Table of Contents

1 Reference

1.1 Macros

1.1.1 WG_LCLFUNCTION(name, …) WG_LCLFUNCTION_TPL(name, …) WG_LCLFUNCTION_END

These macros start and end local function definitions and, as such, must be used within a declarative context. Their syntax is defined here. `WG_LCLFUNCTION` is for use in non-template functions and `WG_LCLFUNCTION_TPL` is for use in template functions. The second argument of the aforementioned macros both accept the following named parameters, in the order specified below:

a) `void`

b) `return`

Used to specify the return type of the local function. If not used, the return type defaults to void.

c) `params`

Used to specify the signature of the local function. If not used, the signature defaults to void.

The number of arguments for this named parameter is limited by `WG_PP_LCLFUNCTION_CONFIG_PARAMS_MAX_ARITY`.

d) `varbind`

Used to bind function variables with variables of the same name in enclosing scopes. For more information see Bound Function Variables.

e) `varset`

Used to set function variables with user declared expressions. For more information see Set Function Variables.

1.1.2 WG_LCLFUNCTION_TYPENAME(name)

Expands to the implementation defined type name of the local function named `name` without invoking `BOOST_TYPEOF`, as a result, top-level qualifiers like const and reference are preserved.

1.1.3 WG_PP_LCLFUNCTION_CONFIG_PARAMS_MAX_ARITY

A user configurable object macro that controls the maximum number of arguments that may be

specified for the named parameter `params`. If set, then it must expand to a non-negative integer, else it defaults to the number 5.

The maximum number of arguments which `params` can handle is equal to:

```
min( WG_PP_LCLFUNCTION_CONFIG_PARAMS_MAX_ARITY, BOOST_PP_LIMIT_REPEAT )
```

Note: in order for this macro to take effect in any particular translation unit, that translation unit must define it before the first inclusion of `LclFunction.hh` .

1.1.4 `WG_LCLFUNCTION_VAR_TYPEOF(function_variable_name)`

This macro is only available within a local function body and, without invoking `BOOST_TYPEOF`, expands to the typename of the user-specified function variable `function_variable_name`.

NOTE: unlike `BOOST_TYPEOF`, this macro does not strip any top level qualifiers like const or reference.

1.2 Function Variables

Function variables are variables that are associated with each local function. They can be accessed directly from within the local function body and preserve their state across multiple calls to the same local function instance. There are two types of function variables: bound and set, and each are described below.

Note: Function variables are not necessarily implemented as member variables of a function instance. It is an implementation detail what form they take. That's why they're specified with `var` prefixed names and not `mem` prefixed names.

Note: It is the user's responsibility to ensure that at the time of a local function invocation all its function variables are valid. (See limitations for more information.)

1.2.1 Bound Function Variables

Bound function variables are specified using the named parameter `varbind`. They are so named because their values are bound to variables of the same name from the enclosing scope(s).

1.2.1.1 `this_`

Bound variables named `this_` are implicitly bound to `this` keyword, this is in keeping with existing BOOST conventions. If implicitly typed and explicitly qualified `const` then the constness will apply to the pointed-to-type and not the type of `this_` (see Rationale).

1.2.2 Set Function Variables

Set function variables are specified using the named parameter `varset`. They are so named because their values are determined by user specified expressions. Such an expression may contain identifiers from the enclosing scope(s), but not any declared in `varset` itself. If the said expression contains any commas, then the whole expression must be wrapped in an extra set of parenthesis (see Limitations). Additionally, neither shall it be an array initializer if compiling for C++03 (see Limitations).

1.2.3 Explicit vs Implicit Types

Function variables may be explicitly or implicitly typed. If implicitly typed, the types of such variables are determined from the variable they are bound to, or, from the expression they are set to, via `BOOST_TYPEOF`. Thus, it is important to remember that implicitly typed variables lose their top-most const or reference qualification (see `BOOST_TYPEOF`).

1.2.3.1 Explicit Type Specification Syntax

Function variables are explicitly typed when their variable is preceded by the declaration `type(type-specifier)`, where `type-specifier` is defined in the C++ standard and the actual type of the associated function variable must be implicitly convertible to it.

1.2.3.2 Implicit Type Specification Syntax

Function variables that are not explicitly typed are implicitly typed. Their declaration may be preceded by the keyword combinations `const`, `ref`, or `const ref`, indicating how their deduced types should be top-most qualified. Hence, an implicitly typed function variable shall not be named "ref".

1.3 Standard Type Names and Constants

1.3.1 Local Function Type Name

The type of any local function object may be obtained from `WG_LCLFUNCTION_TYPENAME`.

1.3.2 Local Function Signature Typedefs and Constants

Each local function defines the following typedefs and constants:

- `function_type`
- `<some-constant-integral-type> arity`
- `result_type`
- `arg1_type`
- `...`
- `argN_type`

`function_type` is an alias for the local function's user specified signature and return type.

`result_type` is an alias for the return type of `function_type`.

`argM_type` is an alias for the M^{th} element of `boost::function_type::parameter_type<function_type>`.

Except for `function_type`, the above are analogues to the type names and constants defined by `boost::function_traits` in Boost.TypeTraits library.

Note: from within the local function body the above can be directly accessed. Outside the local function body one must use the local function type name scope to access them.

1.4 Interactions with Boost.Function

Local functions may be assigned to Boost.Function's `boost::function`, and, respecting all the usual caveats, returned from their scope of definition.

Note: the const qualification of a `boost::function` wrapped local function will not affect how the local function is called, it only affects whether the `boost::function` object can be mutated or not.

1.5 Interactions with other LclFunctions

1.5.1 As Function Variables and Function Parameter Arguments

Because local functions have non-local types, they may be used as arguments to local function variables and local function parameters.

1.5.2 Nesting

Local functions may be nested within one another and may even reuse the same name, with the inner most scope name overriding all others.

1.5.3 Same Line Declarations

Multiple local functions may be declared on the same line, for example, as part of a larger macro definition.

2 Limitations

2.1 C++03 Limitations

2.1.1 Library Boundaries

Local functions may not be usable across library boundaries because their user-provided definitions are ultimately implemented using local types, and in C++03 local types lack linkage. Hence, their symbols may not have been exported.

2.1.2 Set-To Expressions

In C++03 the set-to expressions of a set function variable shall not be array initializer. This is because array initializers may not be used as parameter arguments in C++03. This restriction may be relaxed but the work around was deemed not worth the effort, especially given the existence of better alternative data types such as `std::vector` and `boost::array`.

2.2 General Limitations

2.2.1 Macro Parameters Containing Commas

The preprocessor treats commas appearing within macro parameters as delimiters; hence they are to be avoided in type expressions and wrapped with an extra set of parenthesis when used with value expressions. The former is necessary because there is no known way to extract the user specified type. The latter is necessary to prevent the comma from acting as a delimiter in the preprocessor and, hence, making it appear as if the function-like macro had extra parameter(s).

2.2.2 Local Function Overloading

Local functions may not be overloaded.

2.2.3 Local Function Variables

Local function variables must be valid at the time of any invocation of their associated local function instance. This means that if any local function variable references any non-local function object, that object must be valid at the time of the associated local function invocation.

Implicitly typed local function variables may not be named "ref".

2.2.4 Local Function Definitions

Local functions are ultimately implemented using local class functions, hence any limitations imposed by the language on the latter will also carry over to the former.

3 Grammar

3.1 Special Symbols

a) [...]

Items enclosed in square brackets denote optional grammar entries.

b) `return params type varbind varset const ref`

These tokens are to be regarded as non-punctuation terminals.

3.2 EBNF

```
lclfunction-usage ::=
  lclfunction-start-macro ( name , lclfunction-spec )
  function-body
  WG_LCLFUNCTION_END [;]

lclfunction-start-macro ::=
    WG_LCLFUNCTION
  | WG_LCLFUNCTION_TPL

lclfunction-spec ::=
    void
  | [return nlt-return-tuple]
    [params param-seq]
    [varbind varbind-seq]
    [varset varset-seq]

  | [return nlt-return-tuple]
    [params param-list](V)
    [varbind varbind-list](V)
    [varset varset-seq]

nlt-return-tuple ::=
    ( non-local-type )

param-seq ::=
    ( parameter-declaration )
  | param-seq ( parameter-declaration )

param-list ::=
     parameter-declaration
  |  param-list ,  parameter-declaration

varbind-seq ::= nlt-bound-tuple-seq

varbind-list ::=
     nlt-bound-var-dcln
  |  varbind-list , nlt-bound-var-dcln

varset-seq ::= nlt-set-tuple-seq

nlt-bound-tuple-seq ::=
    nlt-bound-tuple
```

```
  | nlt-bound-tuple-seq nlt-bound-tuple

nlt-set-tuple-seq ::=
    nlt-set-tuple
  | nlt-set-tuple-seq nlt-set-tuple

nlt-bound-tuple ::=
  ( nlt-bound-var-dcln )

nlt-set-tuple ::=
  ( nlt-set-var-dcln )

nlt-bound-var-dcln ::= nlt-type-var-dcln

nlt-set-var-dcln ::= nlt-type-var-dcln , value-expr

nlt-type-var-dcln ::=
    implicit-non-local-type-var-dcln
  | explicit-non-local-type-var-dcln

implicit-non-local-type-var-dcln ::=
  implicit-type-var-dcln

explicit-non-local-type-var-dcln ::=
  explicit-non-local-type var-name

implicit-type-var-dcln ::=
  implicit-type var-name

implicit-type ::= lib-type-qualifiers | empty-string

lib-type-qualifiers ::= const | ref | const ref

explicit-non-local-type ::=
  type( non-local-type-specifier )

(V) Requires a variadic macro supported preprocessor.
function-body ::= See C++ standard.
var-name ::= A C++ variable name.
non-local-type-specifier ::=
  A type-specifier that specifies a non-local type.
type-specifier ::= See C++ standard.
value-expr ::= A C++ expression evaluating to some value.
```

4 Design Rationale

4.1 Preprocessor Limitations

4.1.1 Variadic Macro Support and Multi-Input User Specification

Depending on preprocessor support, multi-input specification may take the form of a single n-tuple, or a sequence of n 1-tuples, where the former is allowed only if variadic macros are supported by the preprocessor. The reasoning for this is that an n-tuple is easier to read and type than a sequence of n 1-tuples. For example:

```
params (int x, int y, std::string label)
```

is assumed to be preferable to:

```
params (int x) (int y) (std::string label)
```

Variadic macro support is not implemented for set variables because in this author's opinion it obfuscates the syntax. This is because set variables come in pairs and any use of variadic macros will flatten the pairs into one comma delimited list of tokens; and this author argues that reading set variables in 2-tuple pairs is more natural than reading them flattened out.

4.2 params Named Parameter Specification Format

The specification for the named parameter `params` needs to be PP parseable because the codegen needs to construct an expanded parameter list from those entries. That is, we need to be able to insert a comma before and after the full list of those entries. As a result, if variadic macros are not supported then `params` must be specified as a sequence of 1-tuples.

4.3 Forwarding Strategies of params Arguments

Any code generated for `WG_PP_LCLFUNCTION` must involve a two step function call where an external function forwards any of the arguments of "params" to an internal function that encapsulates the user defined scope of this macro. The reason for this is to get around the C++03 rule of local classes not being usable in template parameters. In this section we discuss the forwarding strategy used, and another alternative using slightly different syntax for param-tuple arguments of `params`.

For the rest of this this discussion will use variants of the following example:

```
WG_PP_LCLFUNCTION(func, params (...) )
```

which, without loss of generality, we can assume will generate the following code:

```
void interface_func(...);

void impl_func(...);
```

where `interface_func` is the external function that merely forwards all of its parameters to `impl_func`. It is important to note that:

a) in the generation of `interface_func` only TMP tools are available for use, and

b) in the generation of `impl_func` both PPMP and TMP tools are available for use.

Additionally, we don't have much leeway in generating `interface_func` because its signature and return type must be semantically equivalent to the one specified by `WG_PP_LCLFUNCTION` macro; this is because, as its name implies, `interface_func` is the client facing interface to the user specified and defined local function. (If this were not the case, then dispatching `interface_func` based on its signature in TMP code may result in surprising and unexpected behavior.)

4.3.1 Current Forwarding Strategy

The current forwarding strategy for the current syntax of `params` is best illustrated using the following example:

```
WG_PP_LCLFUNCTION(func, params (big_type const x) )
```

will generate:

```
void interface_func(big_type x);

void impl_func(big_type const x);
```

where the body of `interface_func` just calls impl_func(x)

The rationale for `interface_func`'s signature are:

a) Because the token "`big_type const x`" is not PP parseable, only TMP techniques are left to deduce its type. And the only relevant tool available is `function_types::parameter_types`, where the latter can and/or does strip off the top most const qualifier. (Remember, BOOST_TYPEOF is by default not used unless explicitly specified by the user, and in this case is certainly not needed.) Thus, the top-most const qualification of any specified local function parameter argument type is subject to being lost.

b) C++ semantic equivalency with the signature specified by the user. Even though the top-most const qualification of any parameter argument type maybe stripped off for `interface_func`, its signature and the users specified signature are still equivalent under C++ rules.

The rationale for `impl_func`'s signature is:

a) simplicity.

b) codegen limitations. Since the token "`big_type const x`" is not PP parseable, there is no way to correctly optimize it's forwarding in the said signature. This is because TMP tools will have to be used to deduce the type of the token, and any such use will result in the top most const qualifier being lost. Which in the context of `impl_func`'s user defined scope will result in an incorrectly typed parameter potentially being used. (That is `x`'s compiler enforced immutability should not be lost in the user defined local function scope.)

c) compiler optimizations. It is assumed the compiler will optimize away the extra copy at `impl_func`'s call site in `interface_func`'s definition since the latter's code just literally consists of forwarding parameter arguments to the former.

4.3.2 Alternative Forwarding Strategy with Alternative `param-tuple` Syntax

It is possible to efficiently forward `interface_func`'s parameter arguments to `impl_func` if the former's parameter types were exactly deducible. To this end we could modify the `param-tuple` syntax to delimit the parameter type from the parameter name along the lines of the following:

```
WG_PP_LCLFUNCTION(func, params (type(big_type const) x) )
```

This will allow us to generate:

```
void interface_func(big_type const x);

void impl_func(big_type const & x);
```

which correctly and efficiently forwards `interface_func`'s parameter arguments to impl_func. (This is possible because during the preprocessing phase we are able to parse the exact type of `x` and, hence, manipulate it accordingly when generating code.)

This syntax was considered and implemented at one point but was considered too unwieldy and not worth the unproven benefits of efficient forwarding.

4.4 `this_ const` Qualified Bound Function Variables

The `const` qualification of an implicitly typed `this_` bound function variable applies to the pointed-to-

type of `this_` because its type, and hence the type of `this`, are already `const` qualified. Not only would it be redundant, but it would also be impossible to `const` qualify the pointed-to-type of an implicitly typed `this_` had the constness been applied to the pointer type.

5 Tutorial and Examples

5.1 Params

The following is a local function with no internal state nor with any internal variables:

```
int force = 0;
int const mass = 10;
int const velocity = 2;

WG_LCLFUNCTION
(calculateForce,
  params (int & force, int const mass, int const velocity) )
{
  force = mass * velocity;
}
WG_LCLFUNCTION_END

calculateForce(force, mass, velocity);
EXPECT_EQ(20, force);
```

5.2 Varbind

The following is a local function with all its internal variables bound:

```
int force = 0;
int const mass = 10;
int const velocity = 2;

WG_LCLFUNCTION
(calculateForce, varbind (ref force, const mass, const velocity) )
{
}
WG_LCLFUNCTION_END;

calculateForce();
EXPECT_EQ(20, force);
```

5.3 Varset

The following is a local function with its internal variable set by the user:

```
WG_LCLFUNCTION(accumulate, return(int) varset (sum, 0) )
{
  return ++sum;
```

```
}
WG_LCLFUNCTION_END;

int sum = accumulate();
EXPECT_EQ(1, sum);
```

6 Future Features

6.1 Auto Assign Syntax

Consider support for the following syntax:

```
auto accum =
  WG_PP_LCLFUNCTION(return(int) params(int val) varset(tally, 0))
  { ... }
  WG_PP_LCLFUNCTION_END
```

6.2 `make_ref/make_cref`

Creates a reference wrapper for local function objects for use with STL algorithms were the user wants to ensure reference semantics for any and all copies.

7 Open Issues

7.1 Exception Specifications

Does it make sense to support exception specifications, especially in light of the fact that the user defined local function will always be called through a global proxy type whose function operator's exception specification can never be fully generalized. For example, even if the local function supported `noexcept(blah)`, what good will that do if its proxy's function operator did not support that?

8 Alternatives

8.1 Boost.Local_Function

8.1.1 Differences:

1. This library allows the function name to be specified at and only at the top of the local function declaration.
2. This library allows local functions to maintain internal state without referencing external variables (via varset).
3. This library allows the use of globally scoped types as macro parameters, i.e., `::std::pair<..., ...>`.
4. This library uses a different syntax.
5. This library does not have default parameters, unlike Boost::LocalFunction. In the author's view it adds too little expressivity to be worth the effort to implement.
6. This library does not allow the keyword `this` to be misused in a local function definition.
7. This library does not require any special syntax to make a local function recursive.
8. Same line multiple definitions of local functions do not require any special handling.
9. This library does not support exception specifications. For reasons why see the Open Issues

Section.

10. This library does not support the `inline` keyword. In the author's view this is redundant, any local function will automatically be defined within a class definition, hence, it would already be implicitly inline.
11. In the author's humble opinion this library has a much cleaner implementation.
    1. The preprocessor translator is partitioned into a front end and back end.
    2. A well-defined symbol table bridges the two translation layers.
    3. It has well-defined error-generating mechanism with a separate error pass before code generation.

9 Acknowledgments

9.1 Boost.Preprocessor

This library would not have been possible without the invaluable tools of Boost.Preprocessor.

9.2 Boost.Local_Function

Boost.Local_Function was invaluable in driving the specifications of this library.