**Doc No:** N3024=10-0014
**Date:** 2010-02-15
**Authors:** Pablo Halpern
Intel Corp..
phalpern@halpernwightsoftware.com

# Proposal to Simplify pair (rev 4)

## Contents

Background ..... 1
Changes from N2981 ..... 2
Changes from N2945 ..... 2
Changes from N2834 ..... 2
Document Conventions ..... 2
Discussion ..... 3
Proposed Wording ..... 3
Acknowledgements ..... 10
References ..... 10

## 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 (N2981) 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 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 constructing the members of `pair` separately (without calling a `pair` constructor) we can eliminate the need for these extra constructors. This simplification is not currently supported by the core language. What is needed is the ability to construct pair-like objects one member at a time.

This proposal introduces a new class category, "fixed-layout class," which is a slightly less restrictive category than standard-layout class and which would allow member-wise construction. 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` and four new overloads of the `construct` methods of `scoped_allocator_adaptor` 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. Elimination of the variadic and allocator-related constructors from `pair` reduces its constructor count (including the copy constructor) to 7 (or 6 if the redundant move constructor is removed, as per the request for guidance on page 8).

## Proposed Wording

### 3.8 Object Lifetime [basic.life]

Modify paragraphs 5 and 6 as follows:

5 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[37] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.4.2), and using the pointer as if the pointer were of type `void*`, is well-defined. Such a pointer may be dereferenced but the

resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

— the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a delete-expression,

— the object will be or was of a class that is not a fixed-layout class (Clause 9) and the pointer is used to access a non-static data member or call a non-static member function of the object, or

— the object will be or was of a class that is not a fixed-layout class (Clause 9) and the pointer is implicitly converted (4.10) to a pointer to a base class type, or

— the pointer is used as the operand of a `static_cast` (5.2.9) (except when the conversion is to `void*`, or to `void*` and subsequently to `char*`, or `unsigned char*`), or

— the pointer is used as the operand of a `dynamic_cast` (5.2.7). [*Example: ...* -- *end example*]

6 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.4.2), and using the properties of the lvalue which do not depend on its value is well-defined. The program has undefined behavior if:

— an lvalue-to-rvalue conversion (4.1) is applied to such an lvalue,

— the object will be or was of a class that is not a fixed-layout class (Clause 9) and the lvalue is used to access a non-static data member or call a non-static member function of the object, or

— the object will be or was of a class that is not a fixed-layout class (Clause 9) and the lvalue is implicitly converted (4.10) to a reference to a base class type, or

— the lvalue is used as the operand of a `static_cast` (5.2.9) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&` or when the object will be or was of a class that is not a fixed-layout class (Clause 9) and the conversion is to a reference to a (possibly cv-qualified) base class type, or

— the lvalue is used as the operand of a `dynamic_cast` (5.2.7) or as the operand of `typeid`.

— Insert the following new paragraphs between paragraphs 6 and 7:

— The text below is the critical text for allowing a pair to be constructed by separately constructing the `first` and `second` data members.

A pointer or lvalue that refers to allocated storage occupied by an object of fixed-layout class type (Clause 9) can be used to access a non-static data member or base-class subobject of that object. If the lifetime of the object has not yet begun or has already ended, and if the data member or base-class type has non-trivial initialization, then such an access yields an lvalue to allocated storage, subject to the restrictions on such lvalues, above.

The lifetime of an object of fixed-layout class (Clause 9) begins when all of its subobjects with non-trivial initialization have been initialized, even if the constructor for the object has not been invoked. A program shall invoke a destructor on the object only if all such subobjects have been fully initialized, otherwise the behavior of the program is undefined. [*Example:*

```
class A {
  public:
    A(int);   // Non-trivial constructor
};

struct Fixed { // fixed-layout class
  int x;
  A   y;
};

Fixed* makeFixed(int v, int w) {
  Fixed* p = ::operator new(sizeof(Fixed));   // allocated storage
  try {
    p->x = v;
    ::new(&p->y) A(w);   // Initialize y. Might throw.
  }
  catch (...) {
    p->~Fixed();         // Undefined behavior: p->y has not been initialized.
    ::operator delete(p);
    throw;
  }
  return p;              // OK. *p is fully initialized and its lifetime has begin.
}
```

-- *end example*]

Similarly, the lifetime of an object of fixed-layout class (Clause 9) ends when any of its subobjects having non-trivial destructors have been destroyed, even if the destructor for the object has not been invoked. A program shall ensure that all such subobjects are properly destroyed and that the destructor of the object is not invoked after its lifetime has ended, otherwise the behavior of the program is undefined.

It is hard to imagine an implementation where the above example would not "just work," but there is nothing in the current WP 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. The alternative to such a core language feature would be to give special treatment for `pair`. Most people would find such special treatment to be distasteful.

### 9 Classes [class]

Modify paragraph 6 as follows:

6 A *fixed-layout* class is a class that:

— has no virtual functions (10.3) and no virtual base classes (10.1) (including inherited virtual functions and indirect virtual base classes) and

— has the same access control (Clause 11) for all non-static data members,

A *fixed-layout struct* is a fixed -layout class defined with the *class-key* `struct` or the *class-key* `class`. A *fixed -layout union* is a fixed -layout class defined with the *class-key* `union`.

A *standard-layout* class is a fixed-layout class that:

— has no non-static data members of type non-standard-layout class (or array of such types) or reference,

~~— has no virtual functions (10.3) and no virtual base classes (10.1),~~

~~— has the same access control (Clause 11) for all non-static data members,~~

— has no non-standard-layout base classes,

— either has no non-static data members in the most-derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and

— has no base classes of the same type as the first non-static data member.[103]

Modify the example in paragraph 9 as follows:

[ *Example:*

```
struct N { // neither trivial nor standard-layout  none of trivial, standard-layout or fixed-layout
  int i;
  int j;
  virtual ~N();
};

struct T { // trivial but not fixed-layout (nor standard-layout)
  int i;
private:
  int j;
};

struct FL { // fixed-layout but neither standard-layout nor trivial
  N n;
  int i;
  FL();
  ~FL();
};

struct SL { // standard-layout (and fixed-layout) but not trivial
  int i;
  int j;
  ~SL();
};

struct POD { // both trivial and standard-layout (and fixed-layout)
  int i;
```

```
  int j;
};
```

—*end example* ]

Effectively, a fixed-layout class is one where the location and size of each *top-level* subobject is fixed at compile-time. A standard-layout class is a "deep" fixed-layout class, i.e., a fixed-layout class that has fixed layout all the way down to the leaves (and has no references).

Modify the second sentence of [support.types], paragraph 4 as follows:

4 The macro `offsetof(`*type*, *member-designator*`)` accepts a restricted set of type arguments in this International Standard. If type is not a ~~standard~~fixed-layout class (Clause 9), the results are undefined.[197] The expression `offsetof(`*type*, *member-designator*`)` is never type-dependent (14.7.2.2) and it is value-dependent (14.7.2.3) if and only if *type* is dependent. The result of applying the `offsetof` macro to a field that is a `static` data member, a reference data member, or a function member is undefined.

In [meta.unary.comp], add a row to Table 43, as follows:

| | | |
|---|---|---|
| `template <class T> struct is_standard_layout;` | `T` is a standard-layout type (3.9) | `T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound |
| `template <class T> struct is_fixed_layout;` | `T` is a fixed-layout class type (Clause 9), or a standard-layout type (3.9) | `T` shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound |

Most uses of *standard-layout* in the WP should be unchanged. I've identified only the above two cases (`offsetof` and `is_fixed_layout`) where a use of standard-layout should be extended to include fixed-layout.

Section 9.2 [class.mem], pp 14-17 (description of *layout compatible*) could be extended to apply to fixed-layout classes by extending the notion of layout compatible to its base classes. This extension is not needed to solve the `pair` problem, and I want to make sure that there is sufficient motivation (and no technical objections) before proposing this change.

### 20.3.4 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 20.5.2.5 and 20.5.2.6). A specialization of the `pair` template is a fixed-layout struct (Clause 9).

In `struct pair` remove the variadic and allocator-extended constructors:

```
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);   [See request for guidance, below]
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);~~

~~// 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);~~
```

Are `pair(pair&& p)` and `template<class U, class V> pair(pair<U, V>&& p)` both necessary? It would seem like the latter would subsume the former. The former constructor was not in the pre-Frankfurt WP (June 2009, N2914), which was the last WP before concepts were removed. Perhaps the overload is necessary to prevent some kind of ambiguity or to match conversions to pair? I have not removed it here, but we could choose to remove it in a revision of this paper or as an issue.

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.~~

### 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>
    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), add descriptions of new `construct` functions:

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

*Effects:*
```
this->construct(std::addressof(p->first));
    this->construct(std::addressof(p->second));
```

*Throws*: if an exception is thrown while constructing `p->second`, then the destructor for `p->first` is invoked. Any exception thrown by either `construct` call is rethrown.

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

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

*Effects:*
```
this->construct(std::addressof(p->first),std::forward<U>(x));
this->construct(std::addressof(p->second),std::forward<V>(y));
```

*Throws*: if an exception is thrown while constructing `p->second`, then the destructor for `p->first` is invoked. Any exception thrown by either `construct` call is rethrown.

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

*Effects:*
```
this->construct(std::addressof(p->first), x.first);
this->construct(std::addressof(p->second), x.second);
```

*Throws*: if an exception is thrown while constructing `p->second`, then the destructor for `p->first` is invoked. Any exception thrown by either `construct` call is rethrown.

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

*Effects:*
```
this->construct(std::addressof(p->first),std::move(x.first));
this->construct(std::addressof(p->second),std::move(x.second));
```

*Throws*: if an exception is thrown while constructing `p->second`, then the destructor for `p->first` is invoked. Any exception thrown by either `construct` call is rethrown.

## Acknowledgements

Thank you to Clark Nelson and Jens Maurer for reviewing the core language changes.

## References

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