Document Number: P3667R0
Date: 2025-04-10
Reply to: J. Daniel Garcia
e-mail: jdgarcia@inf.uc3m.es
Audience: EWIG

Extending range-for loop with an expression statement

J. Daniel Garcia
Computer Science and Engineering Department
University Carlos III of Madrid

Abstract

This paper proposes extending the range-based for loop syntax with an expression statement. This form of statements minimizes the likelihood of forgetting the update of a variable introduced by an init statement in a range-based for loop.

1 Introduction

Initializers in range-based for loops [1] were introduced in C++20 to solve issues with prvalues that might lead to undefined behavior.

This has led to a new common pattern while iterating:

```
std::array vec {2, 4, 6};
for (int i=0; auto x : vec) {
  std::println("v[{}] = {}", i, x);
  ++i;
}
```

```
v[0] = 2
v[1] = 4
v[2] = 6
```

This allows safe traversal of an iterable, while having a handy way to access to the integer index.
However, one might easily forget updating the index:

```
std::array vec {2, 4, 6};
for (int i=0; auto x : vec) {
  std::println("v[{}] = {}", i, x);
  // Ouch! Forgot to update i
}
```

```
v[0] = 2
v[0] = 4
v[0] = 6
```

An obvious extension is to allow the increment to be part of the for statement. P0614 [1] already outlined as a future direction a range-based for with increment as **for (init; auto x : e; inc)**. However, to the best of my knowledge this has never been proposed.

2 Proposed change

2.1 Before/After

Before	After

`for (int i = 0; auto x : e) {`
`  f(x,i);`
`  ++i;`
`}`
	`for (int i = 0; auto x : e; ++i) {`
	`  f(x,i);`
	`}`

2.2 Discussion

2.2.1 Current use

The use of initializers in for-statements is encouraged by some coding guidelines (e.g. CppCoreGuidelines ES.6). However, the fact that the initialized variable needs to be updated in loop body (and might be forgotten) might lead to logical errors.

This current state has also a teachability impact as the programmer has to remember to add the update of the initialized variable at the end of the loop body.

2.2.2 Expected meaning

It might be considered that the proposed meaning is surprising as a possible interpretation is that in a traditional for loop and a range-based for loop have similarities:

```
for (int i=0; i<max; ++i) {
  //...
}

for (auto x : v) {
  //...
}
```

Consequently the expression **x : v** may be seen as a termination condition, just like the expression **i ¡ max**. However, consider the case of the proposed new syntax.

```
for (int i=0; x : v; ++i) {
  //...
}
```

In this case, the programmer might see that the apparent loop variable is **i** and that it is used in the loop condition.

However, a different view is that we can read **for auto x : v** as *"for each value x in the range v"*. This is more consistent with the name *range-based for loop*.

The fact that we have an unrelated variable **i** was introduced in C++20. So, since then we have the syntax for *range-based for loop with initializers*:

```
for(int i=0; auto x: v) {
  //...
}
```

Thus, in fact, we have **two** loop variables introduced: **i** and **x**. The latter (**x**) is implicitly updated in the for statement, while the former (**i**), needs to be updated in the loop body and may be easily forgotten.

2.2.3 Considering the for-range declaration different from the condition

An alternate design might consider the *for-declaration* different from the condition.

I am **not proposing** all the following possibilities:

```
// Traditional for loop.
// Works today.
for (int i=0; i<max; ++i) {
  //...
}
```

```
// Range−based for loop.
// Works today.
for (x : v) {
  //...
}

// Range−based for loop with initializer.
// Works today.
for (int i=0; x : v) {
  //...
  ++i;
}

// Range−based for loop with initializer and condition.
// Not Proposed.
for (int i=0; x : v; i<10) {
  //...
  ++i
}

// Range−based for loop with initializer, condition and expression
// Not proposed.
for (int i=0; x : v; i<10; ++i) {
  //...
}

// Range−based for loop with initializer, condition and without expression
// Not Proposed.
for (int i=0; x : v; ; ++i) {
  //...
}
```

This would be more complicated from what I currently propose in this paper.

2.2.4 Conflict with multi-range for loops

A conflicting proposal would be to allow multiple ranges in the same for-loop as outlined by P0026 [2].

```
for (int i = 0; auto x : e1; auto y : e2) {
  f(x,y);
}
```

However, such a proposal was presented in 2016 and since then there has been no activity in such direction.

Nevertheless, I am not preventing this approach if it is retaken as a **for-range declaration** is different from an expression.

2.2.5 Library solutions

Although a library solution has not been explored, one might envision such a solution based on some kind of iterator:

```
for (auto i : indexed{e}) {
  std::println("v[{}] = {}", i.index(), i.value());
}
```

Or even in combination with structured bindings:

```
for (auto [i, x] : indexed{e}) {
  std::println("v[{}] = {}", i, x);
}
```

However, those solutions assume a specific type for the index (**int**, **std::size_t**, ...). Besides they do not allow for cases where the updated variable(s) might get an update different from plain increment.

2.2.6 Using views::enumerate

One specific solution would be to use **std::views::enumerate**.

```
std::array v{2.5, 3.0, 4.5, 5.0, 5.5};
for (auto [i, x] : std::views::enumerate(v)) {
  std::println("v[{}] = {}", i, x);
}
```

However, this solution has two drawbacks:

- It is not possible to specify the concrete type of the integer index.
- The initial value of the index is fixed to start at **0**.

2.2.7 Other uses

The proposed solution does not limit itself to integer indexing and may have other uses:
Consider the following:

```
enum class flag { none=0, open=1<<0, close=1<<1, read=1<<2, write=1<<3 };

flag next(flag x); // Get next flag option

std::array names { "<none>", "<open>", "<close>", "<read>", "<write>"};
for (auto f = flag::none; auto n : names; f = next(f)) {
  do_something(f);
  std::println("{}", n);
}
```

In this case, the variable **f** in the *for-loop initializer* is from an enum type. And in each iteration is updated by the expression **f = next(f)**.
Such a loop does not seem easy to express with **std::views::enumerate**.

2.2.8 Current approach

Current approach for the example covered in this paper could be:

```
std::array vec {1, 2, 3, 4, 5, 6};
for (int i=0; auto x : vec) {
  std::println("v[{}] = {}", i++, x);
}
```

However, this presents a maintenance burden. Consider now, that we decide to modifiy the loop to print only even values:

```
std::array vec {1, 2, 3, 4, 5, 6};
for (int i=0; auto x : vec) {
  if (is_even(x)) {
    std::println("v[{}] = {}", i++, x);
  }
}
```

That would incorrectly print indices of even values as the index is only incremented if **is_even(x)** is **true**.
However, consider both loops with the proposed solution:

```
std::array vec {1, 2, 3, 4, 5, 6};
for (int i=0; auto x : vec, ++i) {
  std::println("v[{}] = {}", i, x);
}
for (int i=0; auto x : vec, ++i) {
  if (is_even(x)) {
    std::println("v[{}] = {}", i, x);
  }
}
```

3 Teachability

This proposal improves teachability of the C++ language as it makes regular for loops and range-based for loops more uniform improving regularity of the language.
We can simply teach that a for loop has:

- Optionally, a possibly comma-separated list of initializations in an *init-statement*.
- Optionally, one of the followings:
  - A condition.
  - A *for-range-declaration* separated with colon from a *for-range-initialize*
- Optionally, an expression.

In addition to regularity, this proposal allows to avoid simple errors made by programmers when using range-based for loops.

4 Proposed wording

No proposed wording is provided in this version.

Acknowledgements

Thanks to Thomas Köppe for his previous work on range-based for with initializers and for providing feedback and context.

Thanks to Bjarne Stroustrup, Gabriel Dos Reis and Ville Voutilainen for their comments on early versions of this paper.

References

[1] Thomas Köppe. Range-based for statements with initializer. WG21 proposal P0614R1, ISO/IEC JTC1/SC22/WG21, November 2017.

[2] Matthew McAtamney-Greenwood. multi-range-based for loops. WG21 proposal P0026, ISO/IEC JTC1/SC22/WG21, August 2015.