Doc No: N2908=09-0098

Date: 2009-06-21

Authors: Pablo Halpern
Cilk Arts, Inc.
phalpern@halpernwightsoftware.com

Alan Talbot
cpp@alantalbot.com

Several Proposals to Simplify pair (Rev 1)

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. The previous version of this paper (N2834) proposed 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.

Changes from N2834

This revision reflects 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. Some additional normative text has been added to the `scoped_allocator_adaptor` section.

Document Conventions

All section names and numbers are relative to the March 2009 working draft, N2857.

> 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 constructing the members of `pair` separately (without calling a `pair` constructor) we can eliminate the need for these extra constructors.

This proposal is to eliminates 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` and three new overloads of the `construct` method of `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` will pass the inner allocator directly to constructors of `first` and `second`, without calling the `pair` constructor.

In this way, the logic necessary to implement `emplace` and scoped allocators is put in the appropriate place, without distorting the `pair` interface.

Removing the variadic constructors from `pair` requires adding an r-value reference constructor for move-construction of first and second. (This functionality was handled by one of the variadic versions.) The effective change to pair in this proposal is the elimination of five constructors and the reinstatement of one constructor, for a net reduction of four constructors.

Proposed Wording

20.3.3 Pairs [pairs]

Add language to the introduction in ¶ 1 as follows:

1 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 20.5.2.3 and 20.5.2.4). As an alternative to the constructors provided, an object of a `pair` instantiation may be constructed in uninitialized memory of the correct size and alignment by separately constructing the `first` and `second` members, e.g., using placement new (18.6.1.3 [new.delete.placement]). [*Example*:

```
  pair<X, Y> *p = ::operator new(sizeof(pair<X,Y>));
  new ((void*)&p->first) X(arg1, arg2, arg3);
  new ((void*)&p->second) Y(arg4, arg5);
  //  *p is now fully constructed
```

Note that this example is not exception-safe – *end example*]

We know that something needs to be done to ensure that "constructing" pair this way is legal, but we are not certain that this description is the best way to do it. Alan believes pair should not be singled out for special treatment.   It is hard to imagine an implementation where the above example would not "just work," but there is nothing in the standard that allows an object to be constructed in pieces like this, even if the object being constructed has no virtual functions and no virtual inheritance. Both Alan and Pablo agree that a general language feature would be preferable to special treatment for `pair` and would be happy to remove this description if such a feature were adopted by Core.

In `struct pair` remove the variadic and allocator-extended constructors and add a member-wise move constructor:

```
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);~~
```

```
~~// 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);~~
```

Remove ¶ 6 through ¶ 10 including the duplicate versions of the constructors above:

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

~~6 *Effects*: The constructor initializes first with std::forward<U>(x) and second with std::forward<Args>(args)...~~

~~7 ...~~

~~8 ...~~

~~9 ...~~

~~10 *Effects*: The members first and second are each constructed as ConstructibleWithAllocator objects with constructor arguments (allocator_arg_t(), a, std::forward<U>(x)) and (allocator_ arg_t(), a, std::forward<Args>(args)...), respectively.~~

and insert a new ¶ 6:

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

6 *Effects*: The constructor initializes `first` with `std::forward<U>(x)` and `second` with `std::forward<V>(y).`

20.8.7 Scoped allocator adaptor [allocator.adaptor]

In section [allocator.adaptor] (20.8.7), add new `construct` members for each `scoped_allocator_adapator` and `scoped_allocator_adapator2`:

```
template<Allocator Alloc>
  class scoped_allocator_adaptor
```

```
    : public scoped_allocator_adaptor_base<Alloc>
{
  typedef Alloc outer_allocator_type;
  typedef Alloc inner_allocator_type;

  requires DefaultConstructible<Alloc> scoped_allocator_adaptor();
  scoped_allocator_adaptor(scoped_allocator_adaptor&&);
  scoped_allocator_adaptor(const scoped_allocator_adaptor&);
  scoped_allocator_adaptor(Alloc&& outerAlloc);
  scoped_allocator_adaptor(const Alloc& outerAlloc);

  template <Allocator Alloc2>
   requires Convertible<Alloc2&&, Alloc>
    scoped_allocator_adaptor(scoped_allocator_adaptor<Alloc2>&&);
  template <Allocator Alloc2>
   requires Convertible<const Alloc2&, Alloc>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<Alloc2>&);

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

  // stop recursion
  template <class T, Allocator Alloc2, class... Args>
   requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
    void construct(T* p, allocator_arg_t,
                   const Alloc2&, Args&&... args);

  const Alloc& outer_allocator() const;
  const Alloc& inner_allocator() const;
};

template<Allocator OuterA, Allocator InnerA>
  class scoped_allocator_adaptor2
    : public scoped_allocator_adaptor_base<OuterA>
{
public:
  typedef OuterA outer_allocator_type;
  typedef InnerA inner_allocator_type;

  requires DefaultConstructible<OuterA> && DefaultConstructible<InnerA>
    scoped_allocator_adaptor2();
  scoped_allocator_adaptor2(scoped_allocator_adaptor2&& other);
  scoped_allocator_adaptor2(const scoped_allocator_adaptor2& other);
  scoped_allocator_adaptor2(OuterA&& outerAlloc,
                            InnerA&& innerAlloc);
  scoped_allocator_adaptor2(const OuterA& outerAlloc,
                            const InnerA& innerAlloc);

  template <Allocator OuterA2, Allocator InnerA2>
   requires Convertible<OuterA2&&, OuterA>
         && Convertible<InnerA2&&, InnerA>
    scoped_allocator_adaptor2(
                      scoped_allocator_adaptor2<OuterA2,InnerA2>&&);
  template <Allocator OuterA2, Allocator InnerA2>
   requires Convertible<const OuterA2&, OuterA>
         && Convertible<const InnerA2&, InnerA>
    scoped_allocator_adaptor2(
      const scoped_allocator_adaptor2<OuterA2,InnerA2>&);

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

  // Recursion stop
  template <class T, Allocator Alloc2, class... Args>
   requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
    void construct(T* p, allocator_arg_t,
                   const Alloc2&, Args&&... args);

  const OuterA& outer_allocator() const;
  const InnerA& inner_allocator() const;

private:
    inner_allocator_type inner_alloc;  // for exposition only
  };
```

In section [allocator.adaptor.members] (20.8.7.4), add descriptions of new `construct` functions:

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

Effects: `outer_allocator().construct(p, allocator_arg_t, inner_allocator(), forward<Args>(args)...)`

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

Effects: outer_allocator().construct(&p->first, allocator_arg_t, inner_allocator()); outer_allocator().construct(&p->second, allocator_arg_t, inner_allocator());

```
  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);
```

Effects: outer_allocator().construct(&p->first, allocator_arg_t, inner_allocator(), x.first); outer_allocator().construct(&p->second, allocator_arg_t, inner_allocator(), x.second);

```
  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);
```

Effects: outer_allocator().construct(&p->first, allocator_arg_t, inner_allocator(), move(x.first)); outer_allocator().construct(&p->second, allocator_arg_t, inner_allocator(), move(x.second));

```
  template <class T, Allocator Alloc2, class... Args>
   requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
      void construct(T* p, allocator_arg_t,
                     const Alloc2& a2, Args&&... args);
```

Effects: `outer_allocator().construct(p, allocator_arg_t, a2, forward<Args>(args)...);`

Note: this overloaded version of `construct` prevents recursion into ever deeper inner allocators when the outer allocator is itself a scoped allocator adaptor.

23.2.1 General container requirements [container.requirements.general]

Add language to container requirements at the end of ¶ 3:

3 … then construct_element may pass an inner allocator argument to T's constructor. *—end note*] 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.

We are not certain that this language is actually necessary. Alan believes it may not be necessary to specify how the implementation constructs pair. Pablo would prefer to be explicit. This addition can be removed if there is consensus that it is unnecessary.

References

N2834: Several Proposals to Simplify pair (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2834.pdf)

N2840: Defects and Proposed Resolutions for Allocator Concepts (Rev 2) (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2840.pdf)