C/C++ Dictionary Node Properties

Right-click in the Rule/Results panel of the RuleWizard GUI to access the node properties. Not all node property options will appear in the shortcut menu; only those options that are viable for the parent node in question will be available. When the C/C++ Node Dictionary is chosen in the RuleWizard GUI (**Nodes> Node Dictionary> C,C++**) the following node property options described in this chapter are available:

- Arguments
- AttributeArgumentKind
- AttributeFamily
- Attributes
- BitFieldSize
- BaseType
- Body
- ByReference
- Called For
- Captures
- CapturedVariable
- Catch
- CharacterCount
- Column
- Column-End
- Condition
- Constructor initialization list
- Count
- Context
- DeclaredTypeTraverseReference
- DeclaredWithAutoTypeSpecifier
- DeclaredWithDecltypeAuto
- DefaultValue
- DeleteDeclaration
- DerivationPaths
- DerivationSteps
- Dimension
- DllExport
- DllImport
- Else
- Entity
- FalseChoice
- FieldDesignator
- File
- Filename
- Filter
- Function
- Group
- HasAssocVlaDimension
- HasBracedInitializer
- HasCaptureDefaultByReference
- HasCaptureDefaultByValue
- HasDeducedReturnType
- HasDefaultValue
- HasDirectInitializer
- HasEllipsis
- HasElse
- HasExplicitBaseType
- HasExplicitInitializer
- HasExplicitReturnType
- HasForwardDecl
- HasMulipleParents
- HasNoExceptSpecifier
- HasParameterDecl
- HasStaticInitializer
- HasTrailingReturnType
- HasVirtualFunctionsIncludingInBaseClasses
- HasVoid
- Implementation
- Index
- IndexDesignator
- InheritanceInfo
- InitializationOrderIndex
- Initializer
- InitializerInClass
- Increment
- Init
- IsADLCall

Arguments

For function calls, this property returns Argument nodes that represents the arguments. Example:

```
foo(x+y); // Returns Argument node that represents the x+y
```

For attributes, this property returns Attribute Argument nodes that represent the attribute argument. Example:

```
void f() __attribute__ ((interrupt("IRQ"))); // Returns Attribute Argument
for the interrupt("IRQ")
```

AttributeArgumentKind

The node that represents the kind of Attribute Argument to find. The possible choices are empty, constant, raw token, token, and type. Example:

```
void f1(char*) __attribute__((nonnull())); // AttributeArgumentKind: empty
```

AttributeFamily

The node that represents the family of Attribute to find. The possible choices are GNU, Microsoft, Cosmic, and custom. Example:

```
int x __attribute__ ((unused)); // AttributeFamily: GNU
```

Attributes

The nodes that represent the attributes. Example:

```
// For Global Variable the Attributes property returns Attribute
// node which describes the "__attribute__ ((unused))" attribute:
int x __attribute__ ((unused))
```

BitFieldSize

Represents value of bit field size, such as class. Example:

```
{
int x:5; //bitfieldsize is 5
}
```

BaseType

For Enum nodes with HasExplicitBaseType set as true, returns a node that represents the enum base type. Example:

```
typedef long LONG;
enum struct A : long { A1, A2 }; // BaseType is node representing 'long'
type
enum struct B : LONG { B1, B2 }; // BaseType is node representing 'LONG'
type
```

Body

The set of nodes contained within the body of the current node. Choosing Body allows you to create a rule condition about the code element that is a subnode of the parent node.

For example, the body of Node A returns a “body” that is A’s subnode; the exact definition of the “body” depends on the node itself.

ByReference

Returns true if the variable is captured by reference. Example:

```
[&a]() { return a; }; //ByReference=T
[a]() { return a; }; //ByReference=F
```

Called For

Expression that represents an object for which a function is called.

Captures

A set of nodes that represents entities captured explicitly and implicitly.

CapturedVariable

A parameter, 'this' or a local variable that is captured by lambda capture. Empty for capture-inits. Example:

```
[a]() { return a; }; // CapturedVariable: a
[this]() { return this->a; }; // CapturedVariable: this
[x = a]() { return x; }; // No CapturedVariable
```

Catch

The node that represents the catch part of the try-catch statement.

CharacterCount

Returns the number of characters in a String Constant including implicit \0 at the end. Example:

```
const char *b = "bb"; // CharacterCount: 3
```

Column

Returns the column number where the node begins in the source code.

Column-End

For Statements, returns the column number where the node ends in the source code.

Condition

The node(s) that represents the contents of the conditional statement. Allows you to create a rule condition about the parent node’s statement (if, else, init, increment, for, with, while, switch, and do while).

- IsAlias
- IsArrayDelete
- IsAtomic
- IsAutonomousPrimaryTagDecl
- IsBitfield
- IsBraceNotationCast
- IsBuiltin
- IsCompilerGenerated
- IsConst
- IsConstexpr
- IsConstructor
- IsConstructorInit
- IsConversionOperator
- IsCopyAssignmentOperator
- IsCopyConstructor
- IsDecl
- IsDeclInCommaList
- IsDeducedAuto
- IsDeducedDecltypeAuto
- IsDefaultConstructor
- IsDefaultType
- IsDefaultValue
- IsDefaulted
- IsDeleted
- IsDereference
- IsDestructor
- IsDirect
- IsExplicit
- IsExplicitExtern
- IsExplicitFinal
- IsExplicitOverride
- IsExplicitSigned
- IsExplicitStatic
- IsExplicitUnsigned
- IsExplicitVirtual
- IsExplicitVoidParam
- IsExtern
- IsFinal
- IsGeneric
- IsImplementation
- IsImplicit
- IsInitCapture
- IsInline
- IsInProject
- IsLiteralType
- IsMainSource
- IsMisraC2012CompositeExpression
- IsMisraCpp2008PODType
- IsMoveAssignmentOperator
- IsMoveConstructor
- IsMutable
- IsNoReturn
- IsOperator
- IsOverride
- IsPerfectForwarding
- IsPureVirtual
- IsRegister
- IsRValueReference
- IsScopedEnum
- IsSealed
- IsSigned
- IsStatic
- IsTemplate
- IsTemplateInstance
- IsTemplateType
- IsThreadLocal
- IsThrowAnyExtension
- IsUnsigned
- IsValueInitialized
- IsVariableLengthArray
- IsVirtual
- IsVolatile
- Iterator
- Kind
- Label
- Left Hand Side
- Line
- LineCount

Constructor initialization list

A set of nodes that represents the initialization list.

Count

This property allows you to create a rule condition that restricts a node's quantity.

1. Create a collector
2. Right-click the collector and choose Count

The collector keeps track of the number of times a pattern is found. Count places a condition on what number of instances constitutes a rule violation. For information on determining exactly how counts are calculated, see Working with Node Sets.

Context

This property enables you to create a rule condition around the code element that contains the parent node. The context of Node A returns the node that contains Node A. For example, if you wanted to create a rule that said "always put X inside of Y," you would create a parent node for X, use Context to attach Y, create a collector, then use Count to specify that a count of `$$==0` constitutes a violation.

Some rules can only use Body, while some can only use Context. Some cannot use either and some can use both. If you cannot determine which property to use, choose Body because it will result in better performance than Context. Body and Context are inverse operations in many instances. For example, an expression can be in the context of a statement, but be contained in the condition (rather than the body) of the statement.

DeclaredTypeTraverseReference

When used in 'Direct Check' mode, this property returns the declared type of an entity, just as the 'Type' property does. In 'Indirect Check' mode, the property continues traversal instead of stopping at reference.

DeclaredWithAutoTypeSpecifier

Returns true for variables that are declared with an auto specifier. Example:

```
auto i = 1; // DeclaredWithAutoTypeSpecifier is true
int j = 1; // DeclaredWithAutoTypeSpecifier is false
```

DeclaredWithDecltypeAuto

Returns true for variables that are declared with a decltype(auto) specifier. Example:

```
decltype(auto) i = 1; // DeclaredWithDecltypeAuto is true
int j = 1; // DeclaredWithDecltypeAuto is false
```

DefaultValue

The node that represents the default value of the function parameter. Example:

```
void func(int x=5); //DefaultValue is 5
```

DeleteDeclaration

A node that represents the declaration of a delete operator that is called.

DerivationPaths

For the BaseType node, this property returns a set of BaseClassDerivation nodes. Each node represents a possible inheritance path that leads from the parent class represented by BaseType to the child class. Example:

- Line-End
- Linkage
- MangledName
- MisraC2012EssentialType
- MisraCpp2008UnderlyingType
- NewDeclaration
- NoExceptConstant
- Operand
- OperandsReversed
- OperatorName
- Overrides
- Parameter
- ParamNumber
- Parent
- PartialSpecTemplateArgument List
- Permission
- Private Parent
- Protected Parent
- Public Parent
- RangeExpression
- ReferenceQualifier
- Repeat
- Representation
- Result
- Return type
- ReturnTypeTraverseReference
- Right Hand Side
- ScopePrefix
- SequenceNumber
- Size
- StartLine
- TemplateArgumentList
- TemplateDeclaration
- Template Parameters
- Then
- Throws
- Traverse
- TrueChoice
- Type
- TypeTraverseReference
- UnderlyingElementType
- UsesDesignatedInitializer
- UsesDesignatedInitializers
- UsesOperatorSyntax
- Value
- Virtual Parent
- __condition
- __increment
- __range
- __begin
- __end

```
class C { };
class D : public C { };
class Final : public D, public C { };
```

For the `Final` class InheritanceInfo property, returns three BaseType nodes—one for D class, and two for C class (because C is inherited twice).

- For one of C BaseType nodes, the inheritance path represents the "C->D" path.
- For the other C BaseType node, the inheritance path represents the "C" path. For this inheritance path, the IsDirect property returns true.
- For the D BaseType node, the inheritance path represents the "D" path.

DerivationSteps

For BaseClassDerivation node, returns a set of BaseType nodes that this inheritance path contains. Example:

```
class C { };
class D : public C { };
class Final : public D, public C { };
```

For `Final` class, there are two BaseClassDerivation of class C: one direct and one through class D.

The direct one only contains step C. The one through class D contains steps "C" and "D".

Also see DerivationPaths.

Dimension

The number of elements in the array.

DllExport

Indicates whether function, data, or object is exported to a DLL.

DllImport

Indicates whether function, data, or object is imported from a DLL.

Else

The node that represents the 'else' part of an if-else statement.

Entity

Node or set of nodes contained within the body of the current node. Choosing Entity allows to create a rule condition for the code element that is a subnode of the parent node.

This property can be used for matching function declarations that point to the same function, for example, for each of the following pieces of code:

```
void foo (int i);
void foo (int j);
void foo (int k);
```

"Entity" will return the same node for all foo functions.

FalseChoice

The node that represents the statement processed for a false condition in a '?:' statement.

FieldDesignator

For struct aggregate initializer elements with designated initializers, returns field names. Example:

```
struct S {
int a;
int b;
int c;
};


struct S s = {
.a = 0, // FieldDesignator: "a"
.c = -1 // FieldDesignator: "c"
};
```

File

Returns source file node for the current node.

Filename

The source code filename of the node.

Filter

Returns the same, unchanged node that is passed to it. This allows to select only some nodes from a set of nodes, or to save nodes in a collector for further processing.

Function

A member function that represents the body of a lambda function.

Group

The nodes that represent the attribute group. Example:

```
void fx1() { /* ... */; }
void fx2() __attribute__((weak, alias("fx1")));// Both attributes have the
same Attribute Group
```

HasAssocVlaDimension

Returns `true` if type is variable length array which has associated size expression. For example:

```
char tab[strlen(msg)]; // Type is Array with HasAssocVlaDimension = T
void foo(int tab[*]); // Type is Array with HasAssocVlaDimension = F
```

HasBracedInitializer

Returns true if a global, local, or member variable is initialized with a braced initializer. For example:

```
int x{1}; // HasBracedInitializer=T
int y(1); // HasBracedInitializer=F
int z = 1; // HasBracedInitializer=F
```

HasCaptureDefaultByReference

Returns true if the capture-default of the lambda is &. Example:

```
[&](){ return a; }; //HasCaptureDefaultByReference=T
[&a](){ return a; }; //HasCaptureDefaultByReference=F
```

HasCaptureDefaultByValue

Returns true if the capture-default of the lambda is =. Example:

```
[=](){ return a; }; //HasCaptureDefaultByValue=T
[a](){ return a; }; //HasCaptureDefaultByValue=F
```

HasDeducedReturnType

Returns true for functions with the deduced return type. Example:

```
auto f() { // HasDeducedReturnType is true
return 0;
}
```

HasDefaultValue

Returns `true` for routine parameters with a default value. Example:

```
void foo(int a = 10)
{
}
```

HasDirectInitializer

Returns `true` if a global, local, or member variable is initialized with a braced or parenthesized initializer that immediately follows the declarator. For example:

```
int x(1); // HasDirectInitializer = T
int y = 1; // HasDirectInitializer = F
```

HasEllipsis

Returns `true` if ellipses (...) is used in function definition or in a catch handler. Example:

```
void func(int x, ...) { //HasEllipsis = T
try { }
catch(...) {} //HasEllipsis = T
}
```

HasElse

Returns true when `if` statement contains an `else` condition.

HasExplicitBaseType

For Enum nodes, returns true if the enum base type was explicitly specified. Example:

```
enum class A { A1, A2 }; // HasExplicitBaseType is false
enum B { B1, B2 }; // HasExplicitBaseType is false
typedef long LONG;
enum struct C : long { C1, C2 }; // HasExplicitBaseType is true
enum struct D : LONG { D1, D2 }; // HasExplicitBaseType is true
```

HasExplicitInitializer

For enum Constant, this is the expression used to initialize the constant. Example:

```
enum E {
    ENUM_CONSTANT1 = 22*2,  // Initializer: "a+b" node
    ENUM_CONSTANT2          // Initializer returns empty set
  }
```

HasExplicitReturnType

Returns true if the lambda contains a return type specified explicitly in source code. Example:

```
[]() -> int{ return 0; }; //HasExplicitReturnType=T
[](){ return 0; }; //HasExplicitReturnType=F
```

HasForwardDecl

Returns `true` if a forward declaration exists that corresponds with the current class.

HasMuliple Parents

Returns `true` if the class uses multiple inheritance.

HasNoExceptSpecifier

Returns true if the function type was declared with the 'noexcept' specifier. Example:

```
void f() noexept; // HasNoExceptSpecifier is true
typedef void (*pf)() noexept; // HasNoExceptSpecifier is true
```

HasParameterDecl

Returns true if the lambda contains a parameter declaration specified explicitly in source code. Example:

```
[](){ return 0; }; //HasParameterDecl=T
[]{ return 0; }; //HasParameterDecl=F
```

HasStaticInitializer

Returns `true` for global, static local, and static member variables if the variable has static initialization to a constant. This includes initializing a 'constexpr' variable to a 'constexpr' value.

This kind of initialization is evaluated during compilation and doesn't produce any code.

```
struct A {
    int a1;
    int a2;
};

struct B {
    B(int, int);
    constexpr B(int ab1, int ab2, int): b1(ab1), b2(ab2) {};
    int b1;
    int b2;
};

A a = { 1, 1 }; // HasStaticInitializer = true
B b = { 1, 1 }; // HasStaticInitializer = false, constructor is called
constexpr B c = { 1, 1, 1 }; // HasStaticInitializer = true, constant
expression
```

HasTrailingReturnType

For function types, returns true if the declaration includes the trailing return type. Example:

```
auto f() -> int { // HasTrailingReturnType is true
return 0;
}
```

HasVirtualFunctionsIncludingInBaseClasses

Returns `true` if one or more member functions declared in the class or struct or its base classes is a virtual function. Example:

```
class P1 {                  // class P1
HasVirtualFunctionsIncludingInBaseClasses
    virtual void foo1();
};
class P2 : public P1 {      // class P2
HasVirtualFunctionsIncludingInBaseClasses
    void foo2();
};
class C : public P2 {       // class C
HasVirtualFunctionsIncludingInBaseClasses
    void foo3();
};
```

HasVoid

Indicates true if function declaration/definition uses "void" (or typedef to void) as its argument word. Example:

```
int func(void); //HasVoid=T
typedef void TD_void;
int func(TD_void); //HasVoid=T
```

Implementation

Points to a function declaration that contains a body. Example:

```
1 void foo();
2 void foo() {
3 };
```

The implementation node taken from the function declaration in line 1 points to the function declaration in line 2. The implementation taken from line 2 will point back to line 2.

Index

For case conditions, returns a node that represents a value used as `case` condition. Example:

```
int x;
switch (x) {
  case 0:   // Index is 0
  break;
}
```

For other nodes (call arguments, template argument, constructor initializer), returns index of the argument /initializer. Example:

```
void foo(int x, int y) {
 foo(y,  // Argument y has Index 0
     x); // Argument x has Index 1
}
```

IndexDesignator

For array aggregate initializer elements with designated initializers, returns an array index. Example:

```
int a[] = {
[0] = 1, // IndexDesignator: 0
[5] = 2 // IndexDesignator: 5
};
```

InheritanceInfo

Returns nodes that represent all the base classes—both directly and indirectly inherited.

InitializationOrderIndex

Returns index of an element from constructor initializer list using initialization order. Example:

```
class Test{ public:
 Test(): var0(0),   // InitializationOrderIndex: 1
         var1(1) {} // InitializationOrderIndex: 0
 int var1;
 int var0;
};
```

Initializer

The statement on the right side of an assignment (=) operator or on the right side of a parameter (for example, a constructor, such as initialization). Example:

```
int i = 1; // 1 is initializer
int j(2); // 2 is initializer
```

For `enum Constant`, this is the expression used to initialize the constant. Example:

```
enum E {
    ENUM_CONSTANT1 = 22*2,  // Initializer: "a+b" node
    ENUM_CONSTANT2          // Initializer returns empty set
  }
```

InitializerInClass

This property is `true` if the field (static or not-static) is initialized in the class/struct body. For example (C++11):

```
struct S {
    int a;         // InitializerInClass = false
    int b = 0;   // InitializerInClass = true
};
```

Increment

The node that represents the increment statement in a `for` statement.

Init

The node that represents the initialization statement in a `for` statement.

IsADLCall

Returns `true` for calls that use argument dependent lookup to find the called routine. Example:

```
namespace Ns
{
    struct S {};
    void f(S);
}
void f(int);
void m()
{
    Ns::S s;
    f(s); // IsADLCall = true
    f(0); // IsADLCall = false
    ::f(0); // IsADLCall = false
    (f)(0); // IsADLCall = false
    void (*F)(int) = f; // no call here
}
```

IsAlias

For the Typedef node, returns true if it represents an alias. Example:

```
using ALIAS_INT = int; // IsAlias is true
typedef int TYPEDEF_INT; // IsAlias is false
```

IsArrayDelete

Returns `true` if delete form is used.

IsAtomic

Returns `true` if the current declaration is declared as `_Atomic` (C11). Example:

```
_Atomic int ai1; /* IsAtomic is true */
_Atomic(int) ai2; /* IsAtomic is true */
```

IsAutonomousPrimaryTagDecl

Returns `true` if the type was declared separately. Example:

```
struct S1 { // IsAutonomousPrimaryTagDecl == false
   [...]
} *p;
struct S2 { // IsAutonomousPrimaryTagDecl == true
   [...]
};
```

IsBitfield

Checks if a given identifier is a C++ bitfield. Example:

```
struct BitFields {
int b1 : 2; //b1 is bitfield
int b2 : 4; //b2 is bitfield
int b3;     //b3 is not a bitfield
}
```

IsBraceNotationCast

Returns `true` if the type of functional cast used is brace notation.

```
int a = 10;
int b = int{a};    // IsBraceNotationCast = true
int c = int(a);    // IsBraceNotationCast = false
```

IsBuiltin

Returns `true` if the function or type is a compiler built-in function or type. Example for a gcc compiler:

```
int main() {
    __builtin__exit(111);  /* IsBuiltin = true for __builtin__exit
function node */
}
```

IsCompilerGenerated

Returns `true` if code is generated by compiler (not explicitly written by user).

IsConst

Returns `true` if the current declaration was declared as a `const`.

IsConstexpr

Returns true for variables or functions explicitly declared as constexpr. Example:

```
constexpr int MAX_SIZE = 1024;

constexpr int add(int a, int b)
{
return a + b;
}
```

IsConstructor

Returns `true` if the declared function is a constructor. Example:

```
struct A {
A(); // IsConstructor = T
A(int); // IsConstructor = T
A(A& a); // IsConstructor = T
};
```

IsConstructorInit

Returns `true` if variable is initialized in constructor-like style. For example:

```
int i = 0; //IsConstructorInit == false
int j(0);  //IsConstructorInit == true
```

IsConversionOperator

Returns `true` if declared function is conversion operator. Example:

```
class example {
public:
  operator int();  //IsConversionOperator=T
}
```

IsCopyAssignmentOperator

Returns `true` if the declared function is a copy assignment operator. Example:

```
struct A {
A& operator=(A& other); // IsCopyAssignmentOperator= T
};
```

IsCopyConstructor

Returns `true` if the declared function is a copy constructor. Example:

```
struct A {
A(A& a); // IsCopyConstructor = T
};
```

IsDecl

Returns `true` if the node is a declaration.

IsDeclInCommaList

Returns `true` for variables declared in declarations that contain several comma-separated declarators. Example:

```
int a, b, c;
```

IsDeducedAuto

For types, returns `true` if the type was deduced with an auto specifier. Example:

```
auto f() { // IsDeducedAuto is true for return type of 'f'.
return 0;
}
```

IsDeducedDecltypeAuto

For types, returns `true` if the type was deduced with a `decltype(auto)` specifier. Example:

```
decltype(auto) f() { // IsDeducedDecltypeAuto is true for return type of
'f'.
return 0;
}
```

IsDefaultConstructor

If constructor can be called without any arguments it is considered the default.

IsDefaultType

Set when function return value is not given explicitly. The compiler usually assumes that a function returns int.

IsDefaultValue

Returns `true` if an argument of a function call is generated from a default value. Example:

```
void foo(int x = 0, int y = 0);
void bar() {
    foo(1); // IsDefaultValue = false for argument x, IsDefaultValue =
true for argument y
}
```

IsDefaulted

Returns `true` if a function is declared with the `=default` syntax. Example (C++11):

```
class A {
 public:
 A& operator=(const A&) = default; // IsDefaulted = T
};
```

IsDeleted

Returns `true` if a function is declared with the `= delete` syntax. Example (C++11):

```
class A {
public:
    A& operator=(const A&) = delete;   // IsDeleted = T
};
```

IsDereference

Returns `true` if the vacuous destructor call includes a dereference form. Example:

```
typedef int INT;
int i, *p;

i.INT::~INT(); // IsDereference = false
p->INT::~INT(); // IsDereference = true
```

IsDestructor

Returns `true` if the declared function is a destructor. Example:

```
struct A {
~A(); // IsDestructor = T
};
```

IsDirect

For BaseClassDerivation node, returns true if the associated base class is a direct base class as a result of this derivation. Example:

```
class C { };
   class D : public C { };
   class Final : public D, public C { };
```

For `Final` class, there are two `BaseClassDerivation` of class `C`: one direct and one through class `D`. For the direct one, the `IsDirect` property returns `true`. See also DerivationPaths property.

IsExplicit

A constructor that is declared using an explicit keyword. Such a constructor cannot be used for implicit conversions. For `&a` node, returns `true` if the ampersand was explicit in source.

IsExplicitExtern

A variable or a function that is declared with the `extern` storage class specifier.

IsExplicitFinal

Returns `true` if the function or class was declared with the `final` specifier. Example:

```
struct A {
    virtual void f();
};
struct B final: A // IsExplicitFinal = true for struct B
{
    void f() final; // IsExplicitFinal = true for function B::f
};
```

IsExplicitOverride

Returns `true` if a virtual member function is declared with the function-modifier `override`. Example (C++11):

```
struct A {
    virtual void foo();
};

struct B : A {
    void foo() override ;   // IsExplicitOverride = T
};
```

IsExplicitSigned

Returns `true` if the current item was explicitly declared signed. Example:

```
char c;           // IsExplicitSigned == false
unsigned char uc; // IsExplicitSigned == false
signed char sc;   // IsExplicitSigned == true
int i;            // IsExplicitSigned == false
signed int si;    // IsExplicitSigned == true
unsigned int ui;  // IsExplicitSigned == false
```

IsExplicitStatic

A variable or a function that is declared with the `static` storage class specifier.

IsExplicitUnsigned

Returns `true` if the current item was explicitly declared `unsigned`. Example:

```
char c;           // IsExplicitUnsigned == false
unsigned char uc; // IsExplicitUnsigned == true
signed char sc;   // IsExplicitUnsigned == false
int i;            // IsExplicitUnsigned == false
signed int si;    // IsExplicitUnsigned == false
unsigned int ui;  // IsExplicitUnsigned == true
```

IsExplicitVirtual

Returns `true` if member function is explicitly declared as `virtual`. For example:

```
class P {
    virtual void bar(); // IsExplicitVirtual=T
};
class C : public P {
    void bar();         // IsExplicitVirtual=F
};
```

IsExplicitVoidParam

Indicates `true` if function declaration/definition uses `void` (or `typedef` to void) as its argument word. Example:

```
int func(void); //IsExplicitVoidParam=T
typedef void TD_void;
int func(TD_void); //IsExplicitVoidParam=T
```

IsExtern

For functions and variables, a function or variable that is defined in another compilation unit.

IsFinal

Returns `true` if the function or class was declared with the `final` or `sealed` specifier. Example:

```
struct A {
    virtual void f();
};
struct B final: A // IsExplicitFinal = true and IsFinal = true for struct B
{
    void f() sealed; // IsExplicitFinal = true and IsSealed for function
B::f
};
```

IsGeneric

Returns `true` if the lambda is a generic lambda. Example:

```
[](auto a){ return a; }; //IsGeneric=T
[](int a){ return a; }; //IsGeneric=F
```

IsImplementation

Returns `true` if the node represents implementation code.

IsImplicit

Returns `true` if the captured variable is not specified explicitly on the capture list but is captured through the capture-default. Example:

```
[&](){ return a; }; //IsImplicit=T
[&a](){ return a; }; //IsImplicit=F
```

IsInitCapture

Returns `true` if the entity is captured with the capture-init form. Example:

```
[x = a](){ return x; }; //IsInitCapture=T
[a](){ return a; }; //IsInitCapture=F
```

IsInline

Returns `true` if the function is declared to be inline.

IsInProject

Returns `true` if a file is in user project or if a file is visible as part of the current user project (depending on the configuration of the project)—for example, system headers are typically outside of the user project.

IsLiteralType

Returns `true` for literal types. Example:

```
struct LiteralType
{
int x;
int y;
};
```

IsMainSource

Returns `true` if the file is the main analyzed source file.

IsMisraC2012CompositeExpression

Returns `true` if an expression is a composite expression, according to its definition in the MISRA C:2012 standard. For example:

```
void f(int i) {
    unsigned char uc1, uc2;
    uc1 + uc2;              // (a+b)  IsMisraC2012CompositeExpression: true
    (i > 1) ? uc1 : uc2;    // (a?b:c)IsMisraC2012CompositeExpression:
false
}
```

IsMisraCpp2008PODType

Returns `true` for classes, structures and unions if they are POD (Plain Old Data) types (according to the POD definition specified in the MISRA 2008 standard). Example:

```
class C1 { // IsMisraCpp2008PODType = T
public:
   int i;
};

class C2 { // IsMisraCpp2008PODType = F
private:
   int i;
};
```

IsMoveAssignmentOperator

Returns `true` if the declared function is a move assignment operator. Example (C++11):

```
struct A {
A& operator=(A&& other); // IsMoveAssignmentOperator = T
};
```

IsMoveConstructor

Returns `true` if the declared function is a move constructor. Example (C++11):

```
struct A {
A(A&& a); // IsMoveConstructor = T
};
```

IsMutable

Determines if a field within a class is declared to be mutable. Mutable is a C++ keyword that says that class members can be modified even if a class's object is declared constant.

IsNoReturn

Returns `true` if the function is declared with the `_Noexcept` specifier (in C) or with the `noreturn` attribute (in C++). Example:

```
_Noreturn int fc ( void );   // IsNoReturn is true
[[ noreturn ]] void fcpp();  // IsNoReturn is true
```

IsOperator

Returns `true` if the function is an operator.

IsOverride

Indicates whether function overrides function from base class. Example:

```
class Base {
public:
  void func();
}
class Test: Base {
  void func();  //IsOverride= T
}
```

IsPerfectForwarding

Returns `true` for functions whose arguments are all universal references. Example:

```
template <class T>
void f(T&& t) { // IsPerfectForwarding is true
...
}
```

IsPureVirtual

Indicates that the declaration is only an interface in the current scope.

IsRegister

Returns `true` if a variable is declared using the `register` keyword.

IsRValueReference

Returns `true` for reference types that are of rvalue reference type. Example:

```
void initialize(Collection&& c) {
...
}
```

IsScopedEnum

Returns `true` for Enum nodes if they are scoped enumerations. Example:

```
enum class ScopedEnumClass { A1, A2 }; // IsScopedEnum is true
enum struct ScopedEnumStruct { B1, B2 }; // IsScopedEnum is true
enum NonScopedEnum { C1, C2 }; // IsScopedEnum is false
```

IsSealed

Returns `true` if the function or class was declared with the 'sealed' specifier (a Microsoft extension). Example:

```
struct A {
    virtual void f();
};
struct B sealed: A // IsSealed = true for struct B
{
    void f() sealed; // IsSealed = true for function B::f
};
```

IsSigned

Returns `true` if the current item is signed. Example:

```
char c;           // IsSigned == true  (depends on compiler)
unsigned char uc; // IsSigned == false
signed char sc;   // IsSigned == true
int i;            // IsSigned == true
signed int si;    // IsSigned == true
unsigned int ui;  // IsSigned == false
```

IsStatic

Returns `true` if the given declaration was declared as static.

IsTemplate

Indicates a template declaration.

IsTemplateInstance

Returns `true` when object has been created from template class instantiation. Example:

```
template <class T>
class Base {
};

int main() {
    Base<int> s;   //s is variable type class with IsTemplateInstance = T
    return 0;
};
```

IsTemplateType

Returns `true` if function parameter is template type. Example:

```
template <class T>
class Base {
  void func(T x);  //x is template type parameter
};
```

IsThreadLocal

Returns `true` for variables declared with the `thread_local` (C++) or `_Thread_local` (C) keyword. Examples (C++11 and C11 respectively):

```
thread_local int value = 0; // IsThreadLocal is true
```

```
_Thread_local int value = 0; /* IsThreadLocal is true */
```

IsThrowAnyExtension

Returns `true` if the function type was declared with `noexcept(<false-expression>)` or `noexcept (<template-dependent-argument>)` or `throw(...) (microsoft extension)`. This indicates that any exception may be thrown.

IsUnsigned

Returns true if the current item is unsigned. Example:

```
char c;           // IsUnsigned == false (depends on compiler)
unsigned char uc; // IsUnsigned == true
signed char sc;   // IsUnsigned == false
int i;            // IsUnsigned == false
signed int si;    // IsUnsigned == false
unsigned int ui;  // IsUnsigned == true
```

IsValueInitialized

Returns `true` if a global, local, or member variable was value-initialized. Example:

```
int value_1; // IsValueInitialized is false
int value_2{}; // IsValueInitialized is true
int value_3 = {}; // IsValueInitialized is true
int value_4{0}; // IsValueInitialized is false
```

IsVariableLengthArray

Returns `true` if type is variable length array. For example:

```
char tab[strlen(msg)]; // Type is Array with IsVariableLengthArray = T
```

IsVirtual

Returns `true` if the current function was declared as virtual.

IsVolatile

Returns `true` if the current declaration was declared as a volatile.

Iterator

For range-based for statements, represents the iterator variable. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

Kind

For (C++) cast, the node that represents the kind of the cast to find. Here are the possible attributes (values);

```
// dynamic_cast example
a = dynamic_cast<A*>(new B());


// reinterpret_cast example
a = reinterpret_cast<A*>(new B());


// static_cast example
p = static_cast<int>(t);
```

For String Constant, the node that represents the kind of the string. Here are the possible attributes (values):

```
const char* ch = "ch_t";           /* char */
const char16_t* ch16 = u"ch_16t";  /* char16_t */
const char32_t* ch32 = U"ch_32t";  /* char32_t */
const wchar_t* wch = L"wch_t";     /* wchar_t */
```

Label

A source code label.

Left Hand Side

The node that represents the left side of a binary expression.

Line

Returns the line number where the node begins in the source code.

LineCount

Represents the number of all lines in a file.

Line-End

For Statements, returns the line number where the node ends in the source code.

Linkage

Linkage of a function or variable. Possible choices are:

- no linkage
- internal
- external c++
- external c

MangledName

This property returns the internal (mangled) name of entity. In C, the property returns the same value as the Name property. In C++, the value will be different than the value of the Name property. For example (C++):

```
void foo(); /* name: foo, mangled name: _Z3foov */
namespace N {
void foo(); /* name: foo, mangled: _ZN1N3fooEv */
}
```

MisraC2012EssentialType

Returns the essential type of the node, according to its definition in the MISRA C:2012 standard. For example:

```
void f() {
    unsigned char uc1, uc2;
    uc1 + uc2;              // (a+b) MisraC2012EssentialType: unsigned char
}
```

MisraCpp2008UnderlyingType

Returns the underlying type of the node, according to its definition in the MISRA C++:2008 standard. For example:

```
void f() {
    unsigned char uc1, uc2;
    uc1 + uc2;              // (a+b) MisraCpp2008UnderlyingType: unsigned
char
}
```

NewDeclaration

A node that represents the declaration of a new operator that is called.

NoExceptConstant

Returns expressions that are used with the 'noexcept' specifier. Example:

```
void f1() noexept; // NoExceptConstant is null
void f2() noexept(true); // NoExceptConstant points to expression 'true'
```

Operand

The node that represents the operand of the expression.

OperandsReversed

Returns true if operands of "a[b]" expressions have been reversed. Example:

```
void foo(int index) {
  int array[10];
  index[array]; /* OperandsReversed = True */
  array[index]; /* OperandsReversed = False */
}
```

OperatorName

The name of the operator currently being overloaded.

Overrides

A node that represents the function that is being overwritten.

Parameter

A node that represents the parameter of the given declaration.

ParamNumber

Position of the current parameter in the parameters list of the function declaration. Example:

```
void func(int x, int y); //position of y is 2
```

Parent

For Member Variable, this property represents class or structure of which it is a member.

For Class or Structure, this property represents classes or structures from which it analyzes and inherits class or structure.

For Functions, this property represents the following:

- Class, if function is a member of class
- Namespace, if function is a member of namespace
- Nothing, if function is in global scope.

PartialSpecTemplateArgumentList

Returns the arguments on which the instantiation is based, with respect to the template parameter list of the partial specialization. Example:

```
template<class T, class S, class R>
class List {};
template<class Q, class W>          // Instance of this template
instantiated below has
class List<Q**, W, Q> {};           //   PartialSpecTemplateArgumentList:
bool, int
List<bool**, int, bool> lxx;
```

Permission

A choice between the various access privileges.

Private Parent

Indicates that private inheritance was used.

Protected Parent

Indicates that protected inheritance was used.

Public Parent

Indicates that public inheritance was used.

RangeExpression

For range-based for statements, represents the range expression. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

ReferenceQualifier

This property represents 'ref-qualifier' for the Function type. You can use the following options:

- **default**. The function declared with no ref-qualifier.
- **lvalue.** The function declared with an lvalue ref-qualifier.
- **rvalue.** The function declared with an rvalue ref-qualifier.

Example:

```
class A1 {
public:
  void f1();    // ReferenceQualifier: default
  void f2() &;  // ReferenceQualifier: lvalue
  void f3() &&; // ReferenceQualifier: rvalue
};
```

Repeat

For array aggregate initializer elements with designated initializers, returns the number of repetitions of a designated initializer. Example:

```
int a[] = {
[0 ... 5] = 1, // IndexDesignator: 0, Repeat 6
[6] = 2 // IndexDesignator: 6, Repeat null/0
};
```

Representation

Represents the exact same form of the number as written in code. Example:

```
int x = 0x12;     //0x12 when value indicates 18
```

Result

For a generic selection, this property returns the expression selected at compile time, based on the type of the controlling expression. Example:

```
int is_int = _Generic(10, int : 1, default: 0); // Returns expression '1'
```

Return type

The return type of the current function.

ReturnTypeTraverseReference

In 'Direct Check' mode, this property returns the function return type of an entity--same as 'Type' property. In 'Indirect Check' mode, the property continues traversal instead of stopping at the reference.

Right Hand Side

The node that represents the right side of a binary expression.

ScopePrefix

Prefix of the scope resolution operator. Example:

```
namespace N {
  void foo();
  int x;
}
void bar() {
  N::foo();     // ScopePrefix N
  ::N::x++;     // ScopePrefix ::N
  foo();        // No ScopePrefix
}
```

SequenceNumber

The sequence number is a unique number that is assigned in ascending order to each source line as it is read. Unlike a file name or line number, the sequence number is unambiguous. Example (actual numbers might differ):

```
// header.h - included by source.c
void foo();     // SequenceNumber 2, Line 1

// source.c - includes header.h
void bar();     // SequenceNumber 1, Line 1
#include "header.h"
bar();          // SequenceNumber 3, Line 3
```

Size

Indicates the size of a type (such as, `array`, `char`, `int`, `pointer`, and so on) as returned by `sizeof()`. Returns `0`, if the type is incomplete.

StartLine

Returns the line number where the declaration associated with the node begins in the source code. For example:

```
static void   // line returned by property 'StartLine' for 'Functions'
f(int i) {}   // line returned by property 'Line' for 'Functions'
```

TemplateArgumentList

List of all template types used to define template class.

TemplateDeclaration

For template specialization, returns template declaration that was used for this instantiation. For template template specialization, returns primary template declaration.

Template Parameters

For Class, Struct, Union, Global Function this property represents a set of template parameter nodes. Example:

```
template <class X, class Y> class Z {};
```

X and Y will be returned for the above code.

Then

The node that represents the statement part of an `if` statement (as opposed to the conditional part).

Throws

The node that corresponds the exception object thrown with the function.

Traverse

This property can be used on some compound types, for example typedefs, pointers, references, arrays, functions. In 'Direct Check' mode it returns the type, itself. In 'Indirect Check' mode it traverses and returns types that make up the compound type. For example:

```
typedef int * INT;
INT x1; // traverse: [ int, int *, INT ]
```

TrueChoice

The node that represents the statement processed for a true condition in a `?:` statement.

Type

Returns the type of the node, according to the C/C++ standard. For example:

```
void f() {
    unsigned char uc1, uc2;
    uc1 + uc2;              // (a+b) Type: int
}
```

TypeTraverseReference

When the property is used in 'Direct Check' mode, it returns the direct type of an entity, just as the 'Type' property does. In 'Indirect Check' mode, the property does not stop at reference but continues traversal. For example:

```
// When properties are used in 'Indirect Check' mode,
// the 'Type' property traverses typedefs, but stops
// at references, while TypeTraverseReference traverses
// past them:

typedef int INT;
// Type: [ int, INT ]
// TypeTraverseTeference: [ int, INT ]
INT x1;

// Type: [ INT & ],
// TypeTraverseTeference: [int, INT, INT & ]
INT &x2 = x1;
```

UnderlyingElementType

For array returns underlying element type. Example:

```
int* tab1[1]; // underlying type: int*
float tab2[1][1]; // underlying type: float
typedef double ARR[10];
ARR tab3[1]; // underlying type: double
```

UsesDesignatedInitializer

For aggregate initializer elements, returns `true` if a designated initializer was used.

UsesDesignatedInitializers

For aggregate initializers, returns `true` if a designated initializer was used.

UsesOperatorSyntax

For the a(b) node, returns `true` if the node represents an implicit call to the operator function. Example:

```
class A {
public:
   A& operator+(A& rhs);
};
void f1 (A& a1, A& a2)
{
   a1 + a2;   // UsesOperatorSyntax = true
}
```

Value

For `bool Constant`, this is a boolean property that returns `true` for `true` constant. Example:

```
return true;   // bool Constant: Value is true
return false;  // bool Constant: Value is false
```

For `integer Constant`, `Integer Constant`, and `Real Constant`, this is a number property that returns the value of the constant. Example:

```
int x = 10;   // Integer Constant: Value is 10
int y = 22.5; // Real Constant: Value us 22.5
enum E {
  ENUM_CONSTANT1 = 22,  // enum Constant: Value is 22
  ENUM_CONSTANT2        // enum Constant: Value is 23
}
```

For `String Constant`, this is a string property that returns the string value. Example:

```
char * company = "Parasoft";  // String Constant: Value is Parasoft
```

Virtual Parent

Indicates whether a class is inherited using `virtual` as a keyword. Class example:

```
virtual Base {};
```

__condition

For range-based for statements, represents the expression for the `__begin != __end` test. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

__increment

For range-based for statements, represents the expression for the ++`__begin` increment. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

__range

For range-based for statements, represents the `__range` temporary variable. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

__begin

For range-based for statements, represents the `__begin` temporary variable. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```

__end

For range-based for statements, represents the __end temporary variable. Example:

```
void bar();
void func() {
   for (auto iterator : range_expression) {  // for
      bar();
   }
/* This is equivalent to:
    auto && __range = (range_expression);
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        auto iterator = *__begin;
        bar();
    }
*/
}
```