Class Types in Non-Type Template Parameters

Document #: D0732R0
Date: 2017-11-11
Project: Programming Language C++
Audience: Evolution
Reply-to: Jeff Snyder <jeff-isocpp@caffeinated.me.uk>

1 TL;DR

We should allow non-union class types to appear in non-type template parameters. Require that types used as such, and all of their bases and non-static data members recursively, have a non-user-provided `operator<=>` returning a type that is implicitly convertible to `std::strong_equality`, and contain no references. Mangle values of those types by mangling each of their bases and non-static data members.

2 Introduction

Non-type template parameters are one of the few places in C++ where a visible distinction is made between class types and others: integral types, enum types, pointer types, point-to-member types, reference types and `std::nullptr_t` can all be used as non-type template parameters, but class types cannot. Array types can be used as non-type template parameters, but only syntactically—like function arguments of array type, they decay to pointers.

In C++98 there was a further distinction between class types and others: there were no compile-time constant expressions of class type. This changed in C++11 with the introduction of the `constexpr` keyword, leaving us in a situation where we have a great many tools to produce expressions of class type suitable for use as template arguments, but we cannot declare template parameters to accept them.

It would be desirable to remove this inconsistency in the language, but there are technical barriers to doing so. Jens Maurer's 2012 paper[N3413] provides a detailed analysis of the problems involved. This paper proposes a way to resolve these problems, and start allowing the use of some class types in non-type template parameters.

3 Motivation

Beyond the more abstract benefits of generalising existing language features and removing seemingly-arbitrary inconsistencies/restrictions in the language, there are a few concrete use cases that are addressed by allowing the use of class types in non-type template parameters:

3.1 Wrapper Types

Many APIs define class types that wrap built-in types in order to improve type safety, turn nonsensical operations into compile-time errors, and customise the behaviour of various operators. A prominent example is the `std::chrono` library, where the `time_point` and `duration` types are typically represented as integers internally.

There are clear benefits of using abstractions such as those in `std::chrono` instead of integers holding (e.g.) numbers of nanoseconds, but they come at a hidden cost: a program that uses specific time or duration values in template arguments cannot take advantage `std::chrono`'s abstractions, because its types cannot be used in non-type template parameters.

3.2 Compile-Time String Processing

There have been multiple proposals[N3599][P0424R0] to relax the prohibition on string user-defined literals that take their raw form as individual char template arguments, and so far all of these proposals have failed to gain consensus.

The primary concern that gets raised in response to such proposals is that handling strings as packs of `char` template arguments is extremely costly in terms of the compiler's CPU and memory usage, and allowing such UDLs would make very large packs of `char` values much more common than they are today. This may in turn result in pressure on compiler vendors to perform well with these compile-time string processing workloads, which is difficult due to the strings being represented using a syntax and mechanism that was never intended to support such use cases.

Recently, a proposal[P0424R2] to allow strings in template parameters has gained consensus, but the only valid arguments for those string template parameters are string literals and raw UDLs. Whilst this is significant progress, it still does not allow us to store computed strings in non-type template parameters.

By supporting non-type template parameters of class type, we can represent compile-time strings as a class containing an array of characters (e.g. a `std::fixed_string`[P0259R0]), making the string content part of the value of the non-type template parameter. With this approach, any computed string can be stored in a non-type template parameter.

Why not just use a raw array as the non-type template parameter? Unfortunately, the syntactic space that this would occupy is already taken: since C++98, array-to-pointer decay happens in non-type template parameter declarations, thus it is already possible declare a non-type template parameter of type `T[N]`, but what you actually get is a non-type template parameter of type `T*`.

Elsewhere in C++, the problems arising from array types not having value semantics are resolved by wrapping them in a class, c.f. `std::array`. Using the same approach here solves the problem of efficiently representing compile-time strings in non-type template parameters with a generic language feature, and in a manner consistent with the rest of C++.

4 Overview

At the core of the issues with allowing class types in non-type template parameters is the question of two template instantiations are the same, i.e. given `template<auto> int i;` and two values `a` and `b` of the same type, does the invariant[1]that `&i<a> == &i<b>` if and only if `a == b` hold? Current language rules ensure that it does, and maintaining this invariant seems desirable for the simplicity and teachability of the language.

Current language rules require both the compiler and the linker must be able to tell whether two template instantiations are the same or not, and so they must both be able to tell whether two values used as an argument to the same non-type template parameter are the same. This is typically achieved by including the values of all non-type template arguments in the mangled symbol name for the template being instantiated, such that equality of the resulting string can be used as a proxy for equality of all of the template arguments.

If a template argument is a value of class type, then how do we mangle it into symbol names? If we cannot, then how does the linker tell whether two template arguments of class type are the same? By default, class types do not even have a notion of equality. If we disregard unions, we could say that classes are simply compared memberwise for the purposes of template instantiation. However, doing this would break the invariant that `&i<a> == &i<b>` if and only if `a == b`[2]: the latter may not even be valid, and if it is then it would only give the same results as the former if the relevant `operator==` implemented the same memberwise comparison that we introduced for the purposes of template instantiation.

To resolve this question, we must either:

[1]For brevity, this formulation of the invariant ignores reference types. The actual invariant is that `&i<a> == &i<b>` if and only if `REF_TO_PTR(a) == REF_TO_PTR(b)`, where `REF_TO_PTR(x)` is equivalent to `&x` if `decltype(x)` is a reference type, and `x` otherwise.

[2]See footnote 1

1. Allow the existing invariant to be broken, and accept the complexity and subtleties added to the language by doing this, or
2. Attempt to develop the technology required for compiler and linkers to verify that user-defined equality operators are self-consistent and evaluate them, or
3. Only allow non-type template parameters of class type for classes which have memberwise equality

This paper proposes pursuing the last of these options, since it is the only option that maintains the aforementioned invariant without requiring substantial changes to current linker technology.

How do we tell whether a class type has memberwise equality? Detecting whether a user-defined `operator==` implements memberwise equality would require some heroic analysis in the compiler. If there were such a thing as compiler-generated comparisons, we could require that classes must have compiler-generated memberwise comparison instead of requiring that user-defined equality operators implement memberwise equality. C++ does not currently have any such compiler-generated comparisons, but Herb Sutter's recent proposal[P0515R2] provides almost exactly what we need, by allowing `operator<=>` to be defaulted. To make use of this we need to modify the invariant to be "`&i<a> == &i<b>` if and only if `(a<=>b) == 0`"[3], but this seems reasonable since `(a<=>b) == 0` is equivalent to `a == b` for all valid non-type template arguments in C++17.

By requiring that all classes used as non-type template parameters have an `operator<=>` that is defaulted within the class definition, and so do all of its members and bases recursively, we can ensure all of the following:

- The class type has a comparison operator available
- The class type has the same comparison operator available in all translation units
- The class' comparison operator implements memberwise equality

Together, these guarantees allow us to determine whether two instantiations of a templated entity involving non-type template parameters of a type meeting the requirements above are the same, in a manner that is consistent with comparisons in user code, and without any substantial departure from current compiler and linker technologies.

5 Proposal

5.1 Conceptual Model

In current C++, the closest thing to having a non-type template parameter of class type is having separate non-type template parameters for each of its members (e.g.

[3] The caveat from footnote 1 also applies here

`template<int x_first, int x_second>` instead of `template<pair<int,int> x>`).

This proposal uses that expansion as a conceptual model for classes in non-type template parameters—i.e. using a class type in a non-type template parameter is conceptually equivalent to having separate non-type template parameters for each of its bases, each of its non-array-type non-static data members, and each element of each of its array-type non-static data members.

This analogy is intended to aid in understanding the requirements for types and expressions used in non-type template parameters and arguments; it is not intended as an implementation model—there are likely to be more efficient ways of implementing this proposal, particularly with regard to arrays.

5.2 The Reference Problem

References have two notions of equality in current C++: `operator==` compares the referees, but instantiation of templates that have reference-type non-type template parameters compares the addresses of the referees.

This may become more counter-intuitive than it already is if there is a similar disparity in equality semantics for classes that have non-static data members of reference type. To avoid this problem, this paper does not propose allowing classes which contain references to be used in non-type template parameters.

5.3 The `operator==` Gap

This proposal aims to guarantee that whether two instances of a class are the same for the purposes of template instantiation is consistent with whether they are the same according to a comparison in normal C++ code.

Since the mechanism used to achieve this is based around `operator<=>`, any guarantees we can provide must also be in terms of `operator<=>`. For example, if we have a declaration `template<auto> int v;`, then `&v<a> == &v<b>` is true iff `a<=>b == 0`.

Since there is nothing to prevent a class author from implementing an `operator==` that is inconsistent with `operator<=>`, we cannot guarantee that `&t<a> == &t<b>` is true iff `a==b`, however desirable this may be.

The practical consequence of this is that any generic code that relies on two template instantiations having the same address iff their arguments compare equal must compare the arguments using `operator<=>` directly, instead of `operator==`.

5.4 Interaction with Class Template Argument Deduction

In keeping with the declaration an initialization of variables, we should support the use of class template argument deduction with non-type template parameters. Consider the following example, which initializes a variant of `std::fixed_string` from P0259[P0259R0] non-type template parameter to hold a compile-time string, but has to explicitly provide the string's length in order to do so:

```
namespace std {
    template <typename CharT, std::size_t N>
    struct basic_fixed_string
    {
        constexpr basic_fixed_string(const CharT (&foo)[N])
        { std::copy(foo, foo+N, m_data); }
        CharT m_data[N];
    };

    template <typename CharT, std::size_t N>
    basic_fixed_string(const CharT (&str)[N])->basic_fixed_string<CharT, N - 1>;

    template <std::size_t N>
    using fixed_string = basic_fixed_string<char, N>;
}

template <std::size_t N, std::fixed_string<N> Str>
struct A {};

using hello_A = A<5, "hello">;
```

By using template argument deduction for a non-type template parameter, the declaration and use of `A` could be simplified to the following:

```
template <std::basic_fixed_string Str>
struct B {};

using hello_B = B<"hello">;
```

5.5 Key Wording

This paper proposes extending the set of types that may appear in non-type template parameters with the addition of non-union literal class types that are also *trivially comparable*.

The intention of *trivially comparable* is to define the set of strongly comparable fundamental types, and class types for which all members and bases are strongly comparable, and comparison of the class type is equivalent to comparing all of its members and bases. The following definition attempts to capture that intent:

[*Wording:*

> A type is *trivially comparable* if it is:
>
> - an integral or enumeration type
> - a pointer type
> - a pointer-to-member type
> - `std::nullptr_t`
> - a non-union class type that has all of the following properties:
>   - it has an `operator<=>` that is defined as defaulted within the declaration of the class,
>   - its `operator<=>` returns either `std::strong_ordering` or `std::strong_equality`
>   - its bases all have trivially comparable types
>   - all of its non-static data members either have a trivially comparable type or are arrays of a trivially comparable type

— *end wording*]

There is already a great deal of overlap between the set of types that can appear in non-type template parameters and the set of types that are both literal and trivially comparable. This makes it possible to simplify [temp.param]p4 as follows:

[*Wording:*

> A non-type template-parameter shall have one of the following (optionally cv-qualified) types:
>
> (4.1) — a type that is both literal and trivially comparable
>
> (4.2) — lvalue reference to object or lvalue reference to function,
>
> (4.3) — a type that contains a placeholder type (10.1.7.4).

— *end wording*]

The wording above only handles the requirements on the types used in non-type template parameters. We must also consider the values that are permissible as arguments to non-type template parameters.

In keeping with our conceptual model, the restrictions on non-type template arguments of pointer type in [temp.arg.nontype]p2 should apply to all pointer-type subobjects of classes used as non-type template arguments. This paper does not include any wording for this restriction.

6 Acknowledgements

Many thanks to Louis Dionne, Timur Doumler, Graham Haynes and Richard Smith for their comments on early drafts of this paper.

References

[N3413] Jens Maurer. Allowing arbitrary literal types for non-type template parameters. Proposal N3413, ISO/IEC JTC1/SC22/WG21, September 2012.

[N3599] Richard Smith. Literal operator templates for strings. Proposal N3599, ISO/IEC JTC1/SC22/WG21, March 2013.

[P0424R0] Louis Dionne. Reconsidering literal operator templates for strings. Proposal P0424R0, ISO/IEC JTC1/SC22/WG21, August 2015.

[P0424R2] Louis Dionne. Reconsidering literal operator templates for strings. Proposal P0424R2, ISO/IEC JTC1/SC22/WG21, November 2017.

[P0259R0] Andrew Tomazos Michael Price. fixed_string: a compile-time string. Proposal P0259R1, ISO/IEC JTC1/SC22/WG21, February 2016.

[P0515R2] Herb Sutter. Consistent comparison. Proposal P0515R1, ISO/IEC JTC1/SC22/WG21, September 2017.