Doc. No.: WG21/N0856=X3J16/96-0038
Date: 30 Jan 1996
Project: C++ Standard Library
Reply to: David Vandevoorde
vandevod@cs.rpi.edu

High-performance C++ implementations for valarrays

Description

It is not known how to implement the valarray template (as currently included in the working paper) with realistic performance using the core language. By realistic performance, I think of no more than 20% increase in run-time and O(1) increase in storage requirements for typical arraywise operations.

Discussion

Todd Veldhuizen and myself have (independently) shown how template techniques can be used to implement valarray-like functionality with low overhead. Todd Veldhuizen (who coined the technique ‘‘expression templates’’) provides a detailed discussion on-line:

    http://monet.uwaterloo.ca/
           ~tveldhui/papers/Expression-Templates/exprtmpl.html

I made an implementation with most of the valarray functionality available at:

    ftp://ftp.cs.rpi.edu/pub/vandevod/Valarray/Rel2_0Beta

We both report performance that is within a few percent of hand-coded C for arrays of sizes larger than about 25. For smaller arrays, the overhead can reduce performance dramatically (especially for 1 or 2 element arrays), but still much less so than alternative techniques. Indeed, while for medium-to-large sized arrays (sizes 1000 and up) expression templates lead to run-times about half those of the known alternative techniques (involving extraneous copying and/or run-time expression analysis), small arrays result in ‘‘order of magnitude’’ speed improvements when using expression templates.

There are various ways to enable expression template techniques. The one proposed here results in minimal changes for the user as to what constitutes a valid ‘‘valarray expression’’ (compared with the current specifications).

This is essentially the change that I suggested in my public comment in July 1995 (now backed with a working implementation).

Proposed Resolution

Change:

```
template<typename T>
class valarray;
```

to:

```
template<typename T, typename M = c_array>
class valarray;
```

with the restriction that users can only directly create valarray objects with M = c_array.

I call M the ‘‘storage model’’.

An implementation is free to return valarrays with other storage models as it sees fit, but must accept valarrays with such alternative storage models wherever valarray<T> arguments are currently expected.

For example, the current addition operator:

```
template<typename T>
valarray<T>
operator+(valarray<T> const&, valarray<T> const&);
```

may become:

```
template<typename T, typename M1, typename M2>
valarray<T, R(T, M1, M2)>
operator+(valarray<T, M1> const&, valarray<T, M2> > const& );
```

where R(T, M1, M2) is a storage model that may depend on T, M1 and/or M2. (It may also simply be c_array). The implementation must accept c_array and any other storage model it may generate as substitutions for M1 and M2.