`range-if`

Abstract

This paper proposes adding a ranged `if`-statement, branching based on the emptiness of the supplied range. If the range is non-empty it is equivalent to `range-for`, otherwise an optional `else` part is executed.

Tony Table

Before	Proposed

```// ❗ if in disguise!```
``for(auto & x : a-view-pipeline
	std::views::take(1)) {```
	``` …```
	```}```
		``if(auto & x : a-view-pipeline	std::views::take(1)) {```
			``` …```
			```}```

```auto && r = a-view-pipeline;```
 	
```if(!r.empty()) { // ❌ not all views provide empty()```
``` for(auto & x : r) {```
```   …```
``` }```
```} else {```
``` //fallback for empty range```
```}```
	```if(auto & x : a-view-pipeline) {```
	
	``` …```
	
	```} else {```
	``` //fallback for empty range```
	```}```

```auto && r = a-view-pipeline;```
 	
```// 😐 explicit iterator use```
```if(auto it{r.begin()}; it != r.end()) {```
``` for(; it != r.end(); ++it) { //redundant check```
```   auto & x{*it};```
```   …```
``` }```
```} else {```
``` //fallback for empty range```
```}```
	```if(auto & x : a-view-pipeline) {```
	
	``` …```
	
	```} else {```
	``` //fallback for empty range```
	```}```

```// 😐 iteration flag```
```auto empty{true};```
```for(auto & x : a-view-pipeline) {```
``` empty = false;```
``` …```
```}```
 	
```if(empty) {```
``` //fallback for empty range```
```}```
	```if(auto & x : a-view-pipeline) {```
	
	``` …```
	```} else {```
	
	``` //fallback for empty range```
	```}```

Revisions

R0: Initial version

Motivation

With the adoption of `range-for` and especially with the introduction of `ranges` and `views` it has become possible to express C++ programs in a more declarative fashion than before, allowing regular users to avoid dealing with iterators directly. Unfortunately this programming model only supports loops, not conditionals - there is no simple way to detect an empty `view` and execute some alternative code path.

Readers may initially want to push back on the above assertion, pointing to the `empty` member function that all views inherit from `view_interface`. But said member function is actually constrained and is not available for several types of views, among them `input_view`s (like `generator`). The only way to determine whether any given view is empty is by equality comparing its `begin-iterator` and `end-sentinel`, thereby leaving the „declarative world" and going back to low-level constructs. As obtaining the `begin-iterator` may be non-repeatable (e.g. see the contract of `generator::begin`) naïve constructs like iterator-comparison followed by `range-for` are also highly problematic.

Another „workaround" we've encountered in the wild was the combination of `range-for` and an `iteration-has-taken-place` flag on which a subsequent branch is taken. Neither this nor manual iterator use is a solution we want to promote to regular users. Whilst we could come up with a library-based solution, it would suffer the same limitations as `for_each` does in comparison to `range-for`. Therefore we prefer a dedicated language feature.

Design Space

Admittedly we weren't too happy with re-using the `if` keyword, but remain unconvinced that the considered alternatives are superior:

- `for (`*init-statement*$_{opt}$ *for-range-declaration* : *for-range-initializer*`)` *statement*$_1$ `else` *statement*$_2$
  works in Python with different semantics, but would change the meaning of existing C++ code.

- `if constexpr`$_{opt}$ `for (`*init-statement*$_{opt}$ *for-range-declaration* : *for-range-initializer*`)` *statement*$_1$ `else` *statement*$_2$
  expresses the wrong breaking semantics for the `else`-path.

- `for if constexpr`$_{opt}$`(`*init-statement*$_{opt}$ *for-range-declaration* : *for-range-initializer*`)` *statement*$_1$ `else` *statement*$_2$
  looks like a conditional loop-body, but would express the right breaking semantics for the whole construct.

- Introducing a new keyword is always a hassle for existing codebases.

Our imagined syntax for `range-if` is a combination of the syntaxes of regular `if` and `range-for`, that desugars in a similar fashion to the latter:

An `if` statement of the form

```
if constexpr_opt(init-statement_opt for-range-declaration : for-range-initializer)
statement
```

is equivalent to

```
{
    init-statement_opt
    auto && range = for-range-initializer;
    auto begin = begin-expr;
    auto end = end-expr;
    if constexpr_opt(begin != end)
      do {
        for-range-declaration = *begin;
        statement
      } while(((void)++begin), begin != end);
}
```

and and `if` statement of the form

```
if constexpr_opt(init-statement_opt for-range-declaration : for-range-initializer)
statement_1 else statement_2
```

is equivalent to

```
{
    init-statement_opt
    auto && range = for-range-initializer;
    auto begin = begin-expr;
    auto end = end-expr;
    if constexpr_opt(begin != end)
      do {
        for-range-declaration = *begin;
        statement_1
      } while(((void)++begin), begin != end);
    else
      do { statement_2 } while(false);
}
```

Like all other loops `range-if` supports `break` and `continue`. To ensure these jumps appertain to the same statement in the unprecedented `else` part of `range-if` the proposed desugaring wraps the `else`-statements in a dummy loop.

We are currently not aware of a need for this functionality in the context of `template for`, as based on our reading of P1306R5 expansions over ranges require sized ranges. If such a need ever arises, we expect `template if constexpr_opt` to be an appropriate evolutionary path.

Impact on the Standard
This proposal should be a pure language addition, the proposed syntax is unambiguous and currently invalid.

Implementation Experience
Not yet.

Proposed Wording
Wording will be provided in a future revision, if further work on this subject is encouraged.

Acknowledgements
Thanks to RISC Software GmbH for supporting this work.