IFC Binary Format
Version: 0.25

Gabriel Dos Reis
Microsoft

November 16, 2020

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 incorrekt, incomplet, and a work in progres.

Contents

Chapter 1

Introduction

Chapter 2

IFC

2.1 Overview

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

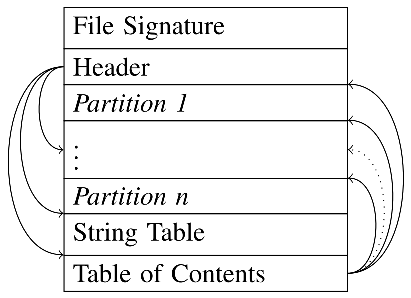

Figure 2.1: IFC file general structure

An IFC represents the *abstract semantics graph* that is the result of elaborating declarations in an input source module interface file. Entities 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:

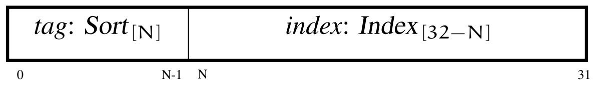
tag: $\mathsf{Sort}_{[N]}$
index: $\mathsf{Index}_{[32-N]}$
0
N-1
N
31

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 Rep-*

resentation (IPR), a work done by Gabriel Dos Reis and Bjarne Stroustrup [1, 2] to define a more regular foundamentional 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 contaning 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

2.6.1 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.

2.6.2 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 abstacted as follows:

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

Figure 2.4: Definition of type *Cardinality*

2.6.3 Extent of entities

At various places, in partitular in partition summaries (§2.9.1), 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*

2.6.4 Generic Indices

The type of generic indices into tables is defined as

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

Figure 2.6: Definition of type *Index*

2.6.5 Sequence

A sequence is generically described by a pair:

start: *Index*
cardinality: *Cardinality*

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.

2.6.6 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.

2.6.7 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*

2.6.8 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*

2.6.9 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
};
```

2.6.10 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 (§2.9.1) the IFC contains.

checksum: *SHA256*
major_version: *Version*
minor_version: *Version*
abi: *Abi*
arch: *Architecture*
dialect: *LanguageVersion*
string_table_bytes: *ByteOffset*
string_table_size: *Cardinality*
unit: *UnitIndex*
src_path: *TextOffset*
global_scope: *ScopeIndex*
toc: *ByteOffset*
partition_count: *Cardinality*
internal: *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 homoegenous tables (called *partitions*) with entries referencing each other.

2.9.1 Partition

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

name: *TextOffset*
offset: *ByteOffset*
cardinality: *Cardinality*
entry_size: *EntitySize*

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 anlysis (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 (§14) 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 §11.1.18.

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*.

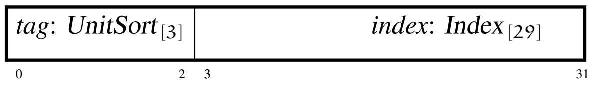

Figure 4.1: `UnitIndex`: Abstract reference of a declaration

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

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

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

4.1 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.

4.2 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.

4.3 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.

4.4 UnitSort::Header

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

4.5 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 reference

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

owner: *TextOffset*
partition: *TextOffset*

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.

5.2 Imported modules

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

Partition name: `"module.imported"`.

5.3 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), designating

start: *Index*
cardinality: *Cardinality*

Figure 6.2: Structure of a *Scope*

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.

index: *DeclIndex*

Figure 6.3: Structure of a *Declarataion* – 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 Declaration heap

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

Partition name: `"heap.decl"`.

7.2 Type heap

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

Partition name: `"heap.type"`.

7.3 Statement heap

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

Partition name: `"heap.stmt"`.

7.4 Expression heap

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

Partition name: `"heap.expr"`.

7.5 Syntax heap

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

Partition name: `"heap.syn"`.

7.6 Form heap

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

Partition name: `"heap.form"`.

7.7 Chart heap

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

Partition name: `"heap.chart"`.

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

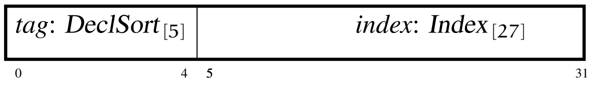

Figure 8.1: *DeclIndex*: Abstract reference of declaration

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.	ExplicitSpecialization	0x1C.	SyntaxTree
0x0D.	ExplicitInstantiation	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.

8.1.1 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
};
```

8.1.2 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.

8.1.3 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 necessarilyto 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.
};
```

8.1.4 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.

8.1.5 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,
};
```

8.1.6 Parameter Level

A template declaration can have many nesting levesl. This is the case of member templates of class templates; that is a member of a clsss 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*

8.1.7 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

8.2.1 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"`.

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

name: *TextOffset*
locus: *SourceLocation*
type: *TypeIndex*
initializer: *ExprIndex*
specifier: *BasicSpecifiers*
access: *Access*

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"`.

8.2.3 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.

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

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 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"`.

8.2.4 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 components:

name: TextOffset
locus: SourceLocation
type: TypeIndex
constraint: ExprIndex
initializer: ExprIndex
level: ParameterLevel
position: ParameterPosition
sort: ParameterSort
properties: ReachableProperties
pack: bool

Figure 8.6: Structure of a template parameter declaration

The *name* field denotes the name of the template parameter. If null, the template parameter was unnamed in the source input. The *locus* field denotes the location of the template parameter. The *type* field designates the type of the parameter. The *initializer* designates a default value, if any.

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"`.

8.2.4.1 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
};
```

8.2.5 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.

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

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"`.

8.2.6 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.

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

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"`.

8.2.7 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:

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

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 (§9.1.2) 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`".

8.2.7.1 Scope traits

Properties specific to scope entities are described by values of the bitmask type *Scope-Traits*:

```
enum class ScopeTraits : uint8_t {
  None      = 0,
  Unnamed   = 1 << 0,
  Inline      = 1 << 1,
  InitializerExported = 1 << 2,
  ClosureType = 1 << 3,
  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 namespace entties. 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::Vendor`: valid only if the scope entity has vendor-defined traits.

8.2.7.2 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*

8.2.8 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:

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

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 enumeraion. The *initializer* is a slice (§2.6.5) of the enumerator partition (§8.2.2). 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"`.

8.2.9 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:

name: *TextOffset*
locus: *SourceLocation*
type: *TypeIndex*
home_scope: *DeclIndex*
aliasee: *TypeIndex*
specifiers: *BasicSpecifiers*
access: *Access*

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 designates the kind of alias: it is `TypeBasis::Typename` for type aliases; it will be `TypeBasis::Namespace` for namespace 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.

Partition name: `"decl.alias"`.

8.2.10 DeclSort::Temploid

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

entity: *ParameterizedEntity*
chart: *ChartIndex*
properties: *ReachableProperties*

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"`.

8.2.10.1 Parameterized Entity

The structure *ParameterizedEntity* has the following layout

index: *Index*
head: *SentenceIndex*
body: *SentenceIndex*
attributes: *SentenceIndex*

Figure 8.14: Structure of a declaration parameterized by a template

The *index* field has the following interpretation: If the the abstract reference (*DeclIndex*) that references the template declaration is an alias template, then *index* is actually of type *TypeIndex* denoting the right hand side of that alias template declaration. Otherwise, it is a value of type *DeclIndex* denoting the current instantiation of that template declaration. This representation will be improved in future releases with uniform representation of type aliases. The *head* field designate the sentence that makes up the non-defining declarative part of the current instantiation. This sentence is no longer meaningful in recent releases of MSVC since any semantics information can be obtained from the entity denoted by *index*. The *body* field denotes the sentence of the defining ("body") part of the current instantiation. This field is meaningful only for templated functions.

8.2.11 DeclSort::Template

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

name: *NameIndex*
locus: *SourceLocation*
home_scope: *DeclIndex*
chart: *ChartIndex*
entity: *ParameterizedEntity*
specifiers: *BasicSpecifiers*
access: *Access*
properties: *ReachableProperties*

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 *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"`.

8.2.12 DeclSort::PartialSpecialization

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

name: *NameIndex*
locus: *SourceLocation*
home_scope: *DeclIndex*
chart: *ChartIndex*
entity: *ParameterizedEntity*
form: *Index*
specifiers: *BasicSpecifiers*
access: *Access*
properties: *ReachableProperties*

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"`.

8.2.13 DeclSort::ExplicitSpecialization

An explicit specialization of a template (class-type or function).

form: *Index*
decl: *DeclIndex*

Figure 8.17: Structure of an explicit specialization declaration

The *form* field is an index into the specialization form partition (named "form.spec"). The *decl* field denotes the declaration under the empty template parameter list. The *properties* field designates the set of reachable semantic properties.

Partition name: `"decl.explicit-specialization"`.

8.2.14 DeclSort::ExplicitInstantiation

An explicit instantiation request of a template specialization.

form: *Index*
decl: *DeclIndex*

Figure 8.18: Structure of an explicit instantiation declaration

The *form* field is an index into the partition of specialization forms. The *decl* field denotes the declarations designated by the explicit instantiation.

Partition name: `"decl.explicit-instantiation"`.

8.2.15 DeclSort::Concept

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

Figure 8.19: Structure of a concept declaration

8.2.16 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:

name: *NameIndex*
locus: *SourceLocation*
type: *TypeIndex*
home_scope: *DeclIndex*
chart: *ChartIndex*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*
properties: *ReachableProperties*

Figure 8.20: 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 (§16.9.2). That representation is subject to removal in future MSVC releases.

Partition name: `"decl.function"`.

8.2.16.1 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 `consteval`, or an immediate function.

`FunctionTraits::Vendor`: the function has vendor-defined traits

8.2.17 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

name: *NameIndex*
locus: *SourceLocation*
type: *TypeIndex*
home_scope: *DeclIndex*
chart: *ChartIndex*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*
properties: *ReachableProperties*

Figure 8.21: 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 aling with their default arguments. The *traits* indicates any additional function-specific traits (§8.2.16.1). The *access* designates the access specifier for this function.

Partition name: `"decl.method"`.

8.2.18 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.

name: *TextOffset*
locus: *SourceLocation*
source: *TypeIndex*
home_scope: *DeclIndex*
eh_spec: *NoexceptSpecification*
chart: *ChartIndex*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*
convention: *CallingConvention*
properties: *ReachableProperties*

Figure 8.22: Structure of a constructor declaration

The *name* field designates the name of the constructor. The *locus* field designates the source location of the constructor. The *source* field designates the parameter type list of the constructor. The *home_scope* field designates the declaration of the enclosing type. The *eh_spec* designates the exception specification of the constructor. The *chart* field designates the function parameter list along with thir default argument expressions. The *traits* field designates function-specific traits (§8.2.16.1). The *specifiers* field designate the usual basic declaration specifiers. The *access* field designates the access specifier of the constructor. The *convention* field indicates the calling convention used by the constructor.

Partition name: `"decl.constructor"`.

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

words: *SentenceIndex*
sort: *NoexceptSort*

Figure 8.23: 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

8.2.19 DeclSort::InheritedConstructor

A *DeclIndex* abstract reference with tag DeclSort::InheritedConstructor designates an inheritated 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

name: *TextOffset*
locus: *SourceLocation*
source: *TypeIndex*
home_scope: *DeclIndex*
eh_spec: *NoexceptSpecification*
chart: *ChartIndex*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*
convention: *CallingConvention*
base_ctor: *DeclIndex*

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

8.2.20 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.

name: *TextOffset*
locus: *SourceLocation*
home_scope: *DeclIndex*
eh_spec: *NoexceptSpecification*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*
convention: *CallingConvention*
properties: *ReachableProperties*

Figure 8.24: 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 (§8.2.16.1) 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"`.

8.2.21 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:

unit: *ModuleReference*
local_index: *DeclIndex*

Figure 8.25: 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"`.

8.2.21.1 *ModuleReference*

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

owner: *TextOffset*
partition: *TextOffset*

Figure 8.26: 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.

8.2.22 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:

name: *TextOffset*
locus: *SourceLocation*
home_scope: *DeclIndex*
resolution: *DeclIndex*
parent: *ExprIndex*
name2: *TextOffset*
specifiers: *BasicSpecifiers*
access: *Access*
hidden: *bool*

Figure 8.27: 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.

8.2.23 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"`.

8.2.24 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:

entity: *TypeIndex*

Figure 8.28: Structure of a friend declaration

If *entity* field is the abstract reference to the type declared as friend.

Partition name: `"decl.friend"`.

Note: This representation will change in future releases.

8.2.25 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

operand: *DeclIndex*
locus: *SourceLocation*

Figure 8.29: Structure of an expansion declaration

Partition name: `"decl.expansion"`.

8.2.26 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

name: *NameIndex*
locus: *SourceLocation*
parameters: *ChartIndex*
specialization: *TypeIndex*
traits: *FunctionTraits*
specifiers: *BasicSpecifiers*
access: *Access*

Figure 8.30: Structure of a deduction guide declaration

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

8.2.27 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"`.

8.2.28 DeclSort::Tuple

A *DeclIndex* abstract reference with tag DeclSort::Tuple designates a sequence of declarations referenced in other source-level contructs, 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

start: *Index*
cardinality: *Cardinality*

Figure 8.31: 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"`.

8.2.29 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

8.2.30 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:

name: *TextOffset*
locus: *SourceLocation*
type: *TypeIndex*
home_scope: *DeclIndex*
specifiers: *BasicSpecifiers*
access: *Access*

Figure 8.32: 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

8.2.31 DeclSort::Property

A *DeclIndex* abstract reference with tag DeclSort::Property designates MSVC's extenion 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

member: *DeclIndex*
getter: *TextOffset*
setter: *TextOffset*

Figure 8.33: 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

8.2.32 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

name: *TextOffset*
ID: *TextOffset*
traits: *SegmentTraits*
type: *SegmentType*

Figure 8.34: Sturucture 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.

8.2.32.1 Code segment traits

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

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

8.2.32.2 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

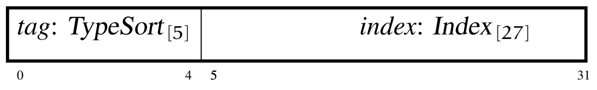

Figure 9.1: *TypeIndex*: Abstract reference of type

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

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

9.1 Type structures

9.1.1 TypeSort::VendorExtension

Partition name: `"type.vendor-extension"`.

9.1.2 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:

basis: *TypeBasis*
precision: *TypePrecision*
sign: *TypeSign*
<padding>: *uint8_t*

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"`.

9.1.2.1 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,
  Auto,
    DecltypeAuto,
};
```

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`
- `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
- `TypeBasis::Empty`: fundamental type resulting from an empty pack expansion
- `TypeBasis::VariableTemplate`: concept of variable template
- `TypeBasis::Auto`: type placeholder `auto`
- `TypeBasis::DecltypeAuto` type placeholder `decltype(auto)`

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

9.1.2.2 Fundamental type precision

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

```
enum class TypePrecision : uint8_t {
  Default,
  Short,
  Long,
  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::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

9.1.2.3 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

9.1.3 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.

decl: *DeclIndex*

Figure 9.3: Structure of a designated type

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

Partition name: `"type.designated"`.

9.1.4 TypeSort::Deduced

A *TypeIndex* value with tag TypeSort::Deduced designates a return type syntactically specified at the source code level as `auto` or `decltype(auto)` supposed to be deduced from the body of the function. The *index* field is an index into the deduced type partition. Each entry in that partition is a structure with a single field

return_type: *TypeIndex*

Figure 9.4: Structure of a deduced type

If the field *return_type* is a null abstract reference, then the type has not actually be deduced as can happen with a non-defining declaration.

Partition name: `"type.deduced"`.

9.1.5 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 wth a single component: the *expr* field.

expr: *ExprIndex*

Figure 9.5: Structure of a syntactic type

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

Partition name: `"type.syntactic"`.

9.1.6 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

pack: *TypeIndex*
mode: *ExpansionMode*

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"`.

9.1.6.1 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
};
```

9.1.7 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.

pointee: *TypeIndex*

Figure 9.7: Structure of a pointer type

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

Partition name: `"type.pointer"`.

9.1.8 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.

scope: *TypeIndex*
member: *TypeIndex*

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"`.

9.1.9 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.

referee: *TypeIndex*

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"`.

9.1.10 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.

referee: *TypeIndex*

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"`.

9.1.11 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.

target: *TypeIndex*
source: *TypeIndex*
eh_spec: *NoexceptSpecification*
convention: *CallingConvention*
traits: *FunctionTypeTraits*

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++ souce-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"`.

9.1.11.1 Function type traits

Certain function types properties are expressed as value of bitmask type *FunctionType-Traits* 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-menber 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.

9.1.11.2 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.

9.1.11.3 Noexcept specification

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

words: *SentenceIndex*
sort: *NoexceptSort*

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.

9.1.12 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.

target: *TypeIndex*
source: *TypeIndex*
scope: *TypeIndex*
eh_spec: *NoexceptSpecification*
convention: *CallingConvention*
traits: *FunctionTypeTraits*

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++ souce-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"`.

9.1.13 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.

element: *TypeIndex*
extent: *ExprIndex*

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"`.

9.1.14 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.

path: *ExprIndex*

Figure 9.15: Structure of a typename type

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

Partition name: `"type.typename"`.

9.1.15 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 entrry in that partition is a structure with two components: the *unqualified* field, and the *qualifiers* field.

unqualified: *TypeIndex*
qualifiers: *Qualifiers*

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"`.

9.1.15.1 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.

9.1.16 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.

type: *TypeIndex*
access: *Access*
shared: *bool*
pack_expanded: *bool*

Figure 9.17: Structure of a base type

The *type* field denotes the type base being used as base-class. The *access* field denotes the access specifier in the inheritance path. The *shared* field indicates whether the base class is virtual or not. The *pack_expanded* field indicate where the base class is the result of pack expansion.

Partition name: `"type.base"`.

Note: This representation is subject to change

9.1.17 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.

expr: *SyntaxIndex*

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 three.

9.1.18 TypeSort::Placeholder

A *TypeIndex* abstract reference 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

constraint: *ExprIndex*
basis: *TypeBasis*

Figure 9.19: Structure of a placeholder type

The *constraint* field designate the (generalized) predicate that the placeholder shall satisfy – at the input source lvel. The *basis* field is a structure denoting the type basis (§9.1.2.1) pattern expected for placeholder type, either `auto` or `decltype(auto)`.

Partition name: `"type.placeholder"`.

9.1.19 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 structrure with two components: a *start* field, and a *cardinality* field.

start: *Index*
cardinality: *Cardinality*

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"`.

9.1.20 TypeSort::Forall

A *TypeIndex* value with tag TypeSort::Forall designates a generalized ∀-type, as can could 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

chart: *ChartIndex*
subject: *TypeIndex*

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.

9.1.21 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.

type: *TypeIndex*

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.

9.1.22 TypeSort::SyntaxTree

A *TypeIndex* value with tag TypeSort::SyntaxTree represents an abstract reference to a type designated by a raw syntax contruct (§14). 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:

syntax: *SyntaxIndex*

Figure 9.23: Structure of a syntax tree type

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

Chapter 10

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*:

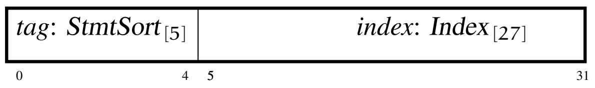

Figure 10.1: *StmtIndex*: Abstract reference of statement

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

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

10.1 Statement structures

10.1.1 VendorExtension

Partition name: `"stmt.vendor-extension"`.

10.1.2 StmtSort::Empty

locus: *SourceLocation*

Figure 10.2: Structure of an empty statement

Partition name: `"stmt.empty"`.

10.1.3 StmtSort::If

initialization: *StmtIndex*
condition: *StmtIndex*
consequence: *StmtIndex*
alternative: *StmtIndex*
locus: *SourceLocation*

Figure 10.3: Structure of an if-statement

Partition name: `"stmt.if"`.

10.1.4 StmtSort::For

initialization: *StmtIndex*
condition: *StmtIndex*
continuation: *StmtIndex*
body: *StmtIndex*
locus: *SourceLocation*

Figure 10.4: Structure of a for-statement

Partition name: `"stmt.for"`.

10.1.5 StmtSort::Case

expr: *ExprIndex*
locus: *SourceLocation*

Figure 10.5: Structure of a case-label

Partition name: `"stmt.case"`.

10.1.6 StmtSort::While

condition: *StmtIndex*
body: *StmtIndex*
locus: *SourceLocation*

Figure 10.6: Structure of a while-statement

Partition name: `"stmt.while"`.

10.1.7 StmtSort::Block

start: *Index*
cardinality: *Cardinality*

Figure 10.7: Structure of a block statement

Partition name: `"stmt.block"`.

10.1.8 StmtSort::Break

locus: *SourceLocation*

Figure 10.8: Structure of a break statement

Partition name: `"stmt.break"`.

10.1.9 StmtSort::Switch

initialization: *StmtIndex*
condition: *ExprIndex*
body: *StmtIndex*
locus: *SourceLocation*

Figure 10.9: Structure of a switch-statement

Partition name: `"stmt.switch"`.

10.1.10 StmtSort::DoWhile

condition: *StmtIndex*
body: *StmtIndex*
locus: *SourceLocation*

Figure 10.10: Structure of a do-statement

Partition name: `"stmt.do-while"`.

10.1.11 StmtSort::Default

locus: *SourceLocation*

Figure 10.11: Structure of a default label

Partition name: `"stmt.default"`.

10.1.12 StmtSort::Continue

locus: *SourceLocation*

Figure 10.12: Structure of a continue statement

Partition name: `"stmt.continue"`.

10.1.13 StmtSort::Expression

expr: *ExprIndex*
locus: *SourceLocation*

Figure 10.13: Structure of an expression-statement

Partition name: `"stmt.expression"`.

10.1.14 StmtSort::Return

expr: *ExprIndex*
function_type: *TypeIndex*
expression_type: *TypeIndex*
locus: *SourceLocation*

Figure 10.14: Structure of a return statement

Partition name: `"stmt.return"`.

10.1.15 StmtSort::VariableDecl

decl: *DeclIndex*
initializer: *ExprIndex*
locus: *SourceLocation*

Figure 10.15: Structure of a variable-declaration statement

Partition name: `"stmt.variable"`.

10.1.16 StmtSort::Expansion

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

operand: *StmtIndex*

Figure 10.16: Structure of an expansion statement

Partition name: `"stmt.expansion"`.

10.1.17 StmtSort::SyntaxTree

TBD

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

Chapter 11

Expressions

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

tag: *ExprSort*$_{[6]}$	*index*: *Index*$_{[26]}$
0 … 5	6 … 31

Figure 11.1: *ExprIndex*: Abstract reference of expression

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

0x00. VendorExtension
0x01. Empty
0x02. Literal
0x03. Lambda
0x04. Type
0x05. NamedDecl
0x06. UnresolvedId
0x07. TemplateId
0x08. UnqualifiedId
0x09. SimpleIdentifier
0x0A. Pointer
0x0B. QualifiedName
0x0C. Path
0x0D. Read
0x0E. Monad
0x0F. Dyad
0x10. Triad
0x11. String
0x12. Temporary
0x13. Call
0x14. MemberInitializer
0x15. MemberAccess
0x16. InheritancePath
0x17. InitializerList
0x18. Cast
0x19. Condition
0x1A. ExpressionList
0x1B. SizeofType
0x1C. Alignof
0x1D. New
0x1E. Delete
0x1F. Typeid
0x20. DestructorCall
0x21. SyntaxTree
0x22. FunctionString

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

11.1 Expression structures

11.1.1 ExprSort::VendorExtension

Partition name: `"expr.vendor-extension"`.

11.1.2 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.

locus: *SourceLocation*
type: *TypeIndex*

Figure 11.2: Structure of an empty expression

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

Partition name: `"expr.empty"`.

11.1.3 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.

locus: *SourceLocation*
type: *TypeIndex*
value: *LitIndex*

Figure 11.3: 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"`.

11.1.3.1 Literal Values

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

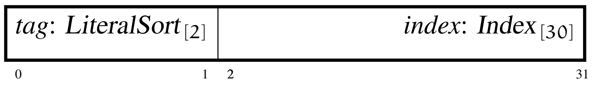

Figure 11.4: *LitIndex*: Abstract reference of literal constants

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. The value at that entry is a 64-bit floating point value, in IEEE 754 little endian format.

11.1.4 ExprSort::Lambda

An *ExprIndex* value with tag ExprSort::Lambda designates a lambda expression in syntactic form (§14). 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:

introducer: *SyntaxIndex*
template_parameters: *SyntaxIndex*
declarator: *SyntaxIndex*
constraint: *SyntaxIndex*
body: *SyntaxIndex*

Figure 11.5: 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"`.

11.1.5 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 enty in that partition has two components: a *type* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
denotation: *TypeIndex*

Figure 11.6: 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::Typename`.

Partition name: `"expr.type"`.

11.1.6 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.

locus: *SourceLocation*
type: *TypeIndex*
resolution: *DeclIndex*

Figure 11.7: 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"`.

11.1.7 ExprtSort::UnresolvedId

A *ExprIndex* value with tag ExprSort::UnresolveId represents a C++ source-level dependent name, or an unresolve 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.

locus: *SourceLocation*
type: *TypeIndex*
name: *NameIndex*

Figure 11.8: Structure of an unresolved name expression

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

Partition name: `"expr.unresolved"`.

11.1.8 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.

locus: *SourceLocation*
type: *TypeIndex*
primary: *ExprIndex*
arguments: *ExprIndex*

Figure 11.9: 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"`.

11.1.9 ExprSort::UnqualifiedId

A *ExprIndex* value with tag ExprSort::UnqualifiedId denotes the C++ grammar term *unqualified-id* or a component of *qualified-id* expression, 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 structure:

locus: *SourceLocation*
type: *TypeIndex*
name: *NameIndex*
resolution: *ExprIndex*
template_keyword: *SourceLocation*

Figure 11.10: 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"`.

11.1.10 ExprSort::SimpleIdentifier

locus: *SourceLocation*
type: *TypeIndex*
name: *NameIndex*

Figure 11.11: Structue of a simple-identifier expression

Partition name: `"expr.simple-identifier"`.

11.1.11 ExprSort::Pointer

locus: *SourceLocation*

Partition name: `"expr.pointer"`.

11.1.12 ExprSort::QualifiedName

locus: *SourceLocation*
type: *TypeIndex*
elements: *ExprIndex*
typename_keyword: *SourceLocation*

Figure 11.12: Structure of a qualified name expression

Partition name: `"expr.qualified-name"`.

11.1.13 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.

locus: *SourceLocation*
type: *TypeIndex*
scope: *ExprIndex*
member: *ExprIndex*

Figure 11.13: Structure of a path expression

The *scope* field denotes the qualifiying 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".

11.1.14 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 represents so-called lvalue-to-rvalue conversions. The *index* field is an index into the read expression partition. Each enty in that partition has three components: a *type* field, an *address* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
address: *ExprIndex*
sort: *ReadConversionSort*

Figure 11.14: 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".

11.1.14.1 *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.

11.1.15 ExprtSort::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 parrition has four components: a *type* field, a *opcat* field, an *argument* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
argument: *ExprIndex*
operator: *MonadicOperator*

Figure 11.15: Structure of a monadic expression

The *operator* field denotes the conceptual monadic operation, usually as written in the input C++ source code, §11.4.2. The *type* field denotes the type of the expression. The *argument* field denotes the argument to the operator. The *locus* denotes the source location.

Partition name: `"expr.monad"`.

11.1.16 ExprSort::Dyad

A *ExprIndex* value with tag ExprSort::Monad represents the application of a (source-level) dyadic operator (§11.4.3) to arguments. The *index* field is an index into the dyadic expression partition. Each entry in that parrition has four components: a *type* field, a *opcat* field, an *arguments* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
arguments: *ExprIndex[2]*
operator: *DyadicOperator*

Figure 11.16: Structure of a dyadic expression

The *operator* field denotes the conceptual dyadic operation (usually as written in the input C++ source code). The *type* field denotes the type of the expression. The *arguments* field denotes the two arguments to the operator. The *locus* denotes the source location.

Partition name: `"expr.dyad"`.

11.1.17 ExprSort::Triad

A *ExprIndex* value with tag ExprSort::Monad represents the application of a (source-level) triadic operator (§11.4.4) to arguments. The *index* field is an index into the triadic expression partition. Each entry in that parrition has four components: a *type* field, a *opcat* field, an *arguments* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
arguments: *ExprIndex[3]*
operator: *TriadicOperator*

Figure 11.17: Structure of a triadic expression

The *opcat* field denotes the conceptual triadic operation (usually as written in the input C++ source code). The *type* field denotes the type of the expression. The *arguments* field denotes the three arguments to the operator. The *locus* denotes the source location.

Partition name: `"expr.triad"`.

11.1.18 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.

locus: *SourceLocation*
type: *TypeIndex*
string_index: *StringIndex*

Figure 11.18: 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:

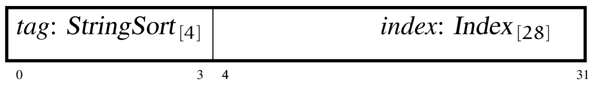

Figure 11.19: *StringIndex*: Abstract reference of string constant

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"`.

11.1.18.1 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.

11.1.18.2 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.

11.1.18.3 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.

11.1.18.4 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.

11.1.18.5 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.

11.1.18.6 String literal structure

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

start: *TextOffset*
length: *Cardinality*
suffix: *TextOffset*

Figure 11.20: 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"`.

11.1.19 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.

locus: *SourceLocation*
type: *TypeIndex*
id: *UniqueID*

Figure 11.21: 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 11.22: Definition of type *UniqueID*

The *locus* field denotes the source location.

Partition name: `"expr.temporary"`.

11.1.20 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.

locus: *SourceLocation*
type: *TypeIndex*
operation: *ExprIndex*
arguments: *ExprIndex*

Figure 11.23: 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. It 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"`.

11.1.21 ExprSort::MemberInitializer

A base or member initializer

locus: *SourceLocation*
type: *TypeIndex*
member: *DeclIndex*
base: *TypeIndex*
initializer: *ExprIndex*

Figure 11.24: Structure of a member-initializer expression

Partition name: `"expr.member-initializer"`.

11.1.22 ExprSort::MemberAccess

The member being accessed + offset

locus: *SourceLocation*
type: *TypeIndex*
member: *DeclIndex*
offset: *ExprIndex*
name: *TextOffset*

Figure 11.25: Structure of a member access expression

Partition name: `"expr.member-access"`.

11.1.23 ExprSort::InheritancePath

The representation of the 'path' to a base-class member

locus: *SourceLocation*
type: *TypeIndex*
path: *ExprIndex*

Figure 11.26: Structure of an inheritance path expression

Partition name: `"expr.inheritance-path"`.

11.1.24 ExprSort::InitializerList

locus: *SourceLocation*
type: *TypeIndex*
elements: *ExprIndex*

Figure 11.27: Structure of an initializer-list expression

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

11.1.25 ExprSort::Cast

locus: *SourceLocation*
type: *TypeIndex*
source: *ExprIndex*
target: *Index*
operator: *DyadicOperator*

Figure 11.28: Structure of a cast expression

Partition name: `"expr.cast"`.

11.1.26 ExprSort::Condition

locus: *SourceLocation*
type: *TypeIndex*
expr: *ExprIndex*

Figure 11.29: Structure of a condition expression

Partition name: `"expr.condition"`.

11.1.27 ExprSort::ExpressionList

left: *SourceLocation*
right: *SourceLocation*
contents: *ExprIndex*
delimiter: *DelimiterSort*

Figure 11.30: Structure of an expression list

with

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

Figure 11.31: Definition of type *DelimiterSort*

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

11.1.28 ExprSort::SizeofType

locus: *SourceLocation*
type: *TypeIndex*
operand: *SyntaxIndex*

Figure 11.32: 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"`.

11.1.29 ExprSort::Alignof

locus: *SourceLocation*
type: *TypeIndex*
operand: *SyntaxIndex*

Figure 11.33: 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"`.

11.1.30 ExprSort::New

double_colon: *SourceLocation*
new_keyword: *SourceLocation*
allocated_type: *SyntaxIndex*
placement: *ExprIndex*
initializer: *ExprIndex*

The field *double_colon* designates the location of the global scope operator `::`, if present. The field *new_keyword* designates the location of the `new` keyword operator. The field *allocated_type* designates the source-level syntax of the type allocated in the new expression. The field *placement* designates the placement, if there is any. The field *initialzier* designates the initializer, if any.

Partition name: `"expr.new"`.

11.1.31 ExprSort::Delete

double_colon: *SourceLocation*
delete_keyword: *SourceLocation*
address: *ExprIndex*

Partition name: `"expr.delete"`.

11.1.32 ExprSort::Typeid

locus: *SourceLocation*
type: *TypeIndex*
operand: *TypeIndex*

Figure 11.34: 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"`.

11.1.33 ExprSort::DestructorCall

locus: *SourceLocation*
type: *TypeIndex*
name: *ExprIndex*
decltype_specifier: *SyntaxIndex*
cleanup: *DestructorSort*

Figure 11.35: Structure of a destructor call

with

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

Figure 11.36: Definition of type *DestructorSort*

Partition name: `"expr.destructor-call"`.

11.1.34 ExprSort::SyntaxTree

syntax: *SyntaxIndex*

Figure 11.37: Structure of a syntactic expression

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

11.1.35 ExprSort::FunctionString

locus: *SourceLocation*
type: *TypeIndex*
macro: *TextOffset*

Figure 11.38: Structure of a function string expression

Partition name: `"expr.function-string"`.

11.1.36 ExprSort::CompoundString

locus: *SourceLocation*
type: *TypeIndex*
prefix: *TextOffset*
string: *ExprIndex*

Figure 11.39: Structure of a compound string expression

Partition name: `"expr.compound-string"`.

11.1.37 ExprSort::StringSequence

locus: *SourceLocation*
type: *TypeIndex*
strings: *ExprIndex*

Figure 11.40: Structure of a string sequence expression

Partition name: `"expr.string-sequence"`.

11.1.38 ExprSort::Initializer

locus: *SourceLocation*
type: *TypeIndex*
expr: *ExprIndex*
sort: *InitializerSort*

Figure 11.41: Structure of an initialzer expression

with

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

Figure 11.42: Definition of type *InitializerSort*

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

0x00. Unknown
0x01. Direct
0x02. Copy

InitializerSort::Unknown No initializer expression of this sort shall be produced. This value serves purely as a sentinel purpose.

InitializerSort::Direct An initializer of this sort represents elaboration of a direct-initialization at the C++ input source level.

InitializerSort::Copy An initializer of this sort represents an elaboration of a copy-initialization at the C++ input source level.

Partition name: `"expr.initializer"`.

11.2 ExprSort::Requires

locus: *SourceLocation*
type: *TypeIndex*
parameters: *SyntaxIndex*
body: *SyntaxtIndex*

Figure 11.43: Structure of a `requires` clause

Partition name: `"expr.requires"`.

11.2.1 ExprSort::UnaryFold

locus: *SourceLocation*
type: *TypeIndex*
expr: *ExprIndex*
operation: *DyadicOperator*
associativity: *Associativity*

Figure 11.44: Structure of a unary fold expression

with

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

Figure 11.45: Definition of type *Associativity*

Partition name: `"expr.unary-fold"`.

11.2.2 ExprSort::BinaryFoid

locus: *SourceLocation*
type: *TypeIndex*
left: *ExprIndex*
right: *ExprIndex*
operation: *DyadicOperator*
associativity: *Associativity*

Figure 11.46: Structure of a binary fold expression

Partition name: `"expr.binary-fold"`.

11.2.3 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

locus: *SourceLocation*
type: *TypeIndex*
source: *ExprIndex*
target: *TypeIndex*
inheritance: *ExprIndex*
override: *ExprIndex*
operator: *DyadicOperator*

Figure 11.47: 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"`.

11.2.4 ExprSort::ProductTypeValue

locus: *SourceLocation*
type: *TypeIndex*
class: *DeclIndex*
members: *ExprIndex*
base_subobjects: *ExprIndex*

Figure 11.48: Structure of a product value expression

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

11.2.5 ExprSort::SumTypeValue

locus: *SourceLocation*
type: *TypeIndex*
variant: *DeclIndex*
discriminant: *ActiveMember*
value: *ExprIndex*

Figure 11.49: Structure of a sum type value expression

with

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

Figure 11.50: Definition of type *ActiveMember*

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

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

11.2.6 ExprSort::SubobjectValue

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

11.2.7 ExprSort::ArrayValue

locus: *SourceLocation*
type: *TypeIndex*
elemnts: *ExprIndex*
element_type: *TypeIndex*

Figure 11.51: Structure of an array value expression

Partition name: `"expr.array-value"`.

11.2.8 ExprSort::DynamicDispatch

locus: *SourceLocation*
type: *TypeIndex*
pivot: *ExprIndex*

Figure 11.52: Structure of a dynamic dispatch expression

Partition name: `"expr.dynamic-dispatch"`.

11.2.9 ExprSort::VirtualFunctionConversion

locus: *SourceLocation*
type: *TypeIndex*
function: *DeclIndex*

Figure 11.53: Structure of a virtual function conversion

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

11.2.10 ExprSort::Placeholder

locus: *SourceLocation*
type: *TypeIndex*

Figure 11.54: Structure of a placeholder expression

Partition name: `"expr.placeholder"`.

11.2.11 ExprSort::Expansion

locus: *SourceLocation*
type: *TypeIndex*
operand: *ExprIndex*

Figure 11.55: Structure of an expansion expression

Partition name: `"expr.expansion"`.

11.2.12 ExprSort::Generic

TBD

Figure 11.56: Structure of a generic expression selection

Partition name: `"expr.generic"`.

11.2.13 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.

locus: *SourceLocation*
type: *TypeIndex*
start: *Index*
cardinality: *Cardinality*

Figure 11.57: 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"`.

11.2.14 ExprSort::Nullptr

locus: *SourceLocation*
type: *TypeIndex*

Figure 11.58: Structure of a `nullptr` expression

Partition name: `"expr.nullptr"`.

11.2.15 ExprSort::This

locus: *SourceLocation*
type: *TypeIndex*

Figure 11.59: Structure of a `this` expression

Partition name: `"expr.this"`.

11.2.16 ExprSort::TemplateReference

A reference to a member of a template

locus: *SourceLocation*
type: *TypeIndex*
member_name: *NameIndex*
member_locus: *SourceLocation*
scope: *TypeIndex*
arguments: *ExprIndex*

Figure 11.60: Structure of a template member expression

The *member_name* field designates the nane 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"`.

11.2.17 ExprSort::PushState

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

locus: *SourceLocation*
type: *TypeIndex*
ctor_call: *ExprIndex*
dtor_call: *ExprIndex*
flags: *EHFlags*

Figure 11.61: Structure of a push state expression

The *flags* field is of type

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

Figure 11.62: Definition of type *EHFlags*

.

Partition name: `"expr.push-state"`.

11.2.18 ExprSort::TypeTraitIntrinsic

A use of a type trait intrinsic

locus: *SourceLocation*
type: *TypeIndex*
arguments: *TypeIndex*
intrinsic: *Operator*

Figure 11.63: Structure of an intrinsic type-trait expression

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

11.2.19 ExprSort::DesignatedInitializer

locus: *SourceLocation*
type: *TypeIndex*
member: *TextOffset*
initializer: *ExprIndex*

Figure 11.64: Structure of a designated initializer

Partition name: `"expr.designated-init"`.

11.2.20 ExprSort::PackedTemplateArguments

locus: *SourceLocation*
type: *TypeIndex*
arguments: *ExpexIndex*

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

11.3 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 to the token sequence expression partition. Each entry of that partition is a structure witht the following components: a *tokens* field, and a *locus* field.

locus: *SourceLocation*
type: *TypeIndex*
words: *SentenceIndex*

Figure 11.65: Structure of a token sequence expression

The *words* field is an index into the sentence partition (§17). 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.

11.3.1 ExprSort::AssignInitializer

Partition name: `"expr.assign-initializer"`.

equal: *SourceLocation*
initializer: *ExprIndex*

11.4 Operators

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

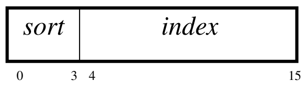

Figure 11.66: *Operator*: Structure of semantic operator

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

11.4.1 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		
0x03.	Nil	0x402.	MsvcLambda

11.4.2 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	0x0C.	Ceil
0x01.	Plus	0x0D.	Floor
0x02.	Negate	0x0E.	Paren
0x03.	Deref	0x0F.	Brace
0x04.	Address	0x10.	Alignas
0x05.	Complement	0x11.	Alignof
0x06.	Not	0x12.	Sizeof
0x07.	PreIncrement	0x13.	Cardinality
0x08.	PreDecrement	0x14.	Typeid
0x09.	PostIncrement	0x15.	Noexcept
0x0A.	PostDecrement	0x16.	Requires
0x0B.	Truncate	0x17.	CoReturn
		0x18.	Await

0x19.	Yield
0x1A.	Throw
0x1B.	New
0x1C.	Delete
0x1D.	DeleteArray
0x1E.	Expand
0x1F.	Read
0x20.	Materialize
0x21.	PseudoDtorCall
0x400.	Msvc
0x401.	MsvcAssume
0x402.	MsvcAlignof
0x403.	MsvcUuidof
0x404.	MsvcIsClass
0x405.	MsvcIsUnion
0x406.	MsvcIsEnum
0x407.	MsvcIsPolymorphic
0x408.	MsvcIsEmpty
0x409.	MsvcIsTriviallyCopyConstructible
0x40A.	MsvcIsTriviallyCopyAssignable
0x40B.	MsvcIsTriviallyDestructible
0x40C.	MsvcHasVirtualDestructor
0x40D.	MsvcIsNothrowCopyConstructible
0x40E.	MsvcIsNothrowCopyAssignable
0x40F.	MsvcIsPod
0x410.	MsvcIsAbstract
0x411.	MsvcIsTrivial
0x412.	MsvcIsTriviallyCopyable
0x413.	MsvcIsStandardLayout
0x414.	MsvcIsLiteralType
0x415.	MsvcIsTriviallyMoveConstructible
0x416.	MsvcHasTrivialMoveAssign
0x417.	MsvcIsTriviallyMoveAssignable
0x418.	MsvcIsNothrowMoveAssignable
0x419.	MsvcUnderlyingType
0x41A.	MsvcIsDestructible
0x41B.	MsvcIsNothrowDestructible
0x41C.	MsvcHasUniqueObjectRepresentations
0x41D.	MsvcIsAggregate
0x41E.	MsvcBuiltinAddressOf
0x41F.	MsvcIsRefClass
0x420.	MsvcIsValueClass
0x421.	MsvcIsSimpleValueClass
0x422.	MsvcIsInterfaceClass
0x423.	MsvcIsDelegate
0x424.	MsvcIsFinal
0x425.	MsvcIsSealed
0x426.	MsvcHasFinalizer
0x427.	MsvcHasCopy
0x428.	MsvcHasAssign
0x429.	MsvcHasUserDestructor
0xFE0.	MsvcConfusion
0xFE1.	MsvcConfusedExpand

11.4.2.1 MonadicOperator::Unknown

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

11.4.2.2 MonadicOperator::Plus

Source-level prefix “+” operator.

11.4.2.3 MonadicOperator::Negate

Source-level prefix “-” operator.

11.4.2.4 MonadicOperator::Deref

Source-level (pointer) dereference "*" operator.

11.4.2.5 MonadicOperator::Address

Source-level address-of "&" operator.

11.4.2.6 MonadicOperator::Complement

Source-level bit complement "~" operator.

11.4.2.7 MonadicOperator::Not

Source-level logical prefix "`!`" (i.e. "`not`") operator.

11.4.2.8 MonadicOperator::PreIncrement

Source-level prefix "++" operator.

11.4.2.9 MonadicOperator::PreDecrement

Source-level prefix "`--`" operator.

11.4.2.10 MonadicOperator::PostIncrement

Source-level postfix "++" operator.

11.4.2.11 MonadicOperator::PostDecrement

Source-level postfix "`--`" operator.

11.4.2.12 MonadicOperator::Truncate

C++ abstract machine operator.

11.4.2.13 MonadicOperator::Ceil

C++ abstract machine operator.

11.4.2.14 MonadicOperator::Floor

C++ abstract machine operator.

11.4.2.15 MonadicOperator::Paren

Source-level parenthesis-enclosing operator.

11.4.2.16 MonadicOperator::Brace

Source-level brace-enclosing operator.

11.4.2.17 MonadicOperator::Alignas

Source-level “`alignas`” operator.

11.4.2.18 MonadicOperator::Alignof

Source-level “`alignof`” operator.

11.4.2.19 MonadicOperator::Sizeof

Source-level “`sizeof`” operator.

11.4.2.20 MonadicOperator::Cardinality

Source-level “`sizeof...`” operator.

11.4.2.21 MonadicOperator::Typeid

Source-level “`typeid`” operator.

11.4.2.22 MonadicOperator::Noexcept

Source-level “`noexpcet`” operator.

11.4.2.23 MonadicOperator::Requires

Source-level “`requires`” operator.

11.4.2.24 MonadicOperator::CoReturn

Source-level “`co_return`” operator.

11.4.2.25 MonadicOperator::Await

Source-level “`co_await`” operator.

11.4.2.26 MonadicOperator::Yield

Source-level “`co_yield`” operator.

11.4.2.27 MonadicOperator::Throw

Source-level “`throw`” operator.

11.4.2.28 MonadicOperator::New

Source-level “`new`” operator.

11.4.2.29 MonadicOperator::Delete

Source-level “`delete`” operator.

11.4.2.30 MonadicOperator::DeleteArray

Source-level “`delete[]`” operator.

11.4.2.31 MonadicOperator::Expand

Source-level pack-expansion operator.

11.4.2.32 MonadicOperator::Read

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

11.4.2.33 MonadicOperator::Materialize

C++ abstract machine class temporary materialization operator.

11.4.2.34 MonadicOperator::PseudoDtorCall

Pseudo-destructor call operator.

11.4.2.35 MonadicOperator::Msvc

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

11.4.2.36 MonadicOperator::MsvcAssume

Source-level “`__assume`” operator (§17.2.1.6).

11.4.2.37 MonadicOperator::MsvcAlignof

Source-level “`__builtin_alignof`” operator (§17.2.1.6).

11.4.2.38 MonadicOperator::MsvcUuidof

Source-level “`__uuidof`” operator (§17.2.1.6).

11.4.2.39 MonadicOperator::MsvcIsClass

Source-level “`__is_class`” operator (§17.2.1.6).

11.4.2.40 MonadicOperator::MsvcIsUnion

Source-level "`__is_union`" operator (§17.2.1.6).

11.4.2.41 MonadicOperator::MsvcIsEnum

Source-level "`__is_enum`" operator (§17.2.1.6).

11.4.2.42 MonadicOperator::MsvcIsPolymorphic

Source-level "`__is_polymorphic`" operator (§17.2.1.6).

11.4.2.43 MonadicOperator::MsvcIsEmpty

Source-level "`__is_empty`" operator (§17.2.1.6).

11.4.2.44 MonadicOperator::MsvcIsTriviallyCopyConstructible

Source-level "`__is_trivially_copy_constructible`" operator (§17.2.1.6).

11.4.2.45 MonadicOperator::MsvcIsTriviallyCopyAssignable

Source-level "`__is_trivially_copy_constructible`" operator (§17.2.1.6).

11.4.2.46 MonadicOperator::MsvcIsTriviallyDestructible

Source-level "`__is_trivially_destructible`" operator (§17.2.1.6).

11.4.2.47 MonadicOperator::MsvcHasVirtualDestructor

Source-level "`__has_virtual_destructor`" operator (§17.2.1.6).

11.4.2.48 MonadicOperator::MsvcIsNothrowCopyConstructible

Source-level "`__is_nothrow_copy_constructible`" operator (§17.2.1.6).

11.4.2.49 MonadicOperator::MsvcIsNothrowCopyAssignable

Source-level "`__is_nothrow_copy_assignable`" operator (§17.2.1.6).

11.4.2.50 MonadicOperator::MsvcIsPod

Source-level "`__is_pod`" operator (§17.2.1.6).

11.4.2.51 MonadicOperator::MsvcIsAbstract

Source-level "`__is_abstract`" operator (§17.2.1.6).

11.4.2.52 MonadicOperator::MsvcIsTrivial

Source-level "`__is_trivial`" operator (§17.2.1.6).

11.4.2.53 MonadicOperator::MsvcIsTriviallyCopyable

Source-level "`__is_trivially_copyable`" operator (§17.2.1.6).

11.4.2.54 MonadicOperator::MsvcIsStandardLayout

Source-level "`__is_standard_layout`" operator (§17.2.1.6).

11.4.2.55 MonadicOperator::MsvcIsLiteralType

Source-level "`__is_literal_type`" operator (§17.2.1.6).

11.4.2.56 MonadicOperator::MsvcIsTriviallyMoveConstructible

Source-level "`__is_trivially_move_constructible`" operator (§17.2.1.6).

11.4.2.57 MonadicOperator::MsvcHasTrivialMoveAssign

Source-level "`__has_trivial_move_assign`" operator (§17.2.1.6).

11.4.2.58 MonadicOperator::MsvcIsTriviallyMoveAssignable

Source-level "`__is_trivially_move_assignable`" operator (§17.2.1.6).

11.4.2.59 MonadicOperator::MsvcIsNothrowMoveAssignable

Source-level "`__is_nothrow_move_assignable`" operator (§17.2.1.6).

11.4.2.60 MonadicOperator::MsvcUnderlyingType

Source-level "`__underlying_type`" operator (§17.2.1.6).

11.4.2.61 MonadicOperator::MsvcIsDestructible

Source-level "`__is_destructible`" operator (§17.2.1.6).

11.4.2.62 MonadicOperator::MsvcIsNothrowDestructible

Source-level "`__is_nothrow_destructible`" operator (§17.2.1.6).

11.4.2.63 MonadicOperator::MsvcHasUniqueObjectRepresentations

Source-level "`__has_unique_object_representations`" operator (§17.2.1.6).

11.4.2.64 MonadicOperator::MsvcIsAggregate

Source-level "`__is_aggregate`" operator (§17.2.1.6).

11.4.2.65 MonadicOperator::MsvcBuiltinAddressOf

Source-level "`__builtin_addressof`" operator (§17.2.1.6).

11.4.2.66 MonadicOperator::MsvcIsRefClass

Source-level "`__is_ref_class`" operator (§17.2.1.6).

11.4.2.67 MonadicOperator::MsvcIsValueClass

Source-level "`__is_value_class`" operator (§17.2.1.6).

11.4.2.68 MonadicOperator::MsvcIsSimpleValueClass

Source-level "`__is_simple_value_class`" operator (§17.2.1.6).

11.4.2.69 MonadicOperator::MsvcIsInterfaceClass

Source-level "`__is_interface_class`" operator (§17.2.1.6).

11.4.2.70 MonadicOperator::MsvcIsDelegate

Source-level "`__is_delegate`" operator (§17.2.1.6).

11.4.2.71 MonadicOperator::MsvcIsFinal

Source-level "`__is_final`" operator (§17.2.1.6).

11.4.2.72 MonadicOperator::MsvcIsSealed

Source-level "`__is_sealed`" operator (§17.2.1.6).

11.4.2.73 MonadicOperator::MsvcHasFinalizer

Source-level "`__has_finalizer`" operator (§17.2.1.6).

11.4.2.74 MonadicOperator::MsvcHasCopy

Source-level "`__has_copy`" operator (§17.2.1.6).

11.4.2.75 MonadicOperator::MsvcHasAssign

Source-level "`__has_assign`" operator (§17.2.1.6).

11.4.2.76 MonadicOperator::MsvcHasUserDestructor

Source-level "`__has_user_destructor`" operator (§17.2.1.6).

11.4.2.77 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.

11.4.2.78 MonadicOperator::MsvcConfusedExpand

Source-level pack expansion "`...`" operator.

11.4.3 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
0x01. Plus
0x02. Minus
0x03. Mult
0x04. Slash
0x05. Modulo
0x06. Remainder
0x07. Bitand
0x08. Bitor
0x09. Bitxor
0x0A. Lshift
0x0B. Rshift
0x0C. Equal
0x0D. NotEqual
0x0E. Less
0x0F. LessEqual
0x10. Greater
0x11. GreaterEqual
0x12. Compare
0x13. LogicAnd
0x14. LogicOr
0x15. Assign
0x16. PlusAssign
0x17. MinusAssign
0x18. MultAssign
0x19. SlashAssign
0x1A. ModuloAssign
0x1B. BitandAssign
0x1C. BitorAssign
0x1D. BitxorAssign
0x1E. LshiftAssign
0x1F. RshiftAssign
0x20. Comma
0x21. Dot
0x22. Arrow
0x23. DotStar
0x24. ArrowStar
0x25. Curry
0x26. Apply
0x27. Index
0x28. DefaultAt
0x29. New
0x2A. NewArray
0x2B. Destruct
0x2C. DestructAt
0x2D. Cleanup

0x2E.	Qualification
0x2F.	Promote
0x30.	Demote
0x31.	Coerce
0x32.	Rewrite
0x33.	Bless
0x34.	Cast
0x35.	ExplicitConversion
0x36.	ReinterpretCast
0x37.	StaticCast
0x38.	ConstCast
0x39.	DynamicCast
0x3A.	Narrow
0x3B.	Widen
0x3C.	Pretend
0x3D.	Closure
0x3E.	ZeroInitialize
0x3F.	ClearStorage
0x400.	Msvc
0x401.	MsvcTryCast
0x402.	MsvcCurry
0x403.	MsvcVirtualCurry
0x404.	MsvcAlign
0x405.	MsvcBitSpan
0x406.	MsvcBitfieldAccess
0x407.	MsvcObscureBitfieldAccess
0x408.	MsvcInitialize
0x409.	MsvcBuiltinOffsetOf
0x40A.	MsvcIsBaseOf
0x40B.	MsvcIsConvertibleTo
0x40C.	MsvcIsTriviallyAssignable
0x40D.	MsvcIsNothrowAssignable
0x40E.	MsvcIsAssignable
0x40F.	MsvcIsAssignableNocheck
0x410.	MsvcBuiltinBitCast
0x411.	MsvcBuiltinIsLayoutCompatible
0x412.	MsvcBuiltinIsPointerInterconvertibleBaseOf
0x413.	MsvcBuiltinIsPointerInterconvertibleWithClass
0x414.	MsvcBuiltinIsCorrespondingMember
0x415.	MsvcIntrinsic

11.4.3.1 DyadicOperator::Unknown

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

11.4.3.2 DyadicOperator::Plus

Source-level infix “+” operator.

11.4.3.3 DyadicOperator::Minus

Source-level infix “-” operator.

11.4.3.4 DyadicOperator::Mult

Source-level infix “*” operator.

11.4.3.5 DyadicOperator::Slash

Source-level infix “/” operator.

11.4.3.6 DyadicOperator::Modulo

Source-level infix “%” operator.

11.4.3.7 DyadicOperator::Remainder

C++ abstract machine.

11.4.3.8 DyadicOperator::Bitand

Source-level infix “&” operator.

11.4.3.9 DyadicOperator::Bitor

Source-level infix “|” operator.

11.4.3.10 DyadicOperator::Bitxor

Source-level infix “^” operator.

11.4.3.11 DyadicOperator::Lshift

Source-level infix “<<” operator.

11.4.3.12 DyadicOperator::Rshift

Source-level infix “>>” operator.

11.4.3.13 DyadicOperator::Equal

Source-level infix “==” operator.

11.4.3.14 DyadicOperator::NotEqual

Source-level infix “!=” operator.

11.4.3.15 DyadicOperator::Less

Source-level infix “<” operator.

11.4.3.16 DyadicOperator::LessEqual

Source-level infix “<=” operator.

11.4.3.17 DyadicOperator::Greater

Source-level infix “>” operator.

11.4.3.18 DyadicOperator::GreaterEqual

Source-level infix “`>=`” operator.

11.4.3.19 DyadicOperator::Compare

Source-level infix “`<=>`” operator.

11.4.3.20 DyadicOperator::LogicAnd

Source-level infix “`&&`” (equivalently “`and`”) operator.

11.4.3.21 DyadicOperator::LogicOr

Source-level infix “`||`” (equivalently “`or`”) operator.

11.4.3.22 DyadicOperator::Assign

Source-level infix “`=`” operator.

11.4.3.23 DyadicOperator::PlusAssign

Source-level infix “`+=`” operator.

11.4.3.24 DyadicOperator::MinusAssign

Source-level infix “`-=`” operator.

11.4.3.25 DyadicOperator::MultAssign

Source-level infix “`*=`” operator.

11.4.3.26 DyadicOperator::SlashAssign

Source-level infix “`/=`” operator.

11.4.3.27 DyadicOperator::ModuloAssign

Source-level infix “`%=`” operator.

11.4.3.28 DyadicOperator::BitandAssign

Source-level infix “`&=`” operator.

11.4.3.29 DyadicOperator::BitorAssign

Source-level infix “`|=`” operator.

11.4.3.30 DyadicOperator::BitxorAssign

Source-level infix “`^=`” operator.

11.4.3.31 DyadicOperator::LshiftAssign

Source-level infix “`<<=`” operator.

11.4.3.32 DyadicOperator::RshiftAssign

Source-level infix “`>>=`” operator.

11.4.3.33 DyadicOperator::Comma

Source-level infix “`,`” operator.

11.4.3.34 DyadicOperator::Dot

C++ abstract machine. Object-based Member selection operator.

11.4.3.35 DyadicOperator::Arrow

Source-level infix “`->`” operator.

Note: Standard C++ considers this operator as monadic when it is overloaded, and applies special rules when used in code, since it is in fact dyadic.

11.4.3.36 DyadicOperator::DotStar

Source-level infix “`.*`” operator.

11.4.3.37 DyadicOperator::ArrowStar

Source-level infix “`->*`” operator.

11.4.3.38 DyadicOperator::Curry

C++ abstract machine. This corresponds to the operation of selecting a non-static member function via the member selection operator, and is in fact supplying an argument for the implicit object parameter of a non-static member function. That is, in a call like `p->f(a, b)`, the expression `p->f` is currying the non-static member function `f` by supplying `p` as argument to the implicit parameter of `f`.

11.4.3.39 DyadicOperator::Apply

Source-level call operation, apply a callable to an argument list.

11.4.3.40 DyadicOperator::Index

Source-level infix “`[]`” operator.

11.4.3.41 DyadicOperator::DefaultAt

C++ abstract machine semantics of default-constructing an object at a given address. Source-level representation of `new(p) T`. The first operand is the address `p`, the second operand is the constructor to call.

11.4.3.42 DyadicOperator::New

C++ abstract machine semantics of allocating storage and constructing an object with a given initializer sequence of expressions. Representation of source-level construct such as `new T(a, b, c)`, where the first operand is the constructor to call, and the second operand is the argument list `(a, b, c)`.

11.4.3.43 DyadicOperator::NewArray

C++ abstract machine semantics of allocating storage and constructing an array of objects. Representation of source-level construct such as `new T[n]`, where the first operand is the constructor to call for each element, and the second operand is the number of elements `n`.

11.4.3.44 DyadicOperator::Destruct

C++ abstract semantics of destructing an object. Representation of source-level construct such as `x.~T()`, where the first operand is `x` – the object to be destructed – and the second operand is the destructor to call.

11.4.3.45 DyadicOperator::DestructAt

C++ abstract semantics of destructing an object. Representation of source-level construct such as `p->~T()`, where the first operand is the address `p` of the object subject to destruction, and the second operand is the destructor to call.

11.4.3.46 DyadicOperator::Cleanup

C++ abstract semantics. Evaluate the first operand, then run the second operand which represents a cleanup (e.g. sequence of destructor calls).

11.4.3.47 DyadicOperator::Qualification

C++ abstract machine. Apply cv-qualifiers (first operand) to an expression (second operand).

11.4.3.48 DyadicOperator::Promote

C++ abstract machine. Apply integral or floating point promotion to an expression (first operand) to arrive at a target type (second operand).

11.4.3.49 DyadicOperator::Demote

C++ abstracr machine. Apply integral or floating point conversion to an expression (first operand) to arrive at a target type (second operand).

11.4.3.50 DyadicOperator::Coerce

C++ abstract machine. Apply coercions that are neither promotions (§11.4.3.48), nor demotions (§11.4.3.49).

11.4.3.51 DyadicOperator::Rewrite

C++ abstract machine. Rewrite a given expression (first operand), into another expression (second operand).

11.4.3.52 DyadicOperator::Bless

11.4.3.53 DyadicOperator::Cast

11.4.3.54 DyadicOperator::ExplicitConversion

11.4.3.55 DyadicOperator::ReinterpretCast

11.4.3.56 DyadicOperator::StaticCast

11.4.3.57 DyadicOperator::ConstCast

11.4.3.58 DyadicOperator::DynamicCast

11.4.3.59 DyadicOperator::Narrow

11.4.3.60 DyadicOperator::Widen

11.4.3.61 DyadicOperator::Pretend

11.4.3.62 DyadicOperator::Closure

11.4.3.63 DyadicOperator::ZeroInitialize

11.4.3.64 DyadicOperator::ClearStorage

11.4.3.65 DyadicOperator::Msvc

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

11.4.3.66 DyadicOperator::MsvcTryCast

11.4.3.67 DyadicOperator::MsvcCurry

11.4.3.68 DyadicOperator::MsvcVirtualCurry

11.4.3.69 DyadicOperator::MsvcAlign

11.4.3.70 DyadicOperator::MsvcBitSpan

11.4.3.71 DyadicOperator::MsvcBitfieldAccess

11.4.3.72 DyadicOperator::MsvcObscureBitfieldAccess

11.4.3.73 DyadicOperator::MsvcInitialize

11.4.3.74 DyadicOperator::MsvcBuiltinOffsetOf

11.4.3.75 DyadicOperator::MsvcIsBaseOf

11.4.3.76 DyadicOperator::MsvcIsConvertibleTo

11.4.3.77 DyadicOperator::MsvcIsTriviallyAssignable

11.4.3.78 DyadicOperator::MsvcIsNothrowAssignable

11.4.3.79 DyadicOperator::MsvcIsAssignable

11.4.3.80 DyadicOperator::MsvcIsAssignableNocheck

11.4.3.81 DyadicOperator::MsvcBuiltinBitCast

11.4.3.82 DyadicOperator::MsvcBuiltinIsLayoutCompatible

11.4.3.83 DyadicOperator::MsvcBuiltinIsPointerInterconvertibleBaseOf

11.4.3.84 DyadicOperator::MsvcBuiltinIsPointerInterconvertibleWithClass

11.4.3.85 DyadicOperator::MsvcBuiltinIsCorrespondingMember

11.4.3.86 DyadicOperator::MsvcIntrinsic

11.4.4 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	0x03. Initialize
0x01. Choice	
0x02. ConstructAt	0x400. Msvc

11.4.4.1 TriadicOperator::Unknown

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

11.4.4.2 TriadicOperator::Choice

Source-level ternary "`?:`" operator.

11.4.4.3 TriadicOperator::ConstructAt

Source-level representation of "`new(p) T(x)`", where

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

11.4.4.4 TriadicOperator::Initialize

C++ abstract machine operation.

11.4.4.5 TriadicOperator::Msvc

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

11.4.5 Storage operators

A sort value OperatorSort::Storage indicates a storage allocation or deallocattion 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

11.4.5.1 StorageOperator::Unknown

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

11.4.5.2 StorageOperator::AllocateSingle

Source-level "`new`" operator.

11.4.5.3 StorageOperator::AllocateArray

Source-level "`new[]`" operator.

11.4.5.4 StorageOperator::DeallocateSingle

Source-level "`delete`" operator.

11.4.5.5 StorageOperator::DeallocateArray

Source-level “`delete[]`” operator.

11.4.5.6 StorageOperator::Msvc

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

11.4.6 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

11.4.6.1 VariadicOperator::Unknown

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

11.4.6.2 VariadicOperator::Collection

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

11.4.6.3 VariadicOperator::Sequence

C++ abstract machine. Like VariadicOperator::Collection (§11.4.6.2) but with a defined left-to-right order of evaluation.

11.4.6.4 VariadicOperator::Msvc

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

11.4.6.5 VariadicOperator::MsvcHasTrivialConstructor

Source-level “`__has_trivial_constructor`” operator (§17.2.1.6).

11.4.6.6 VariadicOperator::MsvcIsConstructible

Source-level “`__is_contructible`” operator (§17.2.1.6).

11.4.6.7 VariadicOperator::MsvcIsNothrowConstructible

Source-level "`__is_nothrow_contructible`" operator (§17.2.1.6).

11.4.6.8 VariadicOperator::MsvcIsTriviallyConstructible

Source-level "`__is_trivially_constructible`" operator (§17.2.1.6).

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

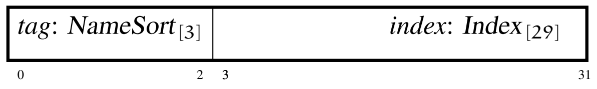

Figure 12.1: *NameIndex*: Abstract reference of names

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 (§2.9.1) denoted by the *tag* field.

12.1 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).

tag = NameSort::Identifier$_{[3]}$
value : *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.

12.2 NameSort::Operator

A *NameIndex* value with tag NameSort::Operator represents an operator function name. The *index* field is an index into the operator partition (§2.9.1). Each entry in that partition has two components: a *category* field denoting the specified operator, and an *encoded* field.

encoded: *TextOffset*
operator: *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 (§11.4) capable of denoting any operator used in a valid C++ program.

Partition name: `"name.operator"`.

12.3 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.

target: *TypeIndex*
encoded: *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"`.

12.4 NameSort::Literal

A *NameIndex* value with tag NameSort::Literal represents a reference to a string literal operator name. The *index* fied is an index into the string literal operator partition (§2.9.1). 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"`.

12.5 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"`.

12.6 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.

primary: *NameIndex*
arguments: *ExprIndex*

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 (§11). 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"`.

12.7 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.

path: *TextOffset*
guard: *TextOffset*

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"`.

12.8 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):

primary_template: *DeclIndex*

Figure 12.9: Structure of a deduction guide name

Partition name: `"name.guide"`.

Chapter 13

Charts

Parameterized declarations are entities that are either templates themselves, or templated in the sense that while not themselves templates they are declared in the lexical scopes of templates (e.g. non-template non-member friend functions lexically defined in a class-definition). The parameter list of such declarations or entities are represented by *chart*s, each designated by a *ChartIndex* abstract reference. Like all abstract references, it is a 32-bit value

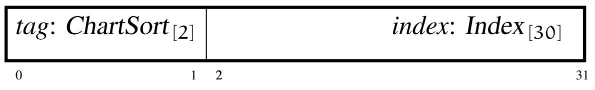

Figure 13.1: `ChartIndex`: Abstract reference of chart

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

0x00. None
0x01. Unilevel
0x02. Multilevel

13.1 ChartSort::None

A *ChartIndex* abstract reference with tag ChartSort::None indicates an empty template parameter list, as in `template<>` at the C++ source level. There is no concrete parameter list stored in the IFC file.

Partition name: `"chart.none"`.

13.2 ChartSort::Unilevel

A *ChartIndex* abstract reference with tag ChartSort::Unilevel designates a sequence non-empty sequence of parameters (§8.2.4).

start: *Index*
cardinality: *Cardinality*
constraint: *ExprIndex*

Figure 13.2: Structure of a unilevel chart

The *start* is an index into the parameter declaration partition (§8.2.4). 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"`.

13.3 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).

start: *Index*
cardinality: *Cardinality*

Figure 13.3: Structure of a multilevel chart

Partition name: `"chart.multilevel"`.

Chapter 14

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

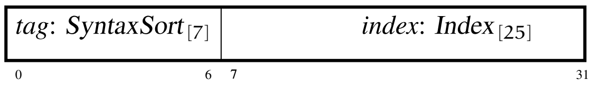

Figure 14.1: `SyntaxIndex`: Abstract reference of syntax fragment

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

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

0x17. FunctionDeclarator
0x18. ArrayOrFunctionDeclarator
0x19. ParameterDeclarator
0x1A. InitDeclarator
0x1B. NewDeclarator
0x1C. SimpleDeclaration
0x1D. ExceptionDeclaration
0x1E. ConditionDeclaration
0x1F. StaticAssertDeclaration
0x20. AliasDeclaration
0x21. ConceptDefinition
0x22. CompoundStatement
0x23. ReturnStatement
0x24. IfStatement
0x25. WhileStatement
0x26. DoWhileStatement
0x27. ForStatement
0x28. InitStatement
0x29. RangeBasedForStatement
0x2A. ForRangeDeclaration
0x2B. LabeledStatement
0x2C. BreakStatement
0x2D. ContinueStatement
0x2E. SwitchStatement
0x2F. GotoStatement
0x30. DeclarationStatement
0x31. ExpressionStatement
0x32. TryBlock
0x33. Handler
0x34. HandlerSeq
0x35. FunctionTryBlock
0x36. TypeIdListElement
0x37. DynamicExceptionSpec
0x38. StatementSeq
0x39. FunctionBody
0x3A. Expression
0x3B. FunctionDefinition
0x3C. MemberFunctionDeclaration
0x3D. TemplateDeclaration
0x3E. RequiresClause
0x3F. SimpleRequirement
0x40. TypeRequirement
0x41. CompoundRequirement
0x42. NestedRequirement
0x43. RequirementBody
0x44. TypeTemplateParameter
0x45. TemplateTemplateParameter
0x46. TypeTemplateArgument
0x47. NonTypeTemplateArgument
0x48. TemplateParameterList
0x49. TemplateArgumentList
0x4A. TemplateId
0x4B. MemInitializer
0x4C. CtorInitializer
0x4D. LambdaIntroducer
0x4E. LambdaDeclarator
0x4F. CaptureDefault
0x50. SimpleCapture
0x51. InitCapture
0x52. ThisCapture
0x53. AttributedStatement
0x54. AttributedDeclaration
0x55. AttributeSpecifierSeq
0x56. AttributeSpecifier
0x57. AttributeUsingPrefix
0x58. Attribute
0x59. AttributeArgumentClause
0x5A. Alignas
0x5B. UsingDeclaration
0x5C. UsingDeclarator
0x5D. UsingDirective
0x5E. ArrayIndex
0x5F. SEHTry
0x60. SEHExcept
0x61. SEHFinally
0x62. SEHLeave

0x63. TypeTraitIntrinsic
0x64. Tuple
0x65. AsmStatement
0x66. NamespaceAliasDefinition
0x67. Super
0x68. UnaryFoldExpression
0x69. BinaryFoldExpression
0x6A. EmptyStatement
0x6B. StructuredBindingDeclaration
0x6C. StructuredBindingIdentifier
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.

14.1 SyntaxSort::VendorExtension

Partition name: `"syntax.vendor-extension"`.

14.2 SyntaxSort::SimpleTypeSpecifier

type: *TypeIndex*
expr: *ExprIndex*
locus: *SourceLocation*

Figure 14.2: Structure of a simple-type-specifier syntax

Partition name: `"syntax.simple-type-specifier"`.

14.3 SyntaxSort::DecltypeSpecifier

expr: *ExprIndex*
decltype_keyword: *SourceLocation*
left_paren: *SourceLocation*
right_paren: *SourceLocation*

Figure 14.3: Structure of the decltype-specifier syntax-tree structure

Partition name: `"syntax.decltype-specifier"`.

14.4 SyntaxSort::PlaceholderTypeSpecifier

Partition name: `"syntax.placeholder-type-specifier"`.

14.5 SyntaxSort::TypeSpecifierSeq

Partition name: `"syntax.type-specifier-seq"`.

14.6 SyntaxSort::DeclSpecifierSeq

Partition name: `"syntax.decl-specifier-seq"`.

14.7 SyntaxSort::VirtualSpecifierSeq

Partition name: `"syntax.virtual-specifier-seq"`.

14.8 SyntaxSort::NoexceptSpecification

Partition name: `"syntax.noexcept-specification"`.

14.9 SyntaxSort::ExplicitSpecifier

Partition name: `"syntax.explicit-specifier"`.

14.10 SyntaxSort::EnumSpecifier

Partition name: `"syntax.enum-specifier"`.

14.11 SyntaxSort::EnumeratorDefinition

Partition name: `"syntax.enumerator-definition"`.

14.12 SyntaxSort::ClassSpecifier

Partition name: `"syntax.class-specifier"`.

14.13 SyntaxSort::MemberSpecification

Partition name: `"syntax.member-specification"`.

14.14 SyntaxSort::MemberDeclaration

Partition name: `"syntax.member-declaration"`.

14.15 SyntaxSort::MemberDeclarator

Partition name: `"syntax.member-declarator"`.

14.16 SyntaxSort::AccessSpecifier

Partition name: `"syntax.access-specifier"`.

14.17 SyntaxSort::BaseSpecifierList

Partition name: `"syntax.base-specifier-list"`.

14.18 SyntaxSort::BaseSpecifier

Partition name: `"syntax.base-specifier"`.

14.19 SyntaxSort::TypeId

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

14.20 SyntaxSort::TrailingReturnType

Partition name: `"syntax.trailing-return-type"`.

14.21 SyntaxSort::Declarator

Partition name: `"syntax.declarator"`.

14.22 SyntaxSort::PointerDeclarator

Partition name: `"syntax.pointer-declarator"`.

14.23 SyntaxSort::ArrayDeclarator

Partition name: `"syntax.array-declarator"`.

14.24 SyntaxSort::FunctionDeclarator

Partition name: `"syntax.function-declarator"`.

14.25 SyntaxSort::ArrayOrFunctionDeclarator

Partition name: `"syntax.array-or-function-declarator"`.

14.26 SyntaxSort::ParameterDeclarator

Partition name: `"syntax.parameter-declarator"`.

14.27 SyntaxSort::InitDeclarator

Partition name: `"syntax.init-declarator"`.

14.28 SyntaxSort::NewDeclarator

Partition name: `"syntax.new-declarator"`.

14.29 SyntaxSort::SimpleDeclaration

Partition name: `"syntax.simple-declaration"`.

14.30 SyntaxSort::ExceptionDeclaration

Partition name: `"syntax.exception-declaration"`.

14.31 SyntaxSort::ConditionDeclaration

Partition name: "`syntax.condition-declaration`".

14.32 SyntaxSort::StaticAssertDeclaration

Partition name: "`syntax.static-assert-declaration`".

14.33 SyntaxSort::AliasDeclaration

Partition name: "`syntax.alias-declaration`".

14.34 SyntaxSort::ConceptDefinition

Partition name: "`syntax.concept-definition`".

14.35 SyntaxSort::CompoundStatement

Partition name: "`syntax.compound-statement`".

14.36 SyntaxSort::ReturnStatement

Partition name: "`syntax.return-statement`".

14.37 SyntaxSort::IfStatement

Partition name: "`syntax.if-statement`".

14.38 SyntaxSort::WhileStatement

Partition name: "`syntax.while-statement`".

14.39 SyntaxSort::DoWhileStatement

Partition name: "`syntax.do-statement`".

14.40 SyntaxSort::ForStatement

Partition name: "`syntax.for-statement`".

14.41 SyntaxSort::InitStatement

Partition name: `"syntax.init-statement"`.

14.42 SyntaxSort::RangeBasedForStatement

Partition name: `"syntax.range-based-for-statement"`.

14.43 SyntaxSort::ForRangeDeclaration

Partition name: `"syntax.for-range-declaration"`.

14.44 SyntaxSort::LabeledStatement

Partition name: `"syntax.labeled-statement"`.

14.45 SyntaxSort::BreakStatement

Partition name: `"syntax.break-statement"`.

14.46 SyntaxSort::ContinueStatement

Partition name: `"syntax.continue-statement"`.

14.47 SyntaxSort::SwitchStatement

Partition name: `"syntax.switch-statement"`.

14.48 SyntaxSort::GotoStatement

Partition name: `"syntax.goto-statement"`.

14.49 SyntaxSort::DeclarationStatement

Partition name: `"syntax.declaration-statement"`.

14.50 SyntaxSort::ExpressionStatement

Partition name: `"syntax.expression-statement"`.

14.51 SyntaxSort::TryBlock

Partition name: `"syntax.try-block"`.

14.52 SyntaxSort::Handler

Partition name: `"syntax.handler"`.

14.53 SyntaxSort::HandlerSeq

Partition name: `"syntax.handler-seq"`.

14.54 SyntaxSort::FunctionTryBlock

Partition name: `"syntax.function-try-block"`.

14.55 SyntaxSort::TypeIdListElement

Partition name: `"syntax.type-id-list-element"`.

14.56 SyntaxSort::DynamicExceptionSpec

Partition name: `"syntax.dynamic-exception-spec"`.

14.57 SyntaxSort::StatementSeq

Partition name: `"syntax.statement-seq"`.

14.58 SyntaxSort::FunctionBody

Partition name: `"syntax.function-body"`.

14.59 SyntaxSort::Expression

Partition name: `"syntax.expression"`.

14.60 SyntaxSort::FunctionDefinition

Partition name: `"syntax.function-definition"`.

14.61 SyntaxSort::MemberFunctionDeclaration

Partition name: `"syntax.member-function-declaration"`.

14.62 SyntaxSort::TemplateDeclaration

Partition name: `"syntax.template-declaration"`.

14.63 SyntaxSort::RequiresClause

Partition name: `"syntax.requires-clause"`.

14.64 SyntaxSort::SimpleRequirement

Partition name: `"syntax.simple-requirement"`.

14.65 SyntaxSort::TypeRequirement

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

14.66 SyntaxSort::CompoundRequirement

Partition name: `"syntax.compound-requirement"`.

14.67 SyntaxSort::NestedRequirement

Partition name: `"syntax.nested-requirement"`.

14.68 SyntaxSort::RequirementBody

Partition name: `"syntax.requirement-body"`.

14.69 SyntaxSort::TypeTemplateParameter

Partition name: `"syntax.type-template-parameter"`.

14.70 SyntaxSort::TemplateTemplateParameter

Partition name: `"syntax.template-template-parameter"`.

14.71 SyntaxSort::TypeTemplateArgument

Partition name: `"syntax.type-template-argument"`.

14.72 SyntaxSort::NonTypeTemplateArgument

Partition name: `"syntax.non-type-template-argument"`.

14.73 SyntaxSort::TemplateParameterList

Partition name: `"syntax.template-parameter-list"`.

14.74 SyntaxSort::TemplateArgumentList

Partition name: `"syntax.template-argument-list"`.

14.75 SyntaxSort::TemplateId

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

14.76 SyntaxSort::MemInitializer

Partition name: `"syntax.mem-initializer"`.

14.77 SyntaxSort::CtorInitializer

Partition name: `"syntax.ctor-initializer"`.

14.78 SyntaxSort::LambdaIntroducer

Partition name: `"syntax.lambda-introducer"`.

14.79 SyntaxSort::LambdaDeclarator

Partition name: `"syntax.lambda-declarator"`.

14.80 SyntaxSort::CaptureDefault

Partition name: `"syntax.capture-default"`.

14.81 SyntaxSort::SimpleCapture

Partition name: `"syntax.simple-capture"`.

14.82 SyntaxSort::InitCapture

Partition name: `"syntax.init-capture"`.

14.83 SyntaxSort::ThisCapture

Partition name: `"syntax.this-capture"`.

14.84 SyntaxSort::AttributedStatement

Partition name: `"syntax.attributed-statement"`.

14.85 SyntaxSort::AttributedDeclaration

Partition name: `"syntax.attributed-declaration"`.

14.86 SyntaxSort::AttributeSpecifierSeq

Partition name: `"syntax.attribute-specifier-seq"`.

14.87 SyntaxSort::AttributeSpecifier

Partition name: `"syntax.attribute-specifier"`.

14.88 SyntaxSort::AttributeUsingPrefix

Partition name: `"syntax.attribute-using-prefix"`.

14.89 SyntaxSort::Attribute

Partition name: `"syntax.attribute"`.

14.90 SyntaxSort::AttributeArgumentClause

Partition name: `"syntax.attribute-argument-clause"`.

14.91 SyntaxSort::Alignas

Partition name: `"syntax.alignas"`.

14.92 SyntaxSort::UsingDeclaration

Partition name: `"syntax.using-declaration"`.

14.93 SyntaxSort::UsingDeclarator

Partition name: `"syntax.using-declarator"`.

14.94 SyntaxSort::UsingDirective

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

14.95 SyntaxSort::ArrayIndex

Partition name: `"syntax.array-index"`.

14.96 SyntaxSort::SEHTry

Partition name: `"syntax.seh-try"`.

14.97 SEHExcept

Partition name: `"syntax.seh-except"`.

14.98 SyntaxSort::SEHFinally

Partition name: `"syntax.seh-finally"`.

14.99 SyntaxSort::SEHLeave

Partition name: `"syntax.seh-leave"`.

14.100 SyntaxSort::TypeTraitIntrinsic

Partition name: `"syntax.type-trait-intrinsic"`.

14.101 SyntaxSort::Tuple

Partition name: `"syntax.tuple"`.

14.102 SyntaxSort::AsmStatement

Partition name: `"syntax.asm-statement"`.

14.103 SyntaxSort::NamespaceAliasDefinition

Partition name: `"syntax.namespace-alias-definition"`.

14.104 SyntaxSort::Super

Partition name: `"syntax.super"`.

14.105 SyntaxSort::UnaryFoldExpression

Partition name: `"syntax.unary-fold-expression"`.

14.106 SyntaxSort::BinaryFoldExpression

Partition name: `"syntax.binary-fold-expression"`.

14.107 SyntaxSort::EmptyStatement

Partition name: `"syntax.empty-statement"`.

14.108 SyntaxSort::StructuredBindingDeclaration

Partition name: `"syntax.structured-binding-declaration"`.

14.109 SyntaxSort::StructuredBindingIdentifier

Partition name: `"syntax.structured-binding-identifier"`.

14.110 SyntaxSort::UsingEnumDeclaration

Partition name: `"syntax.using-enum-declaration"`.

Chapter 15

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 traking macro expansions. For the time being, source location is described in a fairly simplistic way using the unsophisticated structure *SourceLocation* defined as follows:

line: *LineIndex*
column: *Column*

Figure 15.1: Structure of a source location

The *line* field is of type *LineIndex* defined as

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

Figure 15.2: Definition of type *LineIndex*

A value of this type is an index into the partition of file and line (§15.1).
The *column* field is of type *Column* defined as

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

Figure 15.3: 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.

15.1 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 *file*

file: *NameIndex*
line: *LineNumber*

Figure 15.4: Structure of a file-and-line location information

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 15.5: Definition of type *LineNumber*

Partition name: `"src.line"`.

Chapter 16

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 with a declaration, friend declarations, the set of specializations of a template, etc. Each entry in a (parameterized) trait table is a pair structure of the following form

decl: *DeclIndex*
trait: *T*

Figure 16.1: 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.

16.1 Deprecation texts

Partition name: `"trait.deprecated"`.

16.2 Template specializations

Partition name: `"trait.specialization"`.

16.3 Friendship of a class

Partition name: `"trait.friend"`.

16.4 Function Definition

The trait of a definition of a constexpr function or constructor is represented by a structure of defined as follows

parameters: *ChartIndex*
initializers: *ExprIndex*
body: *StmtIndex*

Figure 16.2: 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 (§10) of the function or constructor.

Note: This structure is subject to change.

Partition name: `"trait.mapping-expr"`.

16.5 Function Template Definition

Partition name: `"trait.function-template"`.

16.6 Class Template Definition

Partition name: `"trait.class-template"`.

16.7 Template Alias

Partition name: `"trait.alias-template"`.

16.8 Variable Template Definition

Partition name: `"trait.variable-template"`.

16.9 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* idefined n §8.1.5.

Partition name: `".msvc.trait.vendor-traits"`.

16.9.1 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"`.

16.9.2 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 16.1. The type of the *trait* field is *ChartIndex* (§13.2) listing the sequence of parameter declarations – name, type, and default argument (if any) – in that definition.

Partition name: `".msvc.trait.named-function-parameters"`.

Chapter 17

Token Streams

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* (§17.2) to describe them.

17.1 Sentences

Each MSVC-internal token stream is represented as a *sentence*, a sequence of words (§17.2). Each sentence is referenced by a value of type

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

Figure 17.1: 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:

start: *WordIndex*
cardinality: *Cardinality*
locus: *SourceLocation*

Figure 17.2: Structure of a sentence

The *start* field is an index of type

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

Figure 17.3: Definition of type *WordIndex*

into the word partition (§17.2). 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"`.

17.2 Words

Each entry in the word partition is a 16-byte structure with the following layout

locus: *SourceLocation*
index: *Index*
value: *u16*
sort: *WordSort*
<*padding*>: *u8*

Figure 17.4: 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.

Partition name: `"src.word"`.

17.2.1 Word structures

The type *WordSort* is a set of 8-bit values enumerated as follows

0x00.	Unknown
0x01.	Directive
0x02.	Punctuator
0x03.	Literal
0x04.	Operator
0x05.	Keyword
0x06.	Identifier

The interpertation of the *value* field of a word (§17.4) depends of the value its *sort* field as follows:

- WordSort::Unknown: no particular meaning
- WordSort::Directive: *value* is of type *SourceDirective* (§17.2.1.2)

- WordSort::Punctuator: *value* is of type *SourcePunctuator* (§17.2.1.3)
- WordSort::Literal: *value* is of type *SourceLiteral* (§17.2.1.4)
- WordSort::Operator: *value* is of type *SourceOperator* (§17.2.1.5)
- WordSort::Keyword: *value* is of type *SourceKeyword* (§17.2.1.6)
- WordSort::Identifier: *value* is of type *SourceIdentifier* (§17.2.1.7)

17.2.1.1 WordSort::Unkown

A word (Figure 17.4) with the *sort* field that holds WordSort::Unknown shall not be produced.

17.2.1.2 WordSort::Directive

A word (Figure 17.4) with the *sort* field that holds WorSort::Directive is a *directive word*. The corresponding *value* field is of type

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

Figure 17.5: Definition of type *SourceDirective*

.
Valid values of type *SourceDirective* are enumerated as follows

0x1FFF.	Msvc	0x2010.	MsvcPragmaEndregion
0x2000.	MsvcPragmaPush	0x2011.	MsvcPragmaExecutionCharacterSet
0x2001.	MsvcPragmaPop	0x2012.	MsvcPragmaFenvAccess
0x2002.	MsvcDirectiveStart	0x2013.	MsvcPragmaFileHash
0x2003.	MsvcDirectiveEnd	0x2014.	MsvcPragmaFloatControl
0x2004.	MsvcPragmaAllocText	0x2015.	MsvcPragmaFpContract
0x2005.	MsvcPragmaAutoInline	0x2016.	MsvcPragmaFunction
0x2006.	MsvcPragmaBssSeg	0x2017.	MsvcPragmaBGI
0x2007.	MsvcPragmaCheckStack	0x2018.	MsvcPragmaIdent
0x2008.	MsvcPragmaCodeSeg	0x2019.	MsvcPragmaImplementationKey
0x2009.	MsvcPragmaComment	0x201A.	MsvcPragmaIncludeAlias
0x200A.	MsvcPragmaComponent	0x201B.	MsvcPragmaInitSeg
0x200B.	MsvcPragmaConform	0x201C.	MsvcPragmaInlineDepth
0x200C.	MsvcPragmaConstSeg	0x201D.	MsvcPragmaInlineRecursion
0x200D.	MsvcPragmaDataSeg	0x201E.	MsvcPragmaIntrinsic
0x200E.	MsvcPragmaDeprecated	0x201F.	MsvcPragmaLoop
0x200F.	MsvcPragmaDetectMismatch	0x2020.	MsvcPragmaMakePublic
		0x2021.	MsvcPragmaManaged
		0x2022.	MsvcPragmaMessage

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::MsvcPragmaBGI

SourceDirective::MsvcPragmaIdent

SourceDirective::MsvcPragmaImplementationKey

SourceDirective::MsvcPragmaIncludeAlias

SourceDirective::MsvcPragmaInitSeg

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

17.2.1.3 WordSort::Punctuator

A word (Figure 17.4) with the *sort* field that holds WorSort::Punctuator is a *punctuator word*. The corresponding *value* field is of type

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

Figure 17.6: 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::MsvcNestedTempltateStart 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.

17.2.1.4 WordSort::Literal

A word (Figure 17.4) with the *sort* field that holds WorSort::Literal is a *literal word*. The corresponding *value* field is of type

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

Figure 17.7: 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	0x2001. MsvcStringPrefixMacro
0x2000. MsvcFunctionNameMacro	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 desigates that literal expression (§11.1.3).

SourceLiteral::String A literal word of this value denotes a C++ source level string literal. The *index* index is of type *StringIndex* (Figure 11.19), 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 11.19), 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 – this 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 (§11.1.6) the declarations bound to that identifier.

17.2.1.5 WordSort::Operator

A word (Figure 17.4) with the *sort* field that holds WorSort::Operator is an *operator word*. The corresponding *value* field is of type

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

Figure 17.8: Definition of type *SourceOperator*

Values of type *SourceOperator* are enumerated as follows

0x00.	Unknown
0x01.	Equal
0x02.	Comma
0x03.	Exclaim
0x04.	Plus
0x05.	Dash
0x06.	Star
0x07.	Slash
0x08.	Percent
0x09.	LeftChevron
0x0A.	RightChevron
0x0B.	Tilde
0x0C.	Caret
0x0D.	Bar
0x0E.	Ampersand
0x0F.	PlusPlus
0x10.	DashDash
0x11.	Less
0x12.	LessEqual
0x13.	Greater
0x14.	GreaterEqual
0x15.	EqualEqual
0x16.	ExclaimEqual
0x17.	Diamond
0x18.	PlusEqual
0x19.	DashEqual
0x1A.	StarEqual
0x1B.	SlashEqual
0x1C.	PercentEqual
0x1D.	AmpersandEqual
0x1E.	BarEqual
0x1F.	CaretEqual
0x20.	LeftChevronEqual
0x21.	RightChevronEqual
0x22.	AmpersandAmpersand
0x23.	BarBar
0x24.	Ellipsis
0x25.	Dot
0x26.	Arrow
0x27.	DotStar
0x28.	ArrowStar

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 “->*”.

17.2.1.6 WordSort::Keyword

A word (Figure 17.4) with the *sort* field that holds WorSort::Keyword is a *reserved word*. The corresponding *value* field is of type

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

Figure 17.9: Definition of type *SourceKeyword*

Values of type *SourceKeyword* are classified in two groups.

0x00. Unknown
0x01. Alignas
0x02. Alignof
0x03. Asm
0x04. Auto
0x05. Bool
0x06. Break
0x07. Case
0x08. Catch
0x09. Char
0x0A. Char8T
0x0B. Char16T
0x0C. Char32T
0x0D. Class
0x0E. Concept
0x0F. Const
0x10. Consteval
0x11. Constexpr
0x12. Constinit
0x13. ConstCast
0x14. Continue
0x15. CoAwait
0x16. CoReturn
0x17. CoYield
0x18. Decltype
0x19. Default
0x1A. Delete
0x1B. Do
0x1C. Double
0x1D. DynamicCast
0x1E. Else
0x1F. Enum
0x20. Explicit
0x21. Export
0x22. Extern
0x23. False
0x24. Float
0x25. For
0x26. Friend
0x27. Generic
0x28. Goto
0x29. If
0x2A. Inline
0x2B. Int
0x2C. Long
0x2D. Mutable
0x2E. Namespace
0x2F. New
0x30. Noexcept
0x31. Nullptr
0x32. Operator
0x33. Pragma
0x34. Private
0x35. Protected
0x36. Public
0x37. Register
0x38. ReinterpretCast
0x39. Requires
0x3A. Restrict
0x3B. Return

0x3C.	Short	0x49.	Try
0x3D.	Signed	0x4A.	Typedef
0x3E.	Sizeof	0x4B.	Typeid
0x3F.	Static	0x4C.	Typename
0x40.	StaticAssert	0x4D.	Union
0x41.	StaticCast	0x4E.	Unsigned
0x42.	Struct	0x4F.	Using
0x43.	Switch	0x50.	Virtual
0x44.	Template	0x51.	Void
0x45.	This	0x52.	Volatile
0x46.	ThreadLocal	0x53.	WcharT
0x47.	Throw	0x54.	While
0x48.	True		

and those with values greater than 0x1FFF

0x1FFF.	Msvc	0x2015.	MsvcInt128
0x2000.	MsvcAsm	0x2016.	MsvcInterface
0x2001.	MsvcAssume	0x2017.	MsvcLeave
0x2002.	MsvcAlignof	0x2018.	MsvcMultipleInheritance
0x2003.	MsvcBased	0x2019.	MsvcNullptr
0x2004.	MsvcCdecl	0x201A.	MsvcNovtordisp
0x2005.	MsvcClrcall	0x201B.	MsvcPragma
0x2006.	MsvcDeclspec	0x201C.	MsvcPtr32
0x2007.	MsvcEabi	0x201D.	MsvcPtr64
0x2008.	MsvcEvent	0x201E.	MsvcRestrict
0x2009.	MsvcSehExcept	0x201F.	MsvcSingleInheritance
0x200A.	MsvcFastcall	0x2020.	MsvcSptr
0x200B.	MsvcSehFinally	0x2021.	MsvcStdcall
0x200C.	MsvcForceinline	0x2022.	MsvcSuper
0x200D.	MsvcHook	0x2023.	MsvcThiscall
0x200E.	MsvcIdentifier	0x2024.	MsvcSehTry
0x200F.	MsvcIfExists	0x2025.	MsvcUptr
0x2010.	MsvcIfNotExists	0x2026.	MsvcUuidof
0x2011.	MsvcInt8	0x2027.	MsvcUnaligned
0x2012.	MsvcInt16	0x2028.	MsvcUnhook
0x2013.	MsvcInt32	0x2029.	MsvcVectorcall
0x2014.	MsvcInt64	0x202A.	MsvcVirtualInheritance
		0x202B.	MsvcW64
		0x202C.	MsvcIsClass
		0x202D.	MsvcIsUnion

0x202E.	MsvcIsEnum
0x202F.	MsvcIsPolymorphic
0x2030.	MsvcIsEmpty
0x2031.	MsvcHasTrivialConstructor
0x2032.	MsvcIsTriviallyConstructible
0x2033.	MsvcIsTriviallyCopyConstructible
0x2034.	MsvcIsTriviallyCopyAssignable
0x2035.	MsvcIsTriviallyDestructible
0x2036.	MsvcHasVirtualDestructor
0x2037.	MsvcIsNothrowConstructible
0x2038.	MsvcIsNothrowCopyConstructible
0x2039.	MsvcIsNothrowCopyAssignable
0x203A.	MsvcIsPod
0x203B.	MsvcIsAbstract
0x203C.	MsvcIsBaseOf
0x203D.	MsvcIsConvertibleTo
0x203E.	MsvcIsTrivial
0x203F.	MsvcIsTriviallyCopyable
0x2040.	MsvcIsStandardLayout
0x2041.	MsvcIsLiteralType
0x2042.	MsvcIsTriviallyMoveConsturctible
0x2043.	MsvcHasTrivialMoveAssign
0x2044.	MsvcIsTriviallyMoveAssignable
0x2045.	MsvcIsNothrowMoveAssignable
0x2046.	MsvcIsConstructible
0x2047.	MsvcUnderlyingType
0x2048.	MsvcIsTriviallyAssignable
0x2049.	MsvcIsNothrowAssignable
0x204A.	MsvcIsDestructible
0x204B.	MsvcIsNothrowDestructible
0x204C.	MsvcIsAssignable
0x204D.	MsvcIsAssignableNocheck
0x204E.	MsvcHasUniqueObjectRepresentations
0x204F.	MsvcIsAggregate
0x2050.	MsvcBuiltinAddressOf
0x2051.	MsvcBuiltinOffsetOf
0x2052.	MsvcBuiltinBitCast
0x2053.	MsvcBuiltinIsLayoutCompatible
0x2054.	MsvcBuiltinIsPointerInterconvertibleBaseOf
0x2055.	MsvcBuiltinIsPointerInterconvertibleWithClass
0x2056.	MsvcBuiltinIsCorrespondingMember
0x2057.	MsvcIsRefClass
0x2058.	MsvcIsValueClass
0x2059.	MsvcIsSimpleValueClass
0x205A.	MsvcIsInterfaceClass
0x205B.	MsvcIsDelegate
0x205C.	MsvcIsFinal
0x205D.	MsvcIsSealed
0x205E.	MsvcHasFinalizer
0x205F.	MsvcHasCopy
0x2060.	MsvcHasAssign
0x2061.	MsvcHasUserDestructor
0x2062.	MsvcPackCardinality
0x2063.	MsvcConfusedSizeof
0x2064.	MsvcConfusedAlignas

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 “`consteval`”.

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 thi.s

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 "`__nulptr`".

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::MsvcIsClass A reserved word for the MSVC extension "`__is_class`".

SourceKeyword::MsvcIsUnion A reserved word for the MSVC extension "`__is_union`".

SourceKeyword::MsvcIsEnum A reserved word for the MSVC extension "`__is_enum`".

SourceKeyword::MsvcIsPolymorphic A reserved word for the MSVC extension "`__is_polymorphic`".

SourceKeyword::MsvcIsEmpty A reserved word for the MSVC extension "`__is_empty`".

SourceKeyword::MsvcHasTrivialConstructor A reserved word for the MSVC extension "`__has_trivial_constructor`".

SourceKeyword::MsvcIsTriviallyConstructible A reserved word for the MSVC extension "`__is_trivially_constructible`".

SourceKeyword::MsvcIsTriviallyCopyConstructible A reserved word for the MSVC extension "`__is_trivially_copy_constructible`".

SourceKeyword::MsvcIsTriviallyCopyAssignable A reserved word for the MSVC extension "`__is_trivially_copy_constructible`".

SourceKeyword::MsvcIsTriviallyDestructible A reserved word for the MSVC extension "`__is_trivially_destructible`".

SourceKeyword::MsvcHasVirtualDestructor A reserved word for the MSVC extension "`__has_virtual_destructor`".

SourceKeyword::MsvcIsNothrowConstructible A reserved word for the MSVC extension "`__is_nothrow_constructible`".

SourceKeyword::MsvcIsNothrowCopyConstructible A reserved word for the MSVC extension "`__is_nothrow_copy_constructible`".

SourceKeyword::MsvcIsNothrowCopyAssignable A reserved word for the MSVC extension "`__is_nothrow_copy_assignable`".

SourceKeyword::MsvcIsPod A reserved word for the MSVC extension “`__is_pod`”.

SourceKeyword::MsvcIsAbstract A reserved word for the MSVC extension “`__is_abstract`”.

SourceKeyword::MsvcIsBaseOf A reserved word for the MSVC extension “`__is_base_of`”.

SourceKeyword::MsvcIsConvertibleTo A reserved word for the MSVC extension “`__is_convertible_to`”.

SourceKeyword::MsvcIsTrivial A reserved word for the MSVC extension “`__is_trivial`”.

SourceKeyword::MsvcIsTriviallyCopyable A reserved word for the MSVC extension “`__is_trivially_copyable`”.

SourceKeyword::MsvcIsStandardLayout A reserved word for the MSVC extension “`__is_standard_layout`”.

SourceKeyword::MsvcIsLiteralType A reserved word for the MSVC extension “`__is_literal_type`”.

SourceKeyword::MsvcIsTriviallyMoveConstructible 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::MsvcIsTriviallyMoveAssignable A reserved word for the MSVC extension “`__is_trivially_move_assignable`”.

SourceKeyword::MsvcIsNothrowMoveAssignable A reserved word for the MSVC extension “`__is_nothrow_move_assignable`”.

SourceKeyword::MsvcIsConstructible A reserved word for the MSVC extension “`__is_constructible`”.

SourceKeyword::MsvcUnderlyingType A reserved word for the MSVC extension “`__underlying_type`”.

SourceKeyword::MsvcIsTriviallyAssignable A reserved word for the MSVC extension “`__is_trivially_assignable`”.

SourceKeyword::MsvcIsNothrowAssignable A reserved word for the MSVC extension “`__is_nothrow_assignable`”.

SourceKeyword::MsvcIsDestructible A reserved word for the MSVC extension "`__is_destructible`".

SourceKeyword::MsvcIsNothrowDestructible A reserved word for the MSVC extension "`__is_nothrow_destructible`".

SourceKeyword::MsvcIsAssignable A reserved word for the MSVC extension "`__is_assignable`".

SourceKeyword::MsvcIsAssignableNocheck 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::MsvcIsAggregate 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`".

17.2.1.7 WordSort::Identifier

A word (Figure 17.4) with the *sort* field that holds WorSort::Identifier is an *identifier word*. The corresponding *value* field is of type

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

Figure 17.10: 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	
0x2002. MsvcBuiltinNan	0x2005. MsvcBuiltinNansf

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`.

Chapter 18

Preprocessing

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

tag: *MacroSort*$_{[1]}$	*index*: *Index*$_{[31]}$
0 0	1 31

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

18.1.1 MacroSort::ObjectLike

locus: *SourceLocation*
name: *TextOffset*
body: *FormIndex*

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"`.

18.1.2 MacroSort::FunctionLike

locus: *SourceLocation*
name: *TextOffset*
parameters: *FormIndex*
body: *FormIndex*
arity: *u31*
variadic: *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 *boy* 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-token*s) are grouped into *preprocessing forms*, denoted by abstract references of type *FormIndex* with the following layout:

tag: *FormSort*$_{[4]}$	*index*: *Index*$_{[28]}$
0 … 3	4 … 31

Figure 18.4: *FormIndex*: Abstract reference of preprocessing forms

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

0x00. Identifier

0x01. Number

0x02. Character

0x03. String

0x04. Operator

0x05. Keyword

0x06. Whitespace

0x07. Parameter

0x08. Stringize
0x09. Catenate
0x0A. Pragma
0x0B. Header
0x0C. Parenthesized
0x0D. Tuple
0x0E. Junk

18.2.1 Form structures

18.2.1.1 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

locus: *SourceLocation*
spelling: *TextOffset*

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"`.

18.2.1.2 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

locus: *SourceLocation*
spelling: *TextOffset*

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"`.

18.2.1.3 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

locus: *SourceLocation*
spelling: *TextOffset*

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"`.

18.2.1.4 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

locus: *SourceLocation*
spelling: *TextOffset*

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"`.

18.2.1.5 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

locus: *SourceLocation*
spelling: *TextOffset*
operator: *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 soure-level preprocessing interpretation of that form.

Partition name: `"pp.op"`.

18.2.1.6 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

locus: *SourceLocation*
spelling: *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 existing during the preprocessing phases.

Partition name: `"pp.key"`.

18.2.1.7 FormSort::Whitespace

locus: *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"`.

18.2.1.8 FormSort::Parameter

A *FormIndex* value with tag FormSort::Parameter designates the representation of a macro paramater. The *index* field is an index into the preprocessing parameter form partition. Each structure in that partition has the following layout

locus: *SourceLocation*
spelling: *TextOffset*

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"`.

18.2.1.9 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

locus: *SourceLocation*
operand: *FormIndex*

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"`.

18.2.1.10 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

locus: *SourceLocation*
first: *FormIndex*
second: *FormIndex*

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"`.

18.2.1.11 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

locus: *SourceLocation*
operand: *FormIndex*

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"`.

18.2.1.12 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

locus: *SourceLocation*
spelling: *Textoffset*

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.

18.2.1.13 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

locus: *SourceLocation*
operand: *FormIndex*

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"`.

18.2.1.14 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

start: *Index*
cardinality: *Cardinality*

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"`.

18.2.1.15 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

locus: *SourceLocation*
spelling: *Textoffset*

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

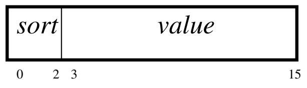

Figure 18.20: *Operator*: Structure of preprocessing form operator

The field *sort* is a 3-bit value of type *WorSort*$_{[3]}$ (§17.2.1), holding either WordSort::Punctuator (§17.2.1.3) or WordSort::Operator (§17.2.1.5). The field *value* is a 13-bit value the interpretation of which is *sort*-dependent: its type is *SourcePunctuator*$_{[13]}$ (Figure 17.6) if *sort* is WordSort::Punctuator; otherwise its type is *SourceOperator*$_{[13]}$ (Figure 17.8) 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

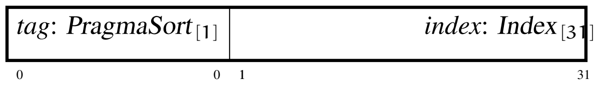

Figure 18.21: *PragmaIndex*: Abstract reference of pragmas

The type *PragmaSort* is a set of 1-bit values enumerated as follows

0x00. VendorExtension

Appendix A

Examples

A.1 Example 1

Consider the following two translation units

```
// TU1.ixx:
  module M1:
  struct S {};
  export using T = S;

  // TU2.ixx:
  module M2:
  import M1;
  export T g() {}
```

A.1.1 IFC file for TU1

Of particular notice:

- the partition of exported modules is empty
- the partition of imported modules is empty
- the global scope exports 'T' with value 'S'
- the class 'S' is marked as non-exported symbols available for 'T'. When a translation unit import 'M1', they can use the name 'T', but not 'S'. Using 'T' to define an object is fine. Same for any context where 'T' is sequired to be a complete type.

A.1.2 IFC file for TU2

- the partition of exported modules is empty

- the partition of imported modules contains one entry: `M1`.

ChangeLog

From 2020-08-21

- Fix type of field *name* of a using-declaration (§8.2.22) as *TextOffset*.
- Fix bit width description in §11.4.
- Detail members of `MonadicOperator` (§11.4.2).
- Detail members of `TriadicOperator` (§11.4.4).
- Detail members of `VariadicOperator` (§11.4.6).

From 2020-08-17

- Fix off-by-one error in bitlengh of *OperatorSort* (Figure 11.66)
- Fix type of field *type* in §11.1.7
- Fix thinko in Figure 8.30
- Remove field *locus* from Figure 18.18

From 2020-08-11

- Add name for deduction guides (§12.8)
- Define structure for deduction guides (§8.2.26)
- Rename partpiton name in §16.4
- Add documentation for preprocessing forms (§18.2)
- Remove field *opcat* from the structure of a ExprSort::Call (§11.1.20)
- Bump version number

From 2020-05-12

- Add new section for semantic operator (§11.4).
- Remove *OperatorCategory* and its description.
- All structures defined in §11 that have a location field have those fields appear first. Similarly if a structure has a *type* field, that field comes after the *locus* field.
- Document `".msvc.trait.named-function-parameters` trait.
- Change `NoexceptSort::Weak` to `NoexceptSort::Inferred`
- Add `FunctionTraits::Immediate`
- Add a new partition name in §7.6
- Remove field *syntax* from the structure of a conversion-function-id (§12.3).
- In the structure of an operator name (§12.2), replace field *category* with *operator* of type *Operator*.
- Define the structure of a syntax tree type (§9.1.22).
- Remove the field *locus* from the structure of a function definition (§16.4)
- Replaces field *default_arguments* with field *chart* in §8.2.16, §8.2.17, §8.2.18, §8.2.19
- Rename field *type* to *denotation* in §11.1.5. The new field *type* designates the sort of type (typically `TypeBasis::Typename`).
- Rename field *kind* to *operator* in §11.1.25. Reorder fields.
- Redefine structures in §11.1.30 and §11.1.31.
- Redefine structure in §11.1.32.
- Rename the field *arg* to *source* in §11.2.3.
- Redefine the structure in §11.2.18.

From 2020-05-05

- Fix DeclSort::Reference, and document *ModuleReference*
- Fix type of field *name* of ExprSort::MemberAccess structure
- Fix type of field *operand* of ExprSort::Alignof structure
- Fix type of field *address* of ExprSort::Delete structure.
- Specify type of field *decltype_specifier* of DestructorCall structure.

- Change field name from *name* ot *macro* in ExprSort::FunctionString structure.
- Change field *name* to *prefix* in ExprSort::CompoundString structure.
- Remove *vtable*, add *locus* fields in ExprSort::ProductTypeValue structure.
- Fix bit width of tag NameSort::Identifier. ‘’

From 2020-04-06

- Add chapter on preprocessing tokens (secrefsec:ifc-pp)
- Remove the field *vtbl* from the structure for ExprSort::ProductTypeValue (§11.2.4)
- The field *prefix* of the structure for sec:ifc:ExprSort:CompoundString (§11.1.35) is now of type *TextOffset*
- The field *macro* of the structure for ExprSort::FunctionString (§11.1.35) is now of type *TextOffset*
- Rename ExprSort::Identifier to ExprSort::UnqualifiedId (§11.1.9). Update partition name.
- Remove the type *Alignment* as no longer used. Fix corresponding field of DeclSort::Variable structure.
- The *name* field of a DeclSort::Scope structure (Figure 8.9) is of type *NameIndex*.
- Really remove all uses of *Offset*
- Add partition name for DeclSort::InheritedConstructor
- Elaborate on types *u8*, *u16*, *u32*.
- Fix off-by-one error in §12.1
- Update partition names in §14.3, §14.110, §11.1.28, §9.1.9

From 2020-01-31

- update format version to 0.10
- Modify most declarations to include *ReachableProperties*
- Redefine the structore of words (§17.2.1)
- Provide stable word sorts and word values
- Define *Index*. Remove *Offset*.

- Update *OperatorCategory* values.
- Prepare for stable decls, types, stmts, exprs sorts.
- Change friend declaration partition name to `"decl.friend"`
- Make sort values hyperlinked to their defining sections

From 2020-01-22

- Update definition of *SyntaxSort*
- Define *SyntaxSort*
- Clarify uses of *SyntaxIndex*
- Document partial specialization, explicit specialization, and explicit instantiation.

From 2019-12-20

- Fix *UnitIndex* tag width - reported by Caleb Sunstrum.
- Document the structure of a sentence, and the type *SentenceIndex*
- Document the structure of a word, and the type *WordIndex*.
- Change mention of *SentenceOffset* to *SentenceIndex*
- Rename mention of *TokenCategory* to *OperatorCategory*.
- Define the type *EHFlags*.
- Define the type *UniqueID*.
- Define the type *ParameterizedEntity*.
- Reinstate the definition of *NoexceptSpecification*
- First field named *body* is actually *constraint*.
- Field *entity* of DeclSort::Friend is actually of type *TypeIndex*
- Remove description of ChartSort::Enclosing.
- Define type *ChartSort*
- Define structure *TypeSort::Deduced*
- Define structure *TypeSort::Forall*
- Remove duplicate an erroneous description of string literal representation.

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`.