What does it take to specify the interface to a software component? **Kevlin Henney** negotiates programming by contract

Sorted

THE LAST COLUMN EXPLORED REQUIREMENTS under three headings[1]: functional requirements, which are automatically testable and are concerned with purpose; operational requirements, which focus on qualities of service and use; and developmental requirements, which are about qualities of implementation. The last two categories are more specific than the common but vague term, *non-functional*.

At the level of individual, fine-grained software elements, such specifications are more commonly phrased as *contracts* [2, 3] rather than *requirements*.

The contract for something like an operation of a class describes an agreement between the class user and its designer. A contract contains requirements that define the operation's correct use and expected outcomes. So, what does it take to draft such a contract?

Sorting provides us with an example that is an apparently simple but sufficiently deep example of contract negotiation. Sorting algorithms appear wrapped up in almost any standard library and introductory computer science course. For this reason, many programmers think that there is nothing more to learn and that such examples are too trivial to be of any use in larger systems. However, it turns out that designing the contract for a sorting operation embraces subtlety enough to avoid any trivial pursuit.

Signature contracts

Sorting is not, generally speaking, the method of any particular collection type. In Java, such an orthogonal capability is normally expressed as a `static` method in a mini-package-like class. C++ supports the slightly more natural expression of such capabilities as global functions. For the purpose of exploring sorting, C++ has the edge because of its support for generic programming, which is more naturally suited to expressing algorithms.

Taking a leaf straight out of the C++ standard library, the following declaration introduces our sorting operation:

```
template<typename iterator>
void sort(iterator begin, iterator end);
```

Types are clearly contracts: they define what is expected between the calling and called code; any transgression of the contract invalidates the claim to correctly executing code. Contract enforcement may be eager, relaxed or apathetic: a statically checked type system ensures conformance to an expected type at compile time, which is the normal model of use in C++ and Java; a dynamically checked type system, as found in scripting languages such as Python and Ruby or accessible in Java via reflection, instead raises any alarm bells at runtime; an unchecked type system, such as going via `void *` in C or C++, just promises the obscurity and unnecessary excitement of undefined behaviour if the terms of the contract are not met.

For C++ template parameters the notion of type is defined by how the type may be used rather than in terms of a declared type, such as a `class`. The `iterator` type of `sort`'s `begin` and `end` arguments should follow STL idioms: iterators should support pointer-like syntax. To be more precise, the iterators should be random-access iterators, so that they can refer to any element of their sequence in constant time. This means that it takes as long to lookup `begin[0]` as it does to lookup any valid `begin[i]`.

A further requirement is that the value type referred to in the iterator range should support copying and a strict ordering based on the conventional less-than operator, `<`. For Java the copying requirement

FACTS AT A GLANCE

- **A contract is effectively an agreement on the requirements fulfilled by a component between the user of the component and its supplier.**
- **Signature contracts specify the name, arguments and related types for a component.**
- **Functional contracts focus on the effects of an operation, and are often expressed in terms of pre- and postconditions.**
- **Operational contracts address the quality of service characteristics for a component.**

would make no sense, because it supports only reference semantics and no value semantics, and the idiomatic turn of phrase for expressing a strict ordering is to implement the `compareTo` method of the core language `Comparable` interface because operator overloading is not supported.

Functional contracts

The conventional view of design by contract [3,4] is restricted to functional constraints. Such contracts tend to be assertive in nature: they are most commonly framed in terms of operation pre- and postconditions. A precondition needs to be true on entry to an operation, whether a method or a global function, and a postcondition needs to be true on successful return.

The only meaningful precondition for `sort` is that `begin` and `end` refer to elements in the same range. However, note that even though the condition is not readily checkable, it is still a valid constraint on the execution of the function: if the iterators are not in the same range, the caller cannot expect a sensible outcome.

As for the postcondition, it would be easy to think that we're almost done: all the elements in the range must be sorted! However, that is not quite enough. First, what does it mean to be sorted? Expressed as generic predicate, a first stab might look like the following:

```
template<typename const_iterator>
bool is_sorted(const_iterator begin, const_iterator end)
{
    const_iterator previous;
    do
    {
        previous = begin++;
    }
    while(begin != end && *previous < *begin);
    return begin == end;
}
```

One problem with this is that it fails for empty sequences, which are, by definition, already sorted. Another more general problem is that it does not allow successive elements to be equal: they must ascend or be damned. So the correct definition of being sorted is that each successive element must not be less than the previous one, and both empty and single-item sequences are sorted by definition:

```
template<typename const_iterator>
bool is_sorted(const_iterator begin, const_iterator end)
{
    if(begin != end)
    {
        const_iterator previous;
        do
        {
            previous = begin++;
        }
        while(begin != end && !(*begin < *previous));
    }
    return begin == end;
}
```

OK, that's being sorted sorted, but the contract is still incomplete. Consider the following pathological implementation:

```
template<typename iterator>
void sort(iterator begin, iterator end)
{
    for(iterator source = begin; begin != end; ++begin)
        *begin = *source;
}
```

This sets all of the elements in the range to have the same value as the initial element. Is it sorted? Most definitely. Is it what we expected? Most definitely not! So, being sorted is not enough: the resulting sequence of values must be a permutation of the original, i.e. it should contain the same values. A sorting algorithm that pursues its own wayward agenda or that loses a value here or there is of little use to anyone.

Operational contracts

Functional contracts are things that in principle you might want to check by assertion, using the respective `assert` mechanism in C++ or Java, but that in practice you may not. Even when they can be expressed in code, pre- and postcondition checking can have profoundly undesirable effects on performance. You may be thinking that it's always OK to have assertions in a debug build because optimisation is not a consideration, but reconsider: a typical implementation of `is_permutation` that uses no additional memory for intermediate data structures is going to perform up to $O(N^2)$ comparisons, where N is the length of the sequence to be sorted and the O notation[6] indicates the order at which the execution time will change as N increases.

To get a feel for why the complexity of <code>is_permutation</code> would $O(N^2)$, consider how you would compare the before and after snapshots of the sequence. The good news is that the sequence after the call is sorted so you can progress through it with the full knowledge that the values are in ascending order, and identical values will be adjacent. The bad news is that the sequence before the call is not necessarily sorted, which means that you will generally need a linear search to find the occurrence or recurrences of a value. So, in the worst case, to confirm that for N sorted elements you have the same elements you will have to search N unsorted elements $O(N)$ times, which gives you $O(N^2)$.

Quadratic complexity means that performing the permutation check on 1000 elements will be 100 times slower than for 100 elements. That's not the kind of debug version you want to have produced as the default development build. The proper way to check such functional contracts is through testing rather than through debugging [5]. Instead of using the general assertion that the result must be sorted and a permutation of the original, test specific input data scenarios against their expected outcomes.

Just as it would be unreasonable to impose a quadratic overhead in the check, it is perhaps equally unreasonable for the body of the sorting operation to cost $O(N^2)$ or above for N random-access elements. Quadratic is the complexity of the infamous bubble sort, as well as its slightly more compact and presentable cousin, selection sort:

```
template<typename iterator>
void sort(iterator begin, iterator end)
{
    for(; begin != end; ++begin)
       std::swap(*begin, *std::min_element(begin, end));
}
```

Short and sweet, but still quadratic. If such complexity is a little

unreasonable, the following is intolerably unreasonable:

```
template<typename iterator>
void sort(iterator begin, iterator end)
{
    while(std::prev_permutation(begin, end))
        ;
}
```

This implementation iterates through the permutations of the sequence. Each permutation is successively more sorted than the previous one, until the sequence is fully sorted and `prev_permutation` returns `false`. Very cute. Very, very, very slow. It is combinatorially slow, with a worst-case scenario that bites like a vampire: a sequence of 5 elements could take up to 120 iterations to sort; a sequence of 10 elements could take up to nearly 4 million.

The performance complexity of the implementation is not merely an "implementation detail": it should be subject to contract. How can you meaningfully program to an interface if you have to find out the implementation to make reasonable use of it?

The C standard library's `qsort` function has both a type-dangerous, `void *` based interface and a complete absence of performance guarantee. Although it is probably safe to assume that library authors would not go out of their way to create a poor implementation, the absence of any guarantee means that you cannot know whether a space or a time trade off is made in its implementation. C++'s `stable_sort` and Java's `Collections.sort` use additional memory to get their best performance. Other sorting algorithms sort in place, and do not therefore hit the heap. Swings and roundabouts — it's worth knowing which one you are on.

Without actually specifying an implementation, it is reasonable to stipulate an *O(N log N)* requirement for the average performance complexity of *sort*, a contract that can be fulfilled by quicksort, its variants and a number of other algorithms. Additionally, we can specify that sorting occurs in place, without demanding additional heap memory, and that in the event of an exception during sorting the sequence is left in a valid, albeit unspecified, state. ■

References

1. Kevlin Henney, "Inside requirements", *Application Development Advisor*, May–June 2003.
2. Butler W Lampson, "Hints for computer system design", *Operating Systems Review*, October 1983 http://research.microsoft.com/~lampson/33-Hints/WebPage.html
3. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
4. Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.
5. Kevlin Henney, "Put to the test", *Application Development Advisor*, November–December 2002.
6. Brian W Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999.