Contract Programming For C++0x

WG21/N1800 and J16/05-0060

Lawrence Crowl and Thorsten Ottosen
lawrence.crowl@sun.com and nesotto@cs.aau.dk

2005-04-27

Overview

This is an annotated version of the presentation given before the EWG in Lillehammer. All comments are specified with italics. In general comments to a page are put on the following page. The details of the proposal may be found in N1773.

Essentials

The idea is to extend

- function declarations with pre- and post-conditions
- class declarations with class invariants
- namespace declarations with namespace invariants

- *Notice that contracts are put on declarations and not on definitions. This is essential if the compiler is to take optimal advantage of the contracts.*
- *Postcondition and invariants are only meant to be executed in debug-builds whereas it might be feasible to include some or all precondition checks.*
- *The precondition can be generated at the call-site so the error is correctly reported in the caller and not in the callee.*

Pre- and postconditions

Example: `vector<T>::push_back()/begin()`

```
void push_back( const T& val )
     precondition  {
        size() < max_size();
     }
     postcondition {
        back() == val;
        size() == __old size() + 1;
        capacity() >= __old capacity();
     }

 iterator begin()
     postcondition( result ) {
        if( empty() )
            result == end();
     }
```

Pre- and postconditions (comments)

- *So this is how pre- and postconditions look like. In the postcondition of `push_back()`, the keyword `__old` is applied to an expression; the meaning is to take a copy of the expression before entering the function body and then compare it with something in the postcondition.*
- *Obviously a better keyword than `__old` must be found.*
- *In the postcondition of `begin()` we see how we can get a const reference to the return value of the function, here we name it `result`. This construct makes it easier to specify postconditions for function with multiple exists.*
- *Also note how we may embed if-statements in the contracts.*
- *Postconditions will disable contracts when calling other functions to avoid problems with infinite recursion.*

Class invariants

Example: `vector<T>` invariant

```
static invariant
{
     is_assignable<T>::value
        : "value_type must be Assignable" ;
     is_copy_constructible<T>::value
        : "value_type must be CopyConstructible" ;
}

invariant
{
    ( size() == 0 ) == empty();
    size() == std::distance( begin(), end() );
    size() == std::distance( rbegin(), rend() );
    size() <= capacity();
    capacity() <= max_size();
}
```

Class invariants (comments)

- *The first invariant is evaluated at compile-time like the proposed* `static_assert()`*. The intend is that the comment is printed by the compiler on failure.*
- *The second invariant is evaluated at runtime in debug builds. Calls to the invariant will be generated at the end of the constructor body and before calls to pre- and postconditions on* `public` *functions. This order is necessary since pre- and postconditions might rely on a valid object.*

Namespace invariants

Example: namespace invariant

```
namespace foo
{
    int     buffer_size;
    int*    buffer;

    invariant
    {
        buffer_size > 0;
        buffer     != 0;
    }

    static invariant
    {
        sizeof( int ) >= 4 : "int must be 32 bit";
    }
}
```

Motivation (1)

Minimize the need for separate documentation and implementation

- Any kind of redundancy eventually leads to code and documentation being out of sync
- Natural language is remarkably bad for precise statements
- So remove redundancy by turning comments into code

```
/**
 * Removes the last element of the container.
 * It is required that such an element exists.
 */
void pop_back();

void pop_back()
    precondition  { not empty(); }
    postcondition { size() == __old size() - 1; }
```

Motivation (2)

It can enable the compiler to generate faster code

- If the compiler can determine the precondition is fulfilled, the call to the precondition can be removed
- The compiler can assume all contracts are true in the body of functions

```
void foo( int i )
    precondition { i % 2 == 0; }
{   ...
    i /= 2; // safe to do 'i »= 1'

inline void foo_bar( int* p )
{ ...
    if( p ) { ... }

void bar( int* p )
    precondition { p != NULL; }
{ ...
    foo_bar(p);
```

Motivation (2) (comments)

- *The compiler can always assume preconditions and invariants to be true before compiling the body of a function; and similar, it can always assume postconditions and invariants to be true after a function call.*
- *In the function* `foo()` *the precondition enables the compiler to optimize a division; this optimization would not have been possible without the precondition because* `i` *might be a negative number.*
- *When inlining* `foo_bar()` *inside* `bar()` *the compiler can propagate the precondition of* `bar()` *to perform dead-code elimination.*

Motivation (3)

- Improves communication between designers and programmers in large projects (Jack Reeves argued that was a major reason for C++'s success in the 90's)
- Inheritance is easier to use correctly (pre- and postconditions inherited)
- Improve C++'s role in programming courses (teachers will love this)
- It might make static analysis tools more powerful
- It will benefit C++'s image as a secure language
- It gives us a safer library for use in teaching C++

```
T operator[]( size_type off )
                 precondition { off < size(); }
```

Motivation (3) (comments)

- *Sometimes it is OK to make virtual functions* `public` *or* `protected`*; pre- and postconditions will be "inherited" so when the programmer override the virtual function, he does not need to repeat these.*
- *Security is becoming a bigger and bigger issue. The C committee are working on a new version of the C library which is less prone to buffer overruns. In C++ we should strive for more general approaches to security instead of just fixing a particular library function.*
- *Teachers and C++ committee members often ask for a more secure version of the standard library for use in programming courses. If we apply contracts to the standard library, we can check many common errors like out-of-range in the subscript operator.*

Motivation (4)

Complements concepts well

```
template< class T >
concept EqualityComparable
{
    bool operator==( T l, T r )
        postcondition( result )
        {
            result == !( l != r );
        }

    bool operator!=( T l, T r )
        postcondition( result )
        {
            result == !( l == r );
        }
};
```

Benefit: you only specify the contracts once

Motivation (4) (comments)

- *The concepts proposals deal exclusively with structure and types.*
- *A concept in the real world has both syntax and semantics—using Contract Programming we can attach semantics to our concept.*
- *Remark: recall that the postcondition will disable all contracts to avoid infinite recursion.*
- *As an example, imagine we could put contracts on a Container concept. Then we did not have to specify the contracts on the N implementations of this concept, but only once in the concept definition itself.*

Observations

- The suggested syntax should allow minimal changes to existing parsers
- We have some implementation experience (Digital Mars C++, D) $\Longrightarrow$ fairly easy to implement (3 man-months)
- Will require ways to disable run-time checks

```
void critical_for_performance()
{
  ...
  !precondition
  {
      // preconditions disabled here
  }
}
```

- (remark: not part of proposal yet)

Elements of runtime assertions

- If an assertion is violated at runtime, `terminate()` is called, but behavior may be customized

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

broken_contract_handler
set_precondition_broken_handler(
        broken_contract_handler r ) throw();
```

- Calling a function inside contract scope is subject to the same requirements as a call inside a const member function $\Longrightarrow$ prohibits accidental side-effects

Elements of runtime assertions (comments)

- *The fact that you cannot call non-const member functions within contracts in member functions will prevent some accidental side-effects.*
- *Moreover, because the compiler knows that contract scope is special, it can utter warnings when it detects a side-effect inside a contract. This is a major benefit compared a library solution, because inside the function body, the compiler cannot say side-effects should not happen.*

Function pointers

- Indirect calls via function pointers will check pre- and post-conditions
- How?
  - step 1:
    - generate function with two entry points
    - entry 1: at the beginning of the precondition
    - entry 2: after the precondition and before function body
  - step 2:
    - a function pointer would point to entry 1
    - a normal function call might dispatch to entry 1 or 2 (it depends)

Subcontracting

Pre- and postconditions are inherited on virtual functions

```
struct Computation
{   virtual int compute( int r ) const = 0
        precondition
        {
            r > 0;
        }
        postcondition( result )
        {
            result > 0;
        }
};
struct MyComputation : public Computation
{
    virtual int compute( int r ) const {
        return (int)std::sqrt( float(r) );
    }
};
```

Subcontracting (comments)

- *In this example, the programmer does not need to repeat the pre- and postconditions when he override the function. So there is no way sub classes can escape the contract of the base class.*
- *The declaration in a sub class can add to the postcondition to make it stronger; we could also allow weaker preconditions, but good examples of that put to use are rare and the misuses many.*

Summary

- Fits many strategic goals of C++
  - Performance
  - Teaching programming
  - Security
- Other benefits
  - Makes comments into code and removes redundancy
  - Complements Concepts
  - Emphasize C++ as a language with design in interfaces
- Fairly easy to implement

Summary (comments)

- *It is worth noticing that something that makes C++ stand out from other languages is that we can distinguish interfaces from implementations; the interface goes in the header file and documents the design of our classes.*
- *This is major benefit of C++. Nevertheless many people do not like this aspect of C++; we need to give those people more rewards for writing a separate declaration and Contract Programming is a big step in that direction.*