Reconsidering concepts in-place syntax

Document number: P2677R0
Date: 2022-10-14
Project: Programming Language C++
Audience: EWG
Author: Mike Spertus
Reply-to: msspertu@amazon.com

Abstract

While "terse notation" for function templates initially seems simple and clear,

```
auto square(auto x) { return x*x; }
```

we have found it difficult to make effective use of terse notation in practice because it lacks the type names for deduced function parameters that so many function templates desire (e.g., for forwarding, use in function template bodies, consistent binding, etc.).

In investigating this problem, we were pleased to find a note in Section 2.1 of Herb Sutter's P0475 R1: Concepts in-place syntax that contemplates allowing names to be added after `auto` like

```
[](auto{T}&& x) { return f(std::forward<T>(x)); }
```

As described below, we believe permitting this compatible extension would make terse function templates expressive enough for regular use, bringing us closer to Bjarne Stroustrup's vision of making generic programs "just" normal programming.

The note in P0745, mentions this only as a possible future proposal, but as it is fully compatible with the approach adopted in C++20 and simply and clearly addresses an important problem that arises in practice, we believe that future is now.

Acknowledgment

We happily acknowledge that this proposal advocates for nothing other than what Herb Sutter contemplated in the note to Section 2.1 of P0745 mentioned above. All of the invention is his, and all of the defects of presentation are mine. Note also that that paper is a great read with additional material far beyond that note that is not in the scope of this paper. I would also like to mention that discussions with Bjarne Stroustrup were invaluable in clarifying my thinking about this.

The Problem

I introduce templates in my C++ course through the terse notation for function templates

```
auto square(integral auto x) { return x*x; }
```

Students new to C++ have no trouble understanding this and are attracted to templates as simple and powerful. Likewise, students with C++ experience find terse notation simpler and easier to write than the traditional long form notation. Unfortunately, this excitement quickly fades as the course progresses to more realistic code where they find that it is not suitable for the vast majority of the non-trivial function templates we encounter, which want to make use the parameter type name that is provided by the long form notation but not the terse notation.

```
// Parameter type name used in body
template<typename T>
```

```
void f(T t) { vector<T> vt; /* ... */ };

// Consistent binding
template<integral T>
T gcd(T l, T r);

// Perfect forwarding
template<typename X, typename ...Ts>
void wrap(X x, Ts&& ...ts)
    {  /* Do stuff */  x(std::forward<Ts>(ts)...); /* Do stuff */ }

// Independent binding but type names still used in body
template<Animal Predator, Animal Prey>
void simulate(Predator &predator, Prey &prey)
{
    set<decay_t<Predator>> predators;
    set<decay_t<Prey>> preys;
    /* ... */
}
```

Before long, we invariably end up discarding the terse notation (outside of trivial lambdas) and just use the long form for consistency. The students are left wondering why they had to learn two notations for function templates if one is just discarded and frustrated by how cumbersome writing function templates has become after seeing how "function-like" they could be. Outside of teaching, I have heard similar feedback from professional C++ programmers.

Note: While we acknowledge that the examples above are technically possible to write in C++23 terse notation as shown below, doing so does not result in clearer or simpler code than the long form and is not to be recommended.

```
// C++23 terse notation is not an improvement

void f(auto t) { vector<decltype(t)> vt; /* ... */ };

// Subtle behavior change. OK? Depends
auto gcd(auto l, std::type_identity_t<decltype(l)> r) -> decltype(l);

void wrap(auto x, auto && ...ts)
{  /* Do stuff */
   /* Seems to still work because of the following subtle points:
     1) Special casing of decltype for unparenthesized non-type template
         parameter id-expressions (dcl.type.decltype/1.2)
     2) For non-reference type V both forward<V> and forward<V&&> both
        forward as rvalue reference, so behavior is the same as the
        traditional version even though the template argument is different.
   */
   x(std::forward<decltype(ts)>(ts)...);
   /* Do stuff */
}

void simulate(Animal auto &predator, Animal auto &prey)
{
    set<decltype(auto(predator))> predators;    // Uses c++23 auto(x)
    set<decltype(auto(prey))> preys;
    /* ... */
}
```

Solution

When terse notation is applicable, it creates a great experience that delivers on Bjarne Stroustrup's dictum of making generic programming "just" be normal programming, but for terse notation to work, we believe it needs to be suitable for regular use, not just occasional use. As described in the Problem section above, C++23 abbreviated template syntax too often fails to reach that standard.

Fortunately, we contend that with the small change of allowing one to optionally insert a name between curly braces after `auto` as in the aforementioned note in P0745, all of the above examples become natural (note that since the bodies are the same, we only show the declarations).

C++23	Proposed

`template<typename T>`
`void f(T t);`
 	
`template<integral T>`
`T gcd(T l, T r);`
 	
`template<typename X, typename ...Ts>`
`void wrap(X x, Ts&& ...ts);`
 	
`template<Animal Predator, Animal Prey>`
`void`
`simulate(Predator &predator, Prey &prey);`
	
	`void f(auto{T} t);`
	
	`// No more semantic change`
	`auto gcd(integral auto{T} l, T r) -> T;`
	
	`// Forwarding only needs typenames for ts`
	`void wrap(auto x, auto{Ts}&& ...ts);`
	
	`void`
	`simulate(Animal auto{Predator} &predator,`
	`         Animal auto{Prey} &prey);`

Even nicer, this same notation doesn't just apply to function templates, it consistently works whenever `auto` is used for type inference.

```
arithmetic auto{A} a = calculation();
A a2 = refine(a);
```

Limitations

While we have found this ability to optionally name deduced types pleasing to use and sufficient for most function templates, there are some cases that still require the traditional long form notation such as the following:

Function templates with non-deducible template arguments

```
template<typename X, typename ...Ts>
X make_unique(Ts&& ...ts);
```

This example seems intrinsically unsuited to terse notation.

Function templates with non-type template parameters

```
template<typename T, size_t m, size_t n>
Matrix<T, m, n> operator+(Matrix<T, m, n> l, Matrix<T, m, n> r);
```

This example could possibly be addressed in the future by extending `auto` to deduce non-type template parameters.

Function templates with `requires` clauses

```
template<typename T, typename U> requires convertible_to<T, U>;
```

Addressing this example would need support for `requires` clauses in terse notation.

In spite of the fact that this proposal does not completely eliminate the need for traditional template notation for function templates, it seems to cover the vast majority.

Case Study

To see if our experience was representative, we manually inspected the single-header version of Niels Lohmann's JSON for Modern C++ library, which makes extensive and effective use of templates, as a (notional) case study.

We found that the header contained 250 headers, all written in the long form notation. Of these, we found that 100 (40%) of them could be written using C++23 terse notation without having to rewrite their bodies to avoid parameter type names or work around other impediments to writing in terse notation. We did feel free to replace `enable_if` statements with a concepts approach, as that would be an improvement in this context.

With the proposed extension, 214 (86%) Could be written naturally with terse notation. Of the 36 function templates that did not lend themselves to the proposed terse notation, 30 of them had a template parameter that was not deducible from the function arguments, 3 of them had a SFINAE/`requires` constraint that involved two template parameters, and 3 of them had a deducible non-type template parameter.

We feel this lends credence to our belief that this proposal would make the common case for writing function templates simple whereas C++23 terse notation makes the uncommon case simple but leaves the common case hard.

Note: We feel this example may understate the benefits of the proposal for the following reason. Many of the function templates that could be rewritten with C++23 terse notation belonged to groups of related function templates, not all of which lend themselves to C++23 terse notation. For example, while `json_pointer::flatten` works well with C++23 terse notation, but `json_pointer::unflatten` does not. We suspect it would feel weird to write one in terse notation but not the other. Likewise, only some of the comparison operators for `json_pointer` work well with C++23 terse notation. Based on the above, we feel that significantly fewer than 40% of the function templates would be good candidates for C++23 terse notation. By contrast, this appeared to be much less of an issue with our proposed extension.

Other uses of `auto{...}`

As pointed out in P0745, expressions like `new auto{3}` are legal in C++, and subsequently C++23 added `auto{x}` expressions (P0849). We do not believe these result in ambiguity or unacceptable collisions with our proposal (or each other!). If this is incorrect, other notations could be considered.

Past committee discussions

P0745 was reviewed in the 2018 Rapperswil meeting along with A minimal solution to the concepts syntax problem. Interestingly, both papers received consensus with no clear preference between them, which may explain why the committee wanted to wait for the situation to clarify. We do not believe the particular notation of this paper was discussed, even though it is mentioned in P0745.

Experience with C++20's use of `auto` to indicate inference has increased our confidence in moving forward with this particular idea from P0745 at this time (we make no representations about other parts of that paper). We do note that P1079's objection to P0745's use of empty `{}` does not apply to this proposal.