AST Matcher Reference

This document shows all currently implemented matchers. The matchers are grouped by category and node type they match. You can click on matcher names to show the matcher's source documentation.

There are three different basic categories of matchers:

- Node Matchers: Matchers that match a specific type of AST node.
- Narrowing Matchers: Matchers that match attributes on AST nodes.
- Traversal Matchers: Matchers that allow traversal between AST nodes.

Within each category the matchers are ordered by node type they match on. Note that if a matcher can match multiple node types, it will it will appear multiple times. This means that by searching for Matcher<Stmt> you can find all matchers that can be used to match on Stmt nodes.

The exception to that rule are matchers that can match on any node. Those are marked with a * and are listed in the beginning of each category.

Note that the categorization of matchers is a great help when you combine them into matcher expressions. You will usually want to form matcher expressions that read like english sentences by alternating between node matchers and narrowing or traversal matchers, like this:

```
recordDecl(hasDescendant(
    ifStmt(hasTrueExpression(
        expr(hasDescendant(
            ifStmt())))))
```

Node Matchers

Node matchers are at the core of matcher expressions - they specify the type of node that is expected. Every match expression starts with a node matcher, which can then be further refined with a narrowing or traversal matcher. All traversal matchers take node matchers as their arguments.

For convenience, all node matchers take an arbitrary number of arguments and implicitly act as allOf matchers.

Node matchers are the only matchers that support the bind("id") call to bind the matched node to the given string, to be later retrieved from the match callback.

It is important to remember that the arguments to node matchers are predicates on the same node, just with additional information about the type. This is often useful to make matcher expression more readable by inlining bind calls into redundant node matchers inside another node matcher:

```
// This binds the CXXRecordDecl to "id", as the decl() matcher will stay on
// the same node.
recordDecl(decl().bind("id"), hasName("::MyClass"))
```

Return type	Name	Parameters
Matcher<CXXCtorInitializer>	cxxCtorInitializer	Matcher<CXXCtorInitializer>...

```
Matches constructor initializers.

Examples matches i(42).
  class C {
    C() : i(42) {}
    int i;
  };
```

Matcher<Decl>	accessSpecDecl	Matcher<AccessSpecDecl>...

```
Matches C++ access specifier declarations.

Given
  class C {
  public:
    int a;
  };
accessSpecDecl()
  matches 'public:'
```

Matcher<Decl>	classTemplateDecl	Matcher<ClassTemplateDecl>...

```
Matches C++ class template declarations.

Example matches Z
  template<class T> class Z {};
```

Matcher<Decl>	classTemplateSpecializationDecl	Matcher<ClassTemplateSpecializationDecl>...

```
Matches C++ class template specializations.

Given
  template<typename T> class A {};
  template<> class A<double> {};
  A<int> a;
```

```
classTemplateSpecializationDecl()
  matches the specializations A<int> and A<double>
```

Matcher<Decl>	cxxConstructorDecl	Matcher<CXXConstructorDecl>...

```
Matches C++ constructor declarations.

Example matches Foo::Foo() and Foo::Foo(int)
  class Foo {
   public:
    Foo();
    Foo(int);
    int DoSomething();
  };
```

Matcher<Decl>	cxxConversionDecl	Matcher<CXXConversionDecl>...

```
Matches conversion operator declarations.

Example matches the operator.
  class X { operator int() const; };
```

Matcher<Decl>	cxxDestructorDecl	Matcher<CXXDestructorDecl>...

```
Matches explicit C++ destructor declarations.

Example matches Foo::~Foo()
  class Foo {
   public:
    virtual ~Foo();
  };
```

Matcher<Decl>	cxxMethodDecl	Matcher<CXXMethodDecl>...

```
Matches method declarations.

Example matches y
  class X { void y(); };
```

Matcher<Decl>	cxxRecordDecl	Matcher<CXXRecordDecl>...

```
Matches C++ class declarations.

Example matches X, Z
  class X;
  template<class T> class Z {};
```

Matcher<Decl>	decl	Matcher<Decl>...

```
Matches declarations.

Examples matches X, C, and the friend declaration inside C;
  void X();
  class C {
    friend X;
  };
```

Matcher<Decl>	declaratorDecl	Matcher<DeclaratorDecl>...

```
Matches declarator declarations (field, variable, function
and non-type template parameter declarations).

Given
  class X { int y; };
declaratorDecl()
  matches int y.
```

Matcher<Decl>	enumConstantDecl	Matcher<EnumConstantDecl>...

```
Matches enum constants.

Example matches A, B, C
  enum X {
    A, B, C
  };
```

Matcher<Decl>	enumDecl	Matcher<EnumDecl>...

```
Matches enum declarations.

Example matches X
  enum X {
    A, B, C
  };
```

Matcher<Decl>	fieldDecl	Matcher<FieldDecl>...

```
Matches field declarations.

Given
  class X { int m; };
fieldDecl()
  matches 'm'.
```

Matcher<Decl>	friendDecl	Matcher<FriendDecl>...

```
Matches friend declarations.

Given
  class X { friend void foo(); };
friendDecl()
  matches 'friend void foo()'.
```

Matcher<Decl>	functionDecl	Matcher<FunctionDecl>...

```
Matches function declarations.

Example matches f
  void f();
```

Matcher<Decl> functionTemplateDecl Matcher<FunctionTemplateDecl>...

```
Matches C++ function template declarations.

Example matches f
  template<class T> void f(T t) {}
```

Matcher<Decl> labelDecl Matcher<LabelDecl>...

```
Matches a declaration of label.

Given
  goto FOO;
  FOO: bar();
labelDecl()
  matches 'FOO:'
```

Matcher<Decl> linkageSpecDecl Matcher<LinkageSpecDecl>...

```
Matches a declaration of a linkage specification.

Given
  extern "C" {}
linkageSpecDecl()
  matches "extern "C" {}"
```

Matcher<Decl> namedDecl Matcher<NamedDecl>...

```
Matches a declaration of anything that could have a name.

Example matches X, S, the anonymous union type, i, and U;
  typedef int X;
  struct S {
    union {
      int i;
    } U;
  };
```

Matcher<Decl> namespaceAliasDecl Matcher<NamespaceAliasDecl>...

```
Matches a declaration of a namespace alias.

Given
  namespace test {}
  namespace alias = ::test;
namespaceAliasDecl()
  matches "namespace alias" but not "namespace test"
```

Matcher<Decl> namespaceDecl Matcher<NamespaceDecl>...

```
Matches a declaration of a namespace.

Given
  namespace {}
  namespace test {}
namespaceDecl()
  matches "namespace {}" and "namespace test {}"
```

Matcher<Decl> nonTypeTemplateParmDecl Matcher<NonTypeTemplateParmDecl>...

```
Matches non-type template parameter declarations.

Given
  template <typename T, int N> struct C {};
nonTypeTemplateParmDecl()
  matches 'N', but not 'T'.
```

Matcher<Decl> objcCategoryDecl Matcher<ObjCCategoryDecl>...

```
Matches Objective-C category declarations.

Example matches Foo (Additions)
  @interface Foo (Additions)
  @end
```

Matcher<Decl> objcInterfaceDecl Matcher<ObjCInterfaceDecl>...

```
Matches Objective-C interface declarations.

Example matches Foo
  @interface Foo
  @end
```

Matcher<Decl> objcIvarDecl Matcher<ObjCIvarDecl>...

```
Matches Objective-C instance variable declarations.

Example matches _enabled
  @implementation Foo {
    BOOL _enabled;
  }
  @end
```

Matcher<Decl> objcMethodDecl Matcher<ObjCMethodDecl>...

```
Matches Objective-C method declarations.

Example matches both declaration and definition of -[Foo method]
  @interface Foo
  - (void)method;
```

```
  @end

  @implementation Foo
  - (void)method {}
  @end
```

Matcher<Decl>	objcPropertyDecl	Matcher<ObjCPropertyDecl>...

```
Matches Objective-C property declarations.

Example matches enabled
  @interface Foo
  @property BOOL enabled;
  @end
```

Matcher<Decl>	objcProtocolDecl	Matcher<ObjCProtocolDecl>...

```
Matches Objective-C protocol declarations.

Example matches FooDelegate
  @protocol FooDelegate
  @end
```

Matcher<Decl>	parmVarDecl	Matcher<ParmVarDecl>...

```
Matches parameter variable declarations.

Given
  void f(int x);
parmVarDecl()
  matches int x.
```

Matcher<Decl>	recordDecl	Matcher<RecordDecl>...

```
Matches class, struct, and union declarations.

Example matches X, Z, U, and S
  class X;
  template<class T> class Z {};
  struct S {};
  union U {};
```

Matcher<Decl>	staticAssertDecl	Matcher<StaticAssertDecl>...

```
Matches a C++ static_assert declaration.

Example:
  staticAssertExpr()
matches
  static_assert(sizeof(S) == sizeof(int))
in
  struct S {
    int x;
  };
  static_assert(sizeof(S) == sizeof(int));
```

Matcher<Decl>	templateTypeParmDecl	Matcher<TemplateTypeParmDecl>...

```
Matches template type parameter declarations.

Given
  template <typename T, int N> struct C {};
templateTypeParmDecl()
  matches 'T', but not 'N'.
```

Matcher<Decl>	translationUnitDecl	Matcher<TranslationUnitDecl>...

```
Matches the top declaration context.

Given
  int X;
  namespace NS {
  int Y;
  }  namespace NS
decl(hasDeclContext(translationUnitDecl()))
  matches "int X", but not "int Y".
```

Matcher<Decl>	typeAliasDecl	Matcher<TypeAliasDecl>...

```
Matches type alias declarations.

Given
  typedef int X;
  using Y = int;
typeAliasDecl()
  matches "using Y = int", but not "typedef int X"
```

Matcher<Decl>	typeAliasTemplateDecl	Matcher<TypeAliasTemplateDecl>...

```
Matches type alias template declarations.

typeAliasTemplateDecl() matches
  template <typename T>
  using Y = X<T>;
```

Matcher<Decl>	typedefDecl	Matcher<TypedefDecl>...

```
Matches typedef declarations.

Given
  typedef int X;
  using Y = int;
typedefDecl()
  matches "typedef int X", but not "using Y = int"
```

Matcher<Decl> typedefNameDecl Matcher<TypedefNameDecl>...

```
Matches typedef name declarations.

Given
  typedef int X;
  using Y = int;
typedefNameDecl()
  matches "typedef int X" and "using Y = int"
```

Matcher<Decl> unresolvedUsingTypenameDecl Matcher<UnresolvedUsingTypenameDecl>...

```
Matches unresolved using value declarations that involve the
typename.

Given
  template <typename T>
  struct Base { typedef T Foo; };

  template<typename T>
  struct S : private Base<T> {
    using typename Base<T>::Foo;
  };
unresolvedUsingTypenameDecl()
  matches using Base<T>::Foo
```

Matcher<Decl> unresolvedUsingValueDecl Matcher<UnresolvedUsingValueDecl>...

```
Matches unresolved using value declarations.

Given
  template<typename X>
  class C : private X {
    using X::x;
  };
unresolvedUsingValueDecl()
  matches using X::x
```

Matcher<Decl> usingDecl Matcher<UsingDecl>...

```
Matches using declarations.

Given
  namespace X { int x; }
  using X::x;
usingDecl()
  matches using X::x
```

Matcher<Decl> usingDirectiveDecl Matcher<UsingDirectiveDecl>...

```
Matches using namespace declarations.

Given
  namespace X { int x; }
  using namespace X;
usingDirectiveDecl()
  matches using namespace X
```

Matcher<Decl> valueDecl Matcher<ValueDecl>...

```
Matches any value declaration.

Example matches A, B, C and F
  enum X { A, B, C };
  void F();
```

Matcher<Decl> varDecl Matcher<VarDecl>...

```
Matches variable declarations.

Note: this does not match declarations of member variables, which are
"field" declarations in Clang parlance.

Example matches a
  int a;
```

Matcher<NestedNameSpecifierLoc> nestedNameSpecifierLoc Matcher<NestedNameSpecifierLoc>...

```
Same as nestedNameSpecifier but matches NestedNameSpecifierLoc.
```

Matcher<NestedNameSpecifier> nestedNameSpecifier Matcher<NestedNameSpecifier>...

```
Matches nested name specifiers.

Given
  namespace ns {
    struct A { static void f(); };
    void A::f() {}
    void g() { A::f(); }
  }
  ns::A a;
nestedNameSpecifier()
  matches "ns::" and both "A::"
```

Matcher<QualType> qualType Matcher<QualType>...

```
Matches QualTypes in the clang AST.
```

Matcher<Stmt> addrLabelExpr Matcher<AddrLabelExpr>...

```
Matches address of label statements (GNU extension).

Given
  FOO: bar();
  void *ptr = &&FOO;
```

```
  goto *bar;
addrLabelExpr()
  matches '&&FOO'
```

Matcher<Stmt>	arraySubscriptExpr	Matcher<ArraySubscriptExpr>...

```
Matches array subscript expressions.

Given
  int i = a[1];
arraySubscriptExpr()
  matches "a[1]"
```

Matcher<Stmt>	asmStmt	Matcher<AsmStmt>...

```
Matches asm statements.

 int i = 100;
  __asm("mov al, 2");
asmStmt()
  matches '__asm("mov al, 2")'
```

Matcher<Stmt>	atomicExpr	Matcher<AtomicExpr>...

```
Matches atomic builtins.
Example matches __atomic_load_n(ptr, 1)
  void foo() { int *ptr; __atomic_load_n(ptr, 1); }
```

Matcher<Stmt>	binaryConditionalOperator	Matcher<BinaryConditionalOperator>...

```
Matches binary conditional operator expressions (GNU extension).

Example matches a ?: b
  (a ?: b) + 42;
```

Matcher<Stmt>	binaryOperator	Matcher<BinaryOperator>...

```
Matches binary operator expressions.

Example matches a || b
  !(a || b)
```

Matcher<Stmt>	breakStmt	Matcher<BreakStmt>...

```
Matches break statements.

Given
  while (true) { break; }
breakStmt()
  matches 'break'
```

Matcher<Stmt>	cStyleCastExpr	Matcher<CStyleCastExpr>...

```
Matches a C-style cast expression.

Example: Matches (int) 2.2f in
  int i = (int) 2.2f;
```

Matcher<Stmt>	callExpr	Matcher<CallExpr>...

```
Matches call expressions.

Example matches x.y() and y()
  X x;
  x.y();
  y();
```

Matcher<Stmt>	caseStmt	Matcher<CaseStmt>...

```
Matches case statements inside switch statements.

Given
  switch(a) { case 42: break; default: break; }
caseStmt()
  matches 'case 42: break;'.
```

Matcher<Stmt>	castExpr	Matcher<CastExpr>...

```
Matches any cast nodes of Clang's AST.

Example: castExpr() matches each of the following:
  (int) 3;
  const_cast<Expr *>(SubExpr);
  char c = 0;
but does not match
  int i = (0);
  int k = 0;
```

Matcher<Stmt>	characterLiteral	Matcher<CharacterLiteral>...

```
Matches character literals (also matches wchar_t).

Not matching Hex-encoded chars (e.g. 0x1234, which is a IntegerLiteral),
though.

Example matches 'a', L'a'
  char ch = 'a';
  wchar_t chw = L'a';
```

Matcher<Stmt>	compoundLiteralExpr	Matcher<CompoundLiteralExpr>...

```
Matches compound (i.e. non-scalar) literals

Example match: {1}, (1, 2)
```

```
  int array[4] = {1};
  vector int myvec = (vector int)(1, 2);
```

Matcher<Stmt>	compoundStmt	Matcher<CompoundStmt>...

```
Matches compound statements.

Example matches '{}' and '{{}}'in 'for (;;) {{}}'
  for (;;) {{}}
```

Matcher<Stmt>	conditionalOperator	Matcher<ConditionalOperator>...

```
Matches conditional operator expressions.

Example matches a ? b : c
  (a ? b : c) + 42
```

Matcher<Stmt>	continueStmt	Matcher<ContinueStmt>...

```
Matches continue statements.

Given
  while (true) { continue; }
continueStmt()
  matches 'continue'
```

Matcher<Stmt>	cudaKernelCallExpr	Matcher<CUDAKernelCallExpr>...

```
Matches CUDA kernel call expression.

Example matches,
  kernel<<<i,j>>>();
```

Matcher<Stmt>	cxxBindTemporaryExpr	Matcher<CXXBindTemporaryExpr>...

```
Matches nodes where temporaries are created.

Example matches FunctionTakesString(GetStringByValue())
    (matcher = cxxBindTemporaryExpr())
  FunctionTakesString(GetStringByValue());
  FunctionTakesStringByPointer(GetStringPointer());
```

Matcher<Stmt>	cxxBoolLiteral	Matcher<CXXBoolLiteralExpr>...

```
Matches bool literals.

Example matches true
  true
```

Matcher<Stmt>	cxxCatchStmt	Matcher<CXXCatchStmt>...

```
Matches catch statements.

  try {} catch(int i) {}
cxxCatchStmt()
  matches 'catch(int i)'
```

Matcher<Stmt>	cxxConstCastExpr	Matcher<CXXConstCastExpr>...

```
Matches a const_cast expression.

Example: Matches const_cast<int*>(&r) in
  int n = 42;
  const int &r(n);
  int* p = const_cast<int*>(&r);
```

Matcher<Stmt>	cxxConstructExpr	Matcher<CXXConstructExpr>...

```
Matches constructor call expressions (including implicit ones).

Example matches string(ptr, n) and ptr within arguments of f
    (matcher = cxxConstructExpr())
  void f(const string &a, const string &b);
  char *ptr;
  int n;
  f(string(ptr, n), ptr);
```

Matcher<Stmt>	cxxDefaultArgExpr	Matcher<CXXDefaultArgExpr>...

```
Matches the value of a default argument at the call site.

Example matches the CXXDefaultArgExpr placeholder inserted for the
    default value of the second parameter in the call expression f(42)
    (matcher = cxxDefaultArgExpr())
  void f(int x, int y = 0);
  f(42);
```

Matcher<Stmt>	cxxDeleteExpr	Matcher<CXXDeleteExpr>...

```
Matches delete expressions.

Given
  delete X;
cxxDeleteExpr()
  matches 'delete X'.
```

Matcher<Stmt>	cxxDynamicCastExpr	Matcher<CXXDynamicCastExpr>...

```
Matches a dynamic_cast expression.

Example:
  cxxDynamicCastExpr()
matches
  dynamic_cast<D*>(&b);
```

```
in
  struct B { virtual ~B() {} }; struct D : B {};
  B b;
  D* p = dynamic_cast<D*>(&b);
```

Matcher<Stmt> cxxForRangeStmt Matcher<CXXForRangeStmt>...

```
Matches range-based for statements.

cxxForRangeStmt() matches 'for (auto a : i)'
  int i[] =  {1, 2, 3}; for (auto a : i);
  for(int j = 0; j < 5; ++j);
```

Matcher<Stmt> cxxFunctionalCastExpr Matcher<CXXFunctionalCastExpr>...

```
Matches functional cast expressions

Example: Matches Foo(bar);
  Foo f = bar;
  Foo g = (Foo) bar;
  Foo h = Foo(bar);
```

Matcher<Stmt> cxxMemberCallExpr Matcher<CXXMemberCallExpr>...

```
Matches member call expressions.

Example matches x.y()
  X x;
  x.y();
```

Matcher<Stmt> cxxNewExpr Matcher<CXXNewExpr>...

```
Matches new expressions.

Given
  new X;
cxxNewExpr()
  matches 'new X'.
```

Matcher<Stmt> cxxNullPtrLiteralExpr Matcher<CXXNullPtrLiteralExpr>...

```
Matches nullptr literal.
```

Matcher<Stmt> cxxOperatorCallExpr Matcher<CXXOperatorCallExpr>...

```
Matches overloaded operator calls.

Note that if an operator isn't overloaded, it won't match. Instead, use
binaryOperator matcher.
Currently it does not match operators such as new delete.
FIXME: figure out why these do not match?

Example matches both operator<<((o << b), c) and operator<<(o, b)
    (matcher = cxxOperatorCallExpr())
  ostream &operator<< (ostream &out, int i) { };
  ostream &o; int b = 1, c = 1;
  o << b << c;
```

Matcher<Stmt> cxxReinterpretCastExpr Matcher<CXXReinterpretCastExpr>...

```
Matches a reinterpret_cast expression.

Either the source expression or the destination type can be matched
using has(), but hasDestinationType() is more specific and can be
more readable.

Example matches reinterpret_cast<char*>(&p) in
  void* p = reinterpret_cast<char*>(&p);
```

Matcher<Stmt> cxxStaticCastExpr Matcher<CXXStaticCastExpr>...

```
Matches a C++ static_cast expression.

See also: hasDestinationType
See also: reinterpretCast

Example:
  cxxStaticCastExpr()
matches
  static_cast<long>(8)
in
  long eight(static_cast<long>(8));
```

Matcher<Stmt> cxxStdInitializerListExpr Matcher<CXXStdInitializerListExpr>...

```
Matches C++ initializer list expressions.

Given
  std::vector<int> a({ 1, 2, 3 });
  std::vector<int> b = { 4, 5 };
  int c[] = { 6, 7 };
  std::pair<int, int> d = { 8, 9 };
cxxStdInitializerListExpr()
  matches "{ 1, 2, 3 }" and "{ 4, 5 }"
```

Matcher<Stmt> cxxTemporaryObjectExpr Matcher<CXXTemporaryObjectExpr>...

```
Matches functional cast expressions having N != 1 arguments

Example: Matches Foo(bar, bar)
  Foo h = Foo(bar, bar);
```

Matcher<Stmt> cxxThisExpr Matcher<CXXThisExpr>...

```
Matches implicit and explicit this expressions.
```

```
Example matches the implicit this expression in "return i".
    (matcher = cxxThisExpr())
struct foo {
  int i;
  int f() { return i; }
};
```

Matcher<Stmt> cxxThrowExpr Matcher<CXXThrowExpr>...

```
Matches throw expressions.

  try { throw 5; } catch(int i) {}
cxxThrowExpr()
  matches 'throw 5'
```

Matcher<Stmt> cxxTryStmt Matcher<CXXTryStmt>...

```
Matches try statements.

  try {} catch(int i) {}
cxxTryStmt()
  matches 'try {}'
```

Matcher<Stmt> cxxUnresolvedConstructExpr Matcher<CXXUnresolvedConstructExpr>...

```
Matches unresolved constructor call expressions.

Example matches T(t) in return statement of f
    (matcher = cxxUnresolvedConstructExpr())
  template <typename T>
  void f(const T& t) { return T(t); }
```

Matcher<Stmt> declRefExpr Matcher<DeclRefExpr>...

```
Matches expressions that refer to declarations.

Example matches x in if (x)
  bool x;
  if (x) {}
```

Matcher<Stmt> declStmt Matcher<DeclStmt>...

```
Matches declaration statements.

Given
  int a;
declStmt()
  matches 'int a'.
```

Matcher<Stmt> defaultStmt Matcher<DefaultStmt>...

```
Matches default statements inside switch statements.

Given
  switch(a) { case 42: break; default: break; }
defaultStmt()
  matches 'default: break;'.
```

Matcher<Stmt> designatedInitExpr Matcher<DesignatedInitExpr>...

```
Matches C99 designated initializer expressions [C99 6.7.8].

Example: Matches { [2].y = 1.0, [0].x = 1.0 }
  point ptarray[10] = { [2].y = 1.0, [0].x = 1.0 };
```

Matcher<Stmt> doStmt Matcher<DoStmt>...

```
Matches do statements.

Given
  do {} while (true);
doStmt()
  matches 'do {} while(true)'
```

Matcher<Stmt> explicitCastExpr Matcher<ExplicitCastExpr>...

```
Matches explicit cast expressions.

Matches any cast expression written in user code, whether it be a
C-style cast, a functional-style cast, or a keyword cast.

Does not match implicit conversions.

Note: the name "explicitCast" is chosen to match Clang's terminology, as
Clang uses the term "cast" to apply to implicit conversions as well as to
actual cast expressions.

See also: hasDestinationType.

Example: matches all five of the casts in
  int((int)(reinterpret_cast<int>(static_cast<int>(const_cast<int>(42)))))
but does not match the implicit conversion in
  long ell = 42;
```

Matcher<Stmt> expr Matcher<Expr>...

```
Matches expressions.

Example matches x()
  void f() { x(); }
```

Matcher<Stmt> exprWithCleanups Matcher<ExprWithCleanups>...

```
Matches expressions that introduce cleanups to be run at the end
```

```
of the sub-expression's evaluation.

Example matches std::string()
  const std::string str = std::string();
```

Matcher<Stmt> floatLiteral Matcher<FloatingLiteral>...

```
Matches float literals of all sizes encodings, e.g.
1.0, 1.0f, 1.0L and 1e10.

Does not match implicit conversions such as
  float a = 10;
```

Matcher<Stmt> forStmt Matcher<ForStmt>...

```
Matches for statements.

Example matches 'for (;;) {}'
  for (;;) {}
  int i[] =  {1, 2, 3}; for (auto a : i);
```

Matcher<Stmt> gnuNullExpr Matcher<GNUNullExpr>...

```
Matches GNU __null expression.
```

Matcher<Stmt> gotoStmt Matcher<GotoStmt>...

```
Matches goto statements.

Given
  goto FOO;
  FOO: bar();
gotoStmt()
  matches 'goto FOO'
```

Matcher<Stmt> ifStmt Matcher<IfStmt>...

```
Matches if statements.

Example matches 'if (x) {}'
  if (x) {}
```

Matcher<Stmt> implicitCastExpr Matcher<ImplicitCastExpr>...

```
Matches the implicit cast nodes of Clang's AST.

This matches many different places, including function call return value
eliding, as well as any type conversions.
```

Matcher<Stmt> implicitValueInitExpr Matcher<ImplicitValueInitExpr>...

```
Matches implicit initializers of init list expressions.

Given
  point ptarray[10] = { [2].y = 1.0, [2].x = 2.0, [0].x = 1.0 };
implicitValueInitExpr()
  matches "[0].y" (implicitly)
```

Matcher<Stmt> initListExpr Matcher<InitListExpr>...

```
Matches init list expressions.

Given
  int a[] = { 1, 2 };
  struct B { int x, y; };
  B b = { 5, 6 };
initListExpr()
  matches "{ 1, 2 }" and "{ 5, 6 }"
```

Matcher<Stmt> integerLiteral Matcher<IntegerLiteral>...

```
Matches integer literals of all sizes encodings, e.g.
1, 1L, 0x1 and 1U.

Does not match character-encoded integers such as L'a'.
```

Matcher<Stmt> labelStmt Matcher<LabelStmt>...

```
Matches label statements.

Given
  goto FOO;
  FOO: bar();
labelStmt()
  matches 'FOO:'
```

Matcher<Stmt> lambdaExpr Matcher<LambdaExpr>...

```
Matches lambda expressions.

Example matches [&](){return 5;}
  [&](){return 5;}
```

Matcher<Stmt> materializeTemporaryExpr Matcher<MaterializeTemporaryExpr>...

```
Matches nodes where temporaries are materialized.

Example: Given
  struct T {void func();};
  T f();
  void g(T);
materializeTemporaryExpr() matches 'f()' in these statements
  T u(f());
  g(f());
```

```
but does not match
  f();
  f().func();
```

Matcher<Stmt> memberExpr Matcher<MemberExpr>...

```
Matches member expressions.

Given
  class Y {
    void x() { this->x(); x(); Y y; y.x(); a; this->b; Y::b; }
    int a; static int b;
  };
memberExpr()
  matches this->x, x, y.x, a, this->b
```

Matcher<Stmt> nullStmt Matcher<NullStmt>...

```
Matches null statements.

  foo();;
nullStmt()
  matches the second ';'
```

Matcher<Stmt> objcMessageExpr Matcher<ObjCMessageExpr>...

```
Matches ObjectiveC Message invocation expressions.

The innermost message send invokes the "alloc" class method on the
NSString class, while the outermost message send invokes the
"initWithString" instance method on the object returned from
NSString's "alloc". This matcher should match both message sends.
  [[NSString alloc] initWithString:@"Hello"]
```

Matcher<Stmt> opaqueValueExpr Matcher<OpaqueValueExpr>...

```
Matches opaque value expressions. They are used as helpers
to reference another expressions and can be met
in BinaryConditionalOperators, for example.

Example matches 'a'
  (a ?: c) + 42;
```

Matcher<Stmt> parenExpr Matcher<ParenExpr>...

```
Matches parentheses used in expressions.

Example matches (foo() + 1)
  int foo() { return 1; }
  int a = (foo() + 1);
```

Matcher<Stmt> parenListExpr Matcher<ParenListExpr>...

```
Matches paren list expressions.
ParenListExprs don't have a predefined type and are used for late parsing.
In the final AST, they can be met in template declarations.

Given
  template<typename T> class X {
    void f() {
      X x(*this);
      int a = 0, b = 1; int i = (a, b);
    }
  };
parenListExpr() matches "*this" but NOT matches (a, b) because (a, b)
has a predefined type and is a ParenExpr, not a ParenListExpr.
```

Matcher<Stmt> predefinedExpr Matcher<PredefinedExpr>...

```
Matches predefined identifier expressions [C99 6.4.2.2].

Example: Matches __func__
  printf("%s", __func__);
```

Matcher<Stmt> returnStmt Matcher<ReturnStmt>...

```
Matches return statements.

Given
  return 1;
returnStmt()
  matches 'return 1'
```

Matcher<Stmt> stmt Matcher<Stmt>...

```
Matches statements.

Given
  { ++a; }
stmt()
  matches both the compound statement '{ ++a; }' and '++a'.
```

Matcher<Stmt> stmtExpr Matcher<StmtExpr>...

```
Matches statement expression (GNU extension).

Example match: ({ int X = 4; X; })
  int C = ({ int X = 4; X; });
```

Matcher<Stmt> stringLiteral Matcher<StringLiteral>...

```
Matches string literals (also matches wide string literals).

Example matches "abcd", L"abcd"
```

```
  char *s = "abcd";
  wchar_t *ws = L"abcd";
```

Matcher<Stmt>	substNonTypeTemplateParmExpr	Matcher<SubstNonTypeTemplateParmExpr>...

```
Matches substitutions of non-type template parameters.

Given
  template <int N>
  struct A { static const int n = N; };
  struct B : public A<42> {};
substNonTypeTemplateParmExpr()
  matches "N" in the right-hand side of "static const int n = N;"
```

Matcher<Stmt>	switchCase	Matcher<SwitchCase>...

```
Matches case and default statements inside switch statements.

Given
  switch(a) { case 42: break; default: break; }
switchCase()
  matches 'case 42: break;' and 'default: break;'.
```

Matcher<Stmt>	switchStmt	Matcher<SwitchStmt>...

```
Matches switch statements.

Given
  switch(a) { case 42: break; default: break; }
switchStmt()
  matches 'switch(a)'.
```

Matcher<Stmt>	unaryExprOrTypeTraitExpr	Matcher<UnaryExprOrTypeTraitExpr>...

```
Matches sizeof (C99), alignof (C++11) and vec_step (OpenCL)

Given
  Foo x = bar;
  int y = sizeof(x) + alignof(x);
unaryExprOrTypeTraitExpr()
  matches sizeof(x) and alignof(x)
```

Matcher<Stmt>	unaryOperator	Matcher<UnaryOperator>...

```
Matches unary operator expressions.

Example matches !a
  !a || b
```

Matcher<Stmt>	unresolvedLookupExpr	Matcher<UnresolvedLookupExpr>...

```
Matches reference to a name that can be looked up during parsing
but could not be resolved to a specific declaration.

Given
  template<typename T>
  T foo() { T a; return a; }
  template<typename T>
  void bar() {
    foo<T>();
  }
unresolvedLookupExpr()
  matches foo<T>()
```

Matcher<Stmt>	userDefinedLiteral	Matcher<UserDefinedLiteral>...

```
Matches user defined literal operator call.

Example match: "foo"_suffix
```

Matcher<Stmt>	whileStmt	Matcher<WhileStmt>...

```
Matches while statements.

Given
  while (true) {}
whileStmt()
  matches 'while (true) {}'.
```

Matcher<TemplateArgument>	templateArgument	Matcher<TemplateArgument>...

```
Matches template arguments.

Given
  template <typename T> struct C {};
  C<int> c;
templateArgument()
  matches 'int' in C<int>.
```

Matcher<TemplateName>	templateName	Matcher<TemplateName>...

```
Matches template name.

Given
  template <typename T> class X { };
  X<int> xi;
templateName()
  matches 'X' in X<int>.
```

Matcher<TypeLoc>	typeLoc	Matcher<TypeLoc>...

```
Matches TypeLocs in the clang AST.
```

Matcher<Type>	arrayType	Matcher<ArrayType>...

```
Matches all kinds of arrays.

Given
  int a[] = { 2, 3 };
  int b[4];
  void f() { int c[a[0]]; }
arrayType()
  matches "int a[]", "int b[4]" and "int c[a[0]]";
```

Matcher<Type> atomicType Matcher<AtomicType>...

```
Matches atomic types.

Given
  _Atomic(int) i;
atomicType()
  matches "_Atomic(int) i"
```

Matcher<Type> autoType Matcher<AutoType>...

```
Matches types nodes representing C++11 auto types.

Given:
  auto n = 4;
  int v[] = { 2, 3 }
  for (auto i : v) { }
autoType()
  matches "auto n" and "auto i"
```

Matcher<Type> blockPointerType Matcher<BlockPointerType>...

```
Matches block pointer types, i.e. types syntactically represented as
"void (^)(int)".

The pointee is always required to be a FunctionType.
```

Matcher<Type> builtinType Matcher<BuiltinType>...

```
Matches builtin Types.

Given
  struct A {};
  A a;
  int b;
  float c;
  bool d;
builtinType()
  matches "int b", "float c" and "bool d"
```

Matcher<Type> complexType Matcher<ComplexType>...

```
Matches C99 complex types.

Given
  _Complex float f;
complexType()
  matches "_Complex float f"
```

Matcher<Type> constantArrayType Matcher<ConstantArrayType>...

```
Matches C arrays with a specified constant size.

Given
  void() {
    int a[2];
    int b[] = { 2, 3 };
    int c[b[0]];
  }
constantArrayType()
  matches "int a[2]"
```

Matcher<Type> decayedType Matcher<DecayedType>...

```
Matches decayed type
Example matches i[] in declaration of f.
    (matcher = valueDecl(hasType(decayedType(hasDecayedType(pointerType())))))
Example matches i[1].
    (matcher = expr(hasType(decayedType(hasDecayedType(pointerType())))))
  void f(int i[]) {
    i[1] = 0;
  }
```

Matcher<Type> dependentSizedArrayType Matcher<DependentSizedArrayType>...

```
Matches C++ arrays whose size is a value-dependent expression.

Given
  template<typename T, int Size>
  class array {
    T data[Size];
  };
dependentSizedArrayType
  matches "T data[Size]"
```

Matcher<Type> elaboratedType Matcher<ElaboratedType>...

```
Matches types specified with an elaborated type keyword or with a
qualified name.

Given
  namespace N {
    namespace M {
      class D {};
    }
  }
```

```
  class C {};

  class C c;
  N::M::D d;

elaboratedType() matches the type of the variable declarations of both
c and d.
```

Matcher<Type> enumType Matcher<EnumType>...

```
Matches enum types.

Given
  enum C { Green };
  enum class S { Red };

  C c;
  S s;

enumType() matches the type of the variable declarations of both c and
s.
```

Matcher<Type> functionProtoType Matcher<FunctionProtoType>...

```
Matches FunctionProtoType nodes.

Given
  int (*f)(int);
  void g();
functionProtoType()
  matches "int (*f)(int)" and the type of "g" in C++ mode.
  In C mode, "g" is not matched because it does not contain a prototype.
```

Matcher<Type> functionType Matcher<FunctionType>...

```
Matches FunctionType nodes.

Given
  int (*f)(int);
  void g();
functionType()
  matches "int (*f)(int)" and the type of "g".
```

Matcher<Type> incompleteArrayType Matcher<IncompleteArrayType>...

```
Matches C arrays with unspecified size.

Given
  int a[] = { 2, 3 };
  int b[42];
  void f(int c[]) { int d[a[0]]; };
incompleteArrayType()
  matches "int a[]" and "int c[]"
```

Matcher<Type> injectedClassNameType Matcher<InjectedClassNameType>...

```
Matches injected class name types.

Example matches S s, but not S<T> s.
    (matcher = parmVarDecl(hasType(injectedClassNameType())))
  template <typename T> struct S {
    void f(S s);
    void g(S<T> s);
  };
```

Matcher<Type> lValueReferenceType Matcher<LValueReferenceType>...

```
Matches lvalue reference types.

Given:
  int *a;
  int &b = *a;
  int &&c = 1;
  auto &d = b;
  auto &&e = c;
  auto &&f = 2;
  int g = 5;

lValueReferenceType() matches the types of b, d, and e. e is
matched since the type is deduced as int& by reference collapsing rules.
```

Matcher<Type> memberPointerType Matcher<MemberPointerType>...

```
Matches member pointer types.
Given
  struct A { int i; }
  A::* ptr = A::i;
memberPointerType()
  matches "A::* ptr"
```

Matcher<Type> objcObjectPointerType Matcher<ObjCObjectPointerType>...

```
Matches an Objective-C object pointer type, which is different from
a pointer type, despite being syntactically similar.

Given
  int *a;

  @interface Foo
  @end
  Foo *f;
pointerType()
  matches "Foo *f", but does not match "int *a".
```

Matcher<Type> parenType Matcher<ParenType>...

```
Matches ParenType nodes.

Given
  int (*ptr_to_array)[4];
  int *array_of_ptrs[4];

varDecl(hasType(pointsTo(parenType()))) matches ptr_to_array but not
array_of_ptrs.
```

Matcher<Type> pointerType Matcher<PointerType>...

```
Matches pointer types, but does not match Objective-C object pointer
types.

Given
  int *a;
  int &b = *a;
  int c = 5;

  @interface Foo
  @end
  Foo *f;
pointerType()
  matches "int *a", but does not match "Foo *f".
```

Matcher<Type> rValueReferenceType Matcher<RValueReferenceType>...

```
Matches rvalue reference types.

Given:
  int *a;
  int &b = *a;
  int &&c = 1;
  auto &d = b;
  auto &&e = c;
  auto &&f = 2;
  int g = 5;

rValueReferenceType() matches the types of c and f. e is not
matched as it is deduced to int& by reference collapsing rules.
```

Matcher<Type> recordType Matcher<RecordType>...

```
Matches record types (e.g. structs, classes).

Given
  class C {};
  struct S {};

  C c;
  S s;

recordType() matches the type of the variable declarations of both c
and s.
```

Matcher<Type> referenceType Matcher<ReferenceType>...

```
Matches both lvalue and rvalue reference types.

Given
  int *a;
  int &b = *a;
  int &&c = 1;
  auto &d = b;
  auto &&e = c;
  auto &&f = 2;
  int g = 5;

referenceType() matches the types of b, c, d, e, and f.
```

Matcher<Type> substTemplateTypeParmType Matcher<SubstTemplateTypeParmType>...

```
Matches types that represent the result of substituting a type for a
template type parameter.

Given
  template <typename T>
  void F(T t) {
    int i = 1 + t;
  }

substTemplateTypeParmType() matches the type of 't' but not '1'
```

Matcher<Type> templateSpecializationType Matcher<TemplateSpecializationType>...

```
Matches template specialization types.

Given
  template <typename T>
  class C { };

  template class C<int>;  A
  C<char> var;            B

templateSpecializationType() matches the type of the explicit
instantiation in A and the type of the variable declaration in B.
```

Matcher<Type> templateTypeParmType Matcher<TemplateTypeParmType>...

```
Matches template type parameter types.

Example matches T, but not int.
    (matcher = templateTypeParmType())
  template <typename T> void f(int i);
```

Matcher<Type>	type	Matcher<Type>...

```
Matches Types in the clang AST.
```

Matcher<Type>	typedefType	Matcher<TypedefType>...

```
Matches typedef types.

Given
  typedef int X;
typedefType()
  matches "typedef int X"
```

Matcher<Type>	unaryTransformType	Matcher<UnaryTransformType>...

```
Matches types nodes representing unary type transformations.

Given:
  typedef __underlying_type(T) type;
unaryTransformType()
  matches "__underlying_type(T)"
```

Matcher<Type>	variableArrayType	Matcher<VariableArrayType>...

```
Matches C arrays with a specified size that is not an
integer-constant-expression.

Given
  void f() {
    int a[] = { 2, 3 }
    int b[42];
    int c[a[0]];
  }
variableArrayType()
  matches "int c[a[0]]"
```

Narrowing Matchers

Narrowing matchers match certain attributes on the current node, thus narrowing down the set of nodes of the current type to match on.

There are special logical narrowing matchers (allOf, anyOf, anything and unless) which allow users to create more powerful match expressions.

Return type	Name	Parameters
Matcher<*>	allOf	Matcher<*>, ..., Matcher<*>

```
Matches if all given matchers match.

Usable as: Any Matcher
```

Matcher<*>	anyOf	Matcher<*>, ..., Matcher<*>

```
Matches if any of the given matchers matches.

Usable as: Any Matcher
```

Matcher<*>	anything

```
Matches any node.

Useful when another matcher requires a child matcher, but there's no
additional constraint. This will often be used with an explicit conversion
to an internal::Matcher<> type such as TypeMatcher.

Example: DeclarationMatcher(anything()) matches all declarations, e.g.,
"int* p" and "void f()" in
  int* p;
  void f();

Usable as: Any Matcher
```

Matcher<*>	unless	Matcher<*>

```
Matches if the provided matcher does not match.

Example matches Y (matcher = cxxRecordDecl(unless(hasName("X"))))
  class X {};
  class Y {};

Usable as: Any Matcher
```

Matcher<BinaryOperator>	hasOperatorName	std::string Name

```
Matches the operator Name of operator expressions (binary or
unary).

Example matches a || b (matcher = binaryOperator(hasOperatorName("||")))
  !(a || b)
```

Matcher<CXXBoolLiteralExpr>	equals	ValueT Value

```
Matches literals that are equal to the given value of type ValueT.

Given
  f('false, 3.14, 42);
characterLiteral(equals(0))
  matches 'cxxBoolLiteral(equals(false)) and cxxBoolLiteral(equals(0))
```

```
  match false
floatLiteral(equals(3.14)) and floatLiteral(equals(314e-2))
  match 3.14
integerLiteral(equals(42))
  matches 42

Usable as: Matcher<CharacterLiteral>, Matcher<CXXBoolLiteralExpr>,
           Matcher<FloatingLiteral>, Matcher<IntegerLiteral>
```

Matcher<CXXBoolLiteralExpr>	equals	bool Value
Matcher<CXXBoolLiteralExpr>	equals	double Value
Matcher<CXXBoolLiteralExpr>	equals	unsigned Value
Matcher<CXXCatchStmt>	isCatchAll	

```
Matches a C++ catch statement that has a catch-all handler.

Given
  try {
    ...
  } catch (int) {
    ...
  } catch (...) {
    ...
  }
endcode
cxxCatchStmt(isCatchAll()) matches catch(...) but not catch(int).
```

Matcher<CXXConstructExpr>	argumentCountIs	unsigned N

```
Checks that a call expression or a constructor call expression has
a specific number of arguments (including absent default arguments).

Example matches f(0, 0) (matcher = callExpr(argumentCountIs(2)))
  void f(int x, int y);
  f(0, 0);
```

Matcher<CXXConstructExpr>	isListInitialization	

```
Matches a constructor call expression which uses list initialization.
```

Matcher<CXXConstructExpr>	requiresZeroInitialization	

```
Matches a constructor call expression which requires
zero initialization.

Given
void foo() {
  struct point { double x; double y; };
  point pt[2] = { { 1.0, 2.0 } };
}
initListExpr(has(cxxConstructExpr(requiresZeroInitialization())))
will match the implicit array filler for pt[1].
```

Matcher<CXXConstructorDecl>	isCopyConstructor	

```
Matches constructor declarations that are copy constructors.

Given
  struct S {
    S(); #1
    S(const S &); #2
    S(S &&); #3
  };
cxxConstructorDecl(isCopyConstructor()) will match #2, but not #1 or #3.
```

Matcher<CXXConstructorDecl>	isDefaultConstructor	

```
Matches constructor declarations that are default constructors.

Given
  struct S {
    S(); #1
    S(const S &); #2
    S(S &&); #3
  };
cxxConstructorDecl(isDefaultConstructor()) will match #1, but not #2 or #3.
```

Matcher<CXXConstructorDecl>	isDelegatingConstructor	

```
Matches constructors that delegate to another constructor.

Given
  struct S {
    S(); #1
    S(int) {} #2
    S(S &&) : S() {} #3
  };
  S::S() : S(0) {} #4
cxxConstructorDecl(isDelegatingConstructor()) will match #3 and #4, but not
#1 or #2.
```

Matcher<CXXConstructorDecl>	isExplicit	

```
Matches constructor and conversion declarations that are marked with
the explicit keyword.
```

```
Given
  struct S {
    S(int); #1
    explicit S(double); #2
    operator int(); #3
    explicit operator bool(); #4
  };
cxxConstructorDecl(isExplicit()) will match #2, but not #1.
cxxConversionDecl(isExplicit()) will match #4, but not #3.
```

Matcher<CXXConstructorDecl> isMoveConstructor

```
Matches constructor declarations that are move constructors.

Given
  struct S {
    S(); #1
    S(const S &); #2
    S(S &&); #3
  };
cxxConstructorDecl(isMoveConstructor()) will match #3, but not #1 or #2.
```

Matcher<CXXConversionDecl> isExplicit

```
Matches constructor and conversion declarations that are marked with
the explicit keyword.

Given
  struct S {
    S(int); #1
    explicit S(double); #2
    operator int(); #3
    explicit operator bool(); #4
  };
cxxConstructorDecl(isExplicit()) will match #2, but not #1.
cxxConversionDecl(isExplicit()) will match #4, but not #3.
```

Matcher<CXXCtorInitializer> isBaseInitializer

```
Matches a constructor initializer if it is initializing a base, as
opposed to a member.

Given
  struct B {};
  struct D : B {
    int I;
    D(int i) : I(i) {}
  };
  struct E : B {
    E() : B() {}
  };
cxxConstructorDecl(hasAnyConstructorInitializer(isBaseInitializer()))
  will match E(), but not match D(int).
```

Matcher<CXXCtorInitializer> isMemberInitializer

```
Matches a constructor initializer if it is initializing a member, as
opposed to a base.

Given
  struct B {};
  struct D : B {
    int I;
    D(int i) : I(i) {}
  };
  struct E : B {
    E() : B() {}
  };
cxxConstructorDecl(hasAnyConstructorInitializer(isMemberInitializer()))
  will match D(int), but not match E().
```

Matcher<CXXCtorInitializer> isWritten

```
Matches a constructor initializer if it is explicitly written in
code (as opposed to implicitly added by the compiler).

Given
  struct Foo {
    Foo() { }
    Foo(int) : foo_("A") { }
    string foo_;
  };
cxxConstructorDecl(hasAnyConstructorInitializer(isWritten()))
  will match Foo(int), but not Foo()
```

Matcher<CXXMethodDecl> isConst

```
Matches if the given method declaration is const.

Given
struct A {
  void foo() const;
  void bar();
};

cxxMethodDecl(isConst()) matches A::foo() but not A::bar()
```

Matcher<CXXMethodDecl> isCopyAssignmentOperator

```
Matches if the given method declaration declares a copy assignment
operator.

Given
struct A {
  A &operator=(const A &);
  A &operator=(A &&);
```

```
};

cxxMethodDecl(isCopyAssignmentOperator()) matches the first method but not
the second one.
```

Matcher<CXXMethodDecl> isFinal

```
Matches if the given method or class declaration is final.

Given:
  class A final {};

  struct B {
    virtual void f();
  };

  struct C : B {
    void f() final;
  };
matches A and C::f, but not B, C, or B::f
```

Matcher<CXXMethodDecl> isMoveAssignmentOperator

```
Matches if the given method declaration declares a move assignment
operator.

Given
struct A {
  A &operator=(const A &);
  A &operator=(A &&);
};

cxxMethodDecl(isMoveAssignmentOperator()) matches the second method but not
the first one.
```

Matcher<CXXMethodDecl> isOverride

```
Matches if the given method declaration overrides another method.

Given
  class A {
   public:
    virtual void x();
  };
  class B : public A {
   public:
    virtual void x();
  };
  matches B::x
```

Matcher<CXXMethodDecl> isPure

```
Matches if the given method declaration is pure.

Given
  class A {
   public:
    virtual void x() = 0;
  };
  matches A::x
```

Matcher<CXXMethodDecl> isUserProvided

```
Matches method declarations that are user-provided.

Given
  struct S {
    S(); #1
    S(const S &) = default; #2
    S(S &&) = delete; #3
  };
cxxConstructorDecl(isUserProvided()) will match #1, but not #2 or #3.
```

Matcher<CXXMethodDecl> isVirtual

```
Matches if the given method declaration is virtual.

Given
  class A {
   public:
    virtual void x();
  };
  matches A::x
```

Matcher<CXXMethodDecl> isVirtualAsWritten

```
Matches if the given method declaration has an explicit "virtual".

Given
  class A {
   public:
    virtual void x();
  };
  class B : public A {
   public:
    void x();
  };
  matches A::x but not B::x
```

Matcher<CXXOperatorCallExpr> hasOverloadedOperatorName StringRef Name

```
Matches overloaded operator names.

Matches overloaded operator names specified in strings without the
"operator" prefix: e.g. "<<".
```

```
Given:
  class A { int operator*(); };
  const A &operator<<(const A &a, const A &b);
  A a;
  a << a;   <-- This matches

cxxOperatorCallExpr(hasOverloadedOperatorName("<<"))) matches the
specified line and
cxxRecordDecl(hasMethod(hasOverloadedOperatorName("*")))
matches the declaration of A.

Usable as: Matcher<CXXOperatorCallExpr>, Matcher<FunctionDecl>
```

Matcher<CXXRecordDecl> isDerivedFrom std::string BaseName

```
Overloaded method as shortcut for isDerivedFrom(hasName(...)).
```

Matcher<CXXRecordDecl> isExplicitTemplateSpecialization

```
Matches explicit template specializations of function, class, or
static member variable template instantiations.

Given
  template<typename T> void A(T t) { }
  template<> void A(int N) { }
functionDecl(isExplicitTemplateSpecialization())
  matches the specialization A<int>().

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<CXXRecordDecl> isFinal

```
Matches if the given method or class declaration is final.

Given:
  class A final {};

  struct B {
    virtual void f();
  };

  struct C : B {
    void f() final;
  };
matches A and C::f, but not B, C, or B::f
```

Matcher<CXXRecordDecl> isLambda

```
Matches the generated class of lambda expressions.

Given:
  auto x = []{};

cxxRecordDecl(isLambda()) matches the implicit class declaration of
decltype(x)
```

Matcher<CXXRecordDecl> isSameOrDerivedFrom std::string BaseName

```
Overloaded method as shortcut for
isSameOrDerivedFrom(hasName(...)).
```

Matcher<CXXRecordDecl> isTemplateInstantiation

```
Matches template instantiations of function, class, or static
member variable template instantiations.

Given
  template <typename T> class X {}; class A {}; X<A> x;
or
  template <typename T> class X {}; class A {}; template class X<A>;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  matches the template instantiation of X<A>.

But given
  template <typename T>  class X {}; class A {};
  template <> class X<A> {}; X<A> x;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  does not match, as X<A> is an explicit template specialization.

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<CallExpr> argumentCountIs unsigned N

```
Checks that a call expression or a constructor call expression has
a specific number of arguments (including absent default arguments).

Example matches f(0, 0) (matcher = callExpr(argumentCountIs(2)))
  void f(int x, int y);
  f(0, 0);
```

Matcher<CastExpr> hasCastKind CastKind Kind

```
Matches casts that has a given cast kind.

Example: matches the implicit cast around 0
(matcher = castExpr(hasCastKind(CK_NullToPointer)))
  int *p = 0;
```

Matcher<CharacterLiteral> equals ValueT Value

```
Matches literals that are equal to the given value of type ValueT.

Given
  f('false, 3.14, 42);
characterLiteral(equals(0))
```

```
  matches 'cxxBoolLiteral(equals(false)) and cxxBoolLiteral(equals(0))
  match false
floatLiteral(equals(3.14)) and floatLiteral(equals(314e-2))
  match 3.14
integerLiteral(equals(42))
  matches 42

Usable as: Matcher<CharacterLiteral>, Matcher<CXXBoolLiteralExpr>,
           Matcher<FloatingLiteral>, Matcher<IntegerLiteral>
```

Matcher<CharacterLiteral>	equals	bool Value

Matcher<CharacterLiteral>	equals	double Value

Matcher<CharacterLiteral>	equals	unsigned Value

Matcher<ClassTemplateSpecializationDecl>	templateArgumentCountIs	unsigned N

```
Matches if the number of template arguments equals N.

Given
  template<typename T> struct C {};
  C<int> c;
classTemplateSpecializationDecl(templateArgumentCountIs(1))
  matches C<int>.
```

Matcher<CompoundStmt>	statementCountIs	unsigned N

```
Checks that a compound statement contains a specific number of
child statements.

Example: Given
  { for (;;) {} }
compoundStmt(statementCountIs(0)))
  matches '{}'
  but does not match the outer compound statement.
```

Matcher<ConstantArrayType>	hasSize	unsigned N

```
Matches nodes that have the specified size.

Given
  int a[42];
  int b[2 * 21];
  int c[41], d[43];
  char *s = "abcd";
  wchar_t *ws = L"abcd";
  char *w = "a";
constantArrayType(hasSize(42))
  matches "int a[42]" and "int b[2 * 21]"
stringLiteral(hasSize(4))
  matches "abcd", L"abcd"
```

Matcher<DeclStmt>	declCountIs	unsigned N

```
Matches declaration statements that contain a specific number of
declarations.

Example: Given
  int a, b;
  int c;
  int d = 2, e;
declCountIs(2)
  matches 'int a, b;' and 'int d = 2, e;', but not 'int c;'.
```

Matcher<Decl>	equalsBoundNode	std::string ID

```
Matches if a node equals a previously bound node.

Matches a node if it equals the node previously bound to ID.

Given
  class X { int a; int b; };
cxxRecordDecl(
    has(fieldDecl(hasName("a"), hasType(type().bind("t")))),
    has(fieldDecl(hasName("b"), hasType(type(equalsBoundNode("t"))))))
  matches the class X, as a and b have the same type.

Note that when multiple matches are involved via forEach* matchers,
equalsBoundNodes acts as a filter.
For example:
compoundStmt(
    forEachDescendant(varDecl().bind("d")),
    forEachDescendant(declRefExpr(to(decl(equalsBoundNode("d"))))))
will trigger a match for each combination of variable declaration
and reference to that variable declaration within a compound statement.
```

Matcher<Decl>	equalsNode	const Decl* Other

```
Matches if a node equals another node.

Decl has pointer identity in the AST.
```

Matcher<Decl>	hasAttr	attr::Kind AttrKind

```
Matches declaration that has a given attribute.

Given
  __attribute__((device)) void f() { ... }
```

```
decl(hasAttr(clang::attr::CUDADevice)) matches the function declaration of
f. If the matcher is use from clang-query, attr::Kind parameter should be
passed as a quoted string. e.g., hasAttr("attr::CUDADevice").
```

Matcher<Decl> isExpansionInFileMatching std::string RegExp

```
Matches AST nodes that were expanded within files whose name is
partially matching a given regex.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInFileMatching("AST.*"))
  #include "ASTMatcher.h"
  class X {};
ASTMatcher.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Decl> isExpansionInMainFile

```
Matches AST nodes that were expanded within the main-file.

Example matches X but not Y
  (matcher = cxxRecordDecl(isExpansionInMainFile())
  #include <Y.h>
  class X {};
Y.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Decl> isExpansionInSystemHeader

```
Matches AST nodes that were expanded within system-header-files.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInSystemHeader())
  #include <SystemHeader.h>
  class X {};
SystemHeader.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Decl> isImplicit

```
Matches a declaration that has been implicitly added
by the compiler (eg. implicit defaultcopy constructors).
```

Matcher<Decl> isPrivate

```
Matches private C++ declarations.

Given
  class C {
  public:    int a;
  protected: int b;
  private:   int c;
  };
fieldDecl(isPrivate())
  matches 'int c;'
```

Matcher<Decl> isProtected

```
Matches protected C++ declarations.

Given
  class C {
  public:    int a;
  protected: int b;
  private:   int c;
  };
fieldDecl(isProtected())
  matches 'int b;'
```

Matcher<Decl> isPublic

```
Matches public C++ declarations.

Given
  class C {
  public:    int a;
  protected: int b;
  private:   int c;
  };
fieldDecl(isPublic())
  matches 'int a;'
```

Matcher<DesignatedInitExpr> designatorCountIs unsigned N

```
Matches designated initializer expressions that contain
a specific number of designators.

Example: Given
  point ptarray[10] = { [2].y = 1.0, [0].x = 1.0 };
  point ptarray2[10] = { [2].y = 1.0, [2].x = 0.0, [0].x = 1.0 };
designatorCountIs(2)
  matches '{ [2].y = 1.0, [0].x = 1.0 }',
  but not '{ [2].y = 1.0, [2].x = 0.0, [0].x = 1.0 }'.
```

Matcher<FieldDecl> hasBitWidth unsigned Width

```
Matches non-static data members that are bit-fields of the specified
bit width.
```

```
Given
  class C {
    int a : 2;
    int b : 4;
    int c : 2;
  };
fieldDecl(hasBitWidth(2))
  matches 'int a;' and 'int c;' but not 'int b;'.
```

Matcher<FieldDecl> isBitField

```
Matches non-static data members that are bit-fields.

Given
  class C {
    int a : 2;
    int b;
  };
fieldDecl(isBitField())
  matches 'int a;' but not 'int b;'.
```

Matcher<FloatingLiteral> equals ValueT Value

```
Matches literals that are equal to the given value of type ValueT.

Given
  f('false, 3.14, 42);
characterLiteral(equals(0))
  matches 'cxxBoolLiteral(equals(false)) and cxxBoolLiteral(equals(0))
  match false
floatLiteral(equals(3.14)) and floatLiteral(equals(314e-2))
  match 3.14
integerLiteral(equals(42))
  matches 42

Usable as: Matcher<CharacterLiteral>, Matcher<CXXBoolLiteralExpr>,
           Matcher<FloatingLiteral>, Matcher<IntegerLiteral>
```

Matcher<FloatingLiteral> equals double Value

Matcher<FunctionDecl> hasDynamicExceptionSpec

```
Matches functions that have a dynamic exception specification.

Given:
  void f();
  void g() noexcept;
  void h() noexcept(true);
  void i() noexcept(false);
  void j() throw();
  void k() throw(int);
  void l() throw(...);
functionDecl(hasDynamicExceptionSpec()) and
  functionProtoType(hasDynamicExceptionSpec())
  match the declarations of j, k, and l, but not f, g, h, or i.
```

Matcher<FunctionDecl> hasOverloadedOperatorName StringRef Name

```
Matches overloaded operator names.

Matches overloaded operator names specified in strings without the
"operator" prefix: e.g. "<<".

Given:
  class A { int operator*(); };
  const A &operator<<(const A &a, const A &b);
  A a;
  a << a;   <-- This matches

cxxOperatorCallExpr(hasOverloadedOperatorName("<<"))) matches the
specified line and
cxxRecordDecl(hasMethod(hasOverloadedOperatorName("*")))
matches the declaration of A.

Usable as: Matcher<CXXOperatorCallExpr>, Matcher<FunctionDecl>
```

Matcher<FunctionDecl> isConstexpr

```
Matches constexpr variable and function declarations.

Given:
  constexpr int foo = 42;
  constexpr int bar();
varDecl(isConstexpr())
  matches the declaration of foo.
functionDecl(isConstexpr())
  matches the declaration of bar.
```

Matcher<FunctionDecl> isDefaulted

```
Matches defaulted function declarations.

Given:
  class A { ~A(); };
  class B { ~B() = default; };
functionDecl(isDefaulted())
  matches the declaration of ~B, but not ~A.
```

Matcher<FunctionDecl> isDefinition

```
Matches if a declaration has a body attached.

Example matches A, va, fa
  class A {};
```

```
  class B;  Doesn't match, as it has no body.
  int va;
  extern int vb;  Doesn't match, as it doesn't define the variable.
  void fa() {}
  void fb();  Doesn't match, as it has no body.

Usable as: Matcher<TagDecl>, Matcher<VarDecl>, Matcher<FunctionDecl>
```

Matcher<FunctionDecl> isDeleted

```
Matches deleted function declarations.

Given:
  void Func();
  void DeletedFunc() = delete;
functionDecl(isDeleted())
  matches the declaration of DeletedFunc, but not Func.
```

Matcher<FunctionDecl> isExplicitTemplateSpecialization

```
Matches explicit template specializations of function, class, or
static member variable template instantiations.

Given
  template<typename T> void A(T t) { }
  template<> void A(int N) { }
functionDecl(isExplicitTemplateSpecialization())
  matches the specialization A<int>().

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<FunctionDecl> isExternC

```
Matches extern "C" function declarations.

Given:
  extern "C" void f() {}
  extern "C" { void g() {} }
  void h() {}
functionDecl(isExternC())
  matches the declaration of f and g, but not the declaration h
```

Matcher<FunctionDecl> isInline

```
Matches function and namespace declarations that are marked with
the inline keyword.

Given
  inline void f();
  void g();
  namespace n {
  inline namespace m {}
  }
functionDecl(isInline()) will match ::f().
namespaceDecl(isInline()) will match n::m.
```

Matcher<FunctionDecl> isNoThrow

```
Matches functions that have a non-throwing exception specification.

Given:
  void f();
  void g() noexcept;
  void h() throw();
  void i() throw(int);
  void j() noexcept(false);
functionDecl(isNoThrow()) and functionProtoType(isNoThrow())
  match the declarations of g, and h, but not f, i or j.
```

Matcher<FunctionDecl> isStaticStorageClass

```
Matches variablefunction declarations that have "static" storage
class specifier ("static" keyword) written in the source.

Given:
  static void f() {}
  static int i = 0;
  extern int j;
  int k;
functionDecl(isStaticStorageClass())
  matches the function declaration f.
varDecl(isStaticStorageClass())
  matches the variable declaration i.
```

Matcher<FunctionDecl> isTemplateInstantiation

```
Matches template instantiations of function, class, or static
member variable template instantiations.

Given
  template <typename T> class X {}; class A {}; X<A> x;
or
  template <typename T> class X {}; class A {}; template class X<A>;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  matches the template instantiation of X<A>.

But given
  template <typename T>  class X {}; class A {};
  template <> class X<A> {}; X<A> x;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  does not match, as X<A> is an explicit template specialization.

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<FunctionDecl> isVariadic

```
Matches if a function declaration is variadic.

Example matches f, but not g or h. The function i will not match, even when
compiled in C mode.
  void f(...);
  void g(int);
  template <typename... Ts> void h(Ts...);
  void i();
```

Matcher<FunctionDecl> parameterCountIs unsigned N

```
Matches FunctionDecls and FunctionProtoTypes that have a
specific parameter count.

Given
  void f(int i) {}
  void g(int i, int j) {}
  void h(int i, int j);
  void j(int i);
  void k(int x, int y, int z, ...);
functionDecl(parameterCountIs(2))
  matches void g(int i, int j) {}
functionProtoType(parameterCountIs(2))
  matches void h(int i, int j)
functionProtoType(parameterCountIs(3))
  matches void k(int x, int y, int z, ...);
```

Matcher<FunctionProtoType> hasDynamicExceptionSpec

```
Matches functions that have a dynamic exception specification.

Given:
  void f();
  void g() noexcept;
  void h() noexcept(true);
  void i() noexcept(false);
  void j() throw();
  void k() throw(int);
  void l() throw(...);
functionDecl(hasDynamicExceptionSpec()) and
  functionProtoType(hasDynamicExceptionSpec())
  match the declarations of j, k, and l, but not f, g, h, or i.
```

Matcher<FunctionProtoType> isNoThrow

```
Matches functions that have a non-throwing exception specification.

Given:
  void f();
  void g() noexcept;
  void h() throw();
  void i() throw(int);
  void j() noexcept(false);
functionDecl(isNoThrow()) and functionProtoType(isNoThrow())
  match the declarations of g, and h, but not f, i or j.
```

Matcher<FunctionProtoType> parameterCountIs unsigned N

```
Matches FunctionDecls and FunctionProtoTypes that have a
specific parameter count.

Given
  void f(int i) {}
  void g(int i, int j) {}
  void h(int i, int j);
  void j(int i);
  void k(int x, int y, int z, ...);
functionDecl(parameterCountIs(2))
  matches void g(int i, int j) {}
functionProtoType(parameterCountIs(2))
  matches void h(int i, int j)
functionProtoType(parameterCountIs(3))
  matches void k(int x, int y, int z, ...);
```

Matcher<IntegerLiteral> equals ValueT Value

```
Matches literals that are equal to the given value of type ValueT.

Given
  f('false, 3.14, 42);
characterLiteral(equals(0))
  matches 'cxxBoolLiteral(equals(false)) and cxxBoolLiteral(equals(0))
  match false
floatLiteral(equals(3.14)) and floatLiteral(equals(314e-2))
  match 3.14
integerLiteral(equals(42))
  matches 42

Usable as: Matcher<CharacterLiteral>, Matcher<CXXBoolLiteralExpr>,
           Matcher<FloatingLiteral>, Matcher<IntegerLiteral>
```

Matcher<IntegerLiteral> equals bool Value

Matcher<IntegerLiteral> equals double Value

Matcher<IntegerLiteral> equals unsigned Value

Matcher<MemberExpr> isArrow

```
Matches member expressions that are called with '->' as opposed
```

```
to '.'.

Member calls on the implicit this pointer match as called with '->'.

Given
  class Y {
    void x() { this->x(); x(); Y y; y.x(); a; this->b; Y::b; }
    int a;
    static int b;
  };
memberExpr(isArrow())
  matches this->x, x, y.x, a, this->b
```

Matcher<NamedDecl> hasExternalFormalLinkage

```
Matches a declaration that has external formal linkage.

Example matches only z (matcher = varDecl(hasExternalFormalLinkage()))
void f() {
  int x;
  static int y;
}
int z;

Example matches f() because it has external formal linkage despite being
unique to the translation unit as though it has internal likage
(matcher = functionDecl(hasExternalFormalLinkage()))

namespace {
void f() {}
}
```

Matcher<NamedDecl> hasName std::string Name

```
Matches NamedDecl nodes that have the specified name.

Supports specifying enclosing namespaces or classes by prefixing the name
with '<enclosing>::'.
Does not match typedefs of an underlying type with the given name.

Example matches X (Name == "X")
  class X;

Example matches X (Name is one of "::a::b::X", "a::b::X", "b::X", "X")
  namespace a { namespace b { class X; } }
```

Matcher<NamedDecl> matchesName std::string RegExp

```
Matches NamedDecl nodes whose fully qualified names contain
a substring matched by the given RegExp.

Supports specifying enclosing namespaces or classes by
prefixing the name with '<enclosing>::'.  Does not match typedefs
of an underlying type with the given name.

Example matches X (regexp == "::X")
  class X;

Example matches X (regexp is one of "::X", "^foo::.*X", among others)
  namespace foo { namespace bar { class X; } }
```

Matcher<NamespaceDecl> isAnonymous

```
Matches anonymous namespace declarations.

Given
  namespace n {
  namespace {} #1
  }
namespaceDecl(isAnonymous()) will match #1 but not ::n.
```

Matcher<NamespaceDecl> isInline

```
Matches function and namespace declarations that are marked with
the inline keyword.

Given
  inline void f();
  void g();
  namespace n {
  inline namespace m {}
  }
functionDecl(isInline()) will match ::f().
namespaceDecl(isInline()) will match n::m.
```

Matcher<ObjCMessageExpr> argumentCountIs unsigned N

```
Checks that a call expression or a constructor call expression has
a specific number of arguments (including absent default arguments).

Example matches f(0, 0) (matcher = callExpr(argumentCountIs(2)))
  void f(int x, int y);
  f(0, 0);
```

Matcher<ObjCMessageExpr> hasKeywordSelector

```
Matches when the selector is a keyword selector

objCMessageExpr(hasKeywordSelector()) matches the generated setFrame
message expression in

  UIWebView *webView = ...;
  CGRect bodyFrame = webView.frame;
  bodyFrame.size.height = self.bodyContentHeight;
  webView.frame = bodyFrame;
      ^---- matches here
```

Matcher<ObjCMessageExpr> hasNullSelector

```
Matches when the selector is the empty selector

Matches only when the selector of the objCMessageExpr is NULL. This may
represent an error condition in the tree!
```

Matcher<ObjCMessageExpr> hasSelector std::string BaseName

```
Matches when BaseName == Selector.getAsString()

 matcher = objCMessageExpr(hasSelector("loadHTMLString:baseURL:"));
 matches the outer message expr in the code below, but NOT the message
 invocation for self.bodyView.
    [self.bodyView loadHTMLString:html baseURL:NULL];
```

Matcher<ObjCMessageExpr> hasUnarySelector

```
Matches when the selector is a Unary Selector

 matcher = objCMessageExpr(matchesSelector(hasUnarySelector());
 matches self.bodyView in the code below, but NOT the outer message
 invocation of "loadHTMLString:baseURL:".
    [self.bodyView loadHTMLString:html baseURL:NULL];
```

Matcher<ObjCMessageExpr> matchesSelector std::string RegExp

```
Matches ObjC selectors whose name contains
a substring matched by the given RegExp.
 matcher = objCMessageExpr(matchesSelector("loadHTMLStringmatches the outer message expr in the code below, but NOT the message
 invocation for self.bodyView.
    [self.bodyView loadHTMLString:html baseURL:NULL];
```

Matcher<ObjCMessageExpr> numSelectorArgs unsigned N

```
Matches when the selector has the specified number of arguments

 matcher = objCMessageExpr(numSelectorArgs(0));
 matches self.bodyView in the code below

 matcher = objCMessageExpr(numSelectorArgs(2));
 matches the invocation of "loadHTMLString:baseURL:" but not that
 of self.bodyView
    [self.bodyView loadHTMLString:html baseURL:NULL];
```

Matcher<QualType> asString std::string Name

```
Matches if the matched type is represented by the given string.

Given
  class Y { public: void x(); };
  void z() { Y* y; y->x(); }
cxxMemberCallExpr(on(hasType(asString("class Y *"))))
  matches y->x()
```

Matcher<QualType> equalsBoundNode std::string ID

```
Matches if a node equals a previously bound node.

Matches a node if it equals the node previously bound to ID.

Given
  class X { int a; int b; };
cxxRecordDecl(
    has(fieldDecl(hasName("a"), hasType(type().bind("t")))),
    has(fieldDecl(hasName("b"), hasType(type(equalsBoundNode("t"))))))
  matches the class X, as a and b have the same type.

Note that when multiple matches are involved via forEach* matchers,
equalsBoundNodes acts as a filter.
For example:
compoundStmt(
    forEachDescendant(varDecl().bind("d")),
    forEachDescendant(declRefExpr(to(decl(equalsBoundNode("d"))))))
will trigger a match for each combination of variable declaration
and reference to that variable declaration within a compound statement.
```

Matcher<QualType> hasLocalQualifiers

```
Matches QualType nodes that have local CV-qualifiers attached to
the node, not hidden within a typedef.

Given
  typedef const int const_int;
  const_int i;
  int *const j;
  int *volatile k;
  int m;
varDecl(hasType(hasLocalQualifiers())) matches only j and k.
i is const-qualified but the qualifier is not local.
```

Matcher<QualType> isAnyCharacter

```
Matches QualType nodes that are of character type.

Given
  void a(char);
  void b(wchar_t);
  void c(double);
functionDecl(hasAnyParameter(hasType(isAnyCharacter())))
matches "a(char)", "b(wchar_t)", but not "c(double)".
```

Matcher<QualType> isAnyPointer

```
Matches QualType nodes that are of any pointer type; this includes
the Objective-C object pointer type, which is different despite being
syntactically similar.

Given
  int *i = nullptr;

  @interface Foo
  @end
  Foo *f;

  int j;
varDecl(hasType(isAnyPointer()))
  matches "int *i" and "Foo *f", but not "int j".
```

Matcher<QualType> isConstQualified

```
Matches QualType nodes that are const-qualified, i.e., that
include "top-level" const.

Given
  void a(int);
  void b(int const);
  void c(const int);
  void d(const int*);
  void e(int const) {};
functionDecl(hasAnyParameter(hasType(isConstQualified())))
  matches "void b(int const)", "void c(const int)" and
  "void e(int const) {}". It does not match d as there
  is no top-level const on the parameter type "const int *".
```

Matcher<QualType> isInteger

```
Matches QualType nodes that are of integer type.

Given
  void a(int);
  void b(long);
  void c(double);
functionDecl(hasAnyParameter(hasType(isInteger())))
matches "a(int)", "b(long)", but not "c(double)".
```

Matcher<QualType> isSignedInteger

```
Matches QualType nodes that are of signed integer type.

Given
  void a(int);
  void b(unsigned long);
  void c(double);
functionDecl(hasAnyParameter(hasType(isSignedInteger())))
matches "a(int)", but not "b(unsigned long)" and "c(double)".
```

Matcher<QualType> isUnsignedInteger

```
Matches QualType nodes that are of unsigned integer type.

Given
  void a(int);
  void b(unsigned long);
  void c(double);
functionDecl(hasAnyParameter(hasType(isUnsignedInteger())))
matches "b(unsigned long)", but not "a(int)" and "c(double)".
```

Matcher<QualType> isVolatileQualified

```
Matches QualType nodes that are volatile-qualified, i.e., that
include "top-level" volatile.

Given
  void a(int);
  void b(int volatile);
  void c(volatile int);
  void d(volatile int*);
  void e(int volatile) {};
functionDecl(hasAnyParameter(hasType(isVolatileQualified())))
  matches "void b(int volatile)", "void c(volatile int)" and
  "void e(int volatile) {}". It does not match d as there
  is no top-level volatile on the parameter type "volatile int *".
```

Matcher<RecordDecl> isClass

```
Matches RecordDecl object that are spelled with "class."

Example matches C, but not S or U.
  struct S {};
  class C {};
  union U {};
```

Matcher<RecordDecl> isStruct

```
Matches RecordDecl object that are spelled with "struct."

Example matches S, but not C or U.
  struct S {};
  class C {};
  union U {};
```

Matcher<RecordDecl> isUnion

```
Matches RecordDecl object that are spelled with "union."

Example matches U, but not C or S.
  struct S {};
  class C {};
  union U {};
```

Matcher<Stmt> equalsBoundNode std::string ID

```
Matches if a node equals a previously bound node.

Matches a node if it equals the node previously bound to ID.

Given
  class X { int a; int b; };
cxxRecordDecl(
    has(fieldDecl(hasName("a"), hasType(type().bind("t")))),
    has(fieldDecl(hasName("b"), hasType(type(equalsBoundNode("t"))))))
  matches the class X, as a and b have the same type.

Note that when multiple matches are involved via forEach* matchers,
equalsBoundNodes acts as a filter.
For example:
compoundStmt(
    forEachDescendant(varDecl().bind("d")),
    forEachDescendant(declRefExpr(to(decl(equalsBoundNode("d"))))))
will trigger a match for each combination of variable declaration
and reference to that variable declaration within a compound statement.
```

Matcher<Stmt> equalsNode const Stmt* Other

```
Matches if a node equals another node.

Stmt has pointer identity in the AST.
```

Matcher<Stmt> isExpansionInFileMatching std::string RegExp

```
Matches AST nodes that were expanded within files whose name is
partially matching a given regex.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInFileMatching("AST.*"))
  #include "ASTMatcher.h"
  class X {};
ASTMatcher.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Stmt> isExpansionInMainFile

```
Matches AST nodes that were expanded within the main-file.

Example matches X but not Y
  (matcher = cxxRecordDecl(isExpansionInMainFile())
  #include <Y.h>
  class X {};
Y.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Stmt> isExpansionInSystemHeader

```
Matches AST nodes that were expanded within system-header-files.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInSystemHeader())
  #include <SystemHeader.h>
  class X {};
SystemHeader.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<StringLiteral> hasSize unsigned N

```
Matches nodes that have the specified size.

Given
  int a[42];
  int b[2 * 21];
  int c[41], d[43];
  char *s = "abcd";
  wchar_t *ws = L"abcd";
  char *w = "a";
constantArrayType(hasSize(42))
  matches "int a[42]" and "int b[2 * 21]"
stringLiteral(hasSize(4))
  matches "abcd", L"abcd"
```

Matcher<TagDecl> isDefinition

```
Matches if a declaration has a body attached.

Example matches A, va, fa
  class A {};
  class B;  Doesn't match, as it has no body.
  int va;
  extern int vb;  Doesn't match, as it doesn't define the variable.
  void fa() {}
  void fb();  Doesn't match, as it has no body.

Usable as: Matcher<TagDecl>, Matcher<VarDecl>, Matcher<FunctionDecl>
```

Matcher<TemplateArgument> equalsIntegralValue std::string Value

```
Matches a TemplateArgument of integral type with a given value.

Note that 'Value' is a string as the template argument's value is
an arbitrary precision integer. 'Value' must be euqal to the canonical
representation of that integral value in base 10.
```

```
Given
  template<int T> struct A {};
  C<42> c;
classTemplateSpecializationDecl(
  hasAnyTemplateArgument(equalsIntegralValue("42")))
  matches the implicit instantiation of C in C<42>.
```

Matcher<TemplateArgument> isIntegral

```
Matches a TemplateArgument that is an integral value.

Given
  template<int T> struct A {};
  C<42> c;
classTemplateSpecializationDecl(
  hasAnyTemplateArgument(isIntegral()))
  matches the implicit instantiation of C in C<42>
  with isIntegral() matching 42.
```

Matcher<TemplateSpecializationType> templateArgumentCountIs unsigned N

```
Matches if the number of template arguments equals N.

Given
  template<typename T> struct C {};
  C<int> c;
classTemplateSpecializationDecl(templateArgumentCountIs(1))
  matches C<int>.
```

Matcher<TypeLoc> isExpansionInFileMatching std::string RegExp

```
Matches AST nodes that were expanded within files whose name is
partially matching a given regex.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInFileMatching("AST.*"))
  #include "ASTMatcher.h"
  class X {};
ASTMatcher.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<TypeLoc> isExpansionInMainFile

```
Matches AST nodes that were expanded within the main-file.

Example matches X but not Y
  (matcher = cxxRecordDecl(isExpansionInMainFile())
  #include <Y.h>
  class X {};
Y.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<TypeLoc> isExpansionInSystemHeader

```
Matches AST nodes that were expanded within system-header-files.

Example matches Y but not X
    (matcher = cxxRecordDecl(isExpansionInSystemHeader())
  #include <SystemHeader.h>
  class X {};
SystemHeader.h:
  class Y {};

Usable as: Matcher<Decl>, Matcher<Stmt>, Matcher<TypeLoc>
```

Matcher<Type> booleanType

```
Matches type bool.

Given
 struct S { bool func(); };
functionDecl(returns(booleanType()))
  matches "bool func();"
```

Matcher<Type> equalsBoundNode std::string ID

```
Matches if a node equals a previously bound node.

Matches a node if it equals the node previously bound to ID.

Given
  class X { int a; int b; };
cxxRecordDecl(
    has(fieldDecl(hasName("a"), hasType(type().bind("t")))),
    has(fieldDecl(hasName("b"), hasType(type(equalsBoundNode("t"))))))
  matches the class X, as a and b have the same type.

Note that when multiple matches are involved via forEach* matchers,
equalsBoundNodes acts as a filter.
For example:
compoundStmt(
    forEachDescendant(varDecl().bind("d")),
    forEachDescendant(declRefExpr(to(decl(equalsBoundNode("d"))))))
will trigger a match for each combination of variable declaration
and reference to that variable declaration within a compound statement.
```

Matcher<Type> equalsNode const Type* Other

```
Matches if a node equals another node.

Type has pointer identity in the AST.
```

Matcher<Type> realFloatingPointType

```
Matches any real floating-point type (float, double, long double).

Given
  int i;
  float f;
realFloatingPointType()
  matches "float f" but not "int i"
```

Matcher<Type> voidType

```
Matches type void.

Given
 struct S { void func(); };
functionDecl(returns(voidType()))
  matches "void func();"
```

Matcher<UnaryExprOrTypeTraitExpr> ofKind UnaryExprOrTypeTrait Kind

```
Matches unary expressions of a certain kind.

Given
  int x;
  int s = sizeof(x) + alignof(x)
unaryExprOrTypeTraitExpr(ofKind(UETT_SizeOf))
  matches sizeof(x)
```

Matcher<UnaryOperator> hasOperatorName std::string Name

```
Matches the operator Name of operator expressions (binary or
unary).

Example matches a || b (matcher = binaryOperator(hasOperatorName("||")))
  !(a || b)
```

Matcher<VarDecl> hasAutomaticStorageDuration

```
Matches a variable declaration that has automatic storage duration.

Example matches x, but not y, z, or a.
(matcher = varDecl(hasAutomaticStorageDuration())
void f() {
  int x;
  static int y;
  thread_local int z;
}
int a;
```

Matcher<VarDecl> hasGlobalStorage

```
Matches a variable declaration that does not have local storage.

Example matches y and z (matcher = varDecl(hasGlobalStorage())
void f() {
  int x;
  static int y;
}
int z;
```

Matcher<VarDecl> hasLocalStorage

```
Matches a variable declaration that has function scope and is a
non-static local variable.

Example matches x (matcher = varDecl(hasLocalStorage())
void f() {
  int x;
  static int y;
}
int z;
```

Matcher<VarDecl> hasStaticStorageDuration

```
Matches a variable declaration that has static storage duration.
It includes the variable declared at namespace scope and those declared
with "static" and "extern" storage class specifiers.

void f() {
  int x;
  static int y;
  thread_local int z;
}
int a;
static int b;
extern int c;
varDecl(hasStaticStorageDuration())
  matches the function declaration y, a, b and c.
```

Matcher<VarDecl> hasThreadStorageDuration

```
Matches a variable declaration that has thread storage duration.

Example matches z, but not x, z, or a.
(matcher = varDecl(hasThreadStorageDuration())
void f() {
  int x;
  static int y;
  thread_local int z;
}
int a;
```

Matcher<VarDecl> isConstexpr

```
Matches constexpr variable and function declarations.

Given:
  constexpr int foo = 42;
  constexpr int bar();
varDecl(isConstexpr())
  matches the declaration of foo.
functionDecl(isConstexpr())
  matches the declaration of bar.
```

Matcher<VarDecl> isDefinition

```
Matches if a declaration has a body attached.

Example matches A, va, fa
  class A {};
  class B;  Doesn't match, as it has no body.
  int va;
  extern int vb;  Doesn't match, as it doesn't define the variable.
  void fa() {}
  void fb();  Doesn't match, as it has no body.

Usable as: Matcher<TagDecl>, Matcher<VarDecl>, Matcher<FunctionDecl>
```

Matcher<VarDecl> isExceptionVariable

```
Matches a variable declaration that is an exception variable from
a C++ catch block, or an Objective-C statement.

Example matches x (matcher = varDecl(isExceptionVariable())
void f(int y) {
  try {
  } catch (int x) {
  }
}
```

Matcher<VarDecl> isExplicitTemplateSpecialization

```
Matches explicit template specializations of function, class, or
static member variable template instantiations.

Given
  template<typename T> void A(T t) { }
  template<> void A(int N) { }
functionDecl(isExplicitTemplateSpecialization())
  matches the specialization A<int>().

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<VarDecl> isExternC

```
Matches extern "C" function declarations.

Given:
  extern "C" void f() {}
  extern "C" { void g() {} }
  void h() {}
functionDecl(isExternC())
  matches the declaration of f and g, but not the declaration h
```

Matcher<VarDecl> isStaticStorageClass

```
Matches variablefunction declarations that have "static" storage
class specifier ("static" keyword) written in the source.

Given:
  static void f() {}
  static int i = 0;
  extern int j;
  int k;
functionDecl(isStaticStorageClass())
  matches the function declaration f.
varDecl(isStaticStorageClass())
  matches the variable declaration i.
```

Matcher<VarDecl> isTemplateInstantiation

```
Matches template instantiations of function, class, or static
member variable template instantiations.

Given
  template <typename T> class X {}; class A {}; X<A> x;
or
  template <typename T> class X {}; class A {}; template class X<A>;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  matches the template instantiation of X<A>.

But given
  template <typename T>  class X {}; class A {};
  template <> class X<A> {}; X<A> x;
cxxRecordDecl(hasName("::X"), isTemplateInstantiation())
  does not match, as X<A> is an explicit template specialization.

Usable as: Matcher<FunctionDecl>, Matcher<VarDecl>, Matcher<CXXRecordDecl>
```

Matcher<internal::Matcher<Decl>> isInstantiated

```
Matches declarations that are template instantiations or are inside
template instantiations.

Given
  template<typename T> void A(T t) { T i; }
  A(0);
  A(0U);
```

```
functionDecl(isInstantiated())
  matches 'A(int) {...};' and 'A(unsigned) {...}'.
```

Matcher<internal::Matcher<Expr>>	nullPointerConstant	

```
Matches expressions that resolve to a null pointer constant, such as
GNU's __null, C++11's nullptr, or C's NULL macro.

Given:
  void *v1 = NULL;
  void *v2 = nullptr;
  void *v3 = __null; GNU extension
  char *cp = (char *)0;
  int *ip = 0;
  int i = 0;
expr(nullPointerConstant())
  matches the initializer for v1, v2, v3, cp, and ip. Does not match the
  initializer for i.
```

Matcher<internal::Matcher<NamedDecl>>	hasAnyName	StringRef, ..., StringRef

```
Matches NamedDecl nodes that have any of the specified names.

This matcher is only provided as a performance optimization of hasName.
    hasAnyName(a, b, c)
 is equivalent to, but faster than
    anyOf(hasName(a), hasName(b), hasName(c))
```

Matcher<internal::Matcher<Stmt>>	isInTemplateInstantiation	

```
Matches statements inside of a template instantiation.

Given
  int j;
  template<typename T> void A(T t) { T i; j += 42;}
  A(0);
  A(0U);
declStmt(isInTemplateInstantiation())
  matches 'int i;' and 'unsigned i'.
unless(stmt(isInTemplateInstantiation()))
  will NOT match j += 42; as it's shared between the template definition and
  instantiation.
```

AST Traversal Matchers

Traversal matchers specify the relationship to other nodes that are reachable from the current node.

Note that there are special traversal matchers (has, hasDescendant, forEach and forEachDescendant) which work on all nodes and allow users to write more generic match expressions.

Return type	Name	Parameters
Matcher<*>	eachOf	Matcher<*>, ..., Matcher<*>

```
Matches if any of the given matchers matches.

Unlike anyOf, eachOf will generate a match result for each
matching submatcher.

For example, in:
  class A { int a; int b; };
The matcher:
  cxxRecordDecl(eachOf(has(fieldDecl(hasName("a")).bind("v")),
                       has(fieldDecl(hasName("b")).bind("v"))))
will generate two results binding "v", the first of which binds
the field declaration of a, the second the field declaration of
b.

Usable as: Any Matcher
```

Matcher<*>	forEachDescendant	Matcher<*>

```
Matches AST nodes that have descendant AST nodes that match the
provided matcher.

Example matches X, A, B, C
  (matcher = cxxRecordDecl(forEachDescendant(cxxRecordDecl(hasName("X")))))
  class X {};  Matches X, because X::X is a class of name X inside X.
  class A { class X {}; };
  class B { class C { class X {}; }; };

DescendantT must be an AST base type.

As opposed to 'hasDescendant', 'forEachDescendant' will cause a match for
each result that matches instead of only on the first one.

Note: Recursively combined ForEachDescendant can cause many matches:
  cxxRecordDecl(forEachDescendant(cxxRecordDecl(
    forEachDescendant(cxxRecordDecl())
  )))
will match 10 times (plus injected class name matches) on:
  class A { class B { class C { class D { class E {}; }; }; }; };

Usable as: Any Matcher
```

Matcher<*>	forEach	Matcher<*>

```
Matches AST nodes that have child AST nodes that match the
provided matcher.
```

```
Example matches X, Y
  (matcher = cxxRecordDecl(forEach(cxxRecordDecl(hasName("X"))))
  class X {};  Matches X, because X::X is a class of name X inside X.
  class Y { class X {}; };
  class Z { class Y { class X {}; }; };  Does not match Z.

ChildT must be an AST base type.

As opposed to 'has', 'forEach' will cause a match for each result that
matches instead of only on the first one.

Usable as: Any Matcher
```

Matcher<*> hasAncestor Matcher<*>

```
Matches AST nodes that have an ancestor that matches the provided
matcher.

Given
void f() { if (true) { int x = 42; } }
void g() { for (;;) { int x = 43; } }
expr(integerLiteral(hasAncestor(ifStmt()))) matches 42, but not 43.

Usable as: Any Matcher
```

Matcher<*> hasDescendant Matcher<*>

```
Matches AST nodes that have descendant AST nodes that match the
provided matcher.

Example matches X, Y, Z
    (matcher = cxxRecordDecl(hasDescendant(cxxRecordDecl(hasName("X")))))
  class X {};  Matches X, because X::X is a class of name X inside X.
  class Y { class X {}; };
  class Z { class Y { class X {}; }; };

DescendantT must be an AST base type.

Usable as: Any Matcher
```

Matcher<*> has Matcher<*>

```
Matches AST nodes that have child AST nodes that match the
provided matcher.

Example matches X, Y
  (matcher = cxxRecordDecl(has(cxxRecordDecl(hasName("X"))))
  class X {};  Matches X, because X::X is a class of name X inside X.
  class Y { class X {}; };
  class Z { class Y { class X {}; }; };  Does not match Z.

ChildT must be an AST base type.

Usable as: Any Matcher
Note that has is direct matcher, so it also matches things like implicit
casts and paren casts. If you are matching with expr then you should
probably consider using ignoringParenImpCasts like:
has(ignoringParenImpCasts(expr())).
```

Matcher<*> hasParent Matcher<*>

```
Matches AST nodes that have a parent that matches the provided
matcher.

Given
void f() { for (;;) { int x = 42; if (true) { int x = 43; } } }
compoundStmt(hasParent(ifStmt())) matches "{ int x = 43; }".

Usable as: Any Matcher
```

Matcher<AbstractConditionalOperator> hasCondition Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<AbstractConditionalOperator> hasFalseExpression Matcher<Expr> InnerMatcher

```
Matches the false branch expression of a conditional operator
(binary or ternary).

Example matches b
  condition ? a : b
  condition ?: b
```

Matcher<AbstractConditionalOperator> hasTrueExpression Matcher<Expr> InnerMatcher

```
Matches the true branch expression of a conditional operator.

Example 1 (conditional ternary operator): matches a
  condition ? a : b

Example 2 (conditional binary operator): matches opaqueValueExpr(condition)
  condition ?: b
```

Matcher<AddrLabelExpr> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
```

```
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<ArraySubscriptExpr> hasBase Matcher<Expr> InnerMatcher

```
Matches the base expression of an array subscript expression.

Given
  int i[5];
  void f() { i[1] = 42; }
arraySubscriptExpression(hasBase(implicitCastExpr(
    hasSourceExpression(declRefExpr()))))
  matches i[1] with the declRefExpr() matching i
```

Matcher<ArraySubscriptExpr> hasIndex Matcher<Expr> InnerMatcher

```
Matches the index expression of an array subscript expression.

Given
  int i[5];
  void f() { i[1] = 42; }
arraySubscriptExpression(hasIndex(integerLiteral()))
  matches i[1] with the integerLiteral() matching 1
```

Matcher<ArraySubscriptExpr> hasLHS Matcher<Expr> InnerMatcher

```
Matches the left hand side of binary operator expressions.

Example matches a (matcher = binaryOperator(hasLHS()))
  a || b
```

Matcher<ArraySubscriptExpr> hasRHS Matcher<Expr> InnerMatcher

```
Matches the right hand side of binary operator expressions.

Example matches b (matcher = binaryOperator(hasRHS()))
  a || b
```

Matcher<ArrayTypeLoc> hasElementTypeLoc Matcher<TypeLoc>

```
Matches arrays and C99 complex types that have a specific element
type.

Given
  struct A {};
  A a[7];
  int b[7];
arrayType(hasElementType(builtinType()))
  matches "int b[7]"

Usable as: Matcher<ArrayType>, Matcher<ComplexType>
```

Matcher<ArrayType> hasElementType Matcher<Type>

```
Matches arrays and C99 complex types that have a specific element
type.

Given
  struct A {};
  A a[7];
  int b[7];
arrayType(hasElementType(builtinType()))
  matches "int b[7]"

Usable as: Matcher<ArrayType>, Matcher<ComplexType>
```

Matcher<AtomicTypeLoc> hasValueTypeLoc Matcher<TypeLoc>

```
Matches atomic types with a specific value type.

Given
  _Atomic(int) i;
  _Atomic(float) f;
atomicType(hasValueType(isInteger()))
 matches "_Atomic(int) i"

Usable as: Matcher<AtomicType>
```

Matcher<AtomicType> hasValueType Matcher<Type>

```
Matches atomic types with a specific value type.

Given
  _Atomic(int) i;
  _Atomic(float) f;
atomicType(hasValueType(isInteger()))
 matches "_Atomic(int) i"

Usable as: Matcher<AtomicType>
```

Matcher<AutoType> hasDeducedType Matcher<Type>

Matches AutoType nodes where the deduced type is a specific type.

Note: There is no TypeLoc for the deduced type and thus no
getDeducedLoc() matcher.

Given
  auto a = 1;
  auto b = 2.0;
autoType(hasDeducedType(isInteger()))
  matches "auto a"

Usable as: Matcher<AutoType>

Matcher<BinaryOperator>	hasEitherOperand	Matcher<Expr> InnerMatcher

Matches if either the left hand side or the right hand side of a
binary operator matches.

Matcher<BinaryOperator>	hasLHS	Matcher<Expr> InnerMatcher

Matches the left hand side of binary operator expressions.

Example matches a (matcher = binaryOperator(hasLHS()))
  a || b

Matcher<BinaryOperator>	hasRHS	Matcher<Expr> InnerMatcher

Matches the right hand side of binary operator expressions.

Example matches b (matcher = binaryOperator(hasRHS()))
  a || b

Matcher<BlockPointerTypeLoc>	pointeeLoc	Matcher<TypeLoc>

Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>

Matcher<BlockPointerType>	pointee	Matcher<Type>

Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>

Matcher<CXXConstructExpr>	forEachArgumentWithParam	Matcher<Expr> ArgMatcher, Matcher<ParmVarDecl> ParamMatcher

Matches all arguments and their respective ParmVarDecl.

Given
  void f(int i);
  int y;
  f(y);
callExpr(
  forEachArgumentWithParam(
    declRefExpr(to(varDecl(hasName("y")))),
    parmVarDecl(hasType(isInteger()))
))
  matches f(y);
with declRefExpr(...)
  matching int y
and parmVarDecl(...)
  matching int i

Matcher<CXXConstructExpr>	hasAnyArgument	Matcher<Expr> InnerMatcher

Matches any argument of a call expression or a constructor call
expression.

Given
  void x(int, int, int) { int y; x(1, y, 42); }
callExpr(hasAnyArgument(declRefExpr()))
  matches x(1, y, 42)
with hasAnyArgument(...)
  matching y

Matcher<CXXConstructExpr>	hasArgument	unsigned N, Matcher<Expr> InnerMatcher

Matches the n'th argument of a call expression or a constructor
call expression.

Example matches y in x(y)
    (matcher = callExpr(hasArgument(0, declRefExpr())))
  void x(int) { int y; x(y); }

Matcher<CXXConstructExpr> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<CXXConstructorDecl> forEachConstructorInitializer Matcher<CXXCtorInitializer> InnerMatcher

```
Matches each constructor initializer in a constructor definition.

Given
  class A { A() : i(42), j(42) {} int i; int j; };
cxxConstructorDecl(forEachConstructorInitializer(
  forField(decl().bind("x"))
))
  will trigger two matches, binding for 'i' and 'j' respectively.
```

Matcher<CXXConstructorDecl> hasAnyConstructorInitializer Matcher<CXXCtorInitializer> InnerMatcher

```
Matches a constructor initializer.

Given
  struct Foo {
    Foo() : foo_(1) { }
    int foo_;
  };
cxxRecordDecl(has(cxxConstructorDecl(
  hasAnyConstructorInitializer(anything())
)))
  record matches Foo, hasAnyConstructorInitializer matches foo_(1)
```

Matcher<CXXCtorInitializer> forField Matcher<FieldDecl> InnerMatcher

```
Matches the field declaration of a constructor initializer.

Given
  struct Foo {
    Foo() : foo_(1) { }
    int foo_;
  };
cxxRecordDecl(has(cxxConstructorDecl(hasAnyConstructorInitializer(
    forField(hasName("foo_"))))))
  matches Foo
with forField matching foo_
```

Matcher<CXXCtorInitializer> withInitializer Matcher<Expr> InnerMatcher

```
Matches the initializer expression of a constructor initializer.

Given
  struct Foo {
    Foo() : foo_(1) { }
    int foo_;
  };
cxxRecordDecl(has(cxxConstructorDecl(hasAnyConstructorInitializer(
    withInitializer(integerLiteral(equals(1)))))))
  matches Foo
with withInitializer matching (1)
```

Matcher<CXXForRangeStmt> hasBody Matcher<Stmt> InnerMatcher

```
Matches a 'for', 'while', 'do while' statement or a function
definition that has a given body.

Given
  for (;;) {}
hasBody(compoundStmt())
  matches 'for (;;) {}'
with compoundStmt()
  matching '{}'
```

Matcher<CXXForRangeStmt> hasLoopVariable Matcher<VarDecl> InnerMatcher

```
Matches the initialization statement of a for loop.

Example:
    forStmt(hasLoopVariable(anything()))
matches 'int x' in
    for (int x : a) { }
```

Matcher<CXXForRangeStmt> hasRangeInit Matcher<Expr> InnerMatcher

```
Matches the range initialization statement of a for loop.

Example:
    forStmt(hasRangeInit(anything()))
```

```
matches 'a' in
    for (int x : a) { }
```

Matcher<CXXMemberCallExpr>	onImplicitObjectArgument	Matcher<Expr> InnerMatcher

Matcher<CXXMemberCallExpr>	on	Matcher<Expr> InnerMatcher

```
Matches on the implicit object argument of a member call expression.

Example matches y.x()
  (matcher = cxxMemberCallExpr(on(hasType(cxxRecordDecl(hasName("Y"))))))
  class Y { public: void x(); };
  void z() { Y y; y.x(); }",

FIXME: Overload to allow directly matching types?
```

Matcher<CXXMemberCallExpr>	thisPointerType	Matcher<Decl> InnerMatcher

```
Overloaded to match the type's declaration.
```

Matcher<CXXMemberCallExpr>	thisPointerType	Matcher<QualType> InnerMatcher

```
Matches if the expression's type either matches the specified
matcher, or is a pointer to a type that matches the InnerMatcher.
```

Matcher<CXXMethodDecl>	forEachOverridden	Matcher<CXXMethodDecl> InnerMatcher

```
Matches each method overriden by the given method. This matcher may
produce multiple matches.

Given
  class A { virtual void f(); };
  class B : public A { void f(); };
  class C : public B { void f(); };
cxxMethodDecl(ofClass(hasName("C")),
              forEachOverridden(cxxMethodDecl().bind("b"))).bind("d")
  matches once, with "b" binding "A::f" and "d" binding "C::f" (Note
  that B::f is not overridden by C::f).

The check can produce multiple matches in case of multiple inheritance, e.g.
  class A1 { virtual void f(); };
  class A2 { virtual void f(); };
  class C : public A1, public A2 { void f(); };
cxxMethodDecl(ofClass(hasName("C")),
              forEachOverridden(cxxMethodDecl().bind("b"))).bind("d")
  matches twice, once with "b" binding "A1::f" and "d" binding "C::f", and
  once with "b" binding "A2::f" and "d" binding "C::f".
```

Matcher<CXXMethodDecl>	ofClass	Matcher<CXXRecordDecl> InnerMatcher

```
Matches the class declaration that the given method declaration
belongs to.

FIXME: Generalize this for other kinds of declarations.
FIXME: What other kind of declarations would we need to generalize
this to?

Example matches A() in the last line
    (matcher = cxxConstructExpr(hasDeclaration(cxxMethodDecl(
        ofClass(hasName("A"))))))
  class A {
   public:
    A();
  };
  A a = A();
```

Matcher<CXXNewExpr>	hasDeclaration	Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<CXXRecordDecl>	hasMethod	Matcher<CXXMethodDecl> InnerMatcher

```
Matches the first method of a class or struct that satisfies InnerMatcher.

Given:
  class A { void func(); };
  class B { void member(); };

cxxRecordDecl(hasMethod(hasName("func"))) matches the declaration of
A but not B.
```

Matcher<CXXRecordDecl>	isDerivedFrom	Matcher<NamedDecl> Base

```
Matches C++ classes that are directly or indirectly derived from
a class matching Base.

Note that a class is not considered to be derived from itself.

Example matches Y, Z, C (Base == hasName("X"))
  class X;
  class Y : public X {};  directly derived
  class Z : public Y {};  indirectly derived
  typedef X A;
  typedef A B;
  class C : public B {};  derived from a typedef of X

In the following example, Bar matches isDerivedFrom(hasName("X")):
  class Foo;
  typedef Foo X;
  class Bar : public Foo {};  derived from a type that X is a typedef of
```

Matcher<CXXRecordDecl>	isSameOrDerivedFrom	Matcher<NamedDecl> Base

```
Similar to isDerivedFrom(), but also matches classes that directly
match Base.
```

Matcher<CallExpr>	callee	Matcher<Decl> InnerMatcher

```
Matches if the call expression's callee's declaration matches the
given matcher.

Example matches y.x() (matcher = callExpr(callee(
                                   cxxMethodDecl(hasName("x")))))
  class Y { public: void x(); };
  void z() { Y y; y.x(); }
```

Matcher<CallExpr>	callee	Matcher<Stmt> InnerMatcher

```
Matches if the call expression's callee expression matches.

Given
  class Y { void x() { this->x(); x(); Y y; y.x(); } };
  void f() { f(); }
callExpr(callee(expr()))
  matches this->x(), x(), y.x(), f()
with callee(...)
  matching this->x, x, y.x, f respectively

Note: Callee cannot take the more general internal::Matcher<Expr>
because this introduces ambiguous overloads with calls to Callee taking a
internal::Matcher<Decl>, as the matcher hierarchy is purely
implemented in terms of implicit casts.
```

Matcher<CallExpr>	forEachArgumentWithParam	Matcher<Expr> ArgMatcher, Matcher<ParmVarDecl> ParamMatcher

```
Matches all arguments and their respective ParmVarDecl.

Given
  void f(int i);
  int y;
  f(y);
callExpr(
  forEachArgumentWithParam(
    declRefExpr(to(varDecl(hasName("y")))),
    parmVarDecl(hasType(isInteger()))
))
  matches f(y);
with declRefExpr(...)
  matching int y
and parmVarDecl(...)
  matching int i
```

Matcher<CallExpr>	hasAnyArgument	Matcher<Expr> InnerMatcher

```
Matches any argument of a call expression or a constructor call
expression.

Given
  void x(int, int, int) { int y; x(1, y, 42); }
callExpr(hasAnyArgument(declRefExpr()))
  matches x(1, y, 42)
with hasAnyArgument(...)
  matching y
```

Matcher<CallExpr>	hasArgument	unsigned N, Matcher<Expr> InnerMatcher

```
Matches the n'th argument of a call expression or a constructor
call expression.

Example matches y in x(y)
    (matcher = callExpr(hasArgument(0, declRefExpr())))
  void x(int) { int y; x(y); }
```

Matcher<CallExpr>	hasDeclaration	Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
```

```
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<CaseStmt>	hasCaseConstant	Matcher<Expr> InnerMatcher

```
If the given case statement does not use the GNU case range
extension, matches the constant given in the statement.

Given
  switch (1) { case 1: case 1+1: case 3 ... 4: ; }
caseStmt(hasCaseConstant(integerLiteral()))
  matches "case 1:"
```

Matcher<CastExpr>	hasSourceExpression	Matcher<Expr> InnerMatcher

Matcher<ClassTemplateSpecializationDecl>	hasAnyTemplateArgument	Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl that have at least one TemplateArgument matching the given
InnerMatcher.

Given
  template<typename T> class A {};
  template<> class A<double> {};
  A<int> a;

  template<typename T> f() {};
  void func() { f<int>(); };

classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToType(asString("int"))))
  matches the specialization A<int>

functionDecl(hasAnyTemplateArgument(refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<ClassTemplateSpecializationDecl>	hasTemplateArgument	unsigned N, Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl where the n'th TemplateArgument matches the given InnerMatcher.

Given
  template<typename T, typename U> class A {};
  A<bool, int> b;
  A<int, bool> c;

  template<typename T> f() {};
  void func() { f<int>(); };
classTemplateSpecializationDecl(hasTemplateArgument(
    1, refersToType(asString("int"))))
  matches the specialization A<bool, int>

functionDecl(hasTemplateArgument(0, refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<ComplexTypeLoc>	hasElementTypeLoc	Matcher<TypeLoc>

```
Matches arrays and C99 complex types that have a specific element
type.

Given
  struct A {};
  A a[7];
  int b[7];
arrayType(hasElementType(builtinType()))
  matches "int b[7]"

Usable as: Matcher<ArrayType>, Matcher<ComplexType>
```

Matcher<ComplexType>	hasElementType	Matcher<Type>

```
Matches arrays and C99 complex types that have a specific element
type.

Given
  struct A {};
  A a[7];
  int b[7];
arrayType(hasElementType(builtinType()))
  matches "int b[7]"

Usable as: Matcher<ArrayType>, Matcher<ComplexType>
```

Matcher<CompoundStmt>	hasAnySubstatement	Matcher<Stmt> InnerMatcher

```
Matches compound statements where at least one substatement matches
a given matcher. Also matches StmtExprs that have CompoundStmt as children.

Given
```

```
  { {}; 1+2; }
hasAnySubstatement(compoundStmt())
  matches '{ {}; 1+2; }'
with compoundStmt()
  matching '{}'
```

Matcher<DecayedType>	hasDecayedType	Matcher<QualType> InnerType

```
Matches the decayed type, whos decayed type matches InnerMatcher
```

Matcher<DeclRefExpr>	hasDeclaration	Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<DeclRefExpr>	throughUsingDecl	Matcher<UsingShadowDecl> InnerMatcher

```
Matches a DeclRefExpr that refers to a declaration through a
specific using shadow declaration.

Given
  namespace a { void f() {} }
  using a::f;
  void g() {
    f();     Matches this ..
    a::f();  .. but not this.
  }
declRefExpr(throughUsingDecl(anything()))
  matches f()
```

Matcher<DeclRefExpr>	to	Matcher<Decl> InnerMatcher

```
Matches a DeclRefExpr that refers to a declaration that matches the
specified matcher.

Example matches x in if(x)
    (matcher = declRefExpr(to(varDecl(hasName("x")))))
  bool x;
  if (x) {}
```

Matcher<DeclStmt>	containsDeclaration	unsigned N, Matcher<Decl> InnerMatcher

```
Matches the n'th declaration of a declaration statement.

Note that this does not work for global declarations because the AST
breaks up multiple-declaration DeclStmt's into multiple single-declaration
DeclStmt's.
Example: Given non-global declarations
  int a, b = 0;
  int c;
  int d = 2, e;
declStmt(containsDeclaration(
      0, varDecl(hasInitializer(anything()))))
  matches only 'int d = 2, e;', and
declStmt(containsDeclaration(1, varDecl()))
  matches 'int a, b = 0' as well as 'int d = 2, e;'
  but 'int c;' is not matched.
```

Matcher<DeclStmt>	hasSingleDecl	Matcher<Decl> InnerMatcher

```
Matches the Decl of a DeclStmt which has a single declaration.

Given
  int a, b;
  int c;
declStmt(hasSingleDecl(anything()))
  matches 'int c;' but not 'int a, b;'.
```

Matcher<DeclaratorDecl>	hasTypeLoc	Matcher<TypeLoc> Inner

```
Matches if the type location of the declarator decl's type matches
the inner matcher.

Given
  int x;
declaratorDecl(hasTypeLoc(loc(asString("int"))))
  matches int x
```

Matcher<Decl>	hasDeclContext	Matcher<Decl> InnerMatcher

```
Matches declarations whose declaration context, interpreted as a
Decl, matches InnerMatcher.

Given
  namespace N {
```

```
    namespace M {
      class D {};
    }
  }

cxxRcordDecl(hasDeclContext(namedDecl(hasName("M")))) matches the
declaration of class D.
```

Matcher<DoStmt> hasBody Matcher<Stmt> InnerMatcher

```
Matches a 'for', 'while', 'do while' statement or a function
definition that has a given body.

Given
  for (;;) {}
hasBody(compoundStmt())
  matches 'for (;;) {}'
with compoundStmt()
  matching '{}'
```

Matcher<DoStmt> hasCondition Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<ElaboratedType> hasQualifier Matcher<NestedNameSpecifier> InnerMatcher

```
Matches ElaboratedTypes whose qualifier, a NestedNameSpecifier,
matches InnerMatcher if the qualifier exists.

Given
  namespace N {
    namespace M {
      class D {};
    }
  }
  N::M::D d;

elaboratedType(hasQualifier(hasPrefix(specifiesNamespace(hasName("N"))))
matches the type of the variable declaration of d.
```

Matcher<ElaboratedType> namesType Matcher<QualType> InnerMatcher

```
Matches ElaboratedTypes whose named type matches InnerMatcher.

Given
  namespace N {
    namespace M {
      class D {};
    }
  }
  N::M::D d;

elaboratedType(namesType(recordType(
hasDeclaration(namedDecl(hasName("D")))))) matches the type of the variable
declaration of d.
```

Matcher<EnumType> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<ExplicitCastExpr> hasDestinationType Matcher<QualType> InnerMatcher

```
Matches casts whose destination type matches a given matcher.

(Note: Clang's AST refers to other conversions as "casts" too, and calls
actual casts "explicit" casts.)
```

Matcher<Expr> hasType Matcher<Decl> InnerMatcher

```
Overloaded to match the declaration of the expression's or value
declaration's type.

In case of a value declaration (for example a variable declaration),
this resolves one layer of indirection. For example, in the value
declaration "X x;", cxxRecordDecl(hasName("X")) matches the declaration of
X, while varDecl(hasType(cxxRecordDecl(hasName("X")))) matches the
declaration of x.

Example matches x (matcher = expr(hasType(cxxRecordDecl(hasName("X")))))
            and z (matcher = varDecl(hasType(cxxRecordDecl(hasName("X")))))
 class X {};
```

```
 void y(X &x) { x; X z; }
```

Usable as: Matcher<Expr>, Matcher<ValueDecl>

Matcher<Expr>	hasType	Matcher<QualType> InnerMatcher

```
Matches if the expression's or declaration's type matches a type
matcher.

Example matches x (matcher = expr(hasType(cxxRecordDecl(hasName("X")))))
            and z (matcher = varDecl(hasType(cxxRecordDecl(hasName("X")))))
            and U (matcher = typedefDecl(hasType(asString("int")))
 class X {};
 void y(X &x) { x; X z; }
 typedef int U;
```

Matcher<Expr>	ignoringImpCasts	Matcher<Expr> InnerMatcher

```
Matches expressions that match InnerMatcher after any implicit casts
are stripped off.

Parentheses and explicit casts are not discarded.
Given
  int arr[5];
  int a = 0;
  char b = 0;
  const int c = a;
  int *d = arr;
  long e = (long) 0l;
The matchers
   varDecl(hasInitializer(ignoringImpCasts(integerLiteral())))
   varDecl(hasInitializer(ignoringImpCasts(declRefExpr())))
would match the declarations for a, b, c, and d, but not e.
While
   varDecl(hasInitializer(integerLiteral()))
   varDecl(hasInitializer(declRefExpr()))
only match the declarations for b, c, and d.
```

Matcher<Expr>	ignoringImplicit	ast_matchers::Matcher<Expr> InnerMatcher

```
Matches expressions that match InnerMatcher after any implicit AST
nodes are stripped off.

Parentheses and explicit casts are not discarded.
Given
  class C {};
  C a = C();
  C b;
  C c = b;
The matchers
   varDecl(hasInitializer(ignoringImplicit(cxxConstructExpr())))
would match the declarations for a, b, and c.
While
   varDecl(hasInitializer(cxxConstructExpr()))
only match the declarations for b and c.
```

Matcher<Expr>	ignoringParenCasts	Matcher<Expr> InnerMatcher

```
Matches expressions that match InnerMatcher after parentheses and
casts are stripped off.

Implicit and non-C Style casts are also discarded.
Given
  int a = 0;
  char b = (0);
  void* c = reinterpret_cast<char*>(0);
  char d = char(0);
The matcher
   varDecl(hasInitializer(ignoringParenCasts(integerLiteral())))
would match the declarations for a, b, c, and d.
while
   varDecl(hasInitializer(integerLiteral()))
only match the declaration for a.
```

Matcher<Expr>	ignoringParenImpCasts	Matcher<Expr> InnerMatcher

```
Matches expressions that match InnerMatcher after implicit casts and
parentheses are stripped off.

Explicit casts are not discarded.
Given
  int arr[5];
  int a = 0;
  char b = (0);
  const int c = a;
  int *d = (arr);
  long e = ((long) 0l);
The matchers
   varDecl(hasInitializer(ignoringParenImpCasts(integerLiteral())))
   varDecl(hasInitializer(ignoringParenImpCasts(declRefExpr())))
would match the declarations for a, b, c, and d, but not e.
while
   varDecl(hasInitializer(integerLiteral()))
   varDecl(hasInitializer(declRefExpr()))
would only match the declaration for a.
```

Matcher<FieldDecl>	hasInClassInitializer	Matcher<Expr> InnerMatcher

```
Matches non-static data members that have an in-class initializer.

Given
  class C {
    int a = 2;
    int b = 3;
    int c;
```

```
  };
fieldDecl(hasInClassInitializer(integerLiteral(equals(2))))
  matches 'int a;' but not 'int b;'.
fieldDecl(hasInClassInitializer(anything()))
  matches 'int a;' and 'int b;' but not 'int c;'.
```

Matcher<ForStmt>	hasBody	Matcher<Stmt> InnerMatcher

```
Matches a 'for', 'while', 'do while' statement or a function
definition that has a given body.

Given
  for (;;) {}
hasBody(compoundStmt())
  matches 'for (;;) {}'
with compoundStmt()
  matching '{}'
```

Matcher<ForStmt>	hasCondition	Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<ForStmt>	hasIncrement	Matcher<Stmt> InnerMatcher

```
Matches the increment statement of a for loop.

Example:
    forStmt(hasIncrement(unaryOperator(hasOperatorName("++"))))
matches '++x' in
    for (x; x < N; ++x) { }
```

Matcher<ForStmt>	hasLoopInit	Matcher<Stmt> InnerMatcher

```
Matches the initialization statement of a for loop.

Example:
    forStmt(hasLoopInit(declStmt()))
matches 'int x = 0' in
    for (int x = 0; x < N; ++x) { }
```

Matcher<FunctionDecl>	hasAnyParameter	Matcher<ParmVarDecl> InnerMatcher

```
Matches any parameter of a function declaration.

Does not match the 'this' parameter of a method.

Given
  class X { void f(int x, int y, int z) {} };
cxxMethodDecl(hasAnyParameter(hasName("y")))
  matches f(int x, int y, int z) {}
with hasAnyParameter(...)
  matching int y
```

Matcher<FunctionDecl>	hasAnyTemplateArgument	Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl that have at least one TemplateArgument matching the given
InnerMatcher.

Given
  template<typename T> class A {};
  template<> class A<double> {};
  A<int> a;

  template<typename T> f() {};
  void func() { f<int>(); };

classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToType(asString("int"))))
  matches the specialization A<int>

functionDecl(hasAnyTemplateArgument(refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<FunctionDecl>	hasBody	Matcher<Stmt> InnerMatcher

```
Matches a 'for', 'while', 'do while' statement or a function
definition that has a given body.

Given
  for (;;) {}
hasBody(compoundStmt())
  matches 'for (;;) {}'
with compoundStmt()
  matching '{}'
```

Matcher<FunctionDecl>	hasParameter	unsigned N, Matcher<ParmVarDecl> InnerMatcher

```
Matches the n'th parameter of a function declaration.

Given
  class X { void f(int x) {} };
cxxMethodDecl(hasParameter(0, hasType(varDecl())))
  matches f(int x) {}
with hasParameter(...)
  matching int x
```

Matcher<FunctionDecl>	hasTemplateArgument	unsigned N,

Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl where the n'th TemplateArgument matches the given InnerMatcher.

Given
  template<typename T, typename U> class A {};
  A<bool, int> b;
  A<int, bool> c;

  template<typename T> f() {};
  void func() { f<int>(); };
classTemplateSpecializationDecl(hasTemplateArgument(
    1, refersToType(asString("int"))))
  matches the specialization A<bool, int>

functionDecl(hasTemplateArgument(0, refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<FunctionDecl> returns Matcher<QualType> InnerMatcher

```
Matches the return type of a function declaration.

Given:
  class X { int f() { return 1; } };
cxxMethodDecl(returns(asString("int")))
  matches int f() { return 1; }
```

Matcher<IfStmt> hasCondition Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<IfStmt> hasConditionVariableStatement Matcher<DeclStmt> InnerMatcher

```
Matches the condition variable statement in an if statement.

Given
  if (A* a = GetAPointer()) {}
hasConditionVariableStatement(...)
  matches 'A* a = GetAPointer()'.
```

Matcher<IfStmt> hasElse Matcher<Stmt> InnerMatcher

```
Matches the else-statement of an if statement.

Examples matches the if statement
  (matcher = ifStmt(hasElse(cxxBoolLiteral(equals(true)))))
  if (false) false; else true;
```

Matcher<IfStmt> hasThen Matcher<Stmt> InnerMatcher

```
Matches the then-statement of an if statement.

Examples matches the if statement
  (matcher = ifStmt(hasThen(cxxBoolLiteral(equals(true)))))
  if (false) true; else false;
```

Matcher<ImplicitCastExpr> hasImplicitDestinationType Matcher<QualType> InnerMatcher

```
Matches implicit casts whose destination type matches a given
matcher.

FIXME: Unit test this matcher
```

Matcher<InitListExpr> hasSyntacticForm Matcher<Expr> InnerMatcher

```
Matches the syntactic form of init list expressions
(if expression have it).
```

Matcher<InjectedClassNameType> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<LabelStmt> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
```

```
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<MemberExpr> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<MemberExpr> hasObjectExpression Matcher<Expr> InnerMatcher

```
Matches a member expression where the object expression is
matched by a given matcher.

Given
  struct X { int m; };
  void f(X x) { x.m; m; }
memberExpr(hasObjectExpression(hasType(cxxRecordDecl(hasName("X")))))
  matches "x.m" and "m"
with hasObjectExpression(...)
  matching "x" and the implicit object expression of "m" which has type X*.
```

Matcher<MemberExpr> member Matcher<ValueDecl> InnerMatcher

```
Matches a member expression where the member is matched by a
given matcher.

Given
  struct { int first, second; } first, second;
  int i(second.first);
  int j(first.second);
memberExpr(member(hasName("first")))
  matches second.first
  but not first.second (because the member name there is "second").
```

Matcher<MemberPointerTypeLoc> pointeeLoc Matcher<TypeLoc>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<MemberPointerType> pointee Matcher<Type>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<NamedDecl> hasUnderlyingDecl Matcher<NamedDecl> InnerMatcher

```
Matches a NamedDecl whose underlying declaration matches the given
matcher.

Given
  namespace N { template<class T> void f(T t); }
  template <class T> void g() { using N::f; f(T()); }
unresolvedLookupExpr(hasAnyDeclaration(
    namedDecl(hasUnderlyingDecl(hasName("::N::f")))))
  matches the use of f in g() .
```

Matcher<NestedNameSpecifierLoc>	hasPrefix	Matcher<NestedNameSpecifierLoc> InnerMatcher

```
Matches on the prefix of a NestedNameSpecifierLoc.

Given
  struct A { struct B { struct C {}; }; };
  A::B::C c;
nestedNameSpecifierLoc(hasPrefix(loc(specifiesType(asString("struct A")))))
  matches "A::"
```

Matcher<NestedNameSpecifierLoc>	specifiesTypeLoc	Matcher<TypeLoc> InnerMatcher

```
Matches nested name specifier locs that specify a type matching the
given TypeLoc.

Given
  struct A { struct B { struct C {}; }; };
  A::B::C c;
nestedNameSpecifierLoc(specifiesTypeLoc(loc(type(
  hasDeclaration(cxxRecordDecl(hasName("A")))))))
  matches "A::"
```

Matcher<NestedNameSpecifier>	hasPrefix	Matcher<NestedNameSpecifier> InnerMatcher

```
Matches on the prefix of a NestedNameSpecifier.

Given
  struct A { struct B { struct C {}; }; };
  A::B::C c;
nestedNameSpecifier(hasPrefix(specifiesType(asString("struct A")))) and
  matches "A::"
```

Matcher<NestedNameSpecifier>	specifiesNamespace	Matcher<NamespaceDecl> InnerMatcher

```
Matches nested name specifiers that specify a namespace matching the
given namespace matcher.

Given
  namespace ns { struct A {}; }
  ns::A a;
nestedNameSpecifier(specifiesNamespace(hasName("ns")))
  matches "ns::"
```

Matcher<NestedNameSpecifier>	specifiesType	Matcher<QualType> InnerMatcher

```
Matches nested name specifiers that specify a type matching the
given QualType matcher without qualifiers.

Given
  struct A { struct B { struct C {}; }; };
  A::B::C c;
nestedNameSpecifier(specifiesType(
  hasDeclaration(cxxRecordDecl(hasName("A")))
))
  matches "A::"
```

Matcher<ObjCMessageExpr>	hasArgument	unsigned N, Matcher<Expr> InnerMatcher

```
Matches the n'th argument of a call expression or a constructor
call expression.

Example matches y in x(y)
    (matcher = callExpr(hasArgument(0, declRefExpr())))
  void x(int) { int y; x(y); }
```

Matcher<ObjCMessageExpr>	hasReceiverType	Matcher<QualType> InnerMatcher

```
Matches on the receiver of an ObjectiveC Message expression.

Example
matcher = objCMessageExpr(hasReceiverType(asString("UIWebView *")));
matches the [webView ...] message invocation.
  NSString *webViewJavaScript = ...
  UIWebView *webView = ...
  [webView stringByEvaluatingJavaScriptFromString:webViewJavascript];
```

Matcher<OpaqueValueExpr>	hasSourceExpression	Matcher<Expr> InnerMatcher

Matcher<OverloadExpr>	hasAnyDeclaration	Matcher<Decl> InnerMatcher

```
Matches an OverloadExpr if any of the declarations in the set of
overloads matches the given matcher.

Given
  template <typename T> void foo(T);
  template <typename T> void bar(T);
  template <typename T> void baz(T t) {
    foo(t);
    bar(t);
  }
unresolvedLookupExpr(hasAnyDeclaration(
    functionTemplateDecl(hasName("foo"))))
  matches foo in foo(t); but not bar in bar(t);
```

Matcher<ParenType>	innerType	Matcher<Type>

```
Matches ParenType nodes where the inner type is a specific type.
```

```
Given
  int (*ptr_to_array)[4];
  int (*ptr_to_func)(int);

varDecl(hasType(pointsTo(parenType(innerType(functionType()))))) matches
ptr_to_func but not ptr_to_array.

Usable as: Matcher<ParenType>
```

Matcher<PointerTypeLoc> pointeeLoc Matcher<TypeLoc>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<PointerType> pointee Matcher<Type>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<QualType> hasCanonicalType Matcher<QualType> InnerMatcher

```
Matches QualTypes whose canonical type matches InnerMatcher.

Given:
  typedef int &int_ref;
  int a;
  int_ref b = a;

varDecl(hasType(qualType(referenceType()))) will not match the
declaration of b but varDecl(hasType(qualType(hasCanonicalType(referenceType())))) does.
```

Matcher<QualType> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<QualType> ignoringParens Matcher<QualType> InnerMatcher

```
Matches types that match InnerMatcher after any parens are stripped.

Given
  void (*fp)(void);
The matcher
  varDecl(hasType(pointerType(pointee(ignoringParens(functionType())))))
would match the declaration for fp.
```

Matcher<QualType> pointsTo Matcher<Decl> InnerMatcher

```
Overloaded to match the pointee type's declaration.
```

Matcher<QualType> pointsTo Matcher<QualType> InnerMatcher

```
Matches if the matched type is a pointer type and the pointee type
matches the specified matcher.

Example matches y->x()
  (matcher = cxxMemberCallExpr(on(hasType(pointsTo
     cxxRecordDecl(hasName("Y"))))))
  class Y { public: void x(); };
  void z() { Y *y; y->x(); }
```

Matcher<QualType> references Matcher<Decl> InnerMatcher

```
Overloaded to match the referenced type's declaration.
```

Matcher<QualType> references Matcher<QualType> InnerMatcher

```
Matches if the matched type is a reference type and the referenced
type matches the specified matcher.

Example matches X &x and const X &y
    (matcher = varDecl(hasType(references(cxxRecordDecl(hasName("X"))))))
  class X {
    void a(X b) {
      X &x = b;
      const X &y = b;
    }
  };
```

Matcher<RecordType> — hasDeclaration — Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<ReferenceTypeLoc> — pointeeLoc — Matcher<TypeLoc>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<ReferenceType> — pointee — Matcher<Type>

```
Narrows PointerType (and similar) matchers to those where the
pointee matches a given matcher.

Given
  int *a;
  int const *b;
  float const *f;
pointerType(pointee(isConstQualified(), isInteger()))
  matches "int const *b"

Usable as: Matcher<BlockPointerType>, Matcher<MemberPointerType>,
  Matcher<PointerType>, Matcher<ReferenceType>
```

Matcher<ReturnStmt> — hasReturnValue — Matcher<Expr> InnerMatcher

```
Matches the return value expression of a return statement

Given
  return a + b;
hasReturnValue(binaryOperator())
  matches 'return a + b'
with binaryOperator()
  matching 'a + b'
```

Matcher<StmtExpr> — hasAnySubstatement — Matcher<Stmt> InnerMatcher

```
Matches compound statements where at least one substatement matches
a given matcher. Also matches StmtExprs that have CompoundStmt as children.

Given
  { {}; 1+2; }
hasAnySubstatement(compoundStmt())
  matches '{ {}; 1+2; }'
with compoundStmt()
  matching '{}'
```

Matcher<Stmt> — alignOfExpr — Matcher<UnaryExprOrTypeTraitExpr> InnerMatcher

```
Same as unaryExprOrTypeTraitExpr, but only matching
alignof.
```

Matcher<Stmt> — forFunction — Matcher<FunctionDecl> InnerMatcher

```
Matches declaration of the function the statement belongs to

Given:
F& operator=(const F& o) {
  std::copy_if(o.begin(), o.end(), begin(), [](V v) { return v > 0; });
  return *this;
}
returnStmt(forFunction(hasName("operator=")))
```

```
  matches 'return *this'
  but does match 'return > 0'
```

Matcher<Stmt>	sizeOfExpr	Matcher<UnaryExprOrTypeTraitExpr> InnerMatcher

```
Same as unaryExprOrTypeTraitExpr, but only matching
sizeof.
```

Matcher<SubstTemplateTypeParmType>	hasReplacementType	Matcher<Type>

```
Matches template type parameter substitutions that have a replacement
type that matches the provided matcher.

Given
  template <typename T>
  double F(T t);
  int i;
  double j = F(i);

substTemplateTypeParmType(hasReplacementType(type())) matches int
```

Matcher<SwitchStmt>	forEachSwitchCase	Matcher<SwitchCase> InnerMatcher

```
Matches each case or default statement belonging to the given switch
statement. This matcher may produce multiple matches.

Given
  switch (1) { case 1: case 2: default: switch (2) { case 3: case 4: ; } }
switchStmt(forEachSwitchCase(caseStmt().bind("c"))).bind("s")
  matches four times, with "c" binding each of "case 1:", "case 2:",
"case 3:" and "case 4:", and "s" respectively binding "switch (1)",
"switch (1)", "switch (2)" and "switch (2)".
```

Matcher<SwitchStmt>	hasCondition	Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<TagType>	hasDeclaration	Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<TemplateArgument>	isExpr	Matcher<Expr> InnerMatcher

```
Matches a sugar TemplateArgument that refers to a certain expression.

Given
  template<typename T> struct A {};
  struct B { B* next; };
  A<&B::next> a;
templateSpecializationType(hasAnyTemplateArgument(
  isExpr(hasDescendant(declRefExpr(to(fieldDecl(hasName("next")))))))
  matches the specialization A<&B::next> with fieldDecl(...) matching
    B::next
```

Matcher<TemplateArgument>	refersToDeclaration	Matcher<Decl> InnerMatcher

```
Matches a canonical TemplateArgument that refers to a certain
declaration.

Given
  template<typename T> struct A {};
  struct B { B* next; };
  A<&B::next> a;
classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToDeclaration(fieldDecl(hasName("next"))))
  matches the specialization A<&B::next> with fieldDecl(...) matching
    B::next
```

Matcher<TemplateArgument>	refersToIntegralType	Matcher<QualType> InnerMatcher

```
Matches a TemplateArgument that referes to an integral type.

Given
  template<int T> struct A {};
  C<42> c;
classTemplateSpecializationDecl(
  hasAnyTemplateArgument(refersToIntegralType(asString("int"))))
  matches the implicit instantiation of C in C<42>.
```

Matcher<TemplateArgument>	refersToTemplate	Matcher<TemplateName>

InnerMatcher

```
Matches a TemplateArgument that refers to a certain template.

Given
  template<template <typename> class S> class X {};
  template<typename T> class Y {};"
  X<Y> xi;
classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToTemplate(templateName())))
  matches the specialization X<Y>
```

Matcher<TemplateArgument> refersToType Matcher<QualType> InnerMatcher

```
Matches a TemplateArgument that refers to a certain type.

Given
  struct X {};
  template<typename T> struct A {};
  A<X> a;
classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToType(class(hasName("X")))))
  matches the specialization A<X>
```

Matcher<TemplateSpecializationType> hasAnyTemplateArgument Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl that have at least one TemplateArgument matching the given
InnerMatcher.

Given
  template<typename T> class A {};
  template<> class A<double> {};
  A<int> a;

  template<typename T> f() {};
  void func() { f<int>(); };

classTemplateSpecializationDecl(hasAnyTemplateArgument(
    refersToType(asString("int"))))
  matches the specialization A<int>

functionDecl(hasAnyTemplateArgument(refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<TemplateSpecializationType> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<TemplateSpecializationType> hasTemplateArgument unsigned N, Matcher<TemplateArgument> InnerMatcher

```
Matches classTemplateSpecializations, templateSpecializationType and
functionDecl where the n'th TemplateArgument matches the given InnerMatcher.

Given
  template<typename T, typename U> class A {};
  A<bool, int> b;
  A<int, bool> c;

  template<typename T> f() {};
  void func() { f<int>(); };
classTemplateSpecializationDecl(hasTemplateArgument(
    1, refersToType(asString("int"))))
  matches the specialization A<bool, int>

functionDecl(hasTemplateArgument(0, refersToType(asString("int"))))
  matches the specialization f<int>
```

Matcher<TemplateTypeParmType> hasDeclaration Matcher<Decl> InnerMatcher

```
Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.
```

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>

Matcher<T> findAll Matcher<T> Matcher

Matches if the node or any descendant matches.

Generates results for each match.

For example, in:
  class A { class B {}; class C {}; };
The matcher:
  cxxRecordDecl(hasName("::A"),
                findAll(cxxRecordDecl(isDefinition()).bind("m")))
will generate results for A, B and C.

Usable as: Any Matcher

Matcher<TypedefNameDecl> hasType Matcher<QualType> InnerMatcher

Matches if the expression's or declaration's type matches a type
matcher.

Example matches x (matcher = expr(hasType(cxxRecordDecl(hasName("X")))))
            and z (matcher = varDecl(hasType(cxxRecordDecl(hasName("X")))))
            and U (matcher = typedefDecl(hasType(asString("int"))))
 class X {};
 void y(X &x) { x; X z; }
 typedef int U;

Matcher<TypedefType> hasDeclaration Matcher<Decl> InnerMatcher

Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>

Matcher<Type> hasUnqualifiedDesugaredType Matcher<Type> InnerMatcher

Matches if the matched type matches the unqualified desugared
type of the matched node.

For example, in:
  class A {};
  using B = A;
The matcher type(hasUniqualifeidDesugaredType(recordType())) matches
both B and A.

Matcher<UnaryExprOrTypeTraitExpr> hasArgumentOfType Matcher<QualType> InnerMatcher

Matches unary expressions that have a specific type of argument.

Given
  int a, c; float b; int s = sizeof(a) + sizeof(b) + alignof(c);
unaryExprOrTypeTraitExpr(hasArgumentOfType(asString("int"))
  matches sizeof(a) and alignof(c)

Matcher<UnaryOperator> hasUnaryOperand Matcher<Expr> InnerMatcher

Matches if the operand of a unary operator matches.

Example matches true (matcher = hasUnaryOperand(
                                cxxBoolLiteral(equals(true))))
  !true

Matcher<UnresolvedUsingType> hasDeclaration Matcher<Decl> InnerMatcher

Matches a node if the declaration associated with that node
matches the given matcher.

The associated declaration is:
- for type nodes, the declaration of the underlying type
- for CallExpr, the declaration of the callee
- for MemberExpr, the declaration of the referenced member
- for CXXConstructExpr, the declaration of the constructor
- for CXXNewExpr, the declaration of the operator new

Also usable as Matcher<T> for any T supporting the getDecl() member
function. e.g. various subtypes of clang::Type and various expressions.

Usable as: Matcher<AddrLabelExpr>, Matcher<CallExpr>,
  Matcher<CXXConstructExpr>, Matcher<CXXNewExpr>, Matcher<DeclRefExpr>,
  Matcher<EnumType>, Matcher<InjectedClassNameType>, Matcher<LabelStmt>,
  Matcher<MemberExpr>, Matcher<QualType>, Matcher<RecordType>,
  Matcher<TagType>, Matcher<TemplateSpecializationType>,

```
  Matcher<TemplateTypeParmType>, Matcher<TypedefType>,
  Matcher<UnresolvedUsingType>
```

Matcher<UsingDecl>	hasAnyUsingShadowDecl	Matcher<UsingShadowDecl> InnerMatcher

```
Matches any using shadow declaration.

Given
  namespace X { void b(); }
  using X::b;
usingDecl(hasAnyUsingShadowDecl(hasName("b")))
  matches using X::b
```

Matcher<UsingShadowDecl>	hasTargetDecl	Matcher<NamedDecl> InnerMatcher

```
Matches a using shadow declaration where the target declaration is
matched by the given matcher.

Given
  namespace X { int a; void b(); }
  using X::a;
  using X::b;
usingDecl(hasAnyUsingShadowDecl(hasTargetDecl(functionDecl())))
  matches using X::b but not using X::a
```

Matcher<ValueDecl>	hasType	Matcher<Decl> InnerMatcher

```
Overloaded to match the declaration of the expression's or value
declaration's type.

In case of a value declaration (for example a variable declaration),
this resolves one layer of indirection. For example, in the value
declaration "X x;", cxxRecordDecl(hasName("X")) matches the declaration of
X, while varDecl(hasType(cxxRecordDecl(hasName("X")))) matches the
declaration of x.

Example matches x (matcher = expr(hasType(cxxRecordDecl(hasName("X")))))
            and z (matcher = varDecl(hasType(cxxRecordDecl(hasName("X")))))
 class X {};
 void y(X &x) { x; X z; }

Usable as: Matcher<Expr>, Matcher<ValueDecl>
```

Matcher<ValueDecl>	hasType	Matcher<QualType> InnerMatcher

```
Matches if the expression's or declaration's type matches a type
matcher.

Example matches x (matcher = expr(hasType(cxxRecordDecl(hasName("X")))))
            and z (matcher = varDecl(hasType(cxxRecordDecl(hasName("X")))))
            and U (matcher = typedefDecl(hasType(asString("int"))))
 class X {};
 void y(X &x) { x; X z; }
 typedef int U;
```

Matcher<VarDecl>	hasInitializer	Matcher<Expr> InnerMatcher

```
Matches a variable declaration that has an initializer expression
that matches the given matcher.

Example matches x (matcher = varDecl(hasInitializer(callExpr())))
  bool y() { return true; }
  bool x = y();
```

Matcher<VariableArrayType>	hasSizeExpr	Matcher<Expr> InnerMatcher

```
Matches VariableArrayType nodes that have a specific size
expression.

Given
  void f(int b) {
    int a[b];
  }
variableArrayType(hasSizeExpr(ignoringImpCasts(declRefExpr(to(
  varDecl(hasName("b")))))))
  matches "int a[b]"
```

Matcher<WhileStmt>	hasBody	Matcher<Stmt> InnerMatcher

```
Matches a 'for', 'while', 'do while' statement or a function
definition that has a given body.

Given
  for (;;) {}
hasBody(compoundStmt())
  matches 'for (;;) {}'
with compoundStmt()
  matching '{}'
```

Matcher<WhileStmt>	hasCondition	Matcher<Expr> InnerMatcher

```
Matches the condition expression of an if statement, for loop,
switch statement or conditional operator.

Example matches true (matcher = hasCondition(cxxBoolLiteral(equals(true))))
  if (true) {}
```

Matcher<internal::BindableMatcher<NestedNameSpecifierLoc>>	loc	Matcher<NestedNameSpecifier> InnerMatcher

```
Matches NestedNameSpecifierLocs for which the given inner
NestedNameSpecifier-matcher matches.
```

Matcher<internal::BindableMatcher<TypeLoc>>	loc	Matcher<QualType> InnerMatcher

Matches TypeLocs for which the given inner QualType-matcher matches.