

IFC Binary Format

Version: 0.42

Gabriel Dos Reis
Microsoft

May 25, 2022

Abstract

This document defines the IFC binary format for persistent representation of the abstract semantics graph of a C++ translation unit, in particular for a compiled module interface. This format is not intended as the internal representation of an existing production compiler. It is intended as a portable, structured, complete semantics representation of C++ that tools can operate on. It is incorrect, incomplete, and a work in progress.

©2021 Microsoft.

This material is licensed under Creative Commons Attribution 4.0 International.

Contents

Contents	1
1 Introduction	3
2 IFC File	5
3 String table	13
4 Translation Units	15
5 Modules	17
6 Scopes	19
7 Heaps	21
8 Declarations	23
9 Types	51
10 Expressions	67
11 Statements	121
12 Names	127
13 Charts	131
14 Attributes	133
15 Syntax Tree	139
16 Source Location	181
17 Traits	183
18 Preprocessing Forms	187

19 Words**197****Bibliography****223**

Chapter 1

Introduction

Chapter 2

IFC File

2.1 Overview

Compiled module interfaces are persisted in an IFC format. The general representation is as follows

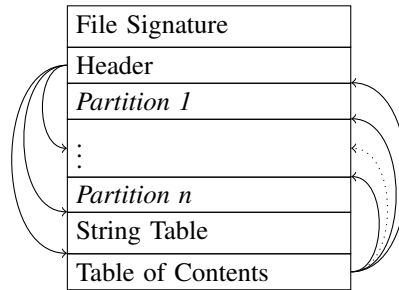


Figure 2.1: IFC file general structure

An IFC represents the *abstract semantics graph* that is the result of elaborating declarations in an input source file, e.g. a module interface file, or a header unit, or indeed any C++ source file that leads to a well-formed translation unit. Declarations and expressions are designated by *abstract references*. An abstract reference is essentially a typed pointer that refers to an index in a partition. The specific partition is given by the *tag* field of the abstract reference, and the index is given by the *index* field of the reference. All abstract references are multiple-byte values with 32-bit precision:

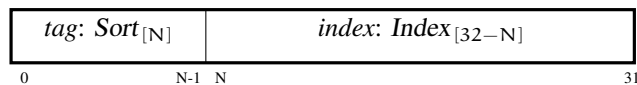


Figure 2.2: Abstract reference parameterized by the sort of designated entity.

The overall aim is to define the binary representations after the *Internal Program Representation* (IPR), a work done by Gabriel Dos Reis and Bjarne Stroustrup [1, 2] to define a more regular foundational semantics for C++, capable of capturing ISO C++ and practical dialects.

2.2 Multiple IFCs per file

The current specification only defines one IFC per containing file. However, the long term goal is to support multiple IFCs per containing file. Where there is a mention of "offset from the beginning of the file", it should be understood "offset from the beginning of the current IFC".

2.3 Type of IFC container

An IFC can be embedded in just any binary file. The VC++ compiler by default generates IFCs in binary files with the `.ifc` extension. However, they can also be embedded in archives (e.g. files with `.a` or `.lib` extensions), or in shared or dynamically linked libraries (e.g. files with `.so`, `.dll`, `.dylib` extensions.)

2.4 On-demand materialization

The IFC format is designed to support (and encourage) “on-demand” materialization of declarations. That is, when the compiler sees an import-declaration, it does *not* bring in all the declarations right away. Rather, the idea is that it only makes visible the set of (toplevel) names exported by the nominated module. An on-demand materialization strategy will only reconstruct declarations upon name lookup (in response to a name use in the importing translation unit) however referenced. The on-demand materialization strategy embodies C++’s long standing philosophy of “you don’t pay for what you don’t use.”

2.5 Endianness

Each multibyte scalar value used in the description of an IFC file header (§2.8) and in the table of contents (§2.9) is stored in little endian format. Multibyte scalar values stored in the partitions use the endianness of the target architecture.

2.6 Basic data types

This document uses a few fundamental data types, `u8`, `u16`, and `32` with the following characteristics:

- `u8`: 1 octet, with alignment 1; usually equivalent to C++’s `uint8_t`
- `u16`: 2 octets, with alignment 2; usually equivalent to C++’s `uint16_t`
- `u32`: 4 octets, with alignment 4; usually equivalent to C++’s `uint32_t`

File offset in bytes

At various places, the locations of certain tables (especially partitions) are described in terms of byte offset from the beginning of the current IFC file. The document uses the following abstract data type for those quantities.

```
enum class ByteOffset : uint32_t { };
```

Figure 2.3: Definition of type *ByteOffset*

Note: The current implementation uses 4-byte for file offsets, but that will change in coming updates to 8-byte in anticipation of large IFC file support.

Cardinality: counting items

At various places, the IFC indicates how many elements there are in a given table. That information is given by a 32-bit integer value abstracted as follows:

```
enum class Cardinality : uint32_t { };
```

Figure 2.4: Definition of type *Cardinality*

Extent of entities

At various places, in particular in partition summaries (§2.9), the IFC needs to indicate the number of bytes contain in entity representation. That information is indicated by a 32-bit value of type

```
enum class EntitySize : uint32_t { };
```

Figure 2.5: Definition of type *EntitySize*

Generic Indices

The type of generic indices into tables is defined as

```
enum class Index : uint32_t { };
```

Figure 2.6: Definition of type *Index*

Sequence

A sequence is generically described by a pair:

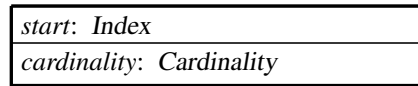


Figure 2.7: Structure of a sequence

The *start* field is an index into the partition the sequence is part of. It designates the first item in the sequence. The *cardinality* designates the number of items in the sequence.

Arrays

Occasionally, a contiguous sequence of n items of type T may be represented inline as having type $n \times T$. Such a group of items is called an array of n elements. Each item in that array is referred to via indices 0 through $n - 1$ inclusive. The inline representation means that the n items are represented consecutively in the containing structure.

Content Hash

The interesting portion of the content of an IFC file is hashed using SHA-256 algorithm, and stored as a value of type *SHA256*: A basic data type with 256 bits width, and with alignment 4.

File Format Versioning

Each IFC header has version information, major and minor of type defined as

```
enum class Version : uint8_t { };
```

Figure 2.8: Definition of type *Version*

ABI

The ABI targeted by an IFC is recorded in a field of the IFC header, of type

```
enum class Abi : uint8_t { };
```

Figure 2.9: Definition of type *Abi*

Architecture

The architecture targeted by an IFC is recorded in a field of the IFC header, of type

```
enum class Architecture : uint8_t {
    Unknown = 0x00,    // Unknown target
    X86      = 0x01,    // x86 (32-bit) target
```

```

X64      = 0x02,    // x64 (64-bit) target
ARM32    = 0x03,    // ARM (32-bit) target
ARM64    = 0x04,    // ARM (64-bit) target
HybridX86ARM64 = 0x05, // Hybrid x86-arm64
};

```

Language Version

The C++ language version is a 32-bit value of type

```
enum class LanguageVersion : uint32_t { };
```

Figure 2.10: Definition of type *LanguageVersion*

2.7 IFC File Signature

Each valid IFC file (see Figure 2.1) starts with the following 4-byte file signature:

```
0x54 0x51 0x45 0x1A
```

2.8 IFC File Header

Following the file signature (§2.7) is a header that describes a checksum, format version information, ABI information, target architecture information, C++ language version, the offset to the string table (§3) and how long it is, the name of the IFC’s module, the filename of the original C++ source file, the index of the global scope, then an indication on where to find the “table of contents” (§2.9), and finally how many partitions (§2.9) the IFC contains.

<i>checksum</i> : SHA256
<i>major_version</i> : Version
<i>minor_version</i> : Version
<i>abi</i> : Abi
<i>arch</i> : Architecture
<i>dialect</i> : LanguageVersion
<i>string_table_bytes</i> : ByteOffset
<i>string_table_size</i> : Cardinality
<i>unit</i> : UnitIndex
<i>src_path</i> : TextOffset
<i>global_scope</i> : ScopeIndex
<i>toc</i> : ByteOffset
<i>partition_count</i> : Cardinality
<i>internal</i> : bool

Figure 2.11: Structure of an IFC binary file header

The interpretation of the fields is as follows:

Content checksum The field *checksum* represents the SHA-256 hash of the portion of the IFC file content starting from right after that field to the end of the IFC file.

Version The fields *major_version* and *minor_version* collectively denote the version of the data structures in the IFC. The current format version is 0.25, meaning *major_version* is 0, and *minor_version* is 25.

Target ABI The field *abi* records the ABI of the target platform of the IFC.

Target Architecture The field *arch* records the architecture targeted by the IFC.

C++ Language Version The field *dialect* records the value of the C++ pre-defined macro `__cplusplus` in effect when the IFC was created.

String Table The field *string_table_bytes* is an offset from the beginning of the IFC to the first byte of the string table (§3). the number of bytes in table is indicated by *string_table_size*.

The string table holds the representation of any single strings or identifiers in the IFC. Consequently, locating it is essential for determining the IFC's module name, and also for locating partitions by name.

Translation Unit Descriptor A classification of the translation unit for which this IFC was generated is described by the field *unit*, value of type *UnitIndex* (§4).

Source Pathname The filename of the C++ source file out of which the IFC was produced is indicated by the field *src_path*, an offset into the string table. In the current implementation, this is an ordinary NUL-terminated narrow string.

Global Scope Every declaration is rooted in the global namespace. The field *global_scope* is index into the scope partition (§6.2), pointing to the description of the global namespace. Traversing the global scope, and recursively any contained declaration, gives the entire abstract semantics graph making up the IFC.

Table of Contents The table of contents is an array of all partition summaries in the IFC. The field *toc* indicates the offset (in bytes) from the beginning of the offset to the first partition descriptor. The number of partition summaries in the table of contents is given by the field *partition_count*.

Internal Unit Whether the IFC is for an exported module unit or not is indicated by *internal*. This field is false for all translation units are produced except non-exported module partitions.

Note The values of the major and minor versions, the ABI, and the architecture fields are not fixed yet. All multi-byte integer values in header are stored according to a little endian format. All multi-byte integer values stored in the partitions are stored according to the endianness of the target architecture. The structure of the field *internal* may change in future revisions.

2.9 IFC Table of Contents

The data in an IFC are essentially homogenous tables (called *partitions*) with entries referencing each other.

The table of contents is written near the end of the IFC file as that arrangement allows one-pass algorithms for writing out IFC files while minimizing the amount of intermediary internal storage needed to compute the full abstract semantics graph.

Partition

Each partition is described by a *partition summary* information with the following layout

<i>name</i> : <i>TextOffset</i>
<i>offset</i> : <i>ByteOffset</i>
<i>cardinality</i> : <i>Cardinality</i>
<i>entry_size</i> : <i>EntitySize</i>

Figure 2.12: Partition summary

name An index into the string pool. It points to the name (a NUL-terminated character string) of the partition.

offset Location (file offset in bytes) of the partition relative to the beginning of the IFC file.

cardinality The number of items in the partition.

entry_size The (common) size of an item in the partition.

So, the byte count of a partition is obtained by multiplying the individual *entry_size* by the *cardinality*.

2.10 Elaboration vs. syntax tree

The IFC, like the IPR, is designed to represent all of C++, including extensions. This means representing faithfully non-template entities as well as template entities. An *elaboration* of an entity is the result of full semantics analysis (e.g. the result of name lookup, type checking, overload resolution, template specialization if needed, etc.) of that entity. A node, in the abstract semantics graph of an IFC, representing a non-template is an elaboration.

By contrast, semantics analysis of templates proceeds in two steps, by language definition. For example, in a template code where *T* is a type parameter, the meaning of the expression `T{ 42 }` depends both on the meaning and structure of the actual argument value for *T*. It could be a constructor invocation, or a conversion function call, or a non-narrowing static cast. Consequently, representations of templates need to be fairly syntactic since only instantiations are fully semantically analyzed. Syntax trees (§15) are used to represent templates. That representation occasionally contains nodes that are elaborations, since a certain amount of semantics analysis is required when parsing template definitions.

Chapter 3

String table

Every IFC has a string table, a contiguous sequence of bytes. A regular C++ identifier is stored in the string table as a NUL-terminated sequence of bytes.

3.1 Index type

The type of the indices used to index into the string table is defined as follows

```
enum class TextOffset : uint32_t { };
```

Figure 3.1: Definition of type *TextOffset*

The representation of identifiers referenced by *TextOffset* indices uses UTF-8 encoding and are NUL-terminated. For string literals, see §10.1.

Chapter 4

Translation Units

Any C++ translation unit can be compiled into an IFC structure. A translation unit so represented can be referenced by abstract reference of type *UnitIndex*.

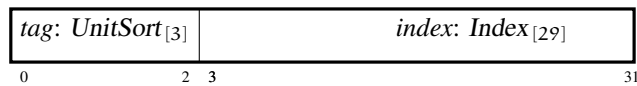


Figure 4.1: `UnitIndex`: Abstract reference of a declaration

The type *UnitSort* is a set of 3-bit values enumerated as follows

0x00. Source	0x03. Header
0x01. Primary	
0x02. Partition	0x04. ExportedTU

with meaning as explained in the sections below. The *index* value has a tag-dependent interpretation as defined below.

4.1 Translation unit structures

UnitSort::Source

A *UnitIndex* value with this tag designates a translation unit defined by a general C++ source file.

The *index* value is undefined and has no meaning.

UnitSort::Primary

A *UnitIndex* value with this tag designates a primary module interface.

The *index* is to be interpreted as a *TextOffset* value (§3.1), which is an offset into the string table (§3). The string at that offset is the name of the module (§5) for which this translation unit is a primary module interface.

UnitSort::Partition

A *UnitIndex* value with this tag designates a partition module unit.

The *index* is to be interpreted as a *TextOffset* value that designates the name of the module partition. The string of that name is *M:P*, obtained as the concatenation of the name *M* of the parent module, the colon character (:), and the relative name *P* of the partition.

UnitSort::Header

A *UnitIndex* value with this tag designates a C++ header unit. The *index* value is undefined and has no meaning.

UnitSort::ExportedTU

A *UnitIndex* value with this tag designates a translation unit compiled by the MSVC compiler with the compiler flags `/module:export` and `/module:name`. Such a translation is processed as if every toplevel declaration was prefixed with the keyword `export`. This is an MSVC extension.

The *index* is to be interpreted as a *TextOffset* value that designates the name specified via the compiler switch `/module:name`.

Note: An IFC unit of this sort is deprecated and scheduled for removal from MSVC.

Chapter 5

Modules

Any translation unit can import any module. Additionally, a module interface can re-export an imported module.

5.1 Module description structures

Module reference

All used modules (whether imported or exported) are represented as module references of type defined as follows

<i>owner: TextOffset</i>
<i>partition: TextOffset</i>

Figure 5.1: Structure of a *ModuleReference*

The fields of a module references have the following meanings:

- owner This value designates the name of the module. A null name indicated the global module.
- partition This value designates the partition of the owning module. When the partition name is null, the reference is to the primary module interface, otherwise it designates the partition of the owning module. When the owner is the global module then the partition designates the source file representing that partition of the global module.

Imported modules

References to all imported modules (which are not also exported) are stored in the imported modules partition.

Partition name: "module.imported".

Exported modules

References to all exported modules are stored in the exported modules partition.

Partition name: `"module.exported"`.

Chapter 6

Scopes

Every non-empty C++ translation unit contains at least one declaration, reachable from the global scope.

6.1 Scope index

A scope is referenced via an abstract reference of type *ScopeIndex* defined as

```
enum class ScopeIndex : uint32_t { };
```

Figure 6.1: Definition of type *ScopeIndex*

A value of type *ScopeIndex* is an index into the scope partition described below. Valid values start at 1. A *ScopeIndex* value 0 indicates a missing scope, not an empty scope.

6.2 Scope descriptor

A scope is a sequence of declarations (§6.3) – this definition is a generalization of standard C++’s. The *start* is an index into the scope member partition (§6.3), des-

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 6.2: Structure of a *Scope*

ignating the first declaration in the scope. The *cardinality* designates the number of declarations in the scope. Only members declared in that scope from that module partition are accounted for in the scope descriptor.

Partition name: "scope.desc".

Note: The *global_scope* field of the table of contents (§2.8) is index into this partition.

6.3 Scope member

A scope member represents a declaration.

A rectangular box with a black border containing the text *index: DeclIndex*.

Figure 6.3: Structure of a *Declaration* – a scope member

The *index* field of a declaration is a *DeclIndex* value designating the entity (§8) referenced by this declaration.

Partition name: "scope.member".

Note: At this point in time, a *Declaration* is just a structure with an index as member. In the future, it may evolve to contain explicitly attributes such as 'imported', 'exported', or 'internal'.

Chapter 7

Heaps

At various places, there is a need to describe a sequence of objects of a given (common) sort but of differing kinds. For example, a namespace contains only declarations, but those declarations can be of different kinds; e.g. function declaration, variable declaration, template declaration, etc. Those tables are represented as sequences of homogenous indices of one sort: a slice of a heap of indices.

7.1 Heap structures

Declaration heap

The declarations heap is a partition consisting entirely of *DeclIndex* values.

Partition name: `"heap.decl"`.

Type heap

The types heap is a partition consisting entirely of *TypeIndex* values.

Partition name: `"heap.type"`.

Statement heap

The statement heap is a partition consisting entirely of *StmtIndex* values.

Partition name: `"heap.stmt"`.

Expression heap

The expression heap is a partition consisting entirely of *ExprIndex* values.

Partition name: `"heap.expr"`.

Syntax heap

The syntax heap is a partition consisting entirely of *SyntaxIndex* values.

Partition name: "heap.syn".

Form heap

The preprocessing form heap is a partition consisting entirely of *FormIndex* values.

Partition name: "heap.form".

Chart heap

The chart heap is a partition consisting entirely of *ChartIndex* values.

Partition name: "heap.chart".

Attribute heap

The attribute heap is a partition consisting entirely of *AttrIndex* values.

Partition name: "heap.attr".

Chapter 8

Declarations

Declarations are indicated by abstract declaration references. This document uses *DeclIndex* to designate a typed abstract reference to a declaration. Like all abstract references, it is a 32-bit value

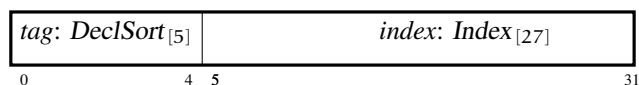


Figure 8.1: *DeclIndex*: Abstract reference of declaration

The type *DeclSort* is a set of 5-bit values enumerated as follows

0x00. VendorExtension	0x10. Method
0x01. Enumerator	0x11. Constructor
0x02. Variable	0x12. InheritedConstructor
0x03. Parameter	0x13. Destructor
0x04. Field	0x14. Reference
0x05. Bitfield	0x15. UsingDeclaration
0x06. Scope	0x16. UsingDirective
0x07. Enumeration	0x17. Friend
0x08. Alias	0x18. Expansion
0x09. Temploid	0x19. DeductionGuide
0x0A. Template	0x1A. Barren
0x0B. PartialSpecialization	0x1B. Tuple
0x0C. Specialization	0x1C. SyntaxTree
0x0D. UnusedSort0	0x1D. Intrinsic
0x0E. Concept	0x1E. Property
0x0F. Function	0x1F. OutputSegment

Note: The individual values a *DeclSort* enumerator is subject to change at any moment until the design stabilizes.

8.1 Declaration vocabulary types

The description of declarations uses a set of common types values as described below.

Access specifiers

Every non-local declaration has an access specifier, of type:

```
enum class Access : uint8_t {
    None,          // No access specifier
    Private,       // "private" for scope member
    Protected,     // "protected" for scope member
    Public,        // "public" for scope member
};
```

Basic specifiers

Certain cumulative properties common to all declarations are described by the bitmask type *BasicSpecifiers*:

```
enum class BasicSpecifiers : uint8_t {
    Cxx          = 0,          // C++ language linkage
    C            = 1 << 0,    // C language linkage
    Internal     = 1 << 1,    //
    Vague        = 1 << 2,    // Vague linkage, e.g. COMDAT, still external
    External     = 1 << 3,    // External linkage.
    Deprecated   = 1 << 4,    // [[deprecated("foo")]]
    InitializedInClass = 1 << 5, // defined or initialized in a class
    NonExported  = 1 << 6,    // Not explicitly exported
    IsMemberOfGlobalModule = 1 << 7 // member of the global module
};
```

Note: The definition of *BasicSpecifiers* may change in the future, and may in fact be part of *Declaration* (§6.3). The numerical values assigned to these symbolic constants are subject to change.

Reachable semantic properties

In certain circumstances, the IFC stores more information than the bare minimum required by the ISO C++ Modules specification. In such cases, it is necessary to know which semantic properties are reachable, outside the owning module, to the importers. In other circumstances, known such additional information is useful in

performing additional checks such as ODR violation detection. The availability of such supplementary information is indicated by the bitmask *ReachableProperties*

```
enum class ReachableProperties : uint8_t {
    None           = 0,          // nothing beyond name, type, scope.
    Initializer     = 1 << 0,    // IPR-initializer exported.
    DefaultArguments = 1 << 1,    // function or template default arguments exported
    Attributes      = 1 << 2,    // standard attributes exported.
    All             = 0xff,       // Everything.
};
```

Object traits

Certain cumulative properties common to all data/object declarations are described by the bitmask type *ObjectTraits*

```
enum class ObjectTraits : uint8_t {
    None           = 0,
    Constexpr      = 1 << 0,
    Mutable        = 1 << 1,
    ThreadLocal    = 1 << 2,
    Inline         = 1 << 3,
    InitializerExported = 1 << 4,
    Vendor         = 1 << 7,
};
```

Note: The definition of *ObjectTraits* is subject to change. The numerical values assigned to these symbolic constants are subject to change.

Vendor traits

Declarations of certain entities may be endowed with vendor-specific traits. The MSVC-specific traits are defined by the following enumeration

```
enum class MsvcTraits : uint32_t {
    None           = 0,
    ForceInline    = 1 << 0,
    Naked          = 1 << 1,
    NoAlias        = 1 << 2,
    NoInline       = 1 << 3,
    Restrict       = 1 << 4,
    SafeBuffers    = 1 << 5,
    DllExport       = 1 << 6,
    DllImport      = 1 << 7,
    CodeSegment    = 1 << 8,
    Novtable       = 1 << 9,
```

```

    IntrinsicType      = 1 << 10,
    EmptyBases         = 1 << 11,
    Process            = 1 << 12,
    Allocate           = 1 << 13,
    SelectAny          = 1 << 14,
    Comdat             = 1 << 15,
    Uuid               = 1 << 16,
};

```

Parameter Level

A template declaration can have many nesting levels. This is the case of member templates of class templates; that is a member of a class template, that is itself a template. Parameter nesting level starts from 1. The nesting level is given by a value of type *ParameterLevel* defined as

```
enum class ParameterLevel : uint32_t { };
```

Figure 8.2: Definition of type *ParameterLevel*

Parameter Position

A parameter at a given level can be identified by its position in its enclosing parameter list. The position of a template parameter is given by a value of type *ParameterPosition*, defined as

```
enum class ParameterPosition : uint32_t { };
```

Figure 8.3: Definition of type *ParameterPosition*

8.2 Declaration structures

DeclSort::VendorExtension

A *DeclIndex* value with tag *DeclSort::VendorExtension* represents an abstract reference to a vendor-specific declaration. This tag value is reserved for encoding vendor-specific extensions.

Partition name: "decl.vendor-extension".

DeclSort::Enumerator

A *DeclIndex* value with tag *DeclSort::Enumerator* represents an abstract reference to an enumerator declaration. The *index* field is an index into the enumerator declaration partition. Each entry in that partition is a structure with the following compo-

nents: a *name* field, a *locus* field, a *type* field, an *initializer* field, a *specifier* field, and an *access* field.

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>specifier</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>

Figure 8.4: Structure of an enumerator declaration

The *name* field denotes the C++ source-level name of the enumerator. The *locus* field denotes the source location. The *type* field denotes the type of the enumerator. The *initializer* field denotes the value or the initializer of the enumerator. The *specifier* field denotes the specifiers of the enumerator. The *access* field denotes the C++ source-level access specifier of the enumerator.

Partition name: "decl.enumerator".

DeclSort::Variable

A *DeclIndex* value with tag *DeclSort::Variable* represents an abstract reference to a variable declaration. Note that static data members are also semantically variables and are represented as such. The *index* field is an index into the variable declaration partition. Each entry in that partition is a structure with the following components: a *name* field, a *locus* field, a *field*, a *home_scope* field, an *initializer* field, an *alignment* field, a *specifier* field, a *traits* field, and an *access* field.

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>alignment</i> : <i>ExprIndex</i>
<i>traits</i> : <i>ObjectTraits</i>
<i>specifier</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.5: Structure of a variable declaration

The *name* field denotes the name of the variable. Note that it can be a plain identifier (a *TextOffset* into the string table), or something as elaborated as a template-id (for specializations of variable templates). The *locus* field denotes the source location. The *type* field denotes the C++ source-level type of the variable. The *home_scope* field denotes the scope declaration that holds the object the variable designates. The *home_scope* is not necessarily the lexical scope of a variable: for instance, a block-scope 'extern' declaration of a variable names a variable whose home scope is in the nearest enclosing namespace scope. The *initializer* field denotes the initializer expression in the variable declaration. The *alignment* field denotes the alignment of the variable. The *specifier* field denotes the declarations specifiers of the variable. The *traits* field denotes additional traits associated with the variable. The *properties* field indicates which semantic properties are reachable to the importers.

Partition name: "decl.variable".

DeclSort::Parameter

A *DeclIndex* value with *tag* *DeclSort::Parameter* is an abstract reference to either a function parameter or a template parameter declaration. The *index* field is an index into the parameter declaration partition. Each entry in that partition is a structure with the following layout

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>constraint</i> : <i>ExprIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>level</i> : <i>ParameterLevel</i>
<i>position</i> : <i>ParameterPosition</i>
<i>sort</i> : <i>ParameterSort</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.6: Structure of a template parameter declaration

and these meanings of the fields:

- *name* denotes the name of the template parameter. If null, the template parameter was unnamed in the source input.
- *locus* denotes the location of the template parameter.
- *type* designates the type of the parameter.
- *constraint* designates the concept predicate used to declare this parameter, if the abbreviated form was used at the input source level.

- *initializer* designates the corresponding default argument, if any.
- *level* denotes the nesting level of this parameter
- *position* denotes the position of this parameter in the parameter list
- *sort* denotes the sort of parameter (function-parameter vs template-parameter)
- *properties* denotes the set of reachable properties of this parameter.

If the parameter declaration was that of a pack, then its type is denoted by a pack expansion (`TypeSort::Expansion`).

Note: This representation will change in the future as it is currently too irregular and too tightly coupled with VC++ internal representation oddities.

Partition name: "decl.parameter".

Parameter sort

The various notions of parameters (function parameter, type template parameter, non-type template parameter, template template parameter) are described by:

```
enum class ParameterSort : uint8_t {
    Object,           // Function parameter
    Type,             // Type template parameter
    NonType,          // Non-type template parameter
    Template,         // Template template parameter
};
```

DeclSort::Field

A *DeclIndex* value with tag `DeclSort::Field` represents an abstract reference to the representation of a non-static data member declaration. The *index* field is an index into the field declaration partition. Each entry in that partition is a structure with the following components: a *name* field, a *locus* field, a *type* field, a *home_scope* field, an *initializer* field, an *alignment* field, a *specifier* field, a *traits* field, and an *access* field.

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>alignment</i> : <i>ExprIndex</i>
<i>traits</i> : <i>ObjectTraits</i>
<i>specifier</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.7: Structure of a field declaration

The *name* field denotes the name of the non-static data member. The *locus* field denotes the source location. The *type* field denotes the C++ source-level type of the non-static data member. The *home_scope* field denotes the scope declaration that holds the member declaration. The *initializer* field denotes the initializer expression in the member declaration. The *alignment* field denotes the alignment of the non-static data member. The *specifier* field denotes the declarations specifiers of the non-static data member. The *traits* fields denotes additional traits associated with the non-static data member.

Partition name: "decl.field".

DeclSort::Bitfield

A *DeclIndex* value with tag *DeclSort::Bitfield* represents an abstract reference to the representation of a bitfield declaration. The *index* field is an index into the bitfield declaration partition. Each entry in that partition is a structure with the following components: a *name* field, a *locus* field, a *type* field, a *home_scope* field, a *width* field, an *initializer* field, a *specifier* field, a *traits* field, and an *access* field.

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>width</i> : <i>ExprIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>traits</i> : <i>ObjectTraits</i>
<i>specifier</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.8: Structure of a bitfield declaration

The *name* field denotes the name of the bitfield. The *locus* field denotes the source location. The *type* field denotes the C++ source-level type of the bitfield. The *home_scope* field denotes the scope declaration that holds the bitfield declaration. The *width* field denotes the number bits specified in the bitfield declaration. The *initializer* field denotes the initializer expression in the bitfield declaration. The *specifier* field denotes the declarations specifiers of the bitfield. The *traits* fields denotes additional traits associated with the bitfield.

Partition name: "decl.bitfield".

DeclSort::Scope

A *DeclIndex* abstract reference with tag *DeclSort::Scope* designates a class-type or a namespace definition. The *index* field of that abstract reference is an index into the scope declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>base</i> : <i>TypeIndex</i>
<i>initializer</i> : <i>ScopeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>alignment</i> : <i>ExprIndex</i>
<i>pack_size</i> : <i>PackSize</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>traits</i> : <i>ScopeTraits</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.9: Structure of a scope declaration

The *name* field designates the name of the scope. The *locus* field designates the source location. The *type* field indicates the kind (§9.1) of scope:

- `TypeBasis::Struct` means the scope was declared as "struct"
- `TypeBasis::Class` means the scope was declared as "class"
- `TypeBasis::Union` means the scope was declared as "union"
- `TypeBasis::Namespace` means the scope was declared as "namespace"
- `TypeBasis::Interface` means the scope was declared as "__interface"

Any other value is invalid.

The *base* field designates the base class(es) in case of inheritance. The *initializer* field designates the body of the scope definition, e.g. the sequence of declarations. Note that valid *ScopeIndex* values start from 1, and 0 indicates absence of scope, e.g. an incomplete class type. The *home_scope* field designates the declaration of the enclosing scope. The *alignment* field designates the alignment value of the scope, in case of class-type. The *pack_size* field designates the packing value applied to the layout of the scope, in case of class-type. The *specifiers* field indicates the (cumulative) basic declaration specifiers that hold for the scope. The *traits* field designates scope-specific properties of the scope. The *access* field designates the access specifier of the scope declaration. The *properties* field designates the set of reachable semantic properties.

Partition name: "decl.scope".

Scope traits

Properties specific to scope entities are described by values of the bitmask type *ScopeTraits*:

```
enum class ScopeTraits : uint8_t {
    None      = 0,
    Unnamed   = 1 << 0,
    Inline     = 1 << 1,
    InitializerExported = 1 << 2,
    ClosureType = 1 << 3,
    Final      = 1 << 4,
    Vendor     = 1 << 7,
};
```

with the following meaning:

- `ScopeTraits::None`: No scope traits.
- `ScopeTraits::Unnamed`: the scope is unnamed in the input source code.
- `ScopeTraits::Inline`: valid only for namespaces. The namespace is declared `inline`.
- `ScopeTraits::InitializedExported`: valid only if the definition of this scope entity is lexically exported, in particular this indicates whether completeness of types is exported.
- `ScopeTraits::ClosureType`: valid only for class types. This trait indicates that the scope represents a closure type.
- `ScopeTraits::Final`: valid only for class types. This trait indicates that the scope is defined `final`.
- `ScopeTraits::Vendor`: valid only if the scope entity has vendor-defined traits.

Class-type layout packing

A value of class layout packing is expressed as value of type *PackSize* defined as

```
enum class PackSize : uint16_t { };
```

Figure 8.10: Definition of type *PackSize*

DeclSort::Enumeration

A *DeclIndex* abstract reference with tag `DeclSort::Enumeration` designates an enumeration declaration. The *index* of that abstract reference is an index into the enumeration declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>base</i> : <i>TypeIndex</i>
<i>initializer</i> : <i>Sequence</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>alignment</i> : <i>ExprIndex</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.11: Structure of an enumeration declaration

The *name* field designates the name of the enumeration type. The *locus* field designates the source location. The *type* field designates the kind of enumeration, with:

- `TypeBasis::Enum` meaning a classic enumeration
- `TypeBasis::Class` or `TypeBasis::Struct` meaning a scoped enumeration

The *base* field designates the underlying type of the enumeration. The *initializer* is a slice (§2.6) of the enumerator partition (§8.2). It designates the sequence of enumerators (if any) declared as part of the enumeration declaration. The *home_scope* field designates the declaration of the enclosing scope of the enumeration. The *alignment* designates the alignment specified in the declaration. A non-zero value indicates an explicit alignment specification in the input source code. The *specifiers* designates the basic generic declaration specifiers of the enumeration. The *access* designates the access specifier. The *properties* designates the set of reachable semantic properties.

Partition name: "decl.enum".

DeclSort::Alias

A *DeclIndex* abstract reference with tag `DeclSort::Alias` designates a type alias declaration. The *index* field of that reference is an index into the type alias declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>aliasee</i> : <i>TypeIndex</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>

Figure 8.12: Structure of a type alias declaration

- The *name* field designates the name of the alias.
- The *locus* field designates the source location.
- The *type* field denotes the kind of alias: it denotes `TypeBasis::Typename` for type aliases; it will be `TypeBasis::Namespace` for namespace aliases; it is abstract reference of sort `TypeSort::Forall` for template aliases.
- The *home_scope* field designates the declaration of the enclosing scope.
- The *aliasee* field designates the type the alias is declared for.
- The *specifiers* field designates the basic declaration specifiers for the alias.
- The *access* field designates the access specifier for the alias.

This structure is also used to represent template aliases – mistakenly called alias templates in the ISO C++ document. For template aliases, the *aliasee* field denotes a `TypeSort::Forall` type (**TypeSort::Forall**), which is the representation of the *template-parameter* list followed by the *type-id* that would syntactically appear on the right hand side of the source-level *using-declaration*.

Partition name: "decl.alias".

DeclSort::Temploid

A member of a parameterized scope – does not have template parameters of its own.

<i>entity</i> : <i>ParameterizedEntity</i>
<i>chart</i> : <i>ChartIndex</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.13: Structure of a templated declaration

The *entity* field represents the declaration being parameterized. The *chart* field designates the set of template parameter lists of the enclosing scope. The *properties* field designates the set of reachable semantics properties.

Partition name: "decl.temploid".

Parameterized Entity

The structure *ParameterizedEntity* has the following layout

<i>decl</i> : <i>DeclIndex</i>
<i>head</i> : <i>SentenceIndex</i>
<i>body</i> : <i>SentenceIndex</i>
<i>attributes</i> : <i>SentenceIndex</i>

Figure 8.14: Structure of a declaration parameterized by a template

The *decl* field denotes the declaration that is being parameterized either directly or indirectly by a set of template parameter lists. The *head* field designates the sentence (§19.1) that makes up the non-defining declarative part of the current instantiation. That sentence is no longer meaningful in recent releases of MSVC since any semantics information can be obtained from the entity denoted by *decl*. The *body* field denotes the sentence of the defining ("body") part of the current instantiation. This field is meaningful only for templated functions.

DeclSort::Template

A template declaration: class, function, constructor, type alias, variable.

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>entity</i> : <i>ParameterizedEntity</i>
<i>type</i> : <i>TypeIndex</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.15: Structure of a template declaration

The *name* field denotes the name of this template. The *locus* field denotes the source location of this declaration. The *home_scope* field designate the home scope of this

template. The *chart* field denotes the set of parameter list to this template. The *entity* field describes the declaration being parameterized by this template. Its structure is defined below. The *type* field denotes the type of this template declaration. The *specifiers* field denotes declaration specifiers for this template. The *access* field denotes the access level of this declaration. The *properties* field designates the set of reachable semantic properties.

Partition name: "decl.template".

DeclSort::PartialSpecialization

A partial specialization of a template (class-type or function).

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>entity</i> : <i>ParameterizedEntity</i>
<i>form</i> : <i>Index</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.16: Structure of a partial specialization declaration

The *name* field denotes the name of the current instantiation of the partial specialization. The *locus* field denotes the source location. The *home_scope* field denotes the parent declaration of this partial specialization. The *chart* field denotes the set of template-parameter lists of this partial specialization. The *entity* field describes the current instantiation of this partial specialization. The *form* field is an index into the partition of specialization form (template and template-argument list) named "form.spec". The *specifiers* field denotes the declaration specifiers of this partial specialization. The *access* field denotes the access level of this partial specialization. The *properties* field denotes the set of reachable semantic properties.

Partition name: "decl.partial-specialization".

DeclSort::Specialization

A *DeclIndex* value with tag *DeclSort::Specialization* designates a specialization of template declaration. The *index* field of that abstract reference is an index into the specialization partition. Each entry in that partition is a structure with the following layout

<i>form</i> : <i>Index</i>
<i>decl</i> : <i>DeclIndex</i>
<i>sort</i> : <i>SpecializationSort</i>

Figure 8.17: Structure of a template declaration specialization

and meanings of the fields

- The value of *form* is an index into the specialization form partition (named "form.spec").
- *decl*, when non null, denotes the declaration produced by the specialization.
- *sort* designates how the specialization is obtained (§8.2).

Partition name: "decl.specialization".

How to specialize a template

The method by which the declaration for a template specialization is produced can be indicated by a value of type

```
enum class SpecializationSort : uint8_t {
    Implicit = 0x0,
    Explicit = 0x1,
    Instantiation = 0x2,
};
```

with the following meaning

- `SpecializationSort::Implicit`: the declaration is an implicit specialization.
- `SpecializationSort::Explicit`: the declaration is an explicit specialization.
- `SpecializationSort::Instantiation`: the declaration is an explicit instantiation.

`DeclSort::UnusedSort0`

No structure associated with this sort.

`DeclSort::Concept`

A *DeclIndex* value with tag `DeclSort::Concept` represents an abstract reference to a concept declaration. The *index* of that abstract reference is an index into the concept definition partition. Each entry in that partition is a structure with the following layout

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>type</i> : <i>TypeIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>constraint</i> : <i>ExprIndex</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>head</i> : <i>SentenceIndex</i>
<i>body</i> : <i>SentenceIndex</i>

Figure 8.18: Structure of a concept declaration

and meaning of the fields:

- *name* designates the name of the concept
- *locus* designates the source location of the concept definition
- *home_scope* designates the declaration of the enclosing scope
- *type* designates the full type of the concept definition
- *chart* is the parameter list list to the concept
- *constraint* is the body of predicate defining the concept
- *specifiers* is the set of basic declaration specifiers
- *access* is the access specifier for the concept definition
- *head* is the sequence of words making up the declarative part of the concept
- *body* is the sequence of words making up the body of the concept definition

Partition name: "decl.concept".

Note: This representation is subject to change in future releases.

DeclSort::Function

A *DeclIndex* value with tag *DeclSort::Function* represents an abstract reference to a function declaration. Note that a static member function is represented as a function. The *index* field is an index into the function declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.19: Structure of a function declaration

The *name* field designates the name of the function. The *locus* field designates the source location. The *type* field designates the type of the function, including the noexcept-specification (which is now part of the type of a function in C++17). The *home_scope* denotes the enclosing scope of the function. This may not be the lexical scope of the declaration. The *chart* denotes the chart (§13) of the function parameter list along with their default arguments. These parameters may be unnamed. The *specifier* denotes basic declaration specifiers; the *traits* field adds additional function traits. Finally, the *access* denotes the access specifier for the function.

Note: The set of parameter declarations in a function definition is listed in a separate trait (§17.9). That representation is subject to removal in future MSVC releases.

Partition name: "decl.function".

Function traits

Certain function-specific cumulative properties are expressed as values of the bitmask type *FunctionTraits* defined as

```
enum class FunctionTraits : uint16_t {
    None      = 0,
    Inline    = 1 << 0,
    Constexpr = 1 << 1,
    Explicit  = 1 << 2,
    Virtual   = 1 << 3,
    NoReturn  = 1 << 4,
    PureVirtual = 1 << 5,
    HiddenFriend = 1 << 6,
    Defaulted  = 1 << 7,
    Deleted    = 1 << 8,
    Constrained = 1 << 9,
    Immediate  = 1 << 10,
```

```
Vendor      = 1 << 15,
};
```

with the following meaning

`FunctionTraits::None`: no property

`FunctionTraits::Inline`: the function is declared `inline`

`FunctionTraits::Constexpr`: the function is declared `constexpr`

`FunctionTraits::Explicit`: the function is declared `explicit`

`FunctionTraits::Virtual`: the function is declared `virtual`

`FunctionTraits::NoReturn`: the function is declared `[[noreturn]]` or `__declspec(noreturn)`

`FunctionTraits::PureVirtual`: the function is pure virtual, e.g. with `= 0`

`FunctionTraits::HiddenFriend`: the function is a hidden friend

`FunctionTraits::Constrained`: the function has `requires`-constraints

`FunctionTraits::Immediate`: the function is a `constexpr`, or an immediate function.

`FunctionTraits::Vendor`: the function has vendor-defined traits stored in the MSVC vendor-specific traits (§17.9).

DeclSort::Method

A *DeclIndex* abstract reference with tag `DeclSort::Method` designates a non-static member function (which is neither a constructor nor a destructor) declaration. The *index* of that abstract reference is an index into the non-static member function declaration partition. Each entry in that partition is a structure with the following components

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.20: Structure of a non-static member function declaration

The *name* field designates the name of the non-static member function. The *locus* field designates the source location of this declaration. The *type* field designates the type of this non-static member function. Note that the type also includes the calling convention. The *home_scope* designates the enclosing type declaration. The *chart* designates the function parameter list along with their default arguments. The *traits* indicates any additional function-specific traits (§8.2). The *access* designates the access specifier for this function.

Partition name: "decl.method".

DeclSort::Constructor

A *DeclIndex* abstract reference with tag *DeclSort::Constructor* designates a constructor declaration. The *index* field of that abstract reference is an index into the constructor declaration partition. Each entry in that partition is a structure with the following components.

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.21: Structure of a constructor declaration

The meaning of the fields is as follows:

- *name* designates the name of the constructor.
- *locus* denotes the source location of the constructor.
- *type* denotes the type of this constructor, a type described in *TypeSort::Tor*.
- *home_scope* denotes the declaration of the enclosing type.
- *chart* denotes the function parameter list along with their default argument expressions.
- *traits* designates function-specific traits (§8.2).
- *specifiers* designate the usual basic declaration specifiers.

- *access* designates the access specifier of the constructor.
- *properties* denotes the set of reachable semantic properties of this constructor declaration.

Partition name: "decl.constructor".

Note: The *name* field is subject of further design modification
The structure *NoexceptSpecification* has the following layout

<i>words</i> : <i>SentenceIndex</i>
<i>sort</i> : <i>NoexceptSort</i>

Figure 8.22: Structure of a *noexcept*-specification

The *words* field denotes the sentence making up the syntax of the *noexcept*-specification. This field is meaningful only for templated functions for which the *noexcept*-specification is a dependent expression. The *sort* field describes the computed semantics, if not dependent. It has type

```
enum class NoexceptSort : uint8_t {
    None,
    False,
    True,
    Expression,
    Inferred,
    Unenforced,
};
```

with the following meaning

- *NoexceptSort::None*: No specification is lexically present in the input source
- *NoexceptSort::False*: The syntax *noexcept(false)* was explicitly used, or the determination has similar semantic effect
- *NoexceptSort::True*: The syntax *noexcept(true)* was explicitly used, or the determination has similar semantic effect.
- *NoexceptSort::Expression*: The syntax *noexcept(expr)* was explicitly used, and no determination could be made because the expression is dependent.
- *NoexceptSort::Inferred*: The *noexcept* specification (for a special member) is inferred and dependent on that of the associated functions the special member invokes from base class subobjects or non-static data members.
- *NoexceptSort::Unenforced*: This is the specification for the static type system, but with no runtime termination enforcement

DeclSort::InheritedConstructor

A *DeclIndex* abstract reference with tag `DeclSort::InheritedConstructor` designates an inherited constructor. The *index* of that abstract reference is an index into the inherited constructor partition. Each entry in that partition is a structure with the following layout

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>chart</i> : <i>ChartIndex</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>base_ctor</i> : <i>DeclIndex</i>

The meaning of the fields is as follows

- *name* designates the name of this constructor
- *locus* denotes the source location of this declaration.
- *type* denotes the type of this constructor, a type described in `TypeSort::Tor`.
- *home_scope* denotes the declaration of the enclosing type.
- *chart* denotes the function parameter list along with their default argument expressions.
- *traits* designates function-specific traits (§8.2).
- *specifiers* designate the usual basic declaration specifiers.
- *access* designates the access specifier of the constructor.
- *base_ctor* denotes the constructor from the base class this constructor inherits.

Partition name: `"decl.inherited-constructor"`.

DeclSort::Destructor

A *DeclIndex* abstract reference with tag `DeclSort::Destructor` designates a destructor declaration. The *index* field of that abstract reference is an index into the destructor declaration partition. Each entry in that partition is a structure with the following components.

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>eh_spec</i> : <i>NoexceptSpecification</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>convention</i> : <i>CallingConvention</i>
<i>properties</i> : <i>ReachableProperties</i>

Figure 8.23: Structure of a destructor declaration

The *name* field designates the name of the destructor declaration. The *locus* field designates the source location of the declaration. The *home_scope* field designates the declaration of the enclosing type. The *eh_spec* field designates the exception specification of the declaration. The *traits* field designates function-specific properties (§8.2) of the declaration. The *specifiers* field designate the usual basic declaration specifiers. The *access* field designates the access specifier of the destructor declaration. The *convention* field designates the calling convention used by the destructor.

Note: The *name* field is subject of further design modification

Partition name: "decl.destructor".

DeclSort::Reference

A *DeclIndex* abstract reference with tag *DeclSort::Reference* designates a reference to a declaration made available by an imported module. The *index* field of that abstract reference is an index into the declaration reference partition. Each entry in that partition is a structure with the following components:

<i>unit</i> : <i>ModuleReference</i>
<i>local_index</i> : <i>DeclIndex</i>

Figure 8.24: Structure of a declaration reference declaration

The *unit* field designates the owning translation unit. The *local_index* is the *DeclIndex* abstract reference assigned to that entity by the current (importing) module unit.

Partition name: "decl.reference".

ModuleReference

The type *ModuleReference* is a structure with the following layout

<i>owner</i> : <i>TextOffset</i>
<i>partition</i> : <i>TextOffset</i>

Figure 8.25: Structure of a module reference

Then the *owner* field is null, then it means the IFC comes from the global module, and the *partition* field is the name of the source file out of which the header unit was built. Otherwise, *owner* is the name of the owning module of the IFC, and the *partition* designates the module partition when it is not null.

DeclSort::UsingDeclaration

A *DeclIndex* abstract reference with tag *DeclSort::UsingDeclaration* designates a using declaration. The *index* field of that abstract reference is an index into the using declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>resolution</i> : <i>DeclIndex</i>
<i>parent</i> : <i>ExprIndex</i>
<i>name2</i> : <i>TextOffset</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>
<i>hidden</i> : <i>bool</i>

Figure 8.26: Structure of a using-declaration structure

The *name* field is the unqualified part of the using declaration. The *locus* field is the source location of the declaration. The *home_scope* field designates the enclosing scope of the declaration. The *resolution* field designates the set of used declarations: either a single declaration, or a tuple of declarations. The *parent* field designates the qualifying part of the declaration. The *name2* is a redundant field referring to the name of the member. The *specifiers* field denotes the basic declaration specifiers. The *hidden* field indicates whether the member is hidden.

Partition name: "decl.using-declaration".

Note: This representation is subject to change.

DeclSort::UsingDirective

A *DeclIndex* abstract reference with tag `DeclSort::UsingDirective` designates a using directive.

TBD.

Note: The structure of this declaration is not yet defined.

Partition name: `"decl.using-directive"`.

DeclSort::Friend

A *DeclIndex* abstract reference with tag `DeclSort::Friend` designates a friend declaration. The *index* field of that abstract reference is an index into the friend declaration partition. Each entry in that partition is a structure with the following component:

<i>entity</i> : <i>ExprIndex</i>

Figure 8.27: Structure of a friend declaration

The *entity* field denotes an expression that references the declared friend. For instance, *entity* would denote a named declaration (`ExprSort::NamedDecl`) if the input source-level friend declaration references a lexically declared entity. If that friend is a specialization of a template then *entity* would denote the corresponding *template-id* (`ExprSort::TemplateId`).

Partition name: `"decl.friend"`.

DeclSort::Expansion

A *DeclIndex* abstract reference with tag `DeclSort::Expansion` designates a declaration obtained by expanding a pack. The *index* field is an index into the expansion declaration partition. Each entry in that partition is a structure with the following layout

<i>operand</i> : <i>DeclIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 8.28: Structure of an expansion declaration

Partition name: `"decl.expansion"`.

DeclSort::DeductionGuide

A *DeclIndex* abstract reference with tag `DeclSort::DeductionGuide` designates a deduction guide declaration. The *index* field is an index into the deduction guide partition. Each entry in that partition is a structure with the following layout

<i>name</i> : <i>NameIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>parameters</i> : <i>ChartIndex</i>
<i>specialization</i> : <i>TypeIndex</i>
<i>traits</i> : <i>FunctionTraits</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>

Figure 8.29: Structure of a deduction guide declaration

Partition name: `"decl.deduction-guide"`.

DeclSort::Barren

A *DeclIndex* abstract reference with tag `DeclSort::Barren` designates a declaration that introduces no name, e.g. *asm-declaration*, *static_assert-declaration*, *attribute-declaration*, *empty-declaration*. The *index* field is an index into the barren declaration partition. Each entry in that partition is a structure with the following layout

TBD.

Partition name: `"decl.barren"`.

DeclSort::Tuple

A *DeclIndex* abstract reference with tag `DeclSort::Tuple` designates a sequence of declarations referenced in other source-level constructs, e.g. a set of bindings during parsing of templated declarations. The *index* field is an index into the tuple declaration partition. Each entry in that partition is a structure with the following layout

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 8.30: Structure of a tuple declaration

The *start* field is an index into the declaration heap partition (§7.1) pointing to the first declaration in the tuple sequence. The *cardinality* field denotes the number of declaration in the tuple sequence.

Partition name: "decl.tuple".

DeclSort::SyntaxTree

A syntax tree in template declaration.

TBD

Partition name: "decl.syntax-tree".

Note: This is a vendor extension, the meta description of which is subject to change

DeclSort::Intrinsic

A *DeclIndex* abstract reference with tag `DeclSort::Intrinsic` designates an intrinsic function declaration. The *index* field of that abstract reference is an index into the intrinsic function declaration partition. Each entry in that partition is a structure with the following components:

<i>name</i> : <i>TextOffset</i>
<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>home_scope</i> : <i>DeclIndex</i>
<i>specifiers</i> : <i>BasicSpecifiers</i>
<i>access</i> : <i>Access</i>

Figure 8.31: Structure of an intrinsic function declaration

Partition name: "decl.intrinsic".

Note: This is a vendor extension, the meta description of which is subject to change

DeclSort::Property

A *DeclIndex* abstract reference with tag `DeclSort::Property` designates MSVC's extension of "property declaration." The *index* field is an index into the property declaration partition. Each entry in that partition is a structure with the following layout

<i>member</i> : <i>DeclIndex</i>
<i>getter</i> : <i>TextOffset</i>
<i>setter</i> : <i>TextOffset</i>

Figure 8.32: Structure of a property declaration

The *member* field designates the declaration of the non-static data member being declared a property. The *getter* and the *setter* fields denote the names of the getters and the setters non-static member functions associated with the property.

Partition name: "decl.property".

Note: This is a vendor extension, the meta description of which is subject to change

DeclSort::OutputSegment

Code segment. These are 'declared' via pragmas.

A *DeclIndex* abstract reference with tag *DeclSort::OutputSegment* designates an MSVC extension declaration of a code segment — typically declared with a pragma. The *index* field is an index into the code segment partition. Each entry in that partition is a structure with the following layout

<i>name</i> : <i>TextOffset</i>
<i>ID</i> : <i>TextOffset</i>
<i>traits</i> : <i>SegmentTraits</i>
<i>type</i> : <i>SegmentType</i>

Figure 8.33: Structure of a code segment declaration

The *name* field denotes the name of the code segment. The *ID* field denotes the class-ID of the code segment. The *traits* field denotes the set of traits of the code segment. The *type* field denotes the code segment type.

Code segment traits

Each code segment is associated with a set of traits encoded as a value of type

```
enum class SegmentTraits : uint32_t { };
```

Code segment type

Each code segment is associated with a “code segment type”, a value of type

```
enum class SegmentType : uint8_t { };
```

Partition name: "decl.segment".

Note: This is a vendor extension, the meta description of which is subject to change

Chapter 9

Types

Similar to declarations, types are also indicated by abstract type references. They are values of type *TypeIndex*, with 32-bit precision and the following layout

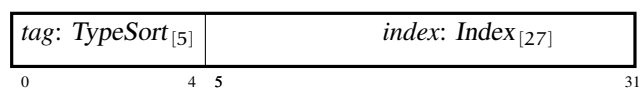


Figure 9.1: *TypeIndex*: Abstract reference of type

The type *TypeSort* is a set of 5-bit values enumerated as follows

0x00. VendorExtension	0x0B. Method
0x01. Fundamental	0x0C. Array
0x02. Designated	0x0D. Typename
0x03. Tor	0x0E. Qualified
0x04. Syntactic	0x0F. Base
0x05. Expansion	0x10. Decltype
0x06. Pointer	0x11. Placeholder
0x07. PointerToMember	0x12. Tuple
0x08. LvalueReference	0x13. Forall
0x09. RvalueReference	0x14. Unaligned
0x0A. Function	0x15. SyntaxTree

9.1 Type structures

TypeSort::VendorExtension

Partition name: "type.vendor-extension".

TypeSort::Fundamental

A *TypeIndex* abstract reference with tag *TypeSort::Fundamental* designates a fundamental type. The *index* field of that abstract reference is an index into the fundamental type partition. Each entry in that partition is a structure with the following components:

<i>basis</i> : <i>TypeBasis</i>
<i>precision</i> : <i>TypePrecision</i>
<i>sign</i> : <i>TypeSign</i>
<padding>: <i>uint8_t</i>

Figure 9.2: Structure of a fundamental type

The *basis* field designates the fundamental basis of the fundamental type. The *precision* field designates the "bit precision" variant of the basis type. The *sign* field indicates the "sign" variant of the basis type.

Partition name: "type.fundamental".

Fundamental type basis

The fundamental types are made out a small set of type basis defined as:

```
enum class TypeBasis : uint8_t {
    Void,
    Bool,
    Char,
    Wchar_t,
    Int,
    Float,
    Double,
    Nullptr,
    Ellipsis,
    SegmentType,
    Class,
    Struct,
    Union,
    Enum,
    Typename,
    Namespace,
    Interface,
    Function,
    Empty,
    VariableTemplate,
```



```

    Concept,
    Auto,
    decltypeAuto,
    Overload
};

```

with the following meaning:

- `TypeBasis::Void`: fundamental type `void`
- `TypeBasis::Bool`: fundamental type `bool`
- `TypeBasis::Char`: fundamental type `char`
- `TypeBasis::Wchar_t`: fundamental type `wchar_t`
- `TypeBasis::Int`: fundamental type `int`
- `TypeBasis::Float`: fundamental type `float`
- `TypeBasis::Double`: fundamental type `double`
- `TypeBasis::Nullptr`: fundamental type `decltype(nullptr)`
- `TypeBasis::Ellipsis`: fundamental type denoted by `...`
- `TypeBasis::SegmentType`. Note: this basis member is subject to removal in future revision.
- `TypeBasis::Class`: fundamental type `class`
- `TypeBasis::Struct`: fundamental type `struct`
- `TypeBasis::Union`: fundamental type `union`
- `TypeBasis::Enum`: fundamental type `enum`
- `TypeBasis::Typename`: fundamental concept `typename`
- `TypeBasis::Namespace`: fundamental type `namespace`
- `TypeBasis::Interface`: fundamental type `__interface`
- `TypeBasis::Function`: fundamental concept of function. Note: this basis member is object to removal in future revision.
- `TypeBasis::Empty`: fundamental type resulting from an empty pack expansion
- `TypeBasis::VariableTemplate`: concept of variable template. Note: this basis member is subject to removal in future revision.
- `TypeBasis::Concept`: fundamental type of a `concept` expression.

- `TypeBasis::Auto`: type placeholder `auto`
- `TypeBasis::DecltypeAuto` type placeholder `decltype(auto)`
- `TypeBasis::Overload`: fundamental type for an overload set.

Note: The current set of type basis is subject to change.

Fundamental type precision

The bit precision of a fundamental type is a value of type *TypePrecision* defined as follows:

```
enum class TypePrecision : uint8_t {
    Default,
    Short,
    Long,
    Bit8,
    Bit16,
    Bit32,
    Bit64,
    Bit128,
};
```

with the following meaning:

- `TypePrecision::Default`: the default precision of the basis type
- `TypePrecision::Short`: the short variant of the basis type
- `TypePrecision::Long`: the long variant of the basis type
- `TypePrecision::Bit8`: the 8-bit variant of the basis type
- `TypePrecision::Bit16`: the 16-bit variant of the basis type
- `TypePrecision::Bit32`: the 32-bit variant of the basis type
- `TypePrecision::Bit64`: the 64-bit variant of the basis type
- `TypePrecision::Bit128`: the 128-bit variant of the basis type

Fundamental type sign

The sign of a fundamental type is expressed as a value of type *TypeSign* defined as follows:

```
enum class TypeSign : uint8_t {
    Plain,
    Signed,
    Unsigned,
};
```

with the following meaning:

- `TypeSign::Plain`: the plain sign of the basis type
- `TypeSign::Signed`: the signed variant of the basis type
- `TypeSign::Unsigned`: the unsigned variant of the basis type

TypeSort::Designated

A *TypeIndex* value with tag `TypeSort::Designated` represents an abstract reference to type denoted by a declaration name. This is the typical use of a class name. The *index* field is an index into the designated type partition. Each entry in that partition is a structure with a single component: the *decl* field.

<i>decl</i> : <i>DeclIndex</i>

Figure 9.3: Structure of a designated type

The *decl* field denotes the declaration of the type.

Partition name: "type.designated".

TypeSort::Tor

A *TypeIndex* value with `TypeSort::Tor` represents an abstract reference to the type of a constructor declaration. ISO C++ does not define the type of a constructor – it even denies a constructor has a name. However, for regularity and handling of the representation of template declaration, it is simpler to assign types to constructors. The *index* field of that abstract reference is an index into the partition of tor types. Each entry in that partition is a structure with the following layout

<i>source</i> : <i>TypeIndex</i>
<i>eh_spec</i> : <i>NoexceptSpecification</i>
<i>convention</i> : <i>CallingConvention</i>

Figure 9.4: Structure of a tor type

The meaning of the fields is as follows

- *source* denotes the sequence of parameter types in the declaration of the corresponding constructor.
- *eh_spec* denotes the exception specification associated with the constructor declaration this type associates with.

- *convention* denotes the calling convention associated with the constructor declaration.

Partition name: "type.tor".

TypeSort::Syntactic

A *TypeIndex* value with tag `TypeSort::Syntactic` represents an abstract reference to type expressed at the C++ source-level as a type-id. Typical examples include a template-id designating a specialization. The *index* field is an index into the syntactic type partition. Each entry in that partition is a structure with a single component: the *expr* field.

<i>expr</i> : <i>ExprIndex</i>

Figure 9.5: Structure of a syntactic type

The *expr* field denotes the C++ source-level type expression.

Partition name: "type.syntactic".

TypeSort::Expansion

A *TypeIndex* abstract reference with tag `TypeSort::Expansion` designates an expansion of a pack type. The *index* field is an index into the expansion type partition. Each entry in that partition is a structure with the following layout

<i>pack</i> : <i>TypeIndex</i>
<i>mode</i> : <i>ExpansionMode</i>

Figure 9.6: Structure of an expansion type

The *pack* field designates the type form under expansion. The *mode* field denotes the mode of expansion.

Partition name: "type.expansion".

Pack expansion mode

During pack expansion, a template parameter can be fully expanded, or only partially expanded. The mode is indicated by a value of type

```
enum class ExpansionMode : uint8_t {
    Full,
```

```

    Partial
};

```

TypeSort::Pointer

A *TypeIndex* value with tag `TypeSort::Pointer` represents an abstract reference to a pointer type. The *index* is an index into the pointer type partition. Each entry in that partition is a structure with one component: the *pointee* field.

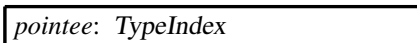


Figure 9.7: Structure of a pointer type

The *pointee* field denotes the type pointed-to: any valid C++ type.

Partition name: "type.pointer".

TypeSort::PointerToMember

A *TypeIndex* value with tag `TypeSort::PointerToMember` represents an abstract reference to a C++ source-level pointer-to-nonstatic-data member type. The *index* field is an index into the pointer-to-member type partition. Each entry in that partition is a structure with two components: the *scope* field, and the *member* field.

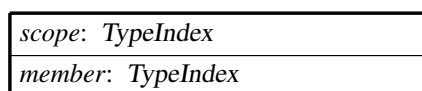


Figure 9.8: Structure of a pointer-to-member type

The *scope* field denotes the enclosing class type. The *member* field denotes the type of the member.

Partition name: "type.pointer-to-member".

TypeSort::LvalueReference

A *TypeIndex* value with tag `TypeSort::LvalueReference` represents an abstract reference to a C++ classic reference type, e.g. `int&`. The *index* field is an index into the lvalue-reference type partition. Each entry in that partition is a structure with one component: the *referee* field.

<i>referee</i> : <i>TypeIndex</i>

Figure 9.9: Structure of an lvalue-reference type

The *referee* field denotes the type referred-to: it is any valid C++ type, including function type.

Partition name: "type.lvalue-reference".

TypeSort::RvalueReference

A *TypeIndex* value with tag `TypeSort::RvalueReference` represents an abstract reference to a C++ rvalue-reference type, e.g. `int&&`. The *index* field is an index into the rvalue-reference type partition. Each entry in that partition is a structure with one component: the *referee* field.

<i>referee</i> : <i>TypeIndex</i>

Figure 9.10: Structure of an rvalue-reference type

The *referee* field denotes the type referred-to: it is any valid C++ type.

Partition name: "type.rvalue-reference".

TypeSort::Function

A *TypeIndex* with tag `TypeSort::Function` represents an abstract reference to a C++ source-level function type. The *index* field is an index into the function type partition. Each entry in that partition is a structure with the following components: a *target* field, a *source* field, an *eh_spec* field, a *convention* field, a *traits* field.

<i>target</i> : <i>TypeIndex</i>
<i>source</i> : <i>TypeIndex</i>
<i>eh_spec</i> : <i>NoexceptSpecification</i>
<i>convention</i> : <i>CallingConvention</i>
<i>traits</i> : <i>FunctionTypeTraits</i>

Figure 9.11: Structure of a function type

The *target* field denotes the return type of the function type. The *source* field denotes the parameter type list. A null *source* value indicates no parameter type. If the function type has at most one parameter type, the *source* denotes that type. Otherwise, it is a

tuple type made of all the parameter types. The *eh_spec* denotes the C++ source-level noexcept-specification. The *convention* field denotes the calling convention of the function type. The *traits* field denotes additional function type traits.

Partition name: "type.function".

Function type traits

Certain function types properties are expressed as value of bitmask type *FunctionTypeTraits* defined as:

```
enum class FunctionTypeTraits : uint8_t {
    None      = 0,
    Const     = 1 << 0,
    Volatile  = 1 << 1,
    Lvalue    = 1 << 2,
    Rvalue    = 1 << 3,
};
```

with the following meaning

- `FunctionTypeTraits::None`: the function type is that of a function that is not non-static member (either a non-member function or a static member function).
- `FunctionTypeTraits::Const`: the function type is that of a function that is a non-static member function declared with a `const` qualifier.
- `FunctionTypeTraits::Volatile`: the function type is that of a function that is a non-static member function declared with a `volatile` qualifier.
- `FunctionTypeTraits::Lvalue`: the function type is that of a function that is a non-static member function declared with an lvalue (&) qualifier.
- `FunctionTypeTraits::Rvalue`: the function type is that of a function that is a non-static member function declared with an rvalue (&&) qualifier.

Calling convention

Function calling conventions are expressed as values of type *CallingConvention* defined as:

```
enum class CallingConvention : uint8_t {
    Cdecl,
    Fast,
    Std,
    This,
    Clr,
    Vector,
    Eabi,
};
```

with the following meaning

- `CallingConvention::Cdecl`: the function is declared with `__cdecl`.
- `CallingConvention::Fast`: the function is declared with `__fastcall`.
- `CallingConvention::Std`: the function is declared with `__stdcall`.
- `CallingConvention::This`: the function is declared with `__thiscall`.
- `CallingConvention::Clr`: the function is declared with `__clrcall`.
- `CallingConvention::Vector`: the function is declared with `__vectorcall`.
- `CallingConvention::Eabi`: the function is declared with `__eabi`.

Note: The calling convention currently details only MSVC.

Noexcept specification

The noexcept-specification of a function is described by a structure with the following components:

<i>words</i> : <i>SentenceIndex</i>
<i>sort</i> : <i>NoexceptSort</i>

Figure 9.12: Structure of a noexcept-specification

The *words* field is meaningful only for function templates and complex noexcept-specification. The *sort* describes the sort of noexcept-specification.

TypeSort::Method

A *TypeIndex* with tag `TypeSort::Method` represents an abstract reference to a C++ source-level non-static member function type. The *index* field is an index into the method type partition. Each entry in that partition is a structure with the following components: a *target* field, a *source* field, a *scope* field, an *eh_spec* field, a *convention* field, a *traits* field.

<i>target</i> : <i>TypeIndex</i>
<i>source</i> : <i>TypeIndex</i>
<i>scope</i> : <i>TypeIndex</i>
<i>eh_spec</i> : <i>NoexceptSpecification</i>
<i>convention</i> : <i>CallingConvention</i>
<i>traits</i> : <i>FunctionTypeTraits</i>

Figure 9.13: Structure of a method type

The *target* field denotes the return type of the function type. The *source* field denotes the parameter type list. The function type has at most one parameter type, the *source* denotes that type. Otherwise, it is a tuple type made of all the parameter types. The *scope* denotes the C++ source-level enclosing class type. The *eh_spec* denotes the C++ source-level noexcept-specification. The *convention* field denotes the calling convention of the function type. The *traits* field denotes additional function type traits.

Partition name: "type.nonstatic-member-function".

TypeSort::Array

A *TypeIndex* value with tag `TypeSort::Array` represents an abstract reference to a C++ source-level builtin array type. The *index* field is an index into the array type partition. Each entry in that partition is a structure with two components: a *element* field, and a *extent* field.

<i>element</i> : <i>TypeIndex</i>
<i>extent</i> : <i>ExprIndex</i>

Figure 9.14: Structure of an array type

The *element* denotes the element type of the array. The *extent* denotes the number of elements in the array type along its outer most dimension.

Partition name: "type.array".

TypeSort::Typename

A *TypeIndex* value with tag `TypeSort::Typename` represents an abstract reference to a dependent type which at the C++ source-level is written as type expression. The *index* field is an index into the typename type partition. Each entry in that partition is a structure with a single component: the *path* field.

<i>path</i> : <i>ExprIndex</i>

Figure 9.15: Structure of a typename type

The *path* field denotes the expression designating the dependent type.

Partition name: "type.typename".

TypeSort::Qualified

A *TypeIndex* value with tag `TypeSort::Qualified` represents an abstract reference to a C++ source-level cv-qualified type. The *index* field is an index into the qualified type partition. Each entry in that partition is a structure with two components: the *unqualified* field, and the *qualifiers* field.

<i>unqualified</i> : <i>TypeIndex</i>
<i>qualifiers</i> : <i>Qualifiers</i>

Figure 9.16: Structure of a qualified type

The *unqualified* field denotes the type without the toplevel cv-qualifiers. The *qualifiers* field denotes the cv-qualifiers.

Partition name: "type.qualified".

Type qualifiers

Standard type qualifiers are given values of type:

```
enum class Qualifier : uint8_t {
    None      = 0,
    Const     = 1 << 0,
    Volatile  = 1 << 1,
    Restrict  = 1 << 2,
};
```

with the following meaning:

- `Qualifiers::None`: No type qualifier
- `Qualifiers::Const`: the type is `const`-qualified
- `Qualifiers::Volatile`: the type is `volatile`-qualified.
- `Qualifiers::Restrict`: the type is `__restrict`-qualified – non-standard, but common extension.

TypeSort::Base

A *TypeIndex* value with tag `TypeSort::Base` represents an abstract reference to a use of a type as a base-class (in a class inheritance) at the C++ source-level. The *index* field is an index into the base type partition. Each entry in that partition is a structure with the following layout

<i>type</i> : <i>TypeIndex</i>
<i>access</i> : <i>Access</i>
<i>specifiers</i> : <i>BaseTypeSpecifiers</i>

Figure 9.17: Structure of a base type

The meanings of the fields are as follows:

- *type* denotes the type base being used as a base type.
- *access* denotes the access specifier in the inheritance path.
- *specifiers* indicates specifiers for the base type (§9.1).

Partition name: "type.base".

Note: This representation is subject to change

Base type specifiers

Base type specifiers are bitmask values of type

```
enum class BaseTypeSpecifiers : uint8_t {
    None      = 0,
    Shared    = 0x01,
    Expanded  = 0x02,
};
```

with the following meaning:

- `BaseTypeSpecifiers::None`: no base type specifiers.
- `BaseTypeSpecifiers::Shared`: the base class is virtual.
- `BaseTypeSpecifiers::Expanded`: the base class is a pack expansion.

Note: Since type expansions are type designators, a base type that is a pack expansion should be designated by `TypeSort::Expansion` (§9.1), not a bitmask value.

TypeSort::Decltype

A *TypeIndex* value with tag `TypeSort::Decltype` represents an abstract reference to the C++ source-level application of `decltype` to an expression. Ideally, the operand should be represented by an *ExprIndex* value. However, in the current implementation the operand is represented as an arbitrary sequence of tokens. The *index* field is an index into the `decltype` type partition. Each entry in that partition is a structure with a single component: the *words* field.

<i>expr</i> : <i>SyntaxIndex</i>

Figure 9.18: Structure of an decltype type

The *expr* field denotes the syntactic representation of the expression operand to decltype.

Partition name: "type.decltype".

Note: The decltype representation will change in the future to change its operand representation to an expression tree.

TypeSort::Placeholder

A *TypeIndex* abstract reference with tag `TypeSort::Placeholder` designates a placeholder type. The *index* field is an index into the placeholder type partition. Each entry in that partition is a structure with the following layout

<i>constraint</i> : <i>ExprIndex</i>
<i>basis</i> : <i>TypeBasis</i>
<i>elaboration</i> : <i>TypeIndex</i>

Figure 9.19: Structure of a placeholder type

The *constraint* field designates the (generalized) predicate that the placeholder shall satisfy – at the input source level. The *basis* field is a structure denoting the type basis (§9.1) pattern expected for placeholder type, either `auto` or `decltype(auto)`. The *elaboration* field, if non null, designates the deduced type corresponding to the placeholder type.

Partition name: "type.placeholder".

TypeSort::Tuple

A *TypeIndex* value with tag `TypeSort::Tuple` represents an abstract reference to tuple type, i.e. finite sequence of types. The *index* field is an index into the tuple type partition. Each entry in that partition is a structure with two components: a *start* field, and a *cardinality* field.

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 9.20: Structure of a tuple type

The *start* field is an index into the type heap partition. It designates the first element in the tuple. The *cardinality* field denotes the number of elements in the tuple.

Partition name: "type.tuple".

TypeSort::Forall

A *TypeIndex* value with tag `TypeSort::Forall` designates a generalized \forall -type, as can be ascribed to a template declaration, even though a template declaration does not have a type in ISO C++. The *index* field is an index into the forall type partition. Each entry in that partition is a structure with the following layout

<i>chart</i> : <i>ChartIndex</i>
<i>subject</i> : <i>TypeIndex</i>

Figure 9.21: Structure of a forall type

The *chart* field designates the set of parameter lists in the template declaration. The *subject* field designated the type of the current instantiation.

Partition name: "type.forall".

Note: This generalized type is not yet used in current version of MSVC. However, they will be used in future releases of MSVC.

TypeSort::Unaligned

A *TypeIndex* value with tag `TypeSort::Unaligned` represents an abstract reference to a type with `__unaligned` specifier. The *index* field is an index into the unaligned type partition. Each entry in that partition is a structure with a single component: the *type* field.

<i>type</i> : <i>TypeIndex</i>

Figure 9.22: Structure of an unaligned type

Partition name: "type.unaligned".

Note: An `__unaligned` type is an MSVC extension, with no clearly defined semantics. This representation should be expressed in a more regular framework of vendor-extension types.

TypeSort::SyntaxTree

A *TypeIndex* value with tag `TypeSort::SyntaxTree` represents an abstract reference to a type designated by a raw syntax construct (§15). The *index* field is an index into the syntax tree type partition. Each entry in that partition is a structure with a single component: the *syntax* field:

<i>syntax</i> : <i>SyntaxIndex</i>

Figure 9.23: Structure of a syntax tree type

Partition name: `"type.syntax-tree"`.

Chapter 10

Expressions

Expressions are indicated by abstract expression references. They are values of type *ExprIndex*, with 32-bit precision and the following layout

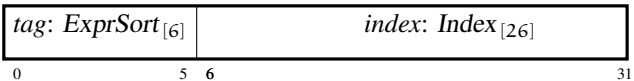


Figure 10.1: *ExprIndex*: Abstract reference of expression

The type *ExprSort* is a set of 6-bit values enumerated as follows

- | | |
|-------------------------------|--------------------------------|
| 0x00. VendorExtension | 0x10. Triad |
| 0x01. Empty | 0x11. String |
| 0x02. Literal | 0x12. Temporary |
| 0x03. Lambda | 0x13. Call |
| 0x04. Type | 0x14. MemberInitializer |
| 0x05. NamedDecl | 0x15. MemberAccess |
| 0x06. UnresolvedId | 0x16. InheritancePath |
| 0x07. TemplateId | 0x17. InitializerList |
| 0x08. UnqualifiedId | 0x18. Cast |
| 0x09. SimpleIdentifier | 0x19. Condition |
| 0x0A. Pointer | 0x1A. ExpressionList |
| 0x0B. QualifiedName | 0x1B. SizeofType |
| 0x0C. Path | 0x1C. Alignof |
| 0x0D. Read | 0x1D. New |
| 0x0E. Monad | 0x1E. Delete |
| 0x0F. Dyad | 0x1F. Typeid |
| | 0x20. DestructorCall |
| | 0x21. SyntaxTree |

0x22. FunctionString	0x30. Placeholder
0x23. CompoundString	0x31. Expansion
0x24. StringSequence	0x32. Generic
0x25. Initializer	0x33. Tuple
0x26. Requires	0x34. Nullptr
0x27. UnaryFold	0x35. This
0x28. BinaryFold	0x36. TemplateReference
0x29. HierarchyConversion	0x37. PushState
0x2A. ProductTypeValue	0x38. TypeTraitIntrinsic
0x2B. SumTypeValue	0x39. DesignatedInitializer
0x2C. SubobjectValue	0x3A. PackedTemplateArguments
0x2D. ArrayValue	0x3B. Tokens
0x2E. DynamicDispatch	0x3C. AssignInitializer
0x2F. VirtualFunctionConversion	

10.1 Expression structures

ExprSort::VendorExtension

Partition name: "expr.vendor-extension".

ExprtSort::Empty

In certain circumstances, expressions are expected but missing, e.g. an empty-pack in a template-argument list. An *ExprIndex* value with tag `ExprSort::Empty` represents a reference to an empty expression. The *index* field is an index into the empty expression partition. Each entry in that partition has two components: a *type* field and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>

Figure 10.2: Structure of an empty expression

The *type* field is a reference to the type of the expression. The *locus* field is a reference to the source location.

Partition name: "expr.empty".

ExprSort::Literal

A *ExprIndex* value with tag `ExprSort::Literal` represents a reference to a literal. The *index* field is an index into the literal expression partition. Each entry in that partition has three components: a *type* field, a *value* field, and a *locus* field.

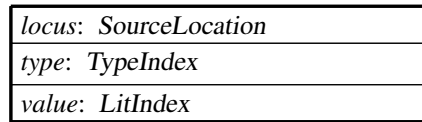


Figure 10.3: Structure of a literal expression

The *type* field represents the type of the expression. The *value* field represents the value of the expression. The *locus* represents the source location of the expression.

Partition name: "expr.literal".

Literal Values

Literal values are represented by abstract references of type *LitIndex*, with the following layout

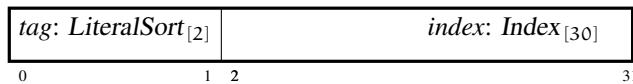


Figure 10.4: *LitIndex*: Abstract reference of literal constants

The possible values of the *tag* field are described by the type *LiteralSort* defined as follows

```
enum class LiteralSort : uint8_t {
    Immediate,
    Integer,
    FloatingPoint,
};
```

The meaning of these tags is as follows:

- `LiteralSort::Immediate`: The *value* field of the abstract reference directly holds a 32-bit unsigned integer value.
- `LiteralSort::Integer`: The *value* field is an index into the "const.i64" partition. The value at that entry is a 64-bit unsigned integer value.
- `LiteralSort::FloatingPoint`: The *value* field is an index into the "const.f64" partition. Each entry in that partition is a 12-byte structure: The first 8 bytes represent a 64-bit floating point value, in IEEE 754 little endian format. The remaining 4 bytes have indeterminate values.

Note: This representation is subject to change in future releases.

ExprSort::Lambda

An *ExprIndex* value with tag `ExprSort::Lambda` designates a lambda expression in syntactic form (§15). The *index* field of that abstract reference is an index into the lambda expression partition. Each entry of that partition is a structure with the following structure:

<i>introducer</i> :	<i>SyntaxIndex</i>
<i>template_parameters</i> :	<i>SyntaxIndex</i>
<i>declarator</i> :	<i>SyntaxIndex</i>
<i>constraint</i> :	<i>SyntaxIndex</i>
<i>body</i> :	<i>SyntaxIndex</i>

Figure 10.5: Structure of a lambda expression

The *introducer* field designates the syntactic element for the lambda introducer. The *template_parameters* field designates the syntactic element for any possible template parameter list. The *declarator* field designates the syntactic element for the declarator part of the lambda expression. The *constraint* field designates the syntactic element for the requires-clause, if any. Finally, the *body* field denotes the syntactic element for the body of the lambda expression.

Partition name: `"expr.lambda"`.

Note: This representation is subject to change

ExprSort::Type

Certain C++ source-level contexts permit both value expressions and type expressions. A *ExprIndex* value with tag `ExprSort::Type` represents a reference to a type expression. The *index* field is an index into the type expression partition. Each entry in that partition has two components: a *type* field, and a *locus* field.

<i>locus</i> :	<i>SourceLocation</i>
<i>type</i> :	<i>TypeIndex</i>
<i>denotation</i> :	<i>TypeIndex</i>

Figure 10.6: Structure of a literal expression

The *denotation* field is a reference to the type designated by this expression structure. The *locus* field is the source location. The *type* field designates the sort of type, typically `TypeBasis::Type`.

Partition name: `"expr.type"`.

ExprSort::NamedDecl

A *ExprIndex* value with tag `ExprSort::NamedDecl` denotes the use of a name of a declaration as an expression. The *index* field is an index into the named declaration expression partition. Each entry in that partition has three components: a *type* field, a *resolution* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>resolution</i> : <i>DeclIndex</i>

Figure 10.7: Structure of use of named declaration expression

The *type* field denotes the type of the expression. The *resolution* field denotes the declaration the name resolved to, e.g. as indicated by the appropriate language rules. The *locus* denotes the source location.

Partition name: `"expr.decl"`.

ExprSort::UnresolvedId

A *ExprIndex* value with tag `ExprSort::UnresolvedId` represents a C++ source-level dependent name, or an unresolved name. The *index* field is an index into the unresolved name expression partition. Each entry in that partition has two components: a *name* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>name</i> : <i>NameIndex</i>

Figure 10.8: Structure of an unresolved name expression

The *name* field denotes the name. The *locus* field denotes the source location.

Partition name: `"expr.unresolved"`.

Note: This structure is subject to removal in future releases.

ExprSort::TemplateId

A *ExprIndex* value with tag `ExprSort::TemplateId` represents a reference to a template-id. The *index* field is an index into the template-id expression partition. Each entry in that partition is a structure with three components: a *primary* field, an *arguments* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>primary</i> : <i>ExprIndex</i>
<i>arguments</i> : <i>ExprIndex</i>

Figure 10.9: Structure of a template-id expression

The *primary* field denotes the primary template. The *arguments* field denotes the template-argument list. If that list is empty or is a singleton, *arguments* denotes that template-argument directly. Otherwise, it denotes a tuple expression. The *locus* denotes the source location.

Partition name: `"expr.template-id"`.

ExprSort::UnqualifiedId

A *ExprIndex* value with tag `ExprSort::UnqualifiedId` denotes the C++ grammar term *unqualified-id* or a component of *qualified-id* expression (`ExprSort::QualifiedIdName`), some of which might not be bound to any declaration, or might be assumed to name a template by fiat (when preceded by the `template` keyword). The *index* field is an index into the identifier expression partition. Each entry in that partition is a structure with the following layout:

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>name</i> : <i>NameIndex</i>
<i>resolution</i> : <i>ExprIndex</i>
<i>template_keyword</i> : <i>SourceLocation</i>

Figure 10.10: Structure of an identifier used to form an expression

The *name* denotes the name of that component in the *qualified-id* expression. The *resolution* denotes the (possible set of) declaration the name refers to, if any. The *template_keyword*, if valid, designate the source location position of the `template` keyword. The *locus* denotes the source location of this expression.

Partition name: "expr.unqualified-id".

Note: This structure is subject to change in future releases.

ExprSort::SimpleIdentifier

A *ExprIndex* value with tag `ExprSort::SimpleIdentifier` denotes the use of a simple name in a templated code. Typically, these identifiers are used as label names. The *index* field is an index into the simple identifier partition. Each entry in that partition is a structure with the following layout:

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>name</i> : <i>NameIndex</i>

Figure 10.11: Structure of a simple-identifier expression

The *locus* field denotes the location of this expression. The *type*, if not null, denotes the type of the expression. The *name* field designates the simple identifier.

Partition name: "expr.simple-identifier".

Note: This structure is subject to removal in future releases.

ExprSort::Pointer

An *ExprIndex* value with tag `ExprSort::Pointer` designates the use of `*` in a *ptr-declarator* as part of the a *declarator*. It is *not* an expression. The *index* field of that abstract reference designates an entry with the following layout

<i>locus</i> : <i>SourceLocation</i>

The *locus* designates the source location of the `*` in the input source code.

Partition name: "expr.pointer".

Note: This sort best belongs to the syntax tree hierarchy and has no bearing to expressions. It is scheduled for removal in future releases of MSVC.

ExprSort::QualifiedName

An *ExprIndex* value with tag `ExprSort::QualifiedName` designates a representation of C++ input source level grammar term *qualified-id*, typically in templated code that is not yet fully semantically analyzed. The *index* field of that abstract reference is an index into the qualified name partition. Each entry of that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>elements</i> : <i>ExprIndex</i>
<i>typename_keyword</i> : <i>SourceLocation</i>

Figure 10.12: Structure of a qualified name expression

The *elements* field designates the sequence of unqualified names (`ExprSort::UnqualifiedId`) in the source level construct. The *template_keyword*, if not null, designates the presence of the `typename` keyword in the input source program to indicate to the parser that the qualified name actually names a type.

Partition name: "expr.qualified-name".

Note: This structure is subject to change in future releases.

ExprSort::Path

A *ExprIndex* value with tag `ExprSort::Path` represents a reference to a qualified-id expression. The *index* field is an index into the path expression partition. Each entry in that partition has three components: a *scope* field, a *member* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>scope</i> : <i>ExprIndex</i>
<i>member</i> : <i>ExprIndex</i>

Figure 10.13: Structure of a path expression

The *scope* field denotes the qualifying part of the expression. The *member* field denotes the referenced member of the *scope*. The *locus* field denotes the source location.

Partition name: "expr.path".

ExprSort::Read

A *ExprIndex* value with tag `ExprSort::Read` represents a reference to an expression that reads from a given memory location. It is also used to represent so-called lvalue-to-rvalue conversions. The *index* field is an index into the read expression partition. Each entry in that partition has three components: a *type* field, an *address* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>address</i> : <i>ExprIndex</i>
<i>sort</i> : <i>ReadConversionSort</i>

Figure 10.14: Structure of a read expression

The *type* field denotes the type of the expression. The *address* field denotes the memory location. The *locus* denotes the source location. The *sort* denotes the sort of conversion performed by this expression.

Partition name: `"expr.read"`.

Note: This structure is subject to change in future releases.

ReadConversionSort

An expression that reads from a memory location can also perform an implicit conversion either on the operand, or on the result of the read. The sort of conversion performed is indicated by a value type *ReadConversionSort* defined as

```
enum class ReadConversionSort : uint8_t {
    Identity,
    Indirection,
    Dereference,
    LvalueToRvalue,
    IntegralConversion,
};
```

with the following semantics

- `ReadConversionSort::Identity`: No special interpretation or conversion applied.
- `ReadConversionSort::Indirection`: This is an indirection via a pointer, or via the name of an entity treated as a pointer.
- `ReadConversionSort::Dereference`: The operand is a reference, and the result of the read is the address of the entity referred to.

- `ReadConversionSort::LvalueToRvalue`: This expression represents an lvalue-to-rvalue conversion
- `ReadConversionSort::IntegralConversion`: The result of the read operation is followed by an integral conversion.

ExprSort::Monad

A *ExprIndex* value with tag `ExprSort::Monad` represents the application of a (source-level) monadic operator to argument. The *index* field is an index into the monadic expression partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>impl</i> : <i>DeclIndex</i>
<i>argument</i> : <i>ExprIndex</i>
<i>assoc</i> : <i>MonadicOperator</i>

Figure 10.15: Structure of a monadic expression

and meanings of the fields:

- *locus* denotes the source location.
- *type* denotes the type of the expression.
- *impl*, when non-null, designates the semantic resolution of the operator denoted by *assoc* if it is user-defined. This semantic resolution can be an overload set (i.e. `DeclSort::Tuple`) if the expression appears in a templated code, accounting for the set of declarations found by applicable name lookup in the template definition context.
- *argument* denotes the argument to the operator.
- *assoc* field denotes the conceptual monadic operation, usually as written in the input C++ source code, §10.2.

Partition name: "expr.monad".

ExprSort::Dyad

A *ExprIndex* value with tag `ExprSort::Dyad` represents the application of a (source-level) dyadic operator (§10.2) to arguments. The *index* field is an index into the dyadic expression partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>impl</i> : <i>DeclIndex</i>
<i>arguments</i> : <i>ExprIndex</i> [2]
<i>assoc</i> : <i>DyadicOperator</i>

Figure 10.16: Structure of a dyadic expression

and meanings of the fields:

- The *locus* denotes the source location.
- The *type* field denotes the type of the expression.
- The *impl* field, when non-null, designates the semantic resolution of the operator denoted by *assoc* if it is user-defined. This semantic resolution can be an overload set (*DeclSort::Tuple*) if the expression appears in a templated code, accounting for the set of declarations found by applicable name lookup in the template definition context.
- The *arguments* field denotes the two arguments to the operator.
- The *assoc* field denotes the conceptual dyadic operation, usually as written in the input C++ source code (§10.2).

Partition name: "expr.dyad".

ExprSort::Triad

A *ExprIndex* value with tag *ExprSort::Monad* represents the application of a (source-level) triadic operator (§10.2) to arguments. The *index* field is an index into the triadic expression partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>impl</i> : <i>DeclIndex</i>
<i>arguments</i> : <i>ExprIndex</i> [3]
<i>assoc</i> : <i>TriadicOperator</i>

Figure 10.17: Structure of a triadic expression

and meaning of the fields:

- *locus* denotes the source location.
- *type* denotes the type of the expression.
- *impl*, when non-null, designates the semantic resolution of the operator denoted by *assoc* if it is user-defined. This semantic resolution can be an overload set (`DeclSort::Tuple`) if the expression appears in a templated code, accounting for the set of declarations found by applicable name lookup in the template definition context.
- *arguments* denotes the three arguments to the operator.
- *assoc* denotes the conceptual triadic operation, usually as written in the input C++ source code (§10.2).

Partition name: `"expr.triad"`.

ExprSort::String

A *ExprIndex* value with tag `ExprSort::String` represents a reference to a string literal expression. The *index* field is an index into the string literal partition (not to be confused with the string table). Each entry of that partition is a structure with the following components: a *string_index* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>string_index</i> : <i>StringIndex</i>

Figure 10.18: Structure of a string literal expression

The *string_index* field is an index into the partition of the representations of string literals. The *locus* denotes the source location.

The interpretation of each string in that table is given by the abstract reference, a *StringIndex*, used to index into the string table. A *StringIndex* value, like any abstract reference, is a 32-bit value:

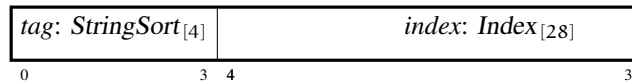


Figure 10.19: *StringIndex*: Abstract reference of string constant

In its current implementation, the tag of the *StringIndex* is given by the following declaration

```
enum class StringSort : uint8_t {  
    Ordinary,  
    UTF8,  
    Char16,  
    Char32,  
    Wide,  
};
```

The string table is always present, and non-empty. It is an array of bytes, the content of which is interpreted according to the abstract reference used to index it. The first entry is the NUL byte, therefore a *StringIndex* with 0 *tag* and 0 *index* represents the empty string.

Partition name: "expr.strings".

StringSort::Ordinary

A *StringIndex* with tag `StringSort::Ordinary` represents an ordinary, narrow NUL-terminated string constant. The *value* field is an index into the string table, pointing to the first byte of the string.

StringSort::UTF8

A *StringIndex* with tag `StringSort::UTF8` represents a UTF-8 narrow NUL-terminated string constant. In terms of C++ source-level construct, it represents a string constant with `u8` prefix. The *value* field is an index into the string table, pointing to the first byte of the string.

StringSort::Char16

A *StringIndex* with tag `StringSort::Char16` represents a `char16_t` string constant with `u` prefix. The *value* field is an index into the string table, pointing to the first byte of the string.

StringSort::Char32

A *StringIndex* with tag `StringSort::Char32` represents a `char32_t` string constant with `u` prefix. The *value* field is an index into the string table, pointing to the first byte of the string.

StringSort::Wide

A *StringIndex* with tag `StringSort::Wide` represents wide string constant with only the `L` prefix. The *value* field is an index into the string table, pointing to the first byte of the string.

String literal structure

Each entry of the partition for string literal representation is a structure with the following components:

<i>start</i> : <i>TextOffset</i>
<i>length</i> : <i>Cardinality</i>
<i>suffix</i> : <i>TextOffset</i>

Figure 10.20: Structure of a string literal

The *start* field is an index into the string table, representing the start of the string. The *length* field denotes the number of bytes taken up by the string, not counting suffix, if any. The *suffix* field is an index into the string table denoting the suffix, if any, of the string literal.

Partition name: `"const.str"`.

ExprSort::Temporary

A *ExprIndex* value with tag `ExprSort::Temporary` represents a reference to a C++ source-level expression designating a temporary object. The *index* field is an index into the temporary expression partition. Each entry in that partition is a structure with the following components: a *type* field, an *id* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>id</i> : <i>UniqueID</i>

Figure 10.21: Structure of a temporary object expression

The *type* field denotes the type of the expression. The *id* field denotes a unique identification of the temporary object. Its value is of type

```
enum class UniqueID : uint32_t { };
```

Figure 10.22: Definition of type *UniqueID*

The *locus* field denotes the source location.

Partition name: `"expr.temporary"`.

ExprSort::Call

A *ExprIndex* value with tag **ExprSort::Call** represents a reference to a call expression. The *index* field is an index into the call expression partition. Each entry in that partition is a structure with five components: a *type* field, an *operation* field, an *arguments* field, a *locus* field, and an *opcat* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>operation</i> : <i>ExprIndex</i>
<i>arguments</i> : <i>ExprIndex</i>

Figure 10.23: Structure of a call expression

The *type* field denotes the type of the expression. Usually, it the return type of the function. The *operation* field denotes the expression being invoked. The *arguments* field denotes the list of arguments to supplied. If that list is empty or a singleton, *arguments* directly denotes that expression. Otherwise it denotes a type expression. The *locus* field denotes the source location.

Partition name: "expr.call".

ExprSort::MemberInitializer

An *ExprIndex* value with tag **ExprSort::MemberInitializer** designates the initialization of a base-class subobject, or a non-static data member, or a call to a delegated constructor. The *index* field of that abstract reference is an index into the member initializer partition. Each entry in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>member</i> : <i>DeclIndex</i>
<i>base</i> : <i>TypeIndex</i>
<i>initializer</i> : <i>ExprIndex</i>

Figure 10.24: Structure of a member-initializer expression

The *locus* field designates the source location of the member initialization. The *type* field denotes the type of the initialization. The *member* field, if not null, designates the non-static data member being initialized. The *base* field, if not null, designates the initialization of a base-class subobject. Both fields *member* and *base* cannot be simultaneously non-null. However, they can be simultaneously null; in that case, the

expression is used to represent a call to a delegated constructor. The *initializer* field denotes the expression performing the initialization.

Partition name: "expr.member-initializer".

Note: This structure is subject to change in future releases.

ExprSort::MemberAccess

An *ExprIndex* value with tag `ExprSort::MemberAccess` designates an access to a non-static data member of an object. The access is expressed in terms of base address (of the object) and an offset (in bytes) to the member. The *index* field of the abstract reference is an index into the member access partition. Each entry of that partition has the following layout:

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>offset</i> : <i>ExprIndex</i>
<i>enclosing</i> : <i>TypeIndex</i>
<i>name</i> : <i>TextOffset</i>

Figure 10.25: Structure of a member access expression

The *locus* field designates the source location of this expression. The *type* field denotes the type of the member access expression. The *offset* is an expression designating the byte offset of the non-static data member from the starting address of its enclosing object. The member being accessed is indirectly described by its *enclosing* class type, and its *name*.

Partition name: "expr.member-access".

Note: This representation is subject to change in future releases.

ExprSort::InheritancePath

An *ExprIndex* value with tag `ExprSort::InheritancePath` designates an expression “path” to a base-class subobject. The *index* field of that abstract reference is an index into the inheritance path partition. Each entry of that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>path</i> : <i>ExprIndex</i>

Figure 10.26: Structure of an inheritance path expression

The *locus* field designates the source location of this expression. The *type* field denotes the type of the expression. the *path* is a sequence of expressions designating each base-class component of the path.

Partition name: "expr.inheritance-path".

ExprSort::InitializerList

An *ExprIndex* value with tag `ExprSort::InitializerList` designates a brace-enclosed comma-separated sequence of expressions. The *index* field is an index into the partition of initializer list expressions. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>elements</i> : <i>ExprIndex</i>

Figure 10.27: Structure of an initializer-list expression

The *elements* field denotes the sequence of expressions enclosed in the brace delimiters.

Partition name: "expr.initializer-list".

ExprSort::Cast

An *ExprIndex* value with tag `ExprSort::Cast` designates a conversion operation. The conversion may be explicit (in the input source code) and implicit (as required by semantics analysis). The *index* field of this abstract reference is a position into the partition of cast expressions. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>source</i> : <i>ExprIndex</i>
<i>target</i> : <i>Index</i>
<i>operator</i> : <i>DyadicOperator</i>

Figure 10.28: Structure of a cast expression

The *source* field denotes the operand expression, whereas the *target* field denotes the type to convert that expression to. The *operator* designates the sort of conversion operation to perform.

Partition name: "expr.cast".

ExprSort::Condition

An *ExprIndex* value with tag `ExprSort::Condition` designates a syntactic representation of an expression used as guarding predicate of an `if`-statement in a templated code. The *index* field of this abstract reference is a position designating an entry of the partition of syntactic condition expressions. Each such an entry is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>expr</i> : <i>ExprIndex</i>

Figure 10.29: Structure of a condition expression

The *expr* field denotes the expression wrapped in this condition structure.

Partition name: "expr.condition".

Note: This structure is scheduled for removal in future releases.

ExprSort::ExpressionList

An *ExprIndex* value with tag `ExprSort::ExpressionList` designates a syntactic representation of a comma-separated sequence of expressions enclosed in a pair of matching delimiters, in a templated code. The *index* field of this abstract reference is a position designating an entry of the partition of syntactic expression-list expressions. Each such entry is a structure with the following layout

<i>left</i> : <i>SourceLocation</i>
<i>right</i> : <i>SourceLocation</i>
<i>contents</i> : <i>ExprIndex</i>
<i>delimiter</i> : <i>DelimiterSort</i>

Figure 10.30: Structure of an expression list

The *left* field represents the source location of the opening delimiter, whereas the *right* field represents the source location of the closing delimiter. The *delimiter* field is of type

```
enum class DelimiterSort : uint8_t {
    Unknown = 0,
    Brace = 1,
    Parenthesis = 2,
};
```

and denotes the sort of delimiter:

- `DelimiterSort::Brace` for matching brace delimiters
- `DelimiterSort::Parenthesis` for matching parenthesis delimiters

Partition name: `"expr.expression-list"`.

Note: This structure is scheduled for removal in future releases.

ExprSort::SizeofType

An *ExprIndex* value with tag `ExprSort::SizeofType` designates a syntactic representation of a `sizeof`-expression where the operand is a type. The *index* field of this abstract reference is a position designating an entry of the partition of syntactic `sizeof`-expressions. Each such entry is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>operand</i> : <i>TypeIndex</i>

Figure 10.31: Structure of `sizeof` expression

The *operand* field designates the type operand to the `sizeof` operator in that expression. The *type* field designates the type of the entire `sizeof`-expression.

Partition name: `"expr.sizeof-type"`.

Note: This structure is subject to removal in future releases.

ExprSort::Alignof

An *ExprIndex* value with tag `ExprSort::Alignof` designates a syntactic representation of a `alignof`-expression. The *index* field of this abstract reference is a position designating an entry of the partition of syntactic `alignof`-expressions. Each such entry is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>operand</i> : <i>TypeIndex</i>

Figure 10.32: Structure of `alignof` expression

The *operand* field designates the operand to the `alignof` operator in that expression. The *type* field designates the type of the entire `alignof`-expression.

Partition name: `"expr.alignof"`.

Note: This representation is subject to change.

ExprSort::New

This structure is no longer emitted. *new-expressions* are now emitted as either monadic, dyadic, or triadic trees as appropriate.

Note: This sort value is available for reuse in future releases.

ExprSort::Delete

This structure is no longer emitted. *delete-expressions* are now emitted with appropriate structures.

Note: This sort value is available for reuse in future releases.

ExprSort::Typeid

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>operand</i> : <i>TypeIndex</i>

Figure 10.33: Structure of a `typeid` expression

The field *operand* designates the type operand to the *typeid*-expression. The field *type* designates the type of the entire expression.

Partition name: "expr.typeid".

ExprSort::DestructorCall

An *ExprIndex* value with tag *ExprSort::DestructorCall* represents an abstract reference to a (pseudo-)destructor call. The *index* field denotes a position of the corresponding entry in the destructor call partition. Each entry of that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>object</i> : <i>ExprIndex</i>
<i>decltype_specifier</i> : <i>SyntaxIndex</i>
<i>cleanup</i> : <i>DestructorSort</i>

Figure 10.34: Structure of a destructor call

with these meanings of the fields:

- *locus* denotes the source location of this expression
- *type* denotes the type of this expression
- *object*, if non-null, denotes the object for which the destructor is called
- *decltype_specifier*, if non-null, denotes the type providing the destructor function
- *cleanup* denotes the sort of destructor, the interpretation of which is determined by this table
 - 0: the sort of destructor is unknown. This should never occur in a valid IFC file.
 - 1: a standard C++ destructor.
 - 2: a CLI/.NET finalizer.

and

```
enum class DestructorSort : uint8_t { };
```

Figure 10.35: Definition of type *DestructorSort*

Partition name: "expr.destructor-call".

Note: This structure is subject to removal in future releases.

ExprSort::SyntaxTree

<i>syntax: SyntaxIndex</i>

Figure 10.36: Structure of a syntactic expression

Partition name: "expr.syntax-tree".

ExprSort::FunctionString

<i>locus: SourceLocation</i>
<i>type: TypeIndex</i>
<i>macro: TextOffset</i>

Figure 10.37: Structure of a function string expression

Partition name: "expr.function-string".

ExprSort::CompoundString

<i>locus: SourceLocation</i>
<i>type: TypeIndex</i>
<i>prefix: TextOffset</i>
<i>string: ExprIndex</i>

Figure 10.38: Structure of a compound string expression

Partition name: "expr.compound-string".

ExprSort::StringSequence

<i>locus: SourceLocation</i>
<i>type: TypeIndex</i>
<i>strings: ExprIndex</i>

Figure 10.39: Structure of a string sequence expression

Partition name: "expr.string-sequence".

ExprSort::Initializer

<i>locus</i> : SourceLocation
<i>type</i> : TypeIndex
<i>expr</i> : ExprIndex
<i>sort</i> : InitializerSort

Partition name: "expr.requires".

ExprSort::UnaryFold

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>expr</i> : <i>ExprIndex</i>
<i>operation</i> : <i>DyadicOperator</i>
<i>associativity</i> : <i>Associativity</i>

Figure 10.43: Structure of a unary fold expression

with

```
enum class Associativity : uint8_t { };
```

Figure 10.44: Definition of type *Associativity*

Partition name: "expr.unary-fold".

ExprSort::BinaryFold

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>left</i> : <i>ExprIndex</i>
<i>right</i> : <i>ExprIndex</i>
<i>operation</i> : <i>DyadicOperator</i>
<i>associativity</i> : <i>Associativity</i>

Figure 10.45: Structure of a binary fold expression

Partition name: "expr.binary-fold".

ExprSort::HierarchyConversion

An *ExprIndex* value with tag *ExprSort::HierarchyConversion* represents an expression that performs a class hierarchy conversion, i.e. class hierarchy navigation. The *index* field is an index into the hierarchy conversion partition. Each entry in that partition is a structure with the following fields

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>source</i> : <i>ExprIndex</i>
<i>target</i> : <i>TypeIndex</i>
<i>inheritance</i> : <i>ExprIndex</i>
<i>override</i> : <i>ExprIndex</i>
<i>operator</i> : <i>DyadicOperator</i>

Figure 10.46: Structure of a hierarchy conversion expression

The *type* field designates the type of the overall expression. The field *source* designates the operand expression. The field *locus* designates the location of the expression.

Partition name: "expr.hierarchy-conversion".

ExprSort::ProductTypeValue

An *ExprIndex* value with tag `ExprSort::ProductTypeValue` designates an object of class type, usually produced as part of a compile-time evaluation. The *index* field of that abstract reference is an index into the product type value partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>structure</i> : <i>TypeIndex</i>
<i>members</i> : <i>ExprIndex</i>
<i>base_subobjects</i> : <i>ExprIndex</i>

Figure 10.47: Structure of a product value expression

The meaning of the fields is as follows

- *locus* denotes the source location of this expression.
- *type* denotes the type of this expression.
- *structure* denotes the class type of the object designated by this expression.
- *members* denotes the sequence of direct non-base class subobjects of the object.
- *base_subobjects* denotes the sequence of base class subobjects of the object.

Partition name: "expr.product-type-value".

Note: This structure is subject to change in future releases

ExprSort::SumTypeValue

An *ExprIndex* value with tag `ExprSort::SumTypeValue` designates an object of a union type, usually produced as part of a compile-time evaluation. The *index* field of that abstract reference is an index into the sum type value partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>variant</i> : <i>TypeIndex</i>
<i>discriminant</i> : <i>ActiveMember</i>
<i>value</i> : <i>ExprIndex</i>

Figure 10.48: Structure of a sum type value expression

with

```
enum class ActiveMember : uint32_t { };
```

Figure 10.49: Definition of type *ActiveMember*

. The active member numbers the non-static data member in the union, starting from 0.

The meaning of the fields is as follows

- *locus* denotes the source location of this expression.
- *type* denotes the type of this expression.
- *variant* denotes the union class type of the object designated by this expression.
- *discriminant* denotes the index of the active member of the union object
- *value* denotes the value of the active member.

Partition name: "expr.sum-type-value".

Note: This structure is subject to change in future releases

ExprSort::SubobjectValue

Partition name: "expr.class-subobject-value".

Note: This structure is scheduled for removal in future releases

ExprSort::ArrayValue

An *ExprIndex* value with tag `ExprSort::ArrayValue` designates an object of an array type, usually produced as part of a compile-time evaluation. The *index* field of that abstract reference is an index into the array value partition. Each entry in that partition is a structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>elements</i> : <i>ExprIndex</i>
<i>element_type</i> : <i>TypeIndex</i>

Figure 10.50: Structure of an array value expression

The meaning of the fields is as follows

- *locus* denotes the source location of this expression.
- *type* denotes the type of this expression.
- *elements* denotes the sequence of subobjects of the array.
- *element_type* denotes the (common) type of the array element.

Partition name: "expr.array-value".

ExprSort::DynamicDispatch

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>pivot</i> : <i>ExprIndex</i>

Figure 10.51: Structure of a dynamic dispatch expression

Partition name: "expr.dynamic-dispatch".

ExprSort::VirtualFunctionConversion

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>function</i> : <i>DeclIndex</i>

Figure 10.52: Structure of a virtual function conversion

Partition name: "expr.virtual-function-conversion".

ExprSort::Placeholder

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>

Figure 10.53: Structure of a placeholder expression

Partition name: "expr.placeholder".

ExprSort::Expansion

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>operand</i> : <i>ExprIndex</i>

Figure 10.54: Structure of an expansion expression

Partition name: "expr.expansion".

ExprSort::Generic

TBD

Figure 10.55: Structure of a generic expression selection

Partition name: "expr.generic".

ExprSort::Tuple

A *ExprIndex* value with tag `ExprSort::Tuple` represents a reference to sequence or more abstract indices to expressions. This is useful for representing expression lists, including template argument lists. The *index* field is index into the tuple expression partition. Each entry in that partition has three components: a *start* field, a *cardinality* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 10.56: Structure of a tuple expression

The *start* field is an index into the expression heap partition. It points to the first expression abstract reference in the tuple. The *cardinality* field denotes the number of expression abstract references in the tuple. The *locus* field denotes the source location of the expression.

Partition name: "expr.tuple".

ExprSort::Nullptr

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>

Figure 10.57: Structure of a nullptr expression

Partition name: "expr.nullptr".

ExprSort::This

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>

Figure 10.58: Structure of a this expression

Partition name: "expr.this".

ExprSort::TemplateReference

A reference to a member of a template

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>member_name</i> : <i>NameIndex</i>
<i>member_locus</i> : <i>SourceLocation</i>
<i>scope</i> : <i>TypeIndex</i>
<i>arguments</i> : <i>ExprIndex</i>

Figure 10.59: Structure of a template member expression

The *member_name* field designates the name of the member; the *member_locus* designates the source location where the member is declared. The *scope* field designates the enclosing scope of the member. The *arguments* designates the set of template arguments to this member. The field *locus* is the source location where the expression appears.

Partition name: "expr.template-reference".

Note: This structure is subject to removal in future releases.

ExprSort::PushState

A EH push-state expression (constructor call + matching destructor call)

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>ctor_call</i> : <i>ExprIndex</i>
<i>dtor_call</i> : <i>ExprIndex</i>
<i>flags</i> : <i>EHFlags</i>

Figure 10.60: Structure of a push state expression

The *flags* field is of type

```
enum class EHFlags : uint16_t { };
```

Figure 10.61: Definition of type *EHFlags*

Partition name: "expr.push-state".

ExprSort::TypeTraitIntrinsic

A use of a type trait intrinsic

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>arguments</i> : <i>TypeIndex</i>
<i>intrinsic</i> : <i>Operator</i>

Figure 10.62: Structure of an intrinsic type-trait expression

Partition name: "expr.type-trait".

ExprSort::DesignatedInitializer

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>member</i> : <i>TextOffset</i>
<i>initializer</i> : <i>ExprIndex</i>

Figure 10.63: Structure of a designated initializer

Partition name: "expr.designated-init".

ExprSort::PackedTemplateArguments

A *ExprIndex* value with tag *ExprIndex::PackedTemplateArguments* represents an abstract reference to a template argument list for a template parameter pack. The *index* field of that abstract reference is an index into the packed template-argument list partition. Each entry in that partition is a structure with the following layout:

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>arguments</i> : <i>ExpexIndex</i>

with the these meanings of the fields:

- *locus* designates the source location of the expression
- *type* designates the type of the expression – currently null
- *arguments* designate the actual argument, or sequence of template-arguments.

Partition name: "expr.packed-template-arguments".

Note: This representation is subject to change.

ExprSort::Tokens

A *ExprIndex* value with tag `ExprSort::Tokens` represents an arbitrary token sequence (yet to be parsed) denoting an expression. The *index* field is an index into the token sequence expression partition. Each entry of that partition is a structure with the following components: a *tokens* field, and a *locus* field.

<i>locus</i> : <i>SourceLocation</i>
<i>type</i> : <i>TypeIndex</i>
<i>words</i> : <i>SentenceIndex</i>

Figure 10.64: Structure of a token sequence expression

The *words* field is an index into the sentence partition (§19). The *locus* field denotes the source location.

Partition name: "expr.tokens".

Note: This kind of representation of expression is discouraged and will be removed in future version of this document.

ExprSort::AssignInitializer

Partition name: "expr.assign-initializer".

<i>equal</i> : <i>SourceLocation</i>
<i>initializer</i> : <i>ExprIndex</i>

10.2 Operators

Elaboration of C++ expressions involves semantic operators which are classified by sort. Semantic operators are 16-bit precision values with the following layout:

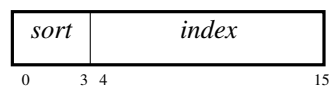


Figure 10.65: *Operator*: Structure of semantic operator

The field *sort*, of type *OperatorSort*, designates the semantic category of the operator. The field *index* is a 12-bit value the interpretation of which is *sort*-dependent as indicated in the subsections below.

The type *OperatorSort* is a set of 4-bit values enumerated as follows.

0x00. Niladic	0x03. Triadic
0x01. Monadic	0x0E. Storage
0x02. Dyadic	0x0F. Variadic

Niladic operators

A sort value *OperatorSort::Niladic* indicates a niladic operator – an operator accepting no argument. The value of the *index* is to be interpreted as a value of type *NiladicOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x400. Msvc
0x01. Phantom	0x401. MsvcConstantObject
0x02. Constant	0x402. MsvcLambda
0x03. Nil	

NiladicOperator::Unknown

An invalid niladic operator, sometimes stands for an undefined value.

NiladicOperator::Phantom

A missing expression in the input source level, e.g. as the array bound the type `int[]`.

NiladicOperator::Constant

A scalar constant, or string literal, or a class type object constant expression.

NiladicOperator::Nil

Representation of the source level semantic equivalent of `void()`.

NiladicOperator::Msvc

This is a marker, not an actual operator. Niladic operators with value greater than this are MSVC extensions.

NiladicOperator::MsvcConstantObject

An MSVC extension for representing class type object constants. Semantically, this is the same as *NiladicOperator::Constant*.

Note: This representation is scheduled for removal in future MSVC releases.

NiladicOperator::MsvcLambda

An MSVC extension to represent a lambda constant. Semantically, this is the same as **NiladicOperator::Constant**.

Note: This representation is scheduled for removal in future MSVC releases.

Monadic operators

A sort value **OperatorSort::Monadic** indicates a monadic operator – an operator accepting one argument. The value of the *index* is to be interpreted as a value of type *MonadicOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x1B. New
0x01. Plus	0x1C. Delete
0x02. Negate	0x1D. DeleteArray
0x03. Deref	0x1E. Expand
0x04. Address	0x1F. Read
0x05. Complement	0x20. Materialize
0x06. Not	0x21. PseudoDtorCall
0x07. PreIncrement	0x22. LookupGlobally
0x08. PreDecrement	0x400. Msvc
0x09. PostIncrement	0x401. MsvcAssume
0x0A. PostDecrement	0x402. MsvcAlignof
0x0B. Truncate	0x403. MsvcUuidof
0x0C. Ceil	0x404. MsvclsClass
0x0D. Floor	0x405. MsvclsUnion
0x0E. Paren	0x406. MsvclsEnum
0x0F. Brace	0x407. MsvclsPolymorphic
0x10. Alignas	0x408. MsvclsEmpty
0x11. Alignof	0x409. MsvclsTriviallyCopyConstructible
0x12. Sizeof	0x40A. MsvclsTriviallyCopyAssignable
0x13. Cardinality	0x40B. MsvclsTriviallyDestructible
0x14. Typeid	0x40C. MsvcHasVirtualDestructor
0x15. Noexcept	0x40D. MsvclsNothrowCopyConstructible
0x16. Requires	0x40E. MsvclsNothrowCopyAssignable
0x17. CoReturn	0x40F. MsvclsPod
0x18. Await	0x410. MsvclsAbstract
0x19. Yield	0x411. MsvclsTrivial
0x1A. Throw	0x412. MsvclsTriviallyCopyable
	0x413. MsvclsStandardLayout
	0x414. MsvclsLiteralType

0x415. <code>MsvcIsTriviallyMoveConstructible</code>	0x421. <code>MsvcIsSimpleValueClass</code>
0x416. <code>MsvcHasTrivialMoveAssign</code>	0x422. <code>MsvcIsInterfaceClass</code>
0x417. <code>MsvcIsTriviallyMoveAssignable</code>	0x423. <code>MsvcIsDelegate</code>
0x418. <code>MsvcIsNothrowMoveAssignable</code>	0x424. <code>MsvcIsFinal</code>
0x419. <code>MsvcUnderlyingType</code>	0x425. <code>MsvcIsSealed</code>
0x41A. <code>MsvcIsDestructible</code>	0x426. <code>MsvcHasFinalizer</code>
0x41B. <code>MsvcIsNothrowDestructible</code>	0x427. <code>MsvcHasCopy</code>
0x41C. <code>MsvcHasUniqueObjectRepresentation</code>	0x428. <code>MsvcHasAssign</code>
0x41D. <code>MsvcIsAggregate</code>	0x429. <code>MsvcHasUserDestructor</code>
0x41E. <code>MsvcBuiltinAddressOf</code>	0xFE0. <code>MsvcConfusion</code>
0x41F. <code>MsvcIsRefClass</code>	0xFE1. <code>MsvcConfusedExpand</code>
0x420. <code>MsvcIsValueClass</code>	0xFE2. <code>MsvcConfusedDependentSizeof</code>

MonadicOperator::Unknown

An invalid monadic operator. This value should never be generated.

MonadicOperator::Plus

Source-level prefix “+” operator.

MonadicOperator::Negate

Source-level prefix “-” operator.

MonadicOperator::Deref

Source-level (pointer) dereference “*” operator.

MonadicOperator::Address

Source-level address-of “&” operator.

MonadicOperator::Complement

Source-level bit complement “~” operator.

MonadicOperator::Not

Source-level logical prefix “!” (i.e. “not”) operator.

MonadicOperator::PreIncrement

Source-level prefix “++” operator.

MonadicOperator::PreDecrement

Source-level prefix “--” operator.

MonadicOperator::PostIncrement

Source-level postfix “++” operator.

MonadicOperator::PostDecrement

Source-level postfix “--” operator.

MonadicOperator::Truncate

C++ abstract machine operator.

MonadicOperator::Ceil

C++ abstract machine operator.

MonadicOperator::Floor

C++ abstract machine operator.

MonadicOperator::Paren

Source-level parenthesis-enclosing operator.

MonadicOperator::Brace

Source-level brace-enclosing operator.

MonadicOperator::Alignas

Source-level “alignas” operator.

MonadicOperator::Alignof

Source-level “alignof” operator.

MonadicOperator::Sizeof

Source-level “sizeof” operator.

MonadicOperator::Cardinality

Source-level “sizeof...” operator.

MonadicOperator::Typeid

Source-level “typeid” operator.

MonadicOperator::Noexcept

Source-level “noexcept” operator.

MonadicOperator::Requires

Source-level “requires” operator.

MonadicOperator::CoReturn

Source-level “co_return” operator.

MonadicOperator::Await

Source-level “co_await” operator.

MonadicOperator::Yield

Source-level “co_yield” operator.

MonadicOperator::Throw

Source-level “throw” operator.

MonadicOperator::New

Source-level “new” operator.

MonadicOperator::Delete

Source-level “delete” operator.

MonadicOperator::DeleteArray

Source-level “delete[]” operator.

MonadicOperator::Expand

Source-level pack-expansion operator.

MonadicOperator::Read

C++ abstract machine lvalue-to-rvalue conversion operator.

MonadicOperator::Materialize

C++ abstract machine class temporary materialization operator.

MonadicOperator::PseudoDtorCall

Pseudo-destructor call operator.

MonadicOperator::LookupGlobally

Any operator in the immediate operand expression, at source-level, is to be looked-up in the global scope. For example, in `:new T(a, b)`, the corresponding storage allocation function operator `new` is to be looked up in the global scope.

MonadicOperator::Msvc

This is a marker, not an actual operator. Monadic operators with value greater than this are MSVC extensions.

MonadicOperator::MsvcAssume

Source-level “`__assume`” operator ([SourceKeyword::MsvcAssume](#)).

MonadicOperator::MsvcAlignof

Source-level “`__builtin_alignof`” operator ([SourceKeyword::MsvcAlignof](#)).

MonadicOperator::MsvcUuidof

Source-level “`__uuidof`” operator ([SourceKeyword::MsvcUuidof](#)).

MonadicOperator::MsvcIsClass

Source-level “`__is_class`” operator ([SourceKeyword::MsvcIsClass](#)).

MonadicOperator::MsvcIsUnion

Source-level “`__is_union`” operator ([SourceKeyword::MsvcIsUnion](#)).

MonadicOperator::MsvcIsEnum

Source-level “`__is_enum`” operator ([SourceKeyword::MsvcIsEnum](#)).

MonadicOperator::MsvcIsPolymorphic

Source-level “`__is_polymorphic`” operator ([SourceKeyword::MsvcIsPolymorphic](#)).

MonadicOperator::MsvcIsEmpty

Source-level “`__is_empty`” operator ([SourceKeyword::MsvcIsEmpty](#)).

MonadicOperator::MsvclsTriviallyCopyConstructible

Source-level “__is_trivially_copy_constructible” operator ([SourceKeyword::MsvcIsTriviallyCopyConstructible](#)).

MonadicOperator::MsvclsTriviallyCopyAssignable

Source-level “__is_trivially_copy_constructible” operator ([SourceKeyword::MsvcIsTriviallyCopyAssignable](#)).

MonadicOperator::MsvclsTriviallyDestructible

Source-level “__is_trivially_destructible” operator ([SourceKeyword::MsvcIsTriviallyDestructible](#)).

MonadicOperator::MsvcHasVirtualDestructor

Source-level “__has_virtual_destructor” operator ([SourceKeyword::MsvcHasVirtualDestructor](#)).

MonadicOperator::MsvclsNothrowCopyConstructible

Source-level “__is_nothrow_copy_constructible” operator ([SourceKeyword::MsvcIsNothrowCopyConstructible](#)).

MonadicOperator::MsvclsNothrowCopyAssignable

Source-level “__is_nothrow_copy_assignable” operator ([SourceKeyword::MsvcIsNothrowCopyAssignable](#)).

MonadicOperator::MsvclsPod

Source-level “__is_pod” operator ([SourceKeyword::MsvcIsPod](#)).

MonadicOperator::MsvclsAbstract

Source-level “__is_abstract” operator ([SourceKeyword::MsvcIsAbstract](#)).

MonadicOperator::MsvclsTrivial

Source-level “__is_trivial” operator ([SourceKeyword::MsvcIsTrivial](#)).

MonadicOperator::MsvclsTriviallyCopyable

Source-level “__is_trivially_copyable” operator ([SourceKeyword::MsvcIsTriviallyCopyable](#)).

MonadicOperator::MsvclsStandardLayout

Source-level “__is_standard_layout” operator ([SourceKeyword::MsvcIsStandardLayout](#)).

MonadicOperator::MsvclsLiteralType

Source-level “__is_literal_type” operator ([SourceKeyword::MsvcIsLiteralType](#)).

MonadicOperator::MsvclsTriviallyMoveConstructible

Source-level “__is_trivially_move_constructible” operator ([SourceKeyword::MsvcIsTriviallyMoveConstructible](#)).

MonadicOperator::MsvcHasTrivialMoveAssign

Source-level “__has_trivial_move_assign” operator ([SourceKeyword::MsvcHasTrivialMoveAssign](#)).

MonadicOperator::MsvclsTriviallyMoveAssignable

Source-level “__is_trivially_move_assignable” operator ([SourceKeyword::MsvcIsTriviallyMoveAssignable](#)).

MonadicOperator::MsvclsNothrowMoveAssignable

Source-level “__is_nothrow_move_assignable” operator ([SourceKeyword::MsvcIsNothrowMoveAssignable](#)).

MonadicOperator::MsvcUnderlyingType

Source-level “__underlying_type” operator ([SourceKeyword::MsvcUnderlyingType](#)).

MonadicOperator::MsvclsDestructible

Source-level “__is_destructible” operator ([SourceKeyword::MsvcIsDestructible](#)).

MonadicOperator::MsvclsNothrowDestructible

Source-level “__is_nothrow_destructible” operator ([SourceKeyword::MsvcIsNothrowDestructible](#)).

MonadicOperator::MsvcHasUniqueObjectRepresentations

Source-level “__has_unique_object_representations” operator ([SourceKeyword::MsvcHasUniqueObjectRepresentations](#)).

MonadicOperator::MsvclsAggregate

Source-level “__is_aggregate” operator ([SourceKeyword::MsvcIsAggregate](#)).

MonadicOperator::MsvcBuiltinAddressOf

Source-level “__builtin_addressof” operator ([SourceKeyword::MsvcBuiltinAddressOf](#)).

MonadicOperator::MsvclsRefClass

Source-level “__is_ref_class” operator ([SourceKeyword::MsvcIsRefClass](#)).

MonadicOperator::MsvclsValueClass

Source-level “__is_value_class” operator ([SourceKeyword::MsvcIsValueClass](#)).

MonadicOperator::MsvcIsSimpleValueClass

Source-level “__is_simple_value_class” operator ([SourceKeyword::MsvcIsSimpleValueClass](#)).

MonadicOperator::MsvcIsInterfaceClass

Source-level “__is_interface_class” operator ([SourceKeyword::MsvcIsInterfaceClass](#)).

MonadicOperator::MsvcIsDelegate

Source-level “__is_delegate” operator ([SourceKeyword::MsvcIsDelegate](#)).

MonadicOperator::MsvcIsFinal

Source-level “__is_final” operator ([SourceKeyword::MsvcIsFinal](#)).

MonadicOperator::MsvcIsSealed

Source-level “__is_sealed” operator ([SourceKeyword::MsvcIsSealed](#)).

MonadicOperator::MsvcHasFinalizer

Source-level “__has_finalizer” operator ([SourceKeyword::MsvcHasFinalizer](#)).

MonadicOperator::MsvcHasCopy

Source-level “__has_copy” operator ([SourceKeyword::MsvcHasCopy](#)).

MonadicOperator::MsvcHasAssign

Source-level “__has_assign” operator ([SourceKeyword::MsvcHasAssign](#)).

MonadicOperator::MsvcHasUserDestructor

Source-level “__has_user_destructor” operator ([SourceKeyword::MsvcHasUserDestructor](#)).

MonadicOperator::MsvcConfusion

This is not a real operator, rather a sentinel value. Monadic operator values greater than this have alternate representations, and signify some infelicities in the MSVC parser. They are scheduled to be removed in future releases.

MonadicOperator::MsvcConfusedExpand

Source-level pack expansion “...” operator.

MonadicOperator::MsvcConfusedDependentSizeof

Source-level sizeof operator, used in an expression where the operand is dependent.

Dyadic operators

A sort value `OperatorSort::Dyadic` indicates a dyadic operator – an operator accepting two arguments. The value of the *index* is to be interpreted as a value of type *DyadicOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x21. Dot
0x01. Plus	0x22. Arrow
0x02. Minus	0x23. DotStar
0x03. Mult	0x24. ArrowStar
0x04. Slash	0x25. Curry
0x05. Modulo	0x26. Apply
0x06. Remainder	0x27. Index
0x07. Bitand	0x28. DefaultAt
0x08. Bitor	0x29. New
0x09. Bitxor	0x2A. NewArray
0x0A. Lshift	0x2B. Destruct
0x0B. Rshift	0x2C. DestructAt
0x0C. Equal	0x2D. Cleanup
0x0D. NotEqual	0x2E. Qualification
0x0E. Less	0x2F. Promote
0x0F. LessEqual	0x30. Demote
0x10. Greater	0x31. Coerce
0x11. GreaterEqual	0x32. Rewrite
0x12. Compare	0x33. Bless
0x13. LogicAnd	0x34. Cast
0x14. LogicOr	0x35. ExplicitConversion
0x15. Assign	0x36. ReinterpretCast
0x16. PlusAssign	0x37. StaticCast
0x17. MinusAssign	0x38. ConstCast
0x18. MultAssign	0x39. DynamicCast
0x19. SlashAssign	0x3A. Narrow
0x1A. ModuloAssign	0x3B. Widen
0x1B. BitandAssign	0x3C. Pretend
0x1C. BitorAssign	0x3D. Closure
0x1D. BitxorAssign	0x3E. ZeroInitialize
0x1E. LshiftAssign	0x3F. ClearStorage
0x1F. RshiftAssign	0x40. Select
0x20. Comma	0x400. Msvc
	0x401. MsvcTryCast
	0x402. MsvcCurry

0x403. MsvcVirtualCurry	0x40D. MsvcIsNothrowAssignable
0x404. MsvcAlign	0x40E. MsvcIsAssignable
0x405. MsvcBitSpan	0x40F. MsvcIsAssignableNocheck
0x406. MsvcBitfieldAccess	0x410. MsvcBuiltinBitCast
0x407. MsvcObscureBitfieldAccess	0x411. MsvcBuiltinIsLayoutCompatible
0x408. MsvcInitialize	0x412. MsvcBuiltinIsPointerInterconvertibleBaseOf
0x409. MsvcBuiltinOffsetOf	0x413. MsvcBuiltinIsPointerInterconvertibleWithClass
0x40A. MsvcIsBaseOf	0x414. MsvcBuiltinIsCorrespondingMember
0x40B. MsvcIsConvertibleTo	0x415. MsvcIntrinsic
0x40C. MsvcIsTriviallyAssignable	0x416. MsvcSaturatedArithmetic

DyadicOperator::Unknown

An invalid dyadic operator. This value should never be generated.

DyadicOperator::Plus

Source level binary operator “+”.

DyadicOperator::Minus

Source level binary operator “-”.

DyadicOperator::Mult

Source level binary operator “*”.

DyadicOperator::Slash

Source level binary operator “/”.

DyadicOperator::Modulo

The Euclidean modulo operator at the abstract machine semantics level. At source level, this corresponds to the binary operator “%”. When both operands are integers, the result is always non-negative, between 0 and the absolute value of the second operand. See also [DyadicOperator::Remainder](#).

DyadicOperator::Remainder

Source level binary operator “%”. When both operands are integers the result is truncated towards 0 as defined the corresponding ISO standards of C11 and up, and C++11 and up, See also [DyadicOperator::Modulo](#).

DyadicOperator::Bitand

Source level binary operator “&”.

DyadicOperator::Bitor

Source level binary operator “|”.

DyadicOperator::Bitxor

Source level binary operator “^”.

DyadicOperator::Lshift

Source level binary operator “<<”.

DyadicOperator::Rshift

Source level binary operator “>>”.

DyadicOperator::Equal

Source level binary operator “==”.

DyadicOperator::NotEqual

Source level binary operator “!=”.

DyadicOperator::Less

Source level binary operator “<”.

DyadicOperator::LessEqual

Source level binary operator “<=”.

DyadicOperator::Greater

Source level binary operator “>”.

DyadicOperator::GreaterEqual

Source level binary operator “>=”.

DyadicOperator::Compare

Source level binary operator “<=>”.

DyadicOperator::LogicAnd

Source level binary operator “&&”.

DyadicOperator::LogicOr

Source level binary operator “||”.

DyadicOperator::Assign

Source level binary operator “=”.

DyadicOperator::PlusAssign

Source level binary operator “+=”.

DyadicOperator::MinusAssign

Source level binary operator “-=”.

DyadicOperator::MultAssign

Source level binary operator “*=”.

DyadicOperator::SlashAssign

Source level binary operator “/=”.

DyadicOperator::ModuloAssign

Source level binary operator “%=”.

DyadicOperator::BitandAssign

Source level binary operator “&=”.

DyadicOperator::BitorAssign

Source level binary operator “|=”.

DyadicOperator::BitxorAssign

Source level binary operator “^=”.

DyadicOperator::LshiftAssign

Source level binary operator “<<=”.

DyadicOperator::RshiftAssign

Source level binary operator “>>=”.

DyadicOperator::Comma

Source level binary operator “,”.

DyadicOperator::Dot

Source level binary operator “.”.

DyadicOperator::Arrow

Source level binary operator “->”.

DyadicOperator::DotStar

Source level binary operator “.*”.

DyadicOperator::ArrowStar

Source level binary operator “->*”.

DyadicOperator::Curry

Abstract machine operation binding the first parameter of a function taking at least one argument.

DyadicOperator::Apply

Abstract machine operation applying a callable to an argument list (conceptually a tuple).

DyadicOperator::Index

Source level binary operator “[]” as in “x[y]”.

DyadicOperator::DefaultAt

Abstraction machine operation corresponding to default construction of an object at a given address, e.g. “new(p) T”.

DyadicOperator::New

Abstract machine operation corresponding to allocating appropriate storage and constructing an object of a given type and with a given initializer, e.g. “new T(x)”.

DyadicOperator::NewArray

Abstract machine operation corresponding to allocating appropriate storage and default constructing an array of a given element type and length, e.g. “new T[n]”.

DyadicOperator::Destruct

Abstract machine operation corresponding to the explicit call of a destructor for an object, e.g. “x.~T()”. See also [DyadicOperator::DestructAt](#).

DyadicOperator::DestructAt

Abstract machine operation corresponding to the explicit call of a destructor through a pointer, e.g. “p->~T()”. See also [DyadicOperator::Destruct](#).

DyadicOperator::Cleanup

Abstract machine operation evaluating the first operand, then running the second operand as a cleanup (object destruction).

DyadicOperator::Qualification

Abstract machine operation corresponding to implicit cv-qualification of the type of the expression, e.g. as in from “T” to “const T”.

DyadicOperator::Promote

Abstract machine operation corresponding to integral or floating point promotion at the source level. See also [DyadicOperator::Demote](#).

DyadicOperator::Demote

Abstract machine operation corresponding to the inverse of an integral or floating point promotion at the source level. See also [DyadicOperator::Promote](#).

DyadicOperator::Coerce

Abstract machine operation corresponding to an implicit conversion at the source level that is neither a promotion ([DyadicOperator::Promote](#)) nor a demotion ([DyadicOperator::Demote](#)).

DyadicOperator::Rewrite

Abstract machine operation where the semantics of the first operand (source-level construct) is defined by that of the second operand.

DyadicOperator::Bless

Abstract machine operation proclaiming a valid object of given type (second operand) at a given address (first operand). Note that this operation is not a placement-new operator (which would entail running a constructor).

DyadicOperator::Cast

A C-style cast operation, e.g. “T)x”

DyadicOperator::ExplicitConversion

A functional cast notation, e.g. “T(x)”.

DyadicOperator::ReinterpretCast

Source level operator “reinterpret_cast”.

DyadicOperator::StaticCast

Source level operator “static_cast”.

DyadicOperator::ConstCast

Source level operator “const_cast”.

DyadicOperator::DynamicCast

Source level operator “dynamic_cast”.

DyadicOperator::Narrow

Abstract machine operation corresponding to the runtime-checked conversion of a pointer of type “B*” (or reference of type “B&”) to a pointer of type “D*” (or reference of type “D&”) where the class “D” is a derived class of “B”.

DyadicOperator::Widen

Abstract machine operation corresponding to the implicit conversion of a pointer of type “D*” (or reference of type “D&”) to a pointer of type “B*” (or a reference of type “B&”), where the class “D” is a derived class of “B”.

DyadicOperator::Pretend

Abstract machine operation generalizing bitcat and reinterpret_cast.

DyadicOperator::Closure

Abstract machine operation pairing a function pointer (the second operand) and an environment of captured variables (the first operand) for proper execution, as in lambda expressions.

DyadicOperator::ZeroInitialize

Abstract machine operation performing zero-initialization of an object or a subobject. See also [DyadicOperator::ClearStorage](#).

DyadicOperator::ClearStorage

Abstract machine operation clearing (i.e. setting all bytes to value 0) a storage span. See also [DyadicOperator::ZeroInitialize](#).

DyadicOperator::Select

Use of the source-level scope resolution operator “:”. The first operand designates the scope, and the second operand designates the member to select.

DyadicOperator::Msvc

This is a marker, not an actual operator. Dyadic operators with value greater than this are MSVC extensions.

DyadicOperator::MsvcTryCast

Abstract machine operation corresponding to the WinRT extension operation of “*try cast*”.

DyadicOperator::MsvcCurry

Abstract machine operation corresponding to the MSVC extension of bound member function, e.g. “*this->fun*” where *fun* is a non-static member function. The first operand designates the object, and the second operand designates the non-static member function. See also [DyadicOperator::MsvcVirtualCurry](#).

DyadicOperator::MsvcVirtualCurry

Abstract machine operation with similar semantics as that of [DyadicOperator::MsvcCurry](#), except the resulting callable requires a dynamic dispatch.

DyadicOperator::MsvcAlign

Abstract machine operation of aligning a pointer to an adequate storage address boundary.

DyadicOperator::MsvcBitSpan

Abstract machine operation describing the span of a bitfield. The first operand describes the offset (in number of bits) from the start of the storage hosting the bitfield. The second operand describes the number of bits spanned by the bitfield.

DyadicOperator::MsvcBitfieldAccess

Abstract machine operation describing access to a bitfield. The first operand designates the start of the storage hosting the bitfield. The second operand describes the span ([DyadicOperator::MsvcBitSpan](#)) of the bitfield.

DyadicOperator::MsvcObscureBitfieldAccess

Abstract machine operation describing access to a bitfield. See also [DyadicOperator::MsvcBitfieldAccess](#).

DyadicOperator::MsvcInitialize

Abstract machine operation describing the initialization of an object. The first operand designates the storage to initialize. The second operand designates the initializer or the function to run to perform initialization (in the case constructor).

DyadicOperator::MsvcBuiltinOffsetOf

Source level “__builtin_offsetof” operator ([SourceKeyword::MsvcBuiltinOffsetOf](#)).

DyadicOperator::MsvcIsBaseOf

Source level “__is_base_of” operator ([SourceKeyword::MsvcIsBaseOf](#)).

DyadicOperator::MsvcIsConvertibleTo

Source level “__is_convertible_to” operator ([SourceKeyword::MsvcIsConvertibleTo](#)).

DyadicOperator::MsvcIsTriviallyAssignable

Source level “__is_trivially_assignable” operator ([SourceKeyword::MsvcIsTriviallyAssignable](#)).

DyadicOperator::MsvcIsNothrowAssignable

Source level “__is_nothrow_assignable” operator ([SourceKeyword::MsvcIsNothrowAssignable](#)).

DyadicOperator::MsvcIsAssignable

Source level “__is_assignable” operator ([SourceKeyword::MsvcIsAssignable](#)).

DyadicOperator::MsvcIsAssignableNocheck

Source level “__is_assignable_no_precondition_check” operator ([SourceKeyword::MsvcIsAssignableNocheck](#))

DyadicOperator::MsvcBuiltinBitCast

Source level “__builtin_bit_cast” operator ([SourceKeyword::MsvcBuiltinBitCast](#)).

DyadicOperator::MsvcBuiltinIsLayoutCompatible

Source level “__builtin_is_layout_compatible” operator ([SourceKeyword::MsvcBuiltinIsLayoutCompatible](#)).

DyadicOperator::MsvcBuiltinIsPointerInterconvertibleBaseOf

Source level “__builtin_is_pointer_interconvertible_base_of” operator ([SourceKeyword::MsvcBuiltinIsPointerInterconvertibleBaseOf](#)).

DyadicOperator::MsvcBuiltinIsPointerInterconvertibleWithClass

Source level “__builtin_is_pointer_interconvertible_with_class” operator ([SourceKeyword::MsvcBuiltinIsPointerInterconvertibleWithClass](#)).

DyadicOperator::MsvcBuiltinIsCorrespondingMember

Source level “__builtin_is_corresponding_member” operator ([SourceKeyword::MsvcBuiltinIsCorrespondingMember](#)).

DyadicOperator::MsvcIntrinsic

Abstract machine operation corresponding to the call of an MSVC intrinsic operator of function. The first operand is an integer constant describing the intrinsic; the second operand is the argument list. See also [ExprSort::Call](#).

DyadicOperator::MsvcSaturatedArithmetic

An MSVC intrinsic for an abstract machine saturated arithmetic operation. If the arithmetic computation described by the first operand overflows, then the result is the saturated value indicated by the second operand.

Triadic operators

A sort value `OperatorSort::Triadic` indicates a triadic operator – an operator accepting three arguments. The value of the *index* is to be interpreted as a value of type *TriadicOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x400. Msvc
0x01. Choice	0xFE0. MsvcConfusion
0x02. ConstructAt	0xFE1. MsvcConfusedPushState
0x03. Initialize	

TriadicOperator::Unknown

An invalid triadic operator. This value should never be generated.

TriadicOperator::Choice

Source-level ternary “?:” operator.

TriadicOperator::ConstructAt

Source-level representation of “new(p) T(x)”, where

- the first operand is the placement p
- the second operand is the constructed type T
- the third operand is the initializing value x

TriadicOperator::Initialize

C++ abstract machine operation.

TriadicOperator::Msvc

This is a marker, not an actual operator. Triadic operators with value greater than this are MSVC extensions.

TriadicOperator::MsvcConfusion

This operator is no longer emitted in this version and up of the IFC.

TriadicOperator::MsvcConfusedPushState

This is an MSVC-specific operator to represent a low-level cleanup action related to exception handling. The three operands to this operator have the following meanings:

1. the first operand designates a constructor call
2. the second operand designates a destructor call
3. the third operand designates an integer constant expression that is a 64-bit integer bitmask describing various semantic aspects of the cleanup action.

Note: This operator is scheduled for removal in future releases of the IFC.

Storage operators

A sort value `OperatorSort::Storage` indicates a storage allocation or deallocation operator. The value of the *index* is to be interpreted as a value of type *StorageOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x03. DeallocateSingle
0x01. AllocateSingle	0x04. DeallocateArray
0x02. AllocateArray	0x7DE. Msvc

StorageOperator::Unknown

An invalid triadic operator. This value should never be generated.

StorageOperator::AllocateSingle

Source-level “new” operator.

StorageOperator::AllocateArray

Source-level “new[]” operator.

StorageOperator::DeallocateSingle

Source-level “delete” operator.

StorageOperator::DeallocateArray

Source-level “delete[]” operator.

StorageOperator::Msvc

This is a marker, not an actual operator. Triadic operators with value greater than this are MSVC extensions.

Variadic operators

A sort value `OperatorSort::Variadic` indicates a variadic operator – an operator accepting any number of arguments. The value of the *index* is to be interpreted as a value of type *VariadicOperator*, which is a set of 13-bit values enumerated as follows.

0x00. Unknown	0x401. MsvcHasTrivialConstructor
0x01. Collection	0x402. MsvcIsConstructible
0x02. Sequence	0x403. MsvcIsNothrowConstructible
0x400. Msvc	0x404. MsvcIsTriviallyConstructible

VariadicOperator::Unknown

An invalid triadic operator. This value should never be generated.

VariadicOperator::Collection

C++ abstract machine. Collection of expressions, with no specific order of evaluation.

VariadicOperator::Sequence

C++ abstract machine. Like `VariadicOperator::Collection` (`VariadicOperator::Collection`) but with a defined left-to-right order of evaluation.

VariadicOperator::Msvc

This is a marker, not an actual operator. Triadic operators with value greater than this are MSVC extensions.

VariadicOperator::MsvcHasTrivialConstructor

Source-level “__has_trivial_constructor” operator (`SourceKeyword::MsvcHasTrivialConstructor`).

VariadicOperator::MsvcIsConstructible

Source-level “__is_constructible” operator (`SourceKeyword::MsvcIsConstructible`).

VariadicOperator::MsvcIsNothrowConstructible

Source-level “__is_nothrow_constructible” operator (`SourceKeyword::MsvcIsNothrowConstructible`).

VariadicOperator::MsvcIsTriviallyConstructible

Source-level “__is_trivially_constructible” operator (`SourceKeyword::MsvcIsTriviallyConstructible`).

Chapter 11

Statements

Note: All data structures described in this chapter are subject to change. Statements are represented in an IFC as part of the body of constexpr or inline function defined in a module interface source file. A statement is designated an abstract reference of type *StmtIndex*:

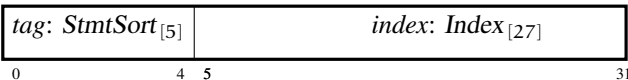


Figure 11.1: *StmtIndex*: Abstract reference of statement

The type *StmtSort* is a set of 5-bit values enumerated as follows

- | | |
|-----------------------|--------------------|
| 0x00. VendorExtension | 0x09. DoWhile |
| 0x01. Empty | 0x0A. Default |
| 0x02. If | 0x0B. Continue |
| 0x03. For | 0x0C. Expression |
| 0x04. Case | 0x0D. Return |
| 0x05. While | 0x0E. VariableDecl |
| 0x06. Block | 0x0F. Expansion |
| 0x07. Break | 0x10. SyntaxTree |
| 0x08. Switch | |

11.1 Statement structures

VendorExtension

Partition name: "stmt.vendor-extension".

StmtSort::Empty

<i>locus: SourceLocation</i>

Figure 11.2: Structure of an empty statement

Partition name: "stmt.empty".

StmtSort::If

<i>initialization: StmtIndex</i>
<i>condition: StmtIndex</i>
<i>consequence: StmtIndex</i>
<i>alternative: StmtIndex</i>
<i>locus: SourceLocation</i>

Figure 11.3: Structure of an if-statement

Partition name: "stmt.if".

StmtSort::For

<i>initialization: StmtIndex</i>
<i>condition: StmtIndex</i>
<i>continuation: StmtIndex</i>
<i>body: StmtIndex</i>
<i>locus: SourceLocation</i>

Figure 11.4: Structure of a for-statement

Partition name: "stmt.for".

StmtSort::Case

<i>expr: ExprIndex</i>
<i>locus: SourceLocation</i>

Figure 11.5: Structure of a case-label

Partition name: "stmt.case".

StmtSort::While

<i>condition: StmtIndex</i>
<i>body: StmtIndex</i>
<i>locus: SourceLocation</i>

Figure 11.6: Structure of a while-statement

Partition name: "stmt.while".

StmtSort::Block

<i>start: Index</i>
<i>cardinality: Cardinality</i>

Figure 11.7: Structure of a block statement

Partition name: "stmt.block".

StmtSort::Break

<i>locus: SourceLocation</i>

Figure 11.8: Structure of a break statement

Partition name: "stmt.break".

StmtSort::Switch

<i>initialization: StmtIndex</i>
<i>condition: ExprIndex</i>
<i>body: StmtIndex</i>
<i>locus: SourceLocation</i>

Figure 11.9: Structure of a switch-statement

Partition name: "stmt.switch".

StmtSort::DoWhile

<i>condition: StmtIndex</i>
<i>body: StmtIndex</i>
<i>locus: SourceLocation</i>

Figure 11.10: Structure of a do-statement

Partition name: "stmt.do-while".

StmtSort::Default

<i>locus: SourceLocation</i>

Figure 11.11: Structure of a default label

Partition name: "stmt.default".

StmtSort::Continue

<i>locus: SourceLocation</i>

Figure 11.12: Structure of a continue statement

Partition name: "stmt.continue".

StmtSort::Expression

<i>expr: ExprIndex</i>
<i>locus: SourceLocation</i>

Figure 11.13: Structure of an expression-statement

Partition name: "stmt.expression".

StmtSort::Return

<i>expr</i> : <i>ExprIndex</i>
<i>function_type</i> : <i>TypeIndex</i>
<i>expression_type</i> : <i>TypeIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 11.14: Structure of a return statement

Partition name: "stmt.return".

StmtSort::VariableDecl

<i>decl</i> : <i>DeclIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 11.15: Structure of a variable-declaration statement

Partition name: "stmt.variable".

StmtSort::Expansion

A *StmtIndex* abstract reference with tag `StmtSort::Expansion` designates a statement expansion.

<i>operand</i> : <i>StmtIndex</i>

Figure 11.16: Structure of an expansion statement

Partition name: "stmt.expansion".

StmtSort::SyntaxTree

TBD

Partition name: "stmt.syntax-tree".

Chapter 12

Names

Names are indicated by abstract references. This document uses *NameIndex* to designate a typed abstract reference to a name. Like all abstract references, it is a 32-bit value

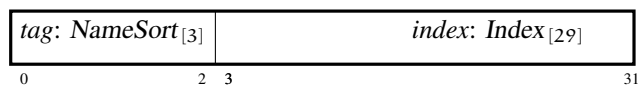


Figure 12.1: *NameIndex*: Abstract reference of names

The type *NameSort* is a set of 3-bit values enumerated as follows

0x00. Identifier	0x04. Template
0x01. Operator	0x05. Specialization
0x02. Conversion	0x06. SourceFile
0x03. Literal	0x07. Guide

In most cases, the *index* field of a *NameIndex* is a numerical index into a partition (§2.9) denoted by the *tag* field.

12.1 Name structures

NameSort::Identifier

A *NameIndex* value with tag *NameSort::Identifier* represents an abstract reference for normal alphabetic identifiers. The *index* field in this case is also the structure representation of the identifier: it is conceptually a value of type *TextOffset*, represented with 29 bits. It is an index into the string table (§3).

<i>tag</i> = NameSort::Identifier _[3]
<i>value</i> : TextOffset _[29]

Figure 12.2: Structure of a *NameIndex* representing an identifier

Note: Identifiers do not have a separate dedicated partition. Rather, identifiers are stored in the IFC's string table.

NameSort::Operator

A *NameIndex* value with tag NameSort::Operator represents an operator function name. The *index* field is an index into the operator partition (§2.9). Each entry in that partition has two components: a *category* field denoting the specified operator, and an *encoded* field.

<i>encoded</i> : TextOffset
<i>operator</i> : Operator

Figure 12.3: Structure of an operator-function name

The *encoded* field is a text encoding of the operator function name. The *operator* field is a 16-bit value (§10.2) capable of denoting any operator used in a valid C++ program.

Partition name: "name.operator".

NameSort::Conversion

A *NameIndex* value with tag NameSort::Conversion represents a conversion-function name. The *index* field is an index into the conversion function name partition. Each entry in that partition is a structure with two components.

<i>target</i> : TypeIndex
<i>encoded</i> : TextOffset

Figure 12.4: Structure of a conversion-function name

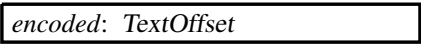
The *target* field is an abstract reference to the target type (§9) of the conversion function. The *encoded* field designates a text encoding of the operator function name.

Note: This data structure is subject to change

Partition name: "name.conversion".

NameSort::Literal

A *NameIndex* value with tag `NameSort::Literal` represents a reference to a string literal operator name. The *index* field is an index into the string literal operator partition (§2.9). Each element in that partition has two components: a *suffix* field, and an *encoded* field.



```

    encoded: TextOffset
  
```

Figure 12.5: Structure of a literal-operator name

The *encoded* field is an index into the string table, and represents the text encoding of the literal operator.

Note: Previous versions of this document showed a *suffix* for the literal operator name. That did not match recent behavior of the VC++ compiler. That field will be added back in the future.

Partition name: "name.literal".

NameSort::Template

A *NameIndex* value with tag `NameSort::Template` represents a reference to an assumed (as opposed to *declared*) template name. This is the case of nested-name of qualified-id where the qualifier is a dependent name and the unqualified part is asserted to name a template. The *index* field is an index into the partition of assumed template names.



```

    name: NameIndex
  
```

Figure 12.6: Structure of an assumed template name

Each entry in that partition is a structure with exactly one field, *name*, that is itself a *NameIndex*. It is an error for the *tag* field of *name* to be `NameSort::Template`.

Partition name: "name.template".

NameSort::Specialization

A *NameIndex* value with tag `NameSort::Specialization` represents a reference to a template-id, i.e. what in C++ source code is a template-name followed by a template-argument list. The *index* field is an index into the template-id partition. Each entry in that partition has two components: a *primary* field, and an *arguments* field.

<i>primary</i> : <i>NameIndex</i>
<i>arguments</i> : <i>ExprIndex</i>

Figure 12.7: Structure of a template-id name

The *primary* field represents the name of the primary template. The *arguments* field represents the template argument list as an expression (§10). If the template-argument list is empty or a singleton, the *arguments* field an abstraction reference to that expression. Otherwise, the *arguments* field is a tuple expression.

Partition name: "name.specialization".

NameSort::SourceFile

A *NameIndex* value with tag `NameSort::SourceFile` represents a reference to a source file name. The *index* field is an index into the partition of source file names. Each entry in that partition has two components: a *path* field, and a *guard* field.

<i>path</i> : <i>TextOffset</i>
<i>guard</i> : <i>TextOffset</i>

Figure 12.8: Structure of a source file name

The *path* field is an index into the string table. The *guard* field is also an index into the string table, and represents the identifier of the source-level include guard of the file, if it has any.

Partition name: "name.source-file".

NameSort::Guide

A *NameIndex* value with tag `NameSort::SourceFile` represents a reference to a user-authored deduction guide name for a class template. Note that deduction guides don't have names at the C++ source level. The *index* field is an index into the deduction guides partition. Each entry in that partition has one component: a *DeclIndex* designating the primary (class) template (`sec:ifc:DeclSort:Template`):

<i>primary_template</i> : <i>DeclIndex</i>
--

Figure 12.9: Structure of a deduction guide name

Partition name: "name.guide".

Charts

<i>tag</i> : <i>ChartSort</i> _[2]	<i>index</i> : <i>Index</i> _[30]
0 1 2	3

The type *ChartSort* is a set of 2-bit values enumerated as follows

13.1 Chart structures

Partition name: "chart.none".

131

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>
<i>constraint</i> : <i>ExprIndex</i>

Figure 13.2: Structure of a unilevel chart

The *start* is an index into the parameter declaration partition (§8.2). The *cardinality* field designates the number of parameters in that parameter list. The *constraint* denotes the condition of the requires clause, if present.

Partition name: "chart.unilevel".

ChartSort::Multilevel

A *ChartIndex* abstract reference with tag `ChartSort::Multilevel` indicates a set of template parameter lists, each of them either an empty template parameter list (tag `ChartSort::None`) or a unilevel template parameter list (tag `ChartSort::Unilevel`).

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 13.3: Structure of a multilevel chart

Partition name: "chart.multilevel".

Attributes

<i>tag</i> : AttrSort _[4]	<i>index</i> : Index _[28]
0 3 4	31

The type *AttrSort* is a set of 4-bit values enumerated as follows

When an input source file contains a C++ attribute (with *AttrIndex* value a) on a declaration (with *DeclIndex* value d), an associative pair (d, a) is entered into the ".msvc.trait.decl-attrs" partition (§17.9).

AttrSort::Nothing

No partition is emitted for attributes of this sort.

AttrSort::Basic

An *AttrIndex* value with tag `AttrSort::Basic` denotes a word (§19.2) used in an attribute, for example `answer` and `42` in “`[[answer(42)]]`” are basic attributes. The *index* field of that abstract reference is an index into the basic attributes partitions. Each entry in that partition is a structure with the following layout

<i>word</i> :	<i>Word</i>
---------------	-------------

Figure 14.2: Structure of a basic attribute

The *word* field represents that word making up this basic attribute.

Partition name: `"attr.basic"`.

AttrSort::Scoped

An *AttrIndex* reference with tag `AttrSort::Scoped` denotes a scoped attribute, that is a basic attribute followed by “`::`” and another basic attribute. For example, `gs1::suppress` in “`[[gs1::suppress(type.1)]]`” is a scoped attribute. The *index* field of that abstract reference is an index into the scoped attribute partition. Each entry in that partition is a structure with the following layout

<i>scope</i> :	<i>Word</i>
<i>member</i> :	<i>Word</i>

Figure 14.3: Structure of a scoped attribute

The *scope* field represents the word appearing before the “`::`” token. The *member* field represents the word appearing after the “`::`” token.

Partition name: `"attr.scoped"`.

AttrSort::Labeled

An *AttrIndex* reference with tag `AttrSort::labeled` denotes a labeled attribute of the form “`key : value`”. For example, while an attribute-like syntax “`[[requires: x > 0]]`” is not yet a standard C++ conformant, it has been proposed for use by contract conditions. Furthermore, C++11 attributes in called attributes can use labeled attributes to provide key-value pairs, as in “`[[revert(commit: 0xdeadbeef, reason: "bonkers")]]`”. The *index* field of that abstract reference is an index into the labeled attribute partition. Each entry in that partition is a structure with the following layout

<i>label</i> : Word
<i>attribute</i> : AttrIndex

Figure 14.4: Structure of a labeled attribute

The *label* field represents the word used as *key* in this attribute. The *attribute* field denotes the *value* in this attribute.

Partition name: "attr.labeled".

AttrSort::Called

An *AttrIndex* reference with tag **AttrSort::Called** denotes an attribute using a syntax similar to that of function call. For example, "[[deprecated("out of fad")]]" and "[[gsl::suppress(type.1)]]" are called attributes.

The *index* field of that abstract reference is an index into the called attributes partition. Each entry in that partition is a structure with the following layout

<i>function</i> : AttrIndex
<i>arguments</i> : AttrIndex

Figure 14.5: Structure of a called attribute

The *function* field denotes the attribute appearing in function position, whereas the *arguments* denotes the sequence of attributes making up the supplied argument list.

Partition name: "attr.called".

AttrSort::Expanded

An *AttrIndex* reference with tag **AttrSort::Expanded** denotes an attribute using the ellipsis after an attribute, in a syntax similar to that of pack expansion. For example, "[[fun(args)...]]" is an expanded attribute.

The *index* field of that abstract reference is an index into the expanded attributes partition. Each entry in that partition is a structure with the following layout

<i>operand</i> : AttrIndex

Figure 14.6: Structure of an expanded attribute

The *operand* field denotes the attribute to be expanded.

Partition name: "attr.expanded".

AttrSort::Factored

An *AttrIndex* reference with tag `AttrSort::Factored` denotes an abbreviation form of attribute sequence where each attribute is either a scoped attribute (`AttrSort::Scoped`) or a called attribute (`AttrSort::Called`) where the attribute in function position is a scoped attribute, and all mentioning the same scope word. For example, “[`using build: opt(2), chk`]” is a factored attribute, and is a short-hand for “[`build::opt(2), build::chk`]”. The *index* field of that abstract reference is an index into the factored attributes partition. Each entry in that partition is a structure with the following layout

<i>factor</i> : Word
<i>terms</i> : <i>AttrIndex</i>

The *factor* field represents the common scope word. The field *terms* represents the rest of the comm-separated list of attributes.

Partition name: “`attr.factored`”.

AttrSort::Elaborated

An *AttrIndex* reference with tag `AttrSort::Elaborated` denotes an attribute where the input source-level tokens have been parsed and semantically analyzed, and the resulting expression embedded in the attribute structure. This functionality supports various popular implementation-defined extensions, and also proposed contract-attribute syntaxes such as “[`assert: condition`]” where *condition* is a Boolean expression stating an expectation at a given program point.

The *index* field of that abstract reference is an index into the elaborated attribute partition. Each entry in that partition is a structure with the following layout

<i>expression</i> : <i>ExprIndex</i>

Figure 14.7: Structure of an elaborated attribute

The *expression* field denotes the result of the parsing and semantic analysis of the source-level words making up the attributes. The input source level words are not retained.

Partition name: “`attr.elaborated`”.

AttrSort::Tuple

An *AttrIndex* reference with tag `AttrSort::Tuple` denotes a sequence of comma-separated attributes. It is a general scheme to make a sequence of attributes appear as if it was just an attribute. This representation brings simplicity, flexibility, and generality to the attribute data structures.

The *index* field of that abstract reference is an index into the tuple attribute partition. Each entry in that partition is a structure with the following layout

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 14.8: Structure of a tuple attributes

The *start* designates the position of the first *AttrIndex* value in the attribute heap (§7.1) corresponding to the first abstract reference in the tuple attribute. The *cardinality* represents the number of *AttrIndex* values in the tuple attribute. If this value is zero, then *start* is not meaningful.

Partition name: "attr.tuple".

Chapter 15

Syntax Tree

The syntax tree part of the compiler is still a work in progress. The MSVC compiler front-end is going through a long internal overhaul with a mixture of representations, none satisfactory. The latest being so called “parse trees” that tries to capture the syntax in the input source code as written. Clearly, that is bound to both complexity and instability. The front-end is moving away from “parse trees”, to a more abstract representation of syntax fragments, but none of that work is complete in the MSVC releases yet.

Each syntax fragment in the “parse trees” can be referred by an abstract reference of type *SyntaxTree* defined as follows

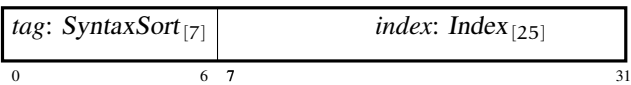


Figure 15.1: SyntaxIndex: Abstract reference of syntax fragment

The type *SyntaxSort* is a set of 7-bit values enumerated as follows

- | | |
|---------------------------------------|----------------------------------|
| 0x00. VendorExtension | 0x0B. ClassSpecifier |
| 0x01. SimpleTypeSpecifier | 0x0C. MemberSpecification |
| 0x02. DecltypeSpecifier | 0x0D. MemberDeclaration |
| 0x03. PlaceholderTypeSpecifier | 0x0E. MemberDeclarator |
| 0x04. TypeSpecifierSeq | 0x0F. AccessSpecifier |
| 0x05. DeclSpecifierSeq | 0x10. BaseSpecifierList |
| 0x06. VirtualSpecifierSeq | 0x11. BaseSpecifier |
| 0x07. NoexceptSpecification | 0x12. Typeld |
| 0x08. ExplicitSpecifier | 0x13. TrailingReturnType |
| 0x09. EnumSpecifier | 0x14. Declarator |
| 0x0A. EnumeratorDefinition | 0x15. PointerDeclarator |
| | 0x16. ArrayDeclarator |
| | 0x17. FunctionDeclarator |

0x18. ArrayOfFunctionDeclarator	0x43. RequirementBody
0x19. ParameterDeclarator	0x44. TypeTemplateParameter
0x1A. InitDeclarator	0x45. TemplateTemplateParameter
0x1B. NewDeclarator	0x46. TypeTemplateArgument
0x1C. SimpleDeclaration	0x47. NonTypeTemplateArgument
0x1D. ExceptionDeclaration	0x48. TemplateParameterList
0x1E. ConditionDeclaration	0x49. TemplateArgumentList
0x1F. StaticAssertDeclaration	0x4A. Templateld
0x20. AliasDeclaration	0x4B. MemInitializer
0x21. ConceptDefinition	0x4C. CtorInitializer
0x22. CompoundStatement	0x4D. LambdaIntroducer
0x23. ReturnStatement	0x4E. LambdaDeclarator
0x24. IfStatement	0x4F. CaptureDefault
0x25. WhileStatement	0x50. SimpleCapture
0x26. DoWhileStatement	0x51. InitCapture
0x27. ForStatement	0x52. ThisCapture
0x28. InitStatement	0x53. AttributedStatement
0x29. RangeBasedForStatement	0x54. AttributedDeclaration
0x2A. ForRangeDeclaration	0x55. AttributeSpecifierSeq
0x2B. LabeledStatement	0x56. AttributeSpecifier
0x2C. BreakStatement	0x57. AttributeUsingPrefix
0x2D. ContinueStatement	0x58. Attribute
0x2E. SwitchStatement	0x59. AttributeArgumentClause
0x2F. GotoStatement	0x5A. Alignas
0x30. DeclarationStatement	0x5B. UsingDeclaration
0x31. ExpressionStatement	0x5C. UsingDeclarator
0x32. TryBlock	0x5D. UsingDirective
0x33. Handler	0x5E. ArrayIndex
0x34. HandlerSeq	0x5F. SEHTry
0x35. FunctionTryBlock	0x60. SEHExcept
0x36. TypeIldListElement	0x61. SEHFinally
0x37. DynamicExceptionSpec	0x62. SEHLeave
0x38. StatementSeq	0x63. TypeTraitIntrinsic
0x39. FunctionBody	0x64. Tuple
0x3A. Expression	0x65. AsmStatement
0x3B. FunctionDefinition	0x66. NamespaceAliasDefinition
0x3C. MemberFunctionDeclaration	0x67. Super
0x3D. TemplateDeclaration	0x68. UnaryFoldExpression
0x3E. RequiresClause	0x69. BinaryFoldExpression
0x3F. SimpleRequirement	0x6A. EmptyStatement
0x40. TypeRequirement	0x6B. StructuredBindingDeclaration
0x41. CompoundRequirement	0x6C. StructuredBindingIdentifier
0x42. NestedRequirement	0x6D. UsingEnumDeclaration

Note: Syntax fragments are used only for the representation of the following syntactic constructs

- concept definition, or any constraint in dependent contexts
- template alias
- lambda in dependent contexts
- exception specification in template alias
- default argument for template parameters – but not for default argument to functions or function templates.

15.1 Auxiliary types

KeywordSyntax

<i>locus:</i> <i>SourceLocation</i>
<i>value:</i> <i>KeywordSort</i>

Figure 15.2: Structure of a keyword syntax

KeywordSort

The type *KeywordSort* is a set of -bit values enumerated as follows

0x00. Nothing	0x07. Default
0x01. Class	0x08. Delete
0x02. Struct	0x09. Mutable
0x03. Union	0x0A. Constexpr
0x04. Public	0x0B. Consteval
0x05. Protected	0x0C. Typename
0x06. Private	

KeywordSort::Nothing No source-level keyword.

KeywordSort::Class Source-level keyword “class”.

KeywordSort::Struct Source-level keyword “struct”.

KeywordSort::Union Source-level keyword “union”.

KeywordSort::Public Source-level keyword “public”.

KeywordSort::Protected Source-level keyword “protected”.

KeywordSort::Private Source-level keyword “private”.

KeywordSort::Default Source-level keyword “default”.

KeywordSort::Delete Source-level keyword “delete”.

KeywordSort::Mutable Source-level keyword “mutable”.

KeywordSort::Constexpr Source-level keyword “constexpr”.

KeywordSort::Consteval Source-level keyword “consteval”.

KeywordSort::Typename Source-level keyword “typename”.

Direction of fold expressions

A fold expression can be left-leaning or right-leaning. The determination of that direction is denoted by values of type *FoldDirection* defined as

```
enum class FoldDirection : uint32_t {
    Unknown,
    Left,
    Right
};
```

with the following meaning

- `FoldDirection::Unknown` indicates that the direction of the fold expression is not yet known.
- `FoldDirection::Left` indicates that the expression is a left fold.
- `FoldDirection::Right` indicates that the expression is a right fold.

15.2 Syntax tree structures

SyntaxSort::VendorExtension

Valid abstract references of this sort have index starting at 1. No structure is defined for this abstract reference as of this version.

Partition name: "syntax.vendor-extension".

SyntaxSort::SimpleTypeSpecifier

<i>type</i> : <i>TypeIndex</i>
<i>expr</i> : <i>ExprIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 15.3: Structure of a simple-type-specifier syntax

Partition name: "syntax.simple-type-specifier".

SyntaxSort::DecltypeSpecifier

<i>expr</i> : <i>ExprIndex</i>
<i>decltype_keyword</i> : <i>SourceLocation</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

Figure 15.4: Structure of the decltype-specifier syntax-tree structure

Partition name: "syntax.decltype-specifier".

SyntaxSort::PlaceholderTypeSpecifier

A *SyntaxIndex* abstract reference with tag `SyntaxSort::PlaceholderTypeSpecifier` designates a placeholder type given by the grammatical element described below. The *index* field is an index into the grammatical placeholder type partition. Each entry in that partition is a structure with the following layout:

<i>type</i> : <i>PlaceholderType</i>
<i>keyword</i> : <i>SourceLocation</i>
<i>locus</i> : <i>SourceLocation</i>

The *type* field is a structure denoting the semantic placeholder type (*PlaceholderType*) as described in [TypeSort::Placeholder](#). The *keyword* field designates the source location of the syntax of the placeholder type (denoted by `auto` or `decltype(auto)`). Finally the *locus* field denotes the source location of the entire syntax for the placeholder type; this source location may be different from that designated by *keyword* if there are additional constraints (as indicated by the use of a concept type).

Partition name: "syntax.placeholder-type-specifier".

SyntaxSort::TypeSpecifierSeq

<i>typename</i> : SyntaxIndex
<i>type</i> : TypeIndex
<i>locus</i> : SourceLocation
<i>qualifiers</i> : Qualifier
<i>unhashed</i> : bool

Partition name: "syntax.type-specifier-seq".

SyntaxSort::DeclSpecifierSeq

<i>type</i> : TypeIndex
<i>typename</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>storage_class</i> : StorageClass
<i>declspec</i> : SentenceIndex
<i>explicit</i> : SyntaxIndex
<i>qualifiers</i> : Qualifier

Partition name: "syntax.decl-specifier-seq".

SyntaxSort::VirtualSpecifierSeq

<i>locus</i> : SourceLocation
<i>final</i> : SourceLocation
<i>override</i> : SourceLocation
<i>pure</i> : bool

Partition name: "syntax.virtual-specifier-seq".

SyntaxSort::NoexceptSpecification

<i>expr</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>left_paren</i> : SourceLocation
<i>right_paren</i> : SourceLocation

Partition name: "syntax.noexcept-specification".

SyntaxSort::ExplicitSpecifier

<i>condition</i> : ExprIndex
<i>locus</i> : SourceLocation
<i>left_paren</i> : SourceLocation
<i>right_paren</i> : SourceLocation

Partition name: "syntax.explicit-specifier".

SyntaxSort::EnumSpecifier

<i>name</i> : ExprIndex
<i>class_key</i> : KeywordSyntax
<i>enumerators</i> : SyntaxIndex
<i>base</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>colon</i> : SourceLocation
<i>left_brace</i> : SourceLocation
<i>right_brace</i> : SourceLocation

Partition name: "syntax.enum-specifier".

SyntaxSort::EnumeratorDefinition

<i>name</i> : TextOffset
<i>initializer</i> : ExprIndex
<i>locus</i> : SourceLocation
<i>equal</i> : SourceLocation
<i>comma</i> : SourceLocation

Partition name: "syntax.enumerator-definition".

SyntaxSort::ClassSpecifier

<i>name</i> : ExprIndex
<i>class_key</i> : KeywordSyntax
<i>bases</i> : SyntaxIndex
<i>members</i> : SyntaxIndex
<i>left_paren</i> : SourceLocation
<i>right_paren</i> : SourceLocation

Partition name: "syntax.class-specifier".

SyntaxSort::MemberSpecification

<i>member_declarations</i> : SyntaxIndex
--

Partition name: "syntax.member-specification".

SyntaxSort::MemberDeclaration

<i>decl_specifiers</i> : SyntaxIndex
<i>declarators</i> : SyntaxIndex
<i>semicolon</i> : SourceLocation

Partition name: "syntax.member-declaration".

SyntaxSort::MemberDeclarator

<i>declarator</i> : SyntaxIndex
<i>constraint</i> : SyntaxIndex
<i>bitwidth</i> : ExprIndex
<i>initializer</i> : ExprIndex
<i>locus</i> : SourceLocation
<i>colon</i> : SourceLocation
<i>comma</i> : SourceLocation

Partition name: "syntax.member-declarator".

SyntaxSort::AccessSpecifier

<i>access</i> : <i>KeywordSyntax</i>
<i>colon</i> : <i>SourceLocation</i>

Partition name: "syntax.access-specifier".

SyntaxSort::BaseSpecifierList

<i>base_specifiers</i> : <i>SyntaxIndex</i>
<i>colon</i> : <i>SourceLocation</i>

Partition name: "syntax.base-specifier-list".

SyntaxSort::BaseSpecifier

<i>designator</i> : <i>ExprIndex</i>
<i>access</i> : <i>KeywordSyntax</i>
<i>virtual</i> : <i>SourceLocation</i>
<i>locus</i> : <i>SourceLocation</i>
<i>virtual</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>

Partition name: "syntax.base-specifier".

SyntaxSort::TypeId

A *SyntaxIndex* value with tag *SyntaxSort::TypeId* designates the source-level syntactic construct *type-id* which is the syntax *type-specified-seq* optionally followed by *abstract-declarator*. The *index* field of that abstract reference is an index into the syntax of *type-id* partition. Each entry of that partition is a structure with the following layout:

<i>type_specifier</i> : <i>SyntaxIndex</i>
<i>abstract_declarator</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>

The *type_specifier* field denotes the *type-specifier-seq* (*SyntaxSort::TypeSpecifierSeq*). The *abstract_declarator* field, if not null, denotes the *abstract-declarator* part (*SyntaxSort::Declarator*). The *locus* designates the source location of the *type-id* construct.

Partition name: "syntax.type-id".

SyntaxSort::TrailingReturnType

A *SyntaxIndex* value with tag `SyntaxSort::TrailingReturnType` denotes the syntax tree of a source-level *trailing-return-type* construct. The *index* field of that abstract reference is an index into the partition of trailing return type syntax tree. Each entry of that partition is a structure with the following layout

<i>target</i> : <i>SyntaxIndex</i>
<i>arrow</i> : <i>SourceLocation</i>

The *target* field denotes the syntax tree of the type specified in *trailing-return-type*. The *arrow* field denotes the source location of the introducing arrow.

Partition name: "syntax.trailing-return-type".

SyntaxSort::Declarator

A *SyntaxIndex* value with tag `SyntaxSort::Declarator` designates a source level syntactic construct that follows the grammar production *declarator* or *abstract-declarator*. The *index* field of that abstract references is an index into the partition of declarator syntax trees. Each entry of that partition is a structure with the following layout

<i>pointer</i> : <i>SyntaxIndex</i>
<i>parenthesized</i> : <i>SyntaxIndex</i>
<i>array_or_function</i> : <i>SyntaxIndex</i>
<i>trailing_target</i> : <i>SyntaxIndex</i>
<i>virtual_specifiers</i> : <i>SyntaxIndex</i>
<i>name</i> : <i>ExprIndex</i>
<i>expander</i> : <i>SourceLocation</i>
<i>locus</i> : <i>SourceLocation</i>
<i>qualifiers</i> : <i>Qualifiers</i>
<i>convention</i> : <i>CallingConvntion</i>
<i>callable</i> : <i>bool</i>

In many ways, many fields of this structure form a discriminated union, meaning that not all of them are simultaneously meaningful and their interpretation are supposed to be mutually exclusive. The *pointer* field, if non null, indicates that this is in fact a pointer-declarator (`SyntaxSort::PointerDeclarator`). The *parenthesized* field, if non null, indicates that outer parenthesis are used as part of the declarator. The *array_or_function* field, if non null, denotes an array (`SyntaxSort::ArrayDeclarator`)

or function declarator ([SyntaxSort::FunctionDeclarator](#)). The *trailing_target*, if non null, denotes a trailing return type ([SyntaxSort::TrailingReturnType](#)) in the declarator. The *virtual_specifiers*, if non null, denotes the *virt-specifier-seq* grammatical element ([SyntaxSort::VirtualSpecifierSeq](#)) in the source level construct. The *name* field, if non null, denotes the name (*id-expression*) introduced by the declarator. The *expander* field, if non null, indicates that the expansion operator `...` was used to introduce the *declarator-id*. The *locus* field denotes the source location of the declarator. The *qualifiers* field denotes the *cv-qualifier-seq* of the declarator. The *convention* field denotes the calling convention specifier in the declarator (if this is a function declarator). The *callable* field indicates that the declarator introduces a callable construct.

Partition name: "syntax.declarator".

SyntaxSort::PointerDeclarator

A *SyntaxIndex* value with tag [SyntaxSort::PointerDeclarator](#) designates a source-level syntactic construct that captures the grammatical production of *ptr-declarator*. The *index* field of that abstract references is an index into the partition of pointer-declarator syntax trees. Each entry of that partition is a structure with the following layout

<i>whole</i> : <i>SyntaxIndex</i>
<i>next</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>sort</i> : <i>PointerDeclaratorSort</i>
<i>qualifiers</i> : <i>Qualifier</i>
<i>convention</i> : <i>CallingConvention</i>
<i>callable</i> : <i>bool</i>

with the type *PointerDeclaratorSort* defining 8-bit values as follows

```
enum class PointerDeclaratorSort : uint8_t {
    None,
    Pointer,
    LvalueReference,
    RvalueReference,
    PointerToMember,
};
```

Partition name: "syntax.pointer-declarator".

SyntaxSort::ArrayDeclarator

A *SyntaxIndex* value with tag `SyntaxSort::ArrayDeclarator` denotes an array declarator syntax tree. The *index* field of that abstract reference is an index into the array declarator syntax tree partition. Each entry of that partition is a structure with the following layout

<i>bound</i> : <i>ExprIndex</i>
<i>left_bracket</i> : <i>SourceLocation</i>
<i>right_bracket</i> : <i>SourceLocation</i>

The *bound* field, if non null, denotes the bound of the array. The *left_bracket* field denotes the source location of the open bracket. The *right_bracket* field denotes the source location of the closing bracket.

Partition name: "syntax.array-declarator".

SyntaxSort::FunctionDeclarator

A *SyntaxIndex* value with tag `SyntaxSort::FunctionDeclarator` denotes a function declarator syntax tree. The *index* field is an index into the function declarator syntax tree partition. Each entry in that partition is a structure with the following layout

<i>parameters</i> : <i>SyntaxIndex</i>
<i>eh_spec</i> : <i>SyntaxIndex</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

The *parameters* denotes the parameter list of the function declarator. The *eh_spec* field, if non null, denotes the exception specification in the function declarator. The *left_paren* field denotes the location of the opening parenthesis of this function declarator. The *right_paren* field denotes the location of the closing parenthesis of this function declarator.

Partition name: "syntax.function-declarator".

SyntaxSort::ArrayOrFunctionDeclarator

A *SyntaxIndex* value with tag `SyntaxSort::ArrayOrFunctionDeclarator` denotes the tower of array or function declarator syntax trees. The *index* field is an index into the partition of array-or-function declarator syntax tree. Each entry of that partition is a structure with the following layout

<i>declarator</i> : <i>SyntaxIndex</i>
<i>next</i> : <i>SyntaxIndex</i>

The *declarator* field denotes the outer declarator, while the *next* field denotes the inner declarator.

Partition name: "syntax.array-or-function-declarator".

SyntaxSort::ParameterDeclarator

<i>decl_specifiers</i> : <i>SyntaxIndex</i>
<i>declarator</i> : <i>SyntaxIndex</i>
<i>default</i> : <i>ExprIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>sort</i> : <i>ParameterSort</i>

Partition name: "syntax.parameter-declarator".

SyntaxSort::InitDeclarator

<i>declarator</i> : <i>SyntaxIndex</i>
<i>constraint</i> : <i>SyntaxIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>comma</i> : <i>SourceLocation</i>

Partition name: "syntax.init-declarator".

SyntaxSort::NewDeclarator

<i>declarator</i> : <i>SyntaxIndex</i>
--

Partition name: "syntax.new-declarator".

SyntaxSort::SimpleDeclaration

<i>decl_specifiers</i> : SyntaxIndex
<i>declarators</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>semicolon</i> : SourceLocation

Partition name: "syntax.simple-declaration".

SyntaxSort::ExceptionDeclaration

<i>type_specifiers</i> : SyntaxIndex
<i>declarator</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>expander</i> : SourceLocation

Partition name: "syntax.exception-declaration".

SyntaxSort::ConditionDeclaration

<i>decl_specifier</i> : SyntaxIndex
<i>initializerion</i> : SyntaxIndex
<i>locus</i> : SourceLocation

Partition name: "syntax.condition-declaration".

SyntaxSort::StaticAssertDeclaration

<i>condition</i> : ExprIndex
<i>message</i> : ExprIndex
<i>locus</i> : SourceLocation
<i>left_paren</i> : SourceLocation
<i>right_paren</i> : SourceLocation
<i>semicolon</i> : SourceLocation
<i>comma</i> : SourceLocation

Partition name: "syntax.static-assert-declaration".

SyntaxSort::AliasDeclaration

<i>name</i> : ExprIndex
<i>aliasee</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>equal</i> : SourceLocation
<i>semicolon</i> : SourceLocation

Partition name: "syntax.alias-declaration".

SyntaxSort::ConceptDefinition

<i>parameters</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>name</i> : TextOffset
<i>initializer</i> : ExprIndex
<i>concept_keyword</i> : SourceLocation
<i>equal</i> : SourceLocation
<i>semicolon</i> : SourceLocation

Partition name: "syntax.concept-definition".

SyntaxSort::CompoundStatement

<i>pragma</i> : SentenceIndex
<i>stmts</i> : SyntaxIndex
<i>left_curly</i> : SourceLocation
<i>right_curly</i> : SourceLocation

Figure 15.5: Structure of a compound statement syntax tree

Partition name: "syntax.compound-statement".

SyntaxSort::ReturnStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>expr</i> : <i>ExprIndex</i>
<i>sort</i> : <i>ReturnSort</i>
<i>return</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.6: Structure of a return statement syntax tree

Partition name: "syntax.return-statement".

```
enum class ReturnSort : uint8_t {
    Return,
    Co_return,
};
```

SyntaxSort::IfStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>initialization</i> : <i>SyntaxIndex</i>
<i>condition</i> : <i>SyntaxIndex</i>
<i>condition</i> : <i>ExprIndex</i>
<i>consequence</i> : <i>SyntaxIndex</i>
<i>alternative</i> : <i>SyntaxIndex</i>
<i>if</i> : <i>SourceLocation</i>
<i>constexpr</i> : <i>SourceLocation</i>
<i>else</i> : <i>SourceLocation</i>

Figure 15.7: Structure of an *if*-statement syntax tree

Partition name: "syntax.if-statement".

SyntaxSort::WhileStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>condition</i> : <i>ExprIndex</i>
<i>body</i> : <i>SyntaxIndex</i>
<i>while</i> : <i>SourceLocation</i>

Figure 15.8: Structure of an *while-statement* syntax tree

Partition name: "syntax.while-statement".

SyntaxSort::DoWhileStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>condition</i> : <i>ExprIndex</i>
<i>body</i> : <i>SynnyaxIndex</i>
<i>do</i> : <i>SourceLocation</i>
<i>while</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.9: Structure of an *do-statement* syntax tree

Partition name: "syntax.do-statement".

SyntaxSort::ForStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>initialization</i> : <i>SyntaxIndex</i>
<i>condution</i> : <i>ExprIndex</i>
<i>continuation</i> : <i>ExprIndex</i>
<i>body</i> : <i>SyntaxIndex</i>
<i>for</i> : <i>SourceLocation</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.10: Structure of an *for-statement* syntax tree

Partition name: "syntax.for-statement".

SyntaxSort::InitStatement

<i>pragma: SentenceIndex</i>
<i>init: SyntaxIndex</i>

Figure 15.11: Structure of an *init-statement* syntax tree

Partition name: "syntax.init-statement".

SyntaxSort::RangeBasedForStatement

<i>pragma: SentenceIndex</i>
<i>init: SyntaxIndex</i>
<i>decl: SyntaxIndex</i>
<i>initializer: SyntaxIndex</i>
<i>body: SyntaxIndex</i>
<i>for: SourceLocation</i>
<i>left_paren: SourceLocation</i>
<i>right_paren: SourceLocation</i>
<i>colon: SourceLocation</i>

Partition name: "syntax.range-based-for-statement".

SyntaxSort::ForRangeDeclaration

<i>specifiers: SyntaxIndex</i>
<i>declarator: SyntaxIndex</i>

Partition name: "syntax.for-range-declaration".

SyntaxSort::LabeledStatement

<i>pragma</i> : <i>SentencedIndex</i>
<i>label</i> : <i>ExprIndex</i>
<i>stmt</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>sort</i> : <i>LabelSort</i>

Partition name: "syntax.labeled-statement".

```
enum class LabelSort : uint8_t {  
    Unknown,  
    Case,  
    Default,  
    Label  
};
```

SyntaxSort::BreakStatement

<i>break</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Partition name: "syntax.break-statement".

SyntaxSort::ContinueStatement

<i>continue</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Partition name: "syntax.continue-statement".

SyntaxSort::SwitchStatement

<i>pragma: SentenceIndex</i>
<i>init: SyntaxIndex</i>
<i>condition: SyntaxIndex</i>
<i>body: SyntaxIndex</i>
<i>switch: SourceLocation</i>

Partition name: "syntax.switch-statement".

SyntaxSort::GotoStatement

<i>pragma: SentenceIndex</i>
<i>target: TextOffset</i>
<i>locus: SourceLocation</i>
<i>label: SourceLocation</i>
<i>semicolon: SourceLocation</i>

Partition name: "syntax.goto-statement".

SyntaxSort::DeclarationStatement

<i>pragma: SentenceIndex</i>
<i>decl: SyntaxIndex</i>

Partition name: "syntax.declaration-statement".

SyntaxSort::ExpressionStatement

<i>pragma: SentenceIndex</i>
<i>expr: ExprIndex</i>
<i>semicolon: SourceLocation</i>

Partition name: "syntax.expression-statement".

SyntaxSort::TryBlock

<i>pragma</i> : <i>SentenceIndex</i>
<i>body</i> : <i>SyntaxIndex</i>
<i>handlers</i> : <i>SyntaxIndex</i>
<i>try</i> : <i>SourceLocation</i>

Partition name: "syntax.try-block".

SyntaxSort::Handler

<i>pragma</i> : <i>SentenceIndex</i>
<i>exception</i> : <i>SyntaxIndex</i>
<i>body</i> : <i>SyntaxIndex</i>
<i>catch</i> : <i>SourceLocaion</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

Partition name: "syntax.handler".

SyntaxSort::HandlerSeq

<i>handlers</i> : <i>SyntaxIndex</i>

Partition name: "syntax.handler-seq".

SyntaxSort::FunctionTryBlock

<i>body</i> : <i>SyntaxIndex</i>
<i>handlers</i> : <i>SyntaxIndex</i>
<i>initializers</i> : <i>SyntaxIndex</i>

Partition name: "syntax.function-try-block".

SyntaxSort::TypeIdListElement

<i>type_id</i> : <i>SyntaxIndex</i>
<i>ellipsis</i> : <i>SourceLocation</i>

Partition name: "syntax.type-id-list-element".

SyntaxSort::DynamicExceptionSpec

<i>type_list</i> : <i>SyntaxIndex</i>
<i>throw</i> : <i>SourceLocation</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>expander</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

Partition name: "syntax.dynamic-exception-spec".

SyntaxSort::StatementSeq

<i>stmts</i> : <i>SyntaxIndex</i>

Partition name: "syntax.statement-seq".

SyntaxSort::FunctionBody

<i>stmts</i> : <i>SyntaxIndex</i>
<i>try_block</i> : <i>SyntaxIndex</i>
<i>initializers</i> : <i>SyntaxIndex</i>
<i>generate</i> : <i>KeywordSyntax</i>
<i>assign</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Partition name: "syntax.function-body".

SyntaxSort::Expression

A *SyntaxIndex* value with tag `SyntaxSort::Expression` designates an expression (§10), typically a non-dependent expression, within a parse tree. The index of that

value designates an entry in the expression syntax partition. Each entry of that partition has the following layout:

<i>expression</i> : <i>ExprIndex</i>

Figure 15.12: Structure of an expression as a parse tree

The *expression* field designates the representation of this expression.

Partition name: "syntax.expression".

SyntaxSort::FunctionDefinition

<i>stmts</i> : <i>SyntaxIndex</i>
<i>try_block</i> : <i>SyntaxIndex</i>
<i>initializers</i> : <i>SyntaxIndex</i>
<i>synthesis</i> : <i>KeywordSyntax</i>
<i>assign</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Partition name: "syntax.function-definition".

SyntaxSort::MemberFunctionDeclaration

<i>definition</i> : <i>SyntaxIndex</i>
--

Partition name: "syntax.member-function-declaration".

SyntaxSort::TemplateDeclaration

<i>parameters</i> : <i>SyntaxIndex</i>
<i>subject</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 15.13: Structure of a template declaration syntax tree

The meaning of the fields is as follows

- *parameters* denotes the *template-parameter* list of this template declaration.

- *subject* denotes the declaration being parameterized by the template parameters.
- *locus* denotes the source location of this declaration.

Partition name: "syntax.template-declaration".

SyntaxSort::RequiresClause

<i>condition: ExprIndex</i>
<i>locus: SourceLocation</i>

Partition name: "syntax.requires-clause".

SyntaxSort::SimpleRequirement

<i>condition: ExprIndex</i>
<i>locus: SourceLocaion</i>

Partition name: "syntax.simple-requirement".

SyntaxSort::TypeRequirement

<i>type: ExprIndex</i>
<i>locus: SourceLocation</i>

Partition name: "syntax.type-requirement".

SyntaxSort::CompoundRequirement

<i>condition: ExprIndex</i>
<i>constraint: ExprIndex</i>
<i>locus: SourceLocation</i>
<i>right_curly: SourceLocation</i>
<i>noexcept: SourceLocaion</i>

Partition name: "syntax.compound-requirement".

SyntaxSort::NestedRequirement

<i>condition</i> : <i>ExprIndex</i>
<i>locus</i> : <i>SourceLocation</i>

Partition name: "syntax.nested-requirement".

SyntaxSort::RequirementBody

<i>requirements</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>right_curly</i> : <i>SourceLocation</i>

Partition name: "syntax.requirement-body".

SyntaxSort::TypeTemplateParameter

<i>name</i> : <i>TextOffset</i>
<i>constraint</i> : <i>SyntaxIndex</i>
<i>argument</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>expander</i> : <i>SourceLocation</i>

Figure 15.14: Structure of a type *template-parameter* syntax tree

The meaning of the fields is as follows:

- *name* denotes the name introduced by this declaration of type *template-parameter*.
- *constraint* denotes any optional type constraint on this parameter.
- *argument* denotes the corresponding default argument (if any).
- *locus* denotes the source location of this declaration.
- *expander*, if non null, denotes the source location of . . . indicating a parameter pack.

Partition name: "syntax.type-template-parameter".

SyntaxSort::TemplateTemplateParameter

<i>name</i> : <i>TextOffset</i>
<i>argument</i> : <i>SyntaxIndex</i>
<i>parameters</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>expander</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>
<i>key</i> : <i>KeywordSyntax</i>

Figure 15.15: Structure of a template *template-parameter* syntax tree

The meaning of the fields is as follows

- *name* denotes the name introduced by this template *template-parameter* declaration.
- *argument*, if non null, denotes the corresponding default argument (if any).
- *parameters* denotes the parameter list associated with this template *template-parameter* declaration.
- *locus* denotes the source location of this declaration.
- *expander*, if non null, denotes the source location of . . . indicating a parameter pack.
- *comma*, if non null, denotes the source location of the separating comma from the next *template-parameter* declaration.
- *key* denotes the keyword (either `typename` or `class`) used to declare this template *template-parameter*.

Partition name: "syntax.template-template-parameter".

SyntaxSort::TypeTemplateArgument

<i>argument</i> : <i>ExprIndex</i>
<i>expander</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>

Figure 15.16: Structure of a type *template-argument*

The meaning of the fields is as follows

- *argument* denotes the type *template-argument*.
- *expander*, if non null, denotes the source location of . . . if this template argument is actually an expansion.
- *comma*, if non null, denotes the source location of the comma separating from the next template argument.

Partition name: "syntax.type-template-argument".

SyntaxSort::NonTypeTemplateArgument

<i>argument</i> : <i>SyntaxIndex</i>
<i>expander</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>

Figure 15.17: Structure of a non-type *template-argument* syntax tree

The meaning of the fields is as follows

- *argument* is an expression that denotes the non-type *template-argument*.
- *expander*, if non null, denotes the source location of . . . if this template argument is actually an expansion.
- *comma*, if non null, denotes the source location of the comma separating from the next template argument.

Partition name: "syntax.non-type-template-argument".

SyntaxSort::TemplateParameterList

<i>parameters</i> : <i>SyntaxIndex</i>
<i>clause</i> : <i>SyntaxIndex</i>
<i>left_angle</i> : <i>SourceLocation</i>
<i>right_angle</i> : <i>SourceLocation</i>

Figure 15.18: Structure of a template parameter list syntax tree

The meaning of the fields is as follows:

- *parameters* denotes the comma-separated sequence of *template-parameter* declarations.

- *left_angle* denotes the source location of the opening < delimiter.
- *right_angle* denotes the source location of the closing > delimiter.

Partition name: "syntax.template-parameter-list".

SyntaxSort::TemplateArgumentList

<i>arguments</i> : SyntaxIndex
<i>left_angle</i> : SourceLocation
<i>right_angle</i> : SourceLocation

Figure 15.19: Structure of a *template-argument-list* syntax tree

The meaning of the fields is as follows:

- *arguments* denotes the comma-separated sequence of template arguments.
- *left_angle* denotes the source location of the opening < delimiter.
- *right_angle* denotes the source location of the closing > delimiter.

Partition name: "syntax.template-argument-list".

SyntaxSort::TemplateId

<i>name</i> : SyntaxIndex
<i>symbol</i> : ExprIndex
<i>arguments</i> : SyntaxIndex
<i>locus</i> : SourceLocation
<i>template</i> : SourceLocation

Figure 15.20: Structure of a *template-id* syntax tree

The meaning of the fields is as follows

- *name*, when not null, designates the *template-name* used in the *template-id* term. This is the case when the template is not known through a declaration.
- *symbol*, when not null, is an expression that denotes the declaration of the primary template that the *template-name* designates.

- *arguments* denotes the angle bracket-enclosed comma-separated sequence of template arguments.
- *locus* denotes the source location of this syntax tree.
- *template*, when not null, indicates the source location of the `template` keyword if used in forming the *template-id* (typically as part of a *qualified-id*).

Partition name: `"syntax.template-id"`.

SyntaxSort::MemInitializer

A *SyntaxIndex* value with tag `SyntaxSort::MemInitializer` represents a reference to a parse tree of *mem-initializer*. The *index* field is an index into the member-initialization partition. Each entry in that partition is a structure with the following layout

<i>member</i> : <i>ExprIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>expander</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>

Figure 15.21: Structure of a *mem-initializer* syntax tree

The meanings of the fields are as follows

- *member* designates the subobject (field or base-class) to be initialized.
- *initialized* designates the expression used to initialize the subobject.
- *expander*, if non-zero, designates the source location of the pack-expansion operator of the *mem-initializer*.
- *comma*, if non-zero, indicates the source location of a comma after the *mem-initializer*.

Partition name: `"syntax.mem-initializer"`.

SyntaxSort::CtorInitializer

A *SyntaxIndex* value with tag `SyntaxSort::CtorInitializer` represents a reference to a parse tree of *ctor-initializer*. The *index* is an index into the constructor initializer partition. Each entry in that partition is a structure with the following layout

<i>initializers</i> : <i>SyntaxIndex</i>
<i>colon</i> : <i>SourceLocation</i>

Figure 15.22: Structure of a *ctor-initializer* syntax tree

The meanings of the fields are as follows

- *initializers* designates the sequence of *mem-initializer* (**SyntaxSort::MemInitializer**) in this *ctor-initializer*.
- *color* designates the source location of the colon in the this *ctor-initializer*.

Partition name: "syntax.ctor-initializer".

SyntaxSort::LambdaIntroducer

A *SyntaxIndex* value with tag `SyntaxSort::LambdaIntroducer` represents a reference to a parse tree of *lambda-introducer*. The *index* is an index into the lambda introducer partition. Each entry in that partition is a structure with the following layout

<i>captures</i> : <i>SyntaxIndex</i>
<i>left_bracket</i> : <i>SourceLocation</i>
<i>right_bracket</i> : <i>SourceLocation</i>

Figure 15.23: Structure of a *lambda-introducer* syntax tree

The meanings of the fields are as follows

- *captures*, if non-null, designates either the *capture-default* (**SyntaxSort::CaptureDefault**) or a sequence of *captures*.
- *left_paren* and *right_paren* designate the source locations of the open bracket and closing brackets, respectively.

Partition name: "syntax.lambda-introducer".

SyntaxSort::LambdaDeclarator

A *SyntaxIndex* value with tag `SyntaxSort::LambdaDeclarator` represents a reference to a parse tree of *lambda-declarator*. The *index* is an index into the lambda declarator partition. Each entry in that partition is a structure with the following layout

<i>parameters</i> : <i>SyntaxIndex</i>
<i>eh_spec</i> : <i>SyntaxIndex</i>
<i>trailing_target</i> : <i>SyntaxIndex</i>
<i>modifier</i> : <i>Keyword</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>
<i>expander</i> : <i>SourceLocation</i>

Figure 15.24: Structure of a *lambda-declarator* syntax tree

The meanings of the fields are as follow

- *parameters*, if non-null, designates the parse tree for the *parameter-declaration-clause*.
- *eh_spec*, if non null, designates the parse tree for the *noexcept-specifier*.
- *trailing_target*, if non-null, designates the parse tree for the *trailing-return-type*.
- *modifier* designates one of the allowed *decl-specifier* in the *decl-specifier-seq* of this *lambda-declarator*.
- *left_paren* and *right_paren*, when non zero, designate the source locations of the opening and closing parentheses enclosing the *parameters*.
- *expander*, if non zero, designates the source location of `...` in this *lambda-declarator*.

Partition name: `"syntax.lambda-declarator"`.

SyntaxSort::CaptureDefault

<i>locus</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>
<i>by_ref</i> : <i>bool</i>

Figure 15.25: Structure of a *capture-default* syntax tree

Partition name: `"syntax.capture-default"`.

SyntaxSort::SimpleCapture

<i>name: ExprIndex</i>
<i>ampersand: SourceLocation</i>
<i>expander: SourceLocation</i>
<i>comma: SourceLocation</i>

Figure 15.26: Structure of a *simple-capture* syntax tree

Partition name: "syntax.simple-capture".

SyntaxSort::InitCapture

<i>name: ExprIndex</i>
<i>initializer: ExprIndex</i>
<i>expander: SourceLocation</i>
<i>ampersand: SourceLocation</i>
<i>comma: SourceLocation</i>

Figure 15.27: Structure of a *init-capture* syntax tree

Partition name: "syntax.init-capture".

SyntaxSort::ThisCapture

<i>locus: SourceLocation</i>
<i>asterisk: SourceLocation</i>
<i>comma: SourceLocation</i>

Figure 15.28: Structure of a *this* or **this* capture syntax tree

Partition name: "syntax.this-capture".

SyntaxSort::AttributedStatement

<i>pragma</i> : <i>SentenceIndex</i>
<i>stmt</i> : <i>SyntaxIndex</i>
<i>attributes</i> : <i>SyntaxIndex</i>

Figure 15.29: Structure of an attributed *statement* syntax tree

Partition name: "syntax.attributed-statement".

SyntaxSort::AttributedDeclaration

<i>locus</i> : <i>SourceLocation</i>
<i>decl</i> : <i>SyntaxIndex</i>
<i>attributes</i> : <i>SyntaxIndex</i>

Figure 15.30: Structure of an attributed *declaration* syntax tree

Partition name: "syntax.attributed-declaration".

SyntaxSort::AttributeSpecifierSeq

<i>attributes</i> : <i>SyntaxIndex</i>
--

Figure 15.31: Structure of a *attribute-specifier-seq* syntax tree

Partition name: "syntax.attribute-specifier-seq".

SyntaxSort::AttributeSpecifier

<i>prefix</i> : <i>SyntaxIndex</i>
<i>attributes</i> : <i>SyntaxIndex</i>
<i>left_brackets</i> : $2 \times \textit{SyntaxIndex}$
<i>right_brackets</i> : $2 \times \textit{SyntaxIndex}$

Figure 15.32: Structure of a *attribute-specifier* syntax tree

Partition name: "syntax.attribute-specifier".

SyntaxSort::AttributeUsingPrefix

<i>scope: ExprIndex</i>
<i>locus: SourceLocation</i>
<i>colon: SourceLocation</i>

Figure 15.33: Structure of an *attribute-using-prefix* syntax tree

Partition name: "syntax.attribute-using-prefix".

SyntaxSort::Attribute

<i>name: ExprIndex</i>
<i>scope: ExprIndex</i>
<i>argument_clause: SyntaxIndex</i>
<i>colons: SourceLocation</i>
<i>expander: SourceLocation</i>
<i>comma: SourceLocaion</i>

Figure 15.34: Structure of an *attribute* syntax tree

Partition name: "syntax.attribute".

SyntaxSort::AttributeArgumentClause

<i>tokens: SentenceIndex</i>
<i>left_paren: SourceLocation</i>
<i>right_paren: SourceLocation</i>

Figure 15.35: Structure of an *attribute-argument-clause* syntax tree

Partition name: "syntax.attribute-argument-clause".

SyntaxSort::Alignas

<i>operand</i> : <i>SyntaxIndex</i>
<i>locus</i> : <i>SourceLocation</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

Figure 15.36: Structure of an *alignas*-specifier syntax tree

Partition name: "syntax.alignas".

SyntaxSort::UsingDeclaration

<i>declarators</i> : <i>SyntaxIndex</i>
<i>keyword</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.37: Structure of a *using-declaration* syntax tree

Partition name: "syntax.using-declaration".

SyntaxSort::UsingDeclarator

<i>qualified_name</i> : <i>ExprIndex</i>
<i>typename_kw</i> : <i>SourceLocation</i>
<i>expander</i> : <i>SourceLocation</i>
<i>comma</i> : <i>SourceLocation</i>

Figure 15.38: Structure of a *using-declarator* syntax tree

Partition name: "syntax.using-declarator".

SyntaxSort::UsingDirective

<i>qualified_name</i> : ExprIndex
<i>using_kw</i> : SourceLocation
<i>namespace_kw</i> : SourceLocation
<i>semicolon</i> : SourceLocation

Figure 15.39: Structure of a *using-directive* syntax tree

Partition name: "syntax.using-directive".

SyntaxSort::ArrayIndex

A *SyntaxIndex* value with tag `SyntaxSort::ArrayIndex` represents a reference to a parse tree of an expression of the form “ary[i]”. The *index* is an index into the array indexing parse tree partition. Each entry in that partition is a structure with the following layout

<i>array</i> : ExprIndex
<i>index</i> : ExprIndex
<i>left_bracket</i> : SourceLocation
<i>right_bracket</i> : SourceLocation

Figure 15.40: Structure of an array indexing expression syntax tree

The meaning of the fields are as follow

- *array* designates the parse tree of the expression denoting the array to index
- *index* designates the parse tree of the entry position
- *left_bracket* and *right_bracket* designate the source locations of the opening and closing brackets in the array indexing expression.

Partition name: "syntax.array-index".

SyntaxSort::SEHTry

<i>body</i> : <i>SyntaxIndex</i>
<i>handler</i> : <i>SyntaxIndex</i>
<i>try_kw</i> : <i>SourceLocation</i>

Figure 15.41: Structure of the MSVC extension SEH __try statement

Partition name: "syntax.seh-try".

SyntaxSort::SEHExcept

<i>condition</i> : <i>ExprIndex</i>
<i>body</i> : <i>SyntaxIndex</i>
<i>except_kw</i> : <i>SourceLocation</i>
<i>left_paren</i> : <i>SourceLocation</i>
<i>right_paren</i> : <i>SourceLocation</i>

Figure 15.42: Structure of the MSVC extension SEH __except handler

Partition name: "syntax.seh-except".

SyntaxSort::SEHFinally

<i>body</i> : <i>SyntaxIndex</i>
<i>finally_kw</i> : <i>SourceLocation</i>

Figure 15.43: Structure of the MSVC extension SEH __finally handler

Partition name: "syntax.seh-finally".

SyntaxSort::SEHLeave

<i>leave_kw</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.44: Structure of the MSVC extension SEH __leave statement

Partition name: "syntax.seh-leave".

SyntaxSort::TypeTraitIntrinsic

<i>arguments:</i> <i>SyntaxIndex</i>
<i>locus:</i> <i>SourceLocation</i>
<i>intrinsic:</i> <i>Operator</i>

Figure 15.45: Structure of an MSVC extension intrinsic expression syntax tree

Partition name: "syntax.type-trait-intrinsic".

SyntaxSort::Tuple

A *SyntaxIndex* value with tag `SyntaxSort::Tuple` represents a reference to sequence or more abstract indices to syntax trees. The *index* field is index into the tuple syntax tree partition. Each entry in that partition is a structure with the following layout

<i>start:</i> <i>Index</i>
<i>cardinality:</i> <i>Cardinality</i>

Figure 15.46: Structure of a tuple syntax tree

with the following meanings for the fields:

- *start* is an index into the syntax tree heap partition. It points to the first syntax tree abstract reference in the tuple.
- *cardinality* denotes the number of syntax tree abstract references in the tuple.

Partition name: "syntax.tuple".

SyntaxSort::AsmStatement

<i>tokens:</i> <i>SentenceIndex</i>
<i>locus:</i> <i>SourceLocation</i>

Figure 15.47: Structure of the MSVC extension inline assembly syntax tree

Partition name: "syntax.asm-statement".

SyntaxSort::NamespaceAliasDefinition

<i>name</i> : ExprIndex
<i>target</i> : ExprIndex
<i>namespace_kw</i> : SourceLocation
<i>assign</i> : SourceLocation
<i>semicolon</i> : SourceLocation

Figure 15.48: Structure of a *namespace-alias-definition* syntax tree

Partition name: "syntax.namespace-alias-definition".

SyntaxSort::Super

<i>locus</i> : SourceLocation

Figure 15.49: Structure of the MSVC extension `__super` expression syntax tree

Partition name: "syntax.super".

SyntaxSort::UnaryFoldExpression

A *SyntaxIndex* value with tag `SyntaxSort::UnaryFoldExpression` represents a reference to a parse tree of a unary *fold-expression*. The *index* is an index into the unary fold expression parse tree partition. Each entry in that partition is a structure with the following layout

<i>direction</i> : FoldDirection
<i>operand</i> : ExprIndex
<i>dyad</i> : DyadicOperator
<i>locus</i> : SourceLocation
<i>eclipsis</i> : SourceLocation
<i>glyph_locus</i> : SourceLocation
<i>right_paren</i> : SourceLocation

Figure 15.50: Structure of a unary *fold-expression* syntax tree

The meanings of the fields are as follow:

- *direction* indicates whether the folding is left-leaning or right-leaning (see §15.1).

- *operand* designates the parse tree of the expression written in this *fold-expression*.
- *dyad* designates the binary operation used for the fold expression.
- *locus* designates the source location of the *fold-expression*.
- *eclipsis* designates the parse tree of the . . .
- *glyph_locus* designates the source location of the operator in the expression.
- *right_paren* designates the source location of the closing parenthesis.

Partition name: "syntax.unary-fold-expression".

SyntaxSort::BinaryFoldExpression

<i>direction</i> : <i>FoldDirection</i>
<i>operands</i> : $2 \times \text{ExprIndex}$
<i>dyad</i> : <i>DyadicOperator</i>
<i>locus</i> : <i>SourceLocation</i>
<i>eclipsis</i> : <i>SourceLocation</i>
<i>glyph_loci</i> : $2 \times \text{SourceLocation}$
<i>right_paren</i> : <i>SourceLocation</i>

Figure 15.51: Structure of a binary *fold-expression* syntax tree

Partition name: "syntax.binary-fold-expression".

SyntaxSort::EmptyStatement

<i>locus</i> : <i>SourceLocation</i>

Figure 15.52: Structure of an empty statement syntax tree

Partition name: "syntax.empty-statement".

SyntaxSort::StructuredBindingDeclaration

<i>locus</i> : <i>SourceLocation</i>
<i>ref</i> : <i>SourceLocation</i>
<i>specifiers</i> : <i>SyntaxIndex</i>
<i>names</i> : <i>SyntaxIndex</i>
<i>initializer</i> : <i>ExprIndex</i>
<i>qualifier</i> : <i>ReferenceQualifier</i>

Figure 15.53: Structure of a structured binding declaration syntax tree

Partition name: "syntax.structured-binding-declaration".

SyntaxSort::StructuredBindingIdentifier

<i>name</i> : <i>ExprIndex</i>
<i>comma</i> : <i>SourceLocation</i>

Figure 15.54: Structure of a structured binding identifier syntax tree

Partition name: "syntax.structured-binding-identifier".

SyntaxSort::UsingEnumDeclaration

<i>name</i> : <i>ExprIndex</i>
<i>using_kw</i> : <i>SourceLocation</i>
<i>enum_kw</i> : <i>SourceLocation</i>
<i>semicolon</i> : <i>SourceLocation</i>

Figure 15.55: Structure of a *using-enum-declaration* syntax tree

Partition name: "syntax.using-enum-declaration".

Chapter 16

Source Location

At various places, especially in the description of declarations (§8), it is necessary to specify locations in input source code. Ideally, there should be distinction between source location as seen by the user (before macro expansions), a span of source location, and location tracking macro expansions. For the time being, source location is described in a fairly simplistic way using the unsophisticated structure *SourceLocation* defined as follows:

<i>line</i> : <i>LineIndex</i>
<i>column</i> : <i>Column</i>

Figure 16.1: Structure of a source location

The *line* field is of type *LineIndex* defined as

```
enum class LineIndex : uint32_t { };
```

Figure 16.2: Definition of type *LineIndex*

A value of this type is an index into the partition of file and line (§16.1).

The *column* field is of type *Column* defined as

```
enum class Column : uint32_t { };
```

Figure 16.3: Definition of type *Column*

A value of this type indicates the position of the character from the beginning of the *line*. Columns are numbered from 0.

16.1 File and line

The file and line source location information is collectively stored in a dedicated partition. Each entry in that partition is a structure with the following components: The

<i>file</i> : <i>NameIndex</i>
<i>line</i> : <i>LineNumber</i>

Figure 16.4: Structure of a file-and-line location information

file field designates the name of the input source file (sec:ifc-source-file-name). The *line* field is the line number in the designated source file. Lines are numbered from 1. A line number is of type *LineNumber* defined as

```
enum class LineNumber : uint32_t { };
```

Figure 16.5: Definition of type *LineNumber*

Partition name: "src.line".

Chapter 17

Traits

A trait is a property of an entity not directly stored in the data structure representing that entity. Traits are stored in associative table partitions. Examples of traits include the deprecation message associated (§17.1) with a declaration, friend declarations (§17.3), the set of specializations of a template (§17.2), etc. Each entry in a (parameterized) trait table is a pair structure of the following form

<i>decl</i> : <i>DeclIndex</i>
<i>trait</i> : <i>T</i>

Figure 17.1: Structure of *AssociatedTrait*<*T*>

where the *decl* field is an abstract reference designating the entity, and the *trait* designates the property associated with the entity. The entries in a trait table are stored by increasing values of the *decl* field.

17.1 Deprecation texts

The type *T* of the *trait* associated with *decl* is *TextOffset*, designating the string literal of the deprecation message.

Partition name: "trait.deprecated".

Note: This representation is subject to change.

17.2 Template specializations

For each template *decl*, the set of declarations for corresponding partial or explicit specializations is stored as a sequence in the scope member partition, and the description of that sequence is given by the template specialization trait.

The type T of the *trait* associated with *decl* is *Sequence* (§2.6) of *Declarations* (Figure 6.3), denoting the sequence of descriptions for each specialization of the template *decl*. The value of the field *trait.start* is an index into the scope member partition (§6.3), and the field *trait.cardinality* gives the number of *Declarations* in that sequence.

Partition name: "trait.specialization".

Note: This representation is subject to change.

17.3 Friendship of a class

For each declaration *decl* of a class type, the set of declared friends of that class type is stored as a sequence in the scope member partition, and the description of that sequence is given by the friend trait.

The type T of the *trait* associated with *decl* is *Sequence* (§2.6) of *Declarations* (Figure 6.3), denoting the sequence of descriptions for each declared friend of *decl*. The value of the field *trait.start* is an index into the scope member partition (§6.3), and the field *trait.cardinality* gives the number of *Declarations* in that sequence.

Partition name: "trait.friend".

Note: This representation is subject to change.

17.4 Function Definition

The trait of a definition of a constexpr function or constructor is represented by a structure of defined as follows

<i>parameters:</i> <i>ChartIndex</i>
<i>initializers:</i> <i>ExprIndex</i>
<i>body:</i> <i>StmtIndex</i>

Figure 17.2: Structure of function definition

The *parameters* field designates the chart of parameters (§13) of this function or constructor. The *initializers* field designates the set of member-initializers, if any, of the constexpr constructor. The *body* field designates the statement body (§11) of the function or constructor.

Note: This structure is subject to change.

Partition name: "trait.mapping-expr".

17.5 Template Alias

MSVC, in its current form, uses syntax trees (§15) to represent declarations of template aliases. For each template alias declaration *decl*, the type *T* of the corresponding *trait* is *SyntaxIndex* (Figure 15.1).

Partition name: "trait.alias-template".

Note: This trait is subject of removal in future releases.

17.6 Declared deduction guides

For each class template *decl* with declared deduction guides, the type *T* of the corresponding *trait* is *DeclIndex* (Figure 8.1). The sort of *trait* is *DeclSort::Tuple* (*DeclSort::Tuple*) when the number of deduction guides is greater than one.

Partition name: "trait.deduction-guides".

17.7 Trailing constraints

For each templated function declaration *decl* with trailing *requires-clause* — hence having a *FunctionTraits::Constrained* (§8.2) — the associated constraint condition is stored (in syntax tree form) in the trailing constraints trait partition. The type *T* of *trait* is *SyntaxIndex* (Figure 15.1).

Partition name: "trait.requires".

Note: This structure is subject to change.

17.8 Declaration attributes

Each declaration *decl* with attributes not understood in the form of *ObjectTraits* or *FunctionTraits* — or more generally attributes of templated declarations — has an entry in the declaration attribute partition. The type *T* of *trait* is *AttrIndex* (Figure 14.1).

Partition name: "trait.attribute".

Note: This representation is subject to change.

17.9 MSVC vendor-specific traits

The MSVC compiler associates a certain set of vendor-specific traits to most declarations. These vendor-specific traits are represented by the 32-bit bitmask type *MsvcTraits* defined in §8.1.

Partition name: `".msvc.trait.vendor-traits"`.

Trait for MSVC UUID

The MSVC compiler allows source-level association of UUID with any non-local declaration. When that occurs, the associated MSVC vendor-specific trait has the `MsvcTraits::Uuid` bit set. The UUID is a 16-byte integer in the MSVC UUID trait table.

Partition name: `".msvc.trait.uuid"`.

Function parameters in functions definitions

There is an infelicity in the current MSVC implementation where function parameter names are available only in defining function declarations. The corresponding parameter names, and associated default arguments, are stored in a trait associated with the declaration, see Figure 17.1. The type of the *trait* field is *ChartIndex* (§13.1) listing the sequence of parameter declarations – name, type, and default argument (if any) – in that definition.

Partition name: `".msvc.trait.named-function-parameters"`.

Attributes on declarations

The MSVC compiler records an attribute on a declaration in this trait table.

Partition name: `".msvc.trait.decl-attrs"`.

Chapter 18

Preprocessing Forms

The header unit specification requires a C++ translator to save the preprocessing state as active at the end of processing a header source file, and reifying that state upon import declaration. The MSVC toolset now requires the selection of conformant preprocessor mode (`/Zc:preprocessor`) when processing header units, either at the header unit construction time or at import time.

18.1 Macro definitions

C++ preprocessor macro definitions are indicated by macro abstract references. This document uses *MacroIndex* as a typed abstract reference to designate a macro definition. Like all abstract references, it is a 32-bit value

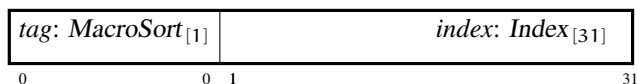


Figure 18.1: *MacroIndex*: Abstract reference of macro definition

The type *MacroSort* is a set of 1-bit values enumerated as follows

0x00. **ObjectLike**

0x01. **FunctionLike**

MacroSort::ObjectLike

<i>locus</i> : <i>SourceLocation</i>
<i>name</i> : <i>TextOffset</i>
<i>body</i> : <i>FormIndex</i>

Figure 18.2: Structure of an object-like macro definition

The field *name* designates the name of the macro, and the field *body* designates its replacement list.

Partition name: "macro.object-like".

MacroSort::FunctionLike

<i>locus</i> : SourceLocation
<i>name</i> : TextOffset
<i>parameters</i> : FormIndex
<i>body</i> : FormIndex
<i>arity</i> : u31
<i>variadic</i> : u1

Figure 18.3: Structure of a function-like macro definition

The field *name* designates the name of the macro. The field *parameters*, when not null, designate the parameter parameter list. The field *body* designates the replacement list of the macro. The field *arity* designates the number of named parameter in the parameter list — this field is in some sense redundant with the *parameters* field. The field *variadic* indicates whether this macro is variadic.

Partition name: "macro.function-like".

18.2 Preprocessing forms

During the processing of input source files from translation phases 1 through 4, syntactic elements (*preprocessing-tokens*) are grouped into *preprocessing forms*, denoted by abstract references of type *FormIndex* with the following layout:

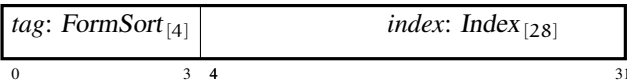


Figure 18.4: *FormIndex*: Abstract reference of preprocessing forms

The type *FormSort* is a set of 4-bit values enumerated as follows

- | | |
|------------------|------------------|
| 0x00. Identifier | 0x04. Operator |
| 0x01. Number | 0x05. Keyword |
| 0x02. Character | 0x06. Whitespace |
| 0x03. String | 0x07. Parameter |

0x08. Stringize	0x0C. Parenthesized
0x09. Catenate	0x0D. Tuple
0x0A. Pragma	0x0E. Junk
0x0B. Header	

Form structures

FormSort::Identifier

A *FormIndex* value with tag `FormSort::Identifier` designates the representation of a *preprocessing-token* that is an *identifier*. The *index* field is an index into the preprocessing identifier form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>TextOffset</i>

Figure 18.5: Structure of an identifier form

The field *locus* designates the source location of this *identifier*. The field *spelling* designates the sequence of characters making up that *identifier*.

Partition name: "pp.ident".

FormSort::Number

A *FormIndex* value with tag `FormSort::Number` designates the representation of a *preprocessing-token* that is a *pp-number*. The *index* field is an index into the preprocessing number form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>TextOffset</i>

Figure 18.6: Structure of a number form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that *pp-number*.

Partition name: "pp.num".

FormSort::Character

A *FormIndex* value with tag `FormSort::Character` designates the representation of a *preprocessing-token* that is a *character-literal* or *user-defined-character-literal*. The

index field is an index into the preprocessing character form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>TextOffset</i>

Figure 18.7: Structure of a character form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that *character-literal* or *user-defined-character-literal*.

Partition name: "pp.char".

FormSort::String

A *FormIndex* value with tag *FormSort::String* designates the representation of a *preprocessing-token* that is a *string-literal* or *user-defined-string-literal*. The *index* field is an index into the preprocessing string form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>TextOffset</i>

Figure 18.8: Structure of a string form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that *string-literal* or *user-defined-string-literal*.

Partition name: "pp.string".

FormSort::Operator

A *FormIndex* value with tag *FormSort::Operator* designates the representation of a *preprocessing-token* that is a *preprocessing-op-or-punc*. The *index* field is an index into the preprocessing operator form partition. Each structure in that partition has the following layout

<i>locus</i> : SourceLocation
<i>spelling</i> : TextOffset
<i>operator</i> : FormOperator

Figure 18.9: Structure of an operator form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that *preprocessing-op-or-punc*. The field *operator* designates the source-level preprocessing interpretation of that form.

Partition name: "pp.op".

FormSort::Keyword

A *FormIndex* value with tag *FormSort::Keyword* designates the representation of a *preprocessing-token* that is *import-keyword*, or *module-keyword*, or *export-keyword*. The *index* field is an index into the preprocessing keyword form partition. Each structure in that partition has the following layout

<i>locus</i> : SourceLocation
<i>spelling</i> : TextOffset

Figure 18.10: Structure of a keyword form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that keyword: *import-keyword*, or *module-keyword*, or *export-keyword*. Note that other standard C++ source-level keywords do not exist during the preprocessing phases.

Partition name: "pp.key".

FormSort::Whitespace

<i>locus</i> : SourceLocation

Figure 18.11: Structure of a whitespace form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that whitespace.

Note: The current version of the MSVC toolset does not emit whitespace forms

Partition name: "pp.space".

FormSort::Parameter

A *FormIndex* value with tag `FormSort::Parameter` designates the representation of a macro parameter. The *index* field is an index into the preprocessing parameter form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>TextOffset</i>

Figure 18.12: Structure of a parameter form

The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up that parameter name.

Partition name: "pp.param".

FormSort::Stringize

A *FormIndex* value with tag `FormSort::Stringize` designates the representation of the application of the stringizing operator (#) to a preprocessing form. The *index* field is an index into the preprocessing stringizing form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>operand</i> : <i>FormIndex</i>

Figure 18.13: Structure of a stringizing form

The field *locus* designates the source location of this form. The field *operand* designates the form that is operand to the stringizing operator.

Partition name: "pp.to-string".

FormSort::Catenate

A *FormIndex* value with tag `FormSort::Catenate` designates the representation of the application of the token catenation operator (##) to two preprocessing forms. The *index* field is an index into the preprocessing catenation form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>first</i> : <i>FormIndex</i>
<i>second</i> : <i>FormIndex</i>

Figure 18.14: Structure of a catenation form

The field *locus* designates the source location of this form. The fields *first* and *second* designate the operands to the catenation operator `##`.

Partition name: `"pp.catenate"`.

FormSort::Pragma

A *FormIndex* value with tag `FormSort::Pragma` designates the representation of the application of the standard `_Pragma` operator or the MSVC extension `__pragma` to a preprocessing form. The *index* field is an index into the preprocessing pragma form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>operand</i> : <i>FormIndex</i>

Figure 18.15: Structure of a pragma form

The field *locus* designates the source location of this form. The field *operand* designates the form that is argument to the pragma operator. If the operand's tag is `FormSort::Tuple` then the pragma is actually the MSVC extension `__pragma`, otherwise the tag must be `FormSort::String` indicating the standard `_Pragma` operator.

Partition name: `"pp.pragma"`.

FormSort::Header

A *FormIndex* value with tag `FormSort::Header` designates the representation of a *preprocessing-token* that is a *header-name*. The *index* field is an index into the preprocessing header-name form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>Textoffset</i>

Figure 18.16: Structure of a header form

The field *locus* designates the source location of this form. The field *spelling* designates the spelling of the header-name that is argument to an `#include` or `import` directive.

Partition name: "pp.header".

Note: The current MSVC toolset release does not emit this form.

FormSort::Parenthesized

A *FormIndex* value with tag `FormSort::Parenthesized` designates the representation of a preprocessing form enclosed in a matching pair of parentheses. The *index* field is an index into the preprocessing parenthesized form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>operand</i> : <i>FormIndex</i>

Figure 18.17: Structure of a parenthesized form

The field *locus* designates the source location of this form. The field *operand* designates the form that is enclosed in the matching parenthesis.

Partition name: "pp.paren".

FormSort::Tuple

A *FormIndex* value with tag `FormSort::Tuple` designates the representation of a sequence of preprocessing forms. The *index* field is an index into the preprocessing tuple form partition. Each structure in that partition has the following layout

<i>start</i> : <i>Index</i>
<i>cardinality</i> : <i>Cardinality</i>

Figure 18.18: Structure of a tuple form

The field *locus* designates the source location of this form. The field *start* designates the position of the *FormIndex* value in the form heap partition that starts the sequence. The field *cardinality* designates the number of *FormIndex* values in the sequence.

Partition name: "pp.tuple".

FormSort::Junk

A *FormIndex* value with tag `FormSort::Junk` designates the representation of an invalid *preprocessing-token*. The *index* field is an index into the preprocessing junk form partition. Each structure in that partition has the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>spelling</i> : <i>Textoffset</i>

Figure 18.19: Structure of a junk form

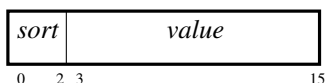
The field *locus* designates the source location of this form. The field *spelling* designates the sequence of characters making up this non-*preprocessing-token*

Partition name: "pp.junk".

Note: The current MSVC toolset does not produce this forms since an IFC is produced only for a successful translation

18.3 Preprocessing operators

Preprocessing form operators (*preprocessing-op-or-punc*) are represented as 16-bit values with the following layout

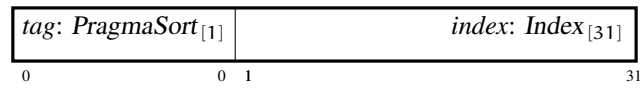
Figure 18.20: *Operator*: Structure of preprocessing form operator

The field *sort* is a 3-bit value of type `WorSort[3]` (§19.2), holding either `WordSort::Punctuator` (§19.2) or `WordSort::Operator` (§19.2). The field *value* is a 13-bit value the interpretation of which is *sort*-dependent: its type is `SourcePunctuator[13]` (Figure 19.6) if *sort* is `WordSort::Punctuator`; otherwise its type is `SourceOperator[13]` (Figure 19.8) if *sort* is `WordSort::Operator`.

18.4 Pragmas

Pragmas are indicated by abstract pragma references. This document uses *PragmaIndex* to designate a typed abstract reference to a pragma. Like all abstract references, it is a 32-bit value

The type *PragmaSort* is a set of 1-bit values enumerated as follows

Figure 18.21: *PragmaIndex*: Abstract reference of pragmas

0x00. VendorExtension

Chapter 19

Words

Ideally, definitions of templates should have an AST representation. At the moment, MSVC lacks AST representation so the body of certain template definitions are represented as sequences of MSVC-internal “tokens” that are replayed every time an instantiation is needed. This part of the IFC will gradually vanish in favor of the semantics graph representation used for non-template entities. Also, these MSVC-internal tokens are nothing like ISO C++ tokens; consequently, this document uses the term *words* (§19.2) to describe them.

19.1 Sentences

Each MSVC-internal token stream is represented as a *sentence*, a sequence of words (§19.2). Each sentence is referenced by a value of type

```
enum class SentenceIndex : uint32_t { };
```

Figure 19.1: Definition of type *SentenceIndex*

A valid value of type *SentenceIndex* denotes an index into the sentence partition described below. A *SentenceIndex* value 0 indicates a missing sentence, not an empty sentence. Valid *SentenceIndex* values start at 1.

Each entry of the the sentence partition has the following structure:

<i>start</i> : <i>WordIndex</i>
<i>cardinality</i> : <i>Cardinality</i>
<i>locus</i> : <i>SourceLocation</i>

Figure 19.2: Structure of a sentence

The *start* field is an index of type

```
enum class WordIndex : uint32_t { };
```

Figure 19.3: Definition of type *WordIndex*

into the word partition (§19.2). It points to the first word in that sentence. The *cardinality* field denotes the numbers of words in the sentence. The *locus* field denotes the source location of the sentence.

Partition name: "src.sentence".

19.2 Words

Each entry in the word partition is a 16-byte structure with the following layout

<i>locus</i> : <i>SourceLocation</i>
<i>index</i> : <i>Index</i>
<i>value</i> : <i>u16</i>
<i>sort</i> : <i>WordSort</i>
<padding>: <i>u8</i>

Figure 19.4: Structure of a word

The *locus* field denotes the location of the word. The *index* field is a 16-bit value, the interpretation of which depends on the *sort* field, which itself is a 8-bit value. In this document, we use the phrase *word* to emphasize that these "tokens" are not tokens in ISO C++ standards sense. They are MSVC-internal parser (complex) data structures, scheduled for removal.

Partition name: "src.word".

Word structures

The type *WordSort* is a set of 8-bit values enumerated as follows

0x00. Unknown	0x04. Operator
0x01. Directive	0x05. Keyword
0x02. Punctuator	0x06. Identifier
0x03. Literal	

The interpretation of the *value* field of a word (§19.4) depends of the value its *sort* field as follows:

- *WordSort::Unknown*: no particular meaning
- *WordSort::Directive*: *value* is of type *SourceDirective* (§19.2)

- WordSort::Punctuator: *value* is of type *SourcePunctuator* (§19.2)
- WordSort::Literal: *value* is of type *SourceLiteral* (§19.2)
- WordSort::Operator: *value* is of type *SourceOperator* (§19.2)
- WordSort::Keyword: *value* is of type *SourceKeyword* (§19.2)
- WordSort::Identifier: *value* is of type *SourceIdentifier* (§19.2)

WordSort::Unknown

A word (Figure 19.4) with the *sort* field that holds WordSort::Unknown shall not be produced.

WordSort::Directive

A word (Figure 19.4) with the *sort* field that holds WordSort::Directive is a *directive word*. The corresponding *value* field is of type

```
enum class SourceDirective : uint16_t { };
```

Figure 19.5: Definition of type *SourceDirective*

Valid values of type *SourceDirective* are enumerated as follows

0x1FFF. <i>Msvc</i>	0x2010. <i>MsvcPragmaEndregion</i>
0x2000. <i>MsvcPragmaPush</i>	0x2011. <i>MsvcPragmaExecutionCharacterSet</i>
0x2001. <i>MsvcPragmaPop</i>	0x2012. <i>MsvcPragmaFenvAccess</i>
0x2002. <i>MsvcDirectiveStart</i>	0x2013. <i>MsvcPragmaFileHash</i>
0x2003. <i>MsvcDirectiveEnd</i>	0x2014. <i>MsvcPragmaFloatControl</i>
0x2004. <i>MsvcPragmaAllocText</i>	0x2015. <i>MsvcPragmaFpContract</i>
0x2005. <i>MsvcPragmaAutoInline</i>	0x2016. <i>MsvcPragmaFunction</i>
0x2006. <i>MsvcPragmaBssSeg</i>	0x2017. <i>MsvcPragmaBGI</i>
0x2007. <i>MsvcPragmaCheckStack</i>	0x2018. <i>MsvcPragmaIdent</i>
0x2008. <i>MsvcPragmaCodeSeg</i>	0x2019. <i>MsvcPragmaImplementationKey</i>
0x2009. <i>MsvcPragmaComment</i>	0x201A. <i>MsvcPragmaIncludeAlias</i>
0x200A. <i>MsvcPragmaComponent</i>	0x201B. <i>MsvcPragmaInitSeg</i>
0x200B. <i>MsvcPragmaConform</i>	0x201C. <i>MsvcPragmaInlineDepth</i>
0x200C. <i>MsvcPragmaConstSeg</i>	0x201D. <i>MsvcPragmaInlineRecursion</i>
0x200D. <i>MsvcPragmaDataSeg</i>	0x201E. <i>MsvcPragmaIntrinsic</i>
0x200E. <i>MsvcPragmaDeprecated</i>	0x201F. <i>MsvcPragmaLoop</i>
0x200F. <i>MsvcPragmaDetectMismatch</i>	0x2020. <i>MsvcPragmaMakePublic</i>
	0x2021. <i>MsvcPragmaManaged</i>
	0x2022. <i>MsvcPragmaMessage</i>

0x2023. MsvcPragmaOMP	0x202E. MsvcPragmaSegment
0x2024. MsvcPragmaOptimize	0x202F. MsvcPragmaSetlocale
0x2025. MsvcPragmaPack	0x2030. MsvcPragmaStartMapRegion
0x2026. MsvcPragmaPointerToMembers	0x2031. MsvcPragmaStopMapRegion
0x2027. MsvcPragmaPopMacro	0x2032. MsvcPragmaStrictGSCheck
0x2028. MsvcPragmaPrefast	0x2033. MsvcPragmaSystemHeader
0x2029. MsvcPragmaPushMacro	0x2034. MsvcPragmaUnmanaged
0x202A. MsvcPragmaRegion	0x2035. MsvcPragmaVtordisp
0x202B. MsvcPragmaRuntimeChecks	0x2036. MsvcPragmaWarning
0x202C. MsvcPragmaSameSeg	0x2037. MsvcPragmaP0include
0x202D. MsvcPragmaSection	0x2038. MsvcPragmaP0line

It should be noted that MSVC transforms preprocessor level `#pragma` directives to post-preprocessor “directives” introduced by the `__pragma` keyword to be processed by the parser.

SourceDirective::Msvc No directive word of this value shall be produced. All MSVC-specific directive words (which all directives are at this time of writing) have values greater than this.

SourceDirective::MsvcPragmaPush

SourceDirective::MsvcPragmaPop

SourceDirective::MsvcDirectiveStart

SourceDirective::MsvcDirectiveEnd

SourceDirective::MsvcPragmaAllocText

SourceDirective::MsvcPragmaAutoInline

SourceDirective::MsvcPragmaBssSeg

SourceDirective::MsvcPragmaCheckStack

SourceDirective::MsvcPragmaCodeSeg

SourceDirective::MsvcPragmaComment

SourceDirective::MsvcPragmaComponent

SourceDirective::MsvcPragmaConform

SourceDirective::MsvcPragmaConstSeg

SourceDirective::MsvcPragmaDataSeg

SourceDirective::MsvcPragmaDeprecated

SourceDirective::MsvcPragmaDetectMismatch

SourceDirective::MsvcPragmaEndregion

SourceDirective::MsvcPragmaExecutionCharacterSet

SourceDirective::MsvcPragmaFenvAccess

SourceDirective::MsvcPragmaFileHash

SourceDirective::MsvcPragmaFloatControl

SourceDirective::MsvcPragmaFpContract

SourceDirective::MsvcPragmaFunction

SourceDirective::MsvcPragmaBGJ

SourceDirective::MsvcPragmalIdent

SourceDirective::MsvcPragmalImplementationKey

SourceDirective::MsvcPragmaIncludeAlias

SourceDirective::MsvcPragmalInitSeg

SourceDirective::MsvcPragmaInlineDepth

SourceDirective::MsvcPragmaInlineRecursion

SourceDirective::MsvcPragmaIntrinsic

SourceDirective::MsvcPragmaLoop

SourceDirective::MsvcPragmaMakePublic

SourceDirective::MsvcPragmaManaged

SourceDirective::MsvcPragmaMessage

SourceDirective::MsvcPragmaOMP

SourceDirective::MsvcPragmaOptimize

SourceDirective::MsvcPragmaPack

SourceDirective::MsvcPragmaPointerToMembers

SourceDirective::MsvcPragmaPopMacro

SourceDirective::MsvcPragmaPrefast

SourceDirective::MsvcPragmaPushMacro

SourceDirective::MsvcPragmaRegion

SourceDirective::MsvcPragmaRuntimeChecks

SourceDirective::MsvcPragmaSameSeg

SourceDirective::MsvcPragmaSection

SourceDirective::MsvcPragmaSegment

SourceDirective::MsvcPragmaSetlocale

SourceDirective::MsvcPragmaStartMapRegion

SourceDirective::MsvcPragmaStopMapRegion

SourceDirective::MsvcPragmaStrictGSCheck

SourceDirective::MsvcPragmaSystemHeader

SourceDirective::MsvcPragmaUnmanaged

SourceDirective::MsvcPragmaVtordisp

SourceDirective::MsvcPragmaWarning

SourceDirective::MsvcPragmaP0include

SourceDirective::MsvcPragmaP0line

WordSort::Punctuator

A word (Figure 19.4) with the *sort* field that holds **WordSort::Punctuator** is a *punctuator word*. The corresponding *value* field is of type

```
enum class SourcePunctuator : uint16_t { };
```

Figure 19.6: Definition of type *SourcePunctuator*

Values of type *SourcePunctuator* are classified in two groups.

0x00. Unknown	0x06. RightBrace
0x01. LeftParenthesis	0x07. Colon
0x02. RightParenthesis	0x08. Question
0x03. LeftBracket	0x09. Semicolon
0x04. RightBracket	0x0A. ColonColon
0x05. LeftBrace	

and those with values greater than 0x1FFF

0x1FFF. Msvc	0x2003. MsvcNestedTemplateStart
0x2000. MsvcZeroWidthSpace	0x2004. MsvcDefaultArgumentStart
0x2001. MsvcEndOfPhrase	0x2005. MsvcAlignasEdictStart
0x2002. MsvcFullStop	0x2006. MsvcDefaultInitStart

SourcePunctuator::Unknown No punctuator word of this value shall be produced.

SourcePunctuator::LeftParenthesis A punctuator word for C++ source level “(”.

SourcePunctuator::RightParenthesis A punctuator word for C++ source level “)”.

SourcePunctuator::LeftBracket A punctuator word for C++ source level “[”.

SourcePunctuator::RightBracket A punctuator word for C++ source level “]”.

SourcePunctuator::LeftBrace A punctuator word for C++ source level “{”.

SourcePunctuator::RightBrace A punctuator word for C++ source level “}”.

SourcePunctuator::Colon A punctuator word for C++ source level “:”.

SourcePunctuator::Question A punctuator word for C++ source level “?”.

SourcePunctuator::Semicolon A punctuator word for C++ source level “;”.

SourcePunctuator::ColonColon A punctuator word for C++ source level “::”.

SourcePunctuator::Msvc No punctuator word of this value shall be produced. All MSVC-specific punctuators have values greater than this.

SourcePunctuator::MsvcZeroWidthSpace A punctuator word of this value acts effectively as a whitespace. Its practical effect is to record a source location position within a sentence.

SourcePunctuator::MsvcEndOfPhrase A punctuator word of this value terminates a word sequence to be parsed or interpreted independently; typically this is the case of default arguments or default member initializers or templated definitions.

SourcePunctuator::MsvcFullStop A punctuator word of this value ends the parsing of the current translation unit.

SourcePunctuator::MsvcNestedTemplateStart A punctuator word of this value designates the start of template declaration that is itself in a templated lexical scope.

SourcePunctuator::MsvcDefaultArgumentStart A punctuator word of this value indicates the beginning of the words comprising a default argument for function parameters.

SourcePunctuator::MsvcAlignasEdictStart A punctuator word of this value marks the start of the words comprising the operands of the `alignas` keyword.

SourcePunctuator::MsvcDefaultInitStart A punctuator word of this value marks the start of the words comprising the initializer for default member initialization.

WordSort::Literal

A word (Figure 19.4) with the *sort* field that holds `WordSort::Literal` is a *literal word*. The corresponding *value* field is of type


```
enum class SourceLiteral : uint16_t { };
```

Figure 19.7: Definition of type *SourceLiteral*

Values of *SourceLiteral* are classified in two groups.

0x00. Unknown	0x02. String
0x01. Scalar	0x03. DefinedString

and those with values greater than 0x1FFF

0x1FFF. Msvc	0x2003. MsvcResolvedType
0x2000. MsvcFunctionNameMacro	0x2004. MsvcDefinedConstant
0x2001. MsvcStringPrefixMacro	0x2005. MsvcCastTargetType
0x2002. MsvcBinding	

SourceLiteral::Unknown No literal word of this value shall be produced.

SourceLiteral::Scalar A literal word of this value denotes a C++ source-level scalar literal. The *index* field is of type *ExprIndex* and designates that literal expression (§10.1).

SourceLiteral::String A literal word of this value denotes a C++ source level string literal. The *index* index is of type *StringIndex* (Figure 10.19), and designates that string literal expression.

SourceLiteral::DefinedString A literal word of this value denotes a C++ source level user-defined string literal. The *index* index is of type *StringIndex* (Figure 10.19), and designates that user-defined string literal expression.

SourceLiteral::Msvc There is no literal word of this value. All MSVC-specific literal words have values greater than this value.

SourceLiteral::MsvcFunctionNameMacro A literal word of this value denotes any of the special function-local macros `__func__`, `__FUNCTION__`, `__FUNCDNAME__`, and `__FUNCSIG__`. The *index* field is of type *TextOffset* and designates the actual special macro name.

SourceLiteral::MsvcStringPrefixMacro A literal word of this value denotes any of the special prefix macros `__LPREFIX`, `__lPREFIX`, `__UPREFIX`, and `__uPREFIX`. The *index* field is of type *TextOffset* and designates the actual special macro name.

SourceLiteral::MsvcBinding A literal word of this value denotes a C++ source-level identifier bound to declaration in the context of a templated definition – that binding designates the result of the first phase of 2-phase name lookup up as applied to templated definitions. The *index* field is of type *ExprIndex* and references (§10.1) the declarations bound to that identifier.

SourceLiteral::MsvcResolvedType A literal word denoting a fully resolved type at the C++ source-level. The *index* field is of type *TypeIndex*.

SourceLiteral::MsvcDefinedConstant A literal word denoting a user-defined constant literal at the C++ source-level. The *index* field is of type *ExprIndex* and refers to a (§10.1) of two elements containing the literal constant value as the first element and the literal suffix identifier as the second.

SourceLiteral::MsvcCastTargetType A literal word denoting a fully resolved type as the target type of a functional cast expression at the C++ source-level. The *index* field is of type *TypeIndex*.

WordSort::Operator

A word (Figure 19.4) with the *sort* field that holds **WordSort::Operator** is an *operator word*. The corresponding *value* field is of type

```
enum class SourceOperator : uint16_t { };
```

Figure 19.8: Definition of type *SourceOperator*

Values of type *SourceOperator* are enumerated as follows

0x00. Unknown	0x0E. Ampersand
0x01. Equal	0x0F. PlusPlus
0x02. Comma	0x10. DashDash
0x03. Exclaim	0x11. Less
0x04. Plus	0x12. LessEqual
0x05. Dash	0x13. Greater
0x06. Star	0x14. GreaterEqual
0x07. Slash	0x15. EqualEqual
0x08. Percent	0x16. ExclaimEqual
0x09. LeftChevron	0x17. Diamond
0x0A. RightChevron	0x18. PlusEqual
0x0B. Tilde	0x19. DashEqual
0x0C. Caret	0x1A. StarEqual
0x0D. Bar	0x1B. SlashEqual
	0x1C. PercentEqual

0x1D. AmpersandEqual	0x23. BarBar
0x1E. BarEqual	0x24. Ellipsis
0x1F. CaretEqual	0x25. Dot
0x20. LeftChevronEqual	0x26. Arrow
0x21. RightChevronEqual	0x27. DotStar
0x22. AmpersandAmpersand	0x28. ArrowStar

The use of an operator at C++ source-level may be bound to a set of declarations in the context of a templated definition – that binding designates the result of the first phase of 2-phase name lookup up as applied to templated definitions. The *index* field is of type *ExprIndex* and references (§10.1) the declarations bound to that operator.

SourceOperator::Unknown No operator word of this value shall be produced.

SourceOperator::Equal An operator word for “=”.

SourceOperator::Comma An operator word for “,”.

SourceOperator::Exclaim An operator word for “!”.

SourceOperator::Plus An operator word for “+”.

SourceOperator::Dash An operator word for “-”.

SourceOperator::Star An operator word for “*”.

SourceOperator::Slash An operator word for “/”.

SourceOperator::Percent An operator word for “%”.

SourceOperator::LeftChevron An operator word for “<<”.

SourceOperator::RightChevron An operator word for “>>”.

SourceOperator::Tilde An operator word for “~”.

SourceOperator::Caret An operator word for “^”.

SourceOperator::Bar An operator word for “|”.

SourceOperator::Ampersand An operator word for “&”.

SourceOperator::PlusPlus An operator word for “++”.

SourceOperator::DashDash An operator word for “--”.

SourceOperator::Less An operator word for “<”.

SourceOperator::LessEqual An operator word for “<=”.

SourceOperator::Greater An operator word for “>”.

SourceOperator::GreaterEqual An operator word for “>=”.

SourceOperator::EqualEqual An operator word for “==”.

SourceOperator::ExclaimEqual An operator word for “!=”.

SourceOperator::Diamond An operator word for “<=>”.

SourceOperator::PlusEqual An operator word for “+=”.

SourceOperator::DashEqual An operator word for “-=”.

SourceOperator::StarEqual An operator word for “*=”.

SourceOperator::SlashEqual An operator word for “/=”.

SourceOperator::PercentEqual An operator word for “%=”.

SourceOperator::AmpersandEqual An operator word for “&=”.

SourceOperator::BarEqual An operator word for “|=”.

SourceOperator::CaretEqual An operator word for “^=”.

SourceOperator::LeftChevronEqual An operator word for “<<=”.

SourceOperator::RightChevronEqual An operator word for “>>=”.

SourceOperator::AmpersandAmpersand An operator word for “&&”.

SourceOperator::BarBar An operator word for “||”.

SourceOperator::Ellipsis An operator word for “...”.

SourceOperator::Dot An operator word for “.”.

SourceOperator::Arrow An operator word for “->”.

SourceOperator::DotStar An operator word for “.*”.

SourceOperator::ArrowStar An operator word for “->*”.

WordSort::Keyword

A word (Figure 19.4) with the *sort* field that holds **WordSort::Keyword** is a *reserved word*. The corresponding *value* field is of type

```
enum class SourceKeyword : uint16_t { };
```

Figure 19.9: Definition of type *SourceKeyword*

Values of type *SourceKeyword* are classified in two groups.

0x00. Unknown	0x14. Continue
0x01. Alignas	0x15. CoAwait
0x02. Alignof	0x16. CoReturn
0x03. Asm	0x17. CoYield
0x04. Auto	0x18. Decltype
0x05. Bool	0x19. Default
0x06. Break	0x1A. Delete
0x07. Case	0x1B. Do
0x08. Catch	0x1C. Double
0x09. Char	0x1D. DynamicCast
0x0A. Char8T	0x1E. Else
0x0B. Char16T	0x1F. Enum
0x0C. Char32T	0x20. Explicit
0x0D. Class	0x21. Export
0x0E. Concept	0x22. Extern
0x0F. Const	0x23. False
0x10. Consteval	0x24. Float
0x11. Constexpr	0x25. For
0x12. Constinit	0x26. Friend
0x13. ConstCast	0x27. Generic
	0x28. Goto
	0x29. If

0x2A. Inline	0x40. StaticAssert
0x2B. Int	0x41. StaticCast
0x2C. Long	0x42. Struct
0x2D. Mutable	0x43. Switch
0x2E. Namespace	0x44. Template
0x2F. New	0x45. This
0x30. Noexcept	0x46. ThreadLocal
0x31. Nullptr	0x47. Throw
0x32. Operator	0x48. True
0x33. Pragma	0x49. Try
0x34. Private	0x4A. Typedef
0x35. Protected	0x4B. Typeid
0x36. Public	0x4C. Typename
0x37. Register	0x4D. Union
0x38. ReinterpretCast	0x4E. Unsigned
0x39. Requires	0x4F. Using
0x3A. Restrict	0x50. Virtual
0x3B. Return	0x51. Void
0x3C. Short	0x52. Volatile
0x3D. Signed	0x53. WcharT
0x3E. Sizeof	0x54. While
0x3F. Static	

and those with values greater than 0x1FFF

0x1FFF. Msvc	0x200C. MsvcForceinline
0x2000. MsvcAsm	0x200D. MsvcHook
0x2001. MsvcAssume	0x200E. MsvcIdentifier
0x2002. MsvcAlignof	0x200F. MsvcIfExists
0x2003. MsvcBased	0x2010. MsvcIfNotExists
0x2004. MsvcCdecl	0x2011. MsvcInt8
0x2005. MsvcCircall	0x2012. MsvcInt16
0x2006. MsvcDeclspec	0x2013. MsvcInt32
0x2007. MsvcEabi	0x2014. MsvcInt64
0x2008. MsvcEvent	0x2015. MsvcInt128
0x2009. MsvcSehExcept	0x2016. MsvcInterface
0x200A. MsvcFastcall	0x2017. MsvcLeave
0x200B. MsvcSehFinally	0x2018. MsvcMultipleInheritance
	0x2019. MsvcNullptr
	0x201A. MsvcNovtordisp

0x201B. MsvcPragma	0x2040. MsvclsStandardLayout
0x201C. MsvcPtr32	0x2041. MsvclsLiteralType
0x201D. MsvcPtr64	0x2042. MsvclsTriviallyMoveConstructible
0x201E. MsvcRestrict	0x2043. MsvcHasTrivialMoveAssign
0x201F. MsvcSingleInheritance	0x2044. MsvclsTriviallyMoveAssignable
0x2020. MsvcSptr	0x2045. MsvclsNothrowMoveAssignable
0x2021. MsvcStdcall	0x2046. MsvclsConstructible
0x2022. MsvcSuper	0x2047. MsvcUnderlyingType
0x2023. MsvcThiscall	0x2048. MsvclsTriviallyAssignable
0x2024. MsvcSehTry	0x2049. MsvclsNothrowAssignable
0x2025. MsvcUptr	0x204A. MsvclsDestructible
0x2026. MsvcUuidof	0x204B. MsvclsNothrowDestructible
0x2027. MsvcUnaligned	0x204C. MsvclsAssignable
0x2028. MsvcUnhook	0x204D. MsvclsAssignableNocheck
0x2029. MsvcVectorcall	0x204E. MsvcHasUniqueObjectRepresentations
0x202A. MsvcVirtualInheritance	0x204F. MsvclsAggregate
0x202B. MsvcW64	0x2050. MsvcBuiltinAddressOf
0x202C. MsvclsClass	0x2051. MsvcBuiltinOffsetOf
0x202D. MsvclsUnion	0x2052. MsvcBuiltinBitCast
0x202E. MsvclsEnum	0x2053. MsvcBuiltinIsLayoutCompatible
0x202F. MsvclsPolymorphic	0x2054. MsvcBuiltinIsPointerInterconvertibleBaseOf
0x2030. MsvclsEmpty	0x2055. MsvcBuiltinIsPointerInterconvertibleWithClass
0x2031. MsvcHasTrivialConstructor	0x2056. MsvcBuiltinIsCorrespondingMember
0x2032. MsvclsTriviallyConstructible	0x2057. MsvclsRefClass
0x2033. MsvclsTriviallyCopyConstructible	0x2058. MsvclsValueClass
0x2034. MsvclsTriviallyCopyAssignable	0x2059. MsvclsSimpleValueClass
0x2035. MsvclsTriviallyDestructible	0x205A. MsvclsInterfaceClass
0x2036. MsvcHasVirtualDestructor	0x205B. MsvclsDelegate
0x2037. MsvclsNothrowConstructible	0x205C. MsvclsFinal
0x2038. MsvclsNothrowCopyConstructible	0x205D. MsvclsSealed
0x2039. MsvclsNothrowCopyAssignable	0x205E. MsvcHasFinalizer
0x203A. MsvclsPod	0x205F. MsvcHasCopy
0x203B. MsvclsAbstract	0x2060. MsvcHasAssign
0x203C. MsvclsBaseOf	0x2061. MsvcHasUserDestructor
0x203D. MsvclsConvertibleTo	0x2062. MsvcPackCardinality
0x203E. MsvclsTrivial	0x2063. MsvcConfusedSizeof
0x203F. MsvclsTriviallyCopyable	0x2064. MsvcConfusedAlignas

In some circumstances, the use of a keyword (e.g. `new` in a *new-expression*) at C++ source-level may be bound to a set of declarations in the context of a templated definition – that binding designates the result of the first phase of 2-phase name lookup up as applied to templated definitions. The *index* field is of type *ExprIndex* and references (§10.1) the declarations bound to that keyword.

SourceKeyword::Unknown No reserved word of this word shall be produced.

SourceKeyword::Alignas A reserved word for “alignas”.

SourceKeyword::Alignof A reserved word for “alignof”.

SourceKeyword::Asm A reserved word for “asm”.

SourceKeyword::Auto A reserved word for “auto”.

SourceKeyword::Bool A reserved word for “bool”.

SourceKeyword::Break A reserved word for “break”.

SourceKeyword::Case A reserved word for “case”.

SourceKeyword::Catch A reserved word for “catch”.

SourceKeyword::Char A reserved word for “char”.

SourceKeyword::Char8T A reserved word for “char8_t”.

SourceKeyword::Char16T A reserved word for “char16_t”.

SourceKeyword::Char32T A reserved word for “char32_t”.

SourceKeyword::Class A reserved word for “class”.

SourceKeyword::Concept A reserved word for “concept”.

SourceKeyword::Const A reserved word for “const”.

SourceKeyword::Consteval A reserved word for “constexpr”.

SourceKeyword::Constexpr A reserved word for “constexpr”.

SourceKeyword::Constinit A reserved word for “constinit”.

SourceKeyword::ConstCast A reserved word for “const_cast”.

SourceKeyword::Continue A reserved word for “continue”.

SourceKeyword::CoAwait A reserved word for “co_await”.

SourceKeyword::CoReturn A reserved word for “co_return”.

SourceKeyword::CoYield A reserved word for “co_yield”.

SourceKeyword::Decltype A reserved word for “decltype”.

SourceKeyword::Default A reserved word for “default”.

SourceKeyword::Delete A reserved word for “delete”.

SourceKeyword::Do A reserved word for “do”.

SourceKeyword::Double A reserved word for “double”.

SourceKeyword::DynamicCast A reserved word for “dynamic_cast”.

SourceKeyword::Else A reserved word for “else”.

SourceKeyword::Enum A reserved word for “enum”.

SourceKeyword::Explicit A reserved word for “explicit”.

SourceKeyword::Export A reserved word for “export”.

SourceKeyword::Extern A reserved word for “extern”.

SourceKeyword::False A reserved word for “false”.

SourceKeyword::Float A reserved word for “float”.

SourceKeyword::For A reserved word for “for”.

SourceKeyword::Friend A reserved word for “friend”.

SourceKeyword::Generic A reserved word for “_Generic”.

SourceKeyword::Goto A reserved word for “goto”.

SourceKeyword::If A reserved word for “if”.

SourceKeyword::Inline A reserved word for “inline”.

SourceKeyword::Int A reserved word for “int”.

SourceKeyword::Long A reserved word for “long”.

SourceKeyword::Mutable A reserved word for “mutable”.

SourceKeyword::Namespace A reserved word for “namespace”.

SourceKeyword::New A reserved word for “new”.

SourceKeyword::Noexcept A reserved word for “noexcept”.

SourceKeyword::Nullptr A reserved word for “nullptr”.

SourceKeyword::Operator A reserved word for “operator”.

SourceKeyword::Pragma A reserved word for “_Pragma”.

SourceKeyword::Private A reserved word for “private”.

SourceKeyword::Protected A reserved word for “protected”.

SourceKeyword::Public A reserved word for “public”.

SourceKeyword::Register A reserved word for “register”.

SourceKeyword::ReinterpretCast A reserved word for “reinterpret_cast”.

SourceKeyword::Requires A reserved word for “requires”.

SourceKeyword::Restrict A reserved word for “restrict”.

SourceKeyword::Return A reserved word for “return”.

SourceKeyword::Short A reserved word for “short”.

- SourceKeyword::Signed** A reserved word for “signed”.
- SourceKeyword::Sizeof** A reserved word for “sizeof”.
- SourceKeyword::Static** A reserved word for “static”.
- SourceKeyword::StaticAssert** A reserved word for “static_assert”.
- SourceKeyword::StaticCast** A reserved word for “static_cast”.
- SourceKeyword::Struct** A reserved word for “struct”.
- SourceKeyword::Switch** A reserved word for “switch”.
- SourceKeyword::Template** A reserved word for “template”.
- SourceKeyword::This** A reserved word for “this”.
- SourceKeyword::ThreadLocal** A reserved word for “thread_local”.
- SourceKeyword::Throw** A reserved word for “throw”.
- SourceKeyword::True** A reserved word for “true”.
- SourceKeyword::Try** A reserved word for “try”.
- SourceKeyword::Typedef** A reserved word for “typedef”.
- SourceKeyword::Typeid** A reserved word for “typeid”.
- SourceKeyword::Typename** A reserved word for “typename”.
- SourceKeyword::Union** A reserved word for “union”.
- SourceKeyword::Unsigned** A reserved word for “unsigned”.
- SourceKeyword::Using** A reserved word for “using”.
- SourceKeyword::Virtual** A reserved word for “virtual”.
- SourceKeyword::Void** A reserved word for “void”.

SourceKeyword::Volatile A reserved word for “volatile”.

SourceKeyword::WcharT A reserved word for “wchar_t”.

SourceKeyword::While A reserved word for “while”.

SourceKeyword::Msvc No reserved word of this value shall be produced. All MSVC-specific reserved words have values greater than this.

SourceKeyword::MsvcAsm A reserved word for the MSVC extension “__asm”.

SourceKeyword::MsvcAssume A reserved word for the MSVC extension “__assume”.

SourceKeyword::MsvcAlignof A reserved word for the MSVC extension “__alignof”.

SourceKeyword::MsvcBased A reserved word for the MSVC extension “__based”.

SourceKeyword::MsvcCdecl A reserved word for the MSVC extension “__cdecl”.

SourceKeyword::MsvcClrcall A reserved word for the MSVC extension “__clrcall”.

SourceKeyword::MsvcDeclspec A reserved word for the MSVC extension “__declspec”.

SourceKeyword::MsvcEabi A reserved word for the MSVC extension “__eabi”.

SourceKeyword::MsvcEvent A reserved word for the MSVC extension “__event”.

SourceKeyword::MsvcSehExcept A reserved word for the MSVC extension “__except”.

SourceKeyword::MsvcFastcall A reserved word for the MSVC extension “__fastcall”.

SourceKeyword::MsvcSehFinally A reserved word for the MSVC extension “__finally”.

SourceKeyword::MsvcForceinline A reserved word for the MSVC extension “__forceinline”.

SourceKeyword::MsvcHook A reserved word for the MSVC extension “__hook”.

SourceKeyword::MsvcIdentifier A reserved word for the MSVC extension “__identifier”.

SourceKeyword::MsvcIfExists A reserved word for the MSVC extension “__if_exists”.

SourceKeyword::MsvcIfNotExists A reserved word for the MSVC extension “__if_not_exists”.

SourceKeyword::MsvcInt8 A reserved word for the MSVC extension “__int8”.

SourceKeyword::MsvcInt16 A reserved word for the MSVC extension “__int16”.

SourceKeyword::MsvcInt32 A reserved word for the MSVC extension “__int32”.

SourceKeyword::MsvcInt64 A reserved word for the MSVC extension “__int64”.

SourceKeyword::MsvcInt128 A reserved word for the MSVC extension “__int128”.

SourceKeyword::MsvcInterface A reserved word for the MSVC extension “__interface”.

SourceKeyword::MsvcLeave A reserved word for the MSVC extension “__leave”.

SourceKeyword::MsvcMultipleInheritance A reserved word for the MSVC extension “__multiple_inheritance”.

SourceKeyword::MsvcNullptr A reserved word for the MSVC extension “__nullptr”.

SourceKeyword::MsvcNovtordisp A reserved word for the MSVC extension “__novtordisp”.

SourceKeyword::MsvcPragma A reserved word for the MSVC extension “__pragma”.

SourceKeyword::MsvcPtr32 A reserved word for the MSVC extension “__ptr32”.

SourceKeyword::MsvcPtr64 A reserved word for the MSVC extension “__ptr64”.

SourceKeyword::MsvcRestrict A reserved word for the MSVC extension “__restrict”.

SourceKeyword::MsvcSingleInheritance A reserved word for the MSVC extension “__single_inheritance”.

SourceKeyword::MsvcSptr A reserved word for the MSVC extension “__sptr”.

SourceKeyword::MsvcStdcall A reserved word for the MSVC extension “__stdcall”.

SourceKeyword::MsvcSuper A reserved word for the MSVC extension “__super”.

SourceKeyword::MsvcThiscall A reserved word for the MSVC extension “__thiscall”.

SourceKeyword::MsvcSehTry A reserved word for the MSVC extension “__try”.

SourceKeyword::MsvcUptr A reserved word for the MSVC extension “__uptr”.

SourceKeyword::MsvcUuidof A reserved word for the MSVC extension “__uuidof”.

SourceKeyword::MsvcUnaligned A reserved word for the MSVC extension “__unaligned”.

SourceKeyword::MsvcUnhook A reserved word for the MSVC extension “__unhook”.

SourceKeyword::MsvcVectorcall A reserved word for the MSVC extension “__vectorcall”.

SourceKeyword::MsvcVirtualInheritance A reserved word for the MSVC extension “__vritual_inheritance”.

SourceKeyword::MsvcW64 A reserved word for the MSVC extension “__w64”.

SourceKeyword::MsvclsClass A reserved word for the MSVC extension “__is_class”.

SourceKeyword::MsvclsUnion A reserved word for the MSVC extension “__is_union”.

SourceKeyword::MsvclsEnum A reserved word for the MSVC extension “__is_enum”.

SourceKeyword::MsvclsPolymorphic A reserved word for the MSVC extension “__is_polymorphic”.

SourceKeyword::MsvclsEmpty A reserved word for the MSVC extension “__is_empty”.

SourceKeyword::MsvcHasTrivialConstructor A reserved word for the MSVC extension “__has_trivial_constructor”.

SourceKeyword::MsvclsTriviallyConstructible A reserved word for the MSVC extension “__is_trivially_constructible”.

SourceKeyword::MsvclsTriviallyCopyConstructible A reserved word for the MSVC extension “__is_trivially_copy_constructible”.

SourceKeyword::MsvclsTriviallyCopyAssignable A reserved word for the MSVC extension “__is_trivially_copy_constructible”.

SourceKeyword::MsvclsTriviallyDestructible A reserved word for the MSVC extension “__is_trivially_destructible”.

SourceKeyword::MsvcHasVirtualDestructor A reserved word for the MSVC extension “__has_virtual_destructor”.

SourceKeyword::MsvclsNothrowConstructible A reserved word for the MSVC extension “__is_nothrow_constructible”.

SourceKeyword::MsvclsNothrowCopyConstructible A reserved word for the MSVC extension “__is_nothrow_copy_constructible”.

SourceKeyword::MsvclsNothrowCopyAssignable A reserved word for the MSVC extension “__is_nothrow_copy_assignable”.

SourceKeyword::MsvclsPod A reserved word for the MSVC extension “__is_pod”.

SourceKeyword::MsvclsAbstract A reserved word for the MSVC extension “__is_abstract”.

SourceKeyword::MsvclsBaseOf A reserved word for the MSVC extension “__is_base_of”.

SourceKeyword::MsvclsConvertibleTo A reserved word for the MSVC extension “__is_convertible_to”.

SourceKeyword::MsvclsTrivial A reserved word for the MSVC extension “__is_trivial”.

SourceKeyword::MsvclsTriviallyCopyable A reserved word for the MSVC extension “__is_trivially_copyable”.

SourceKeyword::MsvclsStandardLayout A reserved word for the MSVC extension “__is_standard_layout”.

SourceKeyword::MsvclsLiteralType A reserved word for the MSVC extension “__is_literal_type”.

SourceKeyword::MsvclsTriviallyMoveConstructible A reserved word for the MSVC extension “__is_trivially_move_constructible”.

SourceKeyword::MsvcHasTrivialMoveAssign A reserved word for the MSVC extension “__has_trivial_move_assign”.

SourceKeyword::MsvclsTriviallyMoveAssignable A reserved word for the MSVC extension “__is_trivially_move_assignable”.

SourceKeyword::MsvclsNothrowMoveAssignable A reserved word for the MSVC extension “__is_nothrow_move_assignable”.

SourceKeyword::MsvclsConstructible A reserved word for the MSVC extension “__is_constructible”.

SourceKeyword::MsvcUnderlyingType A reserved word for the MSVC extension “__underlying_type”.

SourceKeyword::MsvclsTriviallyAssignable A reserved word for the MSVC extension “__is_trivially_assignable”.

SourceKeyword::MsvclsNothrowAssignable A reserved word for the MSVC extension “__is_nothrow_assignable”.

SourceKeyword::MsvclsDestructible A reserved word for the MSVC extension “__is_destructible”.

SourceKeyword::MsvclsNothrowDestructible A reserved word for the MSVC extension “__is_nothrow_destructible”.

SourceKeyword::MsvclsAssignable A reserved word for the MSVC extension “__is_assignable”.

SourceKeyword::MsvclsAssignableNocheck A reserved word for the MSVC extension “__is_assignable_no_precondition_check”.

SourceKeyword::MsvcHasUniqueObjectRepresentations A reserved word for the MSVC extension “__has_unique_object_representations”.

SourceKeyword::MsvclsAggregate A reserved word for the MSVC extension “__is_aggregate”.

SourceKeyword::MsvcBuiltinAddressOf A reserved word for the MSVC extension “__builtin_addressof”.

SourceKeyword::MsvcBuiltinOffsetOf A reserved word for the MSVC extension “__builtin_offsetof”.

SourceKeyword::MsvcBuiltinBitCast A reserved word for the MSVC extension “__builtin_bit_cast”.

SourceKeyword::MsvcBuiltinIsLayoutCompatible A reserved word for the MSVC extension “__builtin_is_layout_compatible”.

SourceKeyword::MsvcBuiltinIsPointerInterconvertibleBaseOf A reserved word for the MSVC extension “__builtin_is_pointer_interconvertible_base_of”.

SourceKeyword::MsvcBuiltinIsPointerInterconvertibleWithClass A reserved word for the MSVC extension “__builtin_is_pointer_interconvertible_with_class”.

SourceKeyword::MsvcBuiltinIsCorrespondingMember A reserved word for the MSVC extension “__builtin_is_corresponding_member”.

SourceKeyword::MsvcIsRefClass A reserved word for the MSVC extension “__is_ref_class”.

SourceKeyword::MsvcIsValueClass A reserved word for the MSVC extension “__is_value_class”.

SourceKeyword::MsvcIsSimpleValueClass A reserved word for the MSVC extension “__is_simple_value_class”.

SourceKeyword::MsvcIsInterfaceClass A reserved word for the MSVC extension “__is_interface_class”.

SourceKeyword::MsvcIsDelegate A reserved word for the MSVC extension “__is_delegate”.

SourceKeyword::MsvcIsFinal A reserved word for the MSVC extension “__is_final”.

SourceKeyword::MsvcIsSealed A reserved word for the MSVC extension “__is_sealed”.

SourceKeyword::MsvcHasFinalizer A reserved word for the MSVC extension “__has_finalizer”.

SourceKeyword::MsvcHasCopy A reserved word for the MSVC extension “__has_copy”.

SourceKeyword::MsvcHasAssign A reserved word for the MSVC extension “__has_assign”.

SourceKeyword::MsvcHasUserDestructor A reserved word for the MSVC extension “__has_user_destructor”.

SourceKeyword::MsvcPackCardinality A reserved word for “sizeof...”.

SourceKeyword::MsvcConfusedSizeof A reserved word for “sizeof”.

SourceKeyword::MsvcConfusedAlignas A reserved word for “alignas”.

WordSort::Identifier

A word (Figure 19.4) with the *sort* field that holds **WordSort::Identifier** is an *identifier word*. The corresponding *value* field is of type

```
enum class SourceIdentifier : uint16_t { };
```

Figure 19.10: Definition of type *SourceIdentifier*

Values of type *SourceIdentifier* are classified in two groups.

0x00. **Plain**

and those with values greater than 0x1FFF

0x1FFF. Msvc	0x2003. MsvcBuiltinNanf
0x2000. MsvcBuiltinHugeVal	0x2004. MsvcBuiltinNans
0x2001. MsvcBuiltinHugeValf	0x2005. MsvcBuiltinNansf
0x2002. MsvcBuiltinNan	

SourceIdentifier::Plain An identifier word with the *value* field holding **SourceIdentifier::Plain** designates a C++ source-level identifier. The spelling of that identifier is indicated by the *index* field which is of type *TexOffset* (§3.1).

SourceIdentifier::Msvc No identifier word has a *value* field holding this value. It serves purely as a marker denoting the starting point of identifiers with built-in meaning to MSVC.

SourceIdentifier::MsvcBuiltinHugeVal An identifier word for `__builtin_huge_val`.

SourceIdentifier::MsvcBuiltinHugeValf An identifier word for `__builtin_huge_valf`.

SourceIdentifier::MsvcBuiltinNan An identifier word for `__builtin_nan`.

SourceIdentifier::MsvcBuiltinNanf An identifier word for `__builtin_nanf`.

SourceIdentifier::MsvcBuiltinNans An identifier word for `__builtin_nans`.

SourceIdentifier::MsvcBuiltinNansf An identifier word for `__builtin_nansf`.

Bibliography

- [1] Gabriel Dos Reis and Bjarne Stroustrup. A Principled, Complete, and Efficient Representation of C++. *Journal of Mathematics in Computer Science*, Special Issue on Polynomial System Solving, System and Control, and Software Science, 2011.
- [2] Internal program representation. <https://github.com/GabrielDosReis/ipr>.