Doc No: N3059=10-0049

Date: 2010-03-12

Authors: Pablo Halpern
Intel Corp..
phalpern@halpernwightsoftware.com

Proposal to Simplify pair (rev 5.2)

Contents

Background

In the C++98 standard, the `pair` class template had only three constructors, excluding the compiler-generated copy-constructor. It was a very simple class template that could be easily understood. A number of language and library features were introduced since then. Constructors were added to take advantage of new language features as well as to implement new features in the `map`, `multimap`, `unordered_map` and `unordered_multimap` containers, for which `pair` plays a central role. Basically, these new constructors were added to support:

- Conversion-construction of the `first` and `second` members
- Move-construction of the pair as a whole, and of its individual members
- `emplace` functions in the map containers
- Passing an allocator to the `first` and `second` members for support of scoped allocators.

Unfortunately, most of these new features were orthogonal, causing a near doubling of the number of constructors to support each one. At one point, `pair` had 14 constructors (excluding the compiler-generated copy constructor)! That number has since been reduced to 9 by identifying redundant constructors. (An editorial error when removing concepts from the WP restored the redundant constructors, bringing the number back to 15, including the defaulted copy constructor.) The previous version of this paper (N3024) proposed a core language change that could be used to reduce the number of constructors, if not back to the 1998 set, at least to a manageable number. Unfortunately, these changes would have eliminated certain guarantees that tools sometimes rely on to detect when an object is constructed or destroyed. This version of the paper proposes no core language changes.

National Body comments and issues

This paper is provides part of a resolution to NB comments UK 241, US 65, US 77 and US 74.1. The remaining part of the resolution was provided by N2982, which was accepted into the WP in the Fall of 2009. This paper also addresses issue 1321.

Changes from N3024

- Removed all core language changes.
- Added piecewise constructor for `pair`.
- Added pack_arguments() function to build a tuple of references.
- Changed effects clauses of `pair` overloads of `scoped_allocator_adaptor::construct` to use the new piecewise constructor.

Changes from N2981

- Added core language to sections 3.8 and section 9 that introduce the notion of a fixed-layout class that can be constructed is pieces.
- Corrected the *effects* clauses of `construct` for `scoped_allocator_adaptor`.
- Updated numbering for N3000 and took into account post-Frankfurt regression whereby redundant constructors were added back in when concepts were removed.

Changes from N2945

- Fixed incorrect description of `scoped_allocator_adapator::construct` for pairs. (Description now matches reference implementation.)

- Miscellaneous corrections.

Changes from N2834

- N2945 and subsequent revisions reflect guidance from a straw poll of the LWG (at the March 2009 meeting in Summit, NJ) expressing interest in proposal 1, 2 and 3 of N2834. Proposal 0 (to do nothing) and proposal 4 (to create a general-purpose way to construct `pair` with arbitrary arguments) were removed.

- Concepts were removed and some additional normative text has been added to the `scoped_allocator_adaptor` section.

Document Conventions

All section names and numbers are relative to the, November 2009 WP, N3000.

> Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Discussion

Part of the problem with containers that are defined in terms of `pair` is the need to pass constructor arguments to both the `first` and `second` data members. This need resulted in a number of pair constructors that mirror the individual constructors of the data members and have nothing to do with `pair` itself. For example, the `emplace` proposal added a variadic constructor for the `second` part of the `pair`, even though such a constructor is not natural or otherwise useful. Similarly, the scoped allocator proposal added constructors that may supply an allocator argument to the construction of `first` and/or `second`. By providing a single constructor that can provide separate constructor arguments for `first` and `second` (piecewise construction), we eliminate the need to support a separate constructor for each possible argument combination.

This proposal introduces a new constructor for pair that takes a `tuple` of constructor arguments for `first`, and another `tuple` of constructor arguments for `second`. The rest of the proposal is to eliminate the `pair` constructors with variadic arguments and the `pair` constructors with allocator arguments. Instead, the `emplace` methods of ordered and unordered maps and multimaps will pass their variadic argument lists directly to the constructor of `second` (via the `tuple` argument) and four new overloads of the `construct` methods of `scoped_allocator_adaptor` will pass the inner allocator directly to constructors of `first` and `second`. In this way, the logic necessary to implement `emplace` and scoped allocators is put in the appropriate place, without distorting the `pair` interface. Elimination of the variadic and allocator-related constructors from `pair` reduces its constructor count (including the copy constructor) to 7.

Proposed Wording

20.3 Utility Components [utility]

Insert the following into the synopsis for `<utility>`:

```
// piecewise construction
struct piecewise_construct_t { };
constexpr piecewise_construct_t piecewise_construct =
   piecewise_construct_t();
template <class... Types> class tuple;   // defined in <tuple>
```

In the body of the section, add a description:

```
namespace std {
  struct piecewise_construct_t { };
  constexpr piecewise_construct_t piecewise_construct =
   piecewise_construct_t()
}
```

The `piecewise_construct_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, `pair` (20.3.4 [pairs]) has a constructor with `piecewise_contruct_t` as the first argument, immediately followed by two `tuple` (20.4) arguments used for piecewise construction of the elements of the `pair`.

20.3.4 Pairs [pairs.pair]

In `struct pair` remove the variadic and allocator-extended constructors and add the new piecewise constructor:

```
pair(const pair&) = default;
constexpr pair();
pair(const T1& x, const T2& y);
template<class U, class V> pair(U&& x, V&& y);
~~pair(pair&& p);~~
template<class U, class V> pair(const pair<U, V>& p);
template<class U, class V> pair(pair<U, V>&& p);
~~template<class U, class... Args>~~
~~    pair(U&& x, Args&&... args);~~
template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
         tuple<Args1...> first_args,
         tuple<Args2...> second_args);

~~// allocator-extended constructors~~
~~template <class Alloc> pair(allocator_arg_t, const Alloc& a);~~
~~template <class Alloc>~~
~~  pair(allocator_arg_t, const Alloc& a, const T1& x, const T2& y);~~
~~template <class U, class V, class Alloc>~~
~~  pair(allocator_arg_t, const Alloc& a, U&& x, V&& y);~~
~~template <class Alloc>~~
~~  pair(allocator_arg_t, const Alloc&, pair&& p);~~
~~template <class U, class V, class Alloc>~~
~~  pair(allocator_arg_t, const Alloc& a, const pair<U, V>& p);~~
~~template <class U, class V, class Alloc>~~
~~  pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);~~
~~template <class U, class... Args, class Alloc>~~
~~  pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);~~
```

Also remove the `uses_allocator` and `constructible_with_allocator_prefix` traits for pair from the synopsis as well as their descriptions in paragraphs 1 and 2:

```
~~  template <class T1, class T2, class Alloc>~~
~~    struct uses_allocator<pair<T1, T2>, Alloc>;~~
~~  template <class T1, class T2>~~
~~    struct constructible_with_allocator_prefix<pair<T1, T2>{>};~~
~~}~~

~~template <class T1, class T2, class Alloc>~~
~~  struct uses_allocator<pair<T1, T2>, Alloc> : true_type { };~~
```

~~*Requires*: Alloc shall be an `Allocator` (20.2.2).~~

~~[*Note*: Specialization of this trait informs other library components that pair can be constructed with an allocator, even though it does not have a nested allocator_type. — *end note*]~~

```
~~template <class T1, class T2>~~
~~  struct constructible_with_allocator_prefix<pair<T1, T2> >~~
~~    : true_type { };~~
```

~~[*Note*: Specialization of this trait informs other library components that pair can be constructed with an allocator prefix argument. — *end note*]~~

Remove ¶ 7 through ¶ 10 including the duplicate versions of the constructors above:

```
~~template<class U, class... Args>~~
~~    pair(U&& x, Args&&... args);~~
```

~~7 *Effects*: The constructor initializes first with std::forward<U>(x) and second with std::forward<Args>(args)...~~

~~8 …~~

~~9 …~~

~~10 *Effects*: The members `first` and `second` are both *allocator constructed* (20.8.7) with a.~~

Add a new paragraph to describe the new constructor:

```
template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
            tuple<Args1...> first_args,
            tuple<Args2...> second_args);
```

Requires: All of the types in `Args1` and `Args2` shall be `CopyConstructible` (Table 34). `T1` shall be constructible from `Args1`. `T2` shall be constructible from `Args2`.

Effects: The constructor initializes `first` with arguments of types `Args1...` obtained by *forwarding* the elements of `first_args` and initializes `second` with arguments of types `Args2...` obtained by *forwarding* the elements of `second_args`. (Here, forwarding an element x of type `U` within a `tuple` means to call `std::forward<U>(x)`). This form of construction, whereby constructor arguments for `first` and `second` are each provided in a separate `tuple`, is called *piecewise* construction.

2.4.1 Tuples [tuple]

Add `pack_arguments` to the synopsis of `<tuple>`:

```
template <class... Types>
  tuple<Vtypes...> make_tuple(Types&&...);
template <class... Types>
  tuple<Atypes...> pack_arguments(Types&&...);
```

20.4.2.4 Tuple creation functions [tuple.creation]

Add a description of `pack_arguments`:

```
template <class... Types>
  tuple< Atypes...> pack_arguments(Types&&... t);
```

Let Ti be each type in Types. Then each Ai in Atypes is Ti& if Ti is an array type and `std::add_rvalue_reference<Ti>::type` otherwise.

Effects: Constructs a tuple of references to the arguments in t suitable for forwarding as arguments to a function. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments (i.e., the program should typically not store the result in a named variable).

Returns: `tuple<`*Atypes*`...>(std::forward<Types>(t)...);`

Note: it is arguable that `add_rvalue_reference` should handle the array case directly. Is this an issue that should be addressed?

20.8.9 Scoped allocator adaptor [allocator.adaptor]

In section [allocator.adaptor] (20.8.7), add new `construct` members for `scoped_allocator_adapator`:

```
  template <class T, class... Args>
    void construct(T* p, Args&&... args);
  template<class T1, class T2, class... Args1, class... Args2>
    void construct(pair<T1,T2>* p, piecewise_construct_t,
                         tuple<Args1...> x, tuple<Args2...> y);
  template <class T1, class T2>
    void construct(pair<T1,T2>* p);
  template<class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, U&& x, V&& y);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, const pair<U,V>& x);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, pair<U,V>&& x);
```

In section [allocator.adaptor.members] (20.8.9.3), modify the `construct` and `destroy` functions (this addresses LWG 1321) and add descriptions of new `construct` functions:

Let *OUTERMOST*(x) be `x` if `x` does not have an `outer_allocator()` member function and *`OUTERMOST`*`(x.outer_allocator())` otherwise. Let *`OUTERMOST_ALLOC_TRAITS`*`(x)` be `allocator_traits<decltype(`*`OUTERMOST`*`(x))>`. [Note: *`OUTERMOST`*`(x)` and *`OUTERMOST_ALLOC_TRAITS`*`(x)` are recursive operations. It is incumbent upon the definition of `outer_allocator()` to ensure that the recursion terminates. It will terminate for all instantiations of `scoped_allocator_adaptor`. — end note]

```
template <class T, class... Args>
  void construct(T* p, Args&&... args);
```

Effects: ~~let OUTERMOST(x) be x if x does not have an outer_allocator() function, and OUTERMOST(x.outer_allocator()) otherwise.~~

- If `uses_allocator<T,inner_allocator_type>::value` is `false` and `is_constructible<T,Args...>::value` and `true`, calls ~~*`OUTERMOST(*this).`*~~ *`OUTERMOST_ALLOC_TRAITS(*this)::`*`construct(`*`OUTERMOST(*this),`*` p, std::forward<Args>(args)...)`.

- Otherwise, if (`uses_allocator<T,inner_allocator_type>::value` is `true` and `is_constructible<T,allocator_arg_t,inner_allocator_type,Args...>::value`) is `true`, then calls ~~*`OUTERMOST(*this).`*~~ *`OUTERMOST_ALLOC_TRAITS(*this)::`*`construct( `*`OUTERMOST(*this),`*` p, allocator_arg, inner_allocator(), std::forward<Args>(args)...)`.

- Otherwise, if (`uses_allocator<T,inner_allocator_type>::value` is `true` and `is_constructible<T,Args...,inner_allocator_type>::value`) is `true`, then calls ~~*`OUTERMOST(*this).`*~~ *`OUTERMOST_ALLOC_TRAITS(*this)::`*`construct(`*`OUTERMOST(*this),`*`p,std::forward<Args>(args)..., inner_allocator())`.

- Otherwise the program is ill formed. [*Note:* an error will result if `uses_allocator` evaluates true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. – *end note*]

```
template<class T1, class T2, class... Args1, class... Args2>
  void construct(pair<T1,T2>* p, piecewise_construct_t,
                    tuple<Args1...> x, tuple<Args2...> y);
```

Requires: All of the types in `Args1` and `Args2` shall be `CopyConstructible` (Table 34)

Effects: Constructs a `tuple`, `xprime`, from `x` by the following rules:

- If `uses_allocator<T1,inner_allocator_type>::value` is `false` and `is_constructible<T,Args1...>::value` is `true`, then `xprime` is `x`.

- Otherwise, if (`uses_allocator<T1,inner_allocator_type>::value` is `true` and `is_constructible<T1,allocator_arg_t,inner_allocator_type,Args1...>::value`) is `true`, then `xprime` is `tuple_cat(tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()), x)`.

- Otherwise, if (`uses_allocator<T1,inner_allocator_type>::value` is `true` and `is_constructible<T1,Args1...,inner_allocator_type>::value`) is `true`, then `xprime` is `tuple_cat(x, tuple<inner_allocator_type&>(inner_allocator())`.

- Otherwise the program is ill formed.

and constructs a `tuple`, `yprime`, from `y` by the following rules:

- If `uses_allocator<T2,inner_allocator_type>::value` is `false` and `is_constructible<T,Args2...>::value` is `true`, then `yprime` is `y`.

- Otherwise, if (`uses_allocator<T2,inner_allocator_type>::value` is `true` and `is_constructible<T2,allocator_arg_t,inner_allocator_type,Args2...>::value`) is `true`, then `yprime` is `tuple_cat(tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()), y)`.

- Otherwise, if (`uses_allocator<T2,inner_allocator_type>::value` is `true` and `is_constructible<T2,Args2...,inner_allocator_type>::value`) is `true`, then `yprime` is `tuple_cat(y, tuple<inner_allocator_type&>(inner_allocator())`.

- Otherwise the program is ill formed.

then this function calls `OUTERMOST_ALLOC_TRAITS(*this)::construct(OUTERMOST(*this), p, piecewise_construct, xprime, yprime)`.

```
template <class T1, class T2>
  void construct(pair<T1,T2>* p);
```

Effects: equivalent to `this->construct(p, piecewise_construct, tuple<>(), tuple<>());`

```
template<class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, U&& x, V&& y);
```

Effects: equivalent to this->construct(p, piecewise_construct,
pack_arguments(std::forward<U>(x)), pack_arguments(std::forward<V>(y)));

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, const pair<U,V>& x);
```

Effects: equivalent to this->construct(p, piecewise_construct,
pack_arguments(x.first), pack_arguments(x.second));

```
template <class T1, class T2, class U, class V>
  void construct(pair<T1,T2>* p, pair<U,V>&& x);
```

Effects: equivalent to this->construct(p, piecewise_construct,
pack_arguments(std::forward<U>(x.first)),
pack_arguments(std::forward<V>(x.second)));

```
template <class T>
  void destroy(T* p);
```

Effects: calls ~~*OUTERMOST*(*this).~~*OUTERMOST_ALLOC_TRAITS*(*this)::destroy(
OUTERMOST(*this),p)

Acknowledgements

I especially want to thank Christopher Jefferson for sharing his implementation of tuple-based construction with me. Thank you to Bjarne for moderating the discussion that directed me to this solution to the pair problem. Thanks, as always, go to Daniel Krügler, for his meticulous review.

References

N2982: Allocators post Removal of C++ Concepts
N2981: Several Proposals to Simplify pair (rev 3)
N3024: Several Proposals to Simplify pair (rev 4)