Doc No: N2834=09-0024
Date: 2009-02-09
Author: Pablo Halpern
Cilk Arts, Inc.

phalpern@halpernwightsoftware.com

Several Proposals to Simplify pair

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, nearly causing a 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. This paper proposes a number of approaches that could be used to reduce the number of constructors, if not back to the 1998 set, at least to a manageable number.

Document Conventions

All section names and numbers are relative to the October 2008 working draft, N2798.

> Existing and proposed working paper text is indented and shown in dark blue. Small 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. Large proposed insertions into the working paper are shown in the same dark blue indented format (no green underline).

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.

Proposal 0: Do nothing

Because of the conversion and move constructors, it is unlikely that we will reduce the set of `pair` constructors below 5. The current number stands at 9, so it is not unreasonable to consider leaving it at that.

Proposal 1: Allow separate construction of `first` and `second`

Discussion

Part of the problem with containers that use `pair` is the need to pass constructor arguments to both the `first` and `second` data members. This need results in a number of constructors that mirror the individual constructors of the data members and have nothing to do with `pair` itself. For example, the `emplace` functions on map containers requires constructing the `second` part of the `pair` with a variadic constructor, even though a variadic constructor is not natural for `pair`. This mechanism can be combined with other proposals (below) for a further reduction in the number of constructors. This proposal by itself does not reduce the number of constructors in `pair`, but it enables several other proposals, below.

Proposed Wording

Modify the introduction to pair in section 20.3.3 [pairs] as follows:

> The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if

they were tuple objects (see 20.5.2.4 and 20.5.2.5). In addition to the constructors provided, an object of a `pair` instantiation may be constructed in uninitialized memory of the correct size and alignment to hold that instantiation by separately constructing the `first` and `second` members of the `pair`.

Proposal 2: Move variadic Construction into `map`

Discussion

If Proposal 1 is adopted, `emplace` could separately construct `first` and `second`. We could then eliminate two variadic constructors from pair but this would require adding back one non-variadic constructor (for move-construction of `first` and `second`). This approach was implemented by Bloomberg in the first embodiment of the scoped allocator model.

Disadvantages

- Marginal benefit (but more possible in combination with other proposals)
- Does not allow variadic construction of local, static, or member `pair` variables.

Proposed Wording

(Adoption of Proposal 1 is assumed) In section 20.3.3 [pairs], remove two constructors from pair and add one back:

```
template<class U, class V>
  requires Constructible<T1, U&&> && Constructible<T2, V&&>
    pair(U&& x, V&& y);
template<class U, class... Args>
  requires Constructible<T1, U&&> && Constructible<T2, Args&&...>
    pair(U&& x, Args&&... args);
```

…

```
template<class U, class... Args, classAllocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, U&&>
        && ConstructibleWithAllocator<T2, Alloc, Args&&...>
  pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);
```

Add language to container requirements in section 23.1.1 [container.rquirements.general], at the end of paragraph 3:

Ordered and unordered associative containers described in this section which compose a `value_type` as a `pair<const Key, T>` construct each `pair` element by separately calling construct on the `first` (Key) and `second` (T) parts.

Proposal 3: Move Allocator-extended construction into scoped_allocator_adaptor

Discussion

The `pair` constructors that take an allocator argument exist to support scoped allocators. If Proposal 1 is adopted, we could consider moving those constructors into specialized `construct` methods within the `scoped_allocator_adaptor` templates. This would save 4 constructors in `pair`, at the cost of 8 new `construct` overloads in each adaptor. Combining this proposal with Proposal 2 would reduce the number of concept map templates to 6 for each adaptor.

Disadvantages

- Does not allow allocator-extended construction of local, static, or member `pair` variables.
- Does not scale well to other scoped-allocator-like ideas or other pair-like templates.

Proposed Wording

(This wording assumes passage of N2829. Minor tweaks could make it valid if N2829 doesn't pass. Assume adoption of Proposal 1.)

In Section 20.3.3 [pairs], remove the allocator-extended constructors from `pair`:

```
// allocator-extended constructors
template<classAllocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc>
        && ConstructibleWithAllocator<T2, Alloc>
  pair(allocator_arg_t, const Alloc& a);
template<class U, class V, classAllocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, const U&>
        && ConstructibleWithAllocator<T2, Alloc, const V&>
  pair(allocator_arg_t, const Alloc& a, const pair<U, V>& p);
template<class U, class V, classAllocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, RvalueOf<U>::type>
        && ConstructibleWithAllocator<T2, Alloc, RvalueOf<V>::type>
  pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);
template<class U, class... Args, classAllocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, U&&>
        && ConstructibleWithAllocator<T2, Alloc, Args&&...>
  pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);
```

In section 20.8.7 [allocator.adaptor], add the following `construct` members for each `scoped_allocator_adapator` and `scoped_allocator_adapator2`:

```
template <class T1, class T2>
```

```
 requires ConstructibleWithAllocator<T1,inner_allocator_type>
       && ConstructibleWithAllocator<T2,inner_allocator_type>
  void construct(pair<T1,T2>* p);
template <class T1, class T2, class U, class V>
 requires ConstructibleWithAllocator<T1,inner_allocator_type,const U&>
       && ConstructibleWithAllocator<T2,inner_allocator_type,const V&>
  void construct(pair<T1,T2>* p, const pair<U,V>& x);
template <class T1, class T2, class U, class V>
 requires ConstructibleWithAllocator<T1,inner_allocator_type,
                                     RvalueOf<U>::type>
       && ConstructibleWithAllocator<T2,inner_allocator_type,
                                     RvalueOf<V>::type>
  void construct(pair<T1,T2>* p, pair<U,V>&& x);
template <class T1, class T2, class U, class... Args>
 requires ConstructibleWithAllocator<T1,inner_allocator_type,U&&>
       && ConstructibleWithAllocator<T2,inner_allocator_type,Args&&...>
  void construct(pair<T1,T2>* p, U&& x, Args&&... args);

// stop recursion
template <class T1, class T2, Allocator Alloc2>
 requires ConstructibleWithAllocator<T1,Alloc2>
       && ConstructibleWithAllocator<T2,Alloc2>
  void construct(pair<T1,T2>* p, allocator_arg_t, const Alloc2&);
template <class T1, class T2, class U, class V, Allocator Alloc2>
 requires ConstructibleWithAllocator<T1,Alloc2,const U&>
       && ConstructibleWithAllocator<T2,Alloc2,const V&>
  void construct(pair<T1,T2>* p, allocator_arg_t, const Alloc2&,
                 const pair<U,V>& x);
template <class T1, class T2, class U, class V, Allocator Alloc2>
 requires ConstructibleWithAllocator<T1,Alloc2,
                                     RvalueOf<U>::type>
       && ConstructibleWithAllocator<T2,Alloc2,
                                     RvalueOf<V>::type>
  void construct(pair<T1,T2>* p, allocator_arg_t, const Alloc2&,
                 pair<U,V>&& x);
template <class T1, class T2, class U, class... Args, Allocator Alloc2>
 requires ConstructibleWithAllocator<T1,Alloc2,U&&>
       && ConstructibleWithAllocator<T2,Alloc2,Args&&...>
  void construct(pair<T1,T2>* p, allocator_arg_t, const Alloc2&,
                 U&& x, Args&&... args);
```

Proposal 4: `arg_tuple` constructors

Discussion

This is an alternative to Proposal 1 for allowing arbitrary constructor arguments to be passed to the `first` and `second` members of `pair`. It should be possible to create a concept for constructing any type from a `tuple`-like object containing the type's constructor arguments. I'll call that type `arg_tuple`. A `pair` constructor could be added that accepts two such "packaged" constructor arguments and passes each one to the constructors of `first` and

`second` accordingly. The advantage of this system is that it is general-purpose (can be used outside of `pair`) and allows local, global, and member variables of `pair` type to be constructed with allocators or other constructor arguments.

Disadvantages

- Inventive – late for this stage of the standard
- No current implementation – requires full concept support from the compiler.

Proposed wording

This wording is incomplete. If there is interest, I can flesh it out.

Add a new `arg_tuple` template that holds *references* to arguments:

```
template <class... Args>
  class arg_tuple : public tuple<Args&...> { ... }
```

Add a new constructor to pair:

```
template <class... A1, class... A2>
  pair(arg_tuple<A1...>, arg_tuple<A2...>);
```

References

N2810: Defects and Proposed Resolutions for Allocator Concepts (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2810.pdf)

N2829: : Defects and Proposed Resolutions for Allocator Concepts (rev 1) (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2829.pdf)