Document number: P3049R0
Date: 2024-04-03
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

*node-handle*s for lists

Abstract

This paper proposes adding *node-handle* support to `list` and `forward_list`.

Tony Table

<table>
<tr><th>Before</th><th>Proposed</th></tr>
<tr><td><pre>
//given:
template<typename T>
void splice_if(list<T> & from, list<T> & to, T val) {
  const auto it{ranges::find(from, val)};

  if(it != from.end())
    to.splice(to.begin(), from, it);
}

//usage:
list<int> & l1 = …; //filled with random ints

//both lists must be available here to move an element
list<int> & l2 = …;
splice_if(l1, l2, 42);
</pre></td><td><pre>
//given:
template<typename T>
list<T>::node_type extract_if(list<T> & from, T val) {
  const auto it{ranges::find(from, val)};

  if(it != from.end()) return from.extract(it);
  return {};
}

//usage:
list<int> & l1 = …; //filled with random ints
auto nh{extract_if(l1, 42)};

//nh can be passed around independently!

// => extraction and insertion can be separated
list<int> & l2 = …;
if(nh) l2.insert(l2.end(), move(nh));
</pre></td></tr>
<tr><td><pre>
//given:
template<typename T>
void splice_if(forward_list<T> & from,
               forward_list<T> & to, T val) {
  //assume there is a ranges::find_before returning
  // the iterator before val or end()
  const auto it{ranges::find_before(from, val)};

  if(it != from.end())
    to.splice_after(to.before_begin(), from, it);
}

//usage:
forward_list<int> & l1 = …; //filled with random ints

//both lists must be available here to move an element
forward_list<int> & l2 = …;
splice_if(l1, l2, 42);
</pre></td><td><pre>
//given:
template<typename T>
auto extract_if(forward_list<T> & from, T val) {

  //assume there is a ranges::find_before returning
  // the iterator before val or end()
  const auto it{ranges::find_before(from, val)};

  if(it != from.end()) return from.extract_after(it);
  return forward_list<T>::node_type{};
}

//usage:
forward_list<int> & l1 = …; //filled with random ints
auto nh{extract_if(l1, 42)};

//nh can be passed around independently!

// => extraction and insertion can be separated
forward_list<int> & l2 = …;
if(nh) l2.insert_after(l2.before_begin(), move(nh));
</pre></td></tr>
</table>

Revisions

R0: Initial version

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

Motivation

[P0083] introduced the *node-handle* API to C++17 after extensive evaluation on adding `list::splice`-like operations to maps and sets. This novel approach wasn't without criticism, e.g. ES 8 from [P0448] reads:

> Node handles are an over-specified solution to the relatively simple problem of moving nodes between associative containers, which can be done with a more conservative interface similar to std::list::splice. There is a lack of consistency with std::list, where splicing and merging can be done but there is no node handle-based interface, yet lists are indeed node based, too. P00832 acknowledges the simpler solution (by Talbot) but dismisses it as it offered "no further advantages": however, the further advantages or use cases node handles allegedly provide are not clear at all.

Whilst we don't agree that advantages of *node-handles* aren't clear - the ability to modify keys in place between extraction and re-insertion, transferable between compatible containers, as well as the general ability to extract unmovable values warrants the new API - we agree with the criticism that there is a lack of consistency with node-based sequences. In order to remedy this, we propose adding a subset of the *node-handle* API to node-based sequence containers, namely `list` and `forward_list`.

We consider this important beyond the question of mere consistency as the *node-handle* API allows for better separation between source- and target-`list` compared to the existing `splice` functionality.

Design Space

The *node-handle* API for (unordered) associative containers can be summarized as follows:

```
using node_type = implementation defined specialization of node_handle;      0

node_type extract(const_iterator pos);                                       1
node_type extract(const key_type & key);                                     2
template<typename Key>
node_type extract(Key && key);                                               3

struct insert_return_type {                                                  4
    iterator  position;
    bool      inserted;
    node_type node;
};

insert_return_type insert(node_type && handle);                              5
iterator insert(const_iterator pos, node_type && handle);                    6
```

Removing aspects related to key lookups (2, 3, 5) and for handling key collisions (4), we arrive at the following API subset for node-based sequence containers, proposed verbatim for `list`:

```
using node_type = implementation defined specialization of node_handle;

node_type extract(const_iterator pos);
iterator insert(const_iterator pos, node_type && handle);
```

Note that whilst this API is syntactically consistent across all classes, the iterator parameter of `insert` has varying meanings:

- Ordered, associative: A location to insert as close as possible to.
- Unordered, associative: A hint for where search for an insertion point could start.
- Sequence: The actual insertion point.

As `forward_list` is singly-linked it cannot efficiently support the same API as other sequence containers. Therefore its API has been adapted in name and semantics, resulting in member functions like `erase_after` instead of `erase`. We follow this design principle and propose the following API:

```
using node_type = implementation defined specialization of node_handle;

node_type extract_after(const_iterator pos);
iterator insert_after(const_iterator pos, node_type && handle);
```

On cross container *node-handle* compatibility

An advanced feature of the *node-handle* API is the ability to transfer nodes between different, yet compatible containers (e.g. between `set<T, C1, A>` and `multiset<T, C2, A>`). Compatibility between containers that fit into the same category is rather easy to achieve, determining whether we can extend the required compatibility requires further analysis, that yields the following results:

Container	Required pointers in node	
	min	max
Binary search tree	2	3
Hashtable with separate chaining	1	2
Doubly linked list	2	
Singly linked list	1	

Note that implementing binary search trees with only two pointers per node, whist theoretically possible implies expensive stateful iterators and is therefore not used in real-world implementations. Further note that currently there is no requirement for `unordered_*`-containers to provide bidirectional iterators[2].

Evaluating existing implementations for node compatibility between hash tables and linked lists unveils implementation divergence: MS-STL[3] uses two pointers per hash table node whereas libstdc++[4] uses one. Therefore we don't propose additional node-type compatibilities.

Extracting multiple nodes at once

One could imagine an extension to the *node-handle* API that only makes sense for node-based sequence containers: extracting several consecutive nodes at once[5] and later batch inserting them. In order to make this work transparently, a *node-handle* would have to represent multiple nodes.

We don't propose this because we can't come up with a convincing use-case for such a facility and suggest future proposals in this area to introduce a dedicated *multi-node-handle* instead of changing the conceptual design of *node-handles*.

Impact on the Standard

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

Implementation Experience

The proposed design has been implemented at https://github.com/MFHava/STL/tree/P3049.

---

[2] Which semantically wouldn't make sense anyways.

[3] https://github.com/microsoft/STL/blob/d6efe9416e4ad7d6e245ae9e96023d413794d1eb/stl/inc/xhash#L332-L335

[4] https://github.com/gcc-mirror/gcc/blob/cebbaa2a84586a7345837f74a53b7a0263bf29ee/libstdc%2B%2B-v3/include/bits/hashtable_policy.h#L317

[5] like: `node_type extract(const_iterator first, const_iterator last)`

Proposed Wording

Wording is relative to [N4971]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[version.syn]

```
#define __cpp_lib_node_extract 201606YYYYMML //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]**

1 A *node handle* is an object that accepts ownership of a single element from a `list` ***[list]***, a `forward_list` ***[forward.list]***, an associative container (***[associative.reqmts]***), or an unordered associative container (***[unord.req]***). It may be used to transfer that ownership to another container with compatible nodes. Containers with compatible nodes have the same node handle type. Elements may be transferred in either direction between container types in the same row of ***[tab:container.node.compat]***.

[DRAFTING NOTE: Even though theoretically possible, we can't mandate additional compatibilities for various reasons.]

…

4 If a user-defined specialization of pair exists for `pair<const Key, T>` or `pair<Key, T>`, where Key is the container's `key_type` and T is the container's `mapped_type`, the behavior of operations involving node handles is undefined.

```
template<unspecified>
class node-handle {
public:
  // These type declarations are described in [container.requirements.general], [associative.reqmts], and [unord.req].
  using value_type     = see below; // not present for map containers
  using key_type       = see below; // ~~not~~only present for ~~set~~map containers
  using mapped_type    = see below; // ~~not~~only present for ~~set~~map containers
  using allocator_type = see below;
…
  // [container.node.observers], observers
  value_type& value() const;        // not present for map containers
  key_type& key() const;            // ~~not~~only present for ~~set~~map containers
  mapped_type& mapped() const;      // ~~not~~only present for ~~set~~map containers
```

[forward.list]

??.?.?.? Overview **[forward.list.overview]**

…

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class forward_list {
…
    using const_iterator = implementation-defined; // see [container.requirements]
    using node_type = see below;

    // [forward.list.cons], construct/copy/destroy
…
    template<container-compatible-range<T> R>
      iterator insert_range_after(const_iterator position, R&& rg);

    node_type extract_after(const_iterator position);
    iterator insert_after(const_iterator position, node_type&& nh);

    iterator erase_after(const_iterator position);
…
  };
}
```

4 An incomplete type T may be used when instantiating `forward_list` if the allocator meets the allocator completeness requirements (***[allocator.requirements.completeness]***). T shall be complete before any member of the resulting specialization of `forward_list` is referenced.

5 `node_type` is a specialization of a *`node-handle`* class template (***[container.node]***), such that the public nested types are the same types as the corresponding types in `forward_list`.

…

??.?.?.? Modifiers **[forward.list.modifiers]**

…

20 *Returns*: An iterator pointing to the last inserted element, or `position` if `rg` is empty.

`node_type extract_after(const_iterator position);`

21 *Preconditions*: The iterator following `position` is dereferenceable.

22 *Effects*: Removes the element pointed to by the iterator following `position`.

23 *Returns*: A `node_type` owning the removed element.

24 *Throws*: Nothing.

25 *Complexity*: Constant.

`iterator insert_after(const_iterator position, node_type&& nh);`

26 *Preconditions*: nh is empty or `get_allocator() == nh.get_allocator()` is `true`.

27 *Effects*: If nh is empty, has no effect and returns `end()`. Otherwise, inserts the element owned by nh after `position` and returns an iterator pointing to the newly inserted element.

28 *Postconditions*: nh is empty.

29 *Throws*: Nothing.

30 *Complexity*: Constant.

`iterator insert_after(const_iterator position, initializer_list<T> il);`

[list]

??.?.?.? Overview **[list.overview]**

...

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class list {
…
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type = see below;

    // [list.cons], construct/copy/destroy
…
    iterator insert(const_iterator position, initializer_list<T> il);

    node_type extract(const_iterator position);
    iterator insert(const_iterator position, node_type&& nh);

    iterator erase(const_iterator position);
…
  };
}
```

3 An incomplete type T may be used when instantiating `list` if the allocator meets the allocator completeness requirements (***[allocator.requirements.completeness]***). T shall be complete before any member of the resulting specialization of `list` is referenced.

4 `node_type` is a specialization of a *`node-handle`* class template (***[container.node]***), such that the public nested types are the same types as the corresponding types in `list`.

...

??.?.?.? Modifiers **[list.modifiers]**

...

`iterator insert(const_iterator position, initializer_list<T>);`

`node_type extract(const_iterator position);`

1 *Preconditions*: `position` is dereferenceable.

2 *Effects*: Removes the element pointed to by `position`.

3 *Returns*: A `node_type` owning the removed element.

4 *Throws*: Nothing.

5 *Complexity*: Constant.

`iterator insert(const_iterator position, node_type&& nh);`

6 *Preconditions*: nh is empty or `get_allocator() == nh.get_allocator()` is `true`.

7 *Effects*: If nh is empty, has no effect and returns `end()`. Otherwise, inserts the element owned by nh before `position` and returns an iterator pointing to the newly inserted element.

8 *Postconditions*: nh is empty.

9 *Throws*: Nothing.

10 *Complexity*: Constant.

```
template<class... Args> reference emplace_front(Args&&... args);
```

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading R0.