Document Number: N4293
Date: 2014-11-23
Reply to: josedaniel.garcia@uc3m.es

C++ language support for contract programming

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

Abstract

This paper presents some ideas to support contract programming in C++ language. The paper reviews discussions on some basic questions about contract programming. The goal of this paper is to present a minimal proposal where contracts are part of operation specifications. It also focuses on allowing enough freedom under implementation defined behaviors.

1 Introduction

Contracts are part of the interface of an operation. They state under which circumstances an operation is guaranteed to work and what conditions are ensured after execution.

In summary contracts define:

- Conditions that must be met prior to an operation execution.
- Conditions that the operation guarantees after an operation execution.
- Conditions that a value from a type must hold and define a valid state.

During the *Urbana-Champaign* meeting the *Evolution Working Group* discussed general directions for contracts programming in C++. As part of this discussion some general questions were addressed:

1. Do we want contracts?
2. Is performance an important feature of contracts?
3. Is correctness an important feature of contracts?
4. Should contracts be in declarations?
5. Should build modes be standardized?

While, there are many other important questions, these questions are used to drive this paper and as an starting point for a contracts programming design. This paper also revises ideas introduced in N4110 [1] at the light of the answers from the committee to those questions.

1.1 Contracts programming for C++

The current situation regarding contracts is that they conceptually exist in libraries (including the standard library). Some contracts specify that exceptions should be thrown under certain conditions. However, other contracts just state that contract violation would lead to undefined behavior.

As an example, in the standard library some operations define a wide contract (those having a well defined behavior for all calls), even though under certain argument values they throw an exception (e.g. `out_of_range`).

The standard library also contains operations defining a narrow contract (those having a well defined behavior for a subset of all calls). Violation of such contracts leads to undefined behavior.

It is important to stress that contracts include not only preconditions, but also postconditions. Furthermore, the latter are as important as preconditions or even more.

In general, contracts describe essential properties of types. Those essential properties are not only the specification of valide states. They potentially could include complexity guarantees, thread safety, memory safety, ... While not all these essential properties can be easily integrated in a contract system, it should not be forgotten that they are also part of the contract.

> **EWG Direction**: There was a great agreement that C++ should have support for contracts programming.

1.2 Contracts and performance/correctness

The basic goal of contracts should be to improve correctness. That is to verify that:

- Preconditions are met before operation calls.
- Postconditions are met met after operation execution.
- Invariants are held.

Besides, the use of contracts has additional effects beyond correctness: better diagnostics of programming errors, check elision, better reasoning about programs, and potential use by external tools.

> **EWG Direction**: There was an agreement that both features are desirable for contracts programming. The key identified feature of contracts was correctness. However, it was agreed that performance is also an important feature.

1.3 Contracts and declarations

Any of the proposed initial designs on contracts require language support. Basically, there are two approaches for specifying contracts:

- A **library** approach with language support.
- A **language** approach with contracts being part of functions specifications.

A **library** approach (with language support) expresses contract assertions in the body of the functions. This makes contracts easy to deploy. However, contract validation is hidden into function bodies what makes much harder to compilers and other tools to reason about program behavior. It has been pointed out by John Lakos that still contracts could be supported by compiler magic so that signatures do not need to be changed. This seems feasible for *templates* and *inline* functions. However, this approach does not seem to be general without additional magic changes (e.g. some additional support from modules).

A **language** approach requires more profound language changes. And advantage of this approach is that contract validation may be part of the function interface instead of the function body. This approach makes easier for compilers and other tools to reason about programs

> **EWG Direction**: There was a great support to specifying contracts in declarations. While contracts in the body were also discussed the committee did not get to any final voting on this second issue.

1.4 Standardized build modes

Build modes have never been part of the C++ standard. However, almost every implementation includes such a concept. Furthermore, the concept of a *debug* mode is hidden behind the `NDEBUG` macro.

One option for contracts programming is to define a set of build modes. For example N4135 [2] proposes four build modes: *none*, *opt*, *debug*, and *safe*. Another option is to avoid bringing build modes into the standard. This approach requires to specify the semantics for the single mode in the standard, while other modes are possible but outside the standard specification.

> **EWG Direction**: There was a consensus that build modes should not be standardized.

1.5 Summary of directions

Given this, the directions for contracts programming can be summarized as follows:

- C++ should have some form of support for contract programming.
- C++ contracts main goal is to support program correctness.
- C++ contracts should also have performance into consideration.
- C++ should support contracts in function declarations.
- C++ should not standardize build modes.

2 Minimal proposal

This paper introduces a minimal proposal with a strawman syntax for it. Thus, syntax here is used only for illustration. Besides, additional features could be elaborated if there is some agreement on the notions presented in this paper.

2.1 Design principles

This proposal follows the guidance quoted in the introduction of this paper. Moreover the design of this proposal tries to follow a set of design principles:

1. An operation contract should allow to express its preconditions and its postconditions as part of its declaration.
2. Violation of a contract should be handled orthogonally from run-time errors due to abnormal conditions.
3. Contracts should be well integrated with polymorphism.

3 Syntax

3.1 Preconditions

A function precondition is part of its declaration:

```
double square_root(double x)
  expects{x>=0};
```

When an operation has multiple preconditions they can be combined with boolean operators.

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

The `expects` clause may contain any expression that is contextually convertible to `bool`. However, no side effects should be allowed in such predicates.

```
void f(int  i)  expects{i}; // OK
void g(int i)  expects{i++}; // Ill-formed
```

In particular, an expression in a precondition may invoke functions as long as they do not have side effects.

```
class my_vector {
//...
  double get_at(int i)
    expects{ valid_index(i) && valid_vec() };
//...
```

```
  bool valid_index(i) const { return i>=0 && i<size; }
  bool valid_vec() const { return vec!=nullptr; }
private:
  double * vec;
  int size;
};
```

3.2 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};
  //...
};
```

Expressions in a postcondition follow the same rules than expression in preconditions.
In the context of a postcondition the function name refers to the returned value of the function.

```
double square_root(double x)
  expects{x>=0.0}
  ensures{square_root>=0.0};
```

When a value may change during function execution previous value may referred through operator `pre`.

```
double increment(int & x)
  ensures{x == pre(x) + 1};
```

Operator `pre` can also be applied to `*this`.

```
class point {
  //...
  void move(double deltax, double deltay)
    ensures{x == pre(x) + deltax && y = pre(y) + deltay};
```

4 Contract checking

There is a tension on to which extent contracts should be checked. A contract design should be able to accommodate different user group needs.

On one, hand there are many situations where a contract can be proved to be satisfied and consequently the associated checks are unnecessary.

As an example, in the following fragment, it can be proved that the expectation of `i<size` en ensured to be true and no check is needed.

```
vector v(100); // ensures size==100
for (int i=0;i<100;++i) {
  v[i] = f(i); // expects i<size
}
```

Besides, if a contract can be proved to be not satisfied at all the program can me made ill-formed.

```
vector v(10); // ensures size==10
for (int i=0;i<100;++i) {
  v[i] = f(i); // expects i<size -> Ill-formed
}
```

On the other hand, there are situations where a contract cannot be proved at compile-time In that case there are basically two options: either do not perform any checking at all, or perform some degree of checking.

Performing no checking at all favors performance. This is what it is currently done in many standard library operations with a *narrow contract*. However, we get this at the price of library undefined behavior.

Performing checks at run-time favors correctness. This is what many debug implementations of the standard library do. However, we get this additional correctness at the price of less performance.

A solution to this tension is to make run-time checks for contracts *implementation-defined*. Thus, every implementation should define which checks are performed at run-time.

In summary, any implementation should offer both:

- A mode where no run-time check at all is performed.
- One or more modes with different degrees of checking at run-time.

In the rest of this section, options and constraints for implementations are explored.

4.1 Non-checked mode

Even in a non-checked mode an implementation is still required to perform checks at compile-time.

It can be argued that allowing an implementation not to perform any run-time checks could lead to undesired undefined behaviors. For example:

```
void f(int i) {
  myvector v{20, 0.0}; // A vec of 20 doubles initialized to 0
  v[2] = 1.0;  // Not checked. OK.
  v[99] = 2.0; // Ill-formed. Contract violation.
  v[i] = 3.0;  // Could be undefined behavior if i>=20
```

However, this is not worse than the current situation with language arrays and the `std::vector` type.

4.2 Checked modes

A *checked mode* performs checks for contracts that cannot be proved at compile-time.

```
void f(int i) {
  myvector v{20, 0.0}; // A vec of 20 doubles initialized to 0
  v[2] = 1.0;  // Check elided.
  v[99] = 2.0; // Ill-formed. Contract violation.
  v[i] = 3.0;  // Run-time contract violation, if i>=20
}
```

Contract checking is a property of the client code. This means that an implementation could eventually allow two translation units to use some function where one translation unit performs checking and a different translation unit does not perform that checking.

```
// libf.h
void f(int i) expects{i>0};

// ej1.cpp with checking
int g(int i) {
  f(i); // Will check i>0
  // ...
}

// ej2.cpp without checking
int h(int i) {
  f(i); // No checking
  // ...
}
```

4.3 Multiple checking modes

This paper makes a distinction between a build mode and a checking mode. A build mode defines the options used to build a program. While the standard does not establish any build mode at all, most vendors support multiple build modes. Typically those include a *release* mode and a *debug* mode. **No standardization is proposed here regarding build modes**.

On the other hand, a *checking mode* defines which level of checking (if any) is active for a specific translation unit.

One approach used in other languages and approaches is to define checking modes in terms of which component of the contract is checked. Those modes could be:

- **none**: No checking performed at all.
- **pre**: Only precondition are checked.
- **post**: Both preconditions and postconditions are checked.

Another approach is to define several checking levels, and specify for every check its level. This level could be numeric, so that any check with a level lower than a threshold is performed. This is also compatible to the levels provided in [2] with a small set of levels.

As an example, this levels could be:

- **rel**: No checking performed at all.
- **min**: Only critical checks are performed.
- **dbg**: Most checks are performed. Suitable for debugging.
- **safe**: All checks are performed, even the most costly ones.

It is important to remark that the checking mode affects to the client code when it is compiled. Different translation units may be compiled with different checking modes.

In any case, how the level of a contract is specified needs further exploration and is left in this paper as an open point.

4.4 The effects of a broken contract

A key element for contracts success is which should be the effect of breaking a contract in a *checked mode*.

Set of principles:

- Contract checking is a property of client code. Clients must be able to make the choice of how much (if any) checking they want to use.
- A mode is needed where no run-time overhead is added over a non contract based version. Still that mode, may provide diagnostics for contracts that can be proved broken during translation.
- Contract support should not introduce unnecessarily new undefined behaviors into the language.

For any checked mode, an implementation is required to perform the checks specified by the associated mode. The behavior in response to a broken contract should be implementation defined.

4.4.1 Constraints on broken contracts behavior

A *checked mode* needs to define its behavior when a checked contract is broken.

Some choices for an implementation could include:

- **Terminate the program**: The program would be terminated. However it would still be allowed to set a *terminate handler*.
- **Call a handler function**: This is very similar to the previous option but with an specific different handler function for broken contracts.
- **Throw an exception**: A pre-defined exception could be thrown when a contract is checked to be broken (e.g. `broken_contract`). However, this option could eventually prevent that any function with a contract could be made `noexcept`.

5 Acknowledgments

Some ideas in this paper were enriched thanks to discussions with Gabriel Dos Reis, Gor Nishanov, Bjarne Stroustrup, John Lakos and Nathan Myers.

Thanks to Bjarne Stroustrup, Gabriel Dos Reis and John Lakos for providing comments on previous drafts of this paper.

References

[1] J. Daniel Garca. Exploring the design space of contract specifications for C++. Working paper N4110, ISO/IEC JTC1/SC22/WG21, July 2014.

[2] Nathan Myers adn John Lakos, Alexei Zakharov, and Alexander Beels. Language Support for Runtime Contract Validation (Revision 9). Working paper N4253, ISO/IEC JTC1/SC22/WG21, November 2014.