Boost.LocalFunction 1.0.0

Lorenzo Caminiti <lorcaminiti@gmail.com>

Table of Contents

This library allows to program functions locally, within other functions, and directly within the scope where they are needed.

Introduction

Local functions (a.k.a., *nested functions*) are a form of *information hiding* and they are useful for dividing procedural tasks into subtasks which are only meaningful locally, avoiding cluttering other parts of the program with functions, variables, etc unrelated to those parts. Therefore, local functions complement other structuring possibilities such as namespaces and classes. Local functions are a feature of many programming languages, notably Pascal and Ada, yet lacking from C++03 (see also [N2511]).

Using C++11 lambda functions, it is possible to implement local functions by naming lambda functions assigning them to local variables. For example (see also `add_cxx11_lambda.cpp`):

```
int main(void) {                                // Some local scope.
    int sum = 0, factor = 10;                   // Variables in scope to bind.

    auto add = [factor, &sum](int num) {        // C++11 only.
        sum += factor * num;
    };

    add(1);                                     // Call the lambda.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      // Assert final summation value.
    return boost::report_errors();
}
```

This library allows to program local functions portably between C++03 and C++11 (and with performances comparable to lambda functions on C++11 compilers). For example (see also `add.cpp`):

```
int main(void) {                                // Some local scope.
    int sum = 0, factor = 10;                   // Variables in scope to bind.

    void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                     // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      // Assert final summation value.
    return boost::report_errors();
}
```

This library supports the following features for local functions:

- Local functions can capture, or better *bind*, any of the variables from the enclosing scope (a function together with its captured variables is also called a *closure*).
- The local function body is programmed using the usual C++ statement syntax (as a consequence, compiler errors and debugging retain their usual meaning and format).
- Local functions can be passed as template parameters so they can be conveniently used with STL algorithms and other templates. [1]
- However, local functions must be specified within a declarative context (e.g., at a point in the code where local variables can be declared) thus they cannot be specified within expressions. [2]

[1] This is a strength with respect to C++03 functors implemented using local classes which cannot be passed as template parameters (see [N2657] and the Alternatives section).
[2] This is a weakness with respect to C++11 lambda functions which can instead be specified also within expressions (see the Alternatives section).

See the Alternatives section for a comparison between this library, C++11 lambda functions, Boost.Phoenix, and other C++ techniques that implement features related to local functions.

Getting Started

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

This Documentation

Programmers should have enough knowledge to use this library after reading the Introduction, Getting Started, and Tutorial sections. The Advanced Topics and Reference sections can be consulted at a later point to gain a more advanced knowledge of the library. All the other sections of this documentation can be considered optional.

Some footnotes are marked by the word "**Rationale**". They explain reasons behind decisions made during the design and implementation of this library.

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. The `BOOST_TEST` macro is conceptually similar to `assert` but a failure of the checked condition does not abort the program, instead it makes `boost::report_errors` return a non-zero program exit code. [3]

Compilers and Platforms

The implementation of this library uses preprocessor and template meta-programming (as supported by Boost.Preprocessor and Boost.MPL), templates with partial specializations and function pointers (similarly to Boost.Function), and automatic type deduction (as supported by Boost.Typeof). The authors originally developed and tested the library on:

1. GNU Compiler Collection (GCC) C++ 4.5.1 on Ubuntu Linux 10.
2. GCC 4.3.4 and 4.5.3 (with and without C++11 features enabled `-std=c++0x`) on Cygwin.
3. Miscrosoft Visual C++ (MSVC) 8.0 on Windows XP and Windows 7.

See the library regressions test results for detailed information on supported compilers and platforms.

Installation

This library is composed of header files only. Therefore there is no pre-compiled object file which needs to be installed or linked. Programmers can simply instruct the C++ compiler where to find the library header files (`-I` option for GCC, `/I` option for MSVC, etc) and they can start compiling code using this library.

The library implementation uses Boost.Typeof to automatically deduce the types of bound variables (see the Tutorial section). In order to compile code in type-of 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 the source code of most examples presented in this documentation).

The followings are part of the library private API, they are not documented, and they should not be directly used by programmers: [4]

- Any symbol defined by files within the `boost/local_function/aux_/` or `boost/local_function/detail/` directory (these header files should not be directly included by programmers).
- Any symbol within the `boost::local_function::aux` or `boost::local_function::detail` namespace.

[3] **Rationale.** 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.

[4] **Rationale.** This library concatenates symbols specified by the programmers (e.g., the local function name) with other symbols (e.g., special prefixes or file line numbers) to make internal symbols with unique names to avoid name clashes. These symbols are separated by the letter "X" when they are concatenated so they read more easily during debugging (the underscore character "_" could not be used instead of the letter "X" because if the original symbols already contained a leading or trailing underscore, the concatenation could result in a symbol with double underscores "__" which is reserved by the C++ standard). The "aux" symbols are private to this library while the "detail" symbols may be used within Boost by other libraries but they are still not part of this library public API.

- Any symbol prefixed by `boost_local_function_aux_...` or `boost_local_function_detail_...` (regardless of its namespace).
- Any symbol prefixed by `BOOST_LOCAL_FUNCTION_AUX_...` or `BOOST_LOCAL_FUNCTION_DETAIL_...` (regardless of its namespace).

Some of the library behaviour can be changed at compile-time by defining special *configuration macros*. If a configuration macro is left undefined, the library will use an appropriate default value for it. All configuration macros are defined in the header file `boost/local_function/config.hpp`. It is strongly recommended not to change the library configuration macro definitions unless strictly necessary.

Tutorial

This section illustrates basic usage of this library.

Local Functions

Local functions are defined using macros from the header file `boost/local_function.hpp`. The macros must be used from within a declarative context (this is a limitation with respect to C++11 lambda functions which can instead be declared also within expressions):

```
#include <boost/local_function.hpp> // This library header.

...
{ // Some declarative context.
    ...
    result-type BOOST_LOCAL_FUNCTION(parameters) {
        body-code
    } BOOST_LOCAL_FUNCTION_NAME(name)
    ...
}
```

The code expanded by the macros declares a function object (or functor) with the local function name specified by `BOOST_LOCAL_FUNCTION_NAME`. [5] The usual C++ scope visibility rules apply to local functions for which a local function is visible only within the enclosing scope in which it is declared.

The local function result type is specified just before the `BOOST_LOCAL_FUNCTION` macro.

The local function body is specified using the usual C++ statement syntax in a code block `{ ... }` between the `BOOST_LOCAL_FUNCTION` and `BOOST_LOCAL_FUNCTION_NAME` macros. The body is specified outside any of the macros so eventual compiler error messages and related line numbers retain their usual meaning and format. [6]

The local function parameters are passed to the `BOOST_LOCAL_FUNCTION` macro as a comma-separated list of tokens (see the No Variadic Macros section for compilers that do not support variadic macros):

```
BOOST_LOCAL_FUNCTION(parameter-type1 parameter-name1, parameter-type2 parameter-name2, ...)
```

The maximum number of parameters that can be passed to a local function is controlled at compile-time by the configuration macro `BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX`. For example, let's program a local function named `add` that adds together two integers `x` and `y` (see also `add_params_only.cpp`):

```
int BOOST_LOCAL_FUNCTION(int x, int y) { // Local function.
    return x + y;
} BOOST_LOCAL_FUNCTION_NAME(add)

BOOST_TEST(add(1, 2) == 3); // Local function call.
```

If the local function has no parameter, it is possible to pass `void` to the `BOOST_LOCAL_FUNCTION` macro (similarly to the C++ syntax that allows to use `result-type function-name(void)` to declare a function with no parameter): [7]

[5] **Rationale.** The local function name must be passed to the macro `BOOST_LOCAL_FUNCTION_NAME` ending the function definition so this macro can declare a local variable with the local function name to hold the local function object. Therefore the local function name cannot be specified within the `BOOST_LOCAL_FUNCTION` and it must appear instead after the local function body (even if that differs from the usual C++ function declaration syntax).

[6] **Rationale.** If the local function body were instead passed as a macro parameter, it would be expanded on a single line of code (because macros always expand as a single line of code). Therefore, eventual compiler error line numbers would all report the same value and would no longer be useful to pinpoint errors.

[7] **Rationale.** The C++03 standard does not allow to pass empty parameters to a macro so the macro cannot be invoked as `BOOST_LOCAL_FUNCTION()`. On C99 compilers with properly implemented empty macro parameter support, it would be possible to allow `BOOST_LOCAL_FUNCTION()` but this is already not the case for MSVC so this syntax is never allowed to ensure better portability.

```
BOOST_LOCAL_FUNCTION(void) // No parameter.
```

For example, let's program a local function that always returns `10` (see also `ten_void.cpp`):

```
int BOOST_LOCAL_FUNCTION(void) { // No parameter.
    return 10;
} BOOST_LOCAL_FUNCTION_NAME(ten)

BOOST_TEST(ten() == 10);
```

Binding Variables

Variables in scope (local variables, enclosing function parameters, data members, etc) can be bound to a local function declaration. Only bound variables, static variables, global variables, functions, and enumerations from the enclosing scope are accessible from within the local function body. The types of bound variables are deduced automatically by this library using Boost.Typeof. [8]

This library introduces the new "keyword" `bind` [9] which is used in place of the parameter type to specify the name of a variable in scope to bind (therefore, `bind` cannot be used as a local function parameter type). A variable can be bound by value:

```
bind variable-name // Bind by value.
```

Or by reference prefixing the variable name with `&`:

```
bind& variable-name // Bind by reference.
```

Furthermore, the "keyword" `bind` can be prefixed by `const` to bind the variable by constant value:

```
const bind variable-name // Bind by constant value.
```

Or by constant reference:

```
const bind& variable-name // Bind by constant value.
```

Note that when `const` is used, it must always precede `bind`. [10]

If a variable is bound by value, then a copy of the variable value is taken at the point of the local function declaration. If a variable is bound by reference instead, the variable will refer to the value it has at the point of the local function call. Furthermore, it is the programmers' responsibility to ensure that variables bound by reference survive the existence scope of the local function otherwise the bound references will be invalid when the local function is called resulting in undefined behaviour (in other words, the usual care in using C++ references must be taken for variables bound by reference).

The type of a bound variable is automatically deduced using Boost.Typeof and it is the exact same type used to declare such a variable in the enclosing scope with the following notes:

[8] **Rationale.** By binding a variable in scope, the local function declaration is specifying that such a variable should be accessible within the local function body regardless of its type. Semantically, this binding should be seen as an "extension" of the scope of the bound variable from the enclosing scope to the scope of the local function body. Therefore, contrary to the semantic of passing a function parameter, the semantic of binding a variable does not depend on the variable type but just on the variable name: "The variable in scope named *x* should be accessible within the local function named *f*". For example, this reduces maintenance because if a bound variable type is changed, the local function declaration does not have to change.

[9] Obviously, the token `bind` is not a keyword of the C++ language. This library parses the token `bind` during macro expansion using preprocessor meta-programming (see the Implementation section). Therefore, `bind` can be considered a new "keyword" only at the preprocessor meta-programming level within the syntax defined by the macros of this library (thus it is referred to as a "keyword" only within quotes).

[10] **Rationale.** The library macros could have been implemented to accept both syntaxes `const bind ...` and `bind const ...` equivalently. However, handling both syntaxes would have complicated the macro implementation without adding any feature so only one syntax `const bind ...` is supported.

- If a bound variable was declared constant in the enclosing scope, it will always be bound by constant value or constant reference even if `bind...` is used instead of `const bind...`. However, if a bound variable was not declared constant in the enclosing scope then it will not be bound as constant unless constant binding is forced using `const bind...` (Note that binding by constant reference is not supported by C++11 lambda functions but it is supported by this library.) [11]

- If a bound variable was declared as a reference in the enclosing scope, it will still be bound by value unless it is explicitly bound by reference using `bind&` or `const bind&`. [12]

When a variable is bound by value (constant or not), its type must be `CopyConstructible` (i.e., its must provide a copy constructor). As with passing parameters to usual C++ functions, programmers might want to bind variables of complex types by (possibly constant) reference instead of by value to avoid expensive copy operations when these variables are bound to a local function.

For example, let's program the local function `add` from the example in the Introduction section. We bind the local variable `factor` by constant value (because its value should not be modified by the local function), the local variable `sum` by non-constant reference (because its value needs to be updated with the summation result), and program the body to perform the summation (see also `add.cpp`):

```
int main(void) {                                // Some local scope.
    int sum = 0, factor = 10;                   // Variables in scope to bind.

    void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                     // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      // Assert final summation value.
    return boost::report_errors();
}
```

Binding the Object `this`

It is also possible to bind the object `this` when it is in scope (e.g., from an enclosing non-static member function). This is done by using the special symbol `this_` (instead of `this`) as the name of the variable to bind in the local function declaration and also to access the object within the local function body. [13]

[11] An historical note: Constant binding of variables in scope was the main use case that originally motivated the authors in developing this library. The authors needed to locally create a chuck of code to assert some correctness conditions while these assertions were not supposed to modify any of the variables they were using (see the Contract++ library). This was achieved by binding by constant reference `const bind&` the variables needed by the assertions and then by programming the local function body to check the assertions. This way if any of the assertions mistakenly changes a bound variable (for example confusing the operator `==` with `=`), the compiler correctly generates an error because the bound variable is of `const` type within the local function body (see also *constant blocks* in the Examples section).

[12] **Rationale.** Variables originally declared as references are bound by value unless `[const] bind&` is used so that references can be bound by both value `[const] bind` and reference `[const] bind&` (this is the same binding semantic adopted by Boost.ScopeExit). However, variables originally declared as constants should never loose their `const` qualifier (to prevent their modification not just in the enclosing scope but also in the local scope) thus they are always bound by constant even if `bind[&]` is used instead of `const bind[&]`.

[13] **Rationale.** The special name `this_` was chosen following Boost practise to postfix with an underscore identifiers that are named after keywords (the C++ keyword `this` in this case). The special symbol `this_` is needed because `this` is a reserved C++ keyword so it cannot be used as the name of the internal parameter that passes the bound object to the local function body. It would have been possible to use `this` (instead of `this_`) within the local function body either at the expenses of copying the bound object (which would introduce run-time overhead and also the stringent requirement that the bound object must have a deep copy constructor) or by relying on an undefined behaviour of `static_cast` (which might not work on all platforms at the cost of portability).

Warning

The library will generate a compile-time error if `this` is mistakenly used instead of `this_` to bind the object in the local function declaration. However, mistakenly using `this` instead of `this_` to access the object within the local function body will leads to undefined behaviour and it will not necessarily generate a compile-time error. [14] Programmers are ultimately responsible to make sure that `this` is never used within a local function.

The object `this` can be bound by value:

```
bind this_ // Bind the object `this` by value.
```

In this case the local function will be able to modify the object when the enclosing scope is not a constant member and it will not be able to modify the object when the enclosing scope is a constant member. Otherwise, the object `this` can be bound by constant value:

```
const bind this_ // Bind the object `this` by constant value.
```

In this case the local function will never be able to modify the object (regardless of whether the enclosing scope is a constant member or not).

Note that the object `this` can never be bound by reference because C++ does not allow to obtain a reference to `this` (the library will generate a compile-time error if programmers try to use `bind& this_` or `const bind& this_`). Note that `this` is a pointer so the pointed object is never copied even if `this` is bound by value (also it is not possible to directly bind `*this` because `*this` is an expression and not a variable name).

For example, let's program a local function `add` similar to the one in the example from the Introduction section but using a member function to illustrate how to bind the object `this` (see also `add_this.cpp`):

```
struct adder {
    adder() : sum_(0) {}

    int sum(const std::vector<int>& nums, const int factor = 10) {

        void BOOST_LOCAL_FUNCTION(const bind factor, bind this_, int num) {
            this_->sum_ += factor * num; // Use `this_` instead of `this`.
        } BOOST_LOCAL_FUNCTION_NAME(add)

        std::for_each(nums.begin(), nums.end(), add);
        return sum_;
    }

private:
    int sum_;
};
```

Note that the local function has access to all class members via the bound object `this_` regardless of their access level (`public`, `protected`, or `private`). [15] Specifically, in the example above the local function updates the private data member `sum_`.

[14] **Rationale.** The local function body cannot be a static member function of the local functor object in order to support recursion (because the local function name is specified by the `BOOST_LOCAL_FUNCTION_NAME` macro only after the body so it must be made available via a functor data member named after the local function and local classes cannot have static data members in C++) and nesting (because the argument binding variable must be declared as a data member so it is visible in a local function nested within the body member function) -- see the Implementation section. Therefore, from within the local function body the variable `this` is visible but it refers to the local functor and not to the bound object.

[15] **Rationale.** This is possible because of the fix to C++ defect 45 that made inner and local types able to access all outer class members regardless of their access level.

Templates

When local functions are programmed within templates, they need to be declared using the special macros `BOOST_LOCAL_FUNCTION_TPL` and `BOOST_LOCAL_FUNCTION_NAME_TPL`: [16]

```
#include <boost/local_function.hpp> // This library header.

...
{ // Some declarative context within a template.
    ...
    result-type BOOST_LOCAL_FUNCTION_TPL(parameters) {
        body-code
    } BOOST_LOCAL_FUNCTION_NAME_TPL(name)
    ...
}
```

The `BOOST_LOCAL_FUNCTION_TPL` and `BOOST_LOCAL_FUNCTION_NAME_TPL` macros have the exact same syntax of the `BOOST_LOCAL_FUNCTION` and `BOOST_LOCAL_FUNCTION_NAME` macros that we have seen so far.

For example, let's program a local function similar to the one from the Introduction section but within a template (see also `add_template.cpp`):

```
template<typename T>
T total(const T& x, const T& y, const T& z) {
    T sum = T(), factor = 10;

    // Must use the `..._TPL` macros within templates.
    T BOOST_LOCAL_FUNCTION_TPL(const bind factor, bind& sum, T num) {
        return sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME_TPL(add)

    add(x);
    T nums[2]; nums[0] = y; nums[1] = z;
    std::for_each(nums, nums + 2, add);

    return sum;
}
```

[16] **Rationale.** Within templates, this library needs to use `typename` to explicitly indicate that some expressions evaluate to a type. Because C++03 does not allow to use `typename` outside templates, the special `..._TPL` macros are used to indicate that the enclosing scope is a template so this library can safely use `typename` to resolve expression type ambiguities. C++11 and other compilers might compile local functions within templates even when the `..._TPL` macros are not used. However, it is recommended to always use the `..._TPL` macros within templates to maximize portability.

Advanced Topics

This section illustrates advanced usage of this library. At the bottom there is also a list of known limitations of this library.

Default Parameters

This library allows to specify default values for the local function parameters. However, the usual C++ syntax for default parameters that uses the assignment symbol = cannot be used. [17] The keyword `default` is used instead:

```
parameter-type parameter-name, default parameter-default-value, ...
```

For example, let's program a local function `add(x, y)` where the second parameter `y` is optional and has a default value of `2` (see also `add_default.cpp`):

```
int BOOST_LOCAL_FUNCTION(int x, int y, default 2) { // Default parameter.
    return x + y;
} BOOST_LOCAL_FUNCTION_NAME(add)

BOOST_TEST(add(1) == 3);
```

Programmers can define a `WITH_DEFAULT` macro similar to the following if they think it improves readability over the above syntax (see also `add_with_default.cpp`): [18]

```
#define WITH_DEFAULT , default
```

```
int BOOST_LOCAL_FUNCTION(int x, int y WITH_DEFAULT 2) { // Default.
    return x + y;
} BOOST_LOCAL_FUNCTION_NAME(add)

BOOST_TEST(add(1) == 3);
```

Commas and Symbols in Macros

The C++ preprocessor does not allow commas `,` within macro parameters unless they are wrapped by round parenthesis `()` (see the Boost.Utility/IdentityType documentation for details). Therefore, using commas within local function parameters and bindings will generate (cryptic) preprocessor errors unless they are wrapped with an extra set of round parenthesis `()` as explained here.

Note

Also macro parameters with commas wrapped by angular parenthesis <> (templates, etc) or square parenthesis `[]` (multidimensional array access, etc) need to be wrapped by the extra round parenthesis `()` as explained here (this is because the preprocessor only recognizes the round parenthesis and it does not recognize angular, square, or any other type of parenthesis). However, macro parameters with commas which are already wrapped by round parenthesis `()` are fine (function calls, some value expressions, etc).

[17] **Rationale.** The assignment symbol `=` cannot be used to specify default parameter values because default values are not part of the parameter type so they cannot be handled using template meta-programming. Default parameter values need to be separated from the rest of the parameter declaration using the preprocessor. Specifically, this library needs to use preprocessor meta-programming to remove default values when constructing the local function type and also to count the number of default values to provide the correct set of call operators for the local functor. Therefore, the symbol `=` cannot be used because it cannot be handled by preprocessor meta-programming (non-alphanumeric symbols cannot be detected by preprocessor meta-programming because they cannot be concatenated by the preprocessor).

[18] The authors do not personally find the use of the `WITH_DEFAULT` macro more readable and they prefer to use the `default` keyword directly. Furthermore, `WITH_DEFAULT` needs to be defined differently for compilers without variadic macros `#define WITH_DEFAULT (default)` so it can only be defined by programmers based on the syntax they decide to use (see the No Variadic Macros section).

In addition, local function parameter types cannot start with non-alphanumeric symbols (alphanumeric symbols are `A-Z`, `a-z`, and `0-9`). [19] The library will generate (cryptic) preprocessor errors if a parameter type starts with a non-alphanumeric symbol.

Let's consider the following example:

```
void BOOST_LOCAL_FUNCTION(
    const std::map<std::string, size_t>& m,                 // (1) Error.
    ::sign_t sign,                                          // (2) Error.
    const size_t& factor,
            default key_sizeof<std::string, size_t>::value, // (3) Error.
    const std::string& separator, default cat(":", " ")     // (4) OK.
) {
    ...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

(1) The parameter type `const std::map<std::string, size_t>&` contains a comma `,` after the first template parameter `std::string`. This comma is not wrapped by any round parenthesis `()` thus it will cause a preprocessor error. [20] The Boost.Utility/IdentityType macro `BOOST_IDENTITY_TYPE((`*`type-with-commas`*`))` defined in the `boost/utility/identity_type.hpp` header can be used to wrap a type within extra parenthesis `()` so to overcome this problem:

```
#include <boost/utility/identity_type.hpp>

void BOOST_LOCAL_FUNCTION(
    BOOST_IDENTITY_TYPE((const std::map<std::string, size_t>&)) m, // OK.
    ...
) {
    ...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

This macro expands to an expression that evaluates (at compile-time) exactly to the specified type (furthermore, this macro does not use variadic macros so it works on any C++03 compiler). Note that a total of two set of parenthesis `()` are needed: The parenthesis to invoke the `BOOST_IDENTITY_TYPE(...)` macro plus the parenthesis to wrap the type expression (and therefore any comma `,` that it contains) passed as parameter to the `BOOST_IDENTITY_TYPE((...))` macro. Finally, the `BOOST_IDENTITY_TYPE` macro must be prefixed by the `typename` keyword `typename BOOST_IDENTITY_TYPE(`*`parenthesized-type`*`)` when used together with the `BOOST_LOCAL_FUNCTION_TPL` macro within templates.

Note

Often, there might be better ways to overcome this limitation that lead to code which is more readable than the one using the `BOOST_IDENTITY_TYPE` macro.

For example, in this case a `typedef` from the enclosing scope could have been used to obtain the following valid and perhaps more readable code:

```
typedef std::map<std::string, size_t> map_type;
void BOOST_LOCAL_FUNCTION(
    const map_type& m, // OK (and more readable).
    ...
) BOOST_LOCAL_FUNCTION_NAME(f)
```

(2) The parameter type `::sign_t` starts with the non-alphanumeric symbols `::` thus it will generate preprocessor errors if used as a local function parameter type. The `BOOST_IDENTITY_TYPE` macro can also be used to overcome this issue:

[19] **Rationale.** This limitation is because this library uses preprocessor token concatenation `##` to inspect the macro parameters (to distinguish between function parameters, bound variables, etc) and the C++ preprocessor does not allow to concatenate non-alphanumeric tokens.

[20] The preprocessor always interprets unwrapped commas as separating macro parameters. Thus in this case the comma will indicate to the preprocessor that the first macro parameter is `const std::map<std::string`, the second macro parameter is `size_t>& m`, etc instead of passing `const std::map<std::string, size_t>& m` as a single macro parameter.

```
void BOOST_LOCAL_FUNCTION(
    ...
    BOOST_IDENTITY_TYPE((::sign_t)) sign, // OK.
    ...
) {
    ...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

Note

Often, there might be better ways to overcome this limitation that lead to code which is more readable than the one using the `BOOST_IDENTITY_TYPE` macro.

For example, in this case the symbols `::` could have been simply dropped to obtain the following valid and perhaps more readable code:

```
void BOOST_LOCAL_FUNCTION(
    ...
    sign_t sign, // OK (and more readable).
    ...
) {
    ...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

(3) The default parameter value `key_sizeof<std::string, size_t>::value` contains a comma `,` after the first template parameter `std::string`. Again, this comma is not wrapped by any parenthesis `()` so it will cause a preprocessor error. Because this is a value expression (and not a type expression), it can simply be wrapped within an extra set of round parenthesis `()`:

```
void BOOST_LOCAL_FUNCTION(
    ...
    const size_t& factor,
            default (key_sizeof<std::string, size_t>::value), // OK.
    ...
) {
    ...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

(4) The default parameter value `cat(":", " ")` is instead fine because it contains a comma `,` which is already wrapped by the parenthesis `()` of the function call `cat(...)`.

Consider the following complete example (see also `macro_commas.cpp`):

```
void BOOST_LOCAL_FUNCTION(
    BOOST_IDENTITY_TYPE((const std::map<std::string, size_t>&)) m,
    BOOST_IDENTITY_TYPE((::sign_t)) sign,
    const size_t& factor,
            default (key_sizeof<std::string, size_t>::value),
    const std::string& separator, default cat(":", " ")
) {
    // Do something...
} BOOST_LOCAL_FUNCTION_NAME(f)
```

Assignments and Returns

Local functions are function objects so it is possible to assign them to other functors like Boost.Function's `boost::function` in order to store the local function into a variable, pass it as a parameter to another function, or return it from the enclosing function.

For example (see also `return_assign.cpp`):

```
void call1(boost::function<int (int) > f) { BOOST_TEST(f(1) == 5); }
void call0(boost::function<int (void)> f) { BOOST_TEST(f() == 5); }

boost::function<int (int, int)> linear(const int& slope) {
    int BOOST_LOCAL_FUNCTION(const bind& slope,
            int x, default 1, int y, default 2) {
        return x + slope * y;
    } BOOST_LOCAL_FUNCTION_NAME(lin)

    boost::function<int (int, int)> f = lin; // Assign to local variable.
    BOOST_TEST(f(1, 2) == 5);

    call1(lin); // Pass to other functions.
    call0(lin);

    return lin; // Return.
}

void call(void) {
    boost::function<int (int, int)> f = linear(2);
    BOOST_TEST(f(1, 2) == 5);
}
```

Warning

As with C++11 lambda functions, programmers are responsible to ensure that bound variables are valid in any scope where the local function object is called. Returning and calling a local function outside its declaration scope will lead to undefined behaviour if any of the bound variable is no longer valid in the scope where the local function is called (see the Examples section for more examples on the extra care needed when returning a local function as a closure). It is always safe instead to call a local function within its enclosing scope.

In addition, a local function can bind and call other local functions. Local functions should always be bound by constant reference `const bind&` to avoid unnecessary copies. For example, the following local function `inc_sum` binds the local function `inc` so `inc_sum` can call `inc` (see aslo `transform.cpp`):

```
int offset = 5;
std::vector<int> v;
std::vector<int> w;

for(int i = 1; i <= 2; ++i) v.push_back(i * 10);
BOOST_TEST(v[0] == 10); BOOST_TEST(v[1] == 20);
w.resize(v.size());

int BOOST_LOCAL_FUNCTION(const bind& offset, int i) {
    return ++i + offset;
} BOOST_LOCAL_FUNCTION_NAME(inc)

std::transform(v.begin(), v.end(), w.begin(), inc);
BOOST_TEST(w[0] == 16); BOOST_TEST(w[1] == 26);

int BOOST_LOCAL_FUNCTION(bind& inc, int i, int j) {
    return inc(i + j); // Call the other bound local function.
} BOOST_LOCAL_FUNCTION_NAME(inc_sum)

offset = 0;
std::transform(v.begin(), v.end(), w.begin(), v.begin(), inc_sum);
BOOST_TEST(v[0] == 27); BOOST_TEST(v[1] == 47);
```

Nesting

It is possible to nest local functions into one another. For example (see also `nesting.cpp`):

```
int x = 0;

void BOOST_LOCAL_FUNCTION(bind& x) {
    void BOOST_LOCAL_FUNCTION(bind& x) { // Nested.
        x++;
    } BOOST_LOCAL_FUNCTION_NAME(g)

    x--;
    g(); // Nested local function call.
} BOOST_LOCAL_FUNCTION_NAME(f)

f();
```

Accessing Types (concepts, etc)

This library never requires to explicitly specify the type of bound variables (e.g., this reduces maintenance because the local function declaration and definition do not have to change even if the bound variable types change as long as the semantics of the local function remain valid). From within local functions, programmers can access the type of a bound variable using the following macro:

```
BOOST_LOCAL_FUNCTION_TYPEOF(bound-variable-name)
```

The `BOOST_LOCAL_FUNCTION_TYPEOF` macro expands to a type expression that evaluates (at compile-time) to the fully qualified type of the bound variable with the specified name. This type expression is fully qualified in the sense that it will be constant if the variable is bound by constant `const bind[&]` and it will also be a reference if the variable is bound by reference `[const] bind&` (if needed, programmers can remove the `const` and `&` qualifiers using `boost::remove_const` and `boost::remove_reference`, see Boost.TypeTraits).

The deduced bound type can be used within the body to check concepts, declare local variables, etc. For example (see also `typeof.cpp` and `addable.hpp`):

```
int sum = 0, factor = 10;

void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
    // Type-of for concept checking.
    BOOST_CONCEPT_ASSERT((Addable<boost::remove_reference<
            BOOST_LOCAL_FUNCTION_TYPEOF(sum)>::type>));
    // Type-of for declarations.
    boost::remove_reference<BOOST_LOCAL_FUNCTION_TYPEOF(
            factor)>::type mult = factor * num;
    sum += mult;
} BOOST_LOCAL_FUNCTION_NAME(add)

add(6);
```

Within templates, `BOOST_LOCAL_FUNCTION_TYPEOF` should not be prefixed by the `typename` keyword but eventual type manipulations need the `typename` prefix as usual (see also `typeof_template.cpp` and `addable.hpp`):

```
template<typename T>
T calculate(const T& factor) {
    T sum = 0;

    void BOOST_LOCAL_FUNCTION_TPL(const bind factor, bind& sum, T num) {
        // Local function `TYPEOF` does not need `typename`.
        BOOST_CONCEPT_ASSERT((Addable<typename boost::remove_reference<
                BOOST_LOCAL_FUNCTION_TYPEOF(sum)>::type>));
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME_TPL(add)

    add(6);
    return sum;
}
```

In this context, it is best to use the `BOOST_LOCAL_FUNCTION_TYPEOF` macro instead of using Boost.Typeof to reduce the number of times that Boost.Typeof is invoked (either the library already internally used Boost.Typeof once, in which case using this macro will not use Boost.Typeof again, or the bound variable type is explicitly specified by programmers as shown be below, in which case using this macro will not use Boost.Typeof at all).

Furthermore, within the local function body it possible to access the result type using `result_type`, the type of the first parameter using `arg1_type`, the type of the second parameter using `arg2_type`, etc. [21]

Specifying Types (no Boost.Typeof)

While not required, it is possible to explicitly specify the type of bound variables so the library will not internally use Boost.Typeof to automatically deduce the types. When specified, the bound variable type must follow the `bind` "keyword" and it must be wrapped within round parenthesis `()`:

```
bind(variable-type) variable-name           // Bind by value with explicit type.
bind(variable-type)& variable-name          // Bind by reference with explicit type.
const bind(variable-type) variable-name     // Bind by constant value with explicit type.
const bind(variable-type)& variable-name    // Bind by constant reference with explicit type.
bind(class-type*) this_                     // Bind object `this` with explicit type.
const bind(class-type*) this_              // Bind object `this` by constant with explicit type.
```

Note that within the local function body it is always possible to abstract the access to the type of a bound variable using `BOOST_LOCAL_FUNCTION_TYPEOF` (even when the bound variable type is explicitly specified in the local function declaration).

The library also uses Boost.Typeof to determine the local function result type (because this type is specified outside the `BOOST_LOCAL_FUNCTION` macro). Thus it is also possible to specify the local function result type as one of the `BOOST_LOCAL_FUNCTION` macro parameters prefixing it by `return` so the library will not use Boost.Typeof to deduce the result type:

```
BOOST_LOCAL_FUNCTION_TYPE(return result-type, ...)
```

Note that the result type must be specified only once either before the macro (without the `return` prefix) or as one of the macro parameters (with the `return` prefix). As always, the result type can be `void` to declare a function that returns nothing (so `return void` is allowed when the result type is specified as one of the macro parameters).

The following example specifies all bound variables and result types (see also `add_typed.cpp`): [22]

[21] **Rationale.** The type names `result_type` and `arg`*N*`_type` follow the Boost.TypeTraits naming conventions for function traits.

[22] In the examples of this documentation, bound variables, function parameters, and the result type are specified in this order because this is the order used by C++11 lambda functions. However, the library accepts bound variables, function parameters, and the result type in any order.

```
struct adder {
    adder(void) : sum_(0) {}

    int sum(const std::vector<int>& nums, const int& factor = 10) {
        // Explicitly specify bound variable and return types (no type-of).
        BOOST_LOCAL_FUNCTION(const bind(const int&) factor,
                bind(adder*) this_, int num, return int) {
            return this_->sum_ += factor * num;
        } BOOST_LOCAL_FUNCTION_NAME(add)

        std::for_each(nums.begin(), nums.end(), add);
        return sum_;
    }

private:
    int sum_;
};
```

Unless necessary, it is recommended to not specify the bound variable and result types. Let the library deduce these types so the local function syntax will be more concise and the local function declaration will not have to change if a bound variable type changes (reducing maintenance).

Note

When all bound variable and result types are explicitly specified, the library implementation will not use Boost.Typeof.

Inlining

Local functions can be declared inline to increase the chances that the compiler will be able to reduce the run-time of the local function call by inlining the generated assembly code. A local function is declared inline by prefixing its name with the keyword `inline`:

```
result-type BOOST_LOCAL_FUNCTION(parameters) {
    ... // Body.
} BOOST_LOCAL_FUNCTION_NAME(inline name) // Inlining.
```

When inlining a local function, note the following:

- On C++03 compliant compilers, inline local functions always have a run-time comparable to their equivalent implementation that uses local functors (see the Alternatives section). However, inline local functions have the important limitation that they cannot be assigned to other functors (like `boost::function`) and they cannot be passed as template parameters.
- On C++11 compilers, `inline` has no effect because this library will automatically generate code that uses C++11 specific features to inline the local function calls whenever possible even if the local function is not declared inline. Furthermore, non C++11 local functions can always be passes as template parameters even when they are declared inline. [23]

[23] **Rationale.** This library uses an indirect function call via a function pointer in order to pass the local function as a template parameter (see the Implementation section). No compiler has yet been observed to be able to inline function calls when they use such indirect function pointer calls. Therefore, inline local functions do not use such indirect function pointer call (so they are more likely to be optimized) but because of that they cannot be passed as template parameters. The indirect function pointer call is needed on C++03 but it is not needed on C++11 (see [N2657] and Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`) thus this library automatically generates local function calls that can be inline on C++11 compilers (even when the local function is not declared inline).

Important

It is recommended to not declare a local function inline unless it is strictly necessary for optimizing pure C++03 compliant code (because in all other cases this library will automatically take advantage of C++11 features to optimize the local function calls while always allowing to pass the local function as a template parameter).

For example, the following local function is declared inline (thus a for-loop needs to be used for portability instead of passing the local function as a template parameter to the `std::for_each` algorithm, see also `add_inline.cpp`):

```
int sum = 0, factor = 10;

void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
    sum += factor * num;
} BOOST_LOCAL_FUNCTION_NAME(inline add) // Inlining.

std::vector<int> v(100);
std::fill(v.begin(), v.end(), 1);

for(size_t i = 0; i < v.size(); ++i) add(v[i]); // Cannot use for_each.
```

Recursion

Local functions can be declared recursive so a local function can recursively call itself from its body (as usual with C++ functions). A local function is declared recursive by prefixing its name with the `recursive` "keyword" (thus `recursive` cannot be used as a local function name):

```
result-type BOOST_LOCAL_FUNCTION(parameters) {
    ... // Body.
} BOOST_LOCAL_FUNCTION_NAME(recursive name) // Recursive.
```

For example, the following local function is used to recursively calculate the factorials of all the numbers in the specified vector (see also `factorial.cpp`):

```
struct calculator {
    std::vector<int> results;

    void factorials(const std::vector<int>& nums) {
        int BOOST_LOCAL_FUNCTION(bind this_, int num,
                bool recursion, default false) {
            int result = 0;

            if(num <= 0) result = 1;
            else result = num * factorial(num - 1, true); // Recursive call.

            if(!recursion) this_->results.push_back(result);
            return result;
        } BOOST_LOCAL_FUNCTION_NAME(recursive factorial) // Recursive.

        std::for_each(nums.begin(), nums.end(), factorial);
    }
};
```

Compilers have not been observed to be able to inline recursive local function calls not even when the recursive local function is also declared inline:

```
... BOOST_LOCAL_FUNCTION_NAME(inline recursive factorial)
```

Recursive local functions should never be called outside their declaration scope. [24]

Warning

If a local function is returned from the enclosing function and called in a different scope, the behaviour is undefined (and it will likely result in a run-time error).

This is not a limitation with respect to C++11 lambda functions because lambdas can never call themselves recursively (in other words, there is no recursive lambda function that can successfully be called outside its declaration scope because there is no recursive lambda function at all).

Overloading

Because local functions are functors, it is possible to overload them using the `boost::overloaded_function` functor of Boost.Functional/OverloadedFunction from the `boost/functional/overloaded_function.hpp` header (see the Boost.Functional/OverloadedFunction documentation for details).

In the following example, the overloaded function object `add` can be called with signatures from either the local function `add_s`, or the local function `add_d`, or the local function `add_d` with its extra default parameter, or the function pointer `add_i` (see also `overload.cpp`):

```
int add_i(int x, int y) { return x + y; }
```

```
std::string s = "abc";
std::string BOOST_LOCAL_FUNCTION(
        const bind& s, const std::string& x) {
    return s + x;
} BOOST_LOCAL_FUNCTION_NAME(add_s)

double d = 1.23;
double BOOST_LOCAL_FUNCTION(const bind d, double x, double y, default 0) {
    return d + x + y;
} BOOST_LOCAL_FUNCTION_NAME(add_d)

boost::overloaded_function<
      std::string (const std::string&)
    , double (double)
    , double (double, double) // Overload giving default param.
    , int (int, int)
> add(add_s, add_d, add_d, add_i); // Overloaded function object.

BOOST_TEST(add("xyz") == "abcxyz"); // Call `add_s`.
BOOST_TEST((4.44 - add(3.21)) <= 0.001); // Call `add_d` (no default).
BOOST_TEST((44.44 - add(3.21, 40.0)) <= 0.001); // Call `add_d`.
BOOST_TEST(add(1, 2) == 3); // Call `add_i`.
```

Exception Specifications

It is possible to program exception specifications for local functions by specifying them after the `BOOST_LOCAL_FUNCTION` macro and before the body code block `{ ... }`.

[24] **Rationale.** This limitation comes from the fact that the global functor used to pass the local function as a template parameter (and eventually returned outside the declarations scope) does not know the local function name so the local function name used for recursive call cannot be set in the global functor. This limitation together with preventing the possibility for inlining are the reasons why local functions are not recursive unless programmers explicitly declare them `recursive`.

Important

Note that the exception specifications only apply to the body code specified by programmers and they do not apply to the rest of the code automatically generated by the macro expansions to implement local functions. For example, even if the body code is specified to throw no exception using `throw () { ... }`, the execution of the library code automatically generated by the macros could still throw (if there is no memory, etc).

For example (see also `add_except.cpp`):

```
double sum = 0.0;
int factor = 10;

void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum,
        double num) throw() { // Throw nothing.
    sum += factor * num;
} BOOST_LOCAL_FUNCTION_NAME(add)

add(100);
```

Storage Classifiers

Local function parameters support the storage classifiers as usual in C++03. The `auto` storage classifier is specified as: [25]

```
auto parameter-type parameter-name
```

The `register` storage classifier is specified as:

```
register parameter-type parameter-name
```

For example (see also `add_classifiers.cpp`):

```
int BOOST_LOCAL_FUNCTION(auto int x, register int y) { // Classifiers.
    return x + y;
} BOOST_LOCAL_FUNCTION_NAME(add)
```

Same Line Expansions

In general, it is not possible to expand the `BOOST_LOCAL_FUNCTION`, `BOOST_LOCAL_FUNCTION_TPL` macros multiple times on the same line. [26]

Therefore, this library provides additional macros `BOOST_LOCAL_FUNCTION_ID` and `BOOST_LOCAL_FUNCTION_ID_TPL` which can be expanded multiple times on the same line as long as programmers specify unique identifiers as the macros' first parameters. The unique identifier can be any token (not just numeric) that can be successfully concatenated by the preprocessor (e.g., `local_function_number_1_at_line_123`). [27]

[25] The `auto` storage classifier is part of the C++03 standard and therefore supported by this library. However, the meaning and usage of the `auto` keyword changed in C++11. Therefore, use the `auto` storage classifier with the usual care in order to avoid writing C++03 code that might not work on C++11.

[26] **Rationale.** The `BOOST_LOCAL_FUNCTION` and `BOOST_LOCAL_FUNCTION_TPL` macros internally use `__LINE__` to generate unique identifiers. Therefore, if these macros are expanded more than on time on the same line, the generated identifiers will no longer be unique and the code will not compile. (This restriction does not apply to MSVC and other compilers that provide the non-standard `__COUNTER__` macro.) Note that the `BOOST_LOCAL_FUNCTION_NAME` macro can always be expanded multiple times on the same line because the unique local function name (and not `__LINE__`) is used by this macro to generate unique identifiers (so there is no need for a `BOOST_LOCAL_FUNCTION_NAME_ID` macro).

[27] Because there are restrictions on the set of tokens that the preprocessor can concatenate and because not all compilers correctly implement these restrictions, it is in general recommended to specify unique identifiers as a combination of alphanumeric tokens.

The BOOST_LOCAL_FUNCTION_ID and BOOST_LOCAL_FUNCTION_ID_TPL macros accept local function parameter declaration lists using the exact same syntax as BOOST_LOCAL_FUNCTION. For example (see also same_line.cpp):

```
#define LOCAL_INC_DEC(offset) \
    int BOOST_LOCAL_FUNCTION_ID(BOOST_PP_CAT(inc, __LINE__), /* unique ID */ \
            const bind offset, const int x) { \
        return x + offset; \
    } BOOST_LOCAL_FUNCTION_NAME(inc) \
    \
    int BOOST_LOCAL_FUNCTION_ID(BOOST_PP_CAT(dec, __LINE__), \
            const bind offset, const int x) { \
        return x - offset; \
    } BOOST_LOCAL_FUNCTION_NAME(dec)

#define LOCAL_INC_DEC_TPL(offset) \
    T BOOST_LOCAL_FUNCTION_ID_TPL(BOOST_PP_CAT(inc, __LINE__), \
            const bind offset, const T x) { \
        return x + offset; \
    } BOOST_LOCAL_FUNCTION_NAME_TPL(inc) \
    \
    T BOOST_LOCAL_FUNCTION_ID_TPL(BOOST_PP_CAT(dec, __LINE__), \
            const bind offset, const T x) { \
        return x - offset; \
    } BOOST_LOCAL_FUNCTION_NAME_TPL(dec)

template<typename T>
void f(T& delta) {
    LOCAL_INC_DEC_TPL(delta) // Multiple local functions on same line.
    BOOST_TEST(dec(inc(123)) == 123);
}

int main(void) {
    int delta = 10;
    LOCAL_INC_DEC(delta) // Multiple local functions on same line.
    BOOST_TEST(dec(inc(123)) == 123);
    f(delta);
    return boost::report_errors();
}
```

As shown by the example above, the BOOST_LOCAL_FUNCTION_ID and BOOST_LOCAL_FUNCTION_ID_TPL macros are especially useful when it is necessary to invoke them multiple times within a user-defined macro (because the preprocessor expands all nested macros on the same line).

Limitations (operators, etc)

The following table summarizes all C++ function features indicating those features that are not supported by this library for local functions.

C++ Function Feature	Local Function Support	Comment
`export`	No.	This is not supported because local functions cannot be templates (plus most C++ compilers do not implement `export` at all).
`template<`*`template-parameter-list`*`>`	No.	This is not supported because local functions are implemented using local classes and C++03 local classes cannot be templates.
`explicit`	No.	This is not supported because local functions are not constructors.
`inline`	Yes.	Local functions can be specified `inline` to improve the chances that C++03 compilers can optimize the local function call run-time (but `inline` local functions cannot be passed as template parameters on C++03 compilers, see the Advanced Topics section).
`extern`	No.	This is not supported because local functions are always defined locally within the enclosing scope and together with their declarations.
`static`	No.	This is not supported because local functions are not member functions.
`virtual`	No.	This is not supported because local functions are not member functions. [a]
`result-type`	Yes.	This is supported (see the Tutorial section).
`function-name`	Yes.	Local functions are named and they can call themselves recursively but they cannot be operators (see the Tutorial and Advanced Topics sections).
`parameter-list`	Yes.	This is supported and it also supports the `auto` and `register` storage classifiers, default parameters, and binding of variables in scope (see the Tutorial and Advanced Topics sections).
Trailing `const` qualifier	No.	This is not supported because local functions are not member functions.
Trailing `volatile` qualifier	No.	This is not supported because local functions are not member functions.

[a] **Rationale.** It would be possible to make a local function class inherit from another local function class. However, this "inheritance" feature is not implemented because it seemed of no use given that local functions can be bound to one another thus they can simply call each other directly without recurring to dynamic binding or base function calls.

Operators

Local functions cannot be operators. Naming a local function `operator...` will generate a compile-time error. [28]

For example, the following code does not compile (see also `operator_error.cpp`):

```
bool BOOST_LOCAL_FUNCTION(const point& p, const point& q) {
    return p.x == q.x && p.y == q.y;
} BOOST_LOCAL_FUNCTION_NAME(operator==) // Error: Cannot use `operator...`.
```

Goto

It is possible to jump with a `goto` within the local function body. For example, the following compiles (see also `goto.cpp`):

```
int error(int x, int y) {
    int BOOST_LOCAL_FUNCTION(int z) {
        if(z > 0) goto success; // OK: Can jump within local function.
        return -1;
    success:
        return 0;
    } BOOST_LOCAL_FUNCTION_NAME(validate)

    return validate(x + y);
}
```

However, it is not possible to jump with a `goto` from within the local function body to to a label defined in the enclosing scope. For example, the following does not compile (see also `goto_error.cpp`):

```
int error(int x, int y) {
    int BOOST_LOCAL_FUNCTION(int z) {
        if(z <= 0) goto failure;    // Error: Cannot jump to enclosing scope.
        else goto success;          // OK: Can jump within local function.
    success:
        return 0;
    } BOOST_LOCAL_FUNCTION_NAME(validate)

    return validate(x + y);
failure:
    return -1;
}
```

[28] **Rationale.** This is the because a local function name must be a valid local variable name (the local variable used to hold the local functor) and operators cannot be used as local variable names.

Examples

This section lists some examples that use this library.

GCC Lambdas (without C++11)

Combing local functions with the non-standard statement expression extension of the GCC compiler, it is possible to implement lambda functions for GCC compilers even without C++11 support.

Warning

This code only works on compilers that support GCC statement expression extension or that support C++11 lambda functions.

For example (see also `gcc_lambda.cpp` and `gcc_cxx11_lambda.cpp`):

With Local Functions (GCC only)

```
int val = 2;
int nums[] = {1, 2, 3};
int* end = nums + 3;

int* iter = std::find_if(nums, end,
    GCC_LAMBDA(const bind val, int num, re↵
turn bool) {
        return num == val;
    } GCC_LAMBDA_END
);
```

C++11 Lambdas

```
int val = 2;
int nums[] = {1, 2, 3};
int* end = nums + 3;

int* iter = std::find_if(nums, end,
    [val](int num) -> bool {
        return num == val;
    }
);
```

The GCC lambda function macros are implemented using local functions (see also `gcc_lambda.hpp`):

```
#   define GCC_LAMBDA_(binds, params, results) \
        ({ /* open statement expression (GCC extension only) */ \
        BOOST_LOCAL_FUNCTION( \
            BOOST_PP_LIST_ENUM(BOOST_PP_LIST_APPEND(binds, \
                BOOST_PP_LIST_APPEND(params, \
                    BOOST_PP_IIF(BOOST_PP_LIST_IS_NIL(results), \
                        (return void, BOOST_PP_NIL) /* default for lambdas */ \
                    , \
                        results \
                    )\
                ) \
            )) \
        )
```

```
#define GCC_LAMBDA_END_(id) \
    BOOST_LOCAL_FUNCTION_NAME(BOOST_PP_CAT(gcc_lambda_, id)) \
    BOOST_PP_CAT(gcc_lambda_, id); \
    }) /* close statement expression (GCC extension only) */
```

This is possible because GCC statement expressions allow to use declaration statements within expressions and therefore to declare a local function within an expression. The macros automatically detect if the compiler supports C++11 lambda functions in which case the implementation uses native lambdas instead of local functions in GCC statement expressions. However, C++11 lambda functions do not support constant binding so it is best to only use `const bind variable` (same as `=variable` for C++11 lambda

functions) and `bind& variable` (same as `&variable` for C++11 lambda functions') because these have the exact same semantic between the local function and the native lambda implementations. Furthermore, local functions allow to bind data members directly while C++11 lambda functions require to access data members via binding the object `this`. Unfortunately, the short-hand binds `&` and `=` of C++11 lambda functions (which automatically bind all variables in scope either by reference or value) are not supported by these GCC lambda function macros because they are not supported by local functions. Finally, the result type `return` *`result-type`* is optional and it is assumed `void` when it is not specified (same as with C++11 lambda functions).

Constant Blocks

It is possible to use local functions to check assertions between variables that are made constant within the asserted expressions. This is advantageous because assertions are not supposed to change the state of the program and ideally the compiler will not compile assertions that modify variables.

For example, consider the following assertion where by mistake we programmed `operator=` instead of `operator==`:

```
int x = 1, y = 2;
assert(x = y); // Mistakenly `=` instead of `==`.
```

Ideally this code will not compile instead this example not only compiles but the assertion even passes the run-time check and no error is generated at all. The [N1613] paper introduces the concept of a *const-block* which could be used to wrap the assertion above and catch the programming error at compile-time. Similarly, the following code will generate a compile-time error when `operator=` is mistakenly used instead of `operator==` because both `x` and `y` are made constants (using local functions) within the block of code performing the assertion (see also `const_block_error.cpp`):

With Local Functions

```
int x = 1, y = 2;
CONST_BLOCK(x, y) { // Constant block.
    assert(x = y);  // Compiler error.
} CONST_BLOCK_END
```

N1613 Const-Blocks

```
int x = 1, y = 2;
const {             // Constant block.
    assert(x = y);  // Compiler error.
}
```

The constant block macros are implemented using local functions (see also `const_block.hpp`):

```
#define CONST_BLOCK_(variables) \
    void BOOST_LOCAL_FUNCTION( \
        BOOST_PP_IIF(BOOST_PP_LIST_IS_NIL(variables), \
            void BOOST_PP_TUPLE_EAT(3) \
        , \
            BOOST_PP_LIST_FOR_EACH_I \
        )(CONST_BLOCK_BIND_, ~, variables) \
    )
```

```
#define CONST_BLOCK_END_(id) \
    BOOST_LOCAL_FUNCTION_NAME(BOOST_PP_CAT(const_block_, id)) \
    BOOST_PP_CAT(const_block_, id)(); /* call local function immediately */
```

The constant block macros are implemented using a local function which binds by constant reference `const bind&` all the specified variables (so the variables are constant within the code block but they do not need to be `CopyConstructible` and no extra copy is performed). The local function executes the `assert` instruction in its body and it is called immediately after it is defined. More in general, constant blocks can be used to evaluate any instruction (not just assertions) within a block were all specified variables are constant.

Unfortunately, constant blocks cannot be implemented with C++11 lambda functions because these do not support constant binding. Variables bound by value using C++11 lambda functions (`variable`, `=variable`, and `=`) are constant but they are required to be

`CopyConstructible` and they introduce potentially expensive copy operations. [29] Of course it is always possible to introduce extra constant variables and bind these variables to the C++11 lambda functions but the constant block code will then have to manage the declaration and initialization of these extra variables plus it will have to use the extra variable names instead of the original variable names:

```
int x = 1, y = 2;
const decltype(x)& const_x = x; // Constant so cannot be modified
const decltype(y)& const_y = y; // and reference so no copy.
[&const_x, &const_y]() {        // Lambda functions (C++11 only).
    assert(const_x = const_y);  // Unfortunately, `const_` names.
}();
```

In many cases the use of an extra constant variable `const_x` can be acceptable but in other cases it might be preferable to maintain the same variable name `x` within the function body.

Scope Exits

Scope exits allow to execute arbitrary code at the exit of the enclosing scope and they are provided by the Boost.ScopeExit library.

For curiosity, here we show how to re-implement scope exits using local functions. One small advantage of scope exits that use local functions is that they support constant binding. Boost.ScopeExit does not directly support constant binding (however, it is always possible to introduce an extra `const` local variable, assign it to the value to bind, and then bind the `const` variable so to effectively have constant binding with Boost.ScopeExit as well).

Note

In general, the authors recommend to use Boost.ScopeExit instead of the code listed by this example whenever possible (because Boost.ScopeExit is a library deliberately designed to support the scope exit construct).

The following example binds `p` by constant reference so this variable cannot be modified within the scope exit body but it is not copied and it will present the value it has at the exit of the enclosing scope and not at the scope exit declaration (see also `scope_exit.cpp`):

With Local Functions

```
person& p = persons_.back();
person::evolution_t checkpoint = p.evolution_;

SCOPE_EXIT(const bind check↵
point, const bind& p, bind this_) {
   if (checkpoint == p.evolution_) this_->per↵
sons_.pop_back();
} SCOPE_EXIT_END
```

Boost.ScopeExit

```
person& p = persons_.back();
person::evolution_t checkpoint = p.evolution_;

BOOST_SCOPE_EXIT(checkpoint, &p, this_) { // ↵
Or extra variable `const_p`.
   if (checkpoint == p.evolution_) this_->per↵
sons_.pop_back();
} BOOST_SCOPE_EXIT_END
```

The scope exit macros are implemented by passing a local function when constructing an object of the following class (see also `scope_exit.hpp`):

[29] Ideally, C++11 lambda functions would allow to bind variables also using `const& variable` (constant reference) and `const&` (all variables by constant reference).

```
struct scope_exit {
    scope_exit(boost::function<void (void)> f): f_(f) {}
    ~scope_exit(void) { f_(); }
private:
    boost::function<void (void)> f_;
};
```

```
#   define SCOPE_EXIT(...) \
        void BOOST_LOCAL_FUNCTION(__VA_ARGS__)
```

```
#define SCOPE_EXIT_END_(id) \
    BOOST_LOCAL_FUNCTION_NAME(BOOST_PP_CAT(scope_exit_func_, id)) \
    scope_exit BOOST_PP_CAT(scope_exit_, id)( \
            BOOST_PP_CAT(scope_exit_func_, id));
```

A local variable within the enclosing scope is used to hold the object so the destructor will be invoked at the exit of the enclosing scope and it will in turn call the local function executing the scope exit instructions. The scope exit local function has no parameter and `void` result type but it supports binding and constant binding.

Boost.Phoenix Functions

Local functions can be used to create Boost.Phoenix functions. For example (see also `phoenix_factorial_local.cpp` and `phoenix_factorial.cpp`):

Local Functions

```
int main(void) {
    using boost::phoenix::arg_names::arg1;

    int BOOST_LOCAL_FUNCTION(int n) { // Un↵
fortunately, monomorphic.
        return (n <= 0) ? 1 : n * factorial_im↵
pl(n - 1);
    } BOOST_LOCAL_FUNCTION_NAME(recurs↵
ive factorial_impl)

    boost::phoenix::function< boost::func↵
tion<int (int)> >
            factorial(factorial_impl); // ↵
Phoenix function from local function.

    int i = 4;
    BOOST_TEST(factorial(i)() == 24);     ↵
 // Call.
    BOOST_TEST(factorial(arg1)(i) == 24); ↵
 // Lazy call.
    return boost::report_errors();
}
```

Global Functor

```
struct factorial_impl { // Phoenix function ↵
from global functor.
    template<typename Sig>
    struct result;

    template<typename This, typename Arg>
    struct result<This (Arg)> : res↵
ult<This (Arg const&)> {};

    template<typename This, typename Arg>
    struct result<This (Arg&)> { ty↵
pedef Arg type; };

    template<typename Arg> // Polymorphic.
    Arg operator()(Arg n) const {
        re↵
turn (n <= 0) ? 1 : n * (*this)(n - 1);
    }
};

int main(void) {
    using boost::phoenix::arg_names::arg1;

    boost::phoenix::function<factorial_im↵
pl> factorial;

    int i = 4;
   BOOST_TEST(factorial(i)() == 24);     // ↵
Call.
   BOOST_TEST(factorial(arg1)(i) == 24);  // ↵
Lazy call.
    return boost::report_errors();
}
```

This is presented here mainly as a curiosity because Boost.Phoenix functions created from local functions have the important limitation that they cannot be polymorphic. [30] Therefore, in many cases creating the Boost.Phoenix function from global functors (possibly with the help of Boost.Phoenix adaptor macros) might be a more useful.

Closures

The following are examples of closures that illustrate how to return local functions to the calling scope (note how extra care is taken in order to ensure that all bound variables remain valid at the calling scope):

Files
`return_inc.cpp`
`return_this.cpp`
`return_setget.cpp`
`return_derivative.cpp`

[30] **Rationale.** Local functions can only be monomorphic because they are implemented using local classes and local classes cannot be templates in C++ (not even in C++11).

GCC Nested Functions

The GCC C compiler supports local functions as a non-standard extension under the name of nested functions. Note that nested functions are exclusively a C extension of the GCC compiler (they are not supported for C++ not even by the GCC compiler, and they are not part of any C or C++ standard, nor they are supported by other compilers like MSVC).

The following examples are taken form the GCC nested function documentation and programmed using local functions:

Files
`gcc_square.cpp`
`gcc_access.cpp`
`gcc_store.cpp`

N-Papers

The following examples are taken from different C++ "N-papers" and programmed using local functions:

Files	**Notes**
`n2550_find_if.cpp`	This example is adapted from [N2550] (C++11 lambda functions): It passes a local function to the STL algorithm `std::find_if`.
`n2529_this.cpp`	This example is adapted from [N2529] (C++11 lambda functions): It binds the object in scope `this` to a local function.

Annex: Alternatives

This section compares the features offered by this library with similar features offered by C++ and by other libraries.

Features

The following table compares local function features.

Local Function Feature	**Boost.LocalFunction**	**C++11 Lambda Function (Not C++03)**	**Local Functor**	**Global Functor (Not Local)**	**Boost.Phoenix**
Can be defined locally	Yes.	Yes.	Yes.	No. Therefore this not really an alternative implementation of local functions but it is listed here just for comparison.	Yes.
Can be defined using C++ statement syntax	Yes. Plus eventual compiler errors and debugging retain their usual meaning and format.	Yes. Plus eventual compiler errors and debugging retain their usual meaning and format.	Yes. Plus eventual compiler errors and debugging retain their usual meaning and format.	Yes. Plus eventual compiler errors and debugging retain their usual meaning and format.	No (it uses C++ expression template syntax).
Can be defined within expressions	No. It can be defined only within declarations.	Yes (plus the local function can be unnamed).	No. It can be defined only within declarations.	No. It can be defined only within declarations.	Yes (plus the local function can be unnamed).
Can be passed as template parameter (e.g., to STL algorithms)	Yes. The C++03 standard does not allow to pass local types as template parameters (see [N2657]) but this library implements a "trick" to get around this limitation (see the Implementation section).	Yes.	No on C++03 compilers (but yes on C++11 compilers and some compilers like MSVC 8.0, see [N2657]).	Yes.	Yes.
Access variables in scope	Yes. The variable names are repeated in the function declaration so they can be bound by value, by constant value, by reference, and by constant reference (the object `this` can also be bound using `this_`).	Yes. The variable names are repeated in the function declaration (plus there is a shorthand syntax to bind all variables in scope at once) so they can be bound by constant value and by reference (the object `this` can also be bound). However, variables cannot be bound by constant references (see below).	No. Programmers must manually program functor data members and explicitly specify their types to access variables in scope.	No. Programmers must manually program functor data members and explicitly specify their types to access variables in scope.	Yes. Variables in scope are accessible as usual within expressions (plus `boost::phoenix::let` can be used to bind variables by constant reference).
Polymorphic in the function parameter type	No (local functions cannot be function templates).	No (C++11 lambdas cannot be function templates).	No (local classes cannot have member function templates).	Yes.	Yes.

C++11 Lambda Function

C++11 lambda functions have most of the features of this library plus some additional feature (see also the example in the Introduction section):

- C++11 lambda functions can be defined within expressions while this library local functions can only be defined at declaration scope.
- C++11 lambda functions are only supported by the C++11 standard so they are not supported by all C++ compilers. This library local functions can be programmed also on C++03 compilers (and they have performances comparable to C++11 lambda functions on C++11 compilers).
- C++11 lambda functions do not allow to bind variables in scope by constant reference. Because a variable cannot be bound by constant reference, C++11 lambda functions can bind a variable by constant only if the variable is `CopyConstructible` and the binding requires a (potentially expensive) extra copy operation. Constant reference binding is instead supported by this library.
- C++11 lambda functions do not allow to bind data members selectively without binding also the object `this` while this library local functions can bind either selected data members or the entire object `this` (using `this_`).
- C++11 lambda functions provide a short-hand syntax to bind all variables in scope at once (`&` or `=`) while this library local function always require to bind variables naming them one-by-one.

For example, for non-copyable objects (see also `noncopyable_cxx11_lambda_error.cpp` and `noncopyable_local_function.cpp`):

C++11 Lambda Function

```
struct n: boost::noncopyable {
    int i;
    n(int _i): i(_i) {}
};


int main(void) {
    n x(-1);

    auto f = [x](void) {    // Error: x is ↵
non-copyable, but if
       assert(x.i == -1);  // bind `&x` then ↵
`x` is not constant.
    };
    f();

    return 0;
}
```

Boost.LocalFunction

```
struct n: boost::noncopyable {
    int i;
    n(int _i): i(_i) {}
};
BOOST_TYPEOF_REGISTER_TYPE(n) // Register for ↵
`bind& x` below.

int main(void) {
    n x(-1);

    void BOOST_LOCAL_FUNC↵
TION(const bind& x) {  // OK: No copy
        assert(x.i == -1);                  ↵
    // and constant.
    } BOOST_LOCAL_FUNCTION_NAME(f)
    f();

    return 0;
}
```

Or, for objects with expensive copy operations (see also `expensive_copy_cxx11_lambda.cpp` and `expensive_copy_local_function.cpp`):

C++11 Lambda Function

```
struct n {
    int i;
    n(int _i): i(_i) {}
    n(n const& x): i(x.i) { // Some time consuming copy operation.
        for (unsigned i = 0; i < 10000; ++i) std::cout << '.';
    }
};


int main(void) {
    n x(-1);

    auto f = [x]() {        // Problem: Expensive copy, but if bind
        assert(x.i == -1);  // by `&x` then `x` is not constant.
    };
    f();

    return 0;
}
```

Boost.LocalFunction

```
struct n {
    int i;
    n(int _i): i(_i) {}
    n(n const& x): i(x.i) { // Some time consuming copy operation.
        for (unsigned i = 0; i < 10000; ++i) std::cout << '.';
    }
};
BOOST_TYPEOF_REGISTER_TYPE(n) // Register for `bind& x` below.

int main(void) {
    n x(-1);

    void BOOST_LOCAL_FUNCTION(const bind& x) {  // OK: No copy expensive
        assert(x.i == -1);
    // copy but constant.
    } BOOST_LOCAL_FUNCTION_NAME(f)
    f();

    return 0;
}
```

When constant binding functionality is needed for C++11 lambda functions, the best alternative might be to bind an extra local variable declared constant and initialized to the original variable (for example, see *constant blocks* implemented with C++11 lambda functions in the Examples section).

Local Functor

The following example compares local functions with C++ local functors (see also `add_local_functor.cpp` and `add.cpp`):

Local Functor

```
int main(void) {
    int sum = 0, factor = 10;

   struct local_add { // Unfortunately, boil↵
erplate code to program the class.
        loc↵
al_add(int& _sum, int _factor): sum(_sum), factor(_factor) {}

       inline void operator()(int num) { // ↵
Body uses C++ statement syntax.
            sum += factor * num;
        }

    private: // Unfortunately, cannot bind ↵
so repeat variable types.
        int& sum; // Access `sum` by refer↵
ence.
        const int factor; // Make `factor` ↵
constant.
    } add(sum, factor);

    add(1);
    int nums[] = {2, 3};
   // Unfortunately, cannot pass as template ↵
parameter to `std::for_each`.
   ↵
 for(size_t i = 0; i < 2; ++i) add(nums[i]);

    BOOST_TEST(sum == 60);
    return boost::report_errors();
}
```

Boost.LocalFunction

```
int main(void) {                                ↵
 // Some local scope.
    int sum = 0, factor = 10;                   ↵
 // Variables in scope to bind.

    void BOOST_LOCAL_FUNC↵
TION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                     ↵
 // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         ↵
 // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      ↵
 // Assert final summation value.
    return boost::report_errors();
}
```

Global Functor

The following example compares local functions with C++ global functors (see also `add_global_functor.cpp` and `add.cpp`):

Global Functor

```
// Unfortunately, cannot be defined locally ↵
(so not a real alternative).
struct global_add { // Unfortunately, boiler↵
plate code to program the class.
    glob↵
al_add(int& _sum, int _factor): sum(_sum), factor(_factor) {}

   inline void operator()(int num) { // Body ↵
uses C++ statement syntax.
        sum += factor * num;
    }

private: // Unfortunately, cannot bind so re↵
peat variable types.
    int& sum; // Access `sum` by reference.
    const int factor; // Make `factor` con↵
stant.
};

int main(void) {
    int sum = 0, factor = 10;

    global_add add(sum, factor);

    add(1);
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add); // ↵
Passed as template parameter.

    BOOST_TEST(sum == 60);
    return boost::report_errors();
}
```

Boost.LocalFunction

```
int main(void) {                                ↵
 // Some local scope.
    int sum = 0, factor = 10;                   ↵
 // Variables in scope to bind.

    void BOOST_LOCAL_FUNC↵
TION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                     ↵
 // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         ↵
 // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      ↵
 // Assert final summation value.
    return boost::report_errors();
}
```

However, note that global functors do not allow to define the function locally so they are not a real alternative implementation of local functions.

Boost.Phoenix

The following example compares local functions with Boost.Phoenix (see also `add_phoenix.cpp` and `add.cpp`):

Boost.Phoenix

```
int main(void) {
    using boost::phoenix::let;
    using boost::phoenix::local_names::_f;
    using boost::phoenix::cref;
    using boost::phoenix::ref;
    using boost::phoenix::arg_names::_1;

    int sum = 0, factor = 10;
    int nums[] = {1, 2, 3};

    // Passed to template, `factor` by con↵
stant, and defined in expression.
   ↵
 std::for_each(nums, nums + 3, let(_f = cref(factor))[
        // Unfortunately, body cannot use ↵
C++ statement syntax.
        ref(sum) += _f * _1, _1 // Access ↵
`sum` by reference.
    ]);

    BOOST_TEST(sum == 60);
    return boost::report_errors();
}
```

Boost.LocalFunction

```
int main(void) {                                ↵
 // Some local scope.
    int sum = 0, factor = 10;                   ↵
 // Variables in scope to bind.

    void BOOST_LOCAL_FUNC↵
TION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                     ↵
 // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);         ↵
 // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                      ↵
 // Assert final summation value.
    return boost::report_errors();
}
```

The comparison in this section does not include the Boost.Lambda library because that library is obsolete and it was replaced by Boost.Phoenix. The Boost.Phoenix library version 3.0 is used for this comparison.

Performances

The following tables compare run-times, compile-times, and binary sizes for the different alternatives to local functions presented in this section.

Overall, this library has compile-times and generates binary sizes similar to the ones of the other approaches. This library run-times on C++03 compilers were measured to be larger than other approaches when compiler optimization is enabled (using `bjam release ...`). However, on compilers that allow to pass local types as template parameters (e.g., MSVC 8.0 or GCC 4.5.3 with C++11 features enabled `-std=c++0x`, see also [N2657] and Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`) this library automatically generates optimized code that runs as fast as the fastest of the other approaches (see the "Boost.LocalFunction" approach below). When this library local function is specified `inline` (see the "Boost.LocalFunction Inline" approach below and the Advanced Topics section) its run-times are always comparable to both the "Local Functor" and "Global Functor" approaches. However, in these cases the local function cannot be portably passed as template parameter (see [N2657] and Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`) so `std::for_each` is replaced by a for-loop (on MSVC the for-loop, and not the local function in fact the same applies to local functors, was measured to have worst performances than using `std::for_each`). Finally, this library run-times are always among the fastest when no compiler optimization is enabled (using `bjam debug ...`).

Note

The run-time performances of this library local functions are explained because on C++03 compliant compilers (e.g., GCC 4.5.3 without `-std=c++0x`) this library needs to use a function pointer in order to portably pass the local function class as a template parameter (see [N2657] and the Implementation section). For all tested compilers, this function pointer prevents the compiler optimization algorithms from inlining the local function calls. Instead, the functors used by other approaches (e.g., Boost.Phoenix) have been observed to allow all tested compilers to inline all the function calls for optimization. This run-time performance cost is not present on compilers that allow to pass local types as template parameters (e.g., MSVC 8.0 or GCC 4.5.3 with C++11 features enabled `-std=c++0x`, see Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`) because this library does not have to use the extra function pointer to implement the local function call (it directly passes the local class type as template parameter).

This run-time performance cost on C++03 compilers might or might not be an issue depending on the performance requirements of specific applications. For example, an application might already be using a number of indirect function calls (function pointers, virtual functions, etc) for which the overhead added by using the one extra function pointer required by the local function call might not be noticeable within the overall program run-time.

Finally, note that only a very simple local function body with just a single instruction was used for the anaylsis presented here (see the source files below). The authors have not studied how this library and the other approaches will perform with respect to each other when a more complex set of instructions is programmed for the local function body (e.g., *if* a more complex set of instructions in the local function body were to inhibit some compiler from inlining function objects also other approaches like C++11 lambda functions and Boost.Phoenix *could* start to show higher run-times even when optimization is enabled).

The following commands were executed from the library example directory to measure compile-time, binary size, and run-time respectively:

```
> touch <FILE_NAME>.cpp                                 # force recompilation
> python chrono.py bjam {release|debug} <FILE_NAME> # compile-time
> size <FILE_NAME>                                      # binary size
> ./<FILE_NAME>                                         # run-time
```

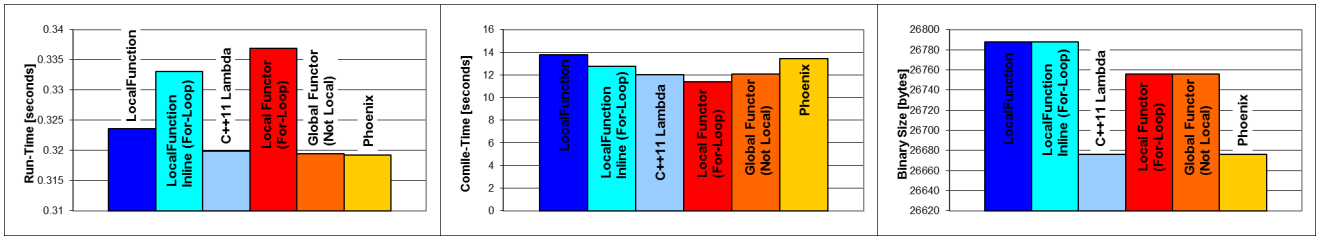
The local function was called `1e8` times to add together all the elements of a vector and the run-time was measured using Boost.Chrono averaging over `10` executions of the vector summation (see the source files below).

Legend	Approach	Source File
	Boost.LocalFunction	`profile_local_function.cpp`
	Boost.LocalFunction inline	`profile_local_function_inline.cpp`
	C++11 Lambda Function [a]	`profile_cxx11_lambda.cpp`
	Local Functor	`profile_local_functor.cpp`
	Global Functor	`profile_global_functor.cpp`
	Boost.Phoenix	`profile_phoenix.cpp`

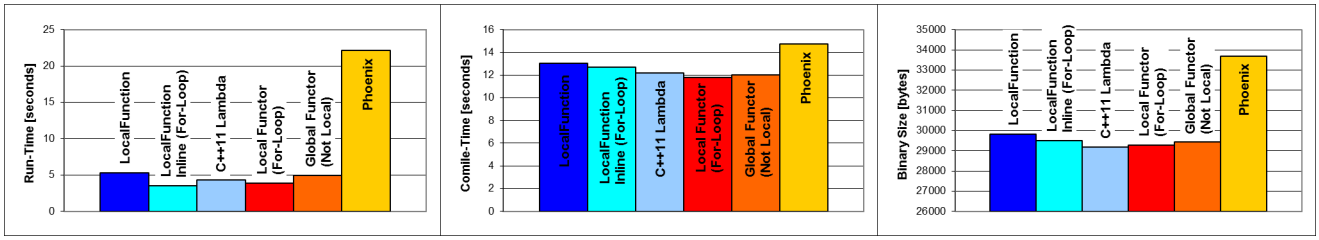
[a] Measurements available only for C++11 compilers.

GCC 4.5.3 With C++11 Lambda Functions and "Local Classes as Template Parameters" (`bjam cxxflags=-std=c++0x ...`)

Compiled with `bjam release ...` for maximum optimization (`-O3 -finline-functions`)

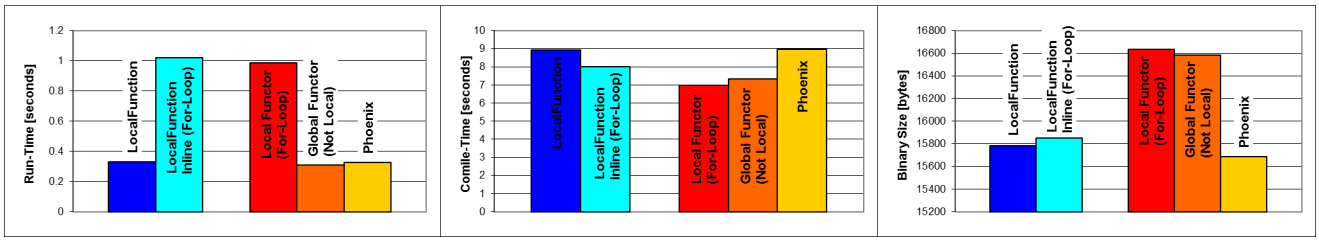


Compiled with `bjam debug ...` for no optimization (`-O0 -fno-inline`)

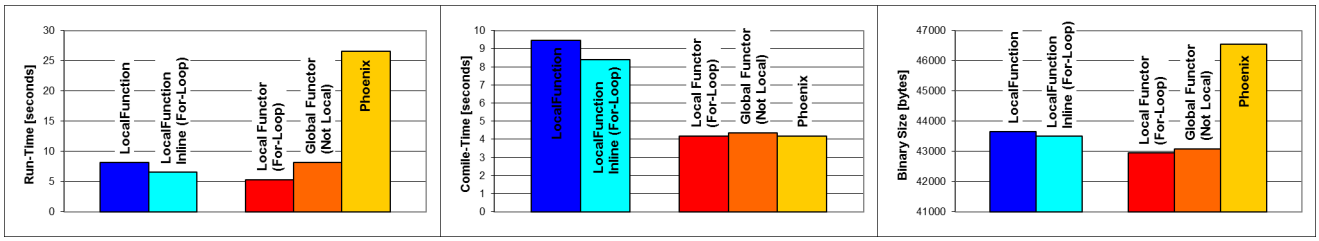


MSVC 8.0 With "Local Classes as Template Parameters" (Without C++11 Lambda Functions)

Compiled with `bjam release ...` for maximum optimization (`/O2 /Ob2`)

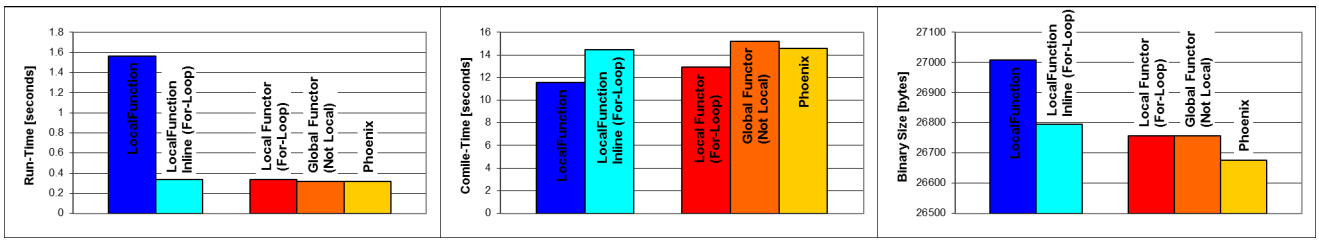


Compiled with `bjam debug ...` for no optimization (`/Od /Ob0`)

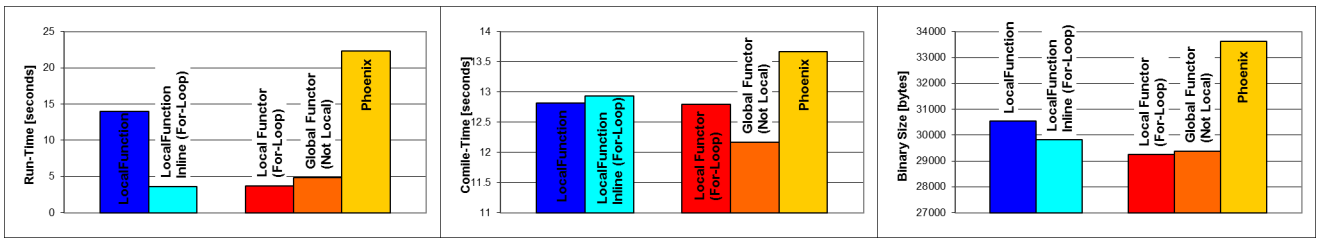


GCC 4.3.4 With C++03 Only (Without C++11 Lambda Functions and Without "Local Classes as Template Parameters")

Compiled with `bjam release ...` for maximum optimization (`-O3 -finline-functions`)



Compiled with `bjam debug ...` for no optimization (`-O0 -fno-inline`)



Annex: No Variadic Macros

This section illustrates an alternative syntax for compilers without variadic macro support.

Sequence Syntax

Most modern compilers support variaid macros (notably, these include GCC, MSVC, and all C++11 compilers). However, in the rare case that programmers need to use this library on a compiler without variadic macros, this library also allows to specify its macro parameters using a Boost.Preprocessor sequence where tokens are separated by round parenthesis `()`:

```
(token1) (token2) ... // All compilers.
```

Instead of the comma-separated list that we have seen so far which requires variadic macros:

```
token1, token2, ... // Only compilers with varidic macros.
```

For example, the following syntax is accepted on all compilers with and without variadic macros (see also `add_seq.cpp`):

```
int main(void) {
    int sum = 0, factor = 10;

    void BOOST_LOCAL_FUNCTION( (const bind factor) (bind& sum) (int num) ) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);

    BOOST_TEST(sum == 60);
    return boost::report_errors();
}
```

However, on compilers with variadic macros the comma-separated syntax we have seen so far is preferred because more readable (see also `add.cpp`):

```
int main(void) {                            // Some local scope.
    int sum = 0, factor = 10;               // Variables in scope to bind.

    void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
        sum += factor * num;
    } BOOST_LOCAL_FUNCTION_NAME(add)

    add(1);                                 // Call the local function.
    int nums[] = {2, 3};
    std::for_each(nums, nums + 2, add);     // Pass it to an algorithm.

    BOOST_TEST(sum == 60);                  // Assert final summation value.
    return boost::report_errors();
}
```

Note that the same macros accept both syntaxes on compilers with variadic macros and only the sequence syntax on compilers without variadic macros. Finally, an empty local function parameter list is always specified using `void` on compilers with and without variadic macros:

```
int BOOST_LOCAL_FUNCTION(void) { // No parameter.
    return 10;
} BOOST_LOCAL_FUNCTION_NAME(ten)

BOOST_TEST(ten() == 10);
```

Examples

For reference, the following is a list of most of the examples presented in this documentation reprogrammed using the sequence syntax instead of the comma-separated syntax (in alphabetic order):

Files
add_classifiers_seq.cpp
add_default_seq.cpp
add_except_seq.cpp
add_inline_seq.cpp
add_params_only_seq.cpp
add_template_seq.cpp
add_this_seq.cpp
add_typed_seq.cpp
add_with_default_seq.cpp
all_decl_seq.cpp
factorial_seq.cpp
macro_commas_seq.cpp
nesting_seq.cpp
overload_seq.cpp
return_assign_seq.cpp
return_derivative_seq.cpp
return_inc_seq.cpp
return_setget_seq.cpp
return_this_seq.cpp
same_line_seq.cpp
transform_seq.cpp
typeof_seq.cpp
typeof_template_seq.cpp

Annex: Implementation

This section gives an overview of the key programming techniques used to implement this library.

Note

The code listed here can be used by curious readers and library maintainers as a reference in trying to understand the library source code. There is absolutely no guarantee that the library implementation uses the exact code listed here.

Local Classes as Template Parameters

This library uses a local class to implement the local function object. However, in C++03 local classes (and therefore the local function objects they implement) cannot be passed as template parameters (e.g., to the `std::for_each` algorithm), this is instead possible in C++11, MSVC, and some other compilers (see [N2657] and Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`). To work around this limitation, this library investigated the following two "tricks" (both tricks can be extended to support function default parameters):

1. The *casting functor trick* uses a non-local functor that calls a static member function of the local class via a function pointer. The static member function then calls the correct local function body after type casting the object from a `void*` pointer (local classes can always be used for type casting via `static_cast` or similar).

2. The *virtual functor trick* derives the local functor class from a non-local base class. The correct overridden implementation of the virtual `operator()` is then called via dynamic binding.

For example (see also `impl_tparam_tricks.cpp`):

```
#include <boost/detail/lightweight_test.hpp>
#include <vector>
#include <algorithm>

// Casting functor trick.
struct casting_func {
    explicit casting_func(void* obj, void (*call)(void*, const int&))
            : obj_(obj), call_(call) {}
    // Unfortunately, function pointer call is not inlined.
    inline void operator()(const int& num) { call_(obj_, num); }
private:
    void* obj_;
    void (*call_)(void*, const int&);
};

// Virtual functor trick.
struct virtual_func {
    struct interface {
        // Unfortunately, virtual function call is not inlined.
        inline virtual void operator()(const int&) {}
    };
    explicit virtual_func(interface& func): func_(&func) {}
    inline void operator()(const int& num) { (*func_)(num); }
private:
    interface* func_;
};

int main(void) {
    int sum = 0, factor = 10;

    // Local class for local function.
    struct local_add : virtual_func::interface {
        explicit local_add(int& _sum, const int& _factor)
                : sum_(_sum), factor_(_factor) {}
        inline void operator()(const int& num) {
            body(sum_, factor_, num);
        }
        inline static void call(void* obj, const int& num) {
            local_add* self = static_cast<local_add*>(obj);
            self->body(self->sum_, self->factor_, num);
        }
    private:
        int& sum_;
        const int& factor_;
        inline void body(int& sum, const int& factor, const int& num) {
            sum += factor * num;
        }
    } add_local(sum, factor);
    casting_func add_casting(&add_local, &local_add::call);
    virtual_func add_virtual(add_local);

    std::vector<int> v(10);
    std::fill(v.begin(), v.end(), 1);

    // std::for_each(v.begin(), v.end(), add_local); // Error but OK on C++11.
    std::for_each(v.begin(), v.end(), add_casting); // OK.
    std::for_each(v.begin(), v.end(), add_virtual); // OK.

    BOOST_TEST(sum == 200);
    return boost::report_errors();
}
```

The casting functor trick measured slightly better run-time performances than the virtual functor trick so the current implementation of this library uses the casting functor trick (probably because in addition to the indirect function call, the virtual functor trick also requires accessing the virtual function table). However, neither one of the two tricks was observed to allow for compiler optimizations that inline the local function calls (because they rely on one indirect function call via either a function pointer or a virtual function respectively). Therefore, on compilers that accept local classes as template parameters (MSVC, C++11, etc, see [N2657] and Boost.Config's `BOOST_NO_LOCAL_CLASS_TEMPLATE_PARAMETERS`), this library automatically generates code that passes the local class type directly as template parameter without using neither one of these two tricks in order to take full advantage of compiler optimizations that inline the local function calls.

Parsing Macros

This library macros can parse the list of specified parameters and detect if any of the bound variable names matches the token `this_` (to generate special code to bind the object in scope), or if the variable is bound by `const` (to generate special code to bind by constant), etc. The parameter tokens are inspected using preprocessor meta-programming and specifically using the macros defined by the files in the `boost/local_function/detail/preprocessor/keyword/` directory. [31]

For example, the following code defines a macro that allows the preprocessor to detect if a set of space-separated tokens ends with `this_` or not (see also `impl_pp_keyword.cpp`):

```
#include <boost/local_function/detail/preprocessor/keyword/thisunderscore.hpp>
#include <boost/local_function/detail/preprocessor/keyword/const.hpp>
#include <boost/local_function/detail/preprocessor/keyword/bind.hpp>
#include <boost/detail/lightweight_test.hpp>

// Expand to 1 if space-separated tokens end with `this_`, 0 otherwise.
#define IS_THIS_BACK(tokens) \
    BOOST_LOCAL_FUNCTION_DETAIL_PP_KEYWORD_IS_THISUNDERSCORE_BACK( \
    BOOST_LOCAL_FUNCTION_DETAIL_PP_KEYWORD_BIND_REMOVE_FRONT( \
    BOOST_LOCAL_FUNCTION_DETAIL_PP_KEYWORD_CONST_REMOVE_FRONT( \
        tokens \
    )))

int main(void) {
    BOOST_TEST(IS_THIS_BACK(const bind this_) == 1);
    BOOST_TEST(IS_THIS_BACK(const bind& x) == 0);
    return boost::report_errors();
}
```

[31] This technique is at the core of even more complex preprocessor parsing macros like the ones that parse the Contract++ syntax.

Reference

Header <boost/local_function.hpp>

Local functions allow to program functions locally, within other functions, and directly within the scope where they are needed.

```
BOOST_LOCAL_FUNCTION(declarations)
BOOST_LOCAL_FUNCTION_TPL(declarations)
BOOST_LOCAL_FUNCTION_ID(id, declarations)
BOOST_LOCAL_FUNCTION_ID_TPL(id, declarations)
BOOST_LOCAL_FUNCTION_NAME(qualified_name)
BOOST_LOCAL_FUNCTION_NAME_TPL(name)
BOOST_LOCAL_FUNCTION_TYPEOF(bound_variable_name)
```

Macro BOOST_LOCAL_FUNCTION

BOOST_LOCAL_FUNCTION — This macro is used to start a local function declaration.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION(declarations)
```

Description

This macro must be used within a declarative context, it must follow the local function result type, it must be followed by the local function body code, and then by the `BOOST_LOCAL_FUNCTION_NAME` macro (see the Tutorial and Advanced Topics sections):

```
{ // Some declarative context.
    ...
    result_type BOOST_LOCAL_FUNCTION(declarations) {
        ... // Body code.
    } BOOST_LOCAL_FUNCTION_NAME(qualified_name)
    ...
}
```

As usual, exceptions specifications can be optionally programmed just after the macro and before the body code block `{ ... }` (but the exception specifications will only apply to the body code and not to the library code automatically generated by the macro expansion, see the Advanced Topics section).

Within templates, the special macros `BOOST_LOCAL_FUNCTION_TPL` and `BOOST_LOCAL_FUNCTION_NAME_TPL` must be used.

Parameters:

declarations	On compilers that support variadic macros, the parameter declarations are defined by the following grammar:
	`declarations:`
	`  void | declaration_tuple | declaration_sequence`
	`declaration_tuple:`
	`  declaration, declaration, ...`
	`declaration_sequence:`
	`  (declaration) (declaration) ...`
	`declaration:`
	`  bound_variable | parameter | default_value | result_type`
	`bound_variable:`
	`  [const] bind [(variable_type)] [&] variable_name`
	`parameter:`
	`  [auto | register] parameter_type parameter_name`
	`default_value:`
	`  default parameter_default_value`
	`result_type:`
	`  return function_result_type`
	On compilers that do not support variadic macros, `declaration_tuple` cannot be used:
	`declarations:`
	`  void | declaration_sequence`
	(Lexical conventions: `token1 | token2` means either `token1` or `token2`; `[token]` means either `token` or nothing; `{expression}` means the token resulting from the expression.)

Note that on compilers that support variadic macros, commas can be used to separate the declarations resembling more closely the usual C++ function declaration syntax (this is the preferred syntax). However, for portability, on all C++ compilers (with and without variadic macros) the same library macros also accept parameter declarations specified as a Boost.Preprocessor sequence separated by round parenthesis `()`.

When binding the object `this`, the special symbol `this_` needs to be used instead of `this` as the name of the variable to bind and also within the local function body to access the object. (Mistakenly using `this` instead of `this_` might not always result in a compiler error and will in general result in undefined behaviour.)

The result type must either be specified just before the macro or within the macro declarations prefixed by `return` (but not in both places).

Within the local function body it possible to access the result type using `result_type`, the type of the first parameter using `arg1_type`, the type of the second parameter using `arg2_type`, etc. The bound variable types can be accessed using `BOOST_LOCAL_FUNCTION_TYPEOF`.

This macro cannot be portably expanded multiple times on the same line. In these cases, use the `BOOST_LOCAL_FUNCTION_ID` macro instead.

The maximum number of local function parameters (excluding bound variables) is specified by the configuration macro `BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX`. The maximum number of bound variables is specified by the configuration macro `BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX`. The configuration macro `BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS` can be used to force optimizations that reduce the local function call run-time overhead.

Note: Local functions are functors so they can be assigned to other functors like `boost::function` (see Boost.Function).

See: Tutorial section, Advanced Topics section, BOOST_LOCAL_FUNCTION_NAME, BOOST_LOCAL_FUNCTION_TPL, BOOST_LOCAL_FUNCTION_NAME_TPL, BOOST_LOCAL_FUNCTION_TYPEOF, BOOST_LOCAL_FUNCTION_ID, BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX, BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX, BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS.

Macro BOOST_LOCAL_FUNCTION_TPL

BOOST_LOCAL_FUNCTION_TPL — This macro is used to start a local function declaration within templates.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_TPL(declarations)
```

Description

This macro must be used instead of BOOST_LOCAL_FUNCTION when declaring a local function within a template. A part from that, this macro has the exact same syntax a BOOST_LOCAL_FUNCTION (see BOOST_LOCAL_FUNCTION for more information):

```
{ // Some declarative context within a template.
    ...
    result_type BOOST_LOCAL_FUNCTION_TPL(declarations) {
        ... // Body code.
    } BOOST_LOCAL_FUNCTION_NAME_TPL(qualified_name)
    ...
}
```

Note that BOOST_LOCAL_FUNCTION_NAME_TPL must be used with this macro instead of BOOST_LOCAL_FUNCTION_NAME.

This macro cannot be portably expanded multiple times on the same line. In these cases, use the BOOST_LOCAL_FUNCTION_ID_TPL macro instead.

Note: C++03 does not allow to use `typename` outside templates. This library internally manipulates types, these operations require `typename` but only within templates. This macro is used to indicate to the library when the enclosing scope is a template so the library can correctly use `typename`.

See: Tutorial section, BOOST_LOCAL_FUNCTION, BOOST_LOCAL_FUNCTION_ID_TPL, BOOST_LOCAL_FUNCTION_NAME_TPL.

Macro BOOST_LOCAL_FUNCTION_ID

BOOST_LOCAL_FUNCTION_ID — This macro allows to declare multiple local functions on the same line.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_ID(id, declarations)
```

Description

This macro is equivalent to BOOST_LOCAL_FUNCTION but it can be expanded multiple times on the same line if different identifiers `id` are provided for each expansion (see the Advanced Topics section).

Parameters:

id	A unique identifier token which can be concatenated by the preprocessor (`__LINE__`, `local_function_number_1_on_line_123`, etc).
declarations	Same as the `declarations` parameter of the `BOOST_LOCAL_FUNCTION` macro.

The `BOOST_LOCAL_FUNCTION_NAME` macro should be used to end each one of the multiple local function declarations as usual (and it will specify a unique name for each local function).

Within templates, the special macros `BOOST_LOCAL_FUNCTION_ID_TPL` must be used.

Note: This macro can be useful when the local function macros are expanded within user-defined macros (because macros all expand on the same line). On some compilers (e.g., MSVC which supports the non-standard `__COUNTER__` macro) it might not be necessary to use this macro but the use of this macro when expanding multiple local function macros on the same line is always necessary to ensure portability (this is because this library can only portably use `__LINE__` to internally generate unique identifiers).

See: Advanced Topics section, `BOOST_LOCAL_FUNCTION`, `BOOST_LOCAL_FUNCTION_NAME`, `BOOST_LOCAL_FUNCTION_ID_TPL`.

Macro BOOST_LOCAL_FUNCTION_ID_TPL

BOOST_LOCAL_FUNCTION_ID_TPL — This macro allows to declare multiple local functions on the same line within templates.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_ID_TPL(id, declarations)
```

Description

This macro must be used instead of `BOOST_LOCAL_FUNCTION_TPL` when declaring multiple local functions on the same line within a template. A part from that, this macro has the exact same syntax as `BOOST_LOCAL_FUNCTION_TPL` (see `BOOST_LOCAL_FUNCTION_TPL` for more information).

Parameters:

id	A unique identifier token which can be concatenated by the preprocessor (`__LINE__`, `local_function_number_1_on_line_123`, etc).
declarations	Same as the `declarations` parameter of the `BOOST_LOCAL_FUNCTION_TPL` macro.

The `BOOST_LOCAL_FUNCTION_NAME` macro should be used to end each one of the multiple local function declarations as usual (and it will specify a unique name for each local function).

Outside template, the macro `BOOST_LOCAL_FUNCTION_ID` should be used to declare multiple local functions on the same line.

Note: This macro can be useful when the local function macros are expanded within user-defined macros (because macros all expand on the same line). On some compilers (e.g., MSVC which supports the non-standard `__COUNTER__` macro) it might not be necessary to use this macro but the use of this macro when expanding multiple local function macros on the same line is always necessary to ensure portability (this is because this library can only portably use `__LINE__` to internally generate unique identifiers).

See: Advanced Topics section, `BOOST_LOCAL_FUNCTION_TPL`, `BOOST_LOCAL_FUNCTION_NAME`, `BOOST_LOCAL_FUNCTION_ID`.

Macro BOOST_LOCAL_FUNCTION_NAME

BOOST_LOCAL_FUNCTION_NAME — This macro is used to end a local function declaration specifying its name.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_NAME(qualified_name)
```

Description

This macro must follow the local function body code block `{ ... }`:

```
{ // Some declarative context.
    ...
    result_type BOOST_LOCAL_FUNCTION(declarations) {
        ... // Body code.
    } BOOST_LOCAL_FUNCTION_NAME(qualified_name)
    ...
}
```

Within templates, the special macros `BOOST_LOCAL_FUNCTION_TPL` and `BOOST_LOCAL_FUNCTION_NAME_TPL` must be used.

Parameters:

`qualified_name`	The name of the local function optionally qualified as follow:
	`name:`
	`  [inline] [recursive] local_function_name`
	(Lexical conventions: `token1 | token2` means either `token1` or `token2`; `[token]` means either `token` or nothing; `{expression}` means the token resulting from the expression.)

The local function name can be qualified by prefixing it with the keyword `inline` (see the Advanced Topics section):

```
BOOST_LOCAL_FUNCTION_NAME(inline local_function_name)
```

This increases the chances that the compiler will be able to inline the local function calls (thus reducing run-time). However, inline local functions cannot be passed as template parameters (e.g., to `std::for_each`) or assigned to other functors (e.g., to `boost::function`). That is true on C++03 compilers but inline local functions can instead be passed as template parameters on C++11 compilers. On C++11 compilers, there is no need to declare a local function lined because this library will automatically use C++11 specific features to inline the local function while always allowing to pass it as a template parameter. This optimization is automatically enabled when the Boost.Config macro `BOOST_NO_CXX11_LOCAL_CLASS_TEMPLATE_PARAMETERS` is not defined but it also be forced using `BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS`.

The local function name can also be qualified by prefixing it with the "keyword" `recursive` (see the Advanced Topics section):

```
BOOST_LOCAL_FUNCTION_NAME(recursive local_function_name)
```

This allows the local function to recursively call itself from its body (as usual in C++). However, recursive local functions should only be called within their declaration scope (otherwise the result is undefined behaviour). Finally, compilers have not been observed to be able to inline recursive local function calls, not even when the recursive local function is also declared inline:

```
BOOST_LOCAL_FUNCTION(inline recursive local_function_name)
```

Note: The local function name cannot be the name of an operator `operator...` and it cannot be the same name of another local function declared within the same enclosing scope (but `boost::overloaded_function` can be used to overload local functions, see Boost.Functional/OverloadedFunction and the Advanced Topics section).

See: Tutorial section, Advanced Topics section, `BOOST_LOCAL_FUNCTION`, `BOOST_LOCAL_FUNCTION_NAME_TPL`.

Macro BOOST_LOCAL_FUNCTION_NAME_TPL

BOOST_LOCAL_FUNCTION_NAME_TPL — This macro is used to end a local function declaration specifying its name within templates.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_NAME_TPL(name)
```

Description

This macro must be used instead of `BOOST_LOCAL_FUNCTION_NAME` when declaring a local function within a template. A part from that, this macro has the exact same syntax a `BOOST_LOCAL_FUNCTION_NAME` (see `BOOST_LOCAL_FUNCTION_NAME` for more information):

```
{ // Some declarative context within a template.
    ...
    result_type BOOST_LOCAL_FUNCTION_TPL(declarations) {
        ... // Body code.
    } BOOST_LOCAL_FUNCTION_NAME_TPL(qualified_name)
    ...
}
```

Note that `BOOST_LOCAL_FUNCTION_TPL` must be used with this macro instead of `BOOST_LOCAL_FUNCTION`.

Note: C++03 does not allow to use `typename` outside templates. This library internally manipulates types, these operations require `typename` but only within templates. This macro is used to indicate to the library when the enclosing scope is a template so the library can correctly use `typename`.

See: Tutorial section, `BOOST_LOCAL_FUNCTION_NAME`, `BOOST_LOCAL_FUNCTION_TPL`.

Macro BOOST_LOCAL_FUNCTION_TYPEOF

BOOST_LOCAL_FUNCTION_TYPEOF — This macro expands to the type of the specified bound variable.

Synopsis

```
// In header: <boost/local_function.hpp>

BOOST_LOCAL_FUNCTION_TYPEOF(bound_variable_name)
```

Description

This macro can be used within the local functions body to refer to the bound variable types so to declare local variables, check concepts (using Boost.ConceptCheck), etc (see the Advanced Topics section). This way the local function can be programmed entirely without explicitly specifying the bound variable types thus facilitating maintenance (e.g., if the type of a bound variable changes in the enclosing scope, the local function code does not have to change).

Parameters:

`bound_variable_name`	The name of one of the local function's bound variables.

The type returned by the macro is fully qualified in that it contains the extra constant and reference qualifiers when the specified variable is bound by constant and by reference. For example, if a variable named `t` of type `T` is:

- Bound by value using `bind t` then `BOOST_LOCAL_FUNCTION_TYPEOF(t)` is `T`.
- Bound by constant value using `const bind t` then `BOOST_LOCAL_FUNCTION_TYPEOF(t)` is `const T`.
- Bound by reference using `bind& t` then `BOOST_LOCAL_FUNCTION_TYPEOF(t)` is `T&`.
- Bound by constant reference using `const bind& t` then `BOOST_LOCAL_FUNCTION_TYPEOF(t)` is `const T&`.

This macro must be prefixed by `typename` when used within templates.

Note: It is best to use this macro instead of Boost.Typeof so to reduce the number of times Boost.Typeof is used to deduce types (see the Advanced Topics section).

See: Advanced Topics section, `BOOST_LOCAL_FUNCTION`.

Header <boost/local_function/config.hpp>

Configuration macros allow to change the behaviour of this library at compile-time.

```
BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX
BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX
BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS
```

Macro BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX

BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX — Maximum number of parameters supported by local functions.

Synopsis

```
// In header: <boost/local_function/config.hpp>

BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX
```

Description

If programmers leave this configuration macro undefined, its default value is `5` (increasing this number might increase compilation time). When defined by programmers, this macro must be a non-negative integer number.

Note: This macro specifies the maximum number of local function parameters excluding bound variables (which are instead specified by `BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX`).

See: Tutorial section, Getting Started section, `BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX`.

Macro BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX

BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX — Maximum number of bound variables supported by local functions.

Synopsis

```
// In header: <boost/local_function/config.hpp>

BOOST_LOCAL_FUNCTION_CONFIG_BIND_MAX
```

Description

If programmers leave this configuration macro undefined, its default value is `10` (increasing this number might increase compilation time). When defined by programmers, this macro must be a non-negative integer number.

Note: This macro specifies the maximum number of bound variables excluding local function parameters (which are instead specified by `BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX`).

See: Tutorial section, Getting Started section, `BOOST_LOCAL_FUNCTION_CONFIG_ARITY_MAX`.

Macro BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS

BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS — Specify when local functions can be passed as template parameters without introducing any run-time overhead.

Synopsis

```
// In header: <boost/local_function/config.hpp>

BOOST_LOCAL_FUNCTION_CONFIG_LOCALS_AS_TPARAMS
```

Description

If this macro is defined to `1`, this library will assume that the compiler allows to pass local classes as template parameters:

```
template<typename T> void f(void) {}

int main(void) {
    struct local_class {};
    f<local_class>();
    return 0;
}
```

This is the case for C++11 compilers and some C++03 compilers (e.g., MSVC), but it is not the case in general for most C++03 compilers (including GCC). This will allow the library to pass local functions as template parameters without introducing any run-time overhead (specifically without preventing the compiler from optimizing local function calls by inlining their assembly code).

If this macro is defined to `0` instead, this library will introduce a run-time overhead associated to resolving a function pointer call in order to still allow to pass the local functions as template parameters.

It is recommended to leave this macro undefined. In this case, the library will automatically define this macro to `0` if the Boost.Config macro `BOOST_NO_CXX11_LOCAL_CLASS_TEMPLATE_PARAMETERS` is defined for the specific compiler, and to `1` otherwise.

See: Getting Started section, Advanced Topics section, `BOOST_LOCAL_FUNCTION_NAME`.

Release Notes

This section lists the major changes between different library releases (in chronological order).

Version 1.0.0 (2012-04-12)

1. Incorporated all comments from the Boost review of this library.
2. Removed local blocks and local exits.
3. Renamed the library from Boost.Local to Boost.LocalFunction.
4. Using `this_` instead of `this` also in the local function declaration (not just the body).
5. Made changes that allow to return local functions (similar to closures).
6. Added GCC lambda and constant block examples.
7. Moved `overloaded_function` to Boost.Functional/OverloadedFunction.
8. Moved `BOOST_IDENTITY_TYPE` to Boost.Utility/IdentityType.
9. Completely removed use of Boost.Typeof when bound and result types are explicitly specified.
10. Added `..._ID` macros for multiple expansions on the same line.
11. Fixed compilation on Boost regression test platforms.

Version 0.2.0 (2011-05-14)

1. Replaced parenthesized syntax with variadic and sequencing macro syntaxes.
2. Profiled library performances against other approaches.
3. Replaced virtual functor trick with casting functor trick (for smaller run-time).
4. Optimized library run-time (rearranging code and not using casting functor trick on compilers that accept local classes as template parameters).
5. Supported inline and recursive local functions.
6. Added type-of macro to expose bound types.
7. Allowed to explicitly specify bound types.
8. Removed using `boost::function` instead of exposing internal local functor as public API.
9. Added functor to overload local functions (and functors in general).
10. Implemented support for nesting local functions, blocks, and exits into one another.

Version 0.1.1 (2011-01-10)

1. Uploaded library source into Boost SVN sandbox.
2. Fixed prev/next arrows and other minor layouts in documentation.
3. Added Release section to documentation.

Version 0.1.0 (2011-01-03)

1. Shared with Boost for first round of comments.

Version 0.0.1 (2010-12-15)

1. Completed development, examples, and documentation.

Bibliography

This section lists all the bibliographic references cited by this documentation.

[N1613] Thorsten Ottosen. *Proposal to add Design by Contract to C++*. The C++ Standards Committee, document no. N1613=04-0053, 2004.

[N2511] Alisdair Meredith. *Named Lambdas and Local Functions*. The C++ Standards Committee, document no. N2511=08-0021, 2008.

[N2529] Jaakko Jarvi, John Freeman, Lawrence Crowl. *Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 3)*. The C++ Standards Committee, document no. N2529=08-0039, 2008.

[N2550] Jaakko Jarvi, John Freeman, Lawrence Crowl. *Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)*. The C++ Standards Committee, document no. N2550=08-0060, 2008.

[N2657] John Spicer. *Local and Unamed Types as Template Arguments*. The C++ Standard Committee, document no. N2657=08-0167, 2008.

Acknowledgments

This section aims to recognize the contributions of *all* the different people that participated directly or indirectly to the design and development of this library.

Many thanks to Steven Watanabe and Vicente Botet for suggesting to me to use Boost.ScopeExit binding to emulate local functions. Many thanks to Alexander Nasonov for clarifying how Boost.ScopeExit binding could be used to implement local functions and for some early work in this direction.

Many thanks to Gregory Crosswhite for using an early version of this library in one of his projects.

Thanks to David Abrahams, Vicente Botet, et al. for suggesting to provide the variadic macro syntax on compilers that support variadic macros.

Thanks to Pierre Morcello for sharing some code that experimented with implementing local functions using Boost.ScopeExit binding (even if this library is not based on such a code).

Thanks to John Bytheway for checking the authors' virtual functor technique that originally allowed this library to pass local functions as template parameters.

Thanks to Jeffrey Lee Hellrung for suggesting the use of the "keyword" `bind` to bind variables in scope and for suggesting to use `bind(`*`type`*`)` to optionally specify the bound variable type. Thanks to Vicente Botet for suggesting to provide a macro to expose the bound variable type to the public API.

Thanks to Steven Watanabe, Vicente Botet, Michael Caisse, Yechezkel Mett, Joel de Guzman, Thomas Heller, et al. for helping with the Alternatives section and with the profiling of the different alternatives.

Many thanks to Jeffrey Lee Hellrung for managing the Boost review of this library. Thanks also to all the people that submitted a Boost review: Andrzej Krzemienski, Edward Diener, Gregory Crosswhite, John Bytheway, Hartmut Kaiser, Krzysztof Czainski, Nat Lindon, Pierre Morcello, Thomas Heller, and Vicente J. Botet. Thanks to everyone that commented on the library during its Boost review.

Finally, 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.