```
def negate(x): return -x
map(negate, range(1, 10))
```

whereas in Perl we would write:

```
map { -$_ } (1..10)
```

and the C++ STL allows us to do the equivalent (Josuttis 1999, 9.6.2):

```
vector<int> coll1;
list<int> coll2;

// initialize coll1

//negate all elements in coll1
transform(coll1.begin(), coll1.end(),          //source range
          back_inserter(coll2),                //destination range
          negate<int>());                      //operation
```

It is perhaps a tragedy of our times that many C++ programmers will write the code just shown using loops over arrays.

Duck typing can create controversy—and it indeed it has. In statically typed languages, such as C++, the compiler will check that the object used in an expression involving a latent type does offer the required interface. In a dynamically typed language such as Smalltalk, this will be discovered at runtime when it produces an error.

This cannot be dismissed out of hand. Strongly typed languages prevent programmers from oversight; they are particularly helpful in big projects, where a good structure aids maintenance. The most important change from traditional Unix C to ANSI C in the 1980s was the introduction of a stronger type system: C got proper function prototypes and function arguments would from now on be checked at compile time. We now frown upon cavalier conversions between types. In short, we exchanged some freedom for some discipline—or, alternatively, chaos for some order.

In Smalltalk, it is generally assumed that chaos should not result, as we should be writing small code fragments while testing them at the same time. Testing is easy in Smalltalk. Because there is no distinct compile and build cycle, we can write small code fragments in a workspace and see directly how our code behaves. Unit testing is also easy; we can write unit tests before our code without worrying about compiler messages referring to undeclared types or methods. It may not be accidental that the community that brought us JUnit shares a lot with the Smalltalk community. We can achieve much of the same in Java by using Java scripting, with BeanShell or Groovy for instance. In fact, strongly typed languages can give us a false sense of security:

> If a program compiles in a strong, statically typed language, it just means that it has passed some tests. It means that the syntax is guaranteed to be correct.... But there's no guarantee of correctness just because the compiler passes your code. If your code seems to run, that's also no guarantee of correctness.