Document: N4319
Date: 2014-11-23
Working Group: EWG
Reply-To: gdr@microsoft.com

Contracts for C++: What Are the Choices?

Gabriel Dos Reis Shuvendu Lahiri Francesco Logozzo Thomas Ball

Jared Parsons

Abstract

This report suggests widening the exploration of the design space of "contracts" beyond the N4135 proposal. In particular, we *advocate first-*class *support for "programming with contracts" in C++* for the benefits of *an*alysis tools. Contracts *bring commentary specifications closer to mechanized scrutiny* and enforcement*, t*herefore exposing potentially *costly* and*/or hard to find bugs. We illustrate our claims with* three frameworks that have been in production and use at Microsoft for over a decade, especially in the area of memory safety and concurrency-related vulnerabilities*.*

1 Introduction

The discussion around "programming by contracts in C++" at the November 2014 WG21 meeting in Urbana, IL, underscored the community's strong preference for an effective and scalable language support. The proposal N4135 "Language Support for Runtime Contract Validation (Revision 8)", its draft revision N4135R1R1, and its previous revisions showed one implementation strategy. It is clear that each organization or group of C++ programmers will have constraints and biases toward particular strategies, in part informed by their own experience, set of problems, and priorities.

Before digging deeper in one particular direction, it is crucial to analyze the design space and it is essential to inform our decisions by decades of experience by various organizations, C++ sub-communities, and lessons from the larger programming community. For instance, Microsoft has years of experience with "programming by contracts", expressed in various forms and extensions to programming languages in the C family, and deployment of analysis tools using contracts in production environments. They have helped find and eliminate classes of vicious bugs related to memory safety and concurrency. These experiences only scratch the surface of what is concretely achievable. They provide – in our view – a baseline for what is possible with a standard notation for contracts in C++.

2 Why do we need contracts?

The benefits of contracts include:

- Runtime checks, complement to static type checking, for early containment of undesired program behavior
- Support for testing
- Executable documentation of functions, and of data structure invariants
- Optimizations (of expensive runtime checks) enabled by statically provable contracts
- Compile-time detection and elimination of bugs through static analysis

3 Ideals for contracts

Ideally, we expect any contract system to

1. Allow direct expression of what is *required* for and what is *ensured* by any unit of code. For instance, pre and post-conditions for functions, normal vs. exceptional behaviors.
2. *Minimize annotation overhead* by exploiting sound language abstraction mechanisms (e.g. functions, classes, templates). Contracts should work well with existing features, and should not hinder evolution.
3. *Preserve the observable behavior* of good program executions.
4. Allow programmers *to opt out* from checking of contracts, for example to support backward compatibility.
5. Enable *static and dynamic analysis tools*, as well as compiler optimizations.
6. Have a *not too verbose syntax*.

We view the requirement of lightweight syntax for "contracts" as essential for wide adoption. An implementation that interprets contracts as purely axiomatic statements should be allowed; but of course, the main benefits come from the checking and verification.

4 Report from the trenches

4.1 CodeContracts

CodeContracts is the official .NET contract system. Contracts are specified via an API included in mscorlib, the core library of .NET.

For instance the contract below specifies that on entry, the reactor should be off, but on exit it has been turned on:

```
public void TurnReactorOn()
{
  Contract.Requires(this.state == State.Off);
  Contract.Ensures(this.state == State.On);

   …

}
```

In its current form, CodeContracts does not require any language support; it is a library-based solution. In order to enforce runtime checks and inheritance of contracts, an MSIL rewriter is run as a post-build step. In the example below, the contract rewriter will move the `Contract.Ensures` at the method exit point. CodeContracts also provides a state of the art static analysis tool to check contracts validity at compile and link time.

Official CodeContracts description: http://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx

API specification: http://msdn.microsoft.com/en-us/library/system.diagnostics.contracts(v=vs.110).aspx

Tools: https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

Some blogs:

http://devjourney.com/blog/2014/02/12/code-contracts-part-1-introduction/

http://codebetter.com/patricksmacchia/2013/12/18/code-contracts-is-the-next-coding-practice-you-should-learn-and-use/

http://programmers.stackexchange.com/questions/211337/why-would-i-use-code-contracts

CodeContracts is popular in the .NET community. It has been downloaded hundreds of thousands times from the Visual Studio Devlabs and Visual Studio Gallery. It is used to annotate more than 4,000 methods and classes from the Base Class Library. It is in use in millions of lines of code (C#, VB) inside and outside of Microsoft.

4.2 SAL

Quoting from http://msdn.microsoft.com/en-us/library/ms182032.aspx:

> *"SAL is the Microsoft source code annotation language. By using source code annotations, you can make the intent behind your code explicit. These annotations also enable automated static analysis tools to analyze your code more accurately, with significantly fewer false positives and false negatives."*

The primary motivation for SAL is the need to document, in executable form at the source level, constraints ensuring *memory safety* and *concurrency safety* of low-level systems code. SAL allows expression of how a function uses its parameters, the assumptions on entry and the guarantees on return. Common SAL annotations deal with relationships between parameters and return values, pointer properties such as non-nullness, nul-terminated strings, and buffer-sizes. For example, the following annotation on `memcpy` (taken from http://msdn.microsoft.com/en-us/library/hh916383.aspx) specifies the relationship between the storages pointed to by `src` and `dest`, and the parameter `count`.

```
void * memcpy(
   _Out_writes_bytes_all_(count) void *dest,
   _In_reads_bytes_(count) const void *src,
   size_t count
);
```

That is a *declarative summary* of the effects of the `memcpy` function, how many bytes are read from `src` and how many bytes must be written to `dest`.

SAL annotations are defined in the header file <sal.h> and distributed with Windows Software Development Kit (SDK) headers and present in VC\include\ directory in official Visual Studio distributions, which include static analysis tools that leverage the SAL annotations.

When evaluated by the tangible results it achieved, SAL is an undisputable success. The SAL framework and associated static analysis tools have been used extensively internally at Microsoft for almost a decade and has resulted in the annotation of significant portions of low-level systems code written in C and C++. SAL helps Microsoft regularly find bugs in production software, as well helping third-party providers who use Visual Studio static analyzers that leverage SAL annotated header files. An early document (http://research.microsoft.com/pubs/70226/tr-2005-139.pdf) about SAL reports several thousand bugs that were found and fixed in Microsoft code bases.

It is possible to express some, but obviously not all, of the declarative summaries expressible in SAL using only the higher level abstraction facilities offered by C++. A standard C++ notation for contracts will obviate the needs for a complex macro-based annotation and also the needs for C++ programmers to resort to ever growing clever declarations made possible by C++14.

A contract system for C++ that does not make SAL redundant, with less annotation efforts, will be considered a failure partly because of SAL's success itself, and partly because it provides a baseline of what is concretely achievable in production environments for decades.

4.3 System C#

System C# is an extension to the C# programming language focused on adding performance and reliability features. Contracts were added as a language feature in the form of `requires` (pre-condition) and `ensures` (post-condition). For example the following guarantees `x` is greater than or equal to `0` and that the value returned from the method will not be null.

```
public string Format(int x)
  requires x >= 0
  ensures return != null
{
   …

}
```

As a language feature contracts required no additional rewriter or post-processing; the compiler just directly inserted the checks. A failure of a contract resulted in immediate process tear down. The language also had a form of C++ `const` modifier that allowed it to enforce that contracts were side effect free. Any attempt to mutate state in a contract resulted in a compilation error.

The System C# language was used in more than seven millions lines of systems oriented code base that ran competitive real world workloads. The deployment of code contracts to this code base uncovered a number of long standing bugs. A large portion were identified simply by highlighting hidden side effects in preconditions.

The language was also capable of generating contract code which could be processed by an abstract interpreter without the need to modify source. This tool was able to identify several bugs in core library

implementations using this mechanism. Analysis outside of the core portion of the code base would require a more scalable implementation of the interpreter.

5 Acknowledgement

We are grateful to Joe Duffy, Dave Sielaff, Herb Sutter, J. Daniel Garcia, and Bjarne Stroustrup for their comments on earlier drafts of this document.

6 References

1. Gabriel Dos Reis, Bjarne Stroustrup, Alisdair Meredith: “Axioms: Semantics Aspects of C++ Concepts”, ISO/IEC JTC1/WG21 doc. no. N2887; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2887.pdf.
2. J. Daniel Garcia: “Exploring the design space of contract specifications for C++”, ISO/IEC JTC1/WG21 doc. no. N4110; http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4110.pdf.
3. Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, Joe Duffy: “Uniqueness and Reference Immutability for Safe Parallelism”, OOPSLA ’12, pp. 21-40; http://research.microsoft.com/apps/pubs/default.aspx?id=170528.
4. Brian Hackett, Manuvir Das, Daniel Wang, Zhe Yang: “Modular Checking for Buffer Overflows in the Large”, ICSE’06, pp. 232-241; http://research.microsoft.com/pubs/70226/tr-2005-139.pdf.
5. John Lakos, Alexei Zakhaarov, Alexander Beels, Nathan Myers: “Language Support for Runtime Contract Validation (Revision 8)”, ISO/IEC JTC1/WG21 doc. no N4135; http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4135.pdf.
6. Alisdair Meredith: “Library Preconditions are a Language Feature”, ISO/IEC JTC1/WG21 doc. no. N4248; http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4248.html.