Reconsidering concepts in-place syntax

Document number: P2677R2
Date: 2023-01-15
Project: Programming Language C++
Audience: EWG
Author: Mike Spertus
Reply-to: msspertu@amazon.com

Changes from P2677R1

- Switching recommended symbol from `~` to `:` along with additional discussion of possible notations
- Discuss an universal reference example of Ville Voutilainen

Abstract

While "terse notation" for function templates holds the promise of making ordinary function templates as natural to write as ordinary functions,

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

we have found that it is 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 as a future proposal allowing names to be added after `auto`

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

As pointed out by Christopher Meerwald, this notation is no longer backwards compatible since the introduction of C++23 `auto{x}` as shown here, so this draft proposes the following notation and adds a Syntax section discussing alternatives.

```
[](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.

Note: The previous revision of this paper proposed `auto~T`. While this remains a possibility, the current revision prefers the `auto:T` notation based on feedback in the Kona meeting that some European keyboard treat the `~` character specially. See the Syntax Options section below for more discussion of this and other alternatives.

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;
    /* ... */
}

// Function templates with `requires` clauses
template<typename T, typename U>
  requires convertible_to<T, U>
void f(T t, U u);
```

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;
    /* ... */
}

template<typename T, typename U>
void f(auto t, auto u)
  requires convertible_to<decltype(t), decltype(u)>;
```

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.

We contend that with the small change of allowing one to optionally adding a tilde followed by an identifier after `auto`, 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);`
 	
`template<typename T, typename U>`
	`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);`

```
  requires convertible_to<T, U>
void f(T t, U u);
```

```
void f(auto:T t, auto:U u)
  requires convertible_to<T, U>;
```

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>
unique_ptr<X> make_unique(Ts&& ...ts);
```

This example seems intrinsically unsuited to terse notation (at least for the nondeducible parameter).

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.

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.

Interaction of concepts with universal references

Universal references can interact poorly with concepts as illustrated by the following C++23 example:

```
int f(integral auto &&x) { return x; }
int i{};
int j = f(i); // ill-formed: integral<int &> == false
```

Since it is reasonable to want to concept constrain universal references, a `requires` clause is necessary in such cases.

```
int f(auto &&x) requires integral<decay_t<decltype(x)>> {
     return x;
}
int i{};
int j = f(i); // OK
```

Furthermore, as Ville Voutilainen points out, universal references are useful, not just in the (arguably) advanced use case of perfect forwarding, but in application code, such as

```
std::string normalPath() {return {};} // fake impl
std::string backupPath() {return {};} // fake impl

struct Image
{
    Image() = default;
    Image(const std::string& savePath) {} // all impls fake
    Image(const std::string& savePath, const Image&) {} // all impls fake
    void generateThumbnail() {} // all impls fake
    void save() {} // all impls fake
};

concept thumbable = ...;
void generateThumbAndSave(auto&& image) // See below for how to constrain to thumbable
{
    image.generateThumbnail();
    image.save();
}

void saveImage(Image& image)
{
  // Leverage universal reference in generateThumbAndSave
  generateThumbAndSave(image);
  generateThumbAndSave(Image{backupPath(), image});;
}

int main()
{
    Image image;
    saveImage(image);
}
```

While we agree that the sharp corners with universal references and concepts are an issue that is worthy of addressing, we believe it is mostly orthogonal to this proposal. Indeed, not only do we see no downside to the current proposal in this case, it modestly improves the notation for constraining the argument to `generateThumbAndSave` satisfy the `thumbable` concept.

In long form

```
template<typename T>
void generateThumbAndSave(T &&image)
  requires thumbable<decay_t(T)> { ... }
```

In C++23 short form

```
void generateThumbAndSave(auto &&image)
  requires thumbable<decay_t(decltype(image))> { ... }
```

In this proposal

```
void generateThumbAndSave(auto:T &&image)
  requires thumbable<decay_t(T)> { ... }
```

In particular, we do not believe this reduces the value of this proposal. Not only is the proposal still better in this case, but constrained universal references seem rare (they did not appear in the case study as all), so we do not believe they detract from our goal of "making the simple case simple."

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, 217 (87%) Could be written naturally with terse notation. Of the 33 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 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.

Syntax options

The R0 revision of this paper and P0745 proposed `auto{T}` as a notation. As shown by Christopher Meerwald, this notation is no longer available since the introduction of C++23 `auto{x}` as shown here.

We consider a number of options that have been suggested in rough order of preference.

`auto:T`

```
void f(auto:T t);
void g(integral auto:T t);
```

This notation seems natural and worth considering. It does not appear to be ambiguous with the use of colon in labeled statements, inheritance, ranged for, or the ternary operator (although this should be reviewed carefully by the Core Working Group)

`auto~T`

```
void f(auto:T t);
void g(integral auto:T t);
```

The R1 revision of this paper proposed `auto~T` both to avoid ambiguity and having the conciseness befitting a terse notation. However, based on feedback that typing `~` on some European keyboards is interpreted as a code for accented characters, we are concerned that it would not meet the goal of making writing typical function templates as convenient and friction-free as writing ordinary functions.

`auto$T`, `auto@T`, ``auto`T``

```
void f(auto$T t);
void g(integral auto$T t);
```

P2558R1, which adds `$` (as well as backtick and `@`) is now in core. While tempting, unused symbols are in short supply, and we believe they should only be used where there is no other alternative

`auto<T>`

```
void f(auto<T> t);
void g(integral auto<T> t);
```

Our concern was that statements like

```
auto<thread> x = vector<thread>;
```

suggest that when matching `auto<T>` to a `vector<T>`, the `auto` reads as corresponding to `vector` rather than `vector<T>`. Also, while currently ill-formed, we can imagine constructs being proposed along these lines in the future and do not want to preclude that. Finally, brace symbols seem desired by many proposals and are in short supply, so we do not wish to use them where not necessary.

`T{auto}`

```
void f(T{auto} t);
void g(T{integral auto} t);
```

Again, this seems technically possible but breaks precedent for introducing names without being preceded by a syntactic introducer and consumes brace symbols.

`T:auto`

```
void f(T:auto t);
void g(T:integral auto)
```

Although we we do not see any any technical objections for this notation, we think it is a negative for readability that `T` and `auto` are no longer adjacent even though `T` is the name for what `auto` deduces. We also are not aware of a precedent for introducing new names without being preceded by an introducer.

`T=auto`

```
void f(T=auto t);
void g(T=integral auto t);
```

In addition to the considerations given for `T:auto` we are worried that code like the below could be confusing to read

```
void h(T = integral auto t = 7)
```