Boost.Function

Douglas Gregor <dgregor -at- cs.indiana.edu>

Table of Contents

Introduction

The Boost.Function library contains a family of class templates that are function object wrappers. The notion is similar to a generalized callback. It shares features with function pointers in that both define a call interface (e.g., a function taking two integer arguments and returning a floating-point value) through which some implementation can be called, and the implementation that is invoked may change throughout the course of the program.

Generally, any place in which a function pointer would be used to defer a call or make a callback, Boost.Function can be used instead to allow the user greater flexibility in the implementation of the target. Targets can be any 'compatible' function object (or function pointer), meaning that the arguments to the interface designated by Boost.Function can be converted to the arguments of the target function object.

History & Compatibility Notes

- **Version 1.30.0**:
  - All features deprecated in version 1.29.0 have been removed from Boost.Function.

    - `boost::function` and `boost::functionN` objects can be assigned to 0 (semantically equivalent to calling `clear()`) and compared against 0 (semantically equivalent to calling `empty()`).
    - The Boost.Function code is now generated entirely by the Preprocessor library, so it is now possible to generate `boost::function` and `boost::functionN` class templates for any number of arguments.
    - The `boost::bad_function_call` exception class was introduced.
- **Version 1.29.0**: Boost.Function has been partially redesigned to minimize the interface and make it cleaner. Several seldom- or never-used features of the older Boost.Function have been deprecated and will be removed in the near future. Here is a list of features that have been deprecated, the likely impact of the deprecations, and how to adjust your code:
    - The `boost::function` class template syntax has changed. The old syntax, e.g., `boost::function<int, float, double, std::string>`, has been changed to a more natural syntax `boost::function<int (float, double, std::string)>`, where all return and argument types are encoded in a single function type parameter. Any other template parameters (e.g., the `Allocator`) follow this single parameter.

      The resolution to this change depends on the abilities of your compiler: if your compiler supports template partial specialization and can parse function types (most do), modify your code to use the newer syntax (preferable) or directly use one of the `functionN` classes whose syntax has not changed. If your compiler does not support template partial specialization or function types, you must take the latter option and use the numbered Boost.Function classes. This option merely requires changing types such as `boost::function<void, int, int>` to `boost::function2<void, int, int>` (adding the number of function arguments to the end of the class name).

      Support for the old syntax with the `boost::function` class template will persist for a short while, but will eventually be removed so that we can provide better error messages and link compatibility.
    - The invocation policy template parameter (`Policy`) has been deprecated and will be removed. There is no direct equivalent to this rarely used feature.
    - The mixin template parameter (`Mixin`) has been deprecated and will be removed. There is not direct equivalent to this rarely used feature.
    - The `set` methods have been deprecated and will be removed. Use the assignment operator instead.

Tutorial

Boost.Function has two syntactical forms: the preferred form and the portable form. The preferred form fits more closely with the C++ language and reduces the number of separate template parameters that need to be considered, often improving readability; however, the preferred form is not supported on all platforms due to compiler bugs. The compatible form will work on all compilers supported by Boost.Function. Consult the table below to determine which syntactic form to use for your compiler.

Preferred syntax	Portable syntax
• GNU C++ 2.95.x, 3.0.x, 3.1.x • Comeau C++ 4.2.45.2 • SGI MIPSpro 7.3.0 • Intel C++ 5.0, 6.0 • Compaq's cxx 6.2 • Microsoft Visual C++ 7.1	• *Any compiler supporting the preferred syntax* • Microsoft Visual C++ 6.0, 7.0 • Borland C++ 5.5.1 • Sun WorkShop 6 update 2 C++ 5.3 • Metrowerks CodeWarrior 8.1

If your compiler does not appear in this list, please try the preferred syntax and report your results to the

Boost list so that we can keep this table up-to-date.

Basic Usage

A function wrapper is defined simply by instantiating the `function` class template with the desired return type and argument types, formulated as a C++ function type. Any number of arguments may be supplied, up to some implementation-defined limit (10 is the default maximum). The following declares a function object wrapper `f` that takes two `int` parameters and returns a `float`:

Preferred syntax	Portable syntax
`boost::function<float (int x, int y)> f;`	`boost::function2<float, int, int> f;`

By default, function object wrappers are empty, so we can create a function object to assign to `f`:

```
struct int_div {
  float operator()(int x, int y) const { return ((float)x)/y; };
};
```

```
f = int_div();
```

Now we can use `f` to execute the underlying function object `int_div`:

```
std::cout << f(5, 3) << std::endl;
```

We are free to assign any compatible function object to `f`. If `int_div` had been declared to take two `long` operands, the implicit conversions would have been applied to the arguments without any user interference. The only limit on the types of arguments is that they be CopyConstructible, so we can even use references and arrays:

Preferred syntax
`boost::function<void (int values[], int n, int& sum, float& avg)> sum_avg;`

Portable syntax
`boost::function4<void, int*, int, int&, float&> sum_avg;`

```
void do_sum_avg(int values[], int n, int& sum, float& avg)
{
  sum = 0;
  for (int i = 0; i < n; i++)
    sum += values[i];
  avg = (float)sum / n;
}
```

```
sum_avg = &do_sum_avg;
```

Invoking a function object wrapper that does not actually contain a function object is a precondition violation, much like trying to call through a null function pointer, and will throw a `bad_function_call` exception). We can check for an empty function object wrapper by using it in a boolean context (it evaluates `true` if the wrapper is not empty) or compare it against `0`. For instance:

```
if (f)
  std::cout << f(5, 3) << std::endl;
else
  std::cout << "f has no target, so it is unsafe to call" << std::endl;
```

Alternatively, `empty()` method will return whether or not the wrapper is empty.

Finally, we can clear out a function target by assigning it to `0` or by calling the `clear()` member function, e.g.,

```
f = 0;
```

Free functions

Free function pointers can be considered singleton function objects with const function call operators, and can therefore be directly used with the function object wrappers:

```
float mul_ints(int x, int y) { return ((float)x) * y; }
```

```
f = &mul_ints;
```

Note that the & isn't really necessary unless you happen to be using Microsoft Visual C++ version 6.

Member functions

In many systems, callbacks often call to member functions of a particular object. This is often referred to as "argument binding", and is beyond the scope of Boost.Function. The use of member functions directly, however, is supported, so the following code is valid:

```
struct X {
  int foo(int);
};
```

<table>
<tr><th>Preferred syntax</th><th>Portable syntax</th></tr>
<tr><td><pre>boost::function<int (X*, int)> f;

f = &X::foo;

X x;
f(&x, 5);</pre></td><td><pre>boost::function2<int, X*, int> f;

f = &X::foo;

X x;
f(&x, 5);</pre></td></tr>
</table>

Several libraries exist that support argument binding. Three such libraries are summarized below:

- Bind. This library allows binding of arguments for any function object. It is lightweight and very portable.

- The C++ Standard library. Using `std::bind1st` and `std::mem_fun` together one can bind the object of a pointer-to-member function for use with Boost.Function:

Preferred syntax	Portable syntax

boost::function<int (int)> f;
X x;
f = std::bind1st(
      std::mem_fun(&X::foo), &x);
f(5); // Call x.foo(5)
	boost::function1<int, int> f;
	X x;
	f = std::bind1st(
	      std::mem_fun(&X::foo), &x);
	f(5); // Call x.foo(5)

- The Lambda library. This library provides a powerful composition mechanism to construct function objects that uses very natural C++ syntax. Lambda requires a compiler that is reasonably conformant to the C++ standard.

References to Function Objects

In some cases it is expensive (or semantically incorrect) to have Boost.Function clone a function object. In such cases, it is possible to request that Boost.Function keep only a reference to the actual function object. This is done using the `ref` and `cref` functions to wrap a reference to a function object:

Preferred syntax	Portable syntax

stateful_type a_function_object;
boost::function<int (int)> f;
f = boost::ref(a_function_object);
 	
boost::function<int (int)> f2(f);
	stateful_type a_function_object;
	boost::function1<int, int> f;
	f = boost::ref(a_function_object);
	
	boost::function1<int, int> f2(f);

Here, `f` will not make a copy of `a_function_object`, nor will `f2` when it is targeted to `f`'s reference to `a_function_object`. Additionally, when using references to function objects, Boost.Function will not throw exceptions during assignment or construction.

Comparing Boost.Function function objects

Function object wrappers can be compared via `==` or `!=` against any function object that can be stored within the wrapper. If the function object wrapper contains a function object of that type, it will be compared against the given function object (which must be either be EqualityComparable or have an overloaded `boost::function_equal`). For instance:

```
int compute_with_X(X*, int);

f = &X::foo;
assert(f == &X::foo);
assert(&compute_with_X != f);
```

When comparing against an instance of `reference_wrapper`, the address of the object in the `reference_wrapper` is compared against the address of the object stored by the function object wrapper:

```
a_stateful_object so1, so2;
f = boost::ref(so1);
```

```
assert(f == boost::ref(so1));
assert(f == so1); // Only if a_stateful_object is EqualityComparable
assert(f != boost::ref(so2));
```

Reference

Definitions

- A function object `f` is *compatible* if for the given set of argument types `Arg1`, `Arg2`, ..., `ArgN` and a return type `ResultType`, the appropriate following function is well-formed:

  ```
  // if ResultType is not void
    ResultType foo(Arg1 arg1, Arg2 arg2, ..., ArgN argN)
    {
      return f(arg1, arg2, ..., argN);
    }

    // if ResultType is void
    ResultType foo(Arg1 arg1, Arg2 arg2, ..., ArgN argN)
    {
      f(arg1, arg2, ..., argN);
    }
  ```

  A special provision is made for pointers to member functions. Though they are not function objects, Boost.Function will adapt them internally to function objects. This requires that a pointer to member function of the form `R (X::*mf)(Arg1, Arg2, ..., ArgN) cv-quals` be adapted to a function object with the following function call operator overloads:

  ```
  template<typename P>
    R operator()(cv-quals P& x, Arg1 arg1, Arg2 arg2, ..., ArgN argN) const
    {
      return (*x).*mf(arg1, arg2, ..., argN);
    }
  ```

- A function object `f` of type `F` is *stateless* if it is a function pointer or if `boost::is_stateless<T>` is true. The construction of or copy to a Boost.Function object from a stateless function object will not cause exceptions to be thrown and will not allocate any storage.

Header <boost/function.hpp>

```
namespace boost {
  class bad_function_call;
  class function_base;
  template<typename R, typename T1, typename T2, ..., typename TN,
           typename Allocator = std::allocator<void> >
    class functionN;
  template<typename T1, typename T2, ..., typename TN, typename Allocator>
    void swap(functionN<T1, T2, ..., TN, Allocator>&,
              functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator==(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
```

```
         typename Functor>
  bool operator==(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator==(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator!=(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
template<typename Signature, typename Allocator = std::allocator<void> >
  class function;
template<typename Signature, typename Allocator>
  void swap(function<Signature, Allocator>&,
            function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&,
                  reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator==(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&,
                  reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator!=(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
```

```
}
```

Class bad_function_call

Class bad_function_call -- An exception type thrown when an instance of a `function` object is empty when invoked.

```
class bad_function_call : public std::runtime_error {
public:
  // construct/copy/destruct
  bad_function_call();
};
```

Description

`bad_function_call` construct/copy/destruct

1. `bad_function_call();`

   Effects   Constructs a `bad_function_call` exception object.

Class function_base

Class function_base -- The common base class for all Boost.Function objects. Objects of type function_base may not be created directly.

```
class function_base {
public:

  // capacity
  bool empty() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;
};
```

Description

`function_base` capacity

1.
```
bool empty() const;
```

Returns `false` if `this` has a target, and `true` otherwise.

Throws Will not throw.

`function_base` target access

1.
```
template<typename Functor> Functor* target();
template<typename Functor> const Functor* target() const;
```

Returns If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

Throws Will not throw.

2.
```
template<typename Functor> bool contains(const Functor& f) const;
```

Returns `true` if `this->target<Functor>()` is non-NULL and `function_equal(*(this->target<Functor>()), f)`

Class template functionN

Class template functionN -- A set of generalized function pointers that can be used for callbacks or wrapping function objects.

```
template<typename R, typename T1, typename T2, ..., typename TN,
         typename Allocator = std::allocator<void> >
class functionN : public function_base {
public:
  // types
  typedef R         result_type;
  typedef Allocator allocator_type;
  typedef T1        argument_type;         // If N == 1
  typedef T1        first_argument_type;   // If N == 2
  typedef T2        second_argument_type;  // If N == 2
  typedef T1        arg1_type;
  typedef T2        arg2_type;
     .
     .
     .
  typedef TN        argN_type;

  // static constants
  static const int arity = N;

  // Lambda library support
  template<typename Args>
  struct sig {
    // types
    typedef result_type type;
  };

  // construct/copy/destruct
  functionN();
  functionN(const functionN&);
  template<typename F> functionN(F);
  functionN& operator=(const functionN&);
  ~functionN();

  // modifiers
  void swap(const functionN&);
  void clear();

  // capacity
  bool empty() const;
  operator safe_bool() const;
  bool operator!() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;

  // invocation
  result_type operator()(arg1_type, arg2_type, ..., argN_type) const;
};

// specialized algorithms
template<typename T1, typename T2, ..., typename TN, typename Allocator>
  void swap(functionN<T1, T2, ..., TN, Allocator>&,
            functionN<T1, T2, ..., TN, Allocator>&);
```

```
// comparison operators
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator==(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator!=(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
```

Description

Class template functionN is actually a family of related classes function0, function1, etc., up to some implementation-defined maximum. In this context, `N` refers to the number of parameters.

`functionN` construct/copy/destruct

1.
   ```
   functionN();
   ```

   Postconditions `this->empty()`

   Throws Will not throw.

2.
   ```
   functionN(const functionN& f);
   ```

   Postconditions Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.

   Throws Will not throw unless copying the target of `f` throws.

3.
```
template<typename F> functionN(F f);
```

Requires	F is a function object Callable from `this`.
Postconditions	`*this` targets a copy of `f` if `f` is nonempty, or `this->empty()` if `f` is empty.
Throws	Will not throw when `f` is a stateless function object.

4.
```
functionN& operator=(const functionN& f);
```

Postconditions	`*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`.
Throws	Will not throw when the target of `f` is a stateless function object or a reference to the function object.

5.
```
~functionN();
```

Effects	If `!this->empty()`, destroys the target of this.

`functionN` modifiers

1.
```
void swap(const functionN& f);
```

Effects	Interchanges the targets of `*this` and `f`.
Throws	Will not throw.

2.
```
void clear();
```

Postconditions	this->empty()
Throws	Will not throw.

`functionN` capacity

1.
```
bool empty() const;
```

Returns	`false` if `this` has a target, and `true` otherwise.
Throws	Will not throw.

2.
```
operator safe_bool() const;
```

Returns A `safe_bool` that evaluates `false` in a boolean context when `this->empty()`, and `true` otherwise.

Throws Will not throw.

3.
```
bool operator!() const;
```

Returns `this->empty()`

Throws Will not throw.

`functionN` target access

1.
```
template<typename Functor> Functor* target();
template<typename Functor> const Functor* target() const;
```

Returns If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

Throws Will not throw.

2.
```
template<typename Functor> bool contains(const Functor& f) const;
```

Returns `true` if `this->target<Functor>()` is non-NULL and `function_equal(*(this->target<Functor>()), f)`

`functionN` invocation

1.
```
result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN) const;
```

Effects `f(a1, a2, ..., aN)`, where `f` is the target of `*this`.

Returns if `R` is `void`, nothing is returned; otherwise, the return value of the call to `f` is returned.

Throws `bad_function_call` if `this->empty()`. Otherwise, may through any exception thrown by the target function `f`.

`functionN` specialized algorithms

1.
```
template<typename T1, typename T2, ..., typename TN, typename Allocator>
  void swap(functionN<T1, T2, ..., TN, Allocator>& f1,
            functionN<T1, T2, ..., TN, Allocator>& f2);
```

Effects `f1.swap(f2)`

Throws Will not throw.

`functionN` comparison operators

1.
```
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>& f, Functor g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(Functor g, const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(reference_wrapper<Functor> g,
                  const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator==(const functionN<T1, T2, ..., TN, Allocator1>& f1,
                  const functionN<U1, U2, ..., UN, Allocator2>& f2);
```

Returns True when `f` stores an object of type `Functor` and one of the following conditions applies:

- `g` is of type `reference_wrapper<Functor>` and `f.target<Functor>() == g.get_pointer()`.
- `g` is not of type `reference_wrapper<Functor>` and `function_equal(*(f.target<Functor>()), g)`.

Notes `functionN` objects are not EqualityComparable.

Rationale The `safe_bool` conversion opens a loophole whereby two `functionN` instances can be compared via `==`, although this is not feasible to implement. The undefined `void operator==` closes the loophole and ensures a compile-time or link-time error.

2.
```
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>& f, Functor g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(Functor g, const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(reference_wrapper<Functor> g,
                  const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
```

```
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator!=(const functionN<T1, T2, ..., TN, Allocator1>& f1,
                  const functionN<U1, U2, ..., UN, Allocator2>& f2);
```

Returns: True when `f` does not store an object of type `Functor` or it stores an object of type `Functor` and one of the following conditions applies:

- `g` is of type `reference_wrapper<Functor>` and `f.target<Functor>() != g.get_pointer()`.
- `g` is not of type `reference_wrapper<Functor>` and `!function_equal(*(f.target<Functor>()), g)`.

Notes: `functionN` objects are not EqualityComparable.

Rationale: The `safe_bool` conversion opens a loophole whereby two `functionN` instances can be compared via `!=`, although this is not feasible to implement. The undefined `void operator!=` closes the loophole and ensures a compile-time or link-time error.

Class template function

Class template function -- A generalized function pointer that can be used for callbacks or wrapping function objects.

```
template<typename Signature,   // Function type R (T1, T2, ..., TN)
         typename Allocator = std::allocator<void> >
class function : public functionN<R, T1, T2, ..., TN, Allocator> {
public:
  // types
  typedef R         result_type;
  typedef Allocator allocator_type;
  typedef T1        argument_type;         // If N == 1
  typedef T1        first_argument_type;   // If N == 2
  typedef T2        second_argument_type;  // If N == 2
  typedef T1        arg1_type;
  typedef T2        arg2_type;
     .
     .
     .
  typedef TN        argN_type;

  // static constants
  static const int arity = N;

  // Lambda library support
  template<typename Args>
  struct sig {
    // types
    typedef result_type type;
  };

  // construct/copy/destruct
  function();
  function(const functionN&);
  function(const function&);
  template<typename F> function(F);
  function& operator=(const functionN&);
  function& operator=(const function&);
  ~function();

  // modifiers
  void swap(const function&);
  void clear();

  // capacity
  bool empty() const;
  operator safe_bool() const;
  bool operator!() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;

  // invocation
  result_type operator()(arg1_type, arg2_type, ..., argN_type) const;
};

// specialized algorithms
template<typename Signature, typename Allocator>
```

```
  void swap(function<Signature, Allocator>&, function<Signature, Allocator>&);

// comparison operators
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&,
                  reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator==(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&,
                  reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator!=(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
```

Description

Class template function is a thin wrapper around the numbered class templates function0, function1, etc. It accepts a function type with N arguments and will will derive from functionN instantiated with the arguments it receives.

The semantics of all operations in class template function are equivalent to that of the underlying functionN object, although additional member functions are required to allow proper copy construction and copy assignment of function objects.

`function` construct/copy/destruct

1.

```
function();
```

Postconditions	`this->empty()`
Throws	Will not throw.

2.

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

Postconditions	Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.
Throws	Will not throw unless copying the target of `f` throws.

3.

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

Postconditions — Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.

Throws — Will not throw unless copying the target of `f` throws.

4. ```
   template<typename F> function(F f);
   ```

   Requires — F is a function object Callable from `this`.

   Postconditions — `*this` targets a copy of `f` if `f` is nonempty, or `this->empty()` if `f` is empty.

   Throws — Will not throw when `f` is a stateless function object.

5. ```
   function& operator=(const functionN& f);
   ```

   Postconditions — `*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`

   Throws — Will not throw when the target of `f` is a stateless function object or a reference to the function object.

6. ```
   function& operator=(const function& f);
   ```

   Postconditions — `*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`

   Throws — Will not throw when the target of `f` is a stateless function object or a reference to the function object.

7. ```
   ~function();
   ```

   Effects — If `!this->empty()`, destroys the target of `this`.

`function` modifiers

1. ```
   void swap(const function& f);
   ```

   Effects — Interchanges the targets of `*this` and `f`.

   Throws — Will not throw.

2. ```
   void clear();
   ```

   Postconditions — `this->empty()`

   Throws — Will not throw.

`function` capacity

1.
```
bool empty() const;
```

Returns: `false` if `this` has a target, and `true` otherwise.

Throws: Will not throw.

2.
```
operator safe_bool() const;
```

Returns: A `safe_bool` that evaluates `false` in a boolean context when `this->empty()`, and `true` otherwise.

Throws: Will not throw.

3.
```
bool operator!() const;
```

Returns: `this->empty()`

Throws: Will not throw.

`function` target access

1.
```
template<typename Functor> Functor* target();
template<typename Functor> const Functor* target() const;
```

Returns: If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

Throws: Will not throw.

2.
```
template<typename Functor> bool contains(const Functor& f) const;
```

Returns: `true` if `this->target<Functor>()` is non-NULL and `function_equal(*(this->target<Functor>()), f)`

`function` invocation

1.
```
result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN) const;
```

Effects: `f(a1, a2, ..., aN)`, where `f` is the target of `*this`.

Returns: if `R` is `void`, nothing is returned; otherwise, the return value of the call to `f` is returned.

Throws bad_function_call if this->empty(). Otherwise, may through any exception thrown by the target function f.

function specialized algorithms

1.

```
template<typename Signature, typename Allocator>
  void swap(function<Signature, Allocator>& f1,
            function<Signature, Allocator>& f2);
```

Effects f1.swap(f2)

Throws Will not throw.

function comparison operators

1.

```
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>& f, Functor g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(Functor g, const function<Signature, Allocator>& f);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(reference_wrapper<Functor> g,
                  const function<Signature, Allocator>& f);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator==(const function<Signature1, Allocator1>& f1,
                  const function<Signature2, Allocator2>& f2);
```

Returns True when f stores an object of type Functor and one of the following conditions applies:

- g is of type reference_wrapper<Functor> and f.target<Functor>() == g.get_pointer().
- g is not of type reference_wrapper<Functor> and function_equals(*(f.target<Functor>()), g).

Notes function objects are not EqualityComparable.

Rationale The safe_bool conversion opens a loophole whereby two function instances can be compared via ==, although this is not feasible to implement. The undefined void operator== closes the loophole and ensures a compile-time or link-time error.

2.

```
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>& f, Functor g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(Functor g, const function<Signature, Allocator>& f);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>& f,
```

```
                  reference_wrapper<Functor> g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(reference_wrapper<Functor> g,
                  const function<Signature, Allocator>& f);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator!=(const function<Signature1, Allocator1>& f1,
                  const function<Signature2, Allocator2>& f2);
```

Returns: True when `f` does not store an object of type `Functor` or it stores an object of type `Functor` and one of the following conditions applies:

- `g` is of type `reference_wrapper<Functor>` and `f.target<Functor>() != g.get_pointer()`.
- `g` is not of type `reference_wrapper<Functor>` and `!function_equals(*(f.target<Functor>()), g)`.

Notes: `function` objects are not EqualityComparable.

Rationale: The `safe_bool` conversion opens a loophole whereby two `function` instances can be compared via `!=`, although this is not feasible to implement. The undefined `void operator!=` closes the loophole and ensures a compile-time or link-time error.

Header <boost/function_equal.hpp>

```
namespace boost {
  template<typename F, typename G> bool function_equal(const F&, const G&);
}
```

Function template function_equal

Function template function_equal --

Compare two function objects for equality.

```
template<typename F, typename G> bool function_equal(const F& f, const G& g);
```

Description

Returns	`f == g`.
Throws	Only if `f == g` throws.

Frequently Asked Questions

1.
Why can't I compare `boost::function` objects with `operator==` or `operator!=`?

Comparison between `boost::function` objects cannot be implemented "well", and therefore will not be implemented. The typical semantics requested for `f == g` given `boost::function` objects `f` and `g` are:

- If `f` and `g` store function objects of the same type, use that type's `operator==` to compare them.
- If `f` and `g` store function objects of different types, return `false`.

The problem occurs when the type of the function objects stored by both `f` and `g` doesn't have an `operator==`: we would like the expression `f == g` to fail to compile, as occurs with, e.g., the standard containers. However, this is not implementable for `boost::function` because it necessarily "erases" some type information after it has been assigned a function object, so it cannot try to call `operator==` later: it must either find a way to call `operator==` now, or it will never be able to call it later. Note, for instance, what happens if you try to put a `float` value into a `boost::function` object: you will get an error at the assignment operator or constructor, not in `operator()`, because the function-call expression must be bound in the constructor or assignment operator.

The most promising approach is to find a method of determining if `operator==` can be called for a particular type, and then supporting it only when it is available; in other situations, an exception would be thrown. However, to date there is no known way to detect if an arbitrary operator expression `f == g` is suitably defined. The best solution known has the following undesirable qualities:

1. Fails at compile-time for objects where `operator==` is not accessible (e.g., because it is `private`).

2. Fails at compile-time if calling `operator==` is ambiguous.

3. Appears to be correct if the `operator==` declaration is correct, even though `operator==` may not compile.

All of these problems translate into failures in the `boost::function` constructors or assignment operator, *even if the user never invokes operator==*. We can't do that to users.

The other option is to place the burden on users that want to use `operator==`, e.g., by providing an `is_equality_comparable` trait they may specialize. This is a workable solution, but is dangerous in practice, because forgetting to specialize the trait will result in unexpected exceptions being thrown from `boost::function`'s `operator==`. This essentially negates the usefulness of `operator==` in the context in which it is most desired: multitarget callbacks. The Signals library has a way around this.

2.

I see void pointers; is this [mess] type safe?

Yes, `boost::function` is type safe even though it uses void pointers and pointers to functions returning void and taking no arguments. Essentially, all type information is encoded in the functions that manage and invoke function pointers and function objects. Only these functions are instantiated with the exact type that is pointed to by the void pointer or pointer to void function. The reason that both are required is that one may cast between void pointers and object pointers safely or between different types of function pointers (provided you don't invoke a function pointer with the wrong type).

3.

Why are there workarounds for void returns? C++ allows them!

Void returns are permitted by the C++ standard, as in this code snippet:

```
void f();
void g() { return f(); }
```

This is a valid usage of `boost::function` because void returns are not used. With void returns, we would attempting to compile ill-formed code similar to:

```
int f();
void g() { return f(); }
```

In essence, not using void returns allows `boost::function` to swallow a return value. This is consistent with allowing the user to assign and invoke functions and function objects with parameters that don't exactly match.

4.

Why (function) cloning?

In November and December of 2000, the issue of cloning vs. reference counting was debated at length and it was decided that cloning gave more predictable semantics. I won't rehash the discussion here, but if it cloning is incorrect for a particular application a reference-counting allocator could be used.

5.

How much overhead does a call through `boost::function` incur?

The cost of `boost::function` can be reasonably consistently measured at around 20ns +/- 10 ns on a modern >2GHz platform versus directly inlining the code.

However, the performance of your application may benefit from or be disadvantaged by `boost::function` depending on how your C++ optimiser optimises. Similar to a standard function pointer, differences of order of 10% have been noted to the benefit or disadvantage of using `boost::function` to call a function that contains a tight loop depending on your compilation circumstances.

[Answer provided by Matt Hurd. See http://article.gmane.org/gmane.comp.lib.boost.devel/33278]

Miscellaneous Notes

Boost.Function vs. Function Pointers

Boost.Function has several advantages over function pointers, namely:

- Boost.Function allows arbitrary compatible function objects to be targets (instead of requiring an exact function signature).
- Boost.Function may be used with argument-binding and other function object construction libraries.
- Boost.Function has predictible behavior when an empty function object is called.

And, of course, function pointers have several advantages over Boost.Function:

- Function pointers are smaller (the size of one pointer instead of three)
- Function pointers are faster (Boost.Function may require two calls through function pointers)
- Function pointers are backward-compatible with C libraries.
- More readable error messages.

Performance

Function object wrapper size

Function object wrappers will be the size of two function pointers plus one function pointer or data pointer (whichever is larger). On common 32-bit platforms, this amounts to 12 bytes per wrapper. Additionally, the function object target will be allocated on the heap.

Copying efficiency

Copying function object wrappers may require allocating memory for a copy of the function object target. The default allocator may be replaced with a faster custom allocator or one may choose to allow the function object wrappers to only store function object targets by reference (using `ref`) if the cost of this cloning becomes prohibitive.

Invocation efficiency

With a properly inlining compiler, an invocation of a function object requires one call through a function pointer. If the call is to a free function pointer, an additional call must be made to that function pointer (unless the compiler has very powerful interprocedural analysis).

Combatting virtual function "bloat"

The use of virtual functions tends to cause 'code bloat' on many compilers. When a class contains a virtual function, it is necessary to emit an additional function that classifies the type of the object. It has been our experience that these auxiliary functions increase the size of the executable significantly when many `boost::function` objects are used.

In Boost.Function, an alternative but equivalent approach was taken using free functions instead of virtual functions. The Boost.Function object essentially holds two pointers to make a valid target call: a void pointer to the function object it contains and a void pointer to an "invoker" that can call the function object, given the function pointer. This invoker function performs the argument and return value conversions Boost.Function provides. A third pointer points to a free function called the "manager", which handles the cloning and destruction of function objects. The scheme is typesafe because the only functions that actually handle the function object, the invoker and the manager, are instantiated given the type of the function object, so they can safely cast the incoming void pointer (the function object pointer) to the appropriate type.

Acknowledgements

Many people were involved in the construction of this library. William Kempf, Jesse Jones and Karl Nelson were all extremely helpful in isolating an interface and scope for the library. John Maddock managed the formal review, and many reviewers gave excellent comments on interface, implementation, and documentation. Peter Dimov led us to the function declarator-based syntax.

Testsuite

Acceptance tests

Test	Type	Description
function_test.cpp	run	Test the capabilities of the `boost::function` class template.
function_n_test.cpp	run	Test the capabilities of the `boost::functionN` class templates.
allocator_test.cpp	run	Test the use of custom allocators.
stateless_test.cpp	run	Test the optimization of stateless function objects in the Boost.Function library.
lambda_test.cpp	run	Test the interaction between Boost.Function and Boost.Lambda.
contains_test.cpp	run	Test the operation of the `target` member function and the equality operators.
function_30.cpp	compile	Test the generation of a Boost.Function function object adaptor accepting 30 arguments.

Test	Type	Description
function_arith_cxx98.cpp	run	Test the first tutorial example.
function_arith_portable.cpp	run	Test the first tutorial example.
sum_avg_cxx98.cpp	run	Test the second tutorial example.
sum_avg_portable.cpp	run	Test the second tutorial example.
mem_fun_cxx98.cpp	run	Test member function example from tutorial.
mem_fun_portable.cpp	run	Test member function example from tutorial.
std_bind_cxx98.cpp	run	Test standard binders example from tutorial.
std_bind_portable.cpp	run	Test standard binders example from tutorial.
function_ref_cxx98.cpp	run	Test `boost::ref` example from tutorial.
function_ref_portable.cpp	run	Test `boost::ref` example from tutorial.

Negative tests

Test	Type	Description
function_test_fail1.cpp	compile-fail	Test the (incorrect!) use of comparisons between Boost.Function function objects.
function_test_fail2.cpp	compile-fail	Test the use of an incompatible function object with Boost.Function