Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/95−0041
WG21/N0641
Date: 31 January 1995
Project: Programming Language C++
Reply to: Andrew Koenig
AT&T Bell Laboratories
PO Box 636
600 Mountain Avenue
Room 2C−306
Murray Hill, NJ 07974 USA
ark@research.att.com

Copy optimization

Several years ago, Jerry Schwarz proposed informally an optimization rule that, among other things, covers the widely-implemented technique of aliasing a function's return value to the place in the caller's memory that value is eventually to be stored.

This note proposes what I think is the most general form of that optimization that is likely to be acceptable: whenever either the user or the implementation creates a copy of an object (using a copy constructor) and the implementation can detect that either the original object or the copy will not be used again before it is destroyed, the implementation can instead create a reference to the original object and arrange to extend the lifetime of the ''original'' object, if necessary, until the ''copy'' would also be destroyed. For example:

```
void f()
{
        string s1 = "hello";
        string s2 = s1;
        // ...
}
```

If `s1` is not used again, the implementation should be allowed to treat `s2` as an alias for `s1`. Similarly, if `s2` is not used again, the implementation should be allowed to eliminate the copy.

This proposal generally does not affect the well-known technique of defining objects that are not used so that their constructors' side effects can occur. The only time it might affect such a technique is when such objects are initialized as copies of other objects of the same type. I have never seen that happen.

To implement this change, I propose a new numbered subclause be added at the end of Clause 12, as follows:

Copying class objects and optimization

Whenever a class object is copied and the implementation can prove that either the original or the copy will never again be used, an implementation is permitted to treat the original and the copy as two different ways of referring to the same object and not generate a copy at all. In that case, the object is destroyed at the later of the times when the original and the copy would have been destroyed without the optimization.[1] For example:

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005

1. Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed.

```
class Thing {
public:
      Thing();
      ~Thing();
      Thing(const Thing&);
      Thing operator=(const Thing&);
      void fun();
};

void f(Thing t) { }
void g(Thing t) { t.fun(); }

main()
{
      Thing t1, t2, t3;
      f(t1);
      g(t2);
      g(t3); t3.fun();
}
```

Here `t1` does not need to be copied when calling `f` because `f` does not use its formal parameter again after copying it. Although `g` uses its parameter, the call to `g(t2)` does not need to copy `t2` because `t2` is not used again after it is passed to `g`. On the other hand, `t3` is used after passing it to `g` so calling `g(t3)` is required to copy `t3`.