Document Numbers: X3J16/96-0106
WG21/N0924
Date: May 28, 1996
Reply To: Bill Gibbons
bill@gibbons.org

Some Possible Future Directions for Numeric Traits

Introduction

There are a number of attributes of C++ types which would be useful for some programs, but which are not currently available through the `numeric_limits` template. Although it is probably too late to add any additional templates and/or add new members to `numeric_limits`, it would be useful to collect a list of these attributes for future use. They might be considered during the next revision of the C++ standard, or they might be provided as extensions by some of the C++ compilation system vendors. This paper does not provide an exhaustive list, but suggests possible kinds of attributes which might be made available.

Type Properties

Most of the properties supported by numeric_limits are actually descriptions of the set of values which the type may represent. But there are other properties which are useful by themselves. Expressed in the form of members of numeric_limits, some such properties are:

```
template<class T> class numeric_limits {
    static const bool is_promoted;     // is this a "promoted" type
    typedef int promoted_type;         // the promoted version of this type
    typedef long widest_similar;       // the widest similar type, i.e.
                                       // signed/unsigned or integral/floating
    static const bool is_pointer;      // if this is a pointer type
    static const bool is_arithmetic;   // if this is an arithmetic type
    static const bool is_user_defined; // if this is a user-defined type
    ....many more possibilities ...
}
```

Type Relationships

There are many interesting attributes of the relationships between types which would be useful to query. Here are some examples, expressed using member templates of `numeric_limits`:

```
template<class T> class numeric_limits {
    template<class U> static const bool is_implicitly_convertible;
    template<class U> static const bool is_conversion_widening;
    template<class U> static const bool is_conversion_value_preserving;
};
```

Or these might be better expressed as members of a two-argument traits template:

```
template<class T, class U> class type_pair_attributes {
    static const bool is_implicitly_convertible;
    static const bool is_widening_conversion;
    static const bool is_value_preserving_conversion;
}
```

Type Functions

There is at least one useful piece of information which can thought of as a mapping from a pair of types to another type: the usual arithmetic conversion rules. Given a pair of types, what is the result type for a typical arithmetic operator? This could be expressed with a member typedef template if such things existed; the alternatives are to wrap the typdef in a member class template, or use a two-argument traits template:

```
template<class T> class numeric_limits {
    template<class U> struct arith_conv { typedef int usual_arith_conv_type; };
};
```

or:

```
template<class T, class U> class type_pair_attributes {
    typedef int usual_arith_conv_type;
};
```

The two-argument traits class appears to be the simpler solution.

Here is an example of how the “usual_arith_conv_type” attribute might be used:

```
template<class T, class U>
typename type_pair_attributes<T,U>::usual_arith_conv_type max(T t, U u)
{ return t > u ? t : u }
```

This template behaves exactly like the builtin operators, including such otherwise hard to generalize cases as using `long` with `unsigned int`, where the result type is implementation-dependent.

Relationship to type_info

The information provided by the type_info class is oriented towards dynamic type information, while the traits templates are oriented towards static type information. But it might be worth considering making the numeric_limits class derive from `type_info`:

```
class type_info {
    ....
    virtual bool is_Promoted() const;
};

template<class T> class numeric_limits : type_info {
    ....
    virtual bool is_Promoted() const { return is_promoted; }
    static const bool ix_Promoted;
};
```

It is not clear yet how useful this would be.

Summary

The `numeric_limits` mechanism is useful for extracting type information and attributes from the environment. The technique is especially useful in templates, where little is known about the types when the template is written. There are many opportunities for extending `numeric_limits` and/or adding a two-argument traits template. While it is too late to add these to this standard, these features may show up as vendor-speific extensions or as part of the next standardization effort on C++.