Science of C++ Programming

Meng Lee & Alexander Stepanov

Hewlett-Packard Laboratories

P.O. Box 10490

Palo Alto, CA 94303-0969

lee@hpl.hp.com & stepanov@hpl.hp.com

January 1994

Abstract

The purpose of this talk is to demonstrate that to transform programming from an art into a science, it is necessary to develop a system of fundamental laws that govern the behavior of software components. We start with a set of axioms that describe the relationships between constructors, assignment and equality, and show that without them even the most basic routines would not work correctly. After briefly describing our object model and introducing a notion of a *nice* or well-behaved class, we proceed to show that similar axioms describe the semantics of iterators, or generalized pointers, and allow one to build generic algorithms for such iterators. C++ is a powerful enough language—the first such language in our experience—to allow the construction of generic programming components that combine mathematical precision, beauty and abstractness with the efficiency of non-generic hand-crafted code. We maintain that the development of such components must be based on a solid theoretical foundation.

"The labours of others, have raise for us an immense reservoir of important facts. We merely lay them on, and communicate them, in a clear and gentle stream..."

Charles Dickens, *The Pickwick Papers*

The message:

1. **There exists a set of *precise concepts* that describe software.**
2. **These concepts are related by *fundamental* laws.**
3. **These laws are *practical*.**

Translation: *not every program that compiles is correct!*

What's wrong with this program?

```
class IntVec {
    int* v;
    int n;
public:
    IntVec(int len) : v(new int[len]), n(len) {}
    IntVec(IntVec&);
    ~IntVec() { delete [] v; }
    int operator==(IntVec& x){ return v == x.v; }
    int& operator[](int i) { return v[i]; }
    int size() {return n;}
};

IntVec::IntVec(IntVec& x) : v(new int[x.size()]), n(x.size()) {
    for (int i = 0; i < size(); i++) (*this)[i] = x[i];
}
```

Definition of correctness:

A component is correct when it satisfies all its intended clients.

Translation:

A class is correct if it works correctly with all algorithms which make sense for it.

We are accumulating a set of correct components gradually; at every step we have to demonstrate that the new addition is working correctly with the already accepted components.

Swap template function

```
template <class T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}

template <class T>
int testOfSwap(T& a, T& b) {
    T oldA = a;
    T oldB = b;
    swap(a, b);
    return a == oldB && b == oldA;
}
```

Swap is the most basic generic algorithm connecting *copy constructor*, *assignment*, and *equality*.

Test of IntVec

```
void initializeIntVec(IntVec& v, int start)
{
    for (int i = 0; i < v.size(); i++) v[i] = start++;
}

main() {
    IntVec a(3);
    IntVec b(3);
    initializeIntVec(a, 0);
    initializeIntVec(b, 1);
    if (testOfSwap(a, b))
      printf("test of swap - passed\n");
    else
      printf("test of swap - failed\n");
}
```

Running test1:

```
cello-59> test1
test of swap - failed
cello-60>
```

LISP eq-like equality is not a correct equality for IntVec.

- Two data structures are equal if they are element-wise equal under the same iteration protocol.
- More generally, two objects are equal if the return results of all their public member functions which return non-iterator, non-pointer types, are equal; moreover, for those member functions which return iterator types pointing to subobjects, results of their dereferencing should be equal.

The corrected equality:

```
int IntVec::operator==(IntVec& x) {
    if (size() != x.size()) return 0;

    for (int i = 0; i < size(); i++)
        if ((*this)[i] != x[i]) return 0;
    return 1;
}
```

Running test2:

```
cello-64> test2
test of swap - passed
cello-65>
```

Multiple swaps

```
main() {
    IntVec a(3);
    IntVec b(3);
    initializeIntVec(a, 0);
    initializeIntVec(b, 1);
    if (testOfSwap(a, b))
      printf("test of swap - passed\n");
    else
      printf("test of swap - failed\n");
    if (testOfSwap(a, b))
      printf("test of swap - passed\n");
    else
      printf("test of swap - failed\n");
}
```

Running test3:

```
cello-65> test3
test of swap - passed
test of swap - failed
cello-66>
```

Assignment:

ARM, Page 334:

...**unless the** user defines `operator=()` for a `class X`, `operator=()` is defined, by **default**, as memberwise **assignment** of the members of `class X`.

- The default **assignment** is inconsistent with the copy constructor.
- **Assignment should be** the destructor followed by the copy constructor.

Corrected assignment:

```
IntVec& IntVec::operator=(IntVec& x) {
    if (this != &x) {
      this->IntVec::~IntVec();
      new (this) IntVec(x);
    }
    return *this;
}
```

Running test4:

```
cello-66> test4
test of swap - passed
test of swap - passed
cello-67>
```

Wouldn't it be nice if this worked?

```
template <class T>
inline T& assignment(T& to, const T& from) {
    if (&to != &from) {
      (&to)->T::~T();
      new (&to) T(from);
    }
    return to;
}
```

Or even nicer:

```
template <class T>
inline T& ::operator=(T& to, const T& from) {
    if (&to != &from) {
      (&to)->T::~T();
      new (&to) T(from);
    }
    return to;
}
```

Theory of objects:

- Every object is either *primitive* or *composite*
- A composite object is made out of other objects that are called its *parts*
- A part is either *local* or *non-local* — data members are local (`v` in `IntVec` points to a non-local part) (the need for non-local parts arises from the need for objects whose size is not known at compile time and also from the need for objects that change their size)
- A part of a part of an object is an (*indirect*) part of this object
- If two objects share a part, then one object is a part of the other (no sharing, objects are disjoint)
- No circularity among objects — an object cannot be a part of itself and, therefore, cannot be part of any of its parts
- When an object is destroyed all its parts are destroyed
- An *applicative* object encapsulates a state (possibly empty) together with an algorithm (`operator()(arguments)` is defined)

- An *iterator* is an object which refers to another object, in particular, it provides `operator*()` returning a reference to the other object
- An ***addressable*** part is a part to which a reference can be obtained (through public member functions)
- An ***accessible*** part is a part of which the value can be determined by public member functions
- Every addressable part is also accessible (if a reference is available, it's trivial to obtain the value)
- An ***opaque*** object is an object with no addressable parts
- Two non-iterator objects are equal when all the corresponding non-iterator accessible parts are equal
- Two iterators are equal when they refer to the same object (i == j iff &*i == &*j)
- An implicit function *area* is defined for all objects
- For the primitive objects the area is equal to `sizeof()`
- For the composite objects the area is equal to the sum of the areas of its parts

- An object is *fixed size* if it has the same set of parts over its lifetime
- An object is *extensible* if not fixed size
- A part is called *permanently placed* if it resides at the same memory location over its lifetime

  (Knowing that a part is permanently placed or not allows us to know how long a pointer which points to it is valid)
- An object is called permanently placed if every part of the object is permanently placed
- An object is called *simple* if it is fixed size and permanently placed

Nice classes

class T is called *nice* if it supports:

- T(const T&)
- ~T()
- T& operator=(const T&)
- int operator==(const T&) const
- int operator!=(const T&) const

A nice class has its constructor, destructor, assignment, equality and inequality linear time in the area of the objects in the class

Niceness is a generalization of first-classness notion in programming languages: nice objects can be passed to functions, returned by functions, stored in data structures and assigned to a variable.

Certain functions constitute a semantically related group. Examples:

- {==, !=}
- {<, >, <=, >=, ==, !=}
- {prefix ++, postfix ++}

Nice classes (2)

such that:

1. **T a(b);** assert(**a** == b);
2. **T** a(b); a.**mutate**(); assert(**a** != b);
3. a = b; assert(**a** == b);
4. a == a (i.e. **&a** == &b implies a == b)
5. **a** == b iff b == **a**
6. (a == b) **&&** (b == c) implies (a == c)
7. a != b iff **!**(**a** == b)

A member function T::s(...) is called *equality preserving* if

a == b **implies** a.s(args) == b.s(args)

A member function of a nice class returning non-iterator value must be equality preserving

Singular values:

A nice class is allowed to have singular values. These are error values which break some of the nice axioms.
Examples:

- IEEE Floating Point Standard postulates that two NANs are not equal to each other.
- Invalid pointer values are not required to be comparable.

Common Lisp *position* function:

```
position predicate sequence & :from-end :start :end :key -> index or nil

(position oddp (list 3 3 3 6 6 6) :from-end :start 2 :end 5)
```

What's wrong with it?

- return type is not always useful—e.g. for lists
- subrange is specified in a wrong way for lists—indexing takes linear time
- the function is not data structure generic—works only for built-in data structures
- multipurpose, but not flexible—cannot have user defined iteration protocol

Find template function

find is
- useful
- generic
- flexible

```
template <class Iterator, class Predicate>
Iterator find(Iterator first, Iterator last, Predicate pred)
    while (first != last && !pred(*first)) first++;
    return first;
}

template <class Iterator, class Predicate>
int testOfFind(Iterator first, Iterator last, Predicate pred) {
    Iterator found = find(first, last, pred);
    return (last == found || pred(*found))
      &&
      (first == found || (!pred(*first) && found == find(++first, last, pred)))
      &&
      found == find(first, found, pred);
}
```

Classification of iterators

Iterator classes are nice classes with operator*() defined and it takes constant time.

- ***trivial*** **iterator:**
- ***forward*** **iterator: ++**
- ***bi-directional*** **iterator: ++, --**
- ***random access*** **iterator: ++, --, +=(int), -=(int), ...**

  where **operations** ++, --, +=(int), etc. take constant time.

For all iterators, a == b iff &*a == &*b.

(it must be true as long as equality is defined between two iterators, even when they are of different classes)

Note on complexity

It has been commonly assumed that the (time and space) complexity of an operation is part of its implementation and should not be specified at the interface level. This assumption is incorrect since it invalidates the main reason for the separation of interfaces and implementations, namely, ability to substitute one module for another with the conforming interface. Such substitution is only meaningful when there is no major performance degradation. That is, very few people would be willing to substitute their stack with a stack that "correctly" implements push and pop, but whose operations take average time linear in the size of the stack.

Depending on the relative complexity of different primitive operations on an abstract data type, clients should choose different algorithms.

Axioms

for forward iterators:

Check these axioms for pointer types!

1. i == j and *i is valid implies *i == *j
2. i == j and *i is valid implies ++i == ++j
3. for any n > 0, *i is valid and i+n is valid implies i+n != i
4. *i is valid implies ++i is valid

for bi-directional iterators:

1. *i implies --(++i) == i

for ranges:

1. [i, i) is a valid range
2. if [i, j) is a valid range and *j is valid then [i, j+1) is a valid range
3. if [i, j) is a valid range and i != j then [i+1, j) is a valid range

- These axioms describe the behavior of valid iterators
- Valid iterators may be obtained either from a container or from a valid iterator

Note on ranges:

A large family of template algorithms is affiliated with forward iterators. All the algorithms use a common idiom of a range [*first*, *last*), that is, they take two iterators, *first* and *last*, and perform a certain computation on all the iterators from *first* to *last*, but excluding *last*.

A range [i, i) is called an *empty* range. Normally, an algorithm does nothing on an empty range. In general, results of algorithms on an invalid range are not defined. It is a programmer's responsibility to assure that ranges are valid since there is no general way which would allow an algorithm to check the validity of a range. (Try to find a way to check whether two pointers to integers (`int*`) define a valid range, that is, they point into the same array.)

Choice of algorithms

Depending on what kind of primitive operations are available on the iterator, different algorithms are used to implement the same function.

For example, inplace rotate : 1 2 3 4 5 -> 4 5 1 2 3

- for forward iterator we use an adaptation of Gries-Mills algorithm which does *n* swaps (*3n* moves)
- for bidirectional iterator we use 3-reverse algorithm which also does *n* swaps (*3n* moves) but with faster inner loop
- for random access iterator we use permutation-cycle algorithm which does *n* + *gcd(n, shift)* moves

rotate (forward iterator)

```
template <class Iterator>
void rotate(Iterator first, Iterator middle, Iterator last)
{
    if (first == middle || middle == last || first == last) return;

    for(Iterator i = middle;;) {
      swap(*first++, *i++);
      if (first == middle) {
        if (i == last) return;
        middle = i;
      } else if (i == last)
        i = middle;
    }
}
```

bidirectionalRotate

```
template <class Iterator>
void bidirectionalReverse(Iterator i, Iterator j)
{
    while (i != j && i != --j)
      swap(*i++, *j);
}

template <class Iterator>
void bidirectionalRotate(Iterator first, Iterator middle, Iterator last)
{
    if (first == middle || middle == last || first == last) return;
    bidirectionalReverse(first, middle);
    bidirectionalReverse(middle, last);
    bidirectionalReverse(first, last);
}
```

randomAccessRotate

```
template <class Iterator, class T>
void rotateCycle(Iterator first, Iterator last, Iterator initial,
                 ptrdiff_t shift, T value)
{
    Iterator ptr1 = initial;
    Iterator ptr2 = ptr1 + shift;
    while (ptr2 != initial) {
      *ptr1 = *ptr2;
      ptr1 = ptr2;
      if (last - ptr2 > shift)
        ptr2 += shift;
      else
        ptr2 = first + (shift - (last - ptr2));
    }
    *ptr1 = value;
}
```

randomAccessRotate(2)

```
template <class Iterator>
void randomAccessRotate(Iterator first, Iterator middle, Iterator last)
{
    if (first == middle || middle == last || first == last) return;
    ptrdiff_t n = gcd(last - first, middle - first);
    while (n--)
      rotateCycle(first, last, first + n, middle - first, *(first + n));
}
```

Language limitations

Since there are no conditional compilation facilities in the language to find out what are the operations defined on the classes, we cannot provide a single version of rotate which calls different algorithms depending on the availability of different operations. So the user has to make the choice among different templates depending on the iterator types.

Classification of components:

- container — manages a set of memory locations, e.g. a vector or a graph
- iterator — provides a traversal protocol through a container
- algorithm — encapsulates a computational process, e.g. lexicographic comparison
- representation — maps one interface into another, e.g. a vector into a stack
- applicative object — encapsulates a state (possibly empty) together with an algorithm, e.g. a state machine

Example program using *find*:

```
main() {
    SimpleVector<int> a(100);
    iota(a.begin(), a.end(), 0);
    int* found = (int*)find(ReverseIterator<int*, int>(a.end()),
                    ReverseIterator<int*, int>(a.begin()),
                    LessThen<int>(5));
}
```

A container: *SimpleVector*

```
template <class T>
class SimpleVector
{
protected:
    T* first;
    T* last;
    void allocate(size_t n){ first = Allocator<T>()(n); last = first + n; }
public:
    SimpleVector() : first(0), last(0) {}
    SimpleVector(size_t n) { allocate(n); }
    size_t size() const { return last - first; }
    int isEmpty() const { return size() == 0; }
    int isNotEmpty() const { return size() != 0; }
    T* begin() const { return first; }
    T* end() const { return last; }
    SimpleVector(const SimpleVector<T>& x){
      allocate(x.size());
```

```
    move(x.begin(), x.end(), begin());
  }
  int operator==(const SimpleVector<T>& x) const{
    return size() == x.size() && equal(begin(), end(), x.begin());
  }
  int operator!=(const SimpleVector<T>& x) const { return !(*this == x); }
  SimpleVector<T>& operator=(const SimpleVector<T>& x){
    if (this != &x) {
      if (size() != x.size()) {
        delete [] first;
        allocate(x.size());
      }
      move(x.begin(), x.end(), begin());
    }
    return *this;
  }
  ~SimpleVector() { delete [] first; }
  T& operator[](size_t n) {return begin()[n];}
};
```

An applicative object: *LessThan*

```
template <class T>
class LessThan {
    T value;
public:
    LessThan(T x) : value(x){}
    int operator==(LessThan<T>& other) const {return value == other.value; }
    int operator!=(LessThan<T>& other) const {return !(*this == other); }
    int operator()(T x) const { return x < value; }
};
```

An abstract representation—*ReverseIterator*

```
template <class Iterator, class T>
class ReverseIterator {
    Iterator current;
public:
    ReverseIterator(Iterator x) : current(x) {}
    T& operator*() const {Iterator tmp = current;return *--tmp; }
    int operator==(ReverseIterator<Iterator, T>& iterator) const
      {return current == iterator.current; }
    int operator!=(ReverseIterator<Iterator, T>& iterator) const
      {return current != iterator.current; }
    ReverseIterator<Iterator, T> operator++() {current--; return *this; }
    ReverseIterator<Iterator, T> operator++(int)
      {ReverseIterator<Iterator, T> tmp = *this; current--; return tmp; }
    ReverseIterator<Iterator, T> operator--() {current++; return *this; }
    ReverseIterator<Iterator, T> operator--(int)
      {ReverseIterator<Iterator, T> tmp = *this; current++; return tmp; }
};
```

Acknowledgements

- The classification of the components was developed jointly with David Musser of Rensselaer Polytechnic Institute. In general, our entire framework is the result of many years of joint work with him on algorithmic libraries in Scheme and Ada. Indeed, he contributed in one way or another to all of our activities.
- Andrew Koenig of AT&T Bell Laboratories pointed to us that C++ requires an object model which is based on value semantics and, thus, fundamentally different from Lisp or Smalltalk object models. He also suggested to us the use of ranges and collaborated with us on the notion of nice classes.
- Mehdi Jazayeri participated in the early stages of this research.
- Milon Mackey and John Wilkes were always helpful with insightful suggestions.
- Bjarne Stroustrup enabled our research by designing a language which allows all of our ideas to be realizable.
- We are very grateful to Bill Worley who started our project in HP Labs. Without him none of this would have been discovered.

Conclusions:

- C++ has matured into a language the core of which describes an elegant abstract machine, which is both highly generic and efficiently implementable.
- This abstract machine consists of:
  - a set of primitive types
  - an extensible type system which allows a user to define a value semantics for a type
  - a typed memory model based on a realistic machine memory model
- Templates and inlining allow us to program this machine without any performance penalty.
- The abstract machine is simple enough so that its behavior can be understood.
- This machine combined with a rigorous set of rules gives us the solid foundation for collecting software knowledge in a systematic, abstract, and practically usable way, and, thus, turning it into a science which will serve the software engineering the same way as calculus serves the traditional engineering disciplines.

Bibliography

1. M.Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, New York, 1990.

2. D. Gries, *The Science of Programming*, Springer-Verlag, 1981.

3. D. Kapur and Srivas, "Computability and Implementability Issues in Abstract Data Types," *Science of Programming*, Feb. 1988

4. D. R. Musser and A. A. Stepanov, "A Library of Generic Algorithms in Ada," *Proc. of 1987 ACM SIGAda International Conference*, Boston, December, 1987.

5. D. R. Musser and A. A. Stepanov, "Generic Programming," invited paper, in P. Gianni, Ed., *ISSAC '88 Symbolic and Algebraic Computation Proceedings, Lecture Notes in Computer Science 358*, Springer-Verlag, 1989.

6. D. R. Musser and A. A. Stepanov, *Ada Generic Library*, Springer-Verlag, 1989.

7. D. R. Musser and A. A. Stepanov, "Algorithm-Oriented Generic Software Library Development," Technical report HPL-92-65(R.1), Hewlett-Packard Laboratories, November 1993.

Appendix

```
template <class Iterator, class T>
void iota(Iterator first, Iterator last, T value)
{
    while (first != last) *first++ = value++;
}

template <class Iterator1, class Iterator2>
Iterator2 move(Iterator1 first, Iterator1 last, Iterator2 result) {
    while (first != last)*result++ = *first++;
    return result;
}

template <class Iterator1, class Iterator2>
Iterator1 mismatch(Iterator1 first, Iterator1 last, Iterator2 otherFirst) {
    while (first != last && *first == *otherFirst++) first++;
    return first;
}
```

Appendix(2)

```
template <class Iterator1, class Iterator2>
int equal(Iterator1 first, Iterator1 last, Iterator2 otherFirst) {
    return mismatch(first, last, otherFirst) == last;
}
```