Concepts for the C++0x Standard Library: Utilities (Revision 3)

Douglas Gregor and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, lums}@osl.iu.edu

Document number: N2622=08-0132
Revises document number: N2322=07-0182
Date: 2008-05-19
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 20 of the C++ Standard Library in order to make full use of concepts [1]. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the latest working draft of the C++ standard (N2588). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications..

Changes since N2322

- Added concept requirements to `make_pair` and `make_tuple`.
- Switched from `ObjectType` to `VariableType` in several components where we need to permit reference types.
- Many cleanups to bring specification up-to-date with changes to concepts wording and the core concepts.

Chapter 20 General utilities library [utilities]

2 The following clauses describe utility and allocator ~~requirements~~concepts, utility components, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 30.

Table 30: General utilities library summary

Subclause	Header(s)
20.1 ~~Requirements~~Concepts	`<concepts>`
20.2 Utility components	`<utility>`
20.3 Tuples	`<tuple>`
[meta] Type traits	`<type_traits>`
20.5 Function objects	`<functional>`

[memory] Memory	`<memory>`
	`<cstdlib>`
	`<cstring>`

[date.time] Date and time	`<ctime>`

20.1 Concepts [utility.concepts]

This new section is specified in a separate document, N2621=08-0131.

20.2 Utility components [utility]

1 This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header `<utility>` synopsis

```
namespace std {
  // 20.2.1, operators:
  namespace rel_ops {
    template<classEqualityComparable T> bool operator!=(const T&, const T&);
    template<classLessThanComparable T> bool operator> (const T&, const T&);
    template<classLessThanComparable T> bool operator<=(const T&, const T&);
    template<classLessThanComparable T> bool operator>=(const T&, const T&);
  }

  // 20.2.2, forward/move:
  template <class T> struct identity;
  template <classVariableType T> T&& forward(typename identity<T>::type&&);
```

```
template <classVariableType T> typename remove_reference<T>::type&& move(T&&);

// 20.2.3, pairs:
template <classObjectType T1,classObjectType T2> struct pair;
template <classEqualityComparable T1, classEqualityComparable T2>
  bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
template <classLessThanComparable T1, classLessThanComparable T2>
  bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
template <classEqualityComparable T1, classEqualityComparable T2>
  bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
template <classLessThanComparable T1, classLessThanComparable T2>
  bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
template <classLessThanComparable T1, classLessThanComparable T2>
  bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
template <classLessThanComparable T1, classLessThanComparable T2>
  bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
template <classSwappable T1, classSwappable T2>
  void swap(pair<T1,T2>&, pair<T1,T2>&);
template <classSwappable T1, classSwappable T2>
  void swap(pair<T1,T2>&&, pair<T1,T2>&);
template <classSwappable T1, classSwappable T2>
  void swap(pair<T1,T2>&, pair<T1,T2>&&);
template <classMoveConstructible T1,
classMoveConstructible T2> pair<V1, V2> make_pair(T1, T2);

// 20.2.3, tuple-like access to pair:
template <class T> class tuple_size;
template <int I, class T> class tuple_element;

template <class T1, class T2> struct tuple_size<std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<0, std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<1, std::pair<T1, T2> >;

template<int I, class T1, class T2>
  requires True<0 <= I && I < 2>
  P& get(std::pair<T1, T2>&);
template<int I, class T1, class T2>
  requires True<0 <= I && I < 2>
  const P& get(const std::pair<T1, T2>&);
}
```

20.2.1 Operators [operators]

By adding concept constraints to the operators in `rel_ops`, we eliminate nearly all of the problems with `rel_ops` that caused them to be banished. We could consider bringing them back into namespace `std`, if they are deemed useful.

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```
template <classEqualityComparable T> bool operator!=(const T& x, const T& y);
```

2 ~~Requires: Type T is EqualityComparable (20.1.1).~~

3 *Returns:* `!(x == y)`.

template <~~class~~LessThanComparable T> bool operator>(const T& x, const T& y);

4 ~~Requires: Type T is LessThanComparable (20.1.2).~~

5 *Returns:* `y < x`.

template <~~class~~LessThanComparable T> bool operator<=(const T& x, const T& y);

6 ~~Requires: Type T is LessThanComparable (20.1.2).~~

7 *Returns:* `!(y < x)`.

template <~~class~~LessThanComparable T> bool operator>=(const T& x, const T& y);

8 ~~Requires: Type T is LessThanComparable (20.1.2).~~

9 *Returns:* `!(x < y)`.

10 In this library, whenever a declaration is provided for an `operator!=`, `operator>`, `operator>=`, or `operator<=`, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this clause.

20.2.2 forward/move helpers [forward]

1 The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions.

```
template <class T> struct identity { typedef T type; };
```

2 [*Note:* The use of `identity` in `forward` forces users to explicitly specify the template parameter. This is necessary to get the correct forwarding semantics. — *end note*]

template <~~class~~VariableType T> T&& forward(typename identity<T>::type&& t);

3 *Returns:* `t`.

template <~~class~~VariableType T> typename remove_reference<T>::type&& move(T&& t);

7 *Returns:* `t`.

20.2.3 Pairs [pairs]

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 20.3.1.4 and 20.3.1.5).

template <~~class~~VariableType T1, ~~class~~VariableType T2>
```
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
```

```
T1 first;
T2 second;
requires DefaultConstructible<T1> && DefaultConstructible<T2> pair();
requires CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);
template<class U, class V>
  requires HasConstructor<T1, U&&> && HasConstructor<T2, V&&>
  pair(U&& x, V&& y);
requires MoveConstructible<T1> && MoveConstructible<T2> pair(pair&& p);
template<class U, class V>
  requires HasConstructor<T1, U> && HasConstructor<T2, V>
  pair(const pair<U, V>& p);
template<class U, class V>
  requires HasConstructor<T1, U&&> && HasConstructor<T2, V&&>
  pair(pair<U, V>&& p);
template<class U, class... Args>
  requires HasConstructor<T1, U&&> && HasConstructor<T2, Args&&...>
  pair(U&& x, Args&&... args);

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

requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p );
template<class U , class V>
  requires MoveAssignable<T1, U> && MoveAssignable<T2, V>
  pair& operator=(pair<U , V>&& p );
```

```
    requires Swappable<T1> && Swappable<T2> void swap(pair&& p );
  };

  template <class T1, class T2, class Alloc>
    struct uses_allocator<pair<T1, T2>, Alloc>;

  template <class T1, class T2, class Alloc>
    concept_map UsesAllocator<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 { };
```

2 *Requires:* Alloc shall be an Allocator ([allocator.requirements]).

3 [*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*]

```
requires DefaultConstructible<T1> && DefaultConstructible<T2> pair();
```

4 *Effects:* Initializes its members as if implemented: `pair() : first(), second() {}`

```
requires CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);
```

5 *Effects:* The constructor initializes `first` with *x* and `second` with *y*.

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

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

```
requires MoveConstructible<T1> && MoveConstructible<T2> pair(pair&& p);
```

7 *Effects:* The constructor initializes `first` with `std::move(p.first)` and `second` with `std::move(p.second)`.

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

8 *Effects:* Initializes members from the corresponding members of the argument, performing implicit conversions as needed.

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

```
  requires Constructible<T1, U&&> && Constructible<T2, V&&>
  pair(pair<U, V>&& p);
```

9 *Effects:* The constructor initializes `first` with `std::move(p.first)` and `second` with `std::move(p.second)`.

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

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

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

11 ~~*Requires:* `Alloc` shall be an `Allocator` ([allocator.requirements]).~~

12 *Effects:* The members `first` and `second` are both *allocator constructed* ([allocator.concepts]) with `a`.

```
requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p);
```

13 *Effects:* Assigns to `first` with `std::move(p.first)` and to `second` with `std::move(p.second)`.

14 *Returns:* `*this`.

```
template<class U, class V>
  requires MoveAssignable<T1, U> && MoveAssignable<T2, V>
  pair& operator=(pair<U, V>&& p);
```

15 *Effects:* Assigns to `first` with `std::move(p.first)` and to `second` with `std::move(p.second)`.

16 *Returns:* `*this`.

```
requires Swappable<T1> && Swappable<T2> void swap(pair&& p);
```

17 *Effects:* Swaps first with p.first and second with p.second.

18 ~~*Requires:* first_type and second_type must be Swappable.~~

```
template <classEqualityComparable T1, classEqualityComparable T2>
  bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

19 *Returns:* x.first == y.first && x.second == y.second.

```
template <classLessThanComparable T1, classLessThanComparable T2>
  bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

20 *Returns:* x.first < y.first || (!(y.first < x.first) && x.second < y.second).

```
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>& y);
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>&& x, pair<T1, T2>& y);
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>&& y);
```

21 *Effects:* x.swap(y)

```
template <classMoveConstructible T1, classMoveConstructible T2>
  pair<V1, V2> make_pair(T1 x, T2 y);
```

22 *Returns:*

```
pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));
```

where V1 and V2 are determined as follows: Let Ui be decay<Ti>::type for each Ti. Then each Vi is X& if Ui equals reference_wrapper<X>, otherwise Vi is Ui.

23 [*Example:* In place of:

```
return pair<int, double>(5, 3.1415926);    // explicit types
```

a C++ program may contain:

```
return make_pair(5, 3.1415926);            // types are deduced
```

— *end example*]

```
tuple_size<pair<T1, T2> >::value
```

24 *Returns:* integral constant expression.

25 *Value:* 2.

```
tuple_element<0, pair<T1, T2> >::type
```

26 *Value:* the type `T1`.

```
tuple_element<1, pair<T1, T2> >::type
```

27 *Value:* the type T2.

```
template<int I, class T1, class T2>
  requires True<0 <= I && I < 2>
  P& get(pair<T1, T2>&);

template<int I, class T1, class T2>
  requires True<0 <= I && I < 2>
  const P& get(const pair<T1, T2>&);
```

28 *Return type:* If `I == 0` then `P` is `T1`, ~~if I == 1 then~~otherwise `P` is `T2`~~, and otherwise the program is ill-formed~~.

29 *Returns:* If `I == 0` returns `p.first`, otherwise returns `p.second`.

20.3 Tuples [tuple]

1 20.3 describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

2 **Header `<tuple>` synopsis**

```
namespace std {
  // 20.3.1, class template tuple:
  template <classVariableType... Types> class tuple;

  // 20.3.1.3, tuple creation functions:
  const unspecified ignore;

  template <classMoveConstructible... Types>
    tuple<VTypes...> make_tuple(Types&&...);

  template<classVariableType... Types>
    tuple<Types&...> tie(Types&...);

  template <classCopyConstructible... TTypes, classCopyConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template <classMoveConstructible... TTypes, classCopyConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, const tuple<UTypes...>&);
  template <classCopyConstructible... TTypes, classMoveConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, tuple<UTypes...>&&);
  template <classMoveConstructible... TTypes, classMoveConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, tuple<UTypes...>&&);

  // 20.3.1.4, tuple helper classes:
  template <class T> class tuple_size;  // undefined
  template <classVariableType... Types> class tuple_size<tuple<Types...> >;
```

```
template <int I, class T> class tuple_element;     // undefined
template <int I, class...  Types>
  requires True<0 <= I && I < sizeof...(Types)> class tuple_element<I, tuple<Types...> >;

// 20.3.1.5, element access:
template <int I, classVariableType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>&);

template <int I, classVariableType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&);

// 20.3.1.6, relational operators:
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class...  TTypes, class...  UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class...  TTypes, class...  UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class...  TTypes, class...  UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class...  TTypes, class...  UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class...  TTypes, class...  UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

} // namespace std
```

20.3.1 Class template `tuple` [tuple.tuple]

```
template <classVariableType... Types>
class tuple
{
public:
  requires DefaultConstructible<Types>... tuple();
  requires CopyConstructible<Types>... explicit tuple(const Types&...);
  template <class... UTypes>
```

```
    requires HasConstructor<Types, UTypes&&>...
    explicit tuple(UTypes&&...);

  requires CopyConstructible<Types>... tuple(const tuple&);
  requires MoveConstructible<Types>... tuple(tuple&&);

  template <class... UTypes>
    requires HasConstructor<Types, UTypes>...
    tuple(const tuple<UTypes...>&);
  template <class... UTypes>
    requires HasConstructor<Types, UTypes&&>...
    tuple(tuple<UTypes...>&&);

  template <class U1, class U2class... UTypes>
    requires HasConstructor<Types, UTypes>...
    tuple(const pair<U1, U2UTypes...>&);      // iff sizeof...(Types) == 2
  template <class U1, class U2class... UTypes>
    requires HasConstructor<Types, UTypes&&>...
    tuple(pair<U1, U2UTypes...>&&);           // iff sizeof...(Types) == 2

  // allocator-extended constructors
  template <classAllocator Alloc>
    requires ConstructibleAsElement<Alloc, Types>...
    tuple(allocator_arg_t, const Alloc& a);
  template <classAllocator Alloc>
    requires ConstructibleAsElement<Alloc, Types, const Types&>...
    explicit tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template <classAllocator Alloc, class... UTypes>
    requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
    explicit tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template <classAllocator Alloc>
    requires ConstructibleAsElement<Alloc, Types, const Types&>...
    tuple(allocator_arg_t, const Alloc& a, const tuple&);
  template <classAllocator Alloc>
    requires ConstructibleAsElement<Alloc, Types, Types&&>...
    tuple(allocator_arg_t, const Alloc& a, tuple&&);
  template <classAllocator Alloc, class... UTypes>
    requires ConstructibleAsElement<Alloc, Types, const UTypes&>...
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template <classAllocator Alloc, class... UTypes>
    requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
  template <classAllocator Alloc, class U1, class U2class... UTypes>
    requires ConstructibleAsElement<Alloc, Types, const UTypes&>...
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2UTypes...>&);
  template <classAllocator Alloc, class U1, class U2class... UTypes>
    requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2UTypes...>&&);

  requires CopyAssignable<Types>... tuple& operator=(const tuple&);
```

```
  requires MoveAssignable<Types>... tuple& operator=(tuple&&);

  template <class... UTypes>
    requires HasCopyAssign<Types, UTypes>...
    tuple& operator=(const tuple<UTypes...>&);
  template <class... UTypes>
    requires HasMoveAssign<Types, UTypes>...
    tuple& operator=(tuple<UTypes...>&&);

  template <class U1, class U2class... UTypes>
    requires HasCopyAssign<Types, UTypes>...
    tuple& operator=(const pair<U1, U2UTypes...>&);  // iff sizeof...(Types) == 2
  template <class U1, class U2class... UTypes>
    requires HasMoveAssign<Types, UTypes>...
    tuple& operator=(pair<U1, U2UTypes...>&&);       // iff sizeof...(Types) == 2
};

template<class... Types, class... Alloc>
  concept_map UsesAllocator<tuple<Types...>, Alloc> { }
```

20.3.1.1 ~~Tuple traits~~ [tuple.traits]

Remove the section [tuple.traits] completely

20.3.1.2 Construction [tuple.cstr]

```
requires DefaultConstructible<Types>... tuple();
```

1 ~~*Requires:* Each type in Types shall be default constructible.~~

2 *Effects:* Default initializes each element.

```
requires CopyConstructible<Types>... tuple(const Types&...);
```

3 ~~*Requires:* Each type in Types shall be copy constructible.~~

4 *Effects:* Copy initializes each element with the value of the corresponding parameter.

```
template <class... UTypes>
  requires HasConstructor<Types, UTypes&&>...
  tuple(UTypes&&... u);
```

5 ~~*Requires:* Each type in Types shall be move constructible from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.~~

6 *Effects:* Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

```
requires CopyConstructible<Types>... tuple(const tuple& u);
```

7 ~~*Requires:* Each type in Types shall be copy constructible.~~

8 *Effects:* Copy constructs each element of `*this` with the corresponding element of `u`.

```
requires MoveConstructible<Types>... tuple(tuple&& u);
```

9 ~~*Requires:* Each type in Types shall be move constructible.~~

10 *Effects:* Move-constructs each element of `*this` with the corresponding element of `u`.

```
template <class...  UTypes>
  requires HasConstructor<Types, UTypes>...
  tuple(const tuple<UTypes...>& u);
```

11 ~~*Requires:* Each type in Types shall be constructible from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.~~

12 *Effects:* Constructs each element of `*this` with the corresponding element of `u`.

13 [*Note:* `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — *end note*]

```
template <class... UTypes>
  requires HasConstructor<Types, UTypes&&>...
  tuple(tuple<UTypes...>&& u);
```

14 ~~*Requires:* Each type in Types shall be move-constructible from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.~~

15 *Effects:* Move-constructs each element of `*this` with the corresponding element of `u`.

[*Note:* `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — *end note*]

`template <`~~`class U1, class U2`~~`class... UTypes>`
  `requires HasConstructor<Types, UTypes>...`
  `tuple(const pair<`~~`U1, U2`~~`UTypes...>&);`

16 ~~*Requires:* The first type in Types shall be constructible from U1 and the second type in Types shall be constructible from U2. `sizeof...(Types) == 2`.~~

17 *Effects:* Constructs the first element with `u.first` and the second element with `u.second`.

`template <`~~`class U1, class U2`~~`class... UTypes>`
  `requires HasConstructor<Types, UTypes&&>...`
  `tuple(pair<`~~`U1, U2`~~`UTypes...>&&);`

18 ~~*Requires:* The first type in Types shall be move constructible from U1 and the second type in Types shall be move-constructible from U2. `sizeof...(Types) == 2`.~~

19 *Effects:* Constructs the first element with `std::move(u.first)` and the second element with `std::move(u.second)`.

```
requires CopyAssignable<Types>... tuple& operator=(const tuple& u);
```

20 ~~*Requires:* Each type in Types shall be assignable.~~

21 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

22 *Returns:* `*this`

```
requires MoveAssignable<Types>... tuple& operator=(tuple&& u);
```

23 ~~*Requires:* Each type in Types shall be move-assignable.~~

24 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

25 *Returns:* `*this`.

```
template <class... UTypes>
  requires CopyAssignable<Types, UTypes>...
  tuple& operator=(const tuple<UTypes...>& u);
```

26 ~~*Requires:* Each type in Types shall be assignable from the corresponding type in UTypes~~.

27 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

28 *Returns:* `*this`

```
template <class... UTypes>
  requires MoveAssignable<Types, UTypes>...
  tuple& operator=(tuple<UTypes...>&& u);
```

29 ~~*Requires:* Each type in Types shall be move-assignable from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.~~

30 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

31 *Returns:* `*this`.

```
template <class U1, class U2class... UTypes>
  requires HasCopyAssign<Types, UTypes>...
  tuple& operator=(const pair<U1, U2UTypes...>&);
```

32 ~~*Requires:* The first type in Types shall be move assignable from U1 and the second type in Types shall be move assignable from U2. `sizeof...(Types) == 2`.~~

33 *Effects:* Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

34 *Returns:* `*this`

35 [*Note:* There are rare conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type `T` and if the parameter passed to the constructor is not of type `T`, but rather a tuple type that is convertible to `T`. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it it possible to compare the "nesting depths" of the source and target tuples and decide to select the element-wise constructor if the source nesting depth is smaller than the target nesting-depth. This can be accomplished using an `enable_if` template or other tools for constrained templates. — *end note*]

```
template <class U1, class U2class... UTypes>
  requires HasMoveAssign<Types, UTypes>...
  tuple& operator=(pair<U1, U2UTypes...>&&);
```

36 ~~*Requires:* The first type in Types shall be assignable from U1 and the second type in Types shall be assignable from U2. `sizeof...(Types) == 2`.~~

37 *Effects:* Assigns `std::move(u.first)` to the first element of `*this` and `std::move(u.second)` to the second element of `*this`.

38 *Returns:* `*this`.

```
template <classAllocator Alloc>
  requires ConstructibleAsElement<Alloc, Types>...
  tuple(allocator_arg_t, const Alloc& a);
template <classAllocator Alloc>
  requires ConstructibleAsElement<Alloc, Types, const Types&>...
  explicit tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <classAllocator Alloc, class... UTypes>
  requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
  explicit tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template <classAllocator Alloc>
  requires ConstructibleAsElement<Alloc, Types, const Types&>...
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <classAllocator Alloc>
  requires ConstructibleAsElement<Alloc, Types, Types&&>...
  tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <classAllocator Alloc, class... UTypes>
  requires ConstructibleAsElement<Alloc, Types, const UTypes&>...
  tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <classAllocator Alloc, class... UTypes>
  requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
  tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <classAllocator Alloc, class U1, class U2class... UTypes>
  requires ConstructibleAsElement<Alloc, Types, const UTypes&>...
  tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2UTypes...>&);
template <classAllocator Alloc, class U1, class U2class... UTypes>
  requires ConstructibleAsElement<Alloc, Types, UTypes&&>...
  tuple(allocator_arg_t, const Alloc& a, pair<U1, U2UTypes...>&&);
```

39 ~~*Requires:* `Alloc` shall be an `Allocator` ([allocator.requirements]).~~

40 *Effects:* Equivalent to the preceding constructors except that the allocator argument is passed conditionally to the constructor of each element. Each member is *allocator constructed* ([allocator.concepts]) with a.

20.3.1.3 Tuple creation functions [tuple.creation]

```
template<classMoveConstructible... Types>
  tuple<VTypes...> make_tuple(Types&&... t);
```

1 Let `Ui` be `decay<Ti>::type` for each `Ti` in `Types`. Then each `Vi` in `VTypes` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

2 *Returns:* `tuple<VTypes...>(std::forward<Types>(t)...)`.

3 [*Example:*

```
int i; float j;
```

```
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

— end example]

```
template<classVariableType... Types>
  tuple<Types&...> tie(Types&... t);
```

4 *Returns:* `tuple<Types&>(t...)`. When an argument in `t` is `ignore`, assigning any value to the corresponding tuple element has no effect.

5 [*Example:* `tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

— end example]

I have collapsed the 8 paragraphs used to describe the four different variants of `tuple_cat` into a single paragraph of description, which eliminates a lot of redundancy and saves some space.

```
template <classCopyConstructible... TTypes, classCopyConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
template <classMoveConstructible... TTypes, classCopyConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, const tuple<UTypes...>& u);
template <classCopyConstructible... TTypes, classMoveConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, tuple<UTypes...>&& u);
template <classMoveConstructible... TTypes, classMoveConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, tuple<UTypes...>&& u);
```

6 ~~*Requires:* All the types in TTypes shall be MoveConstructible (Table [moveconstructible]). All the types in UTypes shall be MoveConstructible (Table [moveconstructible]).~~

7 *Returns:* A `tuple` object constructed by ~~move~~copy- or move-constructing its first `sizeof...(TTypes)` elements from the corresponding elements of `t` and ~~move~~copy- or move-constructing its last `sizeof...(UTypes)` elements from the corresponding elements of `u`.

20.3.1.4 Tuple helper classes [tuple.helper]

```
template <int I, class... Types>
requires True<0 <= I && I < sizeof...(Types)>
class tuple_element<I, tuple<Types...> > {
public:
  typedef TI type;
};
```

4 ~~*Requires:* `0 <= I` and `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

5 *Type:* `TI` is the type of the `I`th element of `Types`, where indexing is zero-based.

20.3.1.5 Element access [tuple.elem]

```
template <int I, classVariableType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);
```

1 ~~*Requires:* `0 <= I` and `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

2

3 *Returns:* A reference to the `I`th element of `t`, where indexing is zero-based.

```
template <int I, classVariableType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);
```

4 ~~*Requires:* `0 <= I` and `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

5

6 *Returns:* A const reference to the `I`th element of `t`, where indexing is zero-based.

7 [*Note:* Constness is shallow. If a `T` in `Types` is some reference type `X&`, the return type is `X&`, not `const X&`. However, if the element type is non-reference type `T`, the return type is `const T&`. This is consistent with how constness is defined to work for member variables of reference type. — *end note*]]

8 [*Note:* The reason `get` is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the `template` keyword. — *end note*]

20.3.1.6 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 ~~*Requires:* For all `i`, where `0 <= i` and `i < sizeof...(Types)`, `get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.~~

2 *Returns:* `true` iff `get<i>(t) == get<i>(u)` for all `i`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

3 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

~~*Requires:* For all `i`, where `0 <= i` and `i < sizeof...(Types)`, `get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.~~

4 *Returns:* The result of a lexicographical comparison between `t` and `u`. The result is defined as: `(bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && ttail < utail)`, where r_{tail} for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e < f` returns `false`.

```
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

5 *Returns:* `!(t == u)`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

6 *Returns:* `u < t`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

7 *Returns:* `!(u < t)`

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

8 *Returns:* `!(t < u)`

9 [*Note:* The above definitions for comparison operators do not require t_{tail} (or u_{tail}) to be constructed. It may not even be possible, as `t` and `u` are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — *end note*]

20.5 Function objects [function.objects]

1 Function objects are objects with an `operator()` defined. In the places where one would expect to pass a pointer to a function to an algorithmic template (clause [algorithms]), the interface is specified to accept an object with an `operator()` defined. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

2 **Header `<functional>` synopsis**

```
namespace std {
  // [base], base:
  template <class Arg, class Result> struct unary_function;
  template <class Arg1, class Arg2, class Result> struct binary_function;

  // [func.ret] result_of:
  template <class> class result_of;    // undefined
  template <class F, class... Args> class result_of<F(ArgTypes...)>;
```

```
// 20.5.5, reference_wrapper:
template <classObjectType T> class reference_wrapper;

template <classObjectType T> reference_wrapper<T> ref(T&);
template <classObjectType T> reference_wrapper<const T> cref(const T&);

template <classObjectType T> reference_wrapper<T> ref(reference_wrapper<T>);
template <classObjectType T> reference_wrapper<const T> cref(reference_wrapper<T>);

// 20.5.6, arithmetic operations:
template <class T> struct plus;
template <class T> struct minus;
template <class T> struct multiplies;
template <class T> struct divides;
template <class T> struct modulus;
template <class T> struct negate;

// 20.5.7, comparisons:
template <class T> struct equal_to;
template <class T> struct not_equal_to;
template <class T> struct greater;
template <class T> struct less;
template <class T> struct greater_equal;
template <class T> struct less_equal;

// 20.5.8, logical operations:
template <class T> struct logical_and;
template <class T> struct logical_or;
template <class T> struct logical_not;

// 20.5.9, bitwise operations:
template <class T> struct bit_and;
template <class T> struct bit_or;
template <class T> struct bit_xor;

// [negators], negators:
template <class Predicate> class unary_negate;
template <class Predicate>
  unary_negate<Predicate>  not1(const Predicate&);
template <class Predicate> class binary_negate;
template <class Predicate>
  binary_negate<Predicate> not2(const Predicate&);

// 20.5.10, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<classCopyConstructible Fn, classCopyConstructible... Types>
  unspecified bind(Fn, Types...);
```

```
template<classReturnable R, classCopyConstructible Fn, classCopyConstructible... Types>
  unspecified bind(Fn, Types...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified _1;
  extern unspecified _2;
              .
              .
              .
  extern unspecified _M;
}

// [depr.lib.binders], binders (deprecated):
template <class Fn> class binder1st;
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn&, const T&);
template <class Fn> class binder2nd;
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn&, const T&);

// 20.5.11, adaptors:
template <classCopyConstructible Arg, classReturnable Result>
  class pointer_to_unary_function;
template <classCopyConstructible Arg, classReturnable Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <classCopyConstructible Arg1, classCopyConstructible Arg2, classReturnable Result>
  class pointer_to_binary_function;
template <classCopyConstructible Arg1, classCopyConstructible Arg2, classReturnable Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// 20.5.12, adaptors:
template<classReturnable S, classClassType T> class mem_fun_t;
template<classReturnable S, classClassType T, classMoveConstructible A> class mem_fun1_t;
template<classReturnable S, classClassType T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template<classReturnable S, classClassType T, classMoveConstructible A>
    mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<classReturnable S, classClassType T> class mem_fun_ref_t;
template<classReturnable S, classClassType T, classCopyConstructible A> class mem_fun1_ref_t;
template<classReturnable S, classClassType T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<classReturnable S, classClassType T, classCopyConstructible A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));

template <classReturnable S, classClassType T> class const_mem_fun_t;
template <classReturnable S, classClassType T, classCopyConstructible A> class const_mem_fun1_t;
template <classReturnable S, classClassType T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
```

```
template <classReturnable S, classClassType T, classCopyConstructible A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <classReturnable S, classClassType T> class const_mem_fun_ref_t;
template <classReturnable S, classClassType T, classCopyConstructible A> class const_mem_fun1_ref_t;
template <classReturnable S, classClassType T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <classReturnable S, classClassType T, classCopyConstructible A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);

// 20.5.13, member function adaptors:
template<classReturnable R, class T> unspecified mem_fn(R T::*);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::*)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* const)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* volatile)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* const volatile)(Args...) pm);

// 20.5.14 polymorphic function wrappers:
class bad_function_call;

template<class> class function; // undefined
template<classReturnable R, classCopyConstructible...  ArgTypes>
  class function<R(ArgTypes...)>;

template<classReturnable R, classCopyConstructible...  ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

template<classReturnable R, classCopyConstructible...  ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
template<classReturnable R, classCopyConstructible...  ArgTypes>
  bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);
template<classReturnable R, classCopyConstructible...  ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
template<classReturnable R, classCopyConstructible...  ArgTypes>
  bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

// [unord.hash], hash function base template:
template <class T> struct hash;

// Hash function specializations
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
```

```
  template <> struct hash<short>;
  template <> struct hash<unsigned short>;
  template <> struct hash<int>;
  template <> struct hash<unsigned int>;
  template <> struct hash<long>;
  template <> struct hash<long long>;
  template <> struct hash<unsigned long>;
  template <> struct hash<unsigned long long>;

  template <> struct hash<float>;
  template <> struct hash<double>;
  template <> struct hash<long double>;

  template<class T> struct hash<T*>;

  template <> struct hash<std::string>;
  template <> struct hash<std::u16string>;
  template <> struct hash<std::u32string>;
  template <> struct hash<std::wstring>;
}
```

20.5.5 Class template `reference_wrapper` [refwrap]

```
template <classObjectType T> class reference_wrapper
  : public unary_function<T1, R>          // see below
  : public binary_function<T1, T2, R>     // see below
{
public :
  // types
  typedef T type;
  typedef see below result_type; // Not always defined

  // construct/copy/destroy
  explicit reference_wrapper(T&);
  reference_wrapper(const reference_wrapper<T>& x);

  // assignment
  reference_wrapper& operator=(const reference_wrapper<T>& x);

  // access
  operator T& () const;
  T& get() const;

  // tcode
  template <class... ArgTypes>
    requires Callable<T, ArgTypes&&...>
    typename result_of<T(ArgTypes...)>::typeCallable<T, ArgTypes&&...>::result_type
  operator() (ArgTypes&&...) const;
};
```

20.5.5.4 `reference_wrapper` invocation [refwrap.invoke]

```
template <class... ArgTypes>
  requires Callable<T, ArgTypes&&...>
  typename result_of<T(ArgTypes...)>::typeCallable<T, ArgTypes&&...>::result_type
    operator()(ArgTypes&&... args) const;
```

1 *Returns:* `INVOKE(get(), std::forward<ArgTypes>(args)...)`. ([func.require])

2 ~~*Remark:* `operator()` is described for exposition only. Implementations are not required to provide an actual `reference_wrapper::operator()`. Implementations are permitted to support `reference_wrapper` function invocation through multiple overloaded operators or through other means.~~

20.5.5.5 reference_wrapper helper functions [refwrap.helpers]

```
template <classObjectType T> reference_wrapper<T> ref(T& t);
```

1 *Returns:* `reference_wrapper<T>(t)`

2 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

3 *Returns:* `ref(t.get())`

4 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<const T> cref(const T& t);
```

5 *Returns:* `reference_wrapper <const T>(t)`

6 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<const T> cref(reference_wrapper<T> t);
```

7 *Returns:* `cref(t.get());`

8 *Throws:* nothing.

20.5.6 Arithmetic operations [arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language ([expr.mul], [expr.add]).

```
template <class T> struct plus : binary_function<T,T,T> {
  requires HasPlus<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns `x + y`.

```
template <class T> struct minus : binary_function<T,T,T> {
  requires HasMinus<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
```

```
};
```

3 `operator()` returns x `-` y.

```
template <class T> struct multiplies : binary_function<T,T,T> {
  requires HasMultiply<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns x `*` y.

```
template <class T> struct divides : binary_function<T,T,T> {
  requires HasDivide<T> && Convertible<T::result_type, T>
  T operator()(const T& x, const T& y) const;
};
```

5 `operator()` returns x `/` y.

```
template <class T> struct modulus : binary_function<T,T,T> {
  requires HasModulus<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

6 `operator()` returns x `%` y.

```
template <class T> struct negate : unary_function<T,T> {
  requires HasNegate<T> && Convertible<T::result_type, T>
    T operator()(const T& x) const;
};
```

7 `operator()` returns `-`x.

20.5.7 Comparisons [comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language ([expr.rel], [expr.eq]).

```
template <class T> struct equal_to : binary_function<T,T,bool> {
  requires EqualityComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns x `==` y.

```
template <class T> struct not_equal_to : binary_function<T,T,bool> {
  requires EqualityComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns x `!=` y.

```
template <class T> struct greater : binary_function<T,T,bool> {
  requires LessThanComparable<T>
```

```
    bool operator()(const T& x, const T& y) const;
};
```

4 operator() returns x > y.

```
template <class T> struct less : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

5 operator() returns x < y.

```
template <class T> struct greater_equal : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

6 operator() returns x >= y.

```
template <class T> struct less_equal : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

7 operator() returns x <= y.

8 For templates greater, less, greater_equal, and less_equal, the specializations for any pointer type yield a total order, even if the built-in operators <, >, <=, >= do not.

20.5.8 Logical operations [logical.operations]

1 The library provides basic function object classes for all of the logical operators in the language ([expr.log.and], [expr.log.or], [expr.unary.op]).

```
template <class T> struct logical_and : binary_function<T,T,bool> {
  requires HasLogicalAnd<T>
    bool operator()(const T& x, const T& y) const;
};
```

2 operator() returns x && y.

```
template <class T> struct logical_or : binary_function<T,T,bool> {
  requires HasLogicalOr<T>
    bool operator()(const T& x, const T& y) const;
};
```

3 operator() returns x || y.

```
template <class T> struct logical_not : unary_function<T,bool> {
  requires HasLogicalNot<T>
    bool operator()(const T& x) const;
};
```

4 `operator()` returns `!`*x*.

20.5.9 Bitwise operations [bitwise.operations]

1 The library provides basic function object classes for all of the bitwise operators in the language ([expr.bit.and], [expr.or], [expr.xor]).

```
template <class T> struct bit_and : binary_function<T,T,T> {
  requires HasBitAnd<T>
    T operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns *x* `&` *y*.

```
template <class T> struct bit_or : binary_function<T,T,T> {
  requires HasBitOr<T>
    T operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns *x* `|` *y*.

```
template <class T> struct bit_xor : binary_function<T,T,T> {
  requires HasBitXor<T>
    T operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns *x* `^` *y*.

20.5.10 Template function `bind` [bind]

1 The template function `bind` returns an object that binds a function object passed as an argument to additional arguments.

2 ~~Binders `bind1st` and `bind2nd` take a function object `fn` of two arguments and a value `x` and return a function object of one argument constructed out of `fn` with the first or second argument correspondingly bound to `x`.~~

20.5.10.1 Function object binders [func.bind]

20.5.10.1.3 Function template `bind` [func.bind.bind]

template<~~class~~CopyConstructible F, ~~class~~CopyConstructible... BoundArgs>
  unspecified bind(F f, BoundArgs... bound_args);

1 *Requires:* ~~F and each Ti in BoundArgs shall be CopyConstructible.~~ *INVOKE* `(f, w1, w2, ..., wN)` ([func.require]) shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(bound_args)`.

2 *Returns:* A forwarding call wrapper `g` with a weak result type ([func.require]). The effect of `g(u1, u2, ..., uM)` shall be *INVOKE*`(f, v1, v2, ..., vN,` ~~`result_of`~~`Callable<F` *cv* ~~`(`~~`,V1, V2, ..., VN`~~`)`~~`>::result_type)`, where *cv* represents the *cv*-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below.

```
template<classReturnable R, classCopyConstructible F, classCopyConstructible... BoundArgs>
  unspecified bind(F f, BoundArgs... bound_args);
```

3 *Requires:* ~~F and each Ti in BoundArgs shall be CopyConstructible.~~ *INVOKE* (f, w1, w2, ..., wN) shall be a valid expression for some values *w1, w2, ..., wN*, where N == sizeof...(bound_args).

4 *Returns:* A forwarding call wrapper g with a nested type result_type defined as a synonym for R. The effect of g(u1, u2, ..., uM) shall be *INVOKE* (f, v1, v2, ..., vN, R), where the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

20.5.11 Adaptors for pointers to functions [function.pointer.adaptors]

1 To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <classCopyConstructible Arg, classReturnable Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
public:
  explicit pointer_to_unary_function(Result (*f)(Arg));
  Result operator()(Arg x) const;
};
```

2 operator() returns *f* (*x*).

```
template <classCopyConstructible Arg, classReturnable Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
```

3 *Returns:* pointer_to_unary_function<Arg, Result>(*f*).

```
template <classCopyConstructible Arg1, classCopyConstructible Arg2, classReturnable Result>
class pointer_to_binary_function :
  public binary_function<Arg1,Arg2,Result> {
public:
  explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
  Result operator()(Arg1 x, Arg2 y) const;
};
```

4 operator() returns *f* (*x*,*y*).

```
template <classCopyConstructible Arg1, classCopyConstructible Arg2, classReturnable Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*f)(Arg1, Arg2));
```

5 *Returns:* pointer_to_binary_function<Arg1,Arg2,Result>(*f*).

6 [*Example:*

```
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
    not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each abc with def in sequence v. — *end example*]

20.5.12 Adaptors for pointers to members [member.pointer.adaptors]

1 The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in 20.5.11.

```
template <classReturnable S, classClassType T> class mem_fun_t
        : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
```

2 `mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <classReturnable S, classClassType T, classCopyConstructible A> class mem_fun1_t
      : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

3 `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<classReturnable S, classClassType T> mem_fun_t<S,T>
   mem_fun(S (T::*f)());
template<classReturnable S, classClassType T, classCopyConstructible A> mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A));
```

4 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <classReturnable S, classClassType T> class mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
```

5 `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <classReturnable S, classClassType T, classCopyConstructible A> class mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
```

6 `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<classReturnable S, classClassType T> mem_fun_ref_t<S,T>
```

```
   mem_fun_ref(S (T::*f)());
template<classReturnable S, classClassType T, classCopyConstructible A> mem_fun1_ref_t<S,T,A>
   mem_fun_ref(S (T::*f)(A));
```

7 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

```
template <classReturnable S, classClassType T> class const_mem_fun_t
      : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*p)() const);
  S operator()(const T* p) const;
};
```

8 `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <classReturnable S, classClassType T, classCopyConstructible A> class const_mem_fun1_t
      : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*p)(A) const);
  S operator()(const T* p, A x) const;
};
```

9 `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<cclassReturnable S, classClassType T> const_mem_fun_t<S,T>
   mem_fun(S (T::*f)() const);
template<classReturnable S, classClassType T, classCopyConstructible A> const_mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A) const);
```

10 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <classReturnable S, classClassType T> class const_mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*p)() const);
  S operator()(const T& p) const;
};
```

11 `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <classReturnable S, classClassType T, classCopyConstructible A> class const_mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
  S operator()(const T& p, A x) const;
};
```

12 `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<~~class~~Returnable S, ~~class~~ClassType T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)() const);
template<~~class~~Returnable S, ~~class~~ClassType T, ~~class~~CopyConstructible A> const_mem_fun1_ref_t<S,T,A>
    mem_fun_ref(S (T::*f)(A) const);
```

13 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

20.5.13 Function template `mem_fn` [func.memfn]

```
template<~~class~~Returnable R, class T> unspecified mem_fn(R T::* pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::*)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* const)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* volatile)(Args...) pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified mem_fn(R (T::* const volatile)(Args...) pm);
```

1 *Returns:* A simple call wrapper ([[func.def]]) `fn` such that the expression `fn(t, a2, ..., aN)` is equivalent to *`INVOKE`*`(pm, t, a2, ..., aN)` ([[func.require]]). `fn` shall have a nested type `result_type` that is a synonym for the return type of `pm` when `pm` is a pointer to member function.

2 The simple call wrapper shall be derived from `std::unary_function<cv T*, Ret>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking no arguments, where *Ret* is `pm`'s return type.

3 The simple call wrapper shall be derived from `std::binary_function<cv T*, T1, Ret>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking one argument of type `T1`, where *Ret* is `pm`'s return type.

4 *Throws:* nothing.

5 ~~*Remarks:* Implementations may implement `mem_fn` as a set of overloaded function templates.~~

20.5.14 Polymorphic function wrappers [func.wrap]

20.5.14.2 Class template `function` [func.wrap.func]

```
namespace std {
  template<class> class function; // undefined

  template<~~class~~Returnable R, ~~class~~CopyConstructible...  ArgTypes>
  class function<R(ArgTypes...)>
    : public unary_function<T1, R>      // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
    : public binary_function<T1, T2, R> // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1 and T2
  {
  public:
    typedef R result_type;
```

```
// 20.5.14.2.1, construct/copy/destroy:
explicit function();
function(unspecified-null-pointer-type);
function(const function&);
function(function&&);
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function(F);
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function(F&&);
template<classAllocator A>
  function(allocator_arg_t, const A&);
template<classAllocator A> function(allocator_arg_t, const A&,
  unspecified-null-pointer-type);
template<classAllocator A> function(allocator_arg_t, const A&,
  const function&);
template<classAllocator A> function(allocator_arg_t, const A&,
  function&&);
template<class F, classAllocator A> function(allocator_arg_t, const A&, F);
template<class F, classAllocator A> function(allocator_arg_t, const A&, F&&);

function& operator=(const function&);
function& operator=(function&&);
function& operator=(unspecified-null-pointer-type);
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function& operator=(F);
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(F&&);
template<class F>
  requires Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(reference_wrapper<F>);

~function();

// [func.wrap.func.mod], function modifiers:
void swap(function&);
template<class F, classAllocator A>
  requires Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  void assign(F, const A&);

// [func.wrap.func.cap], function capacity:
```

```
    operator unspecified-bool-type() const;

    // deleted overloads close possible hole in the type system
    template<class R2, class... ArgTypes2>
        bool operator==(const function<R2(ArgTypes2...)>&) = delete;
    template<class R2, class... ArgTypes2>
        bool operator!=(const function<R2(ArgTypes2...)>&) = delete;

    // [func.wrap.func.inv], function invocation:
    R operator()(ArgTypes...) const;

    // 20.5.14.2.5, function target access:
    const std::type_info& target_type() const;
    template <typename T>
      requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
      T* target();
    template <typename T>
      requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
      const T* target() const;

  private:
    // [func.wrap.func.undef], undefined operators:
    template<class R2, class...  ArgTypes2> bool operator==(const function<R2(ArgTypes2...)>&);
    template<class R2, class...  ArgTypes2> bool operator!=(const function<R2(ArgTypes2...)>&);
  };

  // 20.5.14.2.7, Null pointer comparisons:
  template <~~class~~MoveConstructible R, ~~class~~MoveConstructible...  ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);

  template <~~class~~MoveConstructible R, ~~class~~MoveConstructible...  ArgTypes>
    bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

  template <~~class~~MoveConstructible R, ~~class~~MoveConstructible...  ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);

  template <~~class~~MoveConstructible R, ~~class~~MoveConstructible...  ArgTypes>
    bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

  // 20.5.14.2.8, specialized algorithms:
  template <~~class~~MoveConstructible R, ~~class~~MoveConstructible...  ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);
} // namespace std
```

20.5.14.2.1 `function` construct/copy/destroy [func.wrap.func.con]

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function(F f);
```

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function(F&& f);
```

8 ~~*Requires:* `f` shall be callable for argument types `ArgTypes` and return type `R`.~~

9 *Postconditions:* `!*this` if any of the following hold:

- `f` is a NULL function pointer.
- `f` is a NULL member function pointer.
- `F` is an instance of the `function` class template, and `!f`

10 Otherwise, `*this` targets a copy of `f` or `std::move(f)` if `f` is not a pointer to member function, and targets a copy of `mem_fn(f)` if `f` is a pointer to member function.

11 *Throws:* shall not throw exceptions when `f` is a function pointer or a `reference_wrapper<T>` for some `T`. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy or move constructor.

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  operator=(F f);
```

19 *Effects:* `function(f).swap(*this);`

20 *Returns:* `*this`

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function& operator=(F&& f);
```

21 *Effects:* Replaces the target of `*this` with `f`, leaving `f` in a valid but unspecified state. [*Note:* A valid implementation is `function(f).swap(*this)`.

22 *Returns:* `*this`.

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(reference_wrapper<F> f);
```

23 *Effects:* `function(f).swap(*this);`

24 *Returns:* `*this`

25 *Throws:* nothing.

20.5.14.2.5 function target access [func.wrap.func.targ]

```
const std::type_info& target_type() const;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

2 *Throws:* nothing.

```
template<typename T>
  requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
  T* target();
template<typename T>
  requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
  const T* target() const;
```

3 ~~*Requires:* `T` is a function object type that is Callable (20.5.14.2) for parameter types `ArgTypes` and return type `R`.~~

4 *Returns:* If `type() == typeid(T)`, a pointer to the stored function target; otherwise a null pointer.

5 *Throws:* nothing.

20.5.14.2.7 null pointer comparison operators [func.wrap.func.nullptr]

```
template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);
```

1 *Returns:* `!f`.

2 *Throws:* nothing.

```
template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);
```

3 *Returns:* `(bool) f`.

4 *Throws:* nothing.

20.5.14.2.8 specialized algorithms [func.wrap.func.alg]

```
template<classReturnable R, classCopyConstructible... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

1 *Effects:* `f1.swap(f2);`

Bibliography

[1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.