Variadic Templates for the C++0x Standard Library

Authors: Douglas Gregor, Indiana University
Jaakko Järvi, Texas A&M University
Document number: N2151=07-0011
Date: 2007-01-08
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document describes specific changes to the C++0x working paper that make full use of variadic templates [1, 2] within the Standard Library. This document does *not* change the semantics of any library facility in any detectable way; rather, we are only changing the specification to remove pseudo-code descriptions, remove fixed upper limits on the number of parameters these libraries support, and clarify the intent of these libraries. All changes in this document have been implemented in the GNU implementation of the C++ Standard Library and TR1, libstdc++, using the implementation of variadic templates in the GNU compiler, and the resulting library passes all of its existing regression tests.

We took a few small liberties to clean up the presentation along the way. The main improvement was using DR 106 (collapsing references-to-references) to eliminate excess wording. It's much easier to write "`const T&`" rather than "`T` if `T` is a reference, otherwise `const T&`." Again, these changes affect only the presentation, not the semantics. Only the changed parts of the C++0x library are shown in this document; if a section or paragraph is missing, then it is unchanged.

One change not reflected in the actual text (because we could not find it in the current Working Paper, N2134) is the elimination of the two implementation quantities that place lower bounds on the maximum number of arguments that can be forwarded by call wrappers and the maximum number of elements in one tuple type. These two implementation quantities shall be removed.

Chapter 20 General utilities library [utilities]

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. ~~An implementation can set an upper limit for the number of arguments. The minimum value for this implementation quantity is defined in Annex ??.~~ 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 <class...  Types> class tuple;

  // 20.3.1.2, tuple creation functions:
  const unspecified ignore;

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

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

  // 20.3.1.3, tuple helper classes:
  template <class T> class tuple_size; // undefined
  template <class...  Types> class tuple_size<tuple<Types...> >;

  template <int I, class T> class tuple_element; // undefined
  template <int I, class...  Types> class tuple_element<I, tuple<Types...> >;

  // 20.3.1.4, element access:
  template <int I, class...  Types>
    typename tuple_element<I, tuple<Types...> >::type & get(tuple<Types...>&);

  template <int I, class...  Types>
    typename tuple_element<I, tuple<Types...> >::type const & get(const tuple<Types...>&);

  // 20.3.1.5, relational operators:
  template<class...  TTypes, class...  UTypes>
    bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
    template<class...  TTypes, class...  UTypes>
      bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);

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

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

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

    template<class...  TTypes, class...  UTypes>
      bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);
  } // namespace std
```

20.3.1 Class template `tuple` [tuple.tuple]

1 ~~M denotes the implementation-defined number of template type parameters to the tuple class template, and N denotes the number of template arguments specified in an instantiation.~~

6 ~~[*Example:* Given the instantiation `tuple<int, float, char>`, N is 3. — end *example*]~~

```
  template <class...  Types>
  class tuple
  {
  public:
    tuple();
    explicit tuple(const Types&...);

    tuple(const tuple&);
    template <class...  UTypes>
      tuple(const tuple<UTypes...>&);
    template <class U1, class U2>
      tuple(const pair<U1, U2>&);              // iff ~~N~~sizeof...(Types) == 2

    tuple& operator=(const tuple&);
    template <class...  UTypes>
      tuple& operator=(const tuple<UTypes...>&);
    template <class U1, class U2>
      tuple& operator=(const pair<U1, U2>&); // iff ~~N~~sizeof...(Types) == 2
  };
```

20.3.1.1 Construction [tuple.cnstr]

`tuple();`

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

2 *Effects:* Default initializes each element.

`tuple(const Types&...);`

7 ~~where Pi is Ti if Ti is a reference type, or const Ti& otherwise.~~

8 *Requires:* Each type ~~`Ti`~~`in Types` shall be copy constructible.

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

`tuple(const tuple& u);`

10 *Requires:* Each type ~~`Ti`~~`in Types` shall be copy constructible.

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

`template <class... UTypes> tuple(const tuple<UTypes...>& u);`

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

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

14 [*Note:* ~~In an implementation where one template definition serves for many different values for N,~~ `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. ~~Another way of achieving this is adding an extra integral template parameter which defaults to N (more precisely, a metafunction that computes N), and then defining the converting copy constructor and assignment only for tuples where the extra parameter in the source is N~~. — *end note*]

`template <class U1, class U2> tuple(const pair<U1, U2>& u);`

15 *Requires:* ~~`T1`~~The first type in `Types` shall be constructible from `U1`, ~~`T2`~~the second type in `Types` shall be constructible from `U2`. ~~`N == 2`~~`sizeof...(Types) == 2`.

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

`tuple& operator=(const tuple& u);`

17 *Requires:* Each type ~~`Ti`~~`in Types` shall be assignable.

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

19 *Returns:* `*this`

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

20 *Requires:* Each type ~~`Ti`~~`in Types` shall be assignable from the corresponding type ~~`Ui`~~`in UTypes`. `sizeof...(Types) == sizeof...(UTypes)`

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

22 *Returns:* `*this`

`template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);`

23 *Requires:* ~~`T1`~~The first type in `Types` shall be assignable from `U1`, ~~`T2`~~the second type in `Types` shall be assignable from `U2`. ~~`N == 2`~~`sizeof...(Types) == 2`.

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

25 *Returns:* `*this`

26 [*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*]

20.3.1.2 Tuple creation functions [tuple.creation]

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

1 where each `Vi` in `VTypes` is `X&` if the corresponding cv-unqualified type `Ti` in `Types` is `reference_wrapper<X>`, otherwise `Vi` is `Ti`.

2 ~~The make_tuple function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.~~

3 *Returns:* `tuple<VTypes...>(t...)`.

4 [*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<class...  Types>
  tuple<Types&...> tie(Types&...  t);
```

5 ~~The tie function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.~~

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

7 [*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*]

20.3.1.3 Tuple helper classes [tuple.helper]

```
template <class... Types>
class tuple_size<tuple<Types...> >
  : public integral_constant<size_t, sizeof...(Types)> { };
```

5 ~~Requires: T is an instantiation of class template tuple.~~

10 ~~Type: integral constant expression.~~

15 ~~Value: Number of elements in T.~~

```
template<int I, class... Types>
class tuple_element<I, tuple<Types...> >
{
public:
  typedef TI type;
};
```

16 *Requires:* $0 \leq$ `I` $<$~~N~~`sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.

17 ~~Value: The~~`TI` is the type of the `I`th element of ~~T~~`Types`, where indexing is zero-based.

20.3.1.4 Element access [tuple.elem]

```
template <int I, class...  Types>
  typename tuple_element<I, tuple<Types...> >::type & get(tuple<Types...>& t);
```

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

6 ~~Return type: RJ, where J=I+1. If TJ is a reference type, then RJ is TJ, otherwise RJ is TJ&.~~

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

```
template <int I, class...  Types>
  typename tuple_element<I, tuple<Types...> >::type const & get(const tuple<Types...>& t);
```

8 *Requires:* $0 \leq$ `I` $<$~~N~~`sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.

13 ~~Return type: PJ, where J=I+1. If TJ is a reference type, then PJ is TJ, otherwise PJ is const TJ&~~.

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

15 [*Note:* Constness is shallow. If ~~TJ~~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*.]

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

20.3.1.5 Relational operators [tuple.rel]

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

1 *Requires:* For all `i`, where ~~0 <= i < N~~0 <= i < sizeof...(TTypes), `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>
  bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

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

5 *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)) &&` t_{tail} `<` u_{tail}`)`, 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>
  bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

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

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

7 *Returns:* `u < t`.

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

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

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

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

10 [*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 **??**), 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 {
  // 20.5.4 result_of:
  template <class ~~FunctionCallType~~> class result_of; // undefined
  template <class F, class...  ArgTypes> class result_of<F(ArgTypes...)>;

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

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

  template<class Fn, class...  Types>
    unspecified bind(Fn, Types...);
  template<class R, class Fn, class...  Types>
    unspecified bind(Fn, Types...);

  // ??, member function adaptors:
  template<class R, class T> unspecified mem_fn(R T::*);

  // 20.5.14 polymorphic function wrappers:
  class bad_function_call;

  template<class ~~Function~~> class function; // undefined
  template<class R, class...  ArgTypes> class function<R(ArgTypes...)>;

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

  template<class R1, class R2, class...  ArgTypes1, class...  ArgTypes2>
    void operator==(const function<R1(ArgTypes1...)>&, const function<R2(ArgTypes2...)>&);
  template<class R1, class R2, class...  ArgTypes1, class...  ArgTypes2>
    void operator!=(const function<R1(ArgTypes1...)>&, const function<R2(ArgTypes2...)>&);

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

3 [*Example:* If a C++ program wants to have a by-element addition of two vectors `a` and `b` containing `double` and put the

result into a, it can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

—*end example*]

4 [*Example:* To negate every element of a:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

—*end example*]

5 To enable adaptors and other components to manipulate function objects that take one or two arguments it is required that the function objects correspondingly provide typedefs `argument_type` and `result_type` for function objects that take one argument and `first_argument_type`, `second_argument_type`, and `result_type` for function objects that take two arguments.

20.5.2 Requirements [func.require]

1 Define *INVOKE*`(f, t1, t2, ..., tN)` as follows:

— `(t1.*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is not one of the types described in the previous item;

— `t1.*f` when `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `(*t1).*f` when `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item;

— `f(t1, t2, ..., tN)` in all other cases.

2 Define *INVOKE*`(f, t1, t2, ..., tN, R)` as *INVOKE*`(f, t1, t2, ..., tN)` implicitly converted to `R`.

3 If a call wrapper ([**??**]) has a *weak result type* the type of its member type `result_type` is based on the type `T` of the wrapper's target object ([**??**]):

— if `T` is a function, reference to function, or pointer to function type, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a pointer to member function, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a class type with a member type `result_type`, then `result_type` shall be a synonym for `T::result_-type`;

— otherwise `result_type` shall not be defined.

4 Every call wrapper [**??**] shall be CopyConstructible. A *simple call wrapper* is a call wrapper that is Assignable and whose copy constructor and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called with an argument list ~~`t1, t2, ..., tN` where each `ti` is an lvalue~~containing lvalues. The effect of calling a forwarding call wrapper with one or more arguments that are rvalues is implementation defined. [*Note:* in

a typical implementation forwarding call wrappers have ~~overloaded function call operators~~an overloaded function call operator of the form

```
template<~~class T1, class T2, ..., class TN~~class... ArgTypes>
R  operator()(~~T1& t1, T2& t2, ..., TN& tN~~ArgTypes&... args) cv-qual;
```

—end note]

20.5.4 Function object return types [func.ret]

```
namespace std {
  template <class ~~FunctionCallTypes~~> class result_of; // undefined

  template <class Fn, class.. ArgTypes>
  class result_of<Fn(ArgTypes...)> {
  public :
    // types
    typedef see below type;
  };
} // namespace std
```

1 Given an rvalue `fn` of type `Fn` and values `t1, t2, ..., tN` of types `T1, T2, ..., TN` in `ArgTypes`, respectively, the `type` member is the result type of the expression `f(t1, t2, ...,tN)`. The values `ti` are lvalues when the corresponding type `Ti` is a reference type, and rvalues otherwise.

2 The implementation may determine the `type` member via any means that produces the exact type of the expression `f(t1, t2, ..., tN)` for the given types. [*Note:* The intent is that implementations are permitted to use special compiler hooks *—end note*]

3 If `Fn` is not a function object defined by the standard library, and if either the implementation cannot determine the type of the expression `fn(t1, t2, ..., tN)` or the expression is ill-formed, the implementation shall use the following process to determine the `type` member:

1. If `Fn` is a function pointer or function reference type, `type` shall be the return type of the function type.
2. If `Fn` is a member function pointer type, `type` shall be the return type of the member function type.
3. If `Fn` is a member data pointer type `R T::*`, `type` shall be *cv* `R&` when `T1` is *cv* `U1&`, `R` otherwise.
4. If `Fn` is a possibly *cv*-qualified class type with a member type `result_type`, `type` shall be `typename F::result_type`.
5. If `Fn` is a possibly *cv*-qualified class type with no member named `result_type` or if `typename Fn::result_type` is not a type:
   (a) If `N=0` (no arguments), `type` shall be `void`.
   (b) If `N>0`, `type` shall be `typename Fn::template result<Fn(~~T1, T2,..., TN~~ArgTypes...)>::type`.
6. Otherwise, the program is ill-formed.

20.5.5 Class template `reference_wrapper` [refwrap]

```
template <class 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;

  // invocation
  template <~~class T1, class T2, ..., class TN~~class... ArgTypes>
  typename result_of<T(~~T1, T2, ..., TN~~ArgTypes...)>::type
  operator() (~~T1&, T2&, ..., TN&~~ArgTypes&...) const;
};
```

1 `reference_wrapper<T>` is a CopyConstructible and Assignable wrapper around a reference to an object of type `T`.

2 `reference_wrapper` has a weak result type ([20.5.2]).

3 The template instantiation `reference_wrapper<T>` shall be derived from `std::unary_function<T1, R>` only if the type `T` is any of the following:

- a function type or a pointer to function type taking one argument of type `T1` and returning `R`
- a pointer to member function `R T0::f` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`
- a class type that is derived from `std::unary_function<T1, R>`

4 The template instantiation `reference_wrapper<T>` shall be derived from `std::binary_function<T1, T2, R>` only if the type `T` is any of the following:

- a function type or a pointer to function type taking two arguments of types `T1` and `T2` and returning `R`
- a pointer to member function `R T0::f(T2)` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`
- a class type that is derived from `std::binary_function<T1, T2, R>`

20.5.5.4 reference_wrapper invocation [refwrap.invoke]

```
template <~~class T1, class T2, ..., class TN~~class... ArgTypes>
  typename result_of<T(~~T1, T2, ..., TN~~ArgTypes...)>::type
  operator() (~~T1& a1, T2& a2, ..., TN& aN~~ArgTypes&... args) const;
```

1 *Returns:* `INVOKE(get(), ~~a1, a2, ..., aN~~args...)`. ([20.5.2])

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.

20.5.10 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]

1 This subclause describes a uniform mechanism for binding arguments of function objects.

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

```
template<class F, ~~class T1, class T2, ...., class TN~~class... BoundArgs>
  unspecified bind(F f, ~~T1 t1, T2 t2, ..., TN tN~~BoundArgs... bound_args);
```

1 *Requires:* `F` and each `Ti` in BoundArgs shall be CopyConstructible. `INVOKE (f, w1, w2, ..., wN)` ([20.5.2]) 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 ([20.5.2]). The effect of `g(u1, u2, ..., uM)` shall be `INVOKE(f, v1, v2, ..., vN, result_of<F cv (V1, V2, ..., VN)>::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<class R, class F, ~~class T1, class T2, ...., class TN~~class... BoundArgs>
  unspecified bind(F f, ~~T1 t1, T2 t2, ..., TN tN~~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.

5 The values of the *bound arguments* `v1, v2, ..., vN` and their corresponding types `V1, V2, ..., VN` depend on the type of the corresponding argument `ti` in `bound_args` of type `Ti` in BoundArgs in the call to `bind` and the *cv*-qualifiers *cv* of the call wrapper `g` as follows:

— if `ti` is of type `reference_wrapper<T>` the argument is `ti.get()` and its type `Vi` is `T&`;

— if the value of `std::`~~`tr1::`~~`is_bind_expression<Ti>::value` is `true` the argument is `ti(u1, u2, ..., uM)` and its type `Vi` is `result_of<Ti` *cv* `(U1&, U2&, ..., UM&)>::type`;

— if the value `j` of `std::`~~`tr1::`~~`is_placeholder<Ti>::value` is not zero the argument is `uj` and its type `Vi` is `Uj&`;

— otherwise the value is `ti` and its type `Vi` is `Ti` *cv* `&`.

20.5.14 Polymorphic function wrappers [func.wrap]

1 This subclause describes a polymorphic wrapper class that encapsulates arbitrary function objects.

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

```
namespace std {
  template<class ~~Function~~> class function;

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

    // 20.5.14.2.1, construct/copy/destroy:
    explicit function();
    function(unspecified-null-pointer-type);
    function(const function&);
    template<class F> function(F);

    function& operator=(const function&);
    function& operator=(unspecified-null-pointer-type);
    template<class F> function& operator=(F);
    template<class F> function& operator=(reference_wrapper<F>);

    ~function();

    // ??, function modifiers:
    void swap(function&);

    // ??, function capacity:
    operator unspecified-bool-type() const;

    // 20.5.14.2.4, function invocation:
    R operator()(~~T1, T2, ..., TN~~ArgTypes...) const;

    // 20.5.14.2.5, function target access:
    const std::type_info& target_type() const;
    template <typename T>       T* target();
    template <typename T> const T* target() const;

  private:
    // 20.5.14.2.6, 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 Function~~class R, class... ArgTypes>
    bool operator==(const function<~~Function~~R(ArgTypes...)>&, unspecified-null-pointer-type);

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

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

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

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

1 The `function` class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects ([**??**]), given a call signature ([**??**]), allowing functions to be first-class objects.

2 A function object `f` of type `F` is Callable for argument types `T1, T2, ..., TN` in `ArgTypes` and a return type `R`, if, given lvalues `t1, t2, ..., tN` of types `T1, T2, ..., TN`, respectively, *INVOKE*`(f, t1, t2, ..., tN)` is well-formed ([20.5.2]) and, if `R` is not `void`, convertible to `R`.

3 The `function` class template is a call wrapper ([**??**]) whose call signature ([**??**]) is `R(~~T1, T2, ..., TN~~ArgTypes...)`.

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

```
explicit function();
```

1 *Postconditions:* `!*this`.

2 *Throws:* nothing.

```
function(unspecified-null-pointer-type);
```

3 *Postconditions:* `!*this`.

4 *Throws:* nothing.

```
function(const function& f);
```

5 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

6 *Throws:* shall not throw exceptions if `f`'s target is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object.

`template<class F> function(F f);`

7 *Requires:* `f` shall be callable for argument types ~~`T1, T2, ..., TN`~~`ArgTypes` and return type `R`.

8 *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`

9 Otherwise, `*this` targets a copy of `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.

10 *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 constructor.

20.5.14.2.4 `function` invocation [func.wrap.func.inv]

`R operator()(`~~`T1 t1, T2 t2, ..., TN tN`~~`ArgTypes... args) const`

1 *Effects:* *`INVOKE`*`(f, `~~`t1, t2, ..., tN`~~`args..., R)` ([20.5.2]), where `f` is the target object ([**??**]) of `*this`.

2 *Returns:* nothing, if `R` is `void`, otherwise the return value of *`INVOKE`*`(f, `~~`t1, t2, ..., tN`~~`args..., R)`.

3 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped function object.

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>       T* target();
template<typename T> const T* target() const;
```

3 *Requires:* `T` is a function object type that is Callable ([20.5.14.2]) for parameter types ~~`T1, T2, ..., TN`~~`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.

20.5.14.2.6 undefined operators [func.wrap.func.undef]

```
template<class R2, class... ArgTypes2> bool operator==(const function<R2(ArgTypes2...)>&);
template<class R2, class... ArgTypes2> bool operator!=(const function<R2(ArgTypes2...)>&);
```

1 These member functions shall be left undefined.

2 [*Note:* the boolean-like conversion opens a loophole whereby two `function` instances can be compared via `==` or `!=`. These undefined `void` operators close the loophole and ensure a compile-time error. — *end note*]

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

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

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

1 *Returns:* `!f`.

2 *Throws:* nothing.

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

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

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

4 *Throws:* nothing.

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

```
template<~~class Function~~class R, class... ArgTypes>
  void swap(function<~~Function~~R(ArgTypes...)>& f1, function<~~Function~~R(ArgTypes...)>& f2);
```

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

Bibliography

[1] Douglas Gregor, Jaakko Järvi, Jens Maurer, and Jason Merrill. Proposed wording for variadic tempplates. Technical Report N2152=07-0012, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2007.

[2] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates (revision 3). Number N2080=06-0150 in ANSI/ISO C++ Standard Committee Pre-Portland mailing, October 2006.