

Re: Implementing a templated "round" function?

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.cpp/2004-09/3809.html

From: Siemel Naran (*SiemelNaran_at_REMOVE.att.net*)

Date: 09/30/04

Date: Thu, 30 Sep 2004 08:56:02 GMT

"Jef Driesen" <jefdriesen@nospam.hotmail.com> wrote in message
news:cjgg07\$cvj\$1@ikaria.belnet.be...

> *I need to implement a function to implement the rounding of floating point*
> *values. At the moment i have two different implementations, depending on*
> *the*
> *type of the return value (integer or double).*
>
> *// Integer calculation (fast)*
> *int iround(const double x) {*
> *return (x>=0 ? static_cast<int>(x + 0.5) : static_cast<int>(x + 0.5))*
> *}*
> *// Floating point calculation (slow)*
> *double dround(const double x) {*
> *return (x>=0 ? floor(x + 0.5) : ceil(x + 0.5))*
> *}*

The first function looks suspect because it loses precision. Anyway, if the true and false clauses of operator ? are the same, you can avoid the comparison.

Anyway, have you measured the time difference?

> *But then I have 2 different functions for the same functionality. To solve*
> *this, I created a template function with 2 specialisations:*
>
> *template <typename T> T round(const double x);*
> *template <>*
> *int round(const double x) {*
> *return iround(x);*
> *}*

Should that be

```
template <>
int round<int>(const double x) {
```

comp.lang.c++. Re: Implementing a templated "round" function?

```
> template <>
> double round<double>(const double x) {
> return dround(x);
> }
>
> I can add specialisations for the other data types (float, long, short,
> char, signed/unsigned,...) as well. I did not add a 'default'
> specialisation, because the round function is only usefull for numeric
> types. At this time my function accepts only double precision floating
point
> values, and I would like to have equivalent functions for float, double
and
> long double. I did add a second template parameter S for the source type:
>
> template <typename T, typename S>
> T round(const S x) {
> ...
> }
>
> The problem is now how do I implement this function?
>
> I could add a specialisation for every combination of S and T, but this is
a
> lot of work, and there are only 4 usefull implementations possible:
> - S and T are floating point type -> implementation using floor/ceil
> (dround)
> - S is a floating point type and T an integer type -> implementation using
> static_cast (iround)
> - S is an integer type and T is an integer or floating point type ->
simply
> static_cast to T
> - otherwise no implementation is necessary (should not compile if this is
> possible, to prevent e.g. round<double>(std::complex) )
>
> I was thinking of using only one 'default' specialisation and using
> std::numeric_limits:
>
> template <typename T, typename S>
> T round(const S x)
> {
> if (std::numeric_limits<S>::is_integer)
> // Integer type needs no rounding.
> return static_cast<T>(x);
>
> // Find rounding error.
> const S round_error = std::numeric_limits<S>::round_error();
>
> if (std::numeric_limits<T>::is_integer) {
> // Integer calculation (fastest).
> return (x>=0 ? static_cast<T>(x + round_error) : static_cast<T>(x +
> round_error)
```

Re: Implementing a templated "round" function?

comp.lang.c++. Re: Implementing a templated "round" function?

```
> } else {  
> // Floating point calculation (slower).  
> return (x>=0 ? floor(x + round_error) : ceil(x + round_error))  
> }  
> }  
>  
> But this will results in a performance degradation if the compiler is  
unable  
> to optimize (eliminate) the 'if' statements. And the function has also an  
> implementation for data types other then integer and floating point types.  
> Any advice on this problem?
```

It's worthwhile to check the assembly to see if your compiler does the optimization.

Anyway, you can put the `numeric_limits::is_integer` into the function or class template argument list. Here is the idea:

```
template <typename T, typename S>  
inline  
T round(const S x) {  
    return generic_round<T, S, T::is_integer>(x);  
}
```

```
template <typename T, typename S, bool is_integer>  
inline  
T generic_round(const S x);
```

```
template <typename T, typename S>  
inline  
T generic_round<T, S, true>(const S x) {  
    const S round_error = std::numeric_limits<S>::round_error();  
    return T(x + round_error);  
}
```

etc