Type-Generic Math Functions

WG14/N693 J11/97-056 (Editorial Review Draft 1997-05-23)

Jim Thomas
`jthomas@best.com`

Synopsis

Type-generic macros allow calling a function whose type is determined by the argument type, as is the case for C operators such as `+` and `*`. For example, with a type-generic `cos` macro, the expression `cos((float)x)` will have type `float`. This feature enables writing more portably efficient code, and alleviates need for awkward casting and suffixing in the process of porting or adjusting precision. This is a proposal for type-generic macros for most of the functions in `<math.h>` and `<complex.h>`. The proposed header `<tgmath.h>` includes the headers `<math.h>` and `<complex.h>` and extends them by defining the type-generic macros.

Background

As recent as December 1995, the floating-point and complex arithmetic specifications included type-generic macros, under the name of overloading macros. Then, in response to the wishes of the C committees, these specifications were redrafted removing the overloading macros, with the intention of later considering them as a separate feature.

At the Amsterdam meeting, the committees accepted in principle the floating-point and complex arithmetic specifications (sans overloading) into the C9X draft; and they requested a newly written proposal for the extracted facility, but under a different name than overloading. Such a proposal (N618) was presented as a Monday paper and discussed in detail at the Toronto meeting. A revision (N639) was discussed further in Kona and accepted in principle into the C9X draft. This document incorporates changes agreed upon in Kona.

The scheme for providing two styles of nextafter functions has been simplified in this draft—a change not discussed at Kona.

Proposal

Rationale given here supplements the rationale in the earlier specification for overloading, and is not intended as a complete rationale for the feature.

NOTE TO EDITOR: RATIONALE ITEMS (IN SQUARE BRACKETS []) ARE NOT TO BE INCLUDED IN THE C9X DRAFT.

In 7, insert this subclause:

7.x Type-generic math <tgmath.h>

The header `<tgmath.h>` includes the headers `<math.h>` and `<complex.h>` and defines several *type-generic macros*.

[The term *type-generic* was chosen as more specific than the suggested term *intrinsic*, which already is widely used with a more general meaning. The X3J11 Vocabulary Representative concurs with this choice.

The macros are placed in their own header in order not to silently break old programs that include `<math.h>`, for example with `printf("%e", sin(x));` .

The name `tgmath.h` is shorter than the previous `generic_math.h`, which the committee thought was too long.]

7.x.1 Type-generic macros

Of the `<math.h>` and `<complex.h>` functions without an `f` (`float`) or `l` (`long double`) suffix, several have one or more parameters whose corresponding real type is `double`. For each such function, except `modf`, there is a corresponding *type-generic macro*[1]. The parameters whose corresponding real type is `double` in the function synopsis are *generic parameters*. Use of the macro invokes a function whose corresponding real type and type-domain are determined by the arguments for the generic parameters.[2]

[`modf(double, double*)` is excluded because no way was seen to make it safe without complicating the type resolution.]

Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

First, if any argument for generic parameters has type `long double`, the type determined is `long double`.

Otherwise, if any argument for generic parameters has type `double` or is of integral type, the type determined is `double`.

Otherwise, the type determined is `float`.

[This differs from the earlier overloading proposal in that the type is determined solely by the argument, and may be narrower than the type for expression evaluation. This change was made because the performance costs for computing functions with narrow arguments to wide range and precision might be too high, even if the implementation efficiently evaluates basic operations to wide precision.

Also, this differs from earlier proposals in that integral-type arguments are converted to (at least) `double` instead of `float`. Although converting to `float` would have been more consistent with the usual arithmetic conversions, converting to `double` has the advantages of preserving the value more often on many systems, and of being more compatible with C90 where unsuffixed calls to math functions with integer arguments were calls to `double` functions.]

For each unsuffixed function in `<math.h>` for which there is a function in `<complex.h>` with the same name except for a `c` prefix, the corresponding type-generic

---

[1] Like other function-like macros in the Standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

[2] If the type of the argument is incompatible with the type of the parameter for the selected function, the behavior is undefined.

macro (for both functions) has the same name as the function in **<math.h>**. The corresponding type-generic macro for **fabs** and **cabs** is **fabs**.

<math.h> function	**<complex.h>** function	type-generic macro
acos	**cacos**	**acos**
asin	**casin**	**asin**
atan	**catan**	**atan**
acosh	**cacosh**	**acosh**
asinh	**casinh**	**asinh**
atanh	**catanh**	**atanh**
cos	**ccos**	**cos**
sin	**csin**	**sin**
tan	**ctan**	**tan**
cosh	**ccosh**	**cosh**
sinh	**csinh**	**sinh**
tanh	**ctanh**	**tanh**
exp	**cexp**	**exp**
log	**clog**	**log**
pow	**cpow**	**pow**
sqrt	**csqrt**	**sqrt**
fabs	**cabs**	**fabs**

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

[Having a **g** suffix for the generic macros was considered but thought unnecessary.]

For each unsuffixed function in **<math.h>** without a **c**-prefixed counterpart in **<complex.h>**, the corresponding type-generic macro has the same name as the function. These type-generic macros are

atan2	**hypot**	**lround**
exp2	**erf**	**llround**
expm1	**erfc**	**trunc**
frexp	**gamma**	**remainder**
ldexp	**lgamma**	**remquo**
log10	**ceil**	**copysign**
log1p	**floor**	**nextafter**
log2	**fmod**	**nextafterx**
logb	**nearbyint**	**fdim**
ilogb	**rint**	**fmax**
scalb	**lrint**	**fmin**
scalbn	**llrint**	
cbrt	**round**	

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

[The implementation might, as an extension, endow appropriate ones of these macros with the ability to invoke the complex functions whose names are reserved in 7.14.4.]

For each unsuffixed function in **<complex.h>** that is not a **c**-prefixed counterpart to a function in **<math.h>**, the corresponding type-generic macro has the same name as the function. These type-generic macros are

```
carg        cimag        creal
conj        cproj
```

Use of the macro with any real or complex argument invokes a complex function.

Examples

With the declarations

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

functions invoked by use of type-generic macros are shown in the table:

macro use	invokes
`exp(n)`	`exp(n)`, the function
`acosh(f)`	`acoshf(f)`
`sin(d)`	`sin(d)`, the function
`atan(ld)`	`atanl(ld)`
`log(fc)`	`clogf(fc)`
`sqrt(dc)`	`csqrt(dc)`
`pow(ldc, f)`	`cpowl(ldc, f)`
`remainder(n, n)`	`remainder(n, n)`, the function
`nextafter(d, f)`	`nextafter(d, f)`, the function
`nextafterx(f, ld)`	`nextafterxf(f, ld)`
`copysign(n, ld)`	`copysignl(n, ld)`
`ceil(fc)`	undefined behavior
`rint(dc)`	undefined behavior
`fmax(ldc, ld)`	undefined behavior
`carg(n)`	`carg(n)`, the function
`cproj(f)`	`cprojf(f)`
`creal(d)`	`creal(d)`, the function
`cimag(ld)`	`cimagl(ld)`
`cabs(fc)`	`cabsf(fc)`
`carg(dc)`	`carg(dc)`, the function
`cproj(ldc)`	`cprojl(ldc)`

Add to subclause 7:

7.x.11.y The `nextafterx` function

Synopsis

```
#include <math.h>
double nextafterx(double x, long double y);
```

Description

The `nextafterx` function is equivalent to the `nextafter` function except that the second parameter has type `long double`.[3]

> [The previous draft specified renaming the `double` version of `nextafter` to `nextafterd`, retaining `nextafterf` and `nextafterl`, and using the name `nextafter` for what is here named `nextafterx`. The specification here has a number of advantages:
>
> 1. `nextafter` and `nextafterx` conform to the usual rules for suffixes and type-generic macros. Before, both `nextafterd` and `nextafter` were exceptional on both counts.
> 2. Without the change, `nextafterf` is not the float version of `nextafter`, which is potentially surprising.
> 3. It better matches prior art, which typically has a `nextafter` function with two `double` parameters.
> 4. It avoids changing the name in the current C9X draft from `nextafter` to `nextafterd`.
> 5. Without this change, we would have had to modify the description for the `nextafter` functions from "where `x` and `y` are first converted to the type of the function" to "where the arguments are first converted to the types of the corresponding parameters".]

In the Annex G: IEC 559-Compatible Complex Arithmetic, append the subclause:

G.7 `<tgmath.h>`

Type-generic macros that accept complex arguments also accept imaginary arguments. If an argument is imaginary, the macro expands to an expression whose type is real, imaginary, or complex, as appropriate for the particular function: if the argument is imaginary, then the types of `cos`, `cosh`, `fabs`, `carg`, `cimag`, and `creal` are real; the types of `sin`, `tan`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, and `atanh` are imaginary; and the types of the others are complex.

Given an imaginary argument, each of the type-generic macros `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, and `atanh` is specified by a formula in terms of real functions:

NOTE TO EDITOR: USE FORMULA STYLE FOR THE FOLLOWING.

cos(i*y)	=	cosh(y)	tanh(i*y)	=	i*tan(y)
sin(i*y)	=	i*sinh(y)	asin(i*y)	=	i*asinh(y)
tan(i*y)	=	i*tanh(y)	atan(i*y)	=	i*atanh(y)
cosh(i*y)	=	cos(y)	asinh(i*y)	=	i*asin(y)
sinh(i*y)	=	i*sin(y)	atanh(i*y)	=	i*atan(y)

[3] The result of the `nextafterx` function is determined in the type of the function, without loss of range or precision in a floating second argument.