Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/92-0029
WG21/N0098
Date: January 28, 1992
Project: Programming Language C++
Reply to: Andrew Koenig
AT&T Bell Laboratories
184 Liberty Corner Road
Warren NJ 07059 USA
ark@europa.att.com

Lifetime of temporaries

Andrew Koenig

Motivation

Classes that represent character strings often do something like this:

```
class String {
private:
      char* data;
public:
      String(const char* p) {
            assert(p != 0);
            data = new char[strlen(p)+1];
            strcpy(data, p);
      };
      ~String() { delete data; }
      operator const char*() { return data; }
      int size() const { return strlen(data); }
      // ...
};
```

The conversion to `const char*` is intended to allow things like this:

```
String s = "hello";
printf("%s\n", (const char*) s);
```

Now suppose we add a concatenation operator:

```
String operator+ (const String&, const String&);
```

We might then expect to be able to do this:

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First Street NW, Suite 500, Washington DC 20001

```
String s = "hello";
String t = " world";
printf("%s\n", (const char*) (s+t));
```

But does this really work?

The answer is unfortunate: the present draft leaves it up to the implementation and many (but not all) implementations actually allow this program to work 'correctly.' The expression `s+t` is evaluated into a temporary and `operator const char*` is called for that temporary. After `operator const char*` returns, the draft says the temporary is 'dead' and that it can therefore be destroyed any time until the end of the containing block.

Unfortunately, if the temporary is destroyed before calling `printf`, then the associated destructor will free the auxiliary memory used by that temporary. That, of course, is the same memory addressed by the pointer returned by `operator const char*`, so that by the time `printf` gets around to using that pointer, it points to garbage.

Of course, the current draft does not require C++ implementations to free the temporary that early. Indeed, some implementations leave the temporary around until the end of the enclosing block, in which case that same pointer now points to valid memory.

This disparity between C++ implementations naturally raises the question: what, if anything, should the Standard say about lifetime of temporaries?

Accurate definition is better for users

The present situation is better for implementers than for users. If I am trying to write a portable C++ program, I cannot say

```
printf("%s\n", (const char*) (s+t));
```

because there is no guarantee that it will work on every implementation. On the other hand, implementers will have a strong incentive to destroy temporaries as late as possible because almost all implementers want to make it easy for users to move programs *to* their implementation. Thus we have the distressing situation where most implementations will have this useful behavior available but conscientious users will not be entitled to take advantage of it!

This is a general principle: a Standard whose purpose is to make it easy for people to write portable programs should leave as little unspecified as reasonably possible. For that reason, it is important to look at the lifetime of temporaries and see whether it can be defined more sharply than it is at present.

The alternatives

The Standard might specify any of several possibilities for destroying temporaries. We can associate each possibility with a phrase that describes it:

Immediate destruction. This would require every temporary to be destroyed as soon as its value has been used. It may be necessary to say precisely what is meant by "used," of course. For example, in an expression like `f(x).g()`, it is essential that the value returned by `f` not be destroyed until after `g` has been called! Aside from that, though, there would be few difficulties in agreeing on just what is meant by "immediate."

End of statement. A temporary persists until the end of the statement that created it. That would allow

```
printf("%s\n", (const char*) (s+t));
```

to work but would prohibit

```
const char* p = s + t;
printf("%s\n", p);
```

because the temporary created by `s+t` would have to be destroyed before the `printf` call could get at it. This would mean that by the time `printf` is called, `p` points to garbage.

End of block. This is probably the most liberal policy that makes sense: a temporary stays around until the end of the block in which it was created. This would allow both

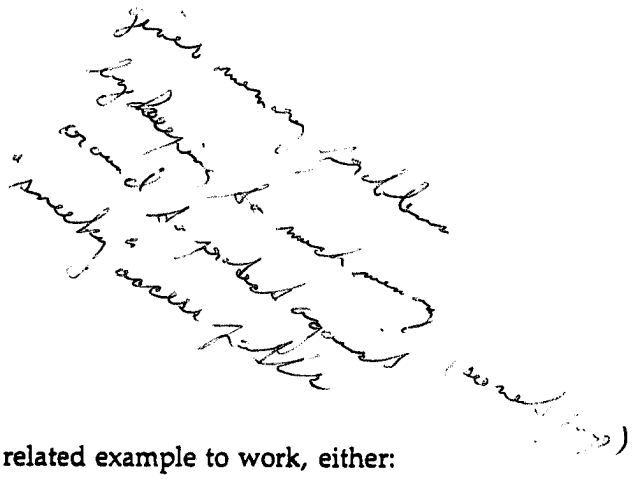
```
printf("%s\n", (const char*) (s+t));
```

and

```
const char* p = s + t;
printf("%s\n", s+t);
```

but would still prohibit this:

```
const char* p;
{
        String s = "hello";
        String t = " world";
        p = s + t;
}
printf("%s\n", p);
```



That seems fair, as no one would expect the following related example to work, either:

```
const char* p;
{
        String s = "hello";
        p = s;
}
printf("%s\n", p);
```

There is another reason not to demand that temporaries persist beyond the block in which they were created: that block might be iterated:

```
const char* p[100];
for (int i = 0; i < 100; i++) {
        String s = "hello";
        p[i] = s + String("");
}
```

If temporaries persist beyond their block, the implementation would have to maintain a dynamic list of all the temporaries that were created in evaluating the multiple instances of `s + ""` in order to guarantee that the elements of `p` point somewhere sensible. Garbage collection may be a desirable thing, but this is not a good way to introduce it!

Conditional temporaries

Temporaries are sometimes created in a conditional context. In that case, it might make sense to destroy them within the 'scope' of that condition *regardless* of the policy that applies to other temporaries.

The trouble is that doing otherwise may have to be implemented in a way that is expensive to the user. Consider, for example, the expression `x?(const char*)(p+q):0`. If `x` is nonzero, then `p+q` is evaluated, yielding a temporary that must be destroyed at some point. If `x` is zero, then that temporary was not constructed and therefore must not be destroyed.

If the destruction of this temporary is deferred until after the `?:` expression is complete, the implementation must remember whether the temporary was constructed in order to decide later whether to destroy it. If there are many such temporaries, much time may be

consumed in testing such flags.

Therefore, even if unconditionally created temporaries persist until the end of the block, it might make sense to destroy conditionally created temporaries earlier. We can distinguish the resulting cases by using the word *conditional* to refer to a strategy in which temporaries created conditionally are destroyed early and *unconditional* for a strategy in which all temporaries are destroyed at the same time. Thus we might have *conditional end of statement*, *unconditional end of statement*, and so on.

Early destruction is hazardous

At one time I thought that immediate destruction was the best course because it made it easy to define the meaning of an expression: evaluate its subexpressions, evaluate its root operator or function, destroy the temporaries in the subexpressions, and yield the result of the root operator.

Two things convinced me otherwise. First, immediate destruction is essentially the strategy adopted by the GNU `g++` compiler. The author of their string library says he receives several complaints a month from users about things like

```
printf("%s\n", (const char*) (s+t));
```

This suggests that immediate destruction has usability problems. I thought at first that those problems could always be traced to classes that returned a pointer (or reference) to data belonging to the class itself, and that the solution was simply not to design string classes with `const char*` conversions but rather require the user to allocate the memory for the result of the conversion. However, consider this:

```
const String& passthru(const String& x) { return x; }
```

This function accepts a reference and returns the same reference. There doesn't seem much wrong with that. But under early destruction, the following will fail:

```
int n = passthru(s+t).size();
```

because the temporary associated with `s+t` will presumably be destroyed before calling `size` on what, as far as the compiler is concerned, is a completely independent object! This suggests that immediate destruction is too early, just as retaining temporaries beyond the end of the block is too late.

There is, however, another possibility. In the example above, the call to `passthru(s+t)` binds a reference to the result of `s+t` because `passthru` takes a `const String&` argument. The current draft says that a temporary with such a reference bound to it persists as long as the reference does, but is not presently clear as to just how long that is when the reference is itself a formal parameter, as here. It might, for instance, make sense to say that such a reference persists until the end of the largest expression enclosing it.

Safety, frugality, and invariants

The main practical issue surrounding lifetime of temporaries is keeping users from being surprised when memory they expected to be able to use vanishes unexpectedly. In that sense, later destruction is always better, because it always reduces the set of circumstances in which users might be unpleasantly surprised.

However, several users have noticed that the amount of memory consumed by temporaries is potentially significant:

```
Matrix a, b, c;
// give values to a, b, and c, and then ...
Matrix m = a + b * c;
```

Here, `b*c` requires a temporary. If that temporary cannot be deleted until the end of the surrounding block, it may wind up consuming a great deal of memory in a context where the

user did not expect it. Moreover, it doesn't work to force earlier destruction by saying

```
{ Matrix m = a + b * c; }
```

because then `m` goes out of scope as well.

Destruction at end of statement might seem to be the best compromise here, but it too violates the principle of least surprise: in all other contexts, where `e1` and `e2` are expressions,

```
e1, e2;
```

has exactly the same effect as

```
e1; e2;
```

If temporaries are destroyed at end of statement, this is no longer true.

The choice

None of the alternatives seems to have a conclusive advantage over the others, although some may have conclusive disadvantages. In general, when choosing between something that is easy to use and something that is easy to implement, C++ has tended towards what is easy to use; that tendency should guide our choice here too. With that in mind, let's examine each of the alternatives. **Immediate destruction.** There is no difference between conditional and unconditional immediate destruction. Immediate destruction is probably easiest to define and implement. It is known to cause problems, however, as seen by existing users of existing classes. It therefore seems like a dubious idea to require all other implementers to force their users to confront the same problems.

Unconditional end of statement.

This has the advantage that it would allow things like

```
printf("%s\n", (const char*)(s+t));
```

and also

```
printf("%s\n", e?(const char*)(s+t):"");
```

Moreover, destroying temporaries after each statement would limit the duration of large temporaries in contexts like

```
Matrix m = a + b * c;
```

and would reduce the amount of time that would have to be spent testing flags for temporaries the did not need to be destroyed.

On the other hand, it would not allow either

```
const char* p = s + t;
printf("%s\n", p);
```

or

```
const char* p = 0;
if (e)
        p = s + t;
printf("%s\n", p);
```

which some people might consider to be cleaner ways of writing the previous examples.

Conditional end of statement. This has the advantage of not requiring the implementation to generate and test extra flags, but it breaks

```
printf("%s\n", e?(const char*)(s+t):"");
```

in addition to the examples that fail under unconditional end of statement. It is worth

noting, however, that this example could be written this way:

```
printf("%s\n", (const char*)(e?(s+t):String("")));
```

in which case it would work under conditional or unconditional destruction.

As noted before, destroying conditionally or unconditionally at end of statement introduces for the first time the notion that

```
e1, e2;
```

and

```
e1; e2;
```

are not equivalent. More generally, destroying at end of statement might discourage users from breaking up big statements into smaller ones.

Unconditional end of block. This is the most liberal policy that is at all workable. However, the extra flags involved could potentially be quite expensive at run time. For example:

```
{
        if (e) goto X;
        if (e1) p1 = s1 + t1;
        if (e2) p2 = s2 + t2;
        if (e3) p3 = s3 + t3;
        // ...
X:
        // ...
}
```

Assume here that `e1`, `e2`, and so on are `int`s, `s1`, `s2`, `t1`, `t2`, and so on are `String`s and `p1`, `p2`, and so on are `const char*`. Then each of the `+` expressions requires a temporary. Under unconditional end of block destruction, these temporaries cannot be destroyed until well after label `X`, which means that the compiler must either test one flag for each temporary or do the flow analysis to recognize that if `e` was nonzero when tested, the rest of the flags are unnecessary.

Conditional end of block. The last example illustrated that in the context of destruction at end of block, it is not enough to use "conditional" to refer only to temporaries created in a conditional context within a single expression. Instead, we must interpret "conditional" within the context of the entire block.

If we do that, we come up with a notion with a certain appeal: under conditional end of block destruction, *every temporary must be destroyed as late as possible in the block in which it was created, with the added stipulation that every control path through the destruction point must also pass through the construction point.*

Although this formulation is a useful first approximation, and covers some cases nicely, It turns out that it does not cover all. Looking first at our earlier example:

```
{
        if (e) goto X;
        if (e1) p1 = s1 + t1;
        if (e2) p2 = s2 + t2;
        if (e3) p3 = s3 + t3;
        // ...
X:
        // ...
}
```

we see that each of the temporaries created by `s1+t1`, `s2+t2`, and so on must be destroyed

at the end of the statement that created it, because control after that is merged with the path that would have been followed had the corresponding *en* been zero. In other words, in a statement of the form

```
if (e) s; else t;
```

where `e` is an expression and `s` and `t` are statements, we see that any temporaries created in `s` must be destroyed in `s`, any temporaries created in `t` must be destroyed in `t`, and temporaries created in `e` will generally persist beyond the `if` statement itself.

Where the rule doesn't quite make it, though, is in something like this:

```
if (e) { s1; if (e1) goto x; s2; } else t; u;
```

Does the `goto` statement affect when temporaries from `e` are destroyed? The presence of the `goto` means that it is possible for `e` to be evaluated without also evaluating `u`.

I believe this problem can be solved by defining a class of 'abnormal transfers of control' that would include `return`, `break`, `continue`, `goto`, and `throw`. For each of these transfers, we could define just what its affect would be on destruction of temporaries created in a block containing the transfer. I have not yet attempted to make this rigorous, but am confident that it is possible.

Of course, either conditional or unconditional end of block destruction can potentially lead to consuming extra memory, a problem that some users have noticed.

Summary

This has been an attempt to analyze the advantages and disadvantages of several possible strategies for destruction of temporaries. I am not recommending any particular choice at this time: I once strongly favored immediate destruction and am presently leaning toward conditional end of block, but the issues are still far from clear. This analysis is therefore intended to form a basis for further discussion.