# Proposal to add an absolute difference function to the C++ Standard Library

Document number: N4318
Date: 2014-09-21
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Jeremy Turnbull <jeremy8258@gmail.com>

## Table of Contents

## I. Introduction

This document proposes the addition of the `abs_diff()` template function to the C++ Standard Library. This function computes the absolute difference between two parameters of a type that supports the `operator<()` and `operator-()` functions, or other functions of equivalent logic, without computing a logically negative value during function execution.

## II. Motivation and Scope

### *Why is this important? What kinds of problems does it address?*

The `std::abs()` function takes only one parameter. When using the `std::abs()` function, computations such as `abs(a-b)` are common. However, because function arguments are fully evaluated before being passed to the function, such an operation could lead to problems when `a<b` and `a` and `b` are of a type where a logically negative value would throw an exception, or worse still, not throw an exception but still be a logic error.

As an example, for fundamental unsigned integer types, the ISO C standard defines a negative result as:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. (ISO/IEC 9899:2011 §6.2.5/9)

As a demonstration, the author wrote this simple program:

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  unsigned int three = 3;
  unsigned int five = 5;
  cout << "The difference between three and five is ";
  cout << three - five << endl;
  cout << "The absolute value of that difference is ";
  cout << abs( three - five ) << endl;
  return 0;
}
```

the output of which is

```
The difference between three and five is 4294967294
The absolute value of that difference is 4.29497e+09
```

As can be seen, per the standard the negative result wraps around from the maximum positive value for the simple minus operation, and in what may be an implementation detail, is converted to a real type when passed to the `std::abs()` function. Both of these solutions seem like an excessive workaround in the context of both of the `std::abs()` and `operation-()` functions when the problem can easily be avoided.

Of course, user-defined types do not benefit from this implicit type conversion. Such types would instead benefit from the proposed function by avoiding a logically negative result during the computation.

In the proposed function, the result of `abs_diff(3u,5u)` is `2`.

## *What is the intended user community? What level of programmers (novice, experienced, expert) is it intended to support?*

As the proposed function supports both fundamental and user-defined types, the intended user community is the entire C++ community. The proposed function can easily be understood by and support all of novice, experienced, and expert C++ programmers.

## *What existing practice is it based on? How widespread is its use? How long has it been in use? Is there a reference implementation and test suite available for inspection?*

Both the function interface and implementation are based on current C++ STL standard practices. Regarding the function's current use, this author was surprised *not* to find from Internet searches on either Google or stackoverflow.com as elegant of a solution as the one proposed herein, despite it being a common problem. Current forum answers depend on architecture-specific details, and most use bit shifting.

A reference implementation can be found in the *Technical Specifications* section of this document.

## III. Impact On the Standard

### *What other library components does does it depend on, and what depends on it?*

The implementation depends on only that `operator<()` and `operator-()` (or functions of equivalent logic) be defined for the given template parameter type. Therefore, the proposed function does not depend on any other library components. Similarly, because the proposed function would be an addition to the standard library, no other library components currently depend on it.

### *Is it a pure extension, or does it require changes to standard components?*

The proposed function *could* be implemented in a separate header. However, given that it is just one function (and its overloads), it makes more sense to this author to include the function in one of the standard library headers, most likely either <algorithm> or <utility>.

### *Can it be implemented using C++11 compilers and libraries, or does it require language or library features that are not part of C++11?*

Yes, the proposed function can be fully implemented using current C++11 compilers and libraries.

## IV. Design Decisions

### *Why did you choose the specific design that you did? What alternatives did you consider, and what are the trade offs?*

Both the function interface and implementation are based on C++ STL standard practices, including using `operator<()` as the standard comparison operator. Regarding the interface, the only alternative this author considered was to return a type of `const T&` value rather than `T`. However, as minus operations often result in a new value (as opposed to returning an existing value in a range), returning a value rather than a reference to a value is the correct choice. Move semantics should be defined by the type implementer, not the proposed function implementer.

### *What are the consequences of your choice, for users and implementers?*

Considering the proposed function uses both template parameters and overloads, users are given both simplicity and flexibility. Simplicity stems from the fact that the proposed function can be used with any type already defined for the user. Furthermore, because this is a function template, the compiler can automatically determine the type of the function parameters, meaning the proposed can be invoked with either normal function syntax or template function syntax. Users are given flexibility in that the function can be used with any type that supports `operator<()` and `operator-()`. Alternatively, if the implementer of a user-defined type has not defined these operator overloads but instead supports equivalent functionality in other named functions, member functions, or function objects, the proposed function can use these alternatives.

Given, however, that the proposed function returns the result by-value, implementers of user-defined types would greatly benefit from defining a move constructor and move assignment operator for their type.

### *What decisions are left up to implementers? If there are any similar libraries in use, how do their design decisions compare to yours?*

First, implementers are free to choose whether to declare the proposed function and its overloads as `inline`. Second, implementers may define all three signatures (as below), or they may define only one function with default values for the `comp` and `diff` parameters. Third, implementers are free to define the function body in any way that is functionally equivalent to

```
return (a<b)? b-a: a-b;
```

Finally, because the design of the proposed function is based on existing functions in the standard library, both the interface and implementation are quite similar to those functions.

## V. Technical Specifications

*(1)*

```
template <typename T>
  decltype(auto)
  std::abs_diff( const T& a, const T& b )
    { if (a<b) return b-a; return a-b; }
```

*(2)*

```
template <typename T, typename Compare>
  decltype(auto)
  std::abs_diff( const T& a, const T& b, const Compare& comp )
    { if (comp(a,b)) return b-a; return a-b; }
```

*(3)*

```
template <typename T, typename Compare, typename Difference>
  decltype(auto)
  std::abs_diff( const T& a, const T& b,
                 const Compare& comp, const Difference& diff )
    { if (comp(a,b)) return diff(b,a); return diff(a,b); }
```

### Absolute difference

Calculates the absolute value of the difference between two elements in such a way that a negative value will never occur during the calculation.

## Template parameters

`T`

Type of the arguments of the function call.
For (1), the type shall support the operations (binary `operator<` and binary `operator-`).
For (2), the type shall support the operation (binary `operator-`).

`Compare`

Type of the comparison object or function.

`Difference`

Type of the difference object or function.

## Parameters

`a, b`

Two variables or objects of the same type.

`comp`

Comparison object.
Binary function that accepts two values of type `T` as arguments and returns a value of or convertible to `bool`. The value returned indicates whether the element passed as the first argument is considered less than the second.
The function shall not modify any of its arguments.
This can either be a function pointer or a function object.

`diff`

Difference object.
Binary function that accepts two values of type `T` as arguments and returns a value of or convertible to type `T`. The value returned indicates the difference of the second argument removed from the first.
The function shall not modify any of its arguments.
This can either be a function pointer or a function object.

## Return value

The absolute value of the difference of the two parameters `a` and `b`.

## Complexity

Constant, assuming that the comparison and difference computations, and constructors of type `T` are also constant.

## Exceptions

Throws if either the comparison, difference operation, or copy or move constructor of type `T` throws.

## VI. Acknowledgements

The author wishes to thank cplusplus.com and stackoverflow.com for their concise yet complete technical explanations, well designed Web sites, and knowledgeable and mature user communities.