in the C++ standards committee about contract programming suggested a library solution for standardization. The proposal contained:

- A definition of a set of build modes (*safe-build*, *debug*, *optimized*).
- A configuration violation-handler mechanism to allow the application developer to make its choice at run-time among several options (including aborting the program, throwing an exception, ...).
- A set of macros for expressing assertions inside implementations of functions.

3.1.3 Limitations of library solutions

While this approach definitely presents key concepts in terms of contract programming, it seems to be focused on preconditions and does not clarify a route for postconditions and/or invariants.

A key point is the fact that a library solution effectively makes the contract part of the implementation. In contrast, a contract should be seen as part of component specification. That approach would also allow that different clients may use the component with or without contract checking.

The introduction of a library solution for contract checking would also prevent a later language-based solution. A language solution may provide opportunities for better development tools. In particular:

- Language-based solution may be a corner-stone to support formal verification tools. Experiences with *SPARK* (`http://www.spark-2014.org/`) have shown that a language-based contract mechanism is essential for formal verification.
- A compiler may use the contract information for better code generation and take advantage of the extra information for code optimization.

3.2 Language solutions

In 2005 a proposal to add contract programming was considered by the Evolution Working Group. The initial proposal [4] contained a discussion on reasons to prefer a language solution over a library solution. Among others, the following were included:

- Side effects cannot be detected by library solutions.
- Assertions about return value or about side-effects cannot be done without manually copying the result.
- There is no clear mechanism for inheriting assertions in public virtual functions.

Proposal [5] was discussed at the Lillehammer meeting. Minutes are rather short:

> (discussed) N1773 05-0033 : Proposal to add Contract Programming to C++ (revision 2) L. Crowl, T. Ottosen
> Straw poll: Evaluate synergies (7/4/3/0)
> (Means: authors get 1 hour floor time on the next meeting to present their findings)

At Mont Tremblant meeting a revised proposal [6] was discussed. However no consensus could be achieved on whether C++0x should pursue contract programming.

The proposal included static assertions and runtime assertions. In this paper the focus is on runtime assertions, as static assertions can be handled by other means in the language. The basic approach included the ability to add preconditions and postconditions to functions and invariants to classes. A simplified example of its use is the following:

```
double sqrt( double r )
    precondition
    {
        r > 0.;
    }
    postcondition( result )
    {
        equal_within_precision ( result * result , r );
    }
```

Another interesting feature was the use of `if` statements to express implication assertions:

```
class vector {
  // ...
   iterator begin()
     postcondition(result) { if (empty()) result==end(); }
  // ...
};
```

In this set of proposals the default behavior when a contract is violated is to call `terminate()`. However, a set of functions are provided to change this default behavior by setting a handler:

```
typedef void (*broken_contract_handler)();

broken_contract_handler set_precondition_broken_handler(broken_contract_handler r) throw();
broken_contract_handler set_postcondition_broken_handler(broken_contract_handler r) throw();
broken_contract_handler set_class_invariant_broken_handler (broken_contract_handler r) throw();
broken_contract_handler set_namespace_invariant_broken_handler(broken_contract_handler r) throw();
```

This model also included specific rules for handling the cases from inheritance and virtual functions:

1. Only the first declaration/definition of F can have a precondition.
2. Pure virtual functions can have contracts.
3. Postconditions from a function in a base class and the overridden version in a derived class are *AND*ed together.
4. If F overrides more than one virtual function due to multiple inheritance, the precondition on F is an *OR*ing of the preconditions of all the functions F override and the postcondition on F is an *AND*ing of the postconditions of all the functions F override and F's own postcondition.

The implementation model proposed implied that all the code was generated at the callee site. This approach allowed to address code bloat issues. For every function with a contract the compiler would generate three functions:

1. **Core function**: This is the function without any checking at all.
2. **Postcondition evaluator**: This one calls the core function and then executes the postcondition checks.
3. **Precondition evaluator**: This one executes de precondition checks and the calls the postcondition evaluator.

Still this model supports the concepts of multiple build modes, where postconditions can be activated/deactivated when compiling the callee and preconditions can be activated/deactivated when compiling the caller.

This model requires to standardize multiple build modes which would be something new to the C++ standard. Besides, it prevents any attempt to optimize out unneeded checks.

3.3 Other languages with contract support

3.3.1 Eiffel

A classical reference in contract programming is the Eiffel language [7, 8].

A typical contract example in Eiffel is as follows:

```
class DICTIONARY [ELEMENT]
feature
        put (x: ELEMENT; key: STRING) is
                require
                        count <= capacity
                        not key.empty
                ensure
                        has (x)
                        item (key) = x
                        count = old count + 1
                end

        ... Interface specifications of other features ...

invariant
        0 <= count
        count <= capacity
end
```

An Eiffel implementation (section 8.8.29 of [8]) must provide facilities for enabling and disabling assertion monitoring. It allows a variety of methods for setting the assertion monitoring level either statically (at compile time) or dynamically (at runtime). This can be done through control information in the Eiffel text or through outside elements such as a user interface or a configuration file.

Eiffel includes preconditions, postconditions, class invariants and loop invariants. check instructions and loop variants. Each of this check can be activated separately either for specific classes, specific *clusters* or the entire system.

At least an implementation must support the following for modes:

1. Statically disable all monitoring for the entire system.
2. Statically enable precondition monitoring for an entire system.
3. Statically enable precondition monitoring for specified classes.
4. Statically enable all assertion monitoring for an entire system.

If an assertion is violated an exception (`ASSERTION_VIOLATION`) is raised to notify that violation.

3.3.2 Ada 2012

Ada [9] has recently incorporated contract programming that was successfully used previously in SPARK. The approach is rather similar to Eiffel.

```
procedure Push (S: in out Stack; E: in Element)
 with Pre => Not_Full (S),
      Post => (S.Length = S'Old.Length + 1) and
              (S.Data (S.Length) = E) and
              (for all J in 1 .. S'Old.Length =>
                 S.Data (J) = S'Old.Data (J));
```

If an assertion is violated an exception is raised to notify that violation.

3.3.3 D

The D programming language (`http://dlang.org`) also provides a contract mechanism.

```
long square_root(long x)
  in
  {
    assert(x >= 0);
  }
  out (result)
  {
    assert((result * result) <= x && (result+1) * (result+1) > x);
  }
  body
  {
    return cast(long)std.math.sqrt(cast(real)x);
  }
```

It also allows the definition of invariants:

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

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

In contrast with other options, D allows any arbitrary statement to appear in a precondition, postcondition or invariant making necessary to use the `assert` statement. This also makes possible that contracts may exhibit side effects.

Again, when a violation occurs an error is thrown. However, in release mode all checks are removed.

4 Exploring the design space

Both, library solutions and language solutions, for contract specifications exhibit advantages and drawbacks. However, a language solution seems much more flexible and opens opportunities for better analysis of programs.

In this section, a first exploration of the design space for a language based solution is performed. The goal is to present the alternatives for such design. In particular, the focus is in the general design and not in the concrete syntax. Thus the specific used syntax is only for exploration.

Remark: *A series of keywords are used for illustration only. Different keywords could be used. Depending on the final design it is likely that they could be contextual keywords.*

4.1 Design principles for a contracts programming mechanism

Before exploring concrete syntax and semantics, this section proposes a set of design principles to guide the rest of the paper:

- An operation contract should allow to express its preconditions and postconditions.
- The language should provide tools for expressing invariants at different scopes.
- Contract specification should be part of the specification of an entity (not the implementation).
- Violation of a contract and abnormal error handling should be handled orthogonally.
- Contracts should be well integrated with polymorphism.

4.2 Operation contracts

Any operation may have an associated contract formed by its preconditions and postconditions.

4.2.1 Expressing preconditions

Any operation specification may have in its declaration a list of preconditions that must be satisfied in order to be able to run properly.

```
double square_root(double x)
  expects(x>=0.0);
```

Question: How are multiple preconditions expressed?

Option 1 Multiple requirement clauses.

This option allows for syntactically delimiting every precondition. This slightly related to the fact that Eiffel's assertions are named.

```
class myvector {
//...
  double get_at(int i)
    expects(i>= 0)
    expects(i<size)
    expects(vec != nullptr);
//...
private:
  double * vec;
  int  size;
};
```

However this option seems to exhibit a verbosity that cannot be easily justified. Besides, the option could lead to developers making use of macros to get rid of this additional verbosity.

Option 2 : Single block of conditions.

A single block with multiple conditions (each one being semicolon terminated) could be used. This would reduce the verbosity of option 1, while keeping clear separation of individual preconditions.

```
class myvector {
//...
  double get_at(int i)
    expects{
      i>= 0;
      i<size;
      vec != nullptr;
    };
//...
private:
  double * vec;
  int  size;
};
```

Option 3 Combine through the `&&` operator.

This seems the simplest and more general approach, by allowing the use of general boolean operators.

```
class myvector {
//...
  double get_at(int i)
    expects(i>= 0 && i<size && vec != nullptr);
//...
private:
  double * vec;
  int size;
};
```

One could argue that this option makes slightly more difficult to identify individual conditions or make the code less readable. However, it is almost syntactically equivalent to option 2. Besides, existing *asserts* could be easily transformed to this form.

Question: What kind of expressions should be allowed in a precondition?

Every precondition is a boolean expression that can be evaluated at run-time. However, a precondition is an expression that should not have any semantic effect on the operation it is associated with. It only states when the operation can be correctly run.

Thus the option taken here is to allow any boolean expression that has no side effect.

```
void f(int i)
  expects(i++ > 0); // Not allowed: Side effect
```

Question: What kind of functions should be admissible to call from a precondition?

Option 1 Any function that has no side-effects.

In principle, it should be safe to call any function that is guaranteed to have no side-effects. However, it is possible that if the function body is not visible to the point where the precondition is defined the absence of side-effects cannot be guaranteed.

Option 2 Only `constexpr` functions.

This option is quite safe, although it is possible is more restricted than required. An argument favoring this option is that it could be revised in a later version.

4.2.2 Expressing postconditions

Any operation specification may have in its declaration a list of postconditions that the developer guarantees to be held after the operation execution.

```
class string {
  // ...

  void reserve( size_type  res_arg=0)
    expects(res_arg < max_size())
    ensures(capacity() >= res_arg);

  // ...
};
```

The variants previously discussed for multiple conditions in preconditions could also be applied to postconditions.

Question: How should the return value be used in postconditions?

A postcondition may be established on the value returned by a function. In that case, it is important to establish a syntax for referring the returned value.

Option 1 Use an arbitrary name.

This could be used by following syntax from D or previous proposal for C++, where this name is an argument to the postcondition clause.

double *square_root*(**double** *x*)
  expects(*x*>=0.0)
  ensures(*result*)(*result*>=0.0);

Option 2 Use a keyword to refer the returned value.

An obvious choice here is to reuse the `return` keyword in the context of a postcondition.

double *square_root*(**double** *x*)
  expects(*x*>=0.0)
  ensures(**return**>=0.0);

However, this reuse of `return` keyword could lead to parsing problems.

Option 3 Use the function name to reference the result.

This option makes the function name to evaluate to the result only in the context of a postcondition.

double *square_root*(**double** *x*)
  expects(*x*>=0.0)
  ensures(*square_root*>=0.0);

Question: How should previous values be expressed in postconditions?

In a postcondition, every argument has two values of interest. The value previous to the operation execution and the value after the value execution. Both values may be of interest, requiring notations to make difference between the initial value and the final value.

In the spirit of making simple things simple, the regular name of a variable should make reference (in the context of a postcondition) to the value after the operation execution. This design decision makes necessary to provide a mechanism to denote a previous value.

Option 1 Use an operator to denote previous value.

Such an operator should probably be named (e.g. `pre`, `prev`, `old`, ...) as any other operator has already a meaning.

void *increment*(**int** & *x*)
  ensures(*x* == *pre*(*x*) + 1);

Remark: Applying operator `pre` to an object implies taking a copy of that object before running the function. Thus, it can only be applied to types that are *copy-constructible*

4.2.3 Invariants

Question: Should class invariants be considered?

Type invariants correspond to class level. Thus, this paper proposes to add them after a class definition.

class *my_vector* {
// ...
}
invariant (*size* >= 0 && *capacity* >= *size*);

An invariant is a condition that needs to be satisfied before the execution of every public member function and is guaranteed to be satisfied also after the execution of every public member function.

Although this paper proposes the idea of class invariants, it is acknowledged that is something that needs further study.

Question: Should `namespace` invariants be considered?

Although previous proposals have considered the idea of *namespace-invariants*, this paper does not address this issue.

Question: Should *loop-invariants* be considered?

Loop-invariants would allow attaching invariants to a loop. One advantage is to explicitly state the conditions that are preserved among iterations of a loop.

```
for (int i=0; predicate(i); advance(i)) {
  do_something(i);
}
invariant(i>=0 && i< 100);
```

4.3 Contract checking

An issue that has been discussed for a long time is whether contracts should be checked at run-time and, if so, whether this is something that should be removed in production builds.

One extreme position is that all checks should always remain in production builds, as pointed out by Hoare [10]:

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

On the other extreme, we may find reasoning about excessive checking and associated run-time costs.

One of the reasons why the C++ standard library is able to provide a good performance is the fact that many preconditions are documented but not checked. It is the responsibility of the caller to perform calls that do not violate requirements. In order to avoid any hurt to that property, the default semantic is that libraries do not include contract checking. It should be the caller the one to decide if the contract needs to be checked or not.

Having contract checking activated by default could lead to repeatedly performing the same (or very similar checks). This cost would be unacceptable for many kind of applications.

To achieve this goal, functions requirements are not part of operations implementation. They belong to the operation interface. This allows that checking if needed is part of the calling code and not of the called code.

Question: How should the caller control contract checking?

Option 1 A compiler flag to control contract checking.

The basic idea is to have one or more compiler flags to enforce contract checking. One drawback of this option is that activation/deactivation would be done externally to source code, leading to a bunch of compiled objects for the same library. Besides, the granularity level for activation/deactivation would be the translation unit.

Option 2 An attribute could be used to express checking.

this option would imply mandating a default and using an attribute for a block-scope to change that default. This approach opens the door to the discussion of whether contract checking has a semantic impact on a program. This controversial issue is not discussed here.

If the default is checked contracts, one could envision the following:

```
void f() {
  myvector v{20}; // A vector of 20 doubles
  v[2] = 3.0; // Contract checked here
  v[42] = 2.1; // Contract violation

  [[unchecked]]
  {
    v[2] = 1.0; // OK and fast
    v[99] = 0.0; // Undefined behavior
```

```
  }
}
```

If the default is unchecked contracts, one could envision the following:

```
void f() {
  myvector v{20}; // A vector of 20 doubles

  [[ checked]]
  {
    v[2] = 3.0; // Contract checked here
    v[42] = 2.1; // Contract violation
  }

  v[2] = 1.0; // OK and fast
  v[99] = 0.0; // Undefined behavior
}
```

To keep backwards compatibility with existing practice in the standards library, the best option is probably to set the default to unchecked code.

Option 3 A keyword for changing checking mode.

This option is similar to the attribute option but uses a keyword avoiding the semantic effect debate of attributes. For example:

```
double get_value(double * v, int sz, int i)
  expects(
    v != nullptr &&
    i >=0 &&
    i < sz &&);

void f() {
  double v[20];
  v[2] = 3.0;
  get_value(v, 20, 2); // OK
  double x = get_value(v, 20, 30); // Undefined behavior

  checked {
    double y = get_value(v, 20, 30); // Precondition check failed
  }
}
```

This approach opens the opportunity for some conditions to be checked at compile time.

Question: Should there be more than one checking mode?

Option 1 Only one checking mode.

This is the simplest option where a call either checks all the contracts or performs no checking at all. One could argue, that in some conditions checking conditions at operation entry is enough and the exit checking should be avoided for performance reasons.

Option 2 Multiple checking modes.

This option could use a set of checking modes. Those checking modes could be a parameter for the checking attribute or clause (or the compiler flags).

Different checking modes could be:

- **none**: No checking at all.
- **preconditions**: Only preconditions are checked.
- **postconditions**: Preconditions and postconditions are checked.
- **full**: Everything is checked.

Question: Will multiple versions lead to code bloat?

The object code of a function will not contain the checking as the proposed model moves this checking to the call site. However, this does not mean that the call site needs to emit this additional code just surrounding every checked call. If checks cannot be proved unnecessary, it might well refactor those checks into an auxiliary function.

Of course, this issue deserves specific investigation if a language solution like the explored in this paper is pursued.

4.4 The effects of a precondition violation

When a precondition violation happens, this is a symptom that the running program is incorrect. Thus the default behavior should be calling `terminate`.

In most cases, this behavior is the desired one. However, alternate behaviors may be explored.

One option, would be to allow the programmer to set one or more contract violation handlers. When a check fails the handler function is called.

Another option is that an exception is thrown when the contract is violated. It is relevant to remark that the exception can be safely thrown in this case (even if the function is `noexcept`) as it would be thrown in the caller side and not in the callee side.

In case of using an exception, it would be reasonable to use a standard exception (e.g. `contract_violation` including information about the call site (file name, line number, function, ...).

4.5 Strong requirements

There are functions with requirements that need to be checked always. Even those requirements may benefit from moving the check to the call site, as they could be eventually elided under specific conditions and they enrich the semantic information of the call site.

```
class vector_double {
  // ...

  double & operator[](int i)
    expects(i<size());

  double & at(int i)
    expects<out_of_range>(i<size());

  // ...
};
```

Strong requirements will be always checked regardless the build mode or the use of any attribute or block keyword that could inhibit other checks.

Without strong checks, preconditions would be duplicated as they would be once as preconditions and again in the code of the function so that they could throw the specific exception. An optimizer could in some cases eliminate the check for the precondition, but would never be able to eliminate the check in the function implementation. Moreover, having strong preconditions allows such functions to be marked as `noexcept`.

4.6 Interactions with inheritance

No major problem is seen in the case of inheritance. Following the path of other existing solutions, should be restricted to virtual functions redefinitions. The general principle is that preconditions cannot be strengthened and postconditions cannot be weakened.

4.7 Interactions with function pointers

The fact that contracts are part of the interface and not of the implementation generates a problem with pointer to functions and pointer to member functions.

Question: Should contracts be checked through pointer to functions?

Option 1 No checking at all.

This is the simplest solution. The decision would be that when a function is invoked through a pointer to function, checked (if any) is bypassed.

void *f*(**int** *i*) **expects**(*i*>0);

using *funcptr* = **void** (∗)(**int**);
funcptr g = *f*;
g(2); *// OK*

Option 2 Make contract part of the function type.

If the contract specification is made part of the function type, then the compiler will be able to generate the contract checking code at the call site when needed.

void *f*(**int** *i*) **expects**(*i*>0);

using *funcptr1* = **void** (∗)(**int**);
using *funcptr2* = **void** (∗)(**int**) **expects**(*i*>0);
funcptr1 g = *f*; *// Error*
funcptr2 h = *f*; *// OK*
h(2); *// Performs checking if needed*

However, this option puts all the burden on the definition of function pointers. It also may have many unwanted effects and questions that would need to be answered (e.g. should overloading on contracts make any sense at all?).

4.8 Interactions with `noexcept`

We believe that the presented approach allows making functions with a contract `noexcept` as the exceptions, if any, would be thrown in the caller site and not in the callee site.

However, this issue clearly needs further study.

4.9 Run-time versus compile-time assertions

Compile time checks should not be treated as part of a contract checking proposal as they can be better handled as an extension to *concepts*.

5 Addressing centralized defensive programming

This section tries to explore how the presented solution would be addressing the issues presented in [3].

5.1 The wide versus narrow contracts debate

The importance of the distinction between wide and narrow contracts must be acknowledged. However, both establish a contract. It is my view that the difference lies in the fact of how a violation is handled.

A *wide contract* only has strong preconditions. Those preconditions are never removed (regardless the checking mode) and the always have the same effect they throw an exception.

const *T* & *std*::*vector*:: *at*(*size_t index*) **const**
  expects<*out_of_range*>(*index*<*size*());

A *narrow contract* may have weak preconditions. Those preconditions may be checked or not, depending on the context and compilation mode.

const *T* & *std*::*vector*:: *at*(*size_t index*) **const**
  expects(*index*<*size*());

One of the reasons that initiated the wide versus narrow contract debate was to establish limitations to the aggressive use on `noexcept` in the standard library specification. It is worth to note that the approach taking in this paper that the contract is part of the interface and not of the implementation would eventually allow that functions with contracts (either narrow or wide) would still be able to be marked `noexcept`.

```
const T & std::vector:: at( size_t  index) const noexcept
  expects<out_of_range>(index<size());
const T & std::vector:: at( size_t  index) const noexcept
  expects(index<size());
```

5.2 Artificially widening of contracts

Expressing a weak contract is not a way of widening that contract.

Given the following example from [3]:

```
class date {
 // ...
 public:
 // ...
 void set_ymd(int year, int month, int day); // narrow
 // Set the value of this object to the specified
 // 'year', 'month', and 'day'. The behavior is
 // undefined unless 'year', 'month', and 'day'
 // together represent a valid/supported date value.
```

We would express it:

```
class date {
  public:
    void set_ymd(int y, int m, int d)
      expects(valid_date(y,m,d));
    //...
  private:
    static bool valid_date(int y, int m, int d);
      // Tell me if it represents a valid/supported date value
```

This rewriting does not change the fact that the contract is narrow. Moreover, nothing stops the writer to mark the function as `noexcept`.

The reasons for not artificially widening narrow contracts (runtime cost, development cost, code size, extensibility, and defensive programming) are all valid and the guideline of not widening otherwise narrow contracts is correct.

Question: Explicitly expressing contracts as part of the interface does not widens an otherwise narrow contract

5.3 Contracts and high-level requirements

A contract solution must satisfy high-level requirements that are different for the library developer and and application developer.

A library developer must be able to:

- Easily implement defensive checks which are active in appropriate defensive build modes. The proposed solution easily achieves this goal as contracts can be activated, although this is not the default mode.
- Easily check that defensive checks are working as intended.

An application owner must be able to:

- Coarsely specify (at compile time) the overall runtime validation overhead.
- Specify precisely (at runtime) the action to take if an error is detected.
- Link translation units compiled with different levels of runtime validation.

This goals are achieved as it is the client the one who makes the choice on whether checks are activated or deactivated.

5.4 Examples

5.4.1 Contract checking

```
std :: size_t  other_strlen (onst char * str)
{
  CONTRACT_ASSERT(str);
  // return length
}
```

This example can be easily expressed as:

```
std :: size_t  other_strlen (onst char * str)
  expects(str!=nullptr)
{
  // return length
}
```

5.4.2 Safe contract checking

```
int get_int_array_element(int *array, int length, int index)
{
 CONTRACT_ASSERT_SAFE(0 <= index );
 CONTRACT_ASSERT_SAFE(index < length);

 return array[index];
}
```

If the distinction between a contract and a *safe* contract is needed, *safe* versions of contracts could be added.

```
int get_int_array_element(int *array, int length, int index)
   expects_safe (
     0<=index &&
     index < length)
{
 return array[index];
}
```

However, this mode could easily lead to may developers defaulting to this mode and resulting in over-checking. In any case, moving the checks to the interface could allow to avoid the checks if they can be proved unneeded.

5.4.3 Violation handling

Setting a contract violation handler is supported by this paper and no significant deviation from N3997 is seen.

5.4.4 Testing contracts

N3997 also provides a mechanism for testing that a function correctly tests against its contract. This is done via macros `TEST_CONTRACT_ASSERT_PASS` and `TEST_CONTRACT_ASSERT_FAIL`. This feature is not currently addressed in this proposal. However, mechanisms could be envisioned if this is really a needed feature.

6 A contract for string

This section outlines a partial specification of a `string` class.

```
template <typename charT, typename traits = char_traits<charT>,
    typename Allocator = allocator<charT> >
class basic_string {
public:

  explicit basic_string (const Allocator & a)
    expects(
      data() != nullptr &&
      size () == 0
    );
```

```
    basic_string (const basic_string & str)
      ensures(valid_copy(str));

    basic_string (const basic_string && str)
      ensures(valid_copy(str) && str. valid_state ());

    basic_string (const basic_strig & str,
                  size_type pos, size_type n = npos,
                  const Allocator & a = Allocator())
      expects<std::out_of_range>(pos <= str.size())
      ensures(valid_copy_n(str, n));

    basic_string (const charT * s, size_type n,
                  const Allocator & a = Allocator())
      expects(s != nullptr)
      ensures(valid_copy_n_cstr(s,n));

    // ...

private:
    bool valid_copy(const basic_string & str) const {
      return data() != nullptr &&
        equal( data(), data() + size, str.data(), str.data() + str. size ()) &&
        size () == str.size() &&
        capacity() >= size();
    }

    bool valid_copy_n(const basic_string & str, size_type n) const {
      auto rlen = min(n, str.size ());
      return data() != nullptr &&
        equal( data(), data() + rlen, str.data(), str.data() + str. size ()) &&
        size () == rlen &&
        capacity() >= size();
    }

    bool valid_copy_n_cstr(const charT * s, size_type n) const {
      return data() != nullptr &&
        equal( data(), data() + n, s, s+n) &&
        size () == n &&
        capacity() >= size();
    }

}
invariant(size() <= capacity());
```

7 Acknowledgments

Bjarne Stroustrup has provided feedback and very useful comments to preliminary versions of this paper.

References

[1] Alisdair Meredith and John Lakos. `noexcept` prevents library validation. Working paper N3248, ISO/IEC JTC1/SC22/WG21, February 2011.

[2] Alisdair Meredith and John Lakos. Conservative use of `noexcept` in the Library. Working paper N3279, ISO/IEC JTC1/SC22/WG21, March 2011.

[3] John Lakos, Alexei Zakharov, and Alexander Beels. Centralized Defensive-Programming Support for Narrow Contracts (revision 5). Working paper N3997, ISO/IEC JTC1/SC22/WG21, May 2014.

[4] Throsten Ottosen. Proposal to add Design by Contract to C++. Working paper N1613, ISO/IEC JTC1/SC22/WG21, March 2004.

[5] Dave Abrahams, Lawrence Crowl, Throsten Ottosen, and James Widman. Proposal to add Contract Programming to C++ (revision 2). Working paper N1773, ISO/IEC JTC1/SC22/WG21, March 2005.

[6] Lawrence Crowl and Throsten Ottosen. Proposal to add Contract Programming to C++ (revision 3). Working paper N1866, ISO/IEC JTC1/SC22/WG21, May 2005.

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

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

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

[10] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford, CA, USA, 1973.