X3J16/94-0143
WG21/N0530

`const` as a Tie-breaker in Overload Resolution

J. Stephen Adamczyk
Edison Design Group, Inc.
jsa@edg.com

July 27, 1994

Introduction

At the Waterloo meeting, Tom Plum's core working group, and later the full committee, briefly discussed the role of `const` et al. as "tie-breakers" in overload resolution. It became clear a short time into the full-committee discussion that there was no consensus on the issue and that additional exposition and discussion were required. The issue was pulled back into the working group for further study. This paper is a summary of the issue to assist such study.

The ARM text

The concept of a tie-breaker in overload resolution is introduced in ARM 13.2:

> Note that functions with arguments of type `T`, `const T`, `volatile T`, `T&`, `const T&`, and `volatile T&` accept exactly the same set of values. Where necessary, `const` and `volatile` are used as tie-breakers as described in rule [1] below.

A few lines later, we find rule [1]:

> Exact match: Sequences of zero or more trivial conversions are better than all other sequences. Of these, those that do not convert `T*` to `const T*`, `T*` to `volatile T*`, `T&` to `const T&`, or `T&` to `volatile T&` are better than those that do.

For example:

```
struct A {};
void f(A*);
void f(const A*);
main ()
{
  A a;
  f(&a);  // Calls f(A*)
}
```

The function `f(A*)` is called in preference to `f(const A*)` because the conversion of the argument for the latter involves one of the tie-breaker conversions.

A couple of points to note:

- Even though this paper's title refers to `const` as a tie-breaker, that's just to have a punchy name for the issue. Clearly `volatile` is also a tie-breaker.
- The conversions of interest are ones that add a type qualifier *under a pointer or reference*. A conversion from "pointer to `int`" to "pointer to `const int`", for example, is a tie-breaker, but one from "pointer to `int`" to "`const` pointer to `int`" is not, nor is one from "`int`" to "`const int`".

What does "tie-breaker" mean?

The problem with the ARM wording is that it doesn't make it clear how the tie-breaker cases affect the process of selecting a "best" function in overload resolution. There are two common interpretations:

1. A match on a particular argument involving a tie-breaker conversion is worse than an otherwise equally-ranked match that does not involve such a conversion ("early tie-breaker").
2. If two function calls are equal on all arguments, ignoring any tie-breakers, the presence of one of these tie-breaker conversions [1] can be used to pick one function match over the other ("late tie-breaker").

An example to illustrate the difference:

```
struct A {};
void f(const A*, short);
void f(A*, int);
main ()
{
  A a;
  f(&a, (short)1);  // ?
}
```

The overload resolution analysis gives:

	`&a`	`(short)1`
`f(const A*, short)`	exact (*)	exact
`f(A*, int)`	exact	promotion

where the conversion marked with "(*)" is a tie-breaker. Under the "early tie-breaker" rule, the second function is the best match on the first argument, and the first function is the best match on the second argument, so the call is ambiguous. Under the "late tie-breaker" rule, the two functions are equally good when tie-breakers are ignored, so the tie-breaker is used to select the second function as the best one.

[1] If there are tie-breakers on several arguments, they must be consistent; otherwise, the two function calls remain equally ranked.

Existing practice

Unfortunately, existing practice varies. cfront, Microsoft Visual C++, Symantec, g++, EDG, and the IBM OS/2 compiler seem to implement the late tie-breaker. Borland, Sun, Lucid, and Taligent seem to implement the early tie-breaker. (I'm doing those from memory for the compilers I don't have access to, so forgive me if I don't have it quite right; the point is that existing practice shows no consensus.)

The argument for each approach

Arguments for an early tie-breaker:

- It's the simplest rule to describe and teach.
- It makes the bizarre cases where this matters ambiguous, which is probably a favor to the programmer.
- With the recent rewrite of Clause 13, it's the status quo in the Working Paper.

Arguments for a late tie-breaker:

- It's what cfront implements, and there are known to be large libraries out there that would be "broken" by a change to an early tie-breaker. (There may also be libraries that would be broken by a change in the other direction; I can comment only on the bug reports I've seen.)
- According to Bjarne Stroustrup, it's what he intended when he wrote the ARM.
- Programmers are likely to think of the difference between `const` and non-`const` versions of a member function as related to applicability rather than as a "cost." That is, the programmer thinks "okay, if I want to be able to handle `const` objects, I need a `const` version of the member function. If I need to do different things for `const` and non-`const` objects, I need two versions of the member function. But what about if I can do the same thing in both cases, and I don't need to modify the object? I just need a `const` version, right?" The problem with that reasoning when an early tie-breaker is used by the compiler is that any call of that member function with a non-`const` object has a slight additional cost, which can cause ambiguities. Consider

  ```
  struct B {
    B(int);
  };
  struct A {
    void f(int) const;
    void f(B);
  };
  main () {
    A a;
    a.f(1);  // ?
  }
  ```

The overload resolution analysis gives:

	`a`	`1`
`f(int) const`	exact (*)	exact
`f(B)`	exact	user-defined conv

With an early tie-breaker, this example is ambiguous. The programmer may be surprised that the "obviously" cheaper `f(int) const` is not chosen.

Aspects that aren't contentious

Several aspects that might not be entirely clear in the ARM wording nevertheless seem not to be contentious:

- The rules for pointers and references are the same:

  ```
  struct A {};
  void f(const A&, short);
  void f(A&, int);
  main ()
  {
    A a;
    f(a, (short)1);  // Same as simple pointer case
  }
  ```

  This is true in spite of the fact that tie-breaker conversions to reference types must now be described as reference initializations that add type qualifiers (that's just a difference in description).

- Whatever rule applies to pointer and reference arguments also applies when binding the object on which a member function is called:

  ```
  struct A {
    void f(short) const;
    void f(int);
  };
  main ()
  {
    A a;
    a.f((short)1);  // Same as simple pointer case
  }
  ```

- Whatever rule applies for exact matches also applies for other kinds of matches:

  ```
  struct A {};
  struct B : public A {};
  void f(const A*, short);
  void f(A*, int);
  main ()
  {
    B b;
    f(&b, (short)1);  // Same as simple pointer case
  }
  ```

  (In this case, the conversion on the first argument is a standard conversion, with addition of a `const` for one of the functions.)

- The tie-breaker conversions should actually include all conversions (or reference bindings) that add type qualifiers, not just the ones listed in the ARM text. For example, `const int*` to `const volatile int*` should be a tie-breaker conversion. Likewise, conversions that add type qualifiers at levels other than the first for multi-level pointers (using the recent extension in that area) should also be considered tie-breaker conversions (e.g., `int **` to `const int *const *`).

- The conversions that add type qualifiers under pointers-to-members (e.g., `int A::*` to `const A::*`) should also be considered tie-breakers.

Summary

I think both the early and late tie-breaker approaches have merit; we should just pick one and be done with it.

I also recommend that we affirm the resolutions on the "non-contentious" issues and incorporate those into the WP, assuming there is in fact consensus on them.