Document number: P4309R0
Date: 2026-07-15
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

Constructing owning *node–handle*s without containers

Abstract
This paper proposes adding a way to construct owning *node–handle*s without containers.

Tony Table

Before	Proposed

`using immovable = …;`
`using container = node-based-container<immovable>;`
`using node = container::node_type;`
 	
`auto factory(auto & input_stream) -> node {`
`  auto in{input_stream.next()};`
`  if(can_ignore(in)) return {}; //empty handle`
 	
`  container c; //allocate expensive data structure 😬`
 	
`  //then allocate the actual object`
`  c.emplace(move(in));`
`  auto nh{c.extract(c.begin())};`
 	
`  if(filter(nh.value())) return {}; //empty handle`
 	
`  return nh;`
`}`
	`using immovable = …;`
	`using container = node-based-container<immovable>;`
	`using node = container::node_type;`
	
	`auto factory(auto & input_stream) -> node {`
	`  auto in{input_stream.next()};`
	`  if(can_ignore(in)) return {}; //empty handle`
	
	`  //allocate only the actual object 😊`
	
	`  node nh{in_place, move(in)};`
	
	`  if(filter(nh.value())) return {}; //empty handle`
	
	`  return nh;`
	`}`

Revisions
R0: Initial version

Motivation
*node–handle*s are a great way to transfer expensive-to-move or immovable objects between node-based containers (which is one of the motivations in P3049 for adding this API to lists).

Whilst the existing *node–handle* API works fairly well, there is one annoyance we've encountered multiple times: being unable to directly create an owning *node–handle* without constructing a container. Creating a container just to emplace one element that is immediately extracted, after which the container is destroyed is silly … yet is necessary with the current API, if a separation from the target-container (one of the motivations of this API) is desired.

We think this use-case should be supported directly and propose an extension to the *node–handle* API.

[1] RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

Design Space

This feature could be provided in different ways: factory functions[2], or additional constructors for *node-handle*. We prefer the latter option as it is the most logical to us - after all, what we want is a way to construct an owning *node-handle*. Additionally this design avoids having to come up with a new name for this operation.

We propose to extended the *node-handle* „meta-template“ that all actual handle types have to conform to in the following way[3]:

```
template<unspecified>
struct node-handle {
  using value_type     = see below;     //not present for map containers
  using key_type       = see below;     //only present for map containers
  using mapped_type    = see below;     //only present for map containers
  using allocator_type = see below;
private:
  using container-node-type = unspecified;                                    //exposition-only
  using ator-traits = allocator_traits<allocator_type>;                       //exposition-only

  typename ator-traits::template rebind_traits<container-node-type>::pointer ptr_; //exposition-only
  optional<allocator_type> alloc_;                                            //exposition-only
public:
  node-handle() noexcept : ptr_(), alloc_() {}

  template<typename... Args>
  explicit
  node-handle(in_place_t, Args &&... args);                                   //this proposal 1
  template<typename... Args>
  explicit
  node-handle(allocator_arg_t, const allocator_type & alloc, Args &&... args); //this proposal 2

  node-handle(node-handle &&) noexcept;
  auto operator=(node-handle &&) -> node-handle &;
  ~node-handle();

  …
};
```

Both proposed constructors are marked explicit to prevent implicit dynamic allocations and take a leading tag parameter - 1 in order to disambiguate the construction without additional arguments from the existing default constructor and 2 to mark the ability to pass an allocator that should be used for allocating the actual node.

The basic idea is that these constructors work in the following way[4]:

```
node nh{in_place, arg...};

//is equivalent to:

node nh{[] {
  some-compatible-container<node> c;
  c.emplace(arg...);
  return c.extract(c.begin());
}()};
```

```
node nh{allocator_arg, alloc, arg...};

//is equivalent to:

node nh{[] {
  some-compatible-container<node> c{alloc};
  c.emplace(arg...);
  return c.extract(c.begin());
}()};
```

Note that `node` will often be associated with multiple, compatible containers who differ - for this use case - only in inconsequential details[5]. As we don’t want a container to be created, we don’t care which compatible container would be used.

---

[2] Either as free functions, or as static member functions of the respective containers.

[3] Note: assumes P3049 being accepted.

[4] The code shown is illustrative, actual code differs between three groups of node-based containers (maps&sets, doubly-linked list, singly-linked list) due to differences in their `emplace` and `extract` APIs.

[5] Hasher, comparisons, support for duplicated keys.

Impact on the Standard

This proposal is a pure library addition. Existing standard library classes are modified in a non-ABI-breaking way.

Proposed Wording

Wording is relative to [N5046]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[version.syn]

```
#define __cpp_lib_node_construct YYYYMML //also in <map>, <set>, <unordered_map>, <unordered_set>, <list>, <forward_list>
```

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]

[container.node]

??.?.?.? Overview **[container.node.overview]**

…

2 If a node handle is not empty, then it contains an allocator that is equal to the allocator of the container when the element was extracted. If a node handle is empty, it contains no allocator.

…

```
template<unspecified>
class node-handle {
…
  using container-node-type = unspecified;                  // exposition only
  using ator-traits = allocator_traits<allocator_type>;     // exposition only

  typename ator-traits::template
    rebind_traits<container-node-type>::pointer ptr_;       // exposition only
  optional<allocator_type> alloc_;                          // exposition only

public:
  // [container.node.cons], constructors, copy, and assignment
  constexpr node-handle() noexcept : ptr_(), alloc_() {}
  template<class... Args>
    constexpr explicit node-handle(in_place_t, Args&&... args);
  template<class... Args>
    constexpr explicit node-handle(allocator_arg_t, const allocator_type& a, Args&&... args);
  constexpr node-handle(node-handle&&) noexcept;
  …
};
```

??.?.?.? Constructors, copy, and assignment **[container.node.cons]**

```
template<class... Args>
  constexpr explicit node-handle(in_place_t, Args&&... args);
template<class... Args>
  constexpr explicit node-handle(allocator_arg_t, const allocator_type& a, Args&&... args);
```

1 Let a be `allocator_type()` for the overload with no parameter a.

2 Let C be a container type so that `C::node_type` is the same as *node-handle* and let c be an instance of C initialized with a.

[DRAFTING NOTE: node handles are shared between compatible containers, but as we don't want instances of C to be constructed, it doesn't matter which compatible container type is used as C.]

3 *Effects*: Constructs a *node-handle* object as if by extracting an object t from c, where t has been emplaced into c with `std::forward<Args>(args)...`.

4 *Remarks*: Implementations should perform no more than one memory allocation.

```
constexpr node-handle(node-handle&& nh) noexcept;
```

~~1~~5 *Effects*: Constructs a *node-handle* object initializing *ptr_* with `nh.ptr_`. Move constructs *alloc_* with `nh.alloc_`. Assigns `nullptr` to `nh.ptr_` and assigns `nullopt` to `nh.alloc_`.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work.