WG14/N307
X3J11/93-054

Why Infix Relational Operators

X3J11/???, WG14/???

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

This is a summary of arguments for infix relational operators as proposed in "Floating-Point C Extensions" (X3J11.1/93-028).

The primary goals of the proposal for infix relational operators are (1) to meet the need to modify programs and to write new programs that deal with NaNs and (2) to cause minimal perturbation to the C language and libraries.

The IEEE floating-point standards specify NaNs as a way of dealing with domain errors—other than the old alternatives of preflighting or crashing. (Domain errors are not an invention of the IEEE standard.) Some codes, for example solvers, written specially for IEEE systems directly exploit NaNs for robustness and efficiency. More generally, robust functions written for (or ported to) IEEE machines will be expected to deal with invalid and NaN input in a predictable way consistent with built-in IEEE operators. Thus programmers' consideration of NaNs will be somewhat common.

The intention of the IEEE standard is that NaNs should pass through many calculations without requiring special code. Hence the ordinary infix arithmetic operators (+, *, etc.) propagate NaNs in a predictable fashion. The case of NaN comparisons is slightly more complicated but the approach should be consistent: the programmer needs to be able to direct NaNs through branches, of which there are many, without having to resort to awkward or inefficient code. Preferably code would not be obfuscated to handle NaNs, which, though useful, are not usually the most interesting aspects of the code.

The current proposal extends the allowable combinations of ordering symbols to include `<>=` and extends the notion of negated operators, previously only `!=`, to the rest of the comparison operators, `!<`, etc. With the understanding that `!` indicates an awareness of the unordered case (hence requires no exception) this small extension handles the entirety of floating-point representations, runs unexceptional old code as before, supports exceptional old code by raising exceptions where crashes might be anticipated, and provides exception free comparisons for new code. On systems without NaNs the new operators can be viewed as notational alternatives (like `!=`). All new operators have the precedence of relational operators. They have been implemented without complication in one shipping compiler (Zortech C++) and in others under development.

The full set of comparison operators have further advantages for C++, where they provide a general facility for partially ordered sets (subsets, substrings, outline items, etc.). As infix operators they all have the same style and overloading properties.

The approach of using an explicit symbol for unordered (provided an appropriate one exists—`?` and `@` seem not to work), instead of the negation (`!`) notion, would be equally

163

WG14/N307
X3J11/93-054

good in most respects. It would be more direct, but would require more explanation on systems without NaNs.

A Boolean macro such as

```
isrelation(x, FPUNORD | FPLESS | FPEQUAL, y)
```

instead of `x !> y` would provide equivalent functionality without change to the language, but would be arguably more awkward, would still require compiler changes if it were to be efficient, would weaken programmer expectations of efficiency, and would intrude on name space. Other function/macro approaches are less conducive to efficient code.