HEWLETT PACKARD

Science of C++ Programming

Alexander A. Stepanov

Hewlett-Packard Laboratories

(joint work with Andrew Koenig, Bell Labs, and Mehdi Jazayeri and Meng Lee, HP Labs)

November 11, 1993

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. 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 message:

1. There exists a set of *basic concepts* that describe software.

2. These concepts are related by *fundamental* laws.

3. These laws are *practical* and *self-evident*.

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

What is *wrong*, anyway?

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.

Swap template function

```
template <class T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
template <class T>
void testOfSwap(T& a, T& b) {
    T oldA = a;
    T oldB = b;
    swap(a, b);
    if (a == oldB && b == oldA)
      printf("test of swap - passed\n");
    else
      printf("test of swap - failed\n");
}
```

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

Test of IntVec

```
main() {
    IntVec a(3, 0);
    IntVec b(3, 1);
    testOfSwap(a, b);
}
```

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 iff they are element-wise equal under the same iteration protocol.
- More generally, two objects are equal iff 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.

Equality is the fundamental concept.

Definition: A class definition is *complete* if its public members and member functions allow one to implement any computable function on its objects.

Theorem: A class definition is complete when its equality operator can be implemented using only its public members and member functions.

(Theorem: A class definition is *efficiently complete* when its equality operator can be implemented using only its public members and member functions in time linear to the *total size* of the object.)

(Kapur and Srivas, *Computability and Implementability Issues in Abstract Data Types*, Science of Programming, Feb. 1988)

Practical rule: When developing a class, make sure to implement equality using only public members and member functions.

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>
```

Is it *really* correct now?

Multiple swaps.

```
main() {
    IntVec a(3, 0);
    IntVec b(3, 1);
    testOfSwap(a, b);
    testOfSwap(a, b);
    testOfSwap(a, b);
}
```

Running test3:

```
cello-65> test3
test of swap - passed
test of swap - failed
test of swap - passed
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
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;
}
```

NICE CLASSES

class T is called *nice* if it supports:

- **T(T&)**
- T& operator=(T&)
- int operator==(T&)
- int operator!=(T&)

such that:

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

Niceness is a generalization of first-classeness 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)

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

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

A class is called *scalar* if

all of its public members and member functions (except the & operator) are 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
```

What's wrong with it?

- return type is not always useful—e.g. for lists
- the function is not data structure generic—works only for built-in data structures
- multipurpose, but not flexible

Find template function

```
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);
}
```

find is

- useful
- generic
- flexible

CLASSIFICATION OF ITERATORS

Iterators are scalar classes with operator*() defined.

- *forward* iterator: ++
- *bi-directional* iterator: --
- *random access* iterator: +=(int) — constant time!

Axioms

for iterators:

Check these axioms for pointer types!

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

Classification of components:

- abstract data type — encapsulates a state, e.g. a vector or a graph
- abstract algorithm — encapsulates a computational process, e.g. lexicographic comparison
- abstract representation — maps one interface into another, e.g. a vector into a stack
- abstract functional object — encapsulates a state together with an algorithm, e.g. a state machine

Example program using find:

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

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 the meaning of 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 and formalized

Functional abstraction Less:

```
template <class T>
class Less {
public:
    Less() {}
    int operator==(Less&) {return 1; }
    int operator!=(Less&) {return 0; }
    int operator()(T x, T y) { return x < y; }
};
```