Allow calling overload sets containing T, const T&

Document: P2665R0

Date: 2022-10-12

Project: Programming language C++

Audience: EWG(I)

Reply-to: Bengt Gustafsson, bengt.gustafsson@beamways.com

Introduction

This proposal allows overload sets containing parameters different only by being *by value* and *by const reference* to be called. Which overload to call is selected freely by the compiler, the intention is that this is done to optimize efficiency.

The proposal also includes the parallel cases for overload sets containing parameters differing only by being *by value* and *by rvalue reference* and overload sets containing all three overloads.

Motivation and scope

This proposal addresses cases where the compiler is best at selecting the optimal calling convention for passing an immutable or rvalue parameter value to a function. This happens mostly in template code where the code's author does not know if the substituted type will be slow or fast to copy. Usually this results in the programmer selecting a reference parameter type which is best for larger values but inferior for smaller values such as primitive types and pointers.

This proposal is however not limited to templated parameters but includes non-templated parameters as well. Guessing the best calling convention can be hard even for known types, especially if the code is to be portable to different platforms.

Examples

Currently it is allowed to create function overloads with both `T` and `const T&` parameter overloads:

```
void f(int);
void f(const int&);
```

Unfortunately you can't call any of these functions in C++20, they are always ambiguous to each other.

This proposal changes this by allowing the compiler to select which of these functions to call. It would typically do this depending on expected performance. This has the advantageous effect that we can add overloads to for instance `vector::push_back` which takes a T by value, and it is up to the compiler to select which overload to call. The same ambiguity happens between T and T&& if the argument is an rvalue so in the push_back case there are two out of three overloads that are viable depending on whether the argument is a lvalue or rvalue:

```
class std::vector<T> {
    void push_back(const T&);  // #1, only callable with lvalues
    void push_back(T&&);       // #2, only callable with rvalues
    void push_back(T);         // #3 (New!), callable with both lvalues and
rvalues.
};
```

The main reason for the compiler to select 1 or 3 for lvalues is the cost of copying the value into the argument for `push_back(T)` compared to the extra indirection level incurred by the reference parameter in `push_back(const T&)`. Note that for some types the optimal calling convention may differ depending on the call site. This is especially true for ABIs where some parameter types are passed in registers. The argument expression evaluation may end up with a value in a register and the fastest may then be to just move the value to the register used by the parameter and call the by value overload. In another call site for the same type the argument may be an lvalue in memory and it may be more performant to call the by reference overload.

The same reasoning holds for the rvalue case, there may be argument expressions that make calling by value overload more efficient while other overloads make the by reference overload more efficient.

Impact on the standard

This proposal only affects the writing that makes an overload set where a parameter only differs between by value and by const lvalue reference or rvalue reference ambiguous. This text is replaced by a writing indicating that the compiler is free to select which overload to call if both exist.

In reality there could be any number of parameters that are ambiguous in this way between pairs of overloads. These pairs form a set of the best overloads that with today's rules can't be separated. In this case the compiler is free to select any of them to call.

Apart from overloads containing by value and by reference parameters of the same base type there are other types of ambiguities that can arise when there is more than one parameter and the overloads have implicit conversions that differ in another rank than **exact match**. Such ambiguities are not affected by this proposal.

ABI-compatibility retained

One aspect of adding a new push_back overload to vector and similar additions to other pre-existing classes is that there may be libraries compiled with older standard library versions without these overloads that can't be recompiled. There are two main situations:

- The previously compiled library relies on functions provided by a recompiled library. This should be no problem as we are not removing any overload, there must still be a definition of `std::vector<int>::push_back(const int&)` generated even if code compiled with a modern compiler that calls push_back on int vectors never calls it. [Note that if the function was inlined in the pre-compiled library the requirement is on the data layout of vector, which is not affected by this proposal].
- If the new code relies on code in the previously compiled library the new code should use the old header files with the old library so there can't exist any ambiguous declaration pairs.

In the case of for instance `std::vector<T>` being used both in the previously compiled library and in recompiled code and the new compiler chooses to call `push_back(const T&)` this call may be linked to the object code of the previously compiled library. This works unless some other ABI-breaking change has been made. If the new compiler chooses to call `push_back(T)` its object code will not be present in the old library and thus the object code must have been compiled with the new compiler. This should not cause any problems.

What if the overloads don't do the same thing, or one is private

If the overloads don't do the same thing semantically in the context of the program's overall functionality the program must be considered deficient. A problem for non-templated code including this type of overloading is that one of the overloads may be impossible to test on one platform as it is never selected by the compiler. This would be a problem in portable code if tests are not run on all platforms. It is however not good practice to run unit tests on one platform and assume that the results will be the same on other platforms.

In a separate proposal P2668 a way to write the desired overloads by one source code function is presented. This overcomes the concern with functions working differently or having different protection level.

Shouldn't the by value parameter be const

Yes, this would be beneficial as it would reduce error rates for cases when the local compiler selects a by value overload while a compiler for another platform running in a CI pipeline selects the by reference overload, thus seeing a const parameter value and causing errors if it is mutated, an error that would have been avoided if the overload set had been const T.

Technical specification

Wording has not been attempted at this stage. Below is just a sketch based on writing on cppreference.com, not the actual standard text.

The wording would probably be just a change in the last phrase of function overload resolution where it today says "If exactly one viable function is better than all others, overload resolution succeeds and this function is called. Otherwise, compilation fails.", to something like "The compiler is free to call any of the functions that are not worse than any other viable function if the conversions of all parameters only differ in the *exact match* rank, for all of the functions in this set. If this is not the case compilation fails".

There could be some complication in describing that the compiler is free to choose overload. But this seems unlikely as there is no need to indicate *how* this selection is to be made in the standard text. For instance an implementation which always selects the by reference overload is compliant. just as an implementation that does extensive analysis. The writing must only make clear that the programmer can't rely on which overload will be called. The rules that a particular compiler uses is a QoI issue, which could result in a Note in the standard text at most.

Acknowledgements

Thanks to my employer ContextVision AB for supporting the author attending standardization meetings.