Document Numbers: X3J16/96-0105
WG21/N0923
Date: May 28, 1996
Reply To: Bill Gibbons
bill@gibbons.org

Proposed Changes to the Built-in Operator Pseudo-Prototypes

Introduction

When a class object is used with an overloadable operator, two sets of functions are considered in overload resolution. The first set is the visible user-declared operator functions. The second set is the built-in operators, which are may be callable if the class declares any operator conversion functions. In order to select the right function or built-in operator, overload resolution must work as if the built-in operators were declared as functions - that is, the built-in operators must have prototypes.

When this mechanism was added to the working paper, two methods for deriving the prototypes for builtins were considered. Given the ideas presented at the time, one method appeared to be better and the prototypes derived from that method were added to the working paper.

On further analysis, it appears that this choice may not have been optimal. There are good reasons for reconsidering this decision and possibly updating the set of pseudo-prototypes for the builtin operators.

Operand Conversions

The description of the “usual arithmetic conversions” which apply to most arithmetic operators is given early in clause 5:

> Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:
>
> — If either operand is of type long double, the other shall be converted to long double.
>
> -- etc --

There are two interpretations of this section:

(1) The operators accept the original operands unchanged, and the conversions are done as part of the implementation of the operation itself.

(2) The conversions are done before the operation is begun; the operation itself only sees operands of the same type.

The original choice of prototypes is essentially based on the first interpretation. The proposed revision is essentially based on the second interpretation.

Ambiguities in Built-in Operations

Given the prototypes:

```
int operator+(int, int);  // in practice, six prototypes
long operator+(long, long);
```

the existing overloading rules are not sufficient to handle simple mixed-type cases, for example:

```
void f(int x, long y) {
       x + y;  // ambiguous according to current overload rules
}
```

This was a strong argument for adopting the other approach, where all promoted type combinations are handled:

```
int  operator+(int, int);   // in practice, 36 prototypes
long operator+(int, long);
long operator+(long, int);
long operator+(long, long);
```

This models the builtin operators correctly, at the cost of using a large number of pseudo-prototypes.

Ambiguities in Operator Overloading

However, this “list all variants” approach causes many additional ambiguities when the pseudo-prototypes are used for their intended purpose, namely overload resolution for operators with class object operands. Consider this case:

```
struct A {
    operator int();
    operator long();
};
void f(A a, long x) {
    a + x;
}
```

The two pseudo-prototypes:

```
long operator+(int, long);
long operator+(long, long);
```

are equally good matches for the arguments. So the call is ambiguous. Most of the operator pseudo-prototypes lead to this kind of ambiguity. For example, this case is taken from a well-known C++ library:

```
struct String {
    operator char *();
    char & operator[](unsigned int);
};
void f(String s) {
    s[5];  // ambiguous
    // best match for first arg:    String::operator[](unsigned int)
    // best match for second arg:   operator[](char *, int)
}
```

These ambiguities make it extremely hard to write robust code which uses conversion operators.

A Different Solution

There is another way to make the pseudo-prototypes model the builtin operators; this other approach was not considered when the current approach was adopted.

With a slight modification to the overloading rules, applied only to otherwise ambiguous cases, the builtin operators can be modeled perfectly and the operator overloading ambiguities can be avoided. The complete proposal is in the paper "Improving Overload Resolution for Arithmetic Parameters", paper number 96-0104/N0922. Basically, that paper proposes that ambiguous calls be reconsidered without narrowing conversions and with "less widening" conversions preferred to "more widening" conversions.

Now consider using this overloading rule, together with simplified pseudo-prototypes

```
int operator+(int, int);  // in practice, six prototypes
long operator+(long, long);
```

The original ambiguity is gone:

```
void f(int x, long y) {
       x + y;  // no longer ambiguous
}
```

because on reconsideration the conversion "long => int" is disallowed, making the second form the only viable function. And the conversion operator ambiguity is gone:

```
struct A {
    operator int();
    operator long();
};
void f(A a, long x) {
    a + x;  // no longer ambiguous
}
```

because the mixed-type prototype "`long operator+(int, long)`" is no longer present.

Existing Practice

Since the current rules have been in place for a year or two, one might think that there would be a lot of existing practice which depends on them. Our experience has been just the opposite.

The existing C++ conformance test suites contain tests which are intended to compile, but which are ill-formed under the current rules (but not the proposed revision). At least two commercially available C++ libraries contain code which is ill-formed under the current rules (but not the proposed rules). It appears that existing practice is a weak argument for the existing rules; indeed, it may be a stronger argument for the revised rules.

Proposal

This proposal depends on the adoption of the proposal in 96-0104/N0922. Given those changes, the psuedo-prototype lists are changed to use only the natural types for the operators. Only binary operators are affected. The original and revised prototypes are given.

Original:

For every pair of promoted arithmetic types *L* and *R*, there exist candidate operator functions of the form

```
LR operator*(L, R);
LR operator/(L, R);
LR operator+(L, R);
LR operator-(L, R);
bool operator<(L, R);
bool operator>(L, R);
bool operator<=(L, R);
bool operator>=(L, R);
bool operator==(L, R);
bool operator!=(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

Revised:

For every promoted arithmetic type *T*, there exist candidate operator functions of the form

```
T operator*(T, T);
T operator/(T, T);
T operator+(T, T);
T operator-(T, T);
bool operator<(T, T);
bool operator>(T, T);
bool operator<=(T, T);
bool operator>=(T, T);
bool operator==(T, T);
bool operator!=(T, T);
```

Original:

For every pair of types *T* and *I*, where *T* is a cv-qualified or cv-unqualified complete object type and *I* is a promoted integral type, there exist candidate operator functions of the form

```
T* operator+(T*, I);
T& operator[](T*, I);
T* operator-(T*, I);
T* operator+(I, T*);
T& operator[](I, T*);
```

Revised:

For every type *T*, where *T* is a cv-qualified or cv-unqualified complete object type, there exist candidate operator functions of the form

```
T* operator+(T*, ptrdiff_t);
T& operator[](T*, ptrdiff_t);
T* operator-(T*, ptrdiff_t);
T* operator+(ptrdiff_t, T*);
T& operator[](ptrdiff_t, T*);
```

Original:

For every pair of promoted integral types *L* and *R*, there exist candidate operator functions of the form

```
LR operator%(L, R);
LR operator&(L, R);
LR operator^(L, R);
LR operator|(L, R);
L operator<<(L, R);
L operator>>(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

Revised:

For every promoted integral type *T*, there exist candidate operator functions of the form

```
T operator%(T, T);
T operator&(T, T);
T operator^(T, T);
T operator|(T, T);
```

For every pair of promoted integral types *L* and *R*, there exist candidate operator functions of the form

```
L operator<<(L, R);
L operator>>(L, R);
```

Original:

For every triple (*L*, *VQ*, *R*), where *L* is an arithmetic type, *VQ* is either `volatile` or empty, and *R* is a promoted arithmetic type, there exist candidate operator functions of the form

```
VQ L& operator=(VQ L&, R);
VQ L& operator*=(VQ L&, R);
VQ L& operator/=(VQ L&, R);
VQ L& operator+=(VQ L&, R);
VQ L& operator-=(VQ L&, R);
```

Revised:

For every pair (*T*, *VQ*,), where *T* is an arithmetic type and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator=(VQ T&, T);
VQ T& operator*=(VQ T&, T);
VQ T& operator/=(VQ T&, T);
VQ T& operator+=(VQ T&, T);
VQ T& operator-=(VQ T&, T);
```

Original:

For every triple (*T*, *VQ*, *I*), where *T* is a cv-qualified or cv-unqualified complete object type, *VQ* is either `volatile` or empty, and *I* is a promoted integral type, there exist candidate operator functions of the form

```
T*VQ& operator+=(T*VQ&, I);
```

```
T*VQ& operator-=(T*VQ&, I);
```

Revised:

> For every pair (*T*, *VQ*), where *T* is a cv-qualified or cv-unqualified complete object type and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ& operator+=(T*VQ&, ptrdiff_t);
T*VQ& operator-=(T*VQ&, ptrdiff_t);
```

Original:

> For every triple (*L*, *VQ*, *R*), where *L* is an integral type, *VQ* is either `volatile` or empty, and *R* is a promoted integral type, there exist candidate operator functions of the form

```
VQ L& operator%=(VQ L&, R);
VQ L& operator<<=(VQ L&, R);
VQ L& operator>>=(VQ L&, R);
VQ L& operator&=(VQ L&, R);
VQ L& operator^=(VQ L&, R);
VQ L& operator|=(VQ L&, R);
```

Revised:

> For every pair (*T*, *VQ*), where *T* is an integral type and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator%=(VQ T&, T);
VQ T& operator&=(VQ T&, T);
VQ T& operator^=(VQ T&, T);
VQ T& operator|=(VQ T&, T);
```

> For every triple (*L*, *VQ*, *R*), where *L* is an integral type, *VQ* is either `volatile` or empty, and *R* is a promoted integral type, there exist candidate operator functions of the form

```
VQ L& operator<<=(VQ L&, R);
VQ L& operator>>=(VQ L&, R);
```

Summary

The pseudo-prototypes for the builtin operators do not work very well for their intended purpose, namely overload resolution when considering the use of operator conversion functions on operands of class type. The alternative requires a compatible modification to the overloading rules which is useful by itself, plus a simplification of the list of pseudo-prototypes which makes them more closely model the behavior of the builtin operators themselves.

These changes make numeric classes much more practical; they explain the behavior of the builtin operators using a simple and natural set of prototypes together with the overloading rules exactly as applied to ordinary functions; and they greatly reduce certain surprising ambiguities.

Little if any existing code would be broken by these changes, but future numeric code would be made much cleaner and simpler.