Boost.Functional/OverloadedFunction 1.0.0

Lorenzo Caminiti <lorcaminiti@gmail.com>

Table of Contents

This library allows to overload different functions into a single function object.

Introduction

Consider the following functions which have distinct signatures:

```
const std::string& identity_s(const std::string& x) // Function (as pointer).
    { return x; }

int identity_i_impl(int x) { return x; }
int (&identity_i)(int) = identity_i_impl; // Function reference.

double identity_d_impl(double x) { return x; }
boost::function<double (double)> identity_d = identity_d_impl; // Functor.
```

Instead of calling them using their separate names (here `BOOST_TEST` is equivalent to `assert`): [1]

```
BOOST_TEST(identity_s("abc") == "abc");
BOOST_TEST(identity_i(123) == 123);
BOOST_TEST(identity_d(1.23) == 1.23);
```

It is possible to use this library to create a single overloaded function object (or functor) named `identity` that aggregates together the calls to the specific functions (see also `functor.cpp` and `identity.hpp`):

```
boost::overloaded_function<
      const std::string& (const std::string&)
    , int (int)
    , double (double)
> identity(identity_s, identity_i, identity_d);

// All calls via single `identity` function.
BOOST_TEST(identity("abc") == "abc");
BOOST_TEST(identity(123) == 123);
BOOST_TEST(identity(1.23) == 1.23);
```

Note how the functions are called via a single overloaded function object `identity` instead of using their different names `identity_s`, `identity_i`, and `identity_d`.

[1] In most of the examples presented in this documentation, the Boost.Detail/LightweightTest (`boost/detail/lightweight_test.hpp`) macro `BOOST_TEST` is used to check correctness conditions (conceptually similar to `assert`). A failure of the checked condition does not abort the execution of the program, it will instead make `boost::report_errors` return a non-zero program exit code. Using Boost.Detail/LightweightTest allows to add the examples to the library regression tests so to make sure that they always compile and run correctly.

Getting Started

This section explains how to setup a system to use this library.

Compilers and Platforms

The authors originally developed and tested this library on:

1. GNU Compiler Collection (GCC) C++ 4.5.3 (with and without C++11 features enabled `-std=c++0x`) on Cygwin.
2. Miscrosoft Visual C++ (MSVC) 8.0 on Windows 7.

See the library regressions test results for detailed information on supported compilers and platforms. Check the library regression test `Jamfile.v2` for any special configuration that might be required for a specific compiler.

Installation

This library is composed of header files only. Therefore there is no pre-compiled object file which needs to be installed. Programmers can simply instruct the compiler where to find the library header files (`-I` option on GCC, `/I` option on MSVC, etc) and compile code using the library.

The maximum number of functions to overload is given by the `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX` configuration macro. The maximum number of function parameters for each of the specified function type is given by the `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX` configuration macro. All configuration macros have appropriate default values when they are left undefined.

Tutorial

This section explains how to use this library.

Overloading

Consider the following functions which have distinct signatures:

```
const std::string& identity_s(const std::string& x) // Function (as pointer).
    { return x; }

int identity_i_impl(int x) { return x; }
int (&identity_i)(int) = identity_i_impl; // Function reference.

double identity_d_impl(double x) { return x; }
boost::function<double (double)> identity_d = identity_d_impl; // Functor.
```

This library header `boost/functional/overloaded_function.hpp` provides a `boost::overloaded_function` class template that creates a single overloaded function object that can be used to call the specified functions instead of using the separate function names (see also `functor.cpp` and `identity.hpp`):

```
boost::overloaded_function<
      const std::string& (const std::string&)
    , int (int)
    , double (double)
> identity(identity_s, identity_i, identity_d);

// All calls via single `identity` function.
BOOST_TEST(identity("abc") == "abc");
BOOST_TEST(identity(123) == 123);
BOOST_TEST(identity(1.23) == 1.23);
```

Note how each function type is passed as a template parameter of `boost::overloaded_function` using the following syntax (this is Boost.Function's preferred syntax):

```
result-type (argument1-type, argument2-type, ...)
```

Then the relative function pointers, function references, or monomorphic function objects are passed to the `boost::overloaded_function` constructor matching the order of the specified template parameters. [2] In the above example, `identity_s` is passed as a function pointer (the function address is automatically taken from the function name by the compiler), `identity_i` as a function reference, and `identity_d` as a function object.

All specified function types must have distinct parameters from one another (so the overloaded calls can be resolved by this library). [3] In order to create an overloaded function object, it is necessary to specify at least two function types (because there is nothing to overload between one or zero functions).

[2] Function pointers are of the form `result-type (*)(argument1-type, ...)` (the C++ compiler is usually able to automatically promote a function name to a function pointer in a context where a function pointer is expected even if the function name is not prefixed by `&`). Function references are of the form `result-type (&)(argument1-type, ...)`. Function types are of the form `result-type (argument1-type, ...)` (note how they lack of both `*` and `&` when compared to function pointers and function references). Finally, monomorphic function objects are instances of classes with a non-template call operator of the form `result-type operator()(argument1-type, ...)`. Unfortunately, it is not possible to support polymorphic function objects (see http://lists.boost.org/Archives/boost/2012/03/191744.php).

[3] Note that in C++ the function result type is not used for overload resolution (to avoid making the overload resolution context dependent). Therefore, at least one of the function parameters must be distinct for each specified function type.

Without Function Types

For convenience, this library also provides the `boost::make_overloaded_function` function template which allows to create the overloaded function object without explicitly specifying the function types. The function types are automatically deduced from the specified functions and the appropriate `boost::overloaded_function` instantiation is returned by `boost::make_overloaded_function`.

The `boost::make_overloaded_function` function template can be useful when used together with Boost.Typeof's `BOOST_AUTO` (or C++11 `auto`). For example (see also `make_decl.cpp` and `identity.hpp`):

```
BOOST_AUTO(identity, boost::make_overloaded_function(
        identity_s, identity_i, identity_d));

BOOST_TEST(identity("abc") == "abc");
BOOST_TEST(identity(123) == 123);
BOOST_TEST(identity(1.23) == 1.23);
```

Note how the overloaded function object `identity` has been created specifying only the functions `identity_s`, `identity_i`, `identity_d` and without specifying the function types `const std::string& (const std::string&)`, `int (int)`, and `double (double)` as required instead by `boost::overloaded_function`. Therefore, `boost::make_overloaded_function` provides a more concise syntax in this context when compared with `boost::overloaded_function`.

Another case where `boost::make_overloaded_function` can be useful is when the overloaded function object is passed to a function template which can hold the specific `boost::overloaded_function` type using a template parameter. For example (see also `make_call.cpp` and `identity.hpp`):

```
template<typename F>
void check(F identity) {
    BOOST_TEST(identity("abc") == "abc");
    BOOST_TEST(identity(123) == 123);
    BOOST_TEST(identity(1.23) == 1.23);
}
```

```
check(boost::make_overloaded_function(identity_s, identity_i, identity_d));
```

The library implementation of `boost::make_overloaded_function` uses Boost.Typeof to automatically deduce some of the function types. In order to compile code in Boost.Typeof emulation mode, all types should be properly registered using `BOOST_TYPEOF_REGISTER_TYPE` and `BOOST_TYPEOF_REGISTER_TEMPLATE`, or appropriate Boost.Typeof headers should be included (see Boost.Typeof for more information). For the above examples, it is sufficient to include the Boost.Typeof header that registers `std::string` (this library does not require to register `boost::function` for Boost.Typeof emulation):

```
#include <boost/typeof/std/string.hpp> // No need to register `boost::function`.
```

Reference

Header <boost/functional/overloaded_function.hpp>

Overload distinct function pointers, function references, and monomorphic function objects into a single function object.

```
namespace boost {
  template<typename F1, typename F2, ... > class overloaded_function;
  template<typename F1, typename F2, ... >
    overloaded_function< __function_type__< F1 >, __function_type__< F2 >,...>
    make_overloaded_function(F1, F2, ...);
}
```

Class template overloaded_function

boost::overloaded_function — Function object to overload functions with distinct signatures.

Synopsis

```
// In header: <boost/functional/overloaded_function.hpp>

template<typename F1, typename F2, ... >
class overloaded_function {
public:
  // construct/copy/destruct
  overloaded_function(const boost::function< F1 > &,
                      const boost::function< F2 > &, ...);

  // public member functions
  boost::function_traits< F1 >::result_type
  operator()(typename boost::function_traits< F1 >::arg1_type,
             typename boost::function_traits< F1 >::arg2_type, ...) const;
  boost::function_traits< F2 >::result_type
  operator()(typename boost::function_traits< F2 >::arg1_type,
             typename boost::function_traits< F2 >::arg2_type, ...) const;
};
```

Description

This function object aggregates together calls to functions of all the specified function types `F1`, `F2`, etc which must have distinct function signatures from one another.

Parameters:

`Fi`	Each function type must be specified using the following syntax (which is Boost.Function's preferred syntax): `result_type (argument1_type, argumgnet2_type, ...)`

In some cases, the `make_overloaded_function` function template can be useful to construct an overloaded function object without explicitly specifying the function types.

At least two distinct function types must be specified (because there is nothing to overload between one or zero functions). The maximum number of functions to overload is given by the `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX` configuration macro. The maximum number of function parameters for each of the specified function types is given by the `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX` configuration macro.

See: Tutorial section, `make_overloaded_function`, `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX`, `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX`, Boost.Function.

`overloaded_function` public construct/copy/destruct

1.
```
overloaded_function(const boost::function< F1 > &,
                    const boost::function< F2 > &, ...);
```

Construct the overloaded function object.

Any function pointer, function reference, and monomorphic function object that can be converted to a `boost::function` function object can be specified as parameter.

Note: Unfortunately, it is not possible to support polymorphic function objects (as explained here).

`overloaded_function` public member functions

1.
```
boost::function_traits< F1 >::result_type
operator()(typename boost::function_traits< F1 >::arg1_type,
           typename boost::function_traits< F1 >::arg2_type, ...) const;
```

Call operator matching the signature of the function type specified as 1st template parameter.

This will in turn invoke the call operator of the 1st function passed to the constructor.

2.
```
boost::function_traits< F2 >::result_type
operator()(typename boost::function_traits< F2 >::arg1_type,
           typename boost::function_traits< F2 >::arg2_type, ...) const;
```

Call operator matching the signature of the function type specified as 2nd template parameter.

This will in turn invoke the call operator of the 2nd function passed to the constructor.

Note: Similar call operators are present for all specified function types `F1`, `F2`, etc (even if not exhaustively listed by this documentation).

Function template make_overloaded_function

boost::make_overloaded_function — Make an overloaded function object without explicitly specifying the function types.

Synopsis

```
// In header: <boost/functional/overloaded_function.hpp>


template<typename F1, typename F2, ... >
  overloaded_function< __function_type__< F1 >, __function_type__< F2 >,...>
  make_overloaded_function(F1 f1, F2 f2, ...);
```

Description

This function template creates and returns an `overloaded_function` object that overloads all the specified functions `f1`, `f2`, etc.

The function types are internally determined from the template parameter types so they do not need to be explicitly specified. Therefore, this function template usually has a more concise syntax when compared with `overloaded_function`. This is especially useful when the explicit type of the returned `overloaded_function` object does not need to be known (e.g., when used with Boost.Typeof's `BOOST_AUTO`, C++11 `auto`, or when the overloaded function object is handled using a function template parameter, see the Tutorial section).

The maximum number of functions to overload is given by the `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX` configuration macro.

Note: In this documentation, `__function_type__` is a placeholder for a symbol that is specific to the implementation of this library.

See: Tutorial section, `overloaded_function`, `BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX`.

Header <boost/functional/overloaded_function/config.hpp>

Change the compile-time configuration of this library.

```


BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX
BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX
```

Macro BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX

BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX — Specify the maximum number of arguments of the functions being overloaded.

Synopsis

```
// In header: <boost/functional/overloaded_function/config.hpp>

BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_ARITY_MAX
```

Description

If this macro is left undefined by the user, it has a default value of 5 (increasing this number might increase compilation time). When specified by the user, this macro must be a non-negative integer number.

See: Getting Started, `boost::overloaded_function`.

Macro BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX

BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX — Specify the maximum number of functions that can be overloaded.

Synopsis

```
// In header: <boost/functional/overloaded_function/config.hpp>

BOOST_FUNCTIONAL_OVERLOADED_FUNCTION_CONFIG_OVERLOAD_MAX
```

Description

If this macro is left undefined by the user, it has a default value of 5 (increasing this number might increase compilation time). When defined by the user, this macro must be an integer number greater or equal than 2 (because at least two distinct functions need to be specified in order to define an overload).

See: Getting Started, `boost::overloaded_function`.

Acknowledgments

Many thanks to Mathias Gaunard for suggesting to implement `boost::overloaded_function` and for some sample code.

Thanks to John Bytheway for suggesting to implement `boost::make_overloaded_function`.

Thanks to Nathan Ridge for suggestions on how to implement `boost::make_overloaded_function`.

Thanks to Robert Stewart for commenting on the library name.

Many thanks to the entire Boost community and mailing list for providing valuable comments about this library and great insights on the C++ programming language.