Document Numbers: X3J16/96-0104
WG21/N0922
Date: May 28, 1996
Reply To: Bill Gibbons
bill@gibbons.org

Improving Overload Resolution for Arithmetic Parameters

The Problem

Function overloading does not work very well for arithmetic types.

This may seem surprising, but it has long been known that certain kinds of problems were harder to solve with overloading that one would expect. It now appears that these difficulties were not mere corner cases, but rather indicated a fundamental problem.

Here is a simple example: suppose you want to provide both an integer and floating-point version of some library function, e.g. the “gamma” function. The obvious approach is to provide just three versions:

```
long gamma(long);
unsigned long gamma(unsigned long);
long double gamma(long double);
```

assuming that one of the “long” version will be called for all integral operands, and the “long double” for all floating-point operands. But overloading does not work that way:

```
int f(int x) {
    return gamma(x);  // ambiguous
}
```

This is ambiguous because the conversions “int => long”, “int => unsigned long” and “int => long double” are all standard conversions, and all are equally ranked for overload resolution.

This is not too hard to fix for single-parameter functions; just add a few more versions:

```
int gamma(int);
unsigned int gamma(unsigned int);
long gamma(long);
unsigned long gamma(unsigned long);
double gamma(double);
long double gamma(long double);
```

(Since promotions are preferred to standard conversions, there is no need to provide versions for types which get promoted.)

Providing six versions instead of three could be cumbersome for a large library, but still manageable. Now consider two-parameter functions, such as a simple “min” and “max”. The naive approach:

```
long max(long, long);
unsigned long max(unsigned long, unsigned long);
long double max(long double, long double);
```

fails for the same reason that the three-function version of “gamma” fails. Now extend “max” to six versions:

```
int max(int, int);
unsigned int max(unsigned int, unsigned int);
long max(long, long);
unsigned long max(unsigned long, unsigned long);
float max(float, float);
double max(double, double);
long double max(long double, long double);
```

This still does not work; consider:

```
void f(int x, long y) {
    return max(x, y);  // ambiguous
}
```

This is ambiguous because the “int” version of “max” is the best match for the first argument, and the “long” version is the best match for the second argument. Since both are callable, and each is strictly better for some argument, the call is ambiguous.

The only solution with the current overloading rules is to provide exact matches for all promoted arithmetic types:

```
int max(int,int)
unsigned int max(int,unsigned int)
long max(int,long)
unsigned long max(int,unsigned long)
double max(int,double)
long double max(int, long double)

unsigned int max(unsigned int, int)
unsigned int max(unsigned int, unsigned int)
long max(unsigned int, long)
unsigned long max(unsigned int, unsigned long)
double max(unsigned int, double)
long double max(unsigned int, long double)

long max(long, int)
long max(long, unsigned int)
long max(long, long)
unsigned long max(long, unsigned long)
double max(long, double)
long double max(long, long double)

unsigned long max(unsigned long, int);
unsigned long max(unsigned long, unsigned int);
unsigned long max(unsigned long, long);
unsigned long max(unsigned long, unsigned long);
double max(unsigned long, double);
long double max(unsigned long, long double);

double max(double, int)
double max(double, unsigned int)
double max(double, long)
double max(double, unsigned long)
double max(double, double)
long double max(double, long double)
```

```
long double max(long double, int)
long double max(long double, unsigned int)
long double max(long double, long)
long double max(long double, unsigned long)
long double max(long double, double)
long double max(long double, long double)
```

Now consider a simple function to limit a value between a lower and upper bound:

```
int limit(int lower, int value, int upper);
```

To provide a complete overload set for this function requires 216 versions (not listed here).

The Cause

The underlying problem here is that implicit arithmetic conversions, unlike all other conversions, are reversible. Not only is "int => long" an implicit conversion, but so is "long => int". This makes functions callable which would otherwise be uncallable, and that leads to overloading ambiguities.

Compare arithmetic conversions with cv-qualifier conversions on pointers:

```
void f1(char *, char *);
void f1(const char *, const char *);
void g1(char *x, const char *y) {
    f1(x, y);
}
```

```
void f2(int, int);
void f2(long, long);
void g2(int x, long y) {
    f2(x, y);
}
```

For the call to f1, the first f1 is a better match for the first argument, and the second is a better match for the second argument. But the second f1 is not viable because the conversion "char * => const char *" is not reversible; that is, "const char * => char *" is not an implicit conversion.

In exactly the same manner, in the call to f2, the first f2 is a better match for the first argument and the second is a better match for the second argument. But the second f2 is viable because the conversion "int => long" is reversible; that is, "long => int" is an implicit conversion.

If the call to f1 was ambiguous under the current overloading rules, there would be an incredible push to fix the overloading rules. Yet the ambiguity in the call to f2 is just as bad for numerical programming; we have just learned to live with it.

Possible Solutions

The simplest solution would be to remove the implicit "narrowing" conversions such as "long => int". But this would break a lot of existing code, so this option is not really viable.

The alternatives involve some kind of change to the overload resolution rules. There are two kinds of changes:

(1) Modify the rules for comparing standard conversions during overload resolution, so that narrowing conversions are less preferred.

(2) Reconsider a call found to be ambiguous under the existing rules, with some modified form of the overloading rules used for the reconsideration.

While (1) might be simpler, it seems dangerous to make such a change this late in the standardization process. But (2) has no effect at all on the meaning of any currently well-formed program, and so it is considerably less risky.

There are two possible modifications to the overloading rules which would improve matching of arithmetic parameters during reconsideration:

— Do not consider implicit narrowing conversions.

— Prefer less-widening conversions to more-widening conversions.

The first change is necessary to prevent spurious ambiguities, as shown in the examples. There has also been some support in the past for deprecating implicit narrowing conversions everywhere in C++; if that is the eventual direction of the language, this change would be a very natural anticipation of the larger change.

The second change is not as important, but it also helps resolve ambiguities; for example, in the gamma function example given earlier. The working paper already supports a subset of this rule: conversions from short integral types to "int" or "long" prefer "int" because it is a promotion.

Also, consider the problems of extending the language with "long long". Now an additional kind of ambiguity is possible:

```
long f(long);
long long f(long long);
void g(int x) {
    f(x);  // ambiguous
}
```

While this problem already exists with functions such as "gamma" above, it is likely to become much worse as "long long" becomes more common. It would be nice if the C++ overloading rules were already sufficient to resolve such cases.

The working paper almost (but not quite) already contains the "prefer less widening" rule. Consider this rule from [over.ics.rank]:

> Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules apply:
> — Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if
> — S1 is a proper subsequence of S2, or, if not that, ...

If we consider the "gamma" example again, we have two conversions, neither of which is currently preferred: "int => long" and "int => long double". This look a lot like a conversion sequence:

```
struct A { };
struct B : A { };
struct C : B { };
void f(A *);
void f(B *);
void g(C *c) {
    f(c);  // OK
}
```

```
void f(long double);
void f(long);
void g(int i) {
    f(i); // currently ambiguous
}
```

The pointer example is unambiguous because of the “proper subsequence” rule in [over.ics.rank]. The arithmetic example is conceptually very similar to the pointer example, and it should work for the same reasons.

Proposal

I propose changing the overloading rules *in a way which only affects currently ill-formed programs*.

When overload resolution fails because of an ambiguity, it is reconsidered ignoring narrowing conversions and giving preference to less-widening conversions over more-widening conversions. The working paper changes are:

To the general description of overloading in [over.match], which currently reads:

— First, a subset of the candidate functions—those that have the proper number of arguments and meet certain other conditions—is selected to form a set of viable functions (13.3.2).
— Then the best viable function is selected based on the implicit conversion sequences (13.3.3.1) needed to match each argument to the corresponding parameter of each viable function.

add the additional item:

— If there is more than one viable function, but no best function, the selection is repeated using the arithmetic widening rules in [xxx].

Add the following section after [over.ics.rank]:

[xxx] Reconsidering arithmetic conversions

When overload resolution is reconsidered because the overall overloading process did not yield a unique best function [over.match], certain implicit conversion sequences are treated differently.

An implicit conversion between two arithmetic, non-enumeration types is considered a *widening* conversion if it is one of the those performed by the “usual arithmetic conversions” [expr], including promotion. The identity conversion is not considered widening.

An implicit conversion between two arithmetic, non-enumeration types is considered a *narrowing* conversion if it is the inverse of a widening conversions. [Note - some conversions are neither widening nor narrowing. - End Note]

During reconsideration, narrowing conversions are ignored. Given two widening conversions “A => B” and “A => C”, if “B => C” is a widening conversion then the first conversion is considered better. Given two widening conversions “A => C” and “B => C”, if “A => B” is a widening conversion then the second conversion is considered better.

List of Widening Conversions

The description of the “usual arithmetic conversions” implies the following possible conversions which might be done on an operand. These are the widening conversions. This list is provided for clarity only;

the list can be derived from the rules for the “usual arithmetic conversions”.

Original Type	Converted Type
bool, char, signed char, unsigned char, short, unsigned short	int, unsigned int (only if value preserving) long, unsigned long, float, double, long double
wchar_t	int, unsigned int, long (only if value preserving) unsigned long
int	unsigned int, long, unsigned long, float, double, long double
unsigned int	long (only if value preserving) unsigned long, float, double, long double
long	unsigned long, float, double, long double
unsigned long	float, double, long double
float	double, long double
double	long double

Where a “value preserving” conversion is one where the result type can represent all of the values which the original type can represent. Note that not all widening conversions are value preserving. This is a consequence of the “usual arithmetic conversion” rules.

Summary

Overload resolution does not adequately resolve ambiguities on arithmetic parameters, even when the best choice is intuitively easy to find. The problem is the bidirectional nature of implicit arithmetic conversions, which allows functions to remain viable when, in similar cases involving non-arithmetic conversions, such functions would be considered nonviable.

This late in the standardization process, any solution to the problem must have minimal impact. The proposed solution has no effect on programs which are well-formed under the existing overloading rules, and so does not break any existing valid code. It does make a large number of currently ill-formed, but very reasonable, programs become well-formed.